Co-funded by the
Erasmus+ Programme
of the European Union

DeCAIR

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

```
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
```

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

```
data.duplicated()
0 False
1 False
2 False
3  True
```

```
data.drop_duplicates()
    k1  k2
0  one   1
1  two   1
2  one   2
```

Removing Duplicates

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

```
data
    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**.

- How to transform?

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

```
after
   k1   k2
0   1   10
1   2    1
2   1   20
3   2  NaN
```

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

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.

```
trans = lambda x: x * x

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

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

Also **.columns**.

There is also rename **index**.

before

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

after

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

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 quartiles**.
- 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 ±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 ±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

- **Python** 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**

To: 'a::b::guido'

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

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

String Object Methods

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

val.index(',')
1

val.find(':')
-1

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

Error if not found

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

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

Python built-in string methods

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

Python built-in string methods – cont.

`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust`, `rjust`	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
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like `findall`, but returns an iterator
`match`	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise `None`
`search`	Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub, subn`	Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` 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.

```
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
```

Note the paranthesis

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

Partial listing of vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	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
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)

Partial listing of vectorized string methods – cont.

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

Partial listing of vectorized string methods – cont.

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

Summary

7.1 Handling Missing Data

7.2 Data Transformation

7.3 String Manipulation