Python PANDAS DataFrame Cheat sheet
V1 date: 08/08/2024

Create / Import / Export

Prepare and record your data in a format easy for analysis:

each row is a dataset (participant/ observation)

Participant	Age	No of siblings	Loves cats
p1	23	2	yes
p2	23	0	yes
p3	25	0	n/a
p4	23	4	yes

each column is a variable

Import pandas and create a pandas dataframe

```
import pandas as pd
pets = pd.DataFrame({
          "animals": ["cat", "dog", "snake", "fish", "fish"],
          "name": ["tilly", "skye", "venom", "blub", "wanda"],
          "age": [8, 7, 2, 1, 20],
          "score": [23.34, 56.3, 20.99, "NaN", 30.05]})
```

animals	name	age	score
cat	tilly	8	23.34
dog	sky	7	56.3
snake	venom	2	20.99
fish	blub	1	NaN
fish	wanda	20	30.05

Import CSV file with data, including header

```
pets = pd.read_csv("animals.csv")
```

Import CSV file without header

```
pets = pd.read_csv("animals.csv", header=None)
```

Export data as CSV file to a specific location

```
path = '~/Python_Projects/2024/animals.csv'
pets.to_csv(path, header=True, index=False, encoding='uf-8')
```

Export data as Excel file

```
pets.to_excel('~/Python_Projects/2024/animals.xlsx',
sheet_name='Sheet1')
```

Data cleaning

Obtain an overview of information in the dataframe such as data types, row numbers and column numbers

```
pets.info()
```

Print the last and the first 5 elements of a series

```
pets.tail()
pets.head()
```

Insert a header into the dataframe

```
pets.columns = ["animals", "name", "age",  "score"]
```

Check data types within a variable, e.g. dataframe for each column

```
pets.dtypes
```

Convert datatype of one column to string

```
pets['animals'] = pets['animals'].astype(str)
```

Convert the datatype of one column at a time to numeric, use error attribute to drop or ignore NaN values

```
pets['score'] = pd.to_numeric(pets['score'], errors='coerce')
```

Combine to_numeric() with apply() to convert multiple columns at once to numeric

Insert new column "date", values as list, series, scalar or NumPy array

```
pets['date'] = ['02.03.23', '12.03.23', '15.03.23',
'20.03.23', '04.04.23']
```

Check the correct presentation of dates, either, convert them by a function yourself (e.g. apply) or try using the module `datetime`

Remove columns from dataframe

```
pets.drop(columns=['date'], inplace=True)
```

Drop rows with any column having NaN/null data

```
pets.dropna()
```

Conditional change values, e.g. score to equal 44 where animal is 'cat'

```
pets.loc[pets.animals == 'cat', 'score'] = 44
```

Replace values with df.replace('orginal', 'new')

```
pets.replace('fish', 'turtle')
```

Fill missing data, NaN, with a value, 0 or mean of the variable

```
pets.fillna(0)
pets['score'] = pets['score'].fillna(pets['score'].median())
```

Replace all NaN elements in column 'age', and 'score', with 0 and 100 respectively.

```
pets.fillna({'age': 0, 'score': 100}, inplace=True)
```

Filter

Access a specific column in the dataframe as series

```
pets['animals']
```

Access a column of dataframe as one-column dataframe

```
pets[['animals']]
```

Filter rows in dataframe pets where score < 30

```
pets.loc[pets.score < 30]
```

Combine filters

```
pets.loc[(pets.score > 30) & (pets.animals == 'fish')]
```

Access selected rows and columns by number df[row, column], access every row of the dataframe, but only columns 0 and 1

```
pets.iloc[:, [0, 1]]
```

Access rows 0 and 2 and columns 1 and 2

```
pets.iloc[[0, 2], [1, 2]]
```

Drop duplicates in a specific column

```
pets.drop_duplicates('animals', keep='last')
```

Combine / split dataframes: merge(), concat()

Merge() combines two or more dataframes with a shared column or index. If the shared column have different names specify with left_on or right_on. There are four types of joins, specify the type by how:

- Inner join: keeps data shared by both tables
- Outer join: keeps all data from both tables
- Left join: keeps all data (all from left & shared) from the left table
- Right join: keeps all data (all from right & shared) from the right table

```
pd.merge(
    left = pets,
    right = visits,
    how = 'inner',
    left_on = 'name',
    right_on ='name')
```

df visits:

name	no. of wins	tournaments
tilly	10	12
sky	12	20
venom	4	4
tabby	5	14

df visits merged with df pets

animals	name	age	score	no. of wins	tournaments
cat	tilly	8	23.34	10	12
dog	sky	7	56.30	12	20
snake	venom	2	20.99	4	4

Concat() combine series or dataframe along an axis based on the index. Join defines the type of combination (union or intersection) and axis whether joined by added columns or rows.

```
pd.concat([pets, visits], join='outer', axis=1)
```

index	animals	name	age	score	name	no. of wins	tournaments
0	cat	tilly	8	23.34	tilly	10	12
1	dog	sky	7	56.3	sky	12	20
2	snake	venom	2	20.99	venom	4	4
3	fish	blub	1	NaN	tabby	5	14
4	fish	wanda	20	30.05	NaN	NaN	NaN

Anti-join to identify differences between two tables

Option 1) Create a boolean table of the shared column or index between the two dataframes with isin(). Use the boolean table as filter to identify differences between the dataframes. E.g. show the rows with names of df pets that are not part of df visits.

```
pets.name.isin(visits.name )
pets.loc[~pets.name.isin(visits.name )]
```

Option 2): Use attribute `'indicator'` of merge(). `'indicator=True'` adds a column with each rows source (both, left only, right only). Apply a filter, e.g. rows only in pets.

```
outer = pd.merge(
    left=pets,
    right=visits,
    how='outer',
    on='name',
    indicator=True)
outer.loc[outer._merge == 'left_only']
```

Splitting dataframes along rows or columns.

Splitting dataframes along rows by row index, the first 3 rows and the rest

```
pets_1 = pets.iloc[:3,:]
pets_2 = pets.iloc[3:,:]
```

Split dataframe in defined number of rows randomly selected, e.g. 50% of the rows

```
pets_p1 = pets.sample(frac=0.5,random_state=1)
pets_p1.reset_index()
```

Split the dataframe by unique values in a row, e.g. group dataframe by animals and store all rows with cats into a new dataframe (see also groupby())

```
pets_grouped = pets.groupby(pets.animals)
pets_grouped.get_group("cat")
```

Groups of data: groupby()

Dataframe grouped by unique values in a column results in a groupby object similar to a dictionary with unique groups as keys

```
p_grouped = pets.groupby(by='animals')
for key, item in p_grouped:
    print(p_grouped.get_group(key), "\n\n")
```

Check the number of groups

```
p_grouped.ngroups
```

Obtain the size of each group df.groupby("column_category").size()

```
pets.groupby("animals").size()
```

Show rows in each group by position, nth() starts with 0. E.g. first row of each group

```
pets.groupby("animals").nth(0)
```

Show one group from the groupby object, can also be used to split the dataframe (see section Combine / split dataframes)

```
p_grouped.get_group('fish')
```

Summary statistics: agg()

Get summary statistics of the data with agg(), the function can be applied to rows and columns. E.g. get: sum(), min(), max(), mean(), median(), count truthy values any(), standard deviation std(), number of non-NA values count(), unique values in each group nunique() ...

Calculate sum of each row (ensure the dtype of the column is numeric)

```
pets.agg('score')
```

Calculate sum and mean for column score and min and max for column age

```
pets.agg({'score': ['sum', 'mean'], 'age': ['min', 'max']})
```

agg() can be applied on the rows as well, e.g. min of each row (code sample)

```
df.agg('min', axis=1)
```

Combine groupby() and agg()

Combine agg() with groupby() to get summary statistics of the grouped data for further analyses.

Extract numeric columns and run summaries on them, e.g. calculate min, max, and median for score and age for each animal group.

```
pets.groupby('animals')[['score', 'age']].agg(['min', 'max',
'sum', 'mean'])
```

Or apply selected summaries to each column, dictionary to sort the output

```
function_dictionary = {'OrderID':'count','Quantity':'mean'}
df.groupby("Product_Category").agg(function_dictionary)
```

Specify column names of the output table, e.g. summarise min and max score and average age for all groups of animals:

```
pets.groupby("animals").agg(
    min_score=("score", "min"),
    max_score=("score", "max"),
    average_age=("age", "mean"))
```

animals	min_score	max_score	average_age
dog	23.34	23.34	8.0
cat	56.30	56.30	7.0
fish	10.00	30.05	10.5
snake	20.99	20.99	2.0

The groupby object itself offers a method to quickly retrieve descriptive standard statistics (count, mean, std, min, max, 25%, 50%, 75%,) for each group within a column, describe(). E.g. check standard statistics of scores for each group of animals.

```
pets.groupby('animals')[['score']].describe()
```

Working with strings: split()

Split strings within table, particularly split into separate columns, e.g. string of address

Apply split() to separate a string of a name stored in one column into two columns, first name and last name: 'first1 last1', 'first2 last2' → 'first1' 'last1' 'first2' 'last2'

```
df[['First Name', 'Last Name']] = df['Name'].str.split(' ',
expand=True)
```

Split an address into separate 3 columns for 'road', 'post code', and 'country': '26 Broomfield Rd, CV5XXX Coventry, UK' → 'street' 'post code', 'city'

```
df[['Street', 'Post code', 'City']] =
df['Address'].str.split(', ', expand=True)
print(df)
```

Run a function on each column element: apply()

If multiple columns interact with each other use apply() instead of agg(). Apply a function to rows or columns of the dataframe, axis=0 apply to each column, axis=1 apply to each row. The apply function takes one variable series as input.

E.g. there is a sport event for 18+ yo with a partner only. Check who can register meeting those two criteria. Define a function to evaluate the data. Apply the function to each row of the dataframe

```
sports = pd.DataFrame({
    'name': pd.Series(['april', 'john', 'lea', 'neo',
'sarah'], dtype="string"),
    'age': [14, 20, 18, 16, 20],
    'partner': [True, False, True, False, True]})
def seriesfunc(series):
    return series.loc['age'] >= 18 and series.loc['partner']
result = sports.apply(seriesfunc, axis=1)
print(result)
```

Use apply to perform multiple calculations and store the results in a new dataframe

```
def function(x):
    d = {}
    d['a_sum'] = x['a'].sum()
    d['a_max'] = x['a'].max()
    d['b_mean'] = x['b'].mean()
    d['c_d_prodsum'] = (x['c'] * x['d']).sum()
    return pd.Series(d, index=['a_sum', 'a_max', 'b_mean',
'c_d_prodsum'])
result = df.groupby('category').apply(function)
```

group	a_sum	a_max	b_mean	c_d_prodsum
0	6	2	0.46	0.6
1	8	4	0.68	0.63

Replace numeric values of wins (1) and losses (0) with strings

```
def text_results(result):
    if result == 1:
        return 'Win'
    else:
      return 'Loss'
result = df.apply(text_results, axis=1)
```

Calculate the sums of each column

```
result = df.apply(sum, axis=0)
```

Pivot tables: pivot() pivot_table()

Pivot tables are particularly useful to organize "stacked" data, meaning there are multiple rows for each subject. Typically there is one row for each subject which is easier for analysis and visualisation. Pivot_table() reshapes the data, transforming rows into columns, and calculates aggregate summary statistics such as sums, counts, averages, etc.

The difference between pivot() and pivot_table(). Both are reshaping tools, but pivot() requires unique index-column combinations and does not provide aggregation capabilities. Pivot_table() allows multi indexes and aggregation functions for summary statistics such as mean or sum.

Pivot_table() has 3 main parameters: index (structure for the new dataframe), columns (unique values for the columns of the new dataframe), and values.

```
df.pivot_table(index='None', columns='None', values='None')
```

The levels in the pivot table will be stored in multi-index objects (hierarchical indexes) on the index and columns of the result dataframe. For multi-indexes, multi columns and values use lists. fill_value replaces missing values with 0. Aggregations are calculated for all columns.

```
df.pivot_table(data, values=['weight', 'price']
        index=['A', 'B'],
        aggfunc=['min', 'median']
        fill_value=0)
```

		min		median	
A	B	weight	price	weight	price
asdf	b1				
	b2				
xyz	b1				
	b2				

Set the attribute margins=True to add a sum of each row and column to the pivot table. And use specific aggregation on each column

```
df.pivot_table(values=['weight', 'price'],
    index=['A'],
    columns='B',
    aggfunc={'weight': 'max', 'price': 'min'})
```