What Is Pandas In Python?

Pandas is an open source Python package that is most widely used for data science/data analysis and machine learning tasks. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays. As one of the most popular data wrangling packages, Pandas works well with many other data science modules inside the Python ecosystem, and is typically included in every Python distribution, from those that come with your operating system to commercial vendor distributions like ActiveState's ActivePython.

What Can You Do With DataFrames Using Pandas?

Pandas makes it simple to do many of the time consuming, repetitive tasks associated with working with data, including:

- Data cleansing
- Data fill
- Data normalization
- Merges and joins
- Data visualization
- Statistical analysis
- Data inspection
- Loading and saving data
- And much more In fact, with Pandas, you can do everything that makes world-leading data scientists vote Pandas as the best data analysis and manipulation tool available.

In [ ]:

```
#Import library of pandas and numpy
import pandas as pd
import numpy as np
```

Pandas series creation

In [ ]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Out[ ]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

In [ ]:
```
# Checking the index of pd_series
data.index
```

Out[ ]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [ ]:
```
data[1:3]
```

Out[ ]:
```
1    0.50
2    0.75
dtype: float64
```

In [ ]:
```
# Pandas series indexs can be defiened
data = pd.Series([0.25, 0.5, 0.75, 1.0],index=['a', 'b', 'c', 'd'])
data
```

Out[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [ ]:
```
# series elements can be call by their indexes
data['b']
```

Out[ ]:
```
0.5
```

In [ ]:
```
# Series as specialized dictionary
population_dict = {'California': 3833252, 'Texas': 26448193,'New York': 19651127,'
population_dict
```

Out[ ]:
```
{'California': 3833252,
 'Texas': 26448193,
 'New York': 19651127,
 'Florida': 19552860,
 'Illinois': 12882135}
```

In [ ]:
```
# Constructing Series objects
pd.Series([2, 4, 6])
```

Out[ ]:
```
0    2
1    4
2    6
dtype: int64
```

In [ ]:
```
# Indexs can be defined aas we want
pd.Series(5, index=[100, 200, 300])
```

Out[ ]:
```
100    5
200    5
300    5
dtype: int64
```

In [ ]:
```
# index defaults to the sorted dictionary keys:
```

```
pd.Series({2:'a', 1:'b', 3:'c'})
```

Out[ ]:
```
2    a
1    b
3    c
dtype: object
```

Data Selection in Series

In [ ]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data
```

Out[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [ ]:
```
data.keys()
```

Out[ ]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [ ]:
```
data.items
```

Out[ ]:
```
<bound method Series.items of a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64>
```

In [ ]:
```
data[0]
```

Out[ ]:
```
0.25
```

A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays—that is, slices, masking, and fancy indexing.

In [ ]:
```
## slicing by explicit index
data['a':'c']
```

Out[ ]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [ ]:
```
# slicing by implicit integer index
data[0:2]
```

Out[ ]:
```
a    0.25
b    0.50
dtype: float64
```

In [ ]:
```
# masking
data[(data > 0.3) & (data < 0.8)]
```

Out[ ]:
```
b    0.50
c    0.75
dtype: float64
```

Notice that when you are slicing with an explicit index (i.e., data['a':'d']), the final index is included in the slice, while when you're slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice.

In [ ]:
```
# fancy indexing
data[['a', 'd']]
```

Out[ ]:
```
a    0.25
d    1.00
dtype: float64
```

DataFrame as two-dimensional array

In [ ]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
 'New York': 141297, 'Florida': 170312,
 'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
 'New York': 19651127, 'Florida': 19552860,
 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[ ]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

In [ ]:
```
# Data Transport
data.T
```

Out[ ]:

	California	Texas	New York	Florida	Illinois
area	423967	695662	141297	170312	149995
pop	38332521	26448193	19651127	19552860	12882135

In [ ]:
```
# Selection via loc
data.loc[:'Florida', :'pop']
```

Out[ ]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860

In [ ]:
```
# Selection via iloc
data.iloc[: , :1]
```

Out[ ]:

	area
California	423967
Texas	695662
New York	141297
Florida	170312
Illinois	149995

Handling missing values in data

isnull() :: Generate a Boolean mask indicating missing values

notnull() :: Opposite of isnull()

dropna() :: Return a filtered version of the data

fillna() :: Return a copy of the data with missing values filled or imputed

In [ ]:

```
#Detecting null values
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()
```

Out[ ]:

```
0    False
1     True
2    False
3     True
dtype: bool
```

In [ ]:

```
data[data.notnull()]
```

Out[ ]:

```
0        1
2    hello
dtype: object
```

In [ ]:

```
# Dropping null values
# dropna() (which removes NA values)
# fillna() (which fills in NA values)
data.dropna()
```

Out[ ]:

```
0        1
2    hello
dtype: object
```

In [ ]:

```
# drop Na for Data frame
# For a DataFrame, there are more options. Consider the following DataFrame:
df = pd.DataFrame([[1, np.nan, 2],
[2, 3, 5],
[np.nan, 4, 6]])
```

In [ ]:

```
df
```

Out[ ]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

In [ ]:
```
# We cannot drop single values from a DataFrame; we can only drop full rows or full
# By default, dropna() will drop all rows in which any null value is present:
df.dropna()
```

Out[ ]:

	0	1	2
1	2.0	3.0	5

In [ ]:
```
# Alternatively, you can drop NA values along a different axis; axis=1 drops all co
df.dropna(axis=1)
```

Out[ ]:

	2
0	2
1	5
2	6

In [ ]:
```
# You can also specify how='all', which will only drop rows/columns that are all nu
df.dropna(axis='columns' , how='all')
```

Out[ ]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

In [ ]:
```
# For finer-grained control, the thresh parameter lets you specify a minimum number
df.dropna(axis='rows' , thresh=3)
```

Out[ ]:

	0	1	2
1	2.0	3.0	5

Filling null values

In [ ]:
```
# Sometimes rather than dropping NA values, you'd rather replace them with a valid
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

Out[ ]:
```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

In [ ]:
```
# We can fill NA entries with a single value, such as zero:
data.fillna(0)
```

Out[ ]:
```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

In [ ]:
```
# forward-fill:: We can specify a forward-fill to propagate the previous value forw
data.fillna(method='ffill')
```

Out[ ]:
```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

In [ ]:
```
# back-fill :: Or we can specify a back-fill to propagate the next values backward
data.fillna(method='bfill')
```

Out[ ]:
```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

In [ ]:
```
# Fill- with mean of data
data.fillna(data.mean())
```

Out[ ]:
```
a    1.0
b    2.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

Combining Datasets: Concat and Append

In [ ]:
```
# Simple Concatenation with pd.concat
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Out[ ]:
```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

In [ ]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
 'New York': 141297, 'Florida': 170312,
 'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
 'New York': 19651127, 'Florida': 19552860,
 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[ ]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

In [ ]:

```
# It also works to concatenate higher-dimensional objects, such as DataFrames:
# One important difference between np.concatenate and pd.concat is that Pandas conc
pd.concat([data, data], axis=1)
```

Out[ ]:

	area	pop	area	pop
California	423967	38332521	423967	38332521
Texas	695662	26448193	695662	26448193
New York	141297	19651127	141297	19651127
Florida	170312	19552860	170312	19552860
Illinois	149995	12882135	149995	12882135

In [ ]:

```
# The append() method | the alternative of pd.concat

data.append([data])
```

```
C:\Users\atifg\AppData\Local\Temp\ipykernel_11232\1602891444.py:3: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a future
version. Use pandas.concat instead.
  data.append([data])
```

Out[ ]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

Aggregation and Grouping

In [ ]:

```
# we will use titanic dataset available in sns library
import seaborn as sns
df = sns.load_dataset('titanic')
```

In [ ]:
```
# Categories of Joins
# The pd.merge() function implements a number of types of joins:
#  the one-to-one,
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
 'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})

df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
 'hire_date': [2004, 2008, 2012, 2014]})
print(df1);
print('________________')
print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
________________
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

In [ ]:
```
# merge one dataset into one
df3 = pd.merge(df1, df2)
df3
```

Out[ ]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [ ]:
```
# Many-to-one joins
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
 'supervisor': ['Carly', 'Guido', 'Steve']})
df5 = pd.merge(df3 ,df4)
df5
```

Out[ ]:

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

In [ ]:
```
# Many-to-many
df6 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineer
'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organizatio
df7 = pd.merge(df5 , df6)
df7
```

Out[ ]:

	employee	group	hire_date	supervisor	skills
0	Bob	Accounting	2008	Carly	math
1	Bob	Accounting	2008	Carly	spreadsheets
2	Jake	Engineering	2012	Guido	coding
3	Jake	Engineering	2012	Guido	linux
4	Lisa	Engineering	2004	Guido	coding
5	Lisa	Engineering	2004	Guido	linux
6	Sue	HR	2014	Steve	spreadsheets
7	Sue	HR	2014	Steve	organization

Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

In [ ]:

```
# For this example will use sns dataset of 'titanic'
import seaborn as sns
data = sns.load_dataset('titanic')
data.head()
```

Out[ ]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	d
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	N
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	N
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	N

In [ ]:

```
print('mean of age:', df['age'].mean())
print('median of age:',df['age'].median())
print('minimum age of colum:',df['age'].min())
print('maximum age of colum:',df['age'].max())
```

```
mean of age: 29.69911764705882
median of age: 28.0
minimum age of colum: 0.42
maximum age of colum: 80.0
```

In [ ]:

```
# we can use simply on function to get all the info of data by using 'describe()' j
data.describe()
```

Out[ ]:

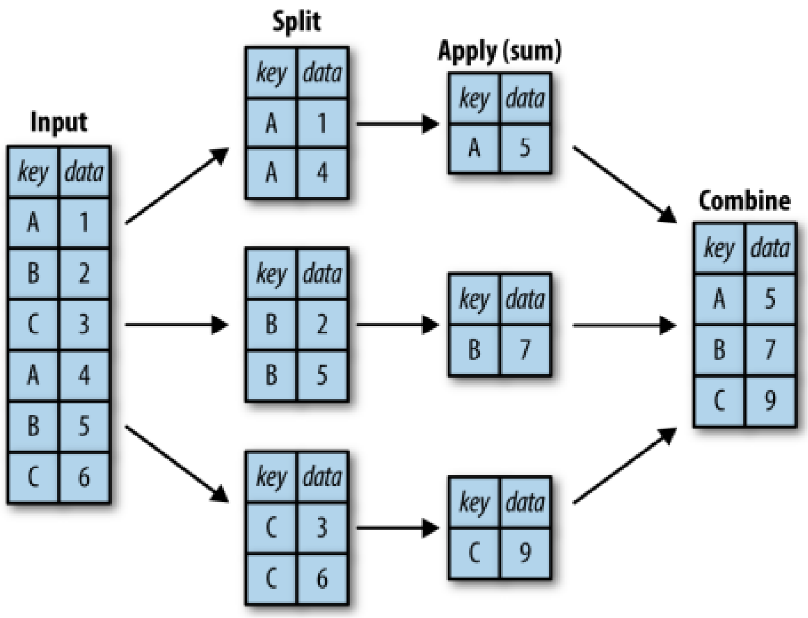
	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the socalled groupby operation.

Split, apply, combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation.



In [ ]:
```
data.head()
```

Out[ ]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	d
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	N
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	N
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	N

In [ ]:

```
data.groupby('age').sum()
```

Out[ ]:

	survived	pclass	sibsp	parch	fare	adult_male	alone
age							
0.42	1	3	0	1	8.5167	0	0
0.67	1	2	1	1	14.5000	0	0
0.75	2	6	4	2	38.5166	0	0
0.83	2	4	1	3	47.7500	0	0
0.92	1	1	1	2	151.5500	0	0
...	...	...	...	...	...	...	...
70.00	0	3	1	1	81.5000	2	1
70.50	0	3	0	0	7.7500	1	1
71.00	0	2	0	0	84.1584	2	2
74.00	0	3	0	0	7.7750	1	1
80.00	1	1	0	0	30.0000	1	1

88 rows × 7 columns

In [ ]:

```
data.groupby(['age' ,'class']).sum()
```

Out[ ]:

age	class	survived	pclass	sibsp	parch	fare	adult_male	alone
0.42	First	0	0	0	0	0.0000	0	0
	Second	0	0	0	0	0.0000	0	0
	Third	1	3	0	1	8.5167	0	0
0.67	First	0	0	0	0	0.0000	0	0
	Second	1	2	1	1	14.5000	0	0
...	...	...	...	...	...	...	...	...
74.00	Second	0	0	0	0	0.0000	0	0
	Third	0	3	0	0	7.7750	1	1
80.00	First	1	1	0	0	30.0000	1	1
	Second	0	0	0	0	0.0000	0	0
	Third	0	0	0	0	0.0000	0	0

264 rows × 7 columns

In [ ]:
```
data.groupby(['age' ,'class']).describe()
```

Out[ ]:

		survived								pclass		...	parch	
		count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max
age	**class**													
0.42	Third	1.0	1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	3.0	...	1.00	1.0
0.67	Second	1.0	1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	2.0	...	1.00	1.0
0.75	Third	2.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0	...	1.00	1.0
0.83	Second	2.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	2.0	2.0	...	1.75	2.0
0.92	First	1.0	1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	...	2.00	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
70.00	Second	1.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	1.0	2.0	...	0.00	0.0
70.50	Third	1.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	1.0	3.0	...	0.00	0.0
71.00	First	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	1.0	...	0.00	0.0
74.00	Third	1.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	1.0	3.0	...	0.00	0.0
80.00	First	1.0	1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	...	0.00	0.0

182 rows × 40 columns

Aggregate, filter, transform

Aggregation.

We're now familiar with GroupBy aggregations with sum(), median(), and the like, but the aggregate() method allows for even more flexibility.

In [ ]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
 'data1': range(6),
 'data2': rng.randint(0, 10, 6)},
 columns = ['key', 'data1', 'data2'])
df
```

Out[ ]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

In [ ]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

Out[ ]:

	data1			data2		
	min	**median**	**max**	**min**	**median**	**max**
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Filtering. A filtering operation allows you to drop data based on the group properties.

In [ ]:
```
#For example, we might want to keep all groups in which the standard deviation is l

def filter_func(x):
 return x['data2'].std() > 4
print(df)
print(df.groupby('key').std())
print(df.groupby('key').filter(filter_func))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
       data1     data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641
  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

Transformation: While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine.

In [ ]:
```
df.groupby('key').transform(lambda x: x - x.mean())
```

Out[ ]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

Pivot Tables

The pivot table takes simple columnwise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data

In [ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
titanic.head()
```

Out[ ]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	d
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	N
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	N
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	N

In [ ]:
```
titanic.pivot_table('survived', index='sex', columns='class')
```

Out[ ]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

In [ ]:
```
# Multilevel pivot tables
#Just as in the GroupBy, the grouping in pivot tables can be specified with multip
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

Out[ ]:

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

Introducing Pandas String Operations

One strength of Python is its relative ease in handling and manipulating string data.

In [ ]:
```
d1 = pd.Series(data = ['peter', 'Paul', None, 'MARY', 'gUIDO'])
d1
```

Out[ ]:
```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

In [ ]:
```
# We can now call a single method that will capitalize all the entries, while skip
d1.str.capitalize()
```

Out[ ]:
```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

Methods similar to Python string methods

Nearly all Python’s built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas str methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

In [ ]:
```
# for string len calcutaion
d1.str.len()
```

Out[ ]:
```
0    5.0
1    4.0
2    NaN
3    4.0
4    5.0
dtype: float64
```

In [ ]:
```
d1.str.startswith('T')
```

Out[ ]:
```
0    False
1    False
2     None
3    False
4    False
dtype: object
```

Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and timespans.

Native Python dates and times: datetime and dateutil Python's basic objects for working with dates and times reside in the built-in date time module. Along with the third-party dateutil module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the datetime type:

In [ ]:
```
from datetime import datetime
print(datetime(year=2015, month=7, day=4))

# Or, using the dateutil module, you can parse dates from a variety of string form
from dateutil import parser
date = parser.parse("4th of July, 2015")
print(date)

# Once you have a datetime object, you can do things like printing the day of the
print(date.strftime('%A'))
```

```
2015-07-04 00:00:00
2015-07-04 00:00:00
Saturday
```

Typed arrays of times: NumPy's datetime64

In [ ]:
```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
```

Out[ ]:
```
array('2015-07-04', dtype='datetime64[D]')
```

In [ ]:
```
np.datetime64('2015-07-04 12:00')
```

Out[ ]:
```
numpy.datetime64('2015-07-04T12:00')
```

Description of date and time codes

Code	Meaning	Time span (relative)	Time span (absolute)
Y	Year	± 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
M	Month	± 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
W	Week	± 1.7e17 years	[1.7e17 BC, 1.7e17 AD]

Code	Meaning	Time span (relative)	Time span (absolute)
D	Day	± 2.5e16 years	[2.5e16 BC, 2.5e16 AD]
h	Hour	± 1.0e15 years	[1.0e15 BC, 1.0e15 AD]
m	Minute	± 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
s	Second	± 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
ms	Millisecond	± 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
us	Microsecond	± 2.9e6 years	[290301 BC, 294241 AD]
ns	Nanosecond	± 292 years	[1678 AD, 2262 AD]
ps	Picosecond	± 106 days	[1969 AD, 1970 AD]
fs	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

Dates and times in Pandas: Best of both worlds

In [ ]:
```
import pandas as pd
date = pd.to_datetime("4th of July, 2015")
date
```

Out[ ]:
```
Timestamp('2015-07-04 00:00:00')
```

In [ ]:
```
# specify data and time by using pd.range function
data = pd.date_range('2012-2-2',  periods=10)
data
```

Out[ ]:
```
DatetimeIndex(['2012-02-02', '2012-02-03', '2012-02-04', '2012-02-05',
               '2012-02-06', '2012-02-07', '2012-02-08', '2012-02-09',
               '2012-02-10', '2012-02-11'],
              dtype='datetime64[ns]', freq='D')
```

Pandas Time Series: Indexing by Time

Where the Pandas time series tools really become useful is when you begin to index data by timestamps.

In [ ]:
```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
 '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

Out[ ]:
```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

Listing of Pandas frequency codes

Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		

This is not the end. Data Manipulation with Pandas comes with endless oppoetunites to analysis of data. Keep paracting and get deep and deeper in pandas to master it. THANK YOU!!