pandas continuation

Queries

query is a method providing SQL-like queries in dataframe

Often it is better than logic indexing - more meaningful, concise and fast

Query syntax

```
df.query(expression)
```

`expression` - string with query. It can refer to columns and index by their names and to variables via `@`

As a result you obtain a proper subset of `df`

```
df.query('column1 > 100')
df.query('column2 < 100 and index != 20')
df.query('column1 == @values')
```

Escaping names

Only valid variable names can be used in query. If you have an irregular column name, escape it with ``` `` ```

```
df.query('`multi word column` == "done"')
```

Series descriptive methods

- `unique` - returns array with unique elements from the series
- `nunique` - returns number of unique elements
- `value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)` - all in one - returns series with each element counts
  - `normalize` - whether to return fraction of element instead of count
  - `sort` - whether to sort by counts
  - `ascending` - whether to start from the lowest
  - `bins` - number of desired intervals on which values will be splitted
  - `dropna` - whether to drop `NA`

```
df =
pd.read_csv('/home/arleg/PycharmProjects/bf_course/1
4.pandas/test_data.tsv', sep='\t', skiprows=9)

df['Allele1 - Forward'].unique()
array(['-', 'T', 'A', 'G', 'C'], dtype=object)

df['Allele1 - Forward'].unique()
5

df['Allele1 - Forward'].value_counts()
-   29
T   19
A   17
G   13
C   12
Name: Allele1 - Forward, dtype: int64
```

Renaming

A method for changing columns or index names in your df

`rename(mapper|axis/columns|index, inplace=False)` - returns renamed `df`

- `mapper` - dict with `old_name: new_name` or function to apply on labels
- `axis` - specify axis to rename (index or columns)
- `columns` - pass mapper here to transform column names
- `index` - pass mapper here to transform index names
- `inplace` - whether to mutate original dataframe, returns `None`

```
df.columns[:7]
```

```
Index(['Sample Name', 'Sample Group', 'Sample
Index', 'SNP Name', 'SNP Index', 'SNP Aux', 'SNP'],
dtype='object')
```

```
df.rename(columns={'Sample Group': 'group', 'Sample
Name': 'sample'}).columns[:7]
```

```
Index(['sample', 'group', 'Sample Index', 'SNP
Name', 'SNP Index', 'SNP Aux', 'SNP'],
dtype='object')
```

```python
df.rename(columns=str.lower).columns[:7]
```

```
Index(['sample name', 'sample group', 'sample
index', 'snp name', 'snp index', 'snp aux', 'snp'],
dtype='object')
```

```python
df.rename(columns={'Sample Group': 'group', 'Sample
Name': 'sample'}, inplace=True)
```

Common patterns

```
# Easier assign different variable names instead of the same df

df = df.rename(...)



# Or you can write so

df.rename(..., inplace=True)
```

Type conversion

`astype` is a useful function for casting

`astype(dtype, copy=True, errors='raise')` - returns df with new types

- `dtype` - new dtype, can be just 1 type for a series, or a mapping from column name to new dtype
- `copy` - better not to change
- `errors` - string to define behaviour on conversion error
  - `raise` - throw an error
  - `ignore` - do nothing with dtype

```
# Subset just to make output more concise
df.loc[:5, 'Allele1 - AB':'Theta'].dtypes
Allele1 - AB        object
Allele2 - AB        object
Allele1 - Plus      object
Allele2 - Plus      object
Chr                 int64
Position            int64
GC Score            float64
Cluster Sep         float64
GT Score            float64
Log R Ratio         float64
Plus/Minus Strand   float64
Theta               float64
dtype: object
```

```
df['GT Score'].head()
0   0.0000
1   0.8076
2   0.8107
3   0.7925
4   0.8670
Name: GT Score, dtype: float64
```

```
df['GT Score'].astype(str)
0      0.0
1   0.8076
2   0.8107
3   0.7925
4   0.867
Name: GT Score, dtype: object
```

```
df = df.astype({'Chr': 'category',
                'Position': float,
                'Theta': np.int})

df.loc[:5, 'Allele1 - AB':'Theta'].dtypes
Allele1 - AB          object
Allele2 - AB          object
Allele1 - Plus        object
Allele2 - Plus        object
Chr                   category
Position              float64
GC Score              float64
Cluster Sep           float64
GT Score              float64
Log R Ratio           float64
Plus/Minus Strand     float64
Theta                 int64
dtype: object
```

```
df.loc[:5, 'Chr':'Theta']
```

```
  Chr  Position   GC Score   Cluster Sep   GT Score   Log R
 Ratio   Plus/Minus Strand   Theta
0    1   10573221.0    0.0000     0.0000 0.0000     -3.8328
    NaN     0
1    1   10673082.0    0.8272     0.8895 0.8076     0.2759
    NaN     0
2    1   10723065.0    0.8316     1.0000 0.8107     0.0657
    NaN     0
3    1   11337555.0    0.3781     1.0000 0.7925     -0.1336
    NaN     0
4    1   11407894.0    0.9038     1.0000 0.8670     0.1763
    NaN     0
```

Subsetting by dtypes

select_dtypes(include=None, exclude=None) - returns part of df with columns with appropriate dtypes

- include - dtype or list with em to keep in df
- exclude - dtype or list with them to remove from df

```
df.select_dtypes('number').dtypes.value_counts()
float64   15
int64     7
dtype: int64

df.select_dtypes(include=['object', int]) \
  .dtypes.value_counts()
object 15
int64     7
dtype: int64

df.select_dtypes(exclude=np.float).dtypes.value_counts()
object 15
int64     7
dtype: int64
```

Subsetting by column names

Really awesome

`filter(items=None, like=None, regex=None, axis=None)` - returns required part of df

- `items` - list with labels to keep (can be substituted with simple indexing [items])
- `like` - keep all labels which satisfy this condition - `like in label`
- `regex` - regular expression to keep all appropriate columns
- `axis` - axis to operate on, columns by default

```python
df.filter(like='Allele1').head()
```

```
 Allele1 - Top Allele1 - Forward Allele1 - Design Allele1 - AB Allele1 - Plus
0         -              -         -         -         -
1         A              T         T         A         -
2         A              T         A         A         -
3         A              T         A         A         -
4         A              A         A         A         -
```

```
df.filter(regex=r'p$').head()
```

```
   Sample Group Allele1 - Top Allele2 - Top  Cluster Sep
0            NaN             -             -      0.0000
1            NaN             A             G      0.8895
2            NaN             A             G      1.0000
3            NaN             A             A      1.0000
4            NaN             A             G      1.0000
```

Grouping

For grouping we can use method `groupby`

`groupby(by=None, axis=0, as_index=True, sort=True)` - returns grouped object (not visible)

- `by` - label, list of them, function or dict which determines how the rows will be grouped
- `axis` - on which axis you should operate
- `as_index` - whether to include grouped columns into index
- `sort` - whether to sort groups

```python
df.groupby('Customer Strand')
```

```
<pandas.core.groupby.groupby.DataFrameGroupBy object
at 0x7f589b360c50>
```

```python
# Show first row in each group
df.groupby('Customer Strand').first()
```

```
               Sample Name  Sample Group  Sample Index      ...     Top Genomic Sequence  CNV Value  CNV Confidence
Customer Strand                                             ...
BOT                    NaN            NaN            1      ...                           NaN     NaN            NaN
TOP                    NaN            NaN            1      ...                           NaN     NaN            NaN
```

Aggregation

`agg` (`aggregate` - obsolete) - function which usually coupled with the grouping

`agg(func, axis=0, *args, **kwargs)` - returns dataframe with 1 row per each group

- `func` - how to aggregate - can be a function, string, list or dict. Function should be aggregation
- axis - axis on which we operate
- `args` and `kwargs` are passed to the func as an arguments

```
df[['Sample Name', 'Sample Group', 'Sample Index',
 'Customer Strand']].groupby('Customer Strand') \
                   .agg('count')
Customer Strand

BOT                    0         0         41
TOP                    0         0         49
```

```
# This case is similar to that
df[['Sample Name', 'Sample Group', 'Sample Index',
'Customer Strand']].groupby('Customer Strand') \
                  .count()
```

```
df[['Sample Index', 'Customer Strand']]
                    .groupby('Customer Strand')
                    .agg(['count', sum, np.mean])
```

```
                    count sum mean
Customer Strand
BOT                    41   41 1
TOP                    49   49 1
```

```
df[['Sample Index', 'Customer Strand', 'Theta']]
          .groupby('Customer Strand')
          .agg({'Theta': ['count', 'mean'],
                'Sample Index': 'sum'})
```

```
                 Theta            Sample Index
                 count     mean          sum
Customer Strand
BOT              41  0.524073        41
TOP              49  0.600367        49
```

Application of functions

We can use an apply function to, well, apply a function for each column or row of dataframe

```
df.apply(func, axis=0, args)
```

- `func` - function to apply, it can be almost any function (aggregation or not aggregation)
- `axis` - axis on which we operate
- `args` - positional arguments to `func`

```
df[['Sample Index', 'Theta']]
                    .apply(lambda x: x + 1).head()
```

```
   Sample Index  Theta
0         2  1.423
1         2  1.521
2         2  1.414
3         2  1.073
4         2  1.436
```

Dummy variables

Binary variables which can be obtained from non-binary

1 variable color with values 'R', 'G', 'B'

Many variables with 0/1 values - color_R, color_G, color_B

```
df['Allele2 - Forward'].head()
0  -
1  C
2  C
3  T
4  G

pd.get_dummies(df['Allele2 - Forward']).head()
   -  A  C  G  T
0  1  0  0  0  0
1  0  0  1  0  0
2  0  0  1  0  0
3  0  0  0  0  1
4  0  0  0  1  0
```

Stacking

Simple almost as for arrays

```
pd.concat(objs, axis=0, ignore_index=False)
```

- objs - tuple with dataframes to concat
- axis - on which axis we operate
- ignore_index - whether to drop previous indices and make new range index (0, 1, …, n - 1)

```
a = pd.DataFrame({'name': ['Sharik', 'Tuzik',
 'Pushok'], 'kindness': ['good', 'nice', 'awesome']})

a
   name kindness
0  Sharik good
1   Tuzik nice
2  Pushok  awesome

b = pd.DataFrame({'name': ['Ugolyok', 'Barsik'],
 'kindness': ['sherstyanoi volchara', 'like a tiger']})
b
   name             kindness
0  Ugolyok  sherstyanoi volchara
1   Barsik       like a tiger
```

```
pd.concat((a, b))
     name              kindness
0   Sharik                  good
1    Tuzik                  nice
2   Pushok               awesome
0  Ugolyok  sherstyanoi volchara
1   Barsik          like a tiger
```

```
pd.concat((a, b), drop_index=True)
     name              kindness
0   Sharik                  good
1    Tuzik                  nice
2   Pushok               awesome
3  Ugolyok  sherstyanoi volchara
4   Barsik          like a tiger
```

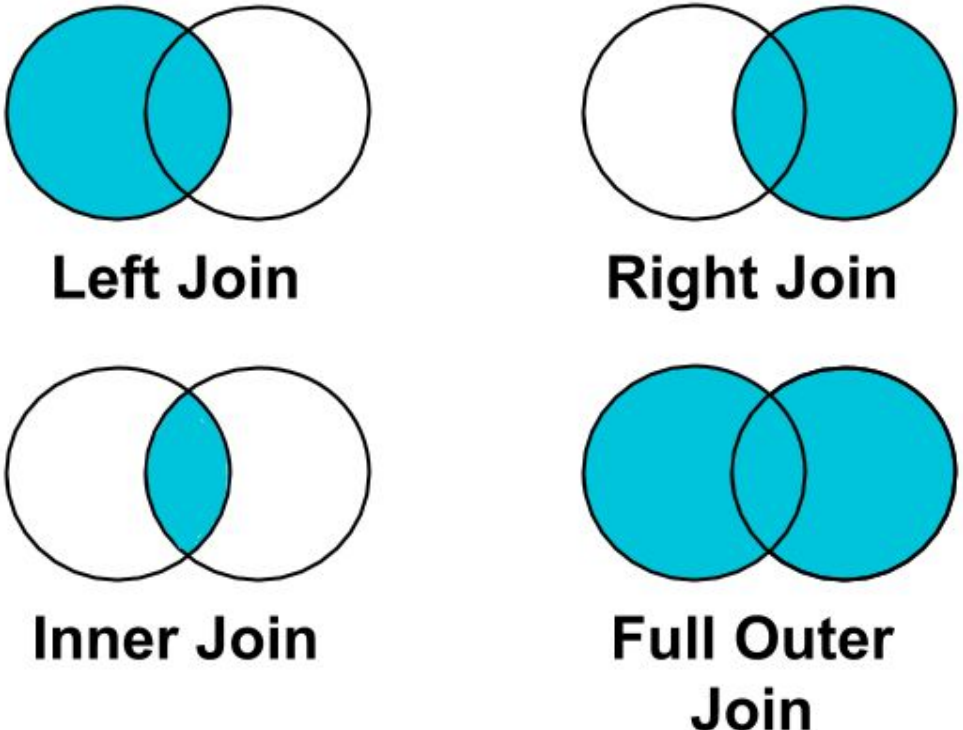
Join

Common operation - unite information from 2 dataframes using same values in their corresponding columns. We have `merge` function for this purpose

```
merge(right, how='inner', on=None, left_on=None,
right_on=None, left_index=False, right_index=False)
```

- `other` - dataframe with which we wanna join original dataframe
- `on` - name of shared by 2 dfs column
- `how` - type of join - left, right, inner, outer

Join types



```
specification = pd.DataFrame({'name': ['Sharik',
 'Tuzik', 'Dili', 'Timosha'], 'species': ['dog',
 'dog', 'crocodile', 'cat']})

specification
    name     species
0   Sharik          dog
1    Tuzik          dog
2     Dili   crocodile
3  Timosha          cat

# Combine 2 dfs by name column, only rows with name
# value present in both will survive
a.merge(specification, on='name')
    name kindness species
0  Sharik     good     dog
1   Tuzik     nice     dog
```

Few index functions before we go to join

Function to make new index in a df

```
set_index(keys, drop=True, append=False, inplace=False)
```

`keys` - name of column which will be an index or list with them

`drop` - whether to drop original index, if `False` will make it a column

`append` - whether to add column to existing index

`inplace` - as usual, whether to change in original df

Function to make get rid of index in a df

```
reset_index(level, drop=False, inplace=False)
```

`level` - name of index which will be excluded, for MultiIndex; by default exclude all index levels

`drop` - whether to drop index, if `False` will make it a column

`append` - whether to add column to existing index

`inplace` - as usual, whether to change in original df

```
a.set_index('name', inplace=True)
a
      kindness
name
Sharik     good
Tuzik      nice
Pushok  awesome

a.loc['Tuzik']
kindness    nice
Name: Tuzik, dtype: object

a.reset_index()
    name kindness
0  Sharik     good
1   Tuzik     nice
2  Pushok  awesome
```

Join

Quite the same as `merge`, but join on indices or on column and index

```
join(other, on=None, how='left')
```

- `other` - dataframe with which we wanna join original dataframe
- on - name of column in left df to use in join with right index; by default join using both indices
- `how` - type of join - left, right, inner, outer

```
specification.join(a, on='name')
     name    species kindness
0   Sharik        dog     good
1    Tuzik        dog     nice
2     Dili  crocodile      NaN
3  Timosha        cat      NaN
```

```
specification.join(a, on='name', how='right')
    name species kindness
0  Sharik     dog     good
1   Tuzik     dog     nice
3  Pushok     NaN  awesome
```