General functions

Data manipulations:

melt(frame[, id_vars, value_vars, var_name, …]

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set

```
df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
                   'B': {0: 1, 1: 3, 2: 5},
                   'C': {0: 2, 1: 4, 2: 6}})
df
```

	A	B	C
0	a	1	2
1	b	3	4
2	c	5	6

```
pd.melt(df, id_vars=['A'], value_vars=['B'])
```

	A	variable	value
0	a	B	1
1	b	B	3
2	c	B	5

pivot(data[, index, columns, values])

Return reshaped DataFrame organized by given index / column values.

```
df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
                           'two'],
                   'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'baz': [1, 2, 3, 4, 5, 6],
                   'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
```

```
df
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo', columns='bar', values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

pivot_table(data[, values, index, columns, …])

Create a spreadsheet-style pivot table as a DataFrame.

```
df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo",
                         "bar", "bar", "bar"],
                   "B": ["one", "one", "one", "two",
                         "one", "two", "two"],
                   "C": ["small", "large", "small",
                         "small", "large", "small",
                         "large"],
                   "D": [1, 2, 2, 3, 3, 4, 7],
                   "E": [2, 4, 5, 6, 6, 9, 9]})
df
```

	A	B	C	D	E
0	foo	one	small	1	2
1	foo	one	large	2	4
2	foo	one	small	2	5
3	foo	two	small	3	6
4	bar	one	large	3	6
5	bar	two	small	4	9
6	bar	two	large	7	9

```
table = pd.pivot_table(df, values='D', index=['A', 'B'],
                       columns=['C'], aggfunc=np.sum)
table
```

A	B	large	small
bar	one	3.0	NaN
	two	7.0	4.0
foo	one	2.0	3.0
	two	NaN	3.0

[crosstab](index, columns[, values, rownames, …])

Compute a simple cross tabulation of two (or more) factors.

```
a = np.array(["foo", "foo", "foo", "foo", "bar", "bar",
              "bar", "bar", "foo", "foo", "foo"], dtype=object)
b = np.array(["one", "one", "one", "two", "one", "one",
              "one", "two", "two", "two", "one"], dtype=object)
c = np.array(["dull", "dull", "shiny", "dull", "dull", "shiny",
              "shiny", "dull", "shiny", "shiny", "shiny"],
             dtype=object)
pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
```

b	one		two	
c	dull	shiny	dull	shiny
a				
bar	1	2	1	0
foo	2	2	1	2

[cut](x, bins[, right, labels, retbins, …])

Bin values into discrete intervals

```
pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
```

```
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], (0.994, 3.0]]
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] < (5.0, 7.0]]
```

qcut(x, q[, labels, retbins, precision, …])

Quantile-based discretization function.

```
: pd.qcut(range(5), 4)
```

```
: [(-0.001, 1.0], (-0.001, 1.0], (1.0, 2.0], (2.0, 3.0], (3.0, 4.0]]
Categories (4, interval[float64]): [(-0.001, 1.0] < (1.0, 2.0] < (2.0, 3.0] < (3.0, 4.0]]
```

merge(left, right[, how, on, left_on, …])

Merge DataFrame or named Series objects with a database-style join.

```
: df1 = pd.DataFrame({'lkey': ['foo', 'bar'],
                      'value': [1, 2]})
  df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz'],
                      'value': [5, 6, 7]})
  df
```

:

	A	B	C	D	E
0	foo	one	small	1	2
1	foo	one	large	2	4
2	foo	one	small	2	5
3	foo	two	small	3	6
4	bar	one	large	3	6
5	bar	two	small	4	9
6	bar	two	large	7	9

```
df2
```

	rkey	value
0	foo	5
1	bar	6
2	baz	7

```
df1.merge(df2, left_on='lkey', right_on='rkey')     # The value columns have the default suffixes,
                                                      #_x and _y, appended.
```

	lkey	value_x	rkey	value_y
0	foo	1	foo	5
1	foo	1	foo	8
2	foo	5	foo	5
3	foo	5	foo	8
4	bar	2	bar	6
5	baz	3	baz	7

```
df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
df1
```

	a	b
0	foo	1
1	bar	2

```
df2
```

	a	c
0	foo	3
1	baz	4

```
df1.merge(df2, how='inner', on='a')
```

	a	b	c
0	foo	1	3

```
df1.merge(df2, how='left', on='a')
```

	a	b	c
0	foo	1	3.0
1	bar	2	NaN

merge_ordered(left, right[, on, left_on, …])

Perform merge with optional filling/interpolation

```
df1 = pd.DataFrame(
    {
        "key": ["a", "c", "e", "a"],
        "lvalue": [1, 2, 3, 1],
        "group": ["a", "a", "a", "b"]
    }
)
df1
```

	key	lvalue	group
0	a	1	a
1	c	2	a
2	e	3	a
3	a	1	b

```
df2 = pd.DataFrame({"key": ["b", "c", "d"], "rvalue": [1, 2, 3]})
df2
```

	key	rvalue
0	b	1
1	c	2
2	d	3

```
: pd.merge_ordered(df1, df2, fill_method="ffill", left_by="group")
```

:

	key	lvalue	group	rvalue
0	a	1	a	NaN
1	b	1	a	1.0
2	c	2	a	2.0
3	d	2	a	3.0
4	e	3	a	3.0
5	a	1	b	NaN
6	b	1	b	1.0
7	c	1	b	2.0
8	d	1	b	3.0

merge_asof(left, right[, on, left_on, …])

Perform an asof merge

```
left = pd.DataFrame({"a": [1, 5, 10], "left_val": ["a", "b", "c"]})
left
```

	a	left_val
0	1	a
1	5	b
2	10	c

```
right = pd.DataFrame({"a": [1, 2, 3, 6], "right_val": [1, 2, 3, 6]})
right
```

	a	right_val
0	1	1
1	2	2
2	3	3
3	6	6

```
pd.merge_asof(left, right, on="a")
```

	a	left_val	right_val
0	1	a	1
1	5	b	3
2	10	c	6

```
]: pd.merge_asof(left, right, on="a", direction="nearest")
```

]:

	a	left_val	right_val
0	1	a	1
1	5	b	6
2	10	c	6

concat(objs[, axis, join, ignore_index, …])

Concatenate pandas objects along a particular axis with optional set logic along the other axes.

```
: s1 = pd.Series(['a', 'b'])
  s2 = pd.Series(['c', 'd'])
  pd.concat([s1, s2])
```

```
: 0    a
  1    b
  0    c
  1    d
  dtype: object
```

get_dummies(data[, prefix, prefix_sep, …])

Convert categorical variable into dummy/indicator variables.

```
: s = pd.Series(list('abca'))
```

```
: pd.get_dummies(s)
```

:

	a	b	c
0	1	0	0
1	0	1	0
2	0	0	1
3	1	0	0

factorize(values[, sort, na_sentinel, size_hint])

Encode the object as an enumerated type or categorical variable.

```
codes, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
codes
```

```
array([0, 0, 1, 2, 0], dtype=int32)
```

unique(values)

Hash table-based unique.

```
pd.unique(pd.Series([2, 1, 3, 3]))
```

```
array([2, 1, 3], dtype=int64)
```

wide_to_long(df, stubnames, i, j[, sep, suffix])

Wide panel to long format.

```
np.random.seed(123)
df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
                   "A1980" : {0 : "d", 1 : "e", 2 : "f"},
                   "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
                   "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
                   "X"     : dict(zip(range(3), np.random.randn(3)))
                  })
df["id"] = df.index
df
```

	A1970	A1980	B1970	B1980	X	id
0	a	d	2.5	3.2	-1.085631	0
1	b	e	1.2	1.3	0.997345	1
2	c	f	0.7	0.1	0.282978	2

```
pd.wide_to_long(df, ["A", "B"], i="id", j="year")
```

		X	A	B
id	year			
0	1970	-1.085631	a	2.5
1	1970	0.997345	b	1.2
2	1970	0.282978	c	0.7
0	1980	-1.085631	d	3.2
1	1980	0.997345	e	1.3
2	1980	0.282978	f	0.1

Top-level missing data

isna(obj)

Detect missing values for an array-like object.

```
array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
array
```

```
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
```

```
pd.isna(array)
```

```
array([[False,  True, False],
       [False, False,  True]])
```

isnull(obj)

Detect missing values for an array-like object.

```
df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
df
```

	0	1	2
0	ant	bee	cat
1	dog	None	fly

```
pd.isnull(df)
```

	0	1	2
0	False	False	False
1	False	True	False

notna(obj)

Detect non-missing values for an array-like object.

```
array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
array
```

```
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
```

```
pd.notna(array)
```

```
array([[ True, False,  True],
       [ True,  True, False]])
```

Top-level conversions

to_numeric(arg[, errors, downcast])

Convert argument to a numeric type.

```
s = pd.Series(['1.0', '2', -3])
pd.to_numeric(s)
```

```
0    1.0
1    2.0
2   -3.0
dtype: float64
```

```
pd.to_numeric(s, downcast='float')
```

```
0    1.0
1    2.0
2   -3.0
dtype: float32
```

```
pd.to_numeric(s, downcast='signed')
```

```
0    1
1    2
2   -3
dtype: int8
```

Top-level dealing with date time like

to_datetime(arg[, errors, dayfirst, ...])

Convert argument to datetime.

```
df = pd.DataFrame({'year': [2015, 2016],
                   'month': [2, 3],
                   'day': [4, 5]})
pd.to_datetime(df)
```

```
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

to_timedelta(arg[, unit, errors])

Convert argument to timedelta.

```
pd.to_timedelta('1 days 06:05:01.00003')
```

```
Timedelta('1 days 06:05:01.000030')
```

```
pd.to_timedelta('15.5us')
```

```
Timedelta('0 days 00:00:00.000015500')
```

date_range([start, end, periods, freq, tz, …])

Return a fixed frequency Datetime Index.

```
pd.date_range(start='1/1/2018', end='1/08/2018')
```

```
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

period_range([start, end, periods, freq, name]

Return a fixed frequency Period index.

```
pd.period_range(start='2017-01-01', end='2018-01-01', freq='M')
```

```
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', '2017-05', '2017-06',
             '2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
             '2018-01'],
            dtype='period[M]', freq='M')
```

timedelta_range([start, end, periods, freq, …])

Return a fixed frequency Timedelta Index, with day as the default frequency.

```
pd.timedelta_range(start='1 day', periods=4)
```

```
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days'], dtype='timedelta64[ns]', freq='D')
```

Top-level dealing with intervals

interval_range([start, end, periods, freq, …])

Return a fixed frequency Interval index.

```
pd.interval_range(start=0, end=5)
```

```
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right',
              dtype='interval[int64]')
```

Top-level evaluation

eval(expr[, parser, engine, truediv, …])

Evaluate a Python expression as a string using various backends.

```
df = pd.DataFrame({"animal": ["dog", "pig"], "age": [10, 20]})
df
```

	animal	age
0	dog	10
1	pig	20

```
pd.eval("double_age = df.age * 2", target=df)
```

	animal	age	double_age
0	dog	10	20
1	pig	20	40

[Note: **"Some of the methods below are not necessary for our course, but it's always good to have knowledge"""".]**