Data Science Workshop-1

ITER, SOA University

Centre for Data Science
SOA University

Contents

- Handling Missing Data, filtering out, Filling in missing data
- Data transformation, Removing duplicates, Transforming using a function, Replacing values, renaming index
- Discretization and binning
- Detecting and filtering outliers
- Permutation and random sampling
- Extension data types
- String Manipulation
- Categorical Data

Data Preaparation

- Data preparation: loading, cleaning, transforming, and rearranging

Handling Missing Data

- The goals of pandas is to make working with missing data as painless as possible
- The isna method gives us a Boolean Series with True where values are null
- The built-in Python None value(key-word) is also treated as NA
- notna Negation of isna, returns True for non-NA values and False for NA values.

isna and notna

```
import pandas as pd
import numpy as np
data=pd.Series([1.2, -3.5, np.nan, 0,None])
data
```

```
0    1.2
1   -3.5
2    NaN
3    0.0
4    NaN
dtype: float64
```

```
data.isna()
```

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

```
data.notna()
```

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

dropna

- To drop nan values, use notna() and boolean indexing

```
data[data.notna()]
```

```
0    1.2
1   -3.5
3    0.0
dtype: float64
```

- dropna(): it returns the Series with only the nonnull data and corresponding index values.

```
data.dropna()
```

```
0    1.2
1   -3.5
3    0.0
dtype: float64
```

dropna(axis)

- If columns which contain missing values are to removed, use axis=1 or axis='columns'
- Default value of axis is 0 or 'index'

```
import pandas as pd
import numpy as np
data=pd.DataFrame([[1.2,-3.5,np.nan,4],
                   [np.nan,0,7,None],
                   [4.7,7.4,-0.7,0.4]])
data
```

Out[4]:

	0	1	2	3
0	1.2	-3.5	NaN	4.0
1	NaN	0.0	7.0	NaN
2	4.7	7.4	-0.7	0.4

```
data.dropna(axis=1)
```

Out[5]:

	1
0	-3.5
1	0.0
2	7.4

dropna(how)

- row or column to be removed from DataFrame, when we have at least one NaN or all NaN

```
data.dropna(how='all')
```

Out[7]:

	0	1	2	3
0	1.2	-3.5	NaN	4.0
1	NaN	0.0	7.0	NaN
2	4.7	7.4	-0.7	0.4

```
data.dropna(how='any')
```

Out[8]:

	0	1	2	3
2	4.7	7.4	-0.7	0.4

dropna(thresh)

- keep only rows containing at least a certain number of non-null values
- thresh=2, means at least two Non-null values should be there in the row.

```
data
```

	0	1	2	3
0	1.2	-3.5	NaN	4.0
1	NaN	0.7	0.0	NaN
2	4.7	7.4	-0.7	0.4

```
data.dropna(thresh=4)
```

	0	1	2	3
2	4.7	7.4	-0.7	0.4

```
data.dropna(thresh=3)
```

	0	1	2	3
0	1.2	-3.5	NaN	4.0
2	4.7	7.4	-0.7	0.4

```
data.dropna(thresh=2)
```

	0	1	2	3
0	1.2	-3.5	NaN	4.0
1	NaN	0.7	0.0	NaN
2	4.7	7.4	-0.7	0.4

Filling Nan values

- Fill NA/NaN values using the specified method.
- Calling fillna with a constant replaces missing values with that value
- We can specify particular values for particular columns, by passing a dict of values to the value keyword.

```
data.fillna(99)
```

Out[29]:

	A	B	C	D
0	1.2	-3.5	99.0	4.0
1	99.0	0.0	7.0	99.0
2	4.7	7.4	-0.7	0.4

```
dic={'A':99,"B":999,'C':9999}
data.fillna(value=dic)
```

Out[31]:

	A	B	C	D
0	1.2	-3.5	9999.0	4.0
1	99.0	0.0	7.0	NaN
2	4.7	7.4	-0.7	0.4

fillna(method,limit)

- Method to use for filling NaN values
- ffill: propagate last valid observation forward to next valid.
- bfill: use next valid observation to fill gap.
- limit: If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill.
- If method is not specified, this is the maximum number of entries along a column where NaNs will be filled.

```
data.fillna(method='ffill')
```

Out[37]:

	A	B	C	D
0	1.2	-3.5	NaN	4.0
1	1.2	0.0	7.0	4.0
2	4.7	7.4	-0.7	0.4

```
data.fillna(0,limit=2)
```

Out[69]:

	A	B	C	D	E
0	1.2	4.7	-3.5	0.0	4.0
1	0.0	0.0	0.0	4.7	7.4
2	0.0	0.0	0.0	7.0	0.0
3	NaN	4.7	7.4	-0.7	0.4

```
data.fillna(method='ffill',limit=2)
```

Out[70]:

	A	B	C	D	E
0	1.2	4.7	-3.5	NaN	4.0
1	1.2	4.7	0.0	4.7	7.4
2	1.2	4.7	0.0	7.0	7.4
3	NaN	4.7	7.4	-0.7	0.4

Data Transformation

- Duplicate rows may be found in a DataFrame
- The method duplicated returns a Boolean Series indicating whether each row is a duplicate (its column values already appeared) or not
- drop_duplicates returns a DataFrame with rows where the duplicated array is False

```
data = pd.DataFrame(
    {"C1": ["a",'b','a','b','a','b'],
     "C2": [1, 1, 1, 2, 2, 2]})
data
```

Out[75]:

	C1	C2
0	a	1
1	b	1
2	a	1
3	b	2
4	a	2
5	b	2

```
data.duplicated()
```

Out[76]:

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

```
data.drop_duplicates()
```

Out[77]:

	C1	C2
0	a	1
1	b	1
3	b	2
4	a	2

drop_duplicates(subset,keep)

- subset: Only consider certain columns for identifying duplicates, by default use all of the columns.
- Keep: Determines which duplicates to keep.
  - 'first' : Drop duplicates except for the first occurrence.
  - 'last' : Drop duplicates except for the last occurrence.

```
data['c3']=['!','@','#','$','#','@']
data.drop_duplicates('c3')
```

	c1	c2	c3
0	a	1	!
1	b	1	@
2	a	1	#
3	b	2	$

```
data.drop_duplicates(
    subset=['c3'],keep='first')
```

	c1	c2	c3
0	a	1	!
1	b	1	@
2	a	1	#
3	b	2	$

```
data.drop_duplicates(
    subset=['c3'],keep='last')
```

	c1	c2	c3
0	a	1	!
3	b	2	$
4	a	2	#
5	b	2	@

- apply works on a row / column of a DataFrame or a series

```
import pandas as pd
import numpy as np
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('abc'), index=['w', 'x', 'y', 'z'])
frame
```

	a	b	c
w	0.187124	-0.978202	-0.915644
x	0.621087	-0.181380	-0.474292
y	-0.222500	-0.050965	-0.386923
z	-0.303024	0.365252	1.381760

```
col = lambda x: x.max()
frame.apply(col)
```

```
a    0.621087
b    0.365252
c    1.381760
dtype: float64
```

apply, map, applymap

- map works element-wise on a Series(A dataframe column can be thought of as a series)
- applymap works element-wise on a DataFrame(The reason for the name applymap is to differentiate from map function for series)

```
ele = lambda x: x**2
frame.applymap(ele)
```

```
frame['a'].map(ele)
```

	a	b	c
w	0.035015	0.956880	0.838405
x	0.385749	0.032899	0.224953
y	0.049506	0.002597	0.149710
z	0.091824	0.133409	1.909261

```
w     0.035015
x     0.385749
y     0.049506
z     0.091824
Name: a, dtype: float64
```

Check Point

- Create the following data frame
- Find the person with maximum height
- Create a BMI column($BMI = Kg/(m^2)$)
- Add a category column, mentioning obessed if BMI>20, else Normal

	Name	Height	Weight
0	Salman	1.68	78
1	Aiswarya	1.63	55
2	Shahid	1.71	72
3	Kareena	1.65	53

Solution to above que

```
frame[frame['Height']==max(frame['Height'])]['Name']
```

```
2    Shahid
```

Solution to above que

```
frame[frame['Height']==max(frame['Height'])]['Name']
```

```
2    Shahid
```

```
ele = lambda x: x**2
x=frame['Height'].map(ele)
y=frame['Weight']
frame['BMI']=y/x
frame
```

Solution to above que

```
frame[frame['Height']==max(frame['Height'])]['Name']
```

```
2    Shahid
```

```
ele = lambda x: x**2
x=frame['Height'].map(ele)
y=frame['Weight']
frame['BMI']=y/x
frame
```

```
OW=lambda x: 'Obessed' if x>20 else 'Normal'
frame['Category']=frame['BMI'].map(OW)
frame
```

	Name	Height	Weight	BMI	Category
0	Salman	1.68	78	27.636054	Obessed
1	Aiswarya	1.63	55	20.700817	Obessed
2	Shahid	1.71	72	24.622961	Obessed
3	Kareena	1.65	53	19.467401	Normal

map,apply,applymap

- map method accepts a dictionary object containing a mapping to do the transformation of values

```
Name_Gen={'Salman':'Male',"Aiswarya":'Female',
         'Shahid':'Male','Kareena':'Female'}
frame["Gender"] = frame["Name"].map(Name_Gen)
frame
```

	Name	Height	Weight	BMI	Category	Gender
0	Salman	1.68	78	27.636054	Obessed	Male
1	Aiswarya	1.63	55	20.700817	Obessed	Female
2	Shahid	1.71	72	24.622961	Obessed	Male
3	Kareena	1.65	53	19.467401	Normal	Female

replace

- df.replace(to_replace,value): Replace values given in to_replace with value.
- to_replace: can be a number or string, or list of values, can also be a dictionary to specify different replacement values for different existing values.
- value: can be a scalar, list matching the length with to re_replace, can also be a dict of values can be used to specify which value to use for each column

replace

	c1	c2	c3
0	a	a	a
1	a	1	!
2	b	1	@
3	a	1	#
4	b	2	$
5	a	2	#
6	b	2	@

```
data.replace(to_replace="a"
             ,value='*')
```

	c1	c2	c3
0	*	*	*
1	*	1	!
2	b	1	@
3	*	1	#
4	b	2	$
5	*	2	#
6	b	2	@

```
data.replace(to_replace
             =["a",'b'],
             value='*')
```

	c1	c2	c3
0	*	*	*
1	*	1	!
2	*	1	@
3	*	1	#
4	*	2	$
5	*	2	#
6	*	2	@

Frame Title

```
data.replace({"a":'*'
              ,'b':'&'})
```

	c1	c2	c3
0	*	*	*
1	*	1	!
2	&	1	@
3	*	1	#
4	&	2	$
5	*	2	#
6	&	2	@

```
data.replace(to_replace
             =["a",'b'],
             value=['*','&'])
```

	c1	c2	c3
0	*	*	*
1	*	1	!
2	&	1	@
3	*	1	#
4	&	2	$
5	*	2	#
6	&	2	@

```
data.replace(to_replace=
             {'c1':{'a':'*',
                    'b':'&'}})
```

	c1	c2	c3
0	*	a	a
1	*	1	!
2	&	1	@
3	*	1	#
4	&	2	$
5	*	2	#
6	&	2	@

Renaming axis labels

- Axis labels can be transformed by a function or mapping to produce new, differently labeled objects.

```python
import pandas as pd
import numpy as np
data=pd.DataFrame(
    {'c1':['a','a','b','a','b','a','b'],
     'c2':['a',1,1,1,2,2,2],
     'c3':['a','!','@','#','$','#','@']},
    index=['d','s','f','g','h','j','k'])
data
```

Out[17]:

	c1	c2	c3
d	a	a	a
s	a	1	!
f	b	1	@
g	a	1	#
h	b	2	$
j	a	2	#
k	b	2	@

```python
def transform(x):
    return x.upper()
data.index.map(transform)
```

Out[19]:

```
Index(['D', 'S', 'F', 'G', 'H', 'J', 'K'], dty
pe='object')
```

In [18]:

```python
data.index = data.index.map(transform)
data
```

Out[18]:

	c1	c2	c3
D	a	a	a
S	a	1	!
F	b	1	@
G	a	1	#
H	b	2	$
J	a	2	#
K	b	2	@

Reanming axis labels

- Use rename to create a transformed version of a dataset without modifying the original
- DataFrame.rename supports two calling conventions
  - (index=index_mapper, columns=columns_mapper)
  - (mapper, axis='index', 'columns')

```
data.rename(index=str.title,
            columns=str.upper)
```

Out[16]:

	C1	C2	C3
D	a	a	a
S	a	1	!
F	b	1	@
G	a	1	#
H	b	2	$
J	a	2	#
K	b	2	@

```
data.rename(str.lower, axis='columns')
```

Out[20]:

	c1	c2	c3
D	a	a	a
S	a	1	!
F	b	1	@
G	a	1	#
H	b	2	$
J	a	2	#
K	b	2	@

```
data.rename(str.lower, axis='index')
```

Out[21]:

	c1	c2	c3
d	a	a	a
s	a	1	!
f	b	1	@
g	a	1	#
h	b	2	$
j	a	2	#
k	b	2	@

Discretization

- pd.cut : when we need to segment and sort data values into bins.
- useful for going from a continuous variable to a categorical variable. For example, cut could convert ages to groups of age ranges
- pandas.cut(x, bins, right, labels, precision)
  - x: array to be binned. Must be 1-dimensional
  - bins: The criteria to bin by. It can be an int or sequence of scalars, or IntervalIndex
    - int : Defines the number of equal-width bins in the range of x
    - sequence of scalars : Defines the bin edges allowing for non-uniform width
    - IntervalIndex : Defines the exact bins to be used
  - right: Indicates whether bins includes the rightmost edge or not. If right == True (the default), then the bins [1, 2, 3, 4] indicate (1,2], (2,3], (3,4] .
  - labels: Specifies the labels for the returned bins.
  - precision: The precision at which to store and display the bins labels. precision=2 option limits the decimal precision to two digits

Discretization

```
x=np.array([1,2,3,7,4])
pd.cut(x,bins=2)
```

```
[(0.994, 4.0], (0.994, 4.0], (0.994, 4.0], (4.0, 7.0], (0.994, 4.0]]
Categories (2, interval[float64, right]): [(0.994, 4.0] < (4.0, 7.0]]
```

```
x=np.array([1,2,3,7,4])
y=[0,2,6]
pd.cut(x,bins=y)
```

```
[(0.0, 2.0], (0.0, 2.0], (2.0, 6.0], NaN, (2.0, 6.0]]
Categories (2, interval[int64, right]): [(0, 2] < (2, 6]]
```

```
bins = pd.IntervalIndex.from_tuples([(0, 2),(2,3),(3,7)])
pd.cut(x, bins)
```

```
[(0, 2], (0, 2], (2, 3], (3, 7], (3, 7]]
Categories (3, interval[int64, right]): [(0, 2] < (2, 3] < (3, 7]]
```

Discretization

```
x=np.array([1,2,3,7,4])
y=[0,2,6]
pd.cut(x,bins=y,right=False)
```

```
[[0.0, 2.0), [2.0, 6.0), [2.0, 6.0), NaN, [2.0, 6.0)]
Categories (2, interval[int64, left]): [[0, 2) < [2, 6)]
```

```
data = np.random.uniform(size=20)
pd.cut(data, 4, precision=2)
```

```
[(0.66, 0.88], (0.66, 0.88], (0.0068, 0.23], (0.45, 0.66], (0.66, 0.88], ...,
(0.45, 0.66], (0.0068, 0.23], (0.45, 0.66], (0.66, 0.88], (0.66, 0.88]]
Length: 20
Categories (4, interval[float64, right]): [(0.0068, 0.23] < (0.23, 0.45] < (0.4
5, 0.66] < (0.66, 0.88]]
```

```
ls=["bad", "medium", "good",'Best']
pd.cut(data, 4, precision=2,labels=ls)
```

```
['Best', 'Best', 'bad', 'good', 'Best', ..., 'good', 'bad', 'good', 'Best', 'Be
st']
Length: 20
Categories (4, object): ['bad' < 'medium' < 'good' < 'Best']
```

Discretizations

- The object pandas returns is a special Categorical object.
- This has two attributes codes, and categories

```
data = np.random.uniform(size=20)
ls=["bad", "medium", "good",'Best']
x=pd.cut(data, 4, precision=2,labels=ls)
x.codes
```

```
array([1, 2, 3, 0, 1, 3, 0, 1, 2, 2, 3, 1, 0, 2, 1, 3, 2, 1, 2, 1],
      dtype=int8)
```

```
x.categories
```

```
Index(['bad', 'medium', 'good', 'Best'], dtype='object')
```

- pd.value_counts(): The bin counts for the result of pandas.cut

```
pd.value_counts(x)
```

```
medium    7
good      6
Best      4
bad       3
dtype: int64
```

pd.qcut()

- Quantile-based discretization function.
- pd.qcut(x,q):
  - x: 1d array or series
  - q: int or list-like of float Number of quantiles. 10 for deciles, 4 for quartiles. Alternatively array of quantiles, e.g. [0, .25, .5, .75, 1.] for quartiles.

```
data = np.random.standard_normal(1000)
quartiles = pd.qcut(data, 4, precision=2)
quartiles
```

```
[(0.56, 3.09], (-2.69, -0.75], (0.56, 3.09], (-0.063, 0.56], (-0.75, -0.063],
..., (0.56, 3.09], (-0.75, -0.063], (-0.063, 0.56], (-0.063, 0.56], (-0.75, -0.
063]]
Length: 1000
Categories (4, interval[float64, right]): [(-2.69, -0.75] < (-0.75, -0.063] <
(-0.063, 0.56] < (0.56, 3.09]]
```

```
pd.value_counts(quartiles)
```

```
(-2.69, -0.75]     250
(-0.75, -0.063]    250
(-0.063, 0.56]     250
(0.56, 3.09]       250
dtype: int64
```

Detecting outliers

- Create a 1000 by 4 matrix, entries from standard normal distribution

Detecting outliers

- Create a 1000 by 4 matrix, entries from standard normal distribution(np.random.standard_normal((1000, 4)))
- find the minimum, maximum, mean, standard deviation, 25percentile, 50percentile and 75percentile.

Detecting outliers

- Create a 1000 by 4 matrix, entries from standard normal distribution(np.random.standard_normal((1000, 4)))
- find the minimum, maximum, mean, standard deviation, 25percentile, 50percentile and 75percentile.(df.describe())
- select all rows having a value exceeding 3 or –3

Detecting outliers

- Create a 1000 by 4 matrix, entries from standard normal distribution(np.random.standard_normal((1000, 4)))
- find the minimum, maximum, mean, standard deviation, 25percentile, 50percentile and 75percentile.(df.describe())
- select all rows having a value exceeding 3 or –3data[(data.abs()>3).any(axis="columns")]
- np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative

Permutation and random sampling

- Permuting (randomly reordering) a numpy array can be done using numpy.random.permutation function.
- one way for permuting dataframes is you permute the row indices or column indices then you pass that to df.iloc or df.take()

```
df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))
df
```

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
sampler = np.random.permutation(5)
sampler
```

Permutation and random sampling

```
df.take(sampler)
```

	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
2	14	15	16	17	18	19	20

```
df.iloc[sampler]
```

	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
2	14	15	16	17	18	19	20

Permutation and random sampling

- By invoking take with axis="columns", we could also select a permutation of the columns
- To select a random subset without replacement (the same row cannot appear twice), you can use the sample method
- a sample with replacement (to allow repeat choices), pass replace=True to sample

```
column_sampler = np.random.permutation(7)
column_sampler
df.take(column_sampler, axis="columns")
```

	0	5	3	1	4	6	2
0	0	5	3	1	4	6	2
1	7	12	10	8	11	13	9
2	14	19	17	15	18	20	16
3	21	26	24	22	25	27	23
4	28	33	31	29	32	34	30

```
df.sample(n=3)
```

	0	1	2	3	4	5	6
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13

```
df.sample(n=3,replace=True)
```

	0	1	2	3	4	5	6
4	28	29	30	31	32	33	34
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

Computing Indicator/Dummy Variables

- converting a categorical variable into a dummy or indicator matrix
- If a column in a DataFrame has k distinct values, you would derive a matrix with k columns containing all 1s and 0s
- pandas has a pandas.get_dummies function for doing this
- pandas.get_dummies(data, prefix,dtype)
  - data: Data of which to get dummy indicators
  - prefix: String to append DataFrame column names
  - dtype: Data type for new columns

Dummy Variable

```python
import pandas as pd
import numpy as np
df=pd.DataFrame({'C1':['A','B','C','A','C'],
                 'C2':range(5)})
df
```

	C1	C2
0	A	0
1	B	1
2	C	2
3	A	3
4	C	4

```python
pd.get_dummies(df['C1'])
```

	A	B	C
0	1	0	0
1	0	1	0
2	0	0	1
3	1	0	0
4	0	0	1

```python
pd.get_dummies(df["C1"], dtype=float)
```

	A	B	C
0	1.0	0.0	0.0
1	0.0	1.0	0.0
2	0.0	0.0	1.0
3	1.0	0.0	0.0
4	0.0	0.0	1.0

```python
pd.get_dummies(df["C1"], dtype=float,prefix='C1')
```

	C1_A	C1_B	C1_C
0	1.0	0.0	0.0
1	0.0	1.0	0.0

```python
dummy=pd.get_dummies(df["C1"], dtype=float,prefix='C1')
df_with_dummy = df[["C2"]].join(dummy)
df_with_dummy
```

	C2	C1_A	C1_B	C1_C
0	0	1.0	0.0	0.0
1	1	0.0	1.0	0.0

Check point

- Collect 10 samples from the uniform distribution in the interval [0,1]
- create 4 bins [0,0.25),[0.25,0.5),[0.5,0.75),[0.75,1), and keep all elements into the respective beans
- create 4 dummy columns, with names as above intervals. If the first element belongs to a particularly interval, in the respective column it should be True, and False otherwise

```
values = np.random.uniform(size=10)
bins = [0, 0.25, 0.5,0.75, 1]
pd.get_dummies(pd.cut(values, bins))
```

- Change the code, to get the exact output

Extension types

- pandas was originally built upon the capabilities present in NumPy
- Many pandas concepts, such as missing data, were implemented using what was available in NumPy
- Building on NumPy led to a number of shortcomings
  - when missing data was introduced into such data, pandas converted the data type to float64 and used np.nan to represent null values
  - Datasets with a lot of string data were computationally expensive
  - Some data types, like time intervals, timedeltas, and timestamps with time zones, could not be supported efficiently without using computationally expensive arrays of Python objects.
- More recently, pandas has developed an extension type system allowing for new data types to be added even if they are not supported natively by NumPy.

Extension types

```
s = pd.Series([1, 2, 3, None])
s
```

```
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

```
s.dtype
```

```
dtype('float64')
```

```
s= pd.Series([1, 2, 3, None], dtype=pd.Int64Dtype())
s
```

```
0       1
1       2
2       3
3    <NA>
dtype: Int64
```

```
s.dtype
```

```
Int64Dtype()
```

Extension types

Pandas extension types provide a way to extend the functionality of pandas by creating custom data types. This can offer several advantages:

- Efficient Storage: Extension types can be more memory-efficient, allowing you to represent data in a more compact form.
- Improved Code Readability: Creating custom extension types can lead to more readable and self-explanatory code
- Domain-Specific Data Handling: Extension types enable you to create data types tailored to specific domains
- Compatibility with Pandas Ecosystem: Extension types can be seamlessly integrated into the broader pandas ecosystem, ensuring compatibility with various libraries and tools that work with pandas data structures.

String Manipulation

- Built-In String Object Methods(Split, strip,join, count, find, replace etc, see Strings PPT)
- Regular Expressions(Regular expressions provide a flexible way to search or match string patterns in text see next few slides)
- String Functions in pandas(For a string operation on pandas objects)
    - String and regular expression methods can be applied (passing a lambda or other function) to each value using data.map, but it will fail on the NA (null) values.
    - To cope with this, pandas Series objects has has str attribute, which skip over NA values.

regular expression

- RegEx can be used to check if a string contains the specified search pattern.
- The re module offers a set of functions that allows us to search a string for a match:
  - search Returns a Match object if there is a match anywhere in the string
    - there is more than one match, only the first occurrence of the match will be returned
    - A Match Object is an object containing information about the search and the result.
    - The Match object has properties and methods used to retrieve information about the search, and the result:
      .span() returns a tuple containing the start-, and end positions of the match. .string returns the string passed into the function .group() returns the part of the string where there was a match
  - findall Returns a list containing all matches

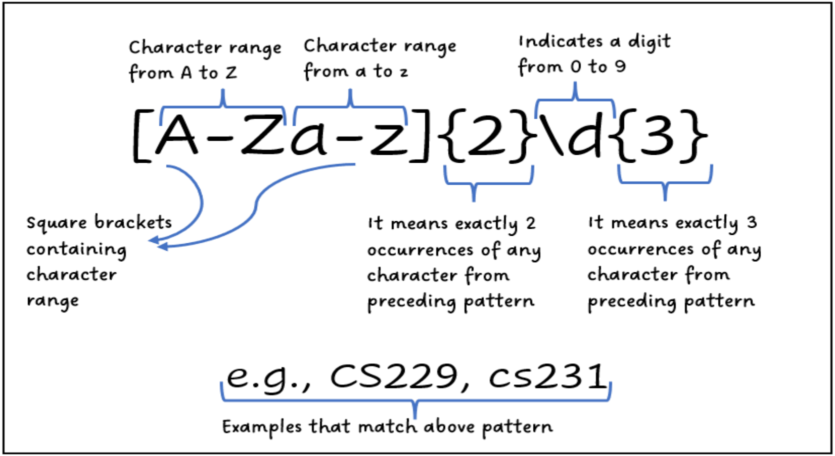
regular expression

```
match=re.search('Hi','Helli Himanshu')
match
```

```
<re.Match object; span=(6, 8), match='Hi'>
```

```
match.group()
```

```
'Hi'
```



regular expression

```
s='SOACS219ITEREE223'
import re
r='[A-Za-z]{2}\d{3}'
match=re.search(r,s)
match.group()
```

```
'CS219'
```

```
re.findall(r,s)
```

```
['CS219', 'EE223']
```

- Can you guess the output of the following code?

```
my_str = '''Hi my name is Ashis and gmail address is
ashisinmath@gmail.com and soa email is ashispati@soa.ac.in'''
r="([\w_.+-]+@[\w]+\.[\w]+)"
#r="([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)"
re.findall(r, my_str)
```

string functions in pandas

- Vectorized string functions for Series and Index.
- **NAs** stay **NA** unless handled otherwise by a particular method.

```
data = {"Ashis": "ashisinmath@gmail.com",
        "Modi": "iamnarendra@gmail.com",
        "Manu": "manoj@soa.ic.in", "Naveen": np.nan}
data = pd.Series(data)
data
```

```
Ashis     ashisinmath@gmail.com
Modi      iamnarendra@gmail.com
Manu            manoj@soa.ic.in
Naveen                      NaN
dtype: object
```

```
data.str.contains("gmail")
```

```
Ashis      True
Modi       True
Manu      False
Naveen      NaN
dtype: object
```

```
data.str[:5]
```

```
Ashis     ashis
Modi      iamna
Manu      manoj
Naveen      NaN
dtype: object
```

```
data.str.split("@")
```

```
Ashis     [ashisinmath, gmail.com]
Modi      [iamnarendra, gmail.com]
Manu            [manoj, soa.ic.in]
Naveen                         NaN
dtype: object
```

Categorical data

- A categorical variable takes on a fixed, number of possible values (categories). Example gender variable.
- Categoricals are a pandas data type corresponding to categorical variables in statistics
- the data structure consists of a categories array and an integer array of codes which point to the real value in the categories array.