Data Manipulation/Pandas

https://www.youtube.com/playlist?list=PL-osiE80TeTsWmV9i9c58mdDCSskIFdDS

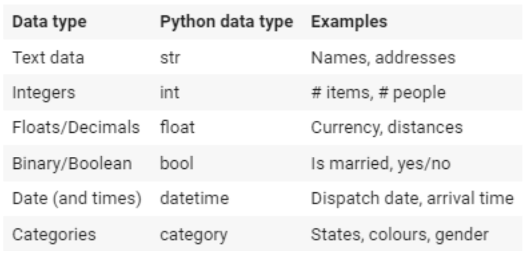
DATA CLEANING - happens at the same time as exploratory data analysis/missing values/visualizing data and statistical properties

EXPLORATORY DATA ANALYSIS (EDA) - an indepth analysis of data | happens at the same time as data cleaning/missing values/visualizing data and statistical properties

MISSING VALUES - happens at the same time as data cleaning/exploratory data analysis/visualizing data and statistical properties

STATISTICAL PROPERTIES - happens at the same time as data cleaning/exploratory data analysis/missing values and visualizing data

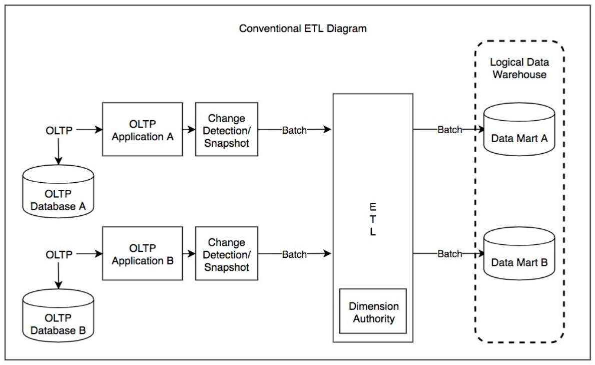
VISUALIZING DATA - happens at the same time as data cleaning/exploratory data analysis/missing values and statistical properties



Data type	Python data type	Examples
Text data	str	Names, addresses
Integers	int	# items, # people
Floats/Decimals	float	Currency, distances
Binary/Boolean	bool	Is married, yes/no
Date (and times)	datetime	Dispatch date, arrival time
Categories	category	States, colours, gender

Extract/Transform/Load (ETL)

https://en.wikipedia.org/wiki/Extract,_transform,_load



3 phase process on data

- Extract data
- Transform data (clean, sanitize, scrub)
- Load output to container

Data can be sourced and uploaded from multiple places

Normally done using apps but also system operators

ETL software normally automates the process and is run manually or on reoccurring schedules, either as single jobs or as batches

ETL systems enforce data type and data validity standards ensuring it conforms structurally to output requirements

Some can display in a presentation-ready format

Extract

Extracting data from multiple sources

Data extraction involves extracting data from homogeneous or heterogeneous sources; data transformation processes data by data cleaning and transforming it

Transform

A series of rules or functions are applied to extracted data in order to prepare it for loading into the end target

Load

Loads data to end target, ranging from a delimited flat file to a data warehouse. Some may overwrite existing information with cumulative information; is frequently done hourly/daily/weekly/monthly. More into a proper storage format/structure for the purposes of querying and analysis; finally, data loading describes the insertion of data into the final target database such as an operational data store, a data mart, data lake or a data warehouse.

ETL processing involves extracting the data from the source system(s). In many cases, this represents the most important aspect of ETL, since extracting data correctly sets the stage for the success of subsequent processes. Most data-warehousing projects combine data from different source systems. Each separate system may also use a different data organization and/or format. Common data-source formats include relational databases, XML, JSON and flat files, but may also include non-relational database structures such as Information Management System (IMS) or other data structures such as Virtual Storage Access Method (VSAM) or Indexed Sequential Access Method (ISAM), or even formats fetched from outside sources by means such as web spidering or screen-scraping. The streaming of the extracted data source and loading on-the-fly to the destination database is another way of performing ETL when no intermediate data storage is required.

An intrinsic part of the extraction involves data validation to confirm whether the data pulled from the sources has the correct/expected values in a given domain (such as a pattern/default or list of values). If the data fails the validation rules, it is rejected entirely or in part. The rejected data is ideally reported back to the source system for further analysis to identify and to rectify the incorrect records.

Data cleansing aims to pass only "proper" data to the target.

- Selecting only certain columns to load and removing null columns
- Translating coded values: the source system codes male as "1" but the warehouse codes male as "M"
- Encoding free-form values: mapping "Male" to "M"
- Deriving a new calculated value: sale_amount = qty * unit_price
- Sorting/ordering data based on a list of columns to improve search performance
- Joining data from multiple sources: lookup/merge/deduplicating
- Aggregating: rollup, summarizing multiple rows of data
- Generating surrogate-key values
- Transposing/Pivoting: turning multiple columns into multiple rows
- Splitting columns: converting a comma-separated-list into individual values in different columns
- Disaggregating repeating columns
- Looking up/validating relevant data: failed validation may result in a full rejection of the data, partial rejection, or no rejection

complex systems maintain a history/audit trail of changes. The constraints defined in the database schema and triggers apply uniqueness/referential integrity/mandatory fields, which contribute to data quality/performance

- A financial institution has information on a customer in multiple departments each has the information listed differently. One lists by name, whereas another lists by number. ETL can bundle/consolidate into a uniform presentation
- Permanently moving information to a new application that uses another database vendor schema.
- An Expense and Cost Recovery System (ECRS) used by accountancies/consultancies/legal firms. Data ends up in the time and billing system, some businesses may utilize raw data for employee productivity reports to Human Resources or equipment usage reports to Facilities Management.

The typical real-life ETL cycle consists of the following execution steps:

1. Cycle initiation
2. Build reference data
3. Extract from sources
4. Validate
5. Transform/clean/apply business rules/check for data integrity/create aggregates or disaggregates
6. Stage (load into tables)

7. Audit reports compliance with business rules/helps to diagnose/repair
8. Publish (to target tables)
9. Archive

ETL can involve considerable complexity, and significant operational problems can occur. Range of data values/data quality may exceed expectations of designers when validation and transformation rules are specified. Data profiling during analysis identifies conditions to be managed by transform rules, amendmening validation rules.

Design analysis establishes scalability across lifetime, including understanding volumes of data that must be processed within service legal agreements. Time available to extract data may change, meaning the same amount of data has to be processed in less time. Some systems have to scale to process terabytes of data. Increasing volumes of data may require designs that can scale from daily batch to multiple-day micro batch to integration with message queues or real-time change-data-capture.

Vendors benchmark record systems at multiple TB (terabytes) per hour (or ~1 GB per second) using powerful servers with multiple CPUs, multiple hard drives, multiple gigabit-network connections, and memory.

The slowest part of an ETL process occurs in initial loading. Databases perform slowly because they have to care for concurrency/integrity maintenance/indices. For better performance employ:

- *Direct path extract* method or bulk unload whenever is possible to reduce load on source
- Most transformation processing outside of the database

Bulk load operations. Access is usually the bottleneck

- Partition tables: keep partitions similar size
- Do validation in the ETL layer before the load: disable integrity checking in the target database tables during the load
- Disable triggers in target database tables during the load: simulate their effect as a separate step
- Generate IDs in the ETL layer not in the database)
- Drop indices before load, recreate them after load
- Use parallel bulk load: works when the table is partitioned or with no indices
- Attempting to parallel load into the same table (partition) causes locks
- Requirements to do insertions/updates/deletions: which rows should be processed in which way in the ETL layer, then process these separately; do bulk load for inserts, updates/deletes go through an API using SQL

Removing duplicates using `distinct` may be slow in database; thus do it outside. If using `distinct` significantly (x100) decreases the number of rows to be extracted, then remove duplications early before unloading.

A source of problems is a big number of dependencies among ETL jobs. For example, job "B" cannot start while job "A" is not finished. Achieve better performance by visualizing all processes on a graph, try to reduce maximising use of parallelism, and make "chains" of consecutive processing short.

Issues occur when data are spread among several databases, and processing is done sequentially. Database replication to copy data between databases can significantly slow down the process. The solution is reduce the processing graph to only three layers:

- Sources
- Central ETL layer
- Targets

This allows processing to take maximum advantage of parallelism. If you need to load data into two databases, you can run the loads in parallel.

Sometimes processing takes place sequentially. Dimensional data are needed before one can retrieve/validate rows for main fact tables.

Parallel processing is a recent development in ETL improving overall performance for handling large data sets**.**

There are 3 types of parallelism operating at once:

- **Data**: splitting one file into smaller files for parallel access

- **Pipeline**: simultaneously running multiple components on one data stream
- **Component**: simultaneously running multiple processes on multiple data streams in the same job

There can be difficulty ensuring consistency in uploaded data as source databases have different update cycles. The system could be required to withhold data until everything synchronizes. Same with warehouses so enforcing synchronization/reconciliation points is necessary

Data warehousing procedures usually subdivide a big ETL process into smaller pieces running sequentially or in parallel. To keep track of data flows, it makes sense to tag each data row with "row_id", and tag each piece of the process with "run_id". In case of a failure, having these IDs help to roll back and rerun the failed piece.

Best practice also calls for *checkpoints*, which are states when certain phases of the process are completed. Once at a checkpoint, it is a good idea to write everything to disk, clean out some temporary files, log the state, etc.

As of 2010, data virtualization had begun to advance ETL processing. The application of data virtualization to ETL allowed solving the most common ETL tasks of data migration and application integration for multiple dispersed data sources. Virtual ETL operates with the abstracted representation of the objects or entities gathered from the variety of relational, semi-structured, and unstructured data sources. ETL tools can leverage object-oriented modeling and work with entities' representations persistently stored in a centrally located hub-and-spoke architecture. Such a collection that contains representations of the entities or objects gathered from the data sources for ETL processing is called a metadata repository and it can reside in memory[7] or be made persistent. By using a persistent metadata repository, ETL tools can transition from one-time projects to persistent middleware, performing data harmonization and data profiling consistently and in near-real time.[8]

Unique keys play an important part in all relational databases, as they tie everything together. A unique key is a column that identifies a given entity, whereas a foreign key is a column in another table that refers to a primary key. Keys can comprise several columns, in which case they are composite keys. In many cases, the primary key is an auto-generated integer that has no meaning for the business entity being represented, but solely exists for the purpose of the relational database - commonly referred to as a surrogate key.

As there is usually more than one data source getting loaded into the warehouse, the keys are an important concern to be addressed. For example: customers might be represented in several data sources, with their Social Security Number as the primary key in one source, their phone number in another, and a surrogate in the third. Yet a data warehouse may require the consolidation of all the customer information into one dimension.

A recommended way to deal with the concern involves adding a warehouse surrogate key, which is used as a foreign key from the fact table.

Usually, updates occur to a dimension's source data, which obviously must be reflected in the data warehouse.

If the primary key of the source data is required for reporting, the dimension already contains that piece of information for each row. If the source data uses a surrogate key, the warehouse must keep track of it even though it is never used in queries or reports; it is done by creating a lookup table that contains the warehouse surrogate key and the originating key.[10] This way, the dimension is not polluted with surrogates from various source systems, while the ability to update is preserved.

The lookup table is used in different ways depending on the nature of the source data. There are 5 types to consider;[10] three are included here:

Type 1The dimension row is simply updated to match the current state of the source system; the warehouse does not capture history; the lookup table is used to identify the dimension row to update or overwrite

Type 2A new dimension row is added with the new state of the source system; a new surrogate key is assigned; source key is no longer unique in the lookup table

Fully loggedA new dimension row is added with the new state of the source system, while the previous dimension row is updated to reflect it is no longer active and time of deactivation.

An established ETL framework may improve connectivity and scalability.[*citation needed*] A good ETL tool must be able to communicate with the many different relational databases and read the various file formats used throughout an organization. ETL tools have started to migrate into Enterprise Application Integration, or even Enterprise Service Bus, systems that now cover much more than just the extraction, transformation, and loading of data. Many ETL vendors now have data profiling, data quality, and metadata capabilities. A common use case for ETL tools include converting CSV files to formats readable by relational databases. A typical translation of millions of records is facilitated by ETL tools that enable users to input csv-like data feeds/files and import them into a database with as little code as possible.

ETL tools are typically used by a broad range of professionals — from students in computer science looking to quickly import large data sets to database architects in charge of company account management, ETL tools have become a convenient tool that can be relied on to get maximum performance. ETL tools in most cases contain a GUI that helps users conveniently transform data, using a visual data mapper, as opposed to writing large programs to parse files and modify data types.

While ETL tools have traditionally been for developers and IT staff, research firm Gartner wrote that the new trend is to provide these capabilities to business users so they can themselves create connections and data integrations when needed, rather than going to the IT staff.[11] Gartner refers to these non-technical users as Citizen Integrators.[12]

Extract, load, transform (ELT) is a variant of ETL where the extracted data is loaded into the target system first.[13] The architecture for the analytics pipeline shall also consider where to cleanse and enrich data[13] as well as how to conform dimensions.[1]

Ralph Kimball and Joe Caserta's book The Data Warehouse ETL Toolkit, (Wiley, 2004), which is used as a textbook for courses teaching ETL processes in data warehousing, addressed this issue.[14]

Cloud-based data warehouses like Amazon Redshift, Google BigQuery, and Snowflake Computing have been able to provide highly scalable computing power. This lets businesses forgo preload transformations and replicate raw data into their data warehouses, where it can transform them as needed using SQL.

After having used ELT, data may be processed further and stored in a data mart.[15]

There are pros and cons to each approach.[16] Most data integration tools skew towards ETL, while ELT is popular in database and data warehouse appliances. Similarly, it is possible to perform TEL (Transform, Extract, Load) where data is first transformed on a blockchain (as a way of recording changes to data, e.g., token burning) before extracting and loading into another data store.[17]

ELT vs Data Prep

ETL (extract, transform and load) has for years been a workhorse technology for enabling analysis of business information. But now it's being joined by a new approach, called data preparation or data wrangling. The two techniques are similar in purpose, but distinct in function and application. Let's take a look a how the two compare.

Both data wrangling and ETL address the same issue: how to convert data that comes from different sources and in a variety of formats into a common structure for analysis and reporting.

ETL does it, as the name implies, by extracting data from various sources, transforming it into a predefined format, then loading it into a data lake or warehouse where it can be accessed by business intelligence or report generator applications. A key distinctive of ETL procedures and tools is that they are designed as elements of a formal workflow intended for use by IT professionals. Because the organization's ability to access and use its most critical data depends on the reliable functioning of the ETL system, there is typically a focus on highly structured policies and procedures, along with active monitoring by skilled workers.

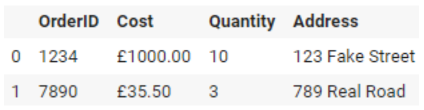
Data wrangling, on the other hand, is intended for use not by IT professionals, but by end users, such as executives, business analysts, and line managers, who are seeking answers to specific questions. The idea is that the people who are closest to the data, and therefore understand it best, should be the ones extracting and analyzing it. Historically, these workers often performed their tasks manually using informal processes and employing general-purpose tools such as spreadsheets. Today, data wrangling is seen as fulfilling a critical need in making data more widely accessible in an organization. The title of Self-Serve Data Preparation has now dignified it, and automated tools to facilitate its use by non-IT professionals are beginning to appear.

With its formal processes for accessing, transporting, vetting and cleansing data, ETL is best suited for handling well-structured data from sources such as databases and ERP, SAS, or CRM applications. Because ETL systems are normally implemented and managed by highly skilled professionals, they reliably produce data of the highest quality. On the other hand, the very structural rigidity that guarantees data quality can significantly reduce an ETL system’s ability to service new analytics use cases in a timely fashion.

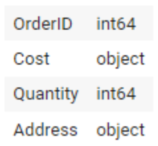
In contrast, the strength of the data wrangling technique is its agility and flexibility. While an ETL system will often be the key component in a company’s data warehouse production environment, data wrangling shines at allowing individual users to get the answers they need quickly. For example, a small team of analysts could employ data wrangling to rapidly evaluate a series of “what-if” scenarios by extracting and analyzing various combinations of data from disparate sources. In such situations, absolute precision may be less important than the ability to access “ballpark” information quickly.

In today’s corporate environment, where the amount of useful data increases daily, combining ETL with self-serve data preparation can help companies maximize the value of the information available to them. And tools to do it are already available. For example, Precisely Connect allows users to create their own data blends without manual coding or tuning.

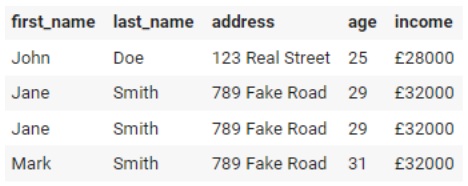
Exploratory Data Analysis



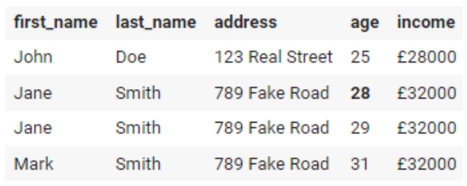
	OrderID	Cost	Quantity	Address
0	1234	£1000.00	10	123 Fake Street
1	7890	£35.50	3	789 Real Road



OrderID	int64
Cost	object
Quantity	int64
Address	object



first_name	last_name	address	age	income
John	Doe	123 Real Street	25	£28000
Jane	Smith	789 Fake Road	29	£32000
Jane	Smith	789 Fake Road	29	£32000
Mark	Smith	789 Fake Road	31	£32000



first_name	last_name	address	age	income
John	Doe	123 Real Street	25	£28000
Jane	Smith	789 Fake Road	**28**	£32000
Jane	Smith	789 Fake Road	29	£32000
Mark	Smith	789 Fake Road	31	£32000

Cost should be a float - but it has a £ symbol attached to it. To use this column as a float, we need to remove the £ sign

Memory usage has dropped by around 8mb by just converting one column to a category

Pandas represents categories as integer types. Other datasets explicitly see a category column encoded as integers

```
# Check the data type
print(flights_df.dtypes)
# Get more info about a column
first5['AIRLINECODE'].describe()
# Look at memory usage
flights_df.info()

# Get the sum of costs
sales['cost'].sum()

# Remove the £ sign
sales['cost'] = sales['cost'].str.strip('£')
# Convert the float
sales['cost'] = sales['cost'].astype('float64')
```

Duplicates

```
# Detects duplicates
first5.duplicated()
# Returns the number of duplicates
flights_df.duplicated().sum()

# Add duplicates to a set
duplicates = {'ORIGAIRPORTNAME': first5.duplicated('ORIGAIRPORTNAME').sum(),
                'DESTAIRPORTNAME': first5.duplicated('DESTAIRPORTNAME').sum()}
# Detects duplicates
first5.duplicated(subset=['DESTAIRPORTNAME'])
# Returns the number of duplicates
```

Duplicate values occur when we have the same values repeated across multiple rows or columns. Duplicate data arises from either bugs/design patterns in data pipelines, or most commonly, due to database joins and data consolidation from various datasets/databases, which may retain the duplicate values

We might see extremely similar entries. This type of duplicate error is most likely due to a data entry issue or a resubmission

```
print('DUPLICATES: ', duplicates)
# Returns the actual duplicates
print(first5[duplicates])

# Sort by column to manually review duplicates
first5.sort_values(by='ORIGAIRPORTNAME')

# Put all means columns in a dictionary
summaries = {"CRSARRTIME": "mean", "ARRTIME": "mean", "ARRDELAY": "mean", "CRSELAPSEDTIME": "mean", "ACTUALELAPSEDTIME": "mean"}

# Split into duplicates and organise by column name
grouped_duplicates = flights_df[duplicates].groupby(["FLIGHTDATE", "AIRLINECODE", "ORIGAIRPORTNAME", "DESTAIRPORTNAME"])
# Return the minimum and reset index to default
grouped_duplicates_min_transactionid = grouped_duplicates["TRANSACTIONID"].min().reset_index()

# Merge
f_df_duplicates = pd.merge(
    grouped_duplicates_min_transactionid, # Merge this
    grouped_duplicates.agg(summaries).reset_index(), # With this
    how="inner" # by preserving the order of the left keys
).sort_values("TRANSACTIONID") # And sort by this column

f_df_duplicates
```

of whatever form Jane had submitted - which was entered into the database without removing her old entry

Note that we can `.sum()` over boolean values. Basically, False is interpreted as 0s and True's as 1. So by summing over the dataframe, we can get the total number of duplicate values!

The second Jane Smith example. The duplicated method would not have returned true because the whole row wasn't an exact duplicate. To tackle this issue, the `.duplicated()` method takes in two arguments: `subset` and `keep`. For the subset argument, provide a list of column names we want to check duplicates over, and the keep argument takes on 1 of 3 values: `first`, `last` or `False`

- `first` : Mark duplicates as True except for the first occurrence.
- `last` : Mark duplicates as True except for the last occurrence.
- `False` : Mark all duplicates as True.

Using `df[duplicates]` we are returned the data points where duplicates exist

So.. how do we deal with duplicate values? Well, we only need one entry - not both - so we have one of two options:

Average over duplicate values on datatypes that make sense

By grouping on relevant columns `.groupby()` and chaining this with the `.agg()` function. Our argument to `.agg()` is a dictionary with key value pairs of column names and the aggregation function we want to apply over them (e.g. sum, difference, mean etc)

Drop one of the duplicated rows (or many in the case of one entry having multiple duplicates)

Example

https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

Getting Started

```
import pandas as pd

# Load the data file
df = pd.read_csv('survey_results_public.csv')
# Load the file that describes what the columns mean
df_schema = pd.read_csv('survey_results_schema.csv')

# Display data frame (first 20 columns)
df
# Display first(head) or last (tail) 10 rows
df.head(10)
df.tail(10)

# An attribute that returns number of columns and rows
# (85 columns and 80,000+ rows)
df.shape
# Returns the datatypes of the columns
# An object is usually a string
df.info()

# Change options to display all columns
# and all rows in the schema
pd.set_option('display.max_columns', 85)
pd.set_option('display.max_rows', 85)
```

Basics

```
# Access a specific column, returns a series
df.['Hobbyist']
# A 1 dimensional array / 'a single column of rows'
df(df['Hobbyist'])
# Does the same thing but can cause side effects
df.email

# Access multiple columns by giving it a list, no longer a series
df[['Employment', 'Hobbyist']]
# Access all columns
df.columns

# Access rows by integer location (iloc)
df.iloc[0] # First row
df.iloc[[0, 1]] # First and second row
df.iloc[[0, 1], 2] # Fist 2 rows of column 2

# Access rows by lable location
df.loc[0] # First row
df.loc[0, 1, 2] # First 3 rowa
df.loc[0, 'Hobbyist'] # Single row for the Hobbyist column
df.loc[[0, 1], 'Hobbyist'] # Fist 2 rows of column Hobbyist
df.loc[[0, 1], ['Hobbyist', 'last']] # Fist 2 rows of column Hobbyist and last
df.loc[0:2, 'Hobbyist':'Employment'] # Slice first 3 rows of the slice Hobbyist to Employm
```

Indexes

```
# Modify the index, doesnt change the actua
df.set_index('Hobbyist')
# Perminatly modifies the df
df.set_index('Hobbyist', inplace=True)
# now we can search by element
df.loc['millie.jackson1@gmail.com']
df.loc[0] # This no longer works
# Reset to default index
df.reset_index(inplace=True)

# Set the index when you open the file
df = pd.read_csv('survey_results_public.csv
df.loc[1] # First respondent
df_schema = pd.read_csv('survey_results_sch
df_schema.loc['Hobbyist'] # Look up the Hob
df_schema.loc['MgrIdiot', 'QuestionText'] #

# Sort alphabetically
df_schema.sort_index()
df_schema.sort_index(ascending=False) # Bac
df_schema.sort_index(ascending=False, inpla
```

Updating Rows and Columns

Filtering

```
# Returns the number of entries in the last column
filt = df['Hobbyist'] == 'Yes'
# Show them all
df[filt]
df.loc[filt]
# This works too
df[df['Hobbyist'] == 'Yes']

# Locate a single item
# & = +
filt = df['Hobbyist'] == 'Yes' & df['first'] == 'John'
df.loc[filt]
# Locate this item or this item
# | = or
filt = df['Hobbyist'] == 'Yes' | df['first'] == 'John'
df.loc[filt]
# Locate everything that isnt
df.loc[~filt]

# Make a filter to find all the salaries above 70000
high_salary = df['ConvertedComp'] > 70000
df.loc[high_salary]

# Apply filter to these columns
df.loc[high_salary, ['Country', 'LanguageWorkedWith', 'ConvertedComp']]

# Use a variable to hold a list of items
countries = ['United States', 'India', 'United Kingdom', 'Germany', 'Canada']
filt = df['Country'].isin(countries)
df.loc[filt, 'Country']

# Search the languages for a specific language, 'does the string contain 'Python'
filt = df['LanguagesWorkedWith'].str.contains('Python', na=False) # control for na edge cases
df.loc[filt, 'LanguagesWorkedWith']
```

Sorting

```
# Sort 2 columns alphabeticly backwards
df.sort_values(by=['last', 'first'], ascending=False)
# Forwards first then backwards
df.sort_values(by=['last', 'first'], ascending=[False, True], inplace=True)
# Return to the origional
df.sort_index()
# Show sorted last names only
df['last'].sort_values

# Examples
df.sort_values(by=['Country', 'ConvertedComp'], ascending=[True, False], inplace=True)
df[['Country', 'ConvertedComp']].head(50)
df['ConvertedComp'].nlargest(10) # Get the 10 largest salarys
df.nlargest(10, 'ConvertedComp') # Gets the details for those 10 largest salarys
```

What % Of People From Each Country Know Python?

```
# How many people from each country
country_respondents = df['Country'].value_counts()
# How many from each country use python
country_uses_python = country_grp['LanguageWorkedWith'].apply(lambda x: x. str.contains('Python').sum())
# Merge both together
python_df = pd.concat([country_respondents, country_uses_python], axis='columns', sort=False)
# Rename to more suitable column names
python_df.rename(columns={'Country': 'NumRespondents', 'LanguageWorkedWith': 'NumKnowsPython'}, inplace=True)
# Make a df of percentage of people who know python by country
python_df['PctKnowsPython'] = (python_df['NumKnowsPython']/python_df['NumRespondents']) * 100
# Sort from highest to lowest percentage
python_df.sort_values(by='PctKnowsPython', ascending=False, inplace=True)
# Show first 50
python_df.head(50)
```

```
# Returns how many values of each row there is (71257 yes, 17626 no)
df['Hobbyist'].value_counts()
```

Add and Remove Rows and Columns

```
# Combine first name and last name columns
df['full_name'] = df['first'] + ' ' + df['last']
# Pass a list of which columns to remove
df.drop(columns=['first', 'last'], inplace=True)
# Split fullname into two columns
df[['first', 'last']] = df['full_name'].str.split(' ', expand=True)

# Add a single row
df.append({'first': 'Tony'}, ignore_index=True)

# Make a new df
people = {
    'first': ['Tony', 'Steve'],
    'last': ['Stark', 'Rogers'],
    'email': ['IronMan@avenge.com', 'Cap@avenge.com']
    }

df2 = pd.DataFrame(people)
# Add 2 df together
# An error hear means that you passed the information in the wrong order 'sort' ignors thi
df.append(df2, ignore_index=True, sort=False)
# Remove a row
df.drop(index=4)
# Remove all instances of last name Doe
df.drop(index=df[df['last'] == 'Doe'].index)
# Or
filt = df['last'] == 'Doe'
df.drop(index=df[filt].index)
```

Grouping And Aggregating

```
# Get the mean salary
df['ConvertedComp'].median()
# Get the mean on all numerical columns
df.median()
# Get different stats 'quick overview'
# Count = number of missing rows
df.describe()
df['ConvertedComp'].count()
# how many said 'yes' or 'no'
df['Hobbyist'].value_counts()

df['SocialMedia'].value_counts() # Get the counts for each answer
df['SocialMedia'].value_counts(normalize=True) # Percentatges
df['Country'].value_count() # How many from each country
country_grp = df.groupby(['Country']) # Returns an object
country_grp.get_group('United States') # Shows all results for a specific country
# Same as doing this
filt = df['Country'] == 'United States'
df.loc[filt]
# To see most popular social media by that country
df.loc[filt]['SocialMedia'].value_counts()
# To see most popular social media for all countries
country_grp['SocialMedia'].value.counts()
country_grp['SocialMedia'].value.counts().loc['India'] # Just for India

country_grp['ConvertedComp'].median() # Show maen salary
country_grp['ConvertedComp'].median().loc['Germany'] # Show maen salary in Germany
country_grp['ConvertedComp'].agg(['median', 'mean']) # Returns the mean and median for eac
country_grp['ConvertedComp'].agg(['median', 'mean']).loc['Canada'] # Returns the mean and
filt = df['Country'] == 'India' # Show info for India
df.loc[filt]['LanguageWorkedWith'].str.contains('Python') # Find all cases of Python use i
df.loc[filt]['LanguageWorkedWith'].str.contains('Python').sum() # Shows how many
country_grp['LanguageWorkedWith'].apply(lambda x: x. str.contains('Python').sum())
```

```
# Change all column names
df.columns = ['first_name', 'last_name', 'e
# Change all column names to uppercase
df.columns = [x.upper() for x in df.columns
# Change all spaces to underscores
df.columns = df.columns.str.replace('', '_'
# Use a dictoinary of names you want to cha
df.rename(columns={'first_name': 'first', '
# Rename items
df.loc[2] = ['John', 'Smith', 'JohnSmith@em
df.loc[2, ['last, 'email']] = ['Doe', 'John
df.loc[2, 'last'] = 'Smith'
df.at[2, 'last'] = 'Doe'"Data Cleaning Prac
# Apply a filter and change the name
filt = (df['email'] == 'JohnDoe@email.com')
df.loc[filt, 'last'] = 'Smith'

# Apply changes to multiple items
df.['email'] = df['email'].str.lower()

# See how many rows in the column
df.apply(len)
# See how many rows in a specific column
len(df['email'])
# See how many columns in a specific row
df.apply(len, axis='columns')
# See the character count for each email
df['email'].apply(len)

# Build a function to pass in as a peramete
def update_email(email):
    return email.upper()

df['email'] = df['email'].apply(update_emai

# Using lambda functions
df['email'] = df ['email'].apply(lambda x:

df.apply(pd.Series.min)
df.apply(lambda x: x.min())

# Only works on df not series
df.applymap(len)
df.applymap(str.lower)

# Only works on Series not df
# Used to substitute one value for another
# Values not changed will become 'NaN' not
df['first'].map({'Corey': 'Chris', 'Jane':
# This changes the one we want and keeps th
df['first'].replace({'Corey': 'Chris', 'Jan

# Examples
df.rename(columns={'ConvertedComp': 'Salary
df['Hobbyist'] = df['Hobbyist'].map({'Yes':
```

Dates And Times

```
# Run a datetime function to see if it is t
df.loc[0, 'Date'].day_name()
# Convert to datetime
df['Date'] = pd.to_datetime(df['Date'])
df['Date'] = pd.to_datetime(df['Date'], for
# You can do this when you read in the file
d_parser = lambda x: pd.datetime.strpttime(
df = pd.read_csv('survey_results_public.csv
# Get the dayname for all values
df['Date'].dt.day_name()
# Make it a new column
df['DayOfWeek'] = df['Date'].dt.day_name()
# Get the earlyest date
df['Date'].min()
# Get the time delta (time between two ever
df['Date'].max() - df['Date'].min()
```

```
# Show just Japan
python_df.loc['Japan']
```

File Formats

```
# CSV
from xmlrpc.server import XMLRPCDocGenerator
df = pd.read_csv('data/file.csv')
df.head()
df.to_csv('data/modified.csv')

# TSV
df = pd.read_csv('data/file.tsv', sep='\t')

# EXEL
pip install xlwt openpyxl xlrd
df.to_excel('data/modified.xlsx')
test = pd.read_excel('data/modified.xlsx', index_col='Respondent')

# JSON
test = pd.read_json('data/modified.json', orient='records', lines=True)
df_nice = pd.json_normalize(df["Employees"]) # Normalize so each column is each value
df.to_json('data/modified.json', orient='records', lines=True)

# SQL
pip install SQLAlchemy psycopg2-binary
from sqlalchemy import create_engine
import psycopg2
engine = create_engine('postgresql://dbuser:dbpass@localhost:5432/sample_db')
df.to_sql('sample_table', engine, if_exists='replace')
sql_df = pd.read_sql('sample_table', engine, index_col='Respondent')
sql_df.head()
sql_df = pd.read_sql('SELECT * FROM sample_table', engine, index_col='Respondent')

# URL
post_df = pd.read_json('https:etc')
post_df.head()

# XML
df = parse_XML("employees.xml", ["name", "email", "department", "age"])
df = pd.read_xml("employees.xml", attrs_only=True, elems_only=True)
df.to_xml("employees_df_export")
```

```
# Make a filter to narrow down the data
filt = (df['Date'] >= '2020')
filt = (df['Date'] >= pd.datetime('2019-01-
df.loc(filt)
# Make a slice
df.set_index('Date', inplace=True) # Make d
df['2019']
df['2020-01':'2020-02']['Close'].mean()

# View by day
df['2020-01':'2020-02'].head(24)
df['2020-01']['High'].max()
# Resample to 3 days
highs = df['High'].resample('3D').max()
highs['2020-01-01']
# Make a graph
%matplot inline
highs.plot()
# Resample whole df by week, using a dictio
df.resample('W').agg({'Close': 'mean', 'Hig
```

Data Cleaning

Examples

```
# Remove rows that dont have any data at all
df.dropnp()
df.dropnp(axis='index', how='any') # Default arguments index = rows any = any na
df.dropnp(axis='index', how='all') # Only removes ros when all values are na

# Checks email column for valid values in email and last name
# We need either an email or a last name
df.dropnp(axis='index', how='any', subset=['email', 'last'])

# Replace all missing values with np.nan
df.replace('NA', np.nan, inplace=True)
df.replace('missing', np.nan, inplace=True)
# Returns a mask of what it is including as na values
df.isna()

# Replace all na with a value
df.fillnp('MISSING')
df.fillnp('0')

# Shows data type of columns
df.dtypes
type(np.nan) # Retures a float
# Convert all values in the age column to a float
df['age'] = df['age'].astype(float)
# Get mean for that column
df['age'].mean()

# Variable for all na values
na_vals = ['NA', 'Missing']
```

```
# Use it at the begining when opening the file to replace all with NaN
df = pd.read_csv('survey_results_public.csv', index_col= 'Respondent', na_values=na_vals)

# Show first 10
df['YearsCode']
# Convert to float
df['YearsCode'] = df['YearsCode'].astype(float) # Wont work
# Find all values that are unique
df['YearsCode'].unique()
# Replace all 'less that one years with '0'
df['YearsCode'].replace('Less that 1 year', 0, inplace=True)
df['YearsCode'].replace('More that 50 year', 51, inplace=True)
df['YearsCode'] = df['YearsCode'].astype(float) # Now it works
df['YearsCode'].mean()
```

Missing Values

`.info()` returns us the amount of null information in each column

Visualizing Data

Statistical Properties