Glossary

February 2, 2026

0.1 Glossary of Key Terms (While coding in Python)

0-based indexing is a way of assigning indices to elements in a sequential, ordered data structure starting from 0, i.e. where the first element of the sequence has index 0.

Boolean is a data type that can be True or False

CSV (file) is an acronym which stands for Comma-Separated Values file. CSV files store tabular data, either numbers, strings, or a combination of the two, in plain text with columns separated by a comma and rows by the carriage return character.

Database is an organized collection of data.

DataFrame is a two-dimensional labeled data structure with columns of (potentially) different type.

Data type is a particular kind of item that can be assigned to a variable, defined by by the values it can take, the programming language in use and the operations that can be performed on it. examples: int (integer), str (string), float, boolean, list

Float is a Python data type designed to store positive and negative decimal numbers by means of a floating point representation.

Function is a group of related statements that perform a specific task.

Integer is a Python data type designed to store positive and negative integer numbers.

Jupyter Notebook is an interactive computing environment that allows you to combine live code, equations, visualizations, and narrative text.

Library/Module is a set of functions and methods grouped together to perform some specific sort of tasks.

List is a Python data structure designed to contain sequences of integers, floats, strings and any combination of the previous. The sequence is ordered and indexed by integers, starting from 0. Elements of a list can be accessed by their index and can be modified.

Loop is a sequence of instructions that is continually repeated until a condition is satisfied.

Matplotlib is a widely used plotting library for creating static, interactive, and animated visualizations in Python.

Method is a function that is specific to a type of data, accessed via `.` and requires () to run, for example `df.sum()`

NaN is an acronym for Not-a-Number and represents that either a value is missing, or the calculation cannot output any meaningful result.

None is an object that represents no value.

NumPy is a library for numerical computing in Python, which provides support for arrays and matrices along with a collection of mathematical functions.

Pandas is a powerful Python library for data manipulation and analysis.

Seaborn is a Python visualization library based on matplotlib that provides a high-level interface for drawing attractive statistical graphics.

String is a Python data type designed to store sequences of characters.

Tuple is a Python data structure designed to contain sequences of integers, floats, strings and any combination of the previous. The sequence is ordered and indexed by integers, starting from 0. Elements of a tuple can be accessed by their index but cannot be modified.

Variable a named quantity that can store a value, a variable can store any type, but always one type for a given value.

0.2 Do I really need Python ?

Python, as a programming language, offers the flexibility to build custom solutions for data visualization, letting you tailor analyses to your specific data and research questions. You can use different options such as Seaborn or Altair (that the course focuses on), or some other options such as plotly module or others. While software like RAWGraphs or Tableau is great for generating visualizations quickly, Python enables you to go deeper into data manipulation, integration, and repeatable analysis.

Please see the below comparison of the tools that we are using for RD course and try to decide the most suitable one for you while exploring your data and data viz story that you are creating.

Feature/Aspect	Python	RAWGraphs	Tableau
Type	Programming language with data visualization libraries	Open-source web-based visualization tool	Commercial data visualization and business intelligence platform
Ease of Use	Requires programming knowledge; flexible and customizable	User-friendly; no coding required	Intuitive interface; minimal coding required
Customization	Highly customizable through various libraries (e.g., Matplotlib, Seaborn, Plotnine)	Limited to provided templates	Extensive customization through built-in features and third-party extensions
Interactivity	Requires additional libraries for interactivity (e.g., Plotly, Bokeh)	Limited interactivity	High interactivity with dashboards and real-time data updates
Use Case	Suitable for complex data analysis and automation tasks	Ideal for quick, static visualizations	Best for interactive dashboards and business intelligence applications

0.3 Function Glossary of Pandas

1. `read_csv()`

Reads a CSV file into a DataFrame. - Use: Import data stored in CSV format into pandas.
- Example: `data = pd.read_csv('filename.csv')`

2. `DataFrame()`

The primary data structure in pandas that organizes data in rows and columns.

- Use: Create a DataFrame from dictionaries, lists, or other data structures.
- Example: `df = pd.DataFrame({'Column1': [1, 2, 3], 'Column2': ['A', 'B', 'C']})`

3. `head()` and `tail()`

head() returns the first few rows, while tail() returns the last few rows of a DataFrame

- Use: Quickly inspect data to understand its structure or for debugging.
- Example: `df.head(5)` or `df.tail(5)`

4. `info()`

Displays a concise summary of a DataFrame, including data types and non-null counts.

- Use: Identify the structure of the data, check for missing values, and review data types.
- Example: `df.info()`

5. `describe()`

Generates descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution.

- Use: Quickly get an overview of numeric columns, such as mean, median, and standard deviation.
- Example: `df.describe()`

6. `loc[]` and `iloc[]`

`loc[]` provides label-based indexing, while `iloc[]` offers integer-based indexing for DataFrame rows and columns.

- Use: Select subsets of data using row and column labels or integer positions.
- Example: `df.loc[2, 'ColumnName'] or df.iloc[2, 0]`

7. `merge()`

Combines two or more DataFrames based on a common key or index.

- Use: Join datasets from different sources where common columns exist.

- Example: `merged_df = pd.merge(df1, df2, on='key_column')`

8. `groupby()`

Splits the data into groups based on some criteria, applies a function, and combines the result.
 - Use: Perform aggregate computations such as sums, counts, or averages on grouped data.
- Example: `grouped = df.groupby('Category')['Value'].mean()`

9. `pivot_table()`

Creates a spreadsheet-style pivot table as a DataFrame.

- Use: Summarize data with multi-dimensional grouping and aggregation.

- Example: `pivot = pd.pivot_table(df, values='Value', index='Category', columns='Type', aggfunc=np.sum)`

10. `sort_values()`

Sorts a DataFrame by one or more columns.
 - Use: Organize data in ascending or descending order based on a column's values.
- Example: `df_sorted = df.sort_values(by='ColumnName', ascending=False)`

11. `drop()`

Removes rows or columns from a DataFrame.

- Use: Clean or modify data structure by eliminating unnecessary or problematic data.

- Example: `df.drop('ColumnName', axis=1, inplace=True)`

12. `dropna()`

Drop missing values (NaN) in the DataFrame with a specified value or method.

- Use: Address missing data before analysis or visualization.

- Example: `df_filled = df.fillna(0)`

13. `apply()`

Applies a function along an axis of the DataFrame (either rows or columns).

- Use: Perform custom computations for each row or column, making it versatile for data transformations.

- Example: `df['NewColumn'] = df['Column'].apply(lambda x: x*2)`

14. `concat()`

Combining DataFrames either vertically (stacking) or horizontally (joining side-by-side).

- Use: Merge multiple DataFrames or add new rows to an existing DataFrame.

- Example: `combined_df = pd.concat([df1, df2], axis=0)`

15. `append()`

Append rows of other to the end of caller, returning a new object

Note: Deprecated since version 1.4.0: Use concat() instead. For further details see Deprecated DataFrame.append and Series.append

16. `drop_duplicates()`

Removes duplicate rows from a DataFrame.

- Use: Clean your data by ensuring only unique records are retained.

- Example: `df_unique = df.drop_duplicates()`

Please explore the above examples while you are working on your own data set. Good to have some errors to resolve or get some direckt results by applying your data to learn more about the basics of pandas module

0.4 Function Glossary of Seaborn

1. `set_theme()`

- Purpose: Configure Seaborn's visual theme and aesthetics globally.
- Use: Sets default styling (colors, fonts, backgrounds) for all subsequent plots.

2. `catplot()`
   - Purpose: Create categorical plots using a high-level interface.
   - Use: Generates various types of plots (box, bar, point, swarm, etc.) by specifying the kind parameter, ideal for comparing distributions across categories.

3. `boxplot()`
   - Purpose: Visualize data distribution through quartiles and potential outliers.
   - Use: Useful for comparing distributions between multiple groups.

4. `violinplot()`
   - Purpose: Display data distributions using kernel density estimation along with basic summary statistics.
   - Use: Offers a richer understanding of the data's distribution shape when comparing multiple groups.

5. `barplot()`
   - Purpose: Plot aggregate statistics for categorical data with error bars.
   - Use: Ideal for showing central tendency (mean, median, etc.) across groups.

6. `countplot()`
   - Purpose: Show the counts of observations in each categorical bin through bars.
   - Use: Best for quickly visualizing the frequency of categories in your data.

7. `scatterplot()`
   - Purpose: Generate scatter plots to display relationships between two continuous variables.
   - Use: Commonly used to visualize correlations or clusters in data.

8. `lineplot()`
   - Purpose: Create line plots to show trends over continuous variables (like time).

- Use: Ideal for visualizing changes or trends in sequential data.

9. `displot()`
   - Purpose: Visualize univariate distributions using histograms, KDE plots, or both.
   - Use: Flexible tool for exploring the statistical distribution of a single variable.

10. `kdeplot()`
    - Purpose: Provide a smoothed estimate of a variable's distribution using kernel density estimation.
    - Use: Often used to overlay on histograms for comparison or to visualize bivariate distributions.

11. `pairplot()`
    - Purpose: Create a grid of plots to examine pairwise relationships among multiple variables.
    - Use: Quickly visualizes relationships and correlations across several numerical features.

12. `heatmap()`
    - Purpose: Render data in a matrix form with color-coded values, often used for correlation matrices.
    - Use: Effective for visualizing relationships in a dataset or highlighting patterns in numerical data.

Please benefit from these functions to try on your data and read through with the `Intro2DataViz-Pandas-Seaborn.ipynb` example to see further examples. For other functionalities, check out the followings;

- Seaborn user-guide tutorials: https://seaborn.pydata.org/tutorial.html#user-guide-and-tutorial
- Seaborn function reference list: https://seaborn.pydata.org/api.html

0.5 Function Glossary of Plotnine

Plotnine is a Python package for data visualization, based on the grammar of graphics. It implements a wide range of plots—including barcharts, linegraphs, scatterplots, maps, and much more.

1. `ggplot()` • Purpose: Main function to initiate any visual image or create a new ggplot object
   - Use: Create your visualization first layer based on grammar of graphics

2. `qplot()` • Purpose: Quick plot creation
   - Use: Create your first quick viz bazed on parameter selection

3. `aes()` • Purpose: Create aesthetic mappings
   - Use: Allows to add mapping part for the second layer

4. `geom()` • Purpose: Base class of all Geoms
   - Use: Geometric objects (geoms) are responsible for the visual representation of data points. There are various specific types to consider based on the type of visualization.

5. `labs()` • Purpose: Add labels for any aesthetics with a scale or title, subtitle & caption
   - Use: This is the additional layer we are adding to mention, title, subtitle etc.

6. `facet()` • Purpose: Base class for all facets
   • Use: Faceting is a way to subset data and plot it on different panels.
7. `Compose()` • Purpose: Base class for those that create plot compositions
   • Use: This functionality allows to combine different plots in a multiple figrue panel easily. The main composing operators are; `/`: Arrange operands side by side, `|`: Arrange operands vertically and `-`: Arrange operands side by side and at the same nesting level.

It is good to explore the available datasets in Plotnine and try new visualizations in an iterative way.

- Plotnine user-guide tutorials: https://plotnine.org/guide/introduction.html
- Plotnine function reference list: https://plotnine.org/reference/

0.6 Function Glossary of Altair

1. `Chart()`

• The core function that creates a new Chart object where you provide your dataset (often a pandas DataFrame).
• Use: Begin your visualization by passing your data source to `alt.Chart(data)`.

2. `mark_*()` • A family of methods that specify the type of geometric mark or plot you want to create(e.g., mark_point(), mark_line(), mark_bar(), mark_area(), mark_circle(), …)
   • Use: Choose the mark that best represents your data. For example, use `mark_line()` for trends over time or `mark_bar()` for representing aggregates.
3. `encode()`
   • A method that maps your data fields to visual properties (channels) like x, y, color, size, and shape.
   • Use: Define the aesthetics of your chart by linking data columns to visual variables. For example, `alt.Chart(data).mark_bar().encode(     x='x:N',  # specify nominal data     y='y:Q',  # specify quantitative data )`.

List of channel options: https://altair-viz.github.io/user_guide/encodings/channels.html

4. `layer()`
   • A method to overlay multiple charts on top of one another within the same coordinate space.
   • Use: Combine different marks (such as points over a line) to provide additional context or annotations. To illustrate it may look like `alt.layer(   base.mark_line(), base.mark_point(),   base.mark_rule() ).interactive()`
5. `facet()`
   • Splits data into multiple sub-charts (facets) based on one or more fields.
   • Use: Create small multiple views for comparing subgroups within your data, each with its own independent plot.
6. `interactive()`
   • Adds built-in interactivity (pan, zoom, tooltips) to a chart.
   • Use: Enhance the user experience by making charts interactive without additional code. For example `alt.Chart(cars).mark_point().encode(     x='Horsepower', y='Miles_per_Gallon',     color='Origin', ).interactive()`

7. `transform_*()`
   - Filters the underlying data based on a query or condition directly within the chart definition.

It is often necessary to transform or filter data in the process of visualizing it. In Altair you can do this one of two ways:

```
1. Before the chart definition, using standard pandas data transformations.
2. Within the chart definition, using Vega-Lite's data transformation tools.
```

In most cases, we suggest that you use the first approach, because it is more straightforward to those who are familiar with data manipulation in Python, and because the pandas package offers much more flexibility than Vega-Lite in available data manipulations. The second approach becomes useful when the data source is not a dataframe, but, for example, a URL pointer to a JSON or CSV file. It can also be useful in a compound chart where different views of the dataset require different transformations.

For example, `transform_filter()` serves to select a subset of data based on a condition or `transform_calculate()`is useful to create a new data column using an arithmetic calculation on an existing column.

8. `save()`
   - A function that allows you to export your chart to various formats (HTML, PNG, JSON).
   - Use: Preserve and share your final visualizations, for example, `chart.save('chart.png')`.

This glossary should help you while working on Altair related examples to start with. Please first see the Altair examples given in `Introduction2DataViz.ipynb` and then move forward with Set 2-3-4 based on your project needs.

0.6.1 Main Steps of Altair-based Data Visualization

1. **Import Libraries**
   - Import necessary libraries such as Altair and Pandas to handle data and create visualizations.
2. **Prepare Your Data**
   - Load and prepare your data using Pandas, ensuring it's in a suitable format for visualization.
3. **Create a Base Chart**
   - Use `alt.Chart()` to define a base chart with your DataFrame as the starting point.
4. **Choose a Mark Type**
   - Select the type of visualization (e.g., bar, line, scatter) using the appropriate `mark_*` method. Such as; `mark_line()`, `mark_point()`, etc.
5. **Encode Data**
   - Map data fields to visual properties like axes, colors, and sizes using the `encode()` method.

As an additional open-source examples gallery would be

- Exploratory Data Visualization with Altair: https://altair-viz.github.io/altair-tutorial/README.html

0.7 Common Troubleshooting Tips

1. **Package Installation and Imports**
   - Ensure required packages are installed using pip if you are using some local environment (e.g., pip install pandas).
   - Verify that you are importing packages correctly (e.g., import pandas as pd).
2. **Syntax and Typographical Errors**
   - Check for spelling mistakes in function names and variable identifiers.
   - Ensure you follow proper Python syntax, including indentation, parentheses, and commas.
3. **Reading Error Messages**
   - Read traceback error messages carefully—they often indicate the source and nature of the problem.
   - Use the error details to search online for solutions or refer to the package's documentation.
4. **Data Compatibility Issues**
   - When using pandas, confirm that your data types are appropriate (e.g., numerical data for plotting).
   - Use functions like `df.info()` and `df.describe()` to inspect your DataFrame structure before visualization.
5. **Debugging Techniques**
   - Use print statements or the debugger (e.g., Python's built-in pdb module) to inspect variable values and code flow.
   - For interactive environments (like Jupyter Notebook), use cell-by-cell execution to identify where issues arise.
6. **Documentation and Community Support**
   - Refer to official documentation for pandas, matplotlib, seaborn, and other libraries for usage examples and best practices.
   - Utilize community forums (e.g., Stack Overflow, GitHub issues) to troubleshoot common problems or use examples from Python content platforms such as RealPython.