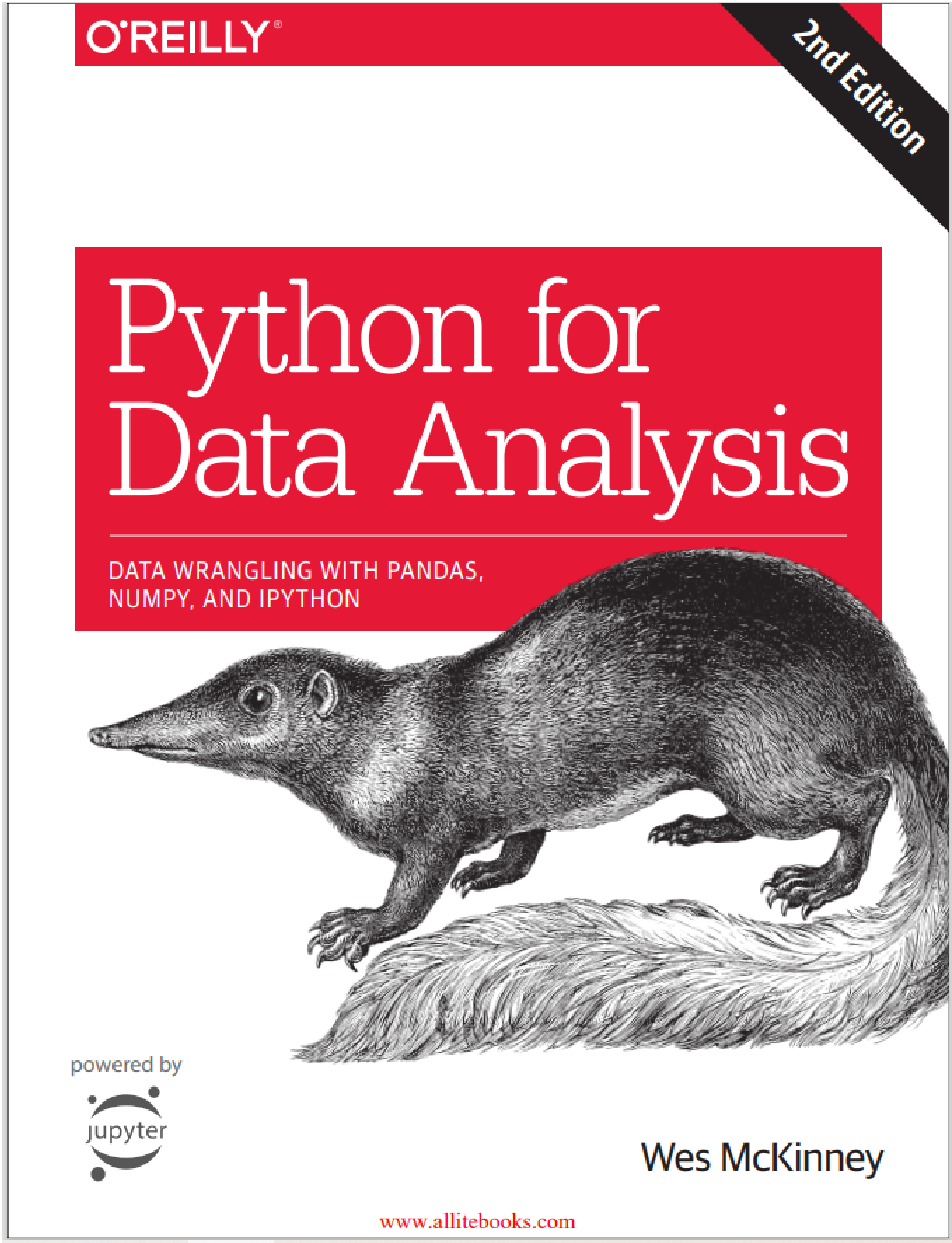
Data Cleaning and Preparation

Book

Python for Data Analysis



CH 7 : Data Cleaning and Preparation

chapter introduction

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of an analyst's time. Sometimes the way that data is

stored in files or databases is not in the right format for a particular task. Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk. Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

If you identify a type of data manipulation that isn't anywhere in this book or else- where in the pandas library, feel free to share your use case on one of the Python mailing lists or on the pandas GitHub site. Indeed, much of the design and imple- mentation of pandas has been driven by the needs of real-world applications.

In this chapter I discuss tools for missing data, duplicate data, string manipulation, and some other analytical data transformations. In the next chapter, I focus on com- bining and rearranging datasets in various ways.

In [6]:
```
import numpy as np
from numpy import nan as NA
import pandas as pd
```

Handling Missing Data

In [2]:
```
dt = pd.Series([np.nan,1,25,np.nan,60,99])
```

In [3]:
```
dt
```

Out[3]:
```
0     NaN
1     1.0
2    25.0
3     NaN
4    60.0
5    99.0
dtype: float64
```

In [4]:
```
dt.dropna()
```

Out[4]:
```
1     1.0
2    25.0
4    60.0
5    99.0
dtype: float64
```

In [5]:
```
dt[dt.notnull()]
```

Out[5]:
```
1     1.0
2    25.0
4    60.0
5    99.0
dtype: float64
```

In [6]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],[NA, NA, NA], [NA, 6.5, 3.]])
```

In [7]:
```
data
```

Out[7]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

if we passing how = 'all' to drop func, it 'll remove all rows that are all NA

In [8]:
```
data.dropna(how = 'all')
```

Out[8]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

if we passing how = 'any' to drop func, it 'll remove all rows that have at least NA

In [9]:
```
data.dropna(how = 'any')
```

Out[9]:

	0	1	2
0	1.0	6.5	3.0

In [10]:
```
data[4] = NA
data[5] = 5.0
```

In [11]:
```
data
```

Out[11]:

	0	1	2	4	5
0	1.0	6.5	3.0	NaN	5.0
1	1.0	NaN	NaN	NaN	5.0
2	NaN	NaN	NaN	NaN	5.0
3	NaN	6.5	3.0	NaN	5.0

we can do the same way to drop columns by passing axis = 1

In [12]:
```
data.dropna(axis = 1, how = 'all')
```

Out[12]:

	0	1	2	5
0	1.0	6.5	3.0	5.0
1	1.0	NaN	NaN	5.0
2	NaN	NaN	NaN	5.0
3	NaN	6.5	3.0	5.0

In [13]:
```
data.dropna(axis = 1, how = 'any')
```

Out[13]:

	5
0	5.0
1	5.0
2	5.0
3	5.0

In [ ]:
```
data = data.drop(6, axis = 1)
```

If you want to keep only rows containing a certain number of observations. You canindicate this with the thresh argument

In [ ]:
```
data.dropna(thresh=2)
```

In [ ]:
```
data
```

In [ ]:
```
[data.dropna(thresh=i) for i in range(1,6)]
```

Filling In Missing Data

to fill NA values, we can replace them with real values:

In [ ]:
```
data.fillna(0)
```

you can use a different fill value for each column:

In [16]:
```
data.fillna({0:0, 1: 1, 2: 2, 3:3, 4:4 , 5:5})
```

Out[16]:

	0	1	2	4	5
0	1.0	6.5	3.0	4.0	5.0
1	1.0	1.0	2.0	4.0	5.0
2	0.0	1.0	2.0	4.0	5.0
3	0.0	6.5	3.0	4.0	5.0

In [15]:
```
data.fillna(data.mean())
```

Out[15]:

	0	1	2	4	5
0	1.0	6.5	3.0	NaN	5.0
1	1.0	6.5	3.0	NaN	5.0
2	1.0	6.5	3.0	NaN	5.0
3	1.0	6.5	3.0	NaN	5.0

Removing Duplicates

In [17]:
```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],'k2': [1, 1, 2, 3, 3, 4,
```

In [18]:
```
data
```

Out[18]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

In [19]:
```
data.duplicated()
```

Out[19]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

drop_duplicates returns a DataFrame where the duplicated array is False

In [20]:
```
data.drop_duplicates()
```

Out[20]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

Both of these methods by default consider all of the columns; alternatively, you canspecify any subset of them to detect duplicates

In [23]:
```
data['v1'] = range(7)
```

In [24]:
```
data
```

Out[24]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

In [25]:
```python
data.drop_duplicates(['k1'])
```

Out[25]:

	k1	k2	v1
0	one	1	0
1	two	1	1

duplicated and drop_duplicates by default keep the first observed value combination. Passing keep='last' will return the last one:

In [26]:
```python
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[26]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

Maping

- **For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame.**
- **we use map function to perform like this transformation**
- **Using map is a convenient way to perform element-wise transformations and otherdata cleaning–related operations.**

In [27]:
```python
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon','Pastrami', 'cornec
                     'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

In [28]:
```
data
```

Out[28]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

In [29]:
```
meat_to_animal = {
 'bacon': 'pig',
 'pulled pork': 'pig',
 'pastrami': 'cow',
 'corned beef': 'cow',
 'honey ham': 'pig',
 'nova lox': 'salmon'
}
```

Here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the str.lower Series method:

In [31]:
```
lowercased = data['food'].str.lower()
```

In [32]:
```
lowercased
```

Out[32]:
```
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

In [33]:
```
data['animal'] = lowercased.map(meat_to_animal)
```

In [34]:
```
data
```

Out[34]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

Replacing Values

Filling in missing data with the fillna method is a special case of more general value replacement. As you've already seen, map can be used to modify a subset of values in an object but replace provides a simpler and more flexible way to do so. Let's con- sider this Series:

In [38]:
```
data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

In [39]:
```
data
```

Out[39]:
```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

- **If you want to replace multiple values at once, you instead pass a list and then thesubstitute value:**

- **To use a different replacement for each value, pass a list of substitutes:**
- **The argument passed can also be a dict:**

In [40]:
```
data.replace(-999, np.nan)
```

Out[40]:
```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

In [41]:
```
data.replace([-999, -1000], np.nan)
```

Out[41]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

In [43]:
```
data.replace([-999, -1000], [np.nan, 0])
```

Out[43]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

In [42]:
```
data.replace({-999: np.nan, -1000: 0})
```

Out[42]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Renaming Axis Indexes

In [46]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                    index=['Ohio', 'Colorado', 'New York'],

                    columns=['one', 'two', 'three', 'four'])
```

In [47]:
```
data
```

Out[47]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

Like a Series, the axis indexes have a map method

In [49]:
```
transform = lambda x: x[:4].upper()

data.index.map(transform)
```

Out[49]:
```
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

In [50]:
```
data.index = data.index.map(transform)
```

In [51]:
```
data
```

Out[51]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [55]:
```
data.rename(index=str.title, columns=str.upper)
```

Out[55]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

Notably, rename can be used in conjunction with a dict-like object providing new val- ues for a subset of the axis labels:

In [ ]:
```
data.rename(index={'OHIO': 'INDIANA'},columns={'three': 'peekaboo'})
```

rename saves you from the chore of copying the DataFrame manually and assigning to its

index and columns attributes. Should you wish to modify a dataset in-place, pass inplace=True:

In [59]:
```
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

In [60]:
```
data
```

Out[60]:

	one	two	three	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Discretization and Binning

In [63]:
```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let's divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use cut, a function in pandas:

In [65]:
```
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
```

In [66]:
```
cats
```

Out[66]:
```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (3
5, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

The object pandas returns is a special Categorical object. The output you see describes the bins computed by pandas.cut. You can treat it like an array of strings indicating the bin name; internally it contains a categories array specifying the dis- tinct category names along with a labeling for the ages data in the codes attribute:

In [67]:
```
cats.codes
```

Out[67]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

In [68]:
```python
cats.categories
```

Out[68]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')
```

In [69]:
```python
pd.value_counts(cats)
```

Out[69]:
```
(18, 25]     5
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

In [73]:
```python
data = pd.DataFrame(np.random.randn(1000, 4))
```

In [74]:
```python
data.describe()
```

Out[74]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.055569	0.005852	0.006234	-0.057624
std	0.975254	1.048993	0.955249	0.976422
min	-3.383996	-3.452649	-2.916598	-3.118643
25%	-0.581172	-0.690927	-0.655540	-0.742554
50%	0.053096	0.005580	0.033484	-0.084417
75%	0.716883	0.727454	0.636604	0.646606
max	3.145359	3.140651	3.025406	2.836423

Suppose you wanted to find values in one of the columns exceeding 3 in absolutevalue:

In [79]:
```python
col = data[2]
```

In [80]:
```
col[np.abs(col) > 3]
```

Out[80]:
```
127    3.025406
Name: 2, dtype: float64
```

To select all rows having a value exceeding 3 or –3, you can use the any method on a boolean DataFrame:

In [82]:
```
data[(np.abs(data) > 3).any(1)]
```

Out[82]:

	0	1	2	3
9	1.182853	3.140651	-0.010715	0.103472
127	-0.573109	-3.200945	3.025406	1.976730
596	-0.024790	-1.537310	1.391747	-3.118643
794	3.145359	-0.843419	1.006886	1.213411
883	0.035233	-3.452649	0.039828	1.309069
987	-3.383996	-0.824807	0.893779	0.988303
991	-0.464648	3.071497	1.982264	-1.456189

Values can be set based on these criteria. Here is code to cap values outside the interval –3 to 3:

In [83]:
```
data[np.abs(data) > 3] = np.sign(data) * 3
```

In [84]:
```
data.describe()
```

Out[84]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.055807	0.006293	0.006208	-0.057505
std	0.973523	1.046380	0.955169	0.976057
min	-3.000000	-3.000000	-2.916598	-3.000000
25%	-0.581172	-0.690927	-0.655540	-0.742554
50%	0.053096	0.005580	0.033484	-0.084417
75%	0.716883	0.727454	0.636604	0.646606
max	3.000000	3.000000	3.000000	2.836423

The statement np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative:

In [88]:
```
data.head()
```

Out[88]:

	0	1	2	3
0	-0.758658	-0.755808	-0.424850	-1.266991
1	0.860276	-1.892115	-0.954768	0.044466
2	0.195563	-1.523395	0.427470	0.645392
3	-0.530915	-0.079130	0.402178	-0.777428
4	-0.556665	-0.343721	-0.200321	-0.818012

In [89]:
```
np.sign(data).head()
```

Out[89]:

	0	1	2	3
0	-1.0	-1.0	-1.0	-1.0
1	1.0	-1.0	-1.0	1.0
2	1.0	-1.0	1.0	1.0
3	-1.0	-1.0	1.0	-1.0
4	-1.0	-1.0	-1.0	-1.0

Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the numpy.random.permutation function. Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

In [99]:
```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
sampler = np.random.permutation(5)
```

In [100]:
```
df
```

Out[100]:

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

In [101]:
```
sampler
```

Out[101]:
```
array([4, 0, 3, 2, 1])
```

In [102]:
```
df.take(sampler)
```

Out[102]:

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

To select a random subset without replacement, you can use the sample method on Series and DataFrame:

In [105]:
```
df.sample(n=3)
```

Out[105]:

	0	1	2	3
2	8	9	10	11
0	0	1	2	3
3	12	13	14	15

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

In [112]:
```
choices = pd.Series([5, 7, -1, 6, 4])
draws = choices.sample(n=10, replace=True)
```

In [113]:
```
draws
```

Out[113]:
```
1    7
1    7
3    6
4    4
2   -1
0    5
4    4
3    6
0    5
2   -1
dtype: int64
```

Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is 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 or DataFrame with k columns containing all 1s and 0s. pandas has a get_dummies functionfor doing this, though devising one yourself is not difficult. Let’s return to an earlier example DataFrame:

In [114]:
```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)})

pd.get_dummies(df['key'])
```

Out[114]:

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. get_dummies has a prefix argu-ment for doing this:

In [115]:
```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = df[['data1']].join(dummies)
df_with_dummy
```

Out[115]:

	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

A useful recipe for statistical applications is to combine get_dummies with a discreti- zation function like cut:

In [120]:
```
np.random.seed(12345)
values = np.random.rand(10)
```

In [121]:
```
values
```

Out[121]:
```
array([0.92961609, 0.31637555, 0.18391881, 0.20456028, 0.56772503,
       0.5955447 , 0.96451452, 0.6531771 , 0.74890664, 0.65356987])
```

In [122]:
```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

In [123]:
```
pd.get_dummies(pd.cut(values, bins))
```

Out[123]:

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0

String Manipulation

Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by ena- bling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

a comma-separated string can be broken into pieces with split:

In [3]:
```
val = 'Mo,Ramdone, Ahmd'
val.split(',')
```

Out[3]: `['Mo', 'Ramdone', ' Ahmd']`

In [4]:
```
pieces = [x.strip() for x in val.split(',')]
```

In [5]:
```
pieces
```

Out[5]: `['Mo', 'Ramdone', 'Ahmd']`

In [7]:
```
name, middle, last = pieces
```

In [10]:
```
name + ' '+ middle + ' '+ last
```

Out[10]: `'Mo Ramdone Ahmd'`

In [11]:
```
" ".join(pieces)
```

Out[11]: `'Mo Ramdone Ahmd'`

The difference between index and find, We use both to detect a substring. They back 1 (True) if found but in case of not finding index raises an exception and find backs (-1).

In [16]:
```
val.index('o')
```

Out[16]: `1`

In [18]:
```
val.find('o')
```

Out[18]: `1`

In [17]:
```
val.index('z')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-8b62d7293a10> in <module>
----> 1 val.index('z')

ValueError: substring not found
```

In [19]:
```
val.find('z')
```

Out[19]: `-1`

In [20]:
```
val.count(',')
```
Out[20]:
```
2
```
In [25]:
```
val.count('m')
```
Out[25]:
```
2
```
In [26]:
```
val.replace(',' , ' ')
```
Out[26]:
```
'Mo Ramdone  Ahmd'
```

Regular Expressions

Regular expressions provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a regex, is a string formed according to the regular expression language. Python's built-in re module is responsible for applying regular expressions to strings; I'll give a number of examples of its use here

In [26]:
```
import re
```

The re module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes.

In [27]:
```
text = "foo bar\t baz \tqux"
```

we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is \s+:

In [28]:
```
re.split('\s+', text)
```
Out[28]:
```
['foo', 'bar', 'baz', 'qux']
```

let's consider a block of text and a regular expression capable of identifying most email addresses:

In [29]:

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

In [30]:

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

In [31]:

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

Using findall on the text produces a list of the email addresses:

In [32]:

```python
regex.findall(text)
```

Out[32]:

```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

In [33]:

```python
m = regex.search(text)
```

In [34]:

```python
m
```

Out[34]:

```
<re.Match object; span=(5, 20), match='dave@google.com'>
```

In [35]:

```python
text[m.start():m.end()]
```

Out[35]:

```
'dave@google.com'
```

In [36]:

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

In [37]:

```python
print(regex.sub('REDACTED', text))
```

```
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

In [38]:

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

In [39]:
```
regex.findall(text)
```

Out[39]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

In [40]:
```
 m = regex.match('wesm@bright.net')
```

In [41]:
```
m.groups()
```

Out[41]:
```
('wesm', 'bright', 'net')
```

In [42]:
```
regex.findall(text)
```

Out[42]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

In [43]:
```
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

```
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Vectorized String Functions in pandas

Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

In [44]:
```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
          'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

In [45]:
```
data = pd.Series(data)
```

In [46]:
```
data
```

Out[46]:
```
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

In [47]:
```
data.isnull()
```

Out[47]:
```
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

In [48]:
```
data.str.contains('gmail')
```

Out[48]:
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

You can apply string and regular expression methods can be applied (passing alambda or other function) to each value using data.map, but it will fail on the NA (null) values. To cope with this, Series has array-oriented methods for string operations that skip NA values. These are accessed through Series's str attribute; for example, we could check whether each email address has 'gmail' in it with str.contains:

Regular expressions can be used, too, along with any re options like IGNORECASE:

In [49]:
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

Out[49]:
```
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
```

In [50]:
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
```

In [51]:
```
matches
```

Out[51]:
```
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

We can similarly slice strings using this syntax:

In [54]:
```
data.str[:5]
```

Out[54]:
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

Conclusion

Effective data preparation can significantly improve productive by enabling you to spend more time analyzing data and less time getting it ready for analysis. We have explored a number of tools in this chapter, but the coverage here is by no means com- prehensive. In the next chapter, we will explore pandas's joining and grouping func- tionality.