Chapter 2

Programming With Tabular Data

As datasets have grown in size, we are increasingly using computing to make sense of such data. It makes sense to initially focus on tabular data (also called structured data). In tabular data, we are guaranteed that every row has a known set of columns and their data types.

Many programming languages have libraries that are specifically designed for manipulating and working with tabular data. Our course will focus on using Python for data analysis. Python is emerging as an industry and academic standard in data science. Within Python we will use the `pandas` library. Pandas provides a basic API for manipulating tabular data.

2.1. Getting Started

To get started with `pandas`, we first must import the necessary Python modules (assuming you have installed them of course!)

```
import pandas as pd
import numpy as np
```

The following text is not meant to be an exhaustive tutorial on pandas; there are many of those freely available on the internet. Rather, we highlight

the key design decisions in the Pandas API and how they link to the previous discussion about structure and measurement.

2.1.1. Data Series and Data Frames

First, we study the core data structures used in pandas. Let's first try to work with single columns of data. A `Series` is a 1D labeled array capable of holding any data Python data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. To create a series, you run the following code:

```
series = pd.Series(data, index=index)
```

In the language of the previous section, data is an iterable set of measurements and index corresponds these measurements to individuals in the population (these have to be the same size!). Index values also have to be unique identifiers.

For example, suppose we wanted to associate the ids and salaries described in the previous section:

```
>>> salary = pd.Series([51428.00,74166.00,81200.00],\
                       index=['001','002','003'])
>>> salary
001    51428.0
002    74166.0
003    81200.0
dtype: float64
```

Individual elements of this series can be accessed by index value and the index itself can be accessed (which is a special object):

```
>>> salary['001']
51428.0
>>> salary.index
Index(['001', '002', '003'], dtype='object')
>>> list(salary.index)
['001', '002', '003']
```

Series objects also allow us to calculate basic population statistics:

```
>>> salary.mean()
68931.33333333333
>>> salary.median()
74166.0
>>> salary.max()
81200.0
>>> salary.min()
51428.0
```

Series objects also allow for batch numerical manipulation. Suppose, we wanted to convert all of the salary values from US dollars to British pounds. Running the following code does what we expect, returning a new series with the converted values:

```
>>> salary * 0.78
001    40113.84
002    57849.48
003    63336.00
dtype: float64
```

The expression $\times 0.78$ is applied to every element of the series. Similarly, element-wise operations can be applied to Series objects of the same size:

```
>>> salary - salary
001    0.0
002    0.0
003    0.0
dtype: float64
```

Perhaps, the most confusing aspect about Series objects (but actually the most powerful!) is slicing and masking. Slicing is straight-forward. Series objects can be treated like standard Python lists and arrays and can be "sliced" (take the first 10 elements, or middle 5 elements, etc.). For example, the following code returns just the first two elements of the Series (which itself is a new series):

```
>>> salary[0:2]
001    51428.0
002    74166.0
dtype: float64
```

Masking is a little more complicated. A mask is a Series object with only Boolean values. One create masks with standard Python comparison operators like:

```
>>> salary > 60000
001    False
002     True
003     True
dtype: bool
>>> salary == 51428.0
001     True
002    False
003    False
dtype: bool
```

These operations apply element-wise and return a new Series of Boolean values. As the name implies, masks can be used to "mask" out elements from another series of the same size. Most commonly we use masks to filter the series from which the mask was originally derived:

```
>>> salary[salary > 60000]
002    74166.0
003    81200.0
dtype: float64
```

Masks can be combined with bitwise logical connectives to create more complex filters:

```
>>> salary[( salary > 60000) & ( salary < 80000)]
002    74166.0
dtype: float64
>>> salary[( salary > 60000) | ( salary < 80000)]
001    51428.0
002    74166.0
003    81200.0
dtype: float64
```

We can think of tabular data as simply collection of named series objects (each column) with the same index. A DataFrame is an object that manages such series.

```
>>> name = pd.Series(['John␣Davies','Janie␣Williams','Phyllis␣Wang'],
                     index=['001','002','003'])
>>> salary = pd.Series([51428.00, 74166.00, 81200.00],
                     index=['001','002','003'])
>>> df = pd.DataFrame({'name': name, 'salary': salary})
>>> df
              name   salary
001     John Davies  51428.0
002  Janie Williams  74166.0
003    Phyllis Wang  81200.0
```

DataFrame objects not only have index values but they also have columns. Each of the columns can be pulled out as an individual series:

```
>>> df['name']
001       John Davies
002    Janie Williams
003      Phyllis Wang
Name: name, dtype: object
>>> df['salary']
001    51428.0
002    74166.0
003    81200.0
Name: salary, dtype: float64
```

Masking for DataFrame objects happens across all of its columns (Series objects). We can create a mask using the same technique as before and get both the name and salary for rows that satisfy the condition:

```
>>> df[df['salary'] > 60000]
               name   salary
002  Janie Williams  74166.0
003    Phyllis Wang  81200.0
```

It is also easy to add more Series objects to a DataFrame. Suppose, we wanted to include the salary in USD as well as GBP:

```
>>> df['salary_gbp'] = salary * 0.78
>>> df
               name   salary  salary_gbp
001     John Davies  51428.0    40113.84
002  Janie Williams  74166.0    57849.48
003    Phyllis Wang  81200.0    63336.00
```

2.1.2. Data Loading

A key aspect of the Pandas library is the ease with which data can be shared and imported from files. We will focus on the most popular format; called "delimited" files. Such formats store two-dimensional arrays of data by separating the values in each row with specific delimiter characters (like commas or tabs). The most popular choice is using Comma-Separated Values (or CSV) files. Due to their wide support, CSV files can be used in data exchange among many applications. A delimited text file is a text file used to store data, in which each line represents a single book, company, or other thing, and each line has

fields separated by the delimiter. Consider the following example of a "comma" delimited file:

```
First, Last, SSN, Date of Birth
Bob, Davis, 123-45-2312, 1991-02-09
```

One challenge with delimited files is when the delimiter appears in one of the data fields. For example,

```
First, Last, SSN, Date of Birth, Comments
Bob, Davis, 123-45-2312, 1991-02-09, Bob is a great worker, but he lacks focus
```

To address such issues, we usually define an "enclosure" character which encloses whole fields when the delimiter might be in the field. Typically, for CSV files, that is a ":

```
First, Last, SSN, Date of Birth, Comments
Bob, Davis, 123-45-2312, 1991-02-09, "Bob is a great worker, but he lacks focus"
```

What happens when we have to use the enclosure character in one of the fields as well? Typically, we define an "escape" character as well that can short circuit the parsing when a special character needs to be used:

```
First, Last, SSN, Date of Birth, Comments
Bob, Davis, 123-45-2312, 1991-02-09, "Bob is a great \"worker\", but he lacks focus"
```

Pandas conveniently allows us to load and save DataFrames from and to CSV files:

```
>>> df = pd.read_csv(filename)
>>> df.to_csv(filename)
```

We encourage you to read the documentation to understand how to use the different options when reading and writing CSV files.

Let's consider a real example dataset to illustrate how to use this API. The Chicago Park District maintains a dataset describing all of the city's park facilities, their type, and location. The file can be downloaded from [1].

[1] https://dev.socrata.com/foundry/data.cityofchicago.org/eix4-gf83

```
>>> df = pd.read_csv('CPD_Facilities.csv')
```

The resulting DataFrame has 4383 rows and 9 columns.

2.2. Grouping and Partitioning Data

A key task in data science is to compare different sub-populations of data. For example, we may want to compare employees from different branches or we may want to compare voters from different regions. Partitioning a dataset on groupable attributes is a core task in the Pandas library. After partitioning, we often compare the populations using some statistic.

As an example dataset, let us consider the openly available Titanic survival dataset. On April 15, 1912, during her maiden voyage, the widely considered "unsinkable" RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren't enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew. We would like to compare the mortality rates between different populations of passengers. The data is available at [2].

We can load this data using Pandas:

```
>>> df = pd.read_csv('titanic.csv')
```

This command loads a dataset with 891 rows and 12 columns. The first column that matters to us is the "survived" column. This is a binary measurement that indicates whether the passenger survived (1) or not (0).

```
>>> df['Survived']
0       0
1       1
2       1
3       1
4       0
       ..
```

[2] We used the training set in this section: https://www.kaggle.com/c/titanic/overview

```
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

The overall mortality rate is defined as one minus the survival rate, which can be calculated as:

```
>>> 1-df['Survived'].mean()
0.6161616161616161
```

This means that nearly 2/3 of passengers who set foot on the ship died!

We would like to understand how different sub-populations fared. In particular, there are two attributes of interest "Sex" and "Pclass' (the class of the ticket purchased by the passenger).

```
>>> df['Sex']
0        male
1      female
        ...
889      male
890      male
Name: Sex, Length: 891, dtype: object
>>> df['Pclass']
0      3
1      1
      ..
889    1
890    3
```

Let's first subdivide the data on Sex. Pandas uses the "groupby" syntax to do so:

```
>>> df.groupby('Sex')
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x112a89c50>
```

On it's own this returns something that's not useful, but we can turn this into a DataFrame by combining it with a statistic. For example, if we calculate the `mean()`, we get:

```
>>> df.groupby('Sex').mean()
        PassengerId  Survived    Pclass  ...     SibSp     Parch        Fare
Sex                                      ...
female   431.028662  0.742038  2.159236  ...  0.694268  0.649682  44.479818
male     454.147314  0.188908  2.389948  ...  0.429809  0.235702  25.523893
```

This code divides the data into two populations based on Sex and calculates the mean along each other column. If we want the mean along a specific column, we can use the following syntax:

```
>>> df.groupby('Sex')['Survived'].mean()
Sex
female    0.742038
male      0.188908
Name: Survived, dtype: float64
```

This initial result shows that the survival rate for women was much higher than the survival rate for men.

We may be worried that the population of women is much smaller and hence the elevated survival rate. To also report the sizes of the population, we can use the more general `agg()` function instead of `mean()`:

```
>>> df.groupby('Sex')['Survived'].agg(['count', 'mean'])
        count      mean
Sex
female    314  0.742038
male      577  0.188908
```

We can also subdivide the data on multiple columns, e.g., Sex and Class.

```
>>> df.groupby(['Sex', 'Pclass'])['Survived'].agg(['count', 'mean'])
               count      mean
Sex    Pclass
female 1          94  0.968085
       2          76  0.921053
       3         144  0.500000
male   1         122  0.368852
       2         108  0.157407
       3         347  0.135447
```

These results show that women in the 1st and 2nd class cabins had a mortality rate of less than 10%.

2.3. Combining Data Frames

The last piece of the Pandas API that we will cover are the functions that combine multiple DataFrame together. Before we go into code, let's start with some conceptual examples of how we may want to combine tabular data.

2.3.1. Horizontal Combination

The first way that we may want to combine data is called a *horizontal* combination, where we take one or more datasets that measure roughly similar properties and combine their rows. Conceptually, a horizontal combination of data is a generalization of a set union. Let $D_1, ..., D_k$ define sets of rows that correspond to individuals from a population. A horizontal combination is a new dataset that has all of individuals:

$$D = \bigcup_{i=1}^{k} D_i$$

While conceptually simple, programming reality is a little more challenging. To be able to combine multiple datasets together they have to have the same columns and we need to decide what to do when that is not the case. Let's go back to our example employee dataset, but now restricted to 4 employees: Now

id	Name	Position	Branch	Salary
001	John Davies	Technician I	Akron	51428.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00
004	Ivan Berenson	Technician II	Traverse City	44211.00

suppose, our company added a new branch (in an example city "Davenport") through an acquisition and got a new dataset of employees: We would like to

id	Name	Position	Salary
A	Kelly Tomlinson	Manager	105149.00
B	Barry Mantle	Associate	79992.00
C	Rufus Smith	Part Time	10200.00
D	Shauna Taylor	Technician I	24101.00

integrate this dataset into the original one to construct a single master employee record. This integration process is not a simple concatenation: (1) The new data does not contain a "branch" column because all of the data are from the same branch, and (2) the id's in the new dataset are of a different format than the current master dataset.

Let's first fix the easy problem–which is adding a new branch column to a dataframe. Let's assume we have two dataframes **`master`** (original dataset) and **`new`** (new branch). We can run the following code to add a column where all of the values are 'Davenport'.

```
>>> new['Branch'] = 'Davenport'
```

Next, we would like to update the id values to be consistent with the original dataset.

```
>>> new['id'] = ['00' + str(num) for num in range(5,9)]
```

Now, the two datasets have the same set of columns and value formats so we can combine them with:

```
>>> df = pd.concat([master, new])
```

2.3.2. Vertical Combination

Sometimes, we may want to enhance one dataset with columns from another. This happens when we have mutliple types of measurements about the same individuals in a population or we have external data that we want to link into our analysis. We will first present a series of examples of "vertical combination" or adding columns to a dataset, and then summarize these different operations into a single programming framework.

2.3.2.1. Vertical Concatenation

The simplest example is what we call vertical concatenation (also called zip in some frameworks). Suppose, we have two DataFrames with rows indexed by exactly the same property—there is a one-to-one correspondence between rows in both DataFrames. For example, we could have one table of salary records:

id	Name	Position	Branch	Salary
001	John Davies	Technician I	Akron	51428.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00

and another table of emergency contact information:

id	ContactName	ContactRel
001	Susanne Davies	Spouse
002	Trent Williams	Sibling
003	Martin Nash	Other

The desired output is:

id	Name	Position	Branch	Salary	ContactName	ContactRel
001	John Davies	Technician I	Akron	51428.00	Susanne Davies	Spouse
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	Trent Williams	Sibling
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	Martin Nash	Other

Since each column of either dataset is a series indexed by the same attribute (in this case employee id), we can combine the datasets using the structure we've seen before:

```
>>> master['ContactName'] = emergency['ContactName']
>>> master['ContactRel'] = emergency['ContactRel']
```

Let's think about why this example is particularly easy. Each row in `master` and `emergency` refer to the same individual. There is a one-to-one relationship

between rows.

The only complication that can arise is if some individuals are contained in one of the datasets but not the other. One must be really careful about these cases because pandas is a little inconsistent with its default behavior when this happens. Consider the following example, where we combine two series that do not completely align on their indexes. One has an index of $[0, 1]$ and another has an index of $[1, 2]$.

```
>>> s1 = pd.Series(['a','b'], index=[0,1])
>>> df = pd.DataFrame({'s1':s1})
  s1
0  a
1  b
>>> s2 = pd.Series([True,False], index=[1,2])
>>> df['s2'] = s2
```

What does the output look like?

```
>>> df
  s1    s2
0  a   NaN
1  b  True
```

The output looks like this because we are trying to add a new series to a DataFrame that already has some indexes. Vertical concatenation in pandas does not change the indexing of existing DataFrames (so the row corresponding to $index = 2$ is lost). For $index = 0$, pandas will put in a dummy NaN placeholder since that value doesn't exist.

2.3.2.2. Vertical Merging

Sometimes, we may want to link dataset that don't necessarily correspond to the same population of data. This is best seen with an example. Let's go back to the employees and salaries data:

Suppose, we had a new dataset that listed the median salaries for each of the cities the branches were located; perhaps for cost-of-living adjustments.

id	Name	Position	Branch	Salary
001	John Davies	Technician I	Akron	51428.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00

Branch	Median Salary
Akron	45331.00
Oswego	61499.00
Nashua	56771.00

A vertical merge augments a dataset by matching attributes across datasets that don't necessarily correspond to the same population. For example, we may want to return, which adds the median salary of the city in which the branch is located to each row:

id	Name	Position	Branch	Salary	Median Salary
001	John Davies	Technician I	Akron	51428.00	45331.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	61499.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	61499.00

One can perform a merge in using the following code:

```
>>> df = master.merge(cities)
```

By default merge will match on columns with exactly the same name. This default behavior can be modified as well as what to do if a suitable match is not found (for example the Branch city is missing in the median salary dataset) [3].

Advanced merging strategies are more common than you would think! To understand, where such a problem might arise, consider the same employee salary dataset with an additional table that represents supervisor and subor-

[3] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html

dinate relationships.

id	Name	Position	Branch	Salary
001	John Davies	Technician I	Akron	51428.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00

the above table is linked with the following table that describes the org-chart relationships:

id	reports_to
001	003
002	003
003	NaN

Both John and Janie report to Phyllis, who does not report to anyone else. Let's consider two different merges on this dataset.

First, let's do the default merge in pandas, linking the two datasets on the id column (you could also use the vertical concatenation for this!!):

```
>>> df = master.merge(supervisors)
```

This code would create a new column specifying which other employee each employee reports to.

id	Name	Position	Branch	Salary	reports_to
001	John Davies	Technician I	Akron	51428.00	003
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	003
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	NaN

This code keeps the subordinates' names but only returns the ids for the supervisors. What if instead we did the following merge:

```
>>> df = master.merge(supervisors, left_on='id', right_on='reports_to')
```

id	Name	Position	Branch	Salary	reports_to
001	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	003
002	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	003

The resulting dataset would be:

This code does the opposite where it keeps the supervisors names but returns the ids of the subordinates. We might prefer one or the other depending on how we want to filter the dataset.

2.3.2.3. A General API

All of the vertical combination strategies described in this section are a special case of a "join" operation. Let's first define a special hypothetical combination of two datasets called the Cartesian Product, which is all pairs of rows from both. This is like the concept of a Cartesian product in math, where $\mathbf{R}$ denotes the real numbers and $\mathbf{R} \times \mathbf{R}$ denotes the Cartesian plane (all pairs of real numbers). For example, given:

id	Name	Position	Branch	Salary
001	John Davies	Technician I	Akron	51428.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00

Branch	MedSalary
Akron	45331.00
Oswego	61499.00
Nashua	56771.00

The Cartesian product is:

Intuitively, the Cartesian product describes all possible matchings of rows–

id	Name	Position	Branch_1	Salary	Branch_2	MedSalary
001	John Davies	Technician I	Akron	51428.00	Akron	45331.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	Akron	45331.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	Akron	45331.00
001	John Davies	Technician I	Akron	51428.00	Oswego	61499.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	Oswego	61499.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	Oswego	61499.00
001	John Davies	Technician I	Akron	51428.00	Nashua	56771.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	Nashua	56771.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	Nashua	56771.00

and all vertical combinations are subsets of the Cartesian product. A general "join" operation returns a subset of the Cartesian product. Every merge described previously can be written in this way. For example, all of the rows where $Branch\_1 = Branch\_2$:

id	Name	Position	Branch_1	Salary	Branch_2	MedSalary
001	John Davies	Technician I	Akron	51428.00	Akron	45331.00
002	Janie Williams	Asst. Supervisor I	Oswego	74166.00	Oswego	61499.00
003	Phyllis Wang	Asst. Supervisor II	Oswego	81200.00	Oswego	61499.00

Thus, just like how we defined horizontal combination as a union of datasets, we can really think of vertical combination as a Cartesian Product of datasets:

$$D = \bigotimes_{i=1}^{k} D_i$$