

# **Data Cleaning and Preparation**

**Prof. Gheith Abandah**

# Reference

- **Chapter 7**
- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
  - Material: <https://github.com/wesm/pydata-book>

# Data Cleaning and Preparation

- Data preparation: loading, cleaning, transforming, and rearranging may take up **80%** or more of an analyst's time.
- Pandas and the built-in Python language features provide **high-level, flexible**, and **fast** set of tools to manipulate data into the right form.

# Outline

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation

# Outline

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation

- Introduction
- Filtering out Missing Data
- Filling in Missing Data

# 7.1 Handling Missing Data

- Missing data is **common**.
- Try first to **find out** why data is missing and try to **correct** it.
- **Cases:**
  - `np.nan`
  - `None`
  - Files with `NA` ,, ...
- **Check** with `isnull` and `notnull`.

```
string_data = pd.Series([  
    'aardvark', 'artichoke',  
    np.nan, 'avocado'])
```

```
string_data[0] = None
```

```
string_data.isnull()
```

```
0    True
```

```
1    False
```

```
2    True
```

```
3    False
```

# Filtering out Missing Data

- How to **check** whether there are **any missing** value?

```
s.isnull().any()  
df.isnull().values.any()
```

- **Options:**

1. **Remove rows with some NA**
2. Remove rows with **all NA**
3. Remove **columns**

```
from numpy import nan as NA
```

# Filtering out Missing Data

- Use **dropna** on a **Series** to get the Series with only the non-null data and index values.

```
s = pd.Series([1, NA, 3.5, NA, 7])
```

```
s.dropna()
```

```
0    1.0
```

```
2    3.5
```

```
4    7.0
```



# Filtering out Missing Data

- Use **dropna** on a **DataFrame** to drop **rows** with NA.
- To drop **columns**, pass **axis=1**.
- Pass **how='all'** to drop rows or columns that are **all NA**.
- **Hint:** To drop when there are *n* or more NA, use **thresh=n**.

```
data = pd.DataFrame([[1., 6.5, 3.],  
                    [1., NA, NA],  
                    [NA, NA, NA]])
```

```
cleaned = data.dropna()
```

```
cleaned
```

```
      0      1      2  
0  1.0  6.5  3.0
```

```
data[2][0] = NA
```

```
data.dropna(axis=1, how='all')
```

```
      0      1  
0  1.0  6.5  
1  1.0  NaN  
2  NaN  NaN
```

# Filling in Missing Data

- **Options:**

1. Fill with a **scalar**
2. Fill **values per column**
3. Fill **forward**
4. Fill **backward**
5. Fill **mean, median, ...**

# Filling in Missing Data

- To **fill** in the “**holes**”, you can use panda’s **fillna** method.
- To use a **different fill value** for each column, pass a **dict**.

For in-place fill, use:

```
_ = data.fillna(0, inplace=True)
```

```
data.fillna(0)
```

```
      0      1      2
0  1.0  6.5  0.0
1  1.0  0.0  0.0
2  0.0  0.0  0.0
```

```
data.fillna({1: 0.5, 2: 0})
```

```
      0      1      2
0  1.0  6.5  0.0
1  1.0  0.5  0.0
2  NaN  0.5  0.0
```

```
data
      0      1      2
0  1.0  6.5  NaN
1  1.0  NaN  NaN
2  NaN  NaN  NaN
```

# Filling in Missing Data

- To **propagate last** valid observation **forward** (backward), use **method=ffill** (**bfill**).
- To fill using **mean** or **median**, use **mean()** or **median()**.

data	0	1	2
0	1.0	6.5	NaN
1	1.0	NaN	NaN
2	NaN	NaN	NaN

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

	0	1	2
0	1.0	6.5	NaN
1	1.0	6.5	NaN
2	1.0	6.5	NaN

```
data.fillna(data.mean())
```

# Outline

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation

- Removing Duplicates
- Transforming Data Using a Function or Mapping
- Replacing Values
- Renaming Axis Indexes
- Discretization and Binning
- Detecting and Filtering Outliers
- Permutation and Random Sampling
- Computing Indicator/Dummy Variables

# Removing Duplicates

- To **check** if your pandas object has duplicate, use  **duplicated**.
- You can easily **remove duplicates** using  **drop\_duplicates**.

```
data.duplicated()
```

```
0 False  
1 False  
2 False  
3  True
```

```
data.drop_duplicates()
```

```
   k1  k2  
0  one  1  
1  two  1  
2  one  2
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	1

# Removing Duplicates

- You can drop duplicates based on a **list of columns**.
- You can also **keep last** duplicate instead of first.

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	1

```
data.drop_duplicates(['k1'],  
                    keep='last')
```

	k1	k2
2	one	2
3	two	1

# Transforming Data Using a Function or Mapping

- You can use the **map** method to **transform** the **items** in a series using a **dict** or **function**.

- How to transform?

before		
	k1	k2
0	one	10
1	two	1
2	one	20
3	two	2

after		
	k1	k2
0	1	large
1	2	small
2	1	large
3	2	small

```
d = {'one': 1, 'two': 2}
```

```
df.k1 = df.k1.map(d)
```

```
df.k2 = df.k2.map(lambda x: 'large'  
                  if x > 9 else 'small')
```



# Replacing Values

- You can replace values in a pandas object using **replace**.

```
df = df.replace(  
    [999, 'one', 'two'],  
    [np.nan, 1, 2])
```

- How to transform?

	before		after	
	k1	k2	k1	k2
0	one	10	1	10
1	two	1	2	1
2	one	20	1	20
3	two	999	2	NaN

Accepts:

- List to list
- List to scalar
- Scalar to scalar
- Dictionary

# Renaming Axis Indexes

- You can also modify axis indices using the **map** method or the **rename** function.

	k1	k2
0	one	10
1	two	1
2	one	20
3	two	999

	K1	K2
0	one	10
1	two	1
4	one	20
9	two	999

```
trans = lambda x: x * x
```

Also **.columns**.

```
df.index = df.index.map(trans)
```

```
df.rename(columns=str.upper,  
         inplace=True)
```

There is also rename **index**.

# Discretization and Binning

- **Continuous data** is often **discretized** or put into **bins** for analysis.
- How to assign marks to grades using pandas **cut**?

```
scores = pd.DataFrame([93, 85, 87, 83, 79, 77, 73, 69, 67, 60, 0],  
                      columns=['Mark'])  
bins = pd.Series([0, 40, 55, 70, 85, 100])  
grades = list('FDCBA')
```

```
cats = pd.cut(scores.Mark.values,  
              bins.values, right=False,  
              labels=grades)  
scores['Grade'] = cats
```

```
pd.value_counts(cats)  
B    4  
A    3  
C    3  
F    1  
D    0
```

scores		
	Mark	Grade
0	93	A
1	85	A
2	87	A
3	83	B
4	79	B
5	77	B
6	73	B
7	69	C
8	67	C
9	60	C
10	0	F

# Discretization and Binning

- You can assign values to equal-size  $n$  bins.

```
data = np.random.rand(20)
array([0.96404512, ..., 0.29071928])

cats = pd.cut(data, 4, precision=2)

cats.codes
array([3, ..., 0], dtype=int8)

cats.categories
IntervalIndex(
  [(0.066, 0.3], (0.3, 0.53],
   (0.53, 0.76], (0.76, 0.99]])
```

# Discretization and Binning

- To bin data on **sample quantiles**, use **qcut**.
- **Example**: Split 1000 random numbers to **4 quantiles**.
- You can also pass **your own quantiles** list (numbers between 0 and 1, inclusive).

```
pd.qcut(data, [0, 0.1, 0.5,  
              0.9, 1.])
```

```
data = np.random.randn(1000)  
cats = pd.qcut(data, 4)  
Categories (4,  
  [(-2.95, -0.68] <  
  (-0.68, -0.0265] <  
  (-0.0265, 0.62] <  
  (0.62, 3.928])  
pd.value_counts(cats)  
(0.62, 3.928]      250  
(-0.0265, 0.62]   250  
(-0.68, -0.0265] 250  
(-2.95, -0.68]   250
```

# Detecting and Filtering Outliers

- You can filter or transform outliers by applying **array operations**.
- How to **detect rows** with outliers outside  $\pm 1.5$ ?

```
data
      0      1      2
0 -0.245685 -1.562837 -1.292501
1 -0.779216 -0.296007  0.168335
2  2.104516 -0.268707  1.272592
3  0.222063  0.181163  0.211443
```

```
data[(np.abs(data) > 1.5).any(1)]
      0      1      2
0 -0.245685 -1.562837 -1.292501
2  2.104516 -0.268707  1.272592
```

# Detecting and Filtering Outliers

- How to **replace outliers** with  $\pm 1.5$ ?

data	0	1	2
0	-0.245685	-1.562837	-1.292501
1	-0.779216	-0.296007	0.168335
2	2.104516	-0.268707	1.272592
3	0.222063	0.181163	0.211443

```
data[np.abs(data) > 1.5] =  
    np.sign(data) * 1.5
```

```
data  
      0      1      2  
0 -0.245685 -1.500000 -1.292501  
1 -0.779216 -0.296007  0.168335  
2  1.500000 -0.268707  1.272592  
3  0.222063  0.181163  0.211443
```

# Permutation and Random Sampling

- **Permute (randomly reorder)** a Series or the rows of a DataFrame by **`np.random.permutation`**.
- To select a **random subset** without replacement, use **`sample`**.

```
df.sample(n=3)
   0  1  2  3
3 12 13 14 15
4 16 17 18 19
2  8  9 10 11
```

```
df = pd.DataFrame(
    np.arange(5*4).reshape((5, 4)))
sampler = np.random.permutation(5)
sampler
array([3, 1, 4, 2, 0])
```

```
df.take(sampler)
   0  1  2  3
3 12 13 14 15
1  4  5  6  7
4 16 17 18 19
2  8  9 10 11
0  0  1  2  3
```



# Permutation and Random Sampling

- To generate a **sample with replacement** (to allow repeat choices), pass **replace=True**.

```
choices = pd.Series(  
    [5, 7, -1, 6, 4])  
draws = choices.sample(n=6,  
    replace=True)
```

```
draws  
4 4  
1 7  
4 4  
2 -1  
0 5  
3 6
```

# Computing Indicator/Dummy Variables

- The `get_dummies()` function is used to convert categorical variable into dummy/indicator variables.
- This is **converting** a **category** to **one-hot** encoding.

```
df = pd.DataFrame({'key':  
    ['b', 'b', 'a', 'c', 'a', 'b'],  
    'data1': range(6)})  
dummies = pd.get_dummies(df['key'],  
    prefix='key')  
df[['data1']].join(dummies)
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

# Outline

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation

- String Object Methods
- Regular Expressions
- Vectorized String Functions in pandas

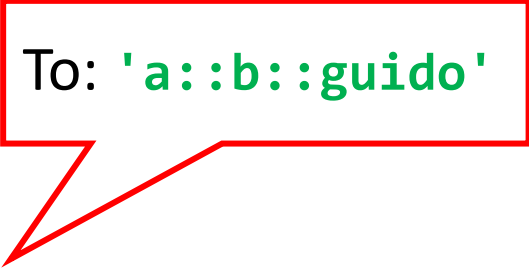
# String Object Methods

- **pandas** enables you to apply **string** and **regular expressions** concisely on whole **arrays** of data.
- In many string munging and scripting applications, **built-in string** methods are **sufficient**.
- **Examples**

```
val = 'a,b, guido'
pieces = [x.strip() for x in
          val.split(',')]

pieces
['a', 'b', 'guido']

'::'.join(pieces)
'a::b::guido'
```



To: 'a::b::guido'

# String Object Methods

```
val = 'a,b, guido'  
'guido' in val  
True
```

```
val.index(',')  
1
```

Error if not found

```
val.find(':')  
-1
```

```
val.count(',')  
2
```

```
val.replace(',', '::')  
'a::b:: guido'
```

```
val.replace(',', '')  
'ab guido'
```

# Python built-in string methods

Argument	Description
<code>count</code>	Return the number of non-overlapping occurrences of substring in the string.
<code>endswith</code>	Returns <code>True</code> if string ends with suffix.
<code>startswith</code>	Returns <code>True</code> if string starts with prefix.
<code>join</code>	Use string as delimiter for concatenating a sequence of other strings.
<code>index</code>	Return position of first character in substring if found in the string; raises <code>ValueError</code> if not found.
<code>find</code>	Return position of first character of <i>first</i> occurrence of substring in the string; like <code>index</code> , but returns <code>-1</code> if not found.
<code>rfind</code>	Return position of first character of <i>last</i> occurrence of substring in the string; returns <code>-1</code> if not found.
<code>replace</code>	Replace occurrences of string with another string.

# Python built-in string methods – cont.

<code>strip,</code> <code>rstrip,</code> <code>rstrip</code>	Trim whitespace, including newlines; equivalent to <code>x.strip()</code> (and <code>rstrip</code> , <code>rstrip</code> , respectively) for each element.
<code>split</code>	Break string into list of substrings using passed delimiter.
<code>lower</code>	Convert alphabet characters to lowercase.
<code>upper</code>	Convert alphabet characters to uppercase.
<code>casefold</code>	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
<code>ljust,</code> <code>rjust</code>	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

---

# Regular Expressions

- **Regular expressions** provide a **flexible** way to **search** or **match string patterns** in text.
- **regex**, is a **string** formed according to the **regular expression language**.
- Python has built-in **re** module.

```
import re
text = "foo    bar\t baz  \tqux"
# split a string with a variable
# number of whitespace characters
re.split('\s+', text)
['foo', 'bar', 'baz', 'qux']
```



# Regular Expressions

- You can compile the regex with **re.compile**, forming a reusable regex object.
- **Better performance** when using the regex multiple times.

```
regex = re.compile('\s+')  
regex.split(text)  
['foo', 'bar', 'baz', 'qux']
```

```
regex.findall(text)  
[' ', '\t ', ' \t']
```

# Regular Expressions

- **Example:** finding email addresses.

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
```

```
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
```

```
regex = re.compile(pattern,
                    flags=re.IGNORECASE)
```

```
regex.findall(text)
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

# Regular expression methods

Argument	Description
<code>findall</code>	Return all non-overlapping matching patterns in a string as a list
<code>finditer</code>	Like <code>findall</code> , but returns an iterator
<code>match</code>	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise <code>None</code>
<code>search</code>	Scan string for match to pattern; returning a match object if so; unlike <code>match</code> , the match can be anywhere in the string as opposed to only at the beginning
<code>split</code>	Break string into pieces at each occurrence of pattern
<code>sub</code> , <code>subn</code>	Replace all ( <code>sub</code> ) or first <code>n</code> occurrences ( <code>subn</code> ) of pattern in string with replacement expression; use symbols <code>\1</code> , <code>\2</code> , <code>...</code> to refer to match group elements in the replacement string

# Vectorized String Functions in pandas

- Panda's Series has **array-oriented methods** for string operations that **skip NA** values through the **str** attribute.

```
obj
```

```
Dave    dave@google.com  
Rob      rob@gmail.com  
Steve   steve@gmail.com  
Wes                NaN
```

```
obj.str[:5]
```

```
Dave    dave@  
Rob     rob@g  
Steve   steve  
Wes     NaN
```

```
obj.str.contains('gmail')
```

```
Dave    False  
Rob     True  
Steve   True  
Wes     NaN
```

# Vectorized String Functions in pandas

- **Example:** find all email addresses and split them to three parts.

Note the paranthesis

**pattern**

```
'([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.\.([A-Z]{2,4}))'
```

```
obj.str.replace(
    '@.', ' ').str.split()
# Or
obj.str.findall(pattern,
                flags=re.IGNORECASE)
Dave      [(dave, google, com)]
Rob       [(rob, gmail, com)]
Steve     [(steve, gmail, com)]
Wes       [NaN]
```

# Partial listing of vectorized string methods

Method	Description
<code>cat</code>	Concatenate strings element-wise with optional delimiter
<code>contains</code>	Return boolean array if each string contains pattern/regex
<code>count</code>	Count occurrences of pattern
<code>extract</code>	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
<code>endswith</code>	Equivalent to <code>x.endswith(pattern)</code> for each element
<code>startswith</code>	Equivalent to <code>x.startswith(pattern)</code> for each element
<code>findall</code>	Compute list of all occurrences of pattern/regex for each string
<code>get</code>	Index into each element (retrieve <i>i</i> -th element)

# Partial listing of vectorized string methods – cont.

<code>isalnum</code>	Equivalent to built-in <code>str.isalnum</code>
<code>isalpha</code>	Equivalent to built-in <code>str.isalpha</code>
<code>isdecimal</code>	Equivalent to built-in <code>str.isdecimal</code>
<code>isdigit</code>	Equivalent to built-in <code>str.isdigit</code>
<code>islower</code>	Equivalent to built-in <code>str.islower</code>
<code>isnumeric</code>	Equivalent to built-in <code>str.isnumeric</code>
<code>isupper</code>	Equivalent to built-in <code>str.isupper</code>
<code>join</code>	Join strings in each element of the Series with passed separator
<code>len</code>	Compute length of each string
<code>lower, upper</code>	Convert cases; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element

# Partial listing of vectorized string methods – cont.

<code>match</code>	Use <code>re.match</code> with the passed regular expression on each element, returning matched groups as list
<code>pad</code>	Add whitespace to left, right, or both sides of strings
<code>center</code>	Equivalent to <code>pad(side='both')</code>
<code>repeat</code>	Duplicate values (e.g., <code>s.str.repeat(3)</code> is equivalent to <code>x * 3</code> for each string)
<code>replace</code>	Replace occurrences of pattern/regex with some other string
<code>slice</code>	Slice each string in the Series
<code>split</code>	Split strings on delimiter or regular expression
<code>strip</code>	Trim whitespace from both sides, including newlines
<code>rstrip</code>	Trim whitespace on right side
<code>lstrip</code>	Trim whitespace on left side



# Summary

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation