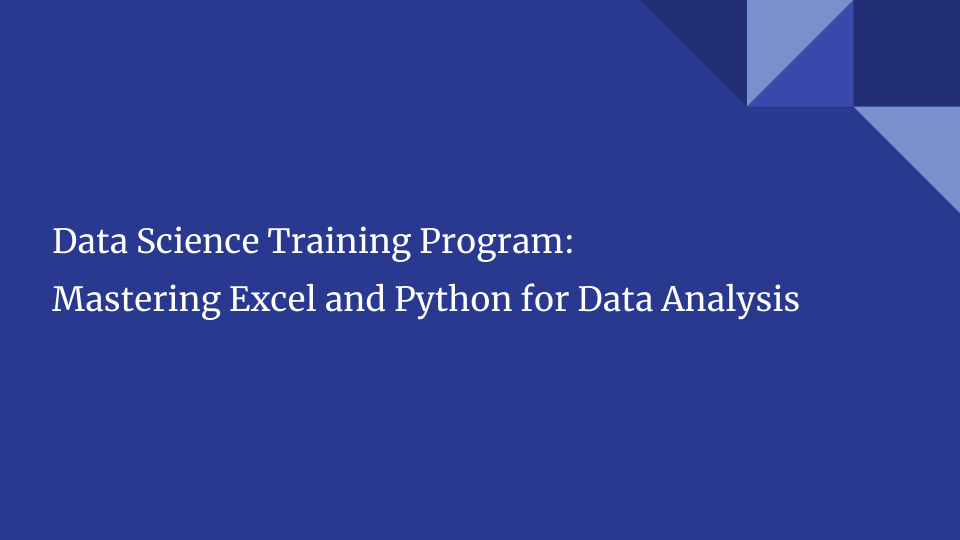
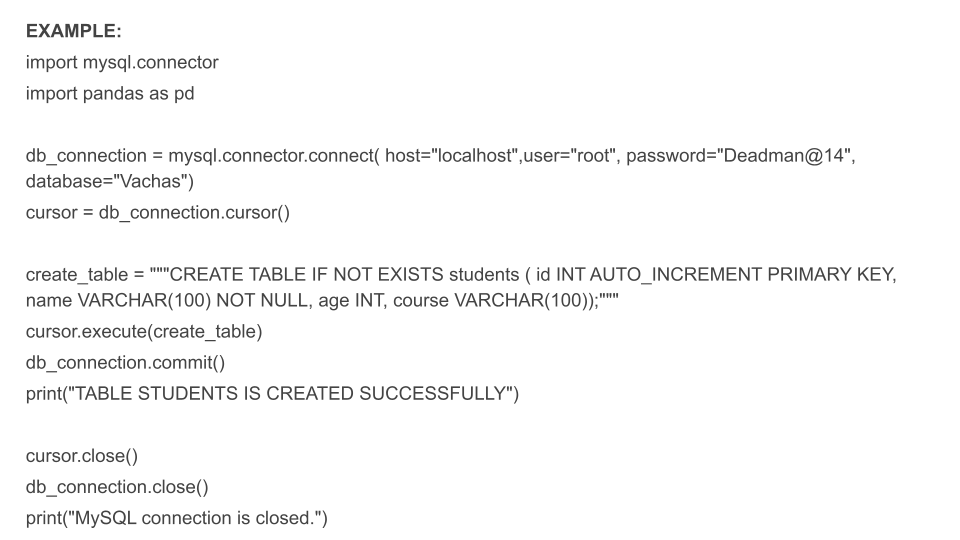
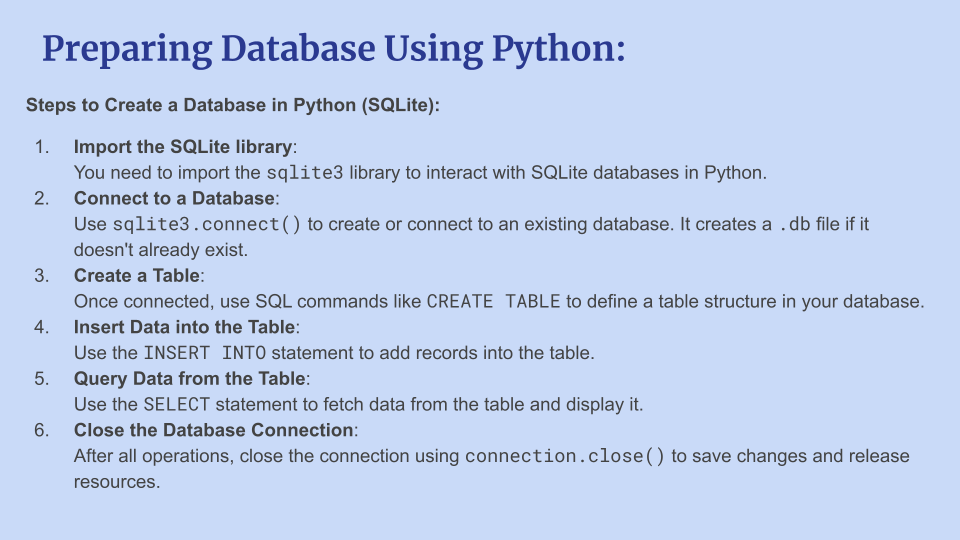
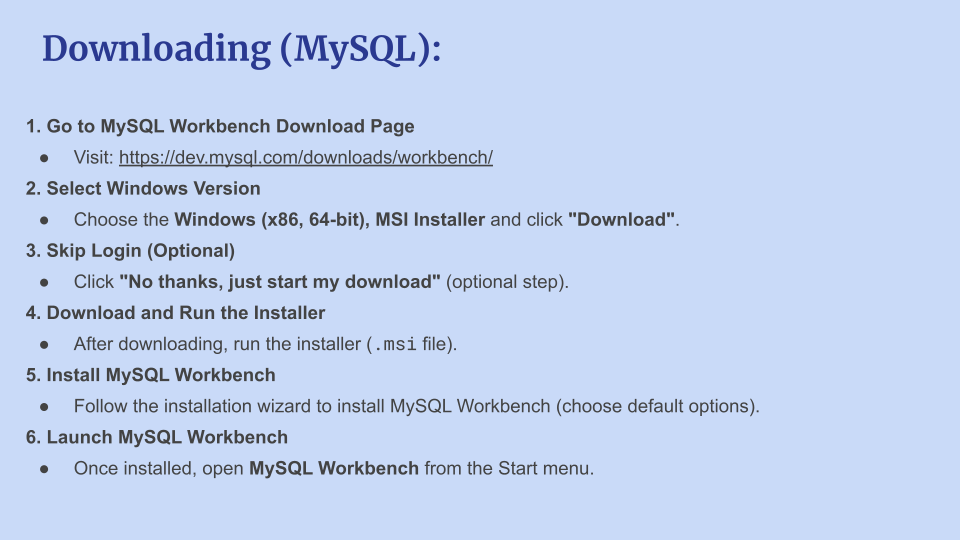
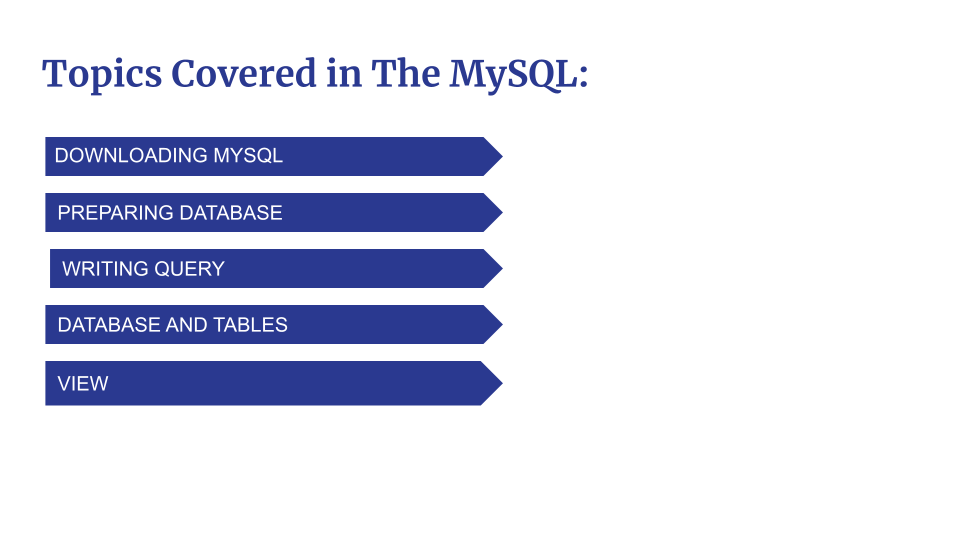
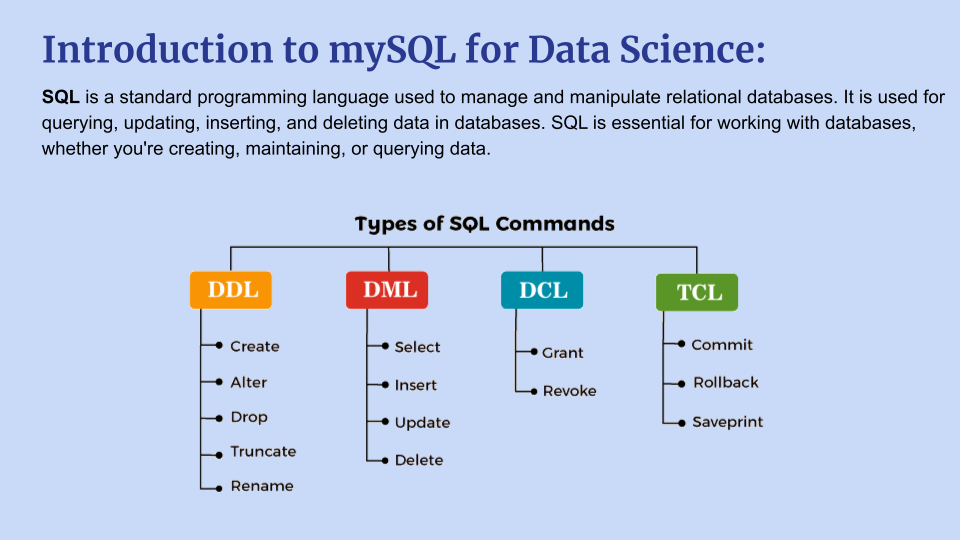
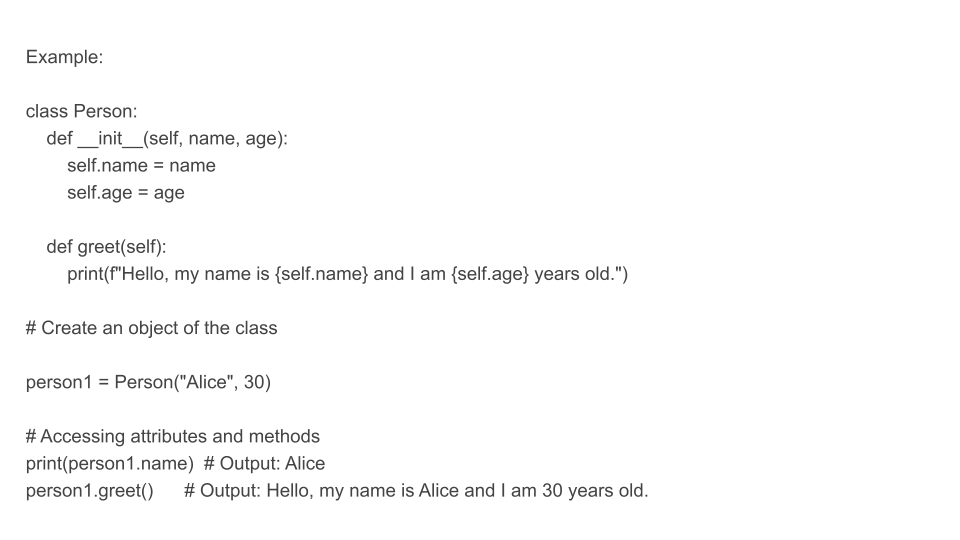
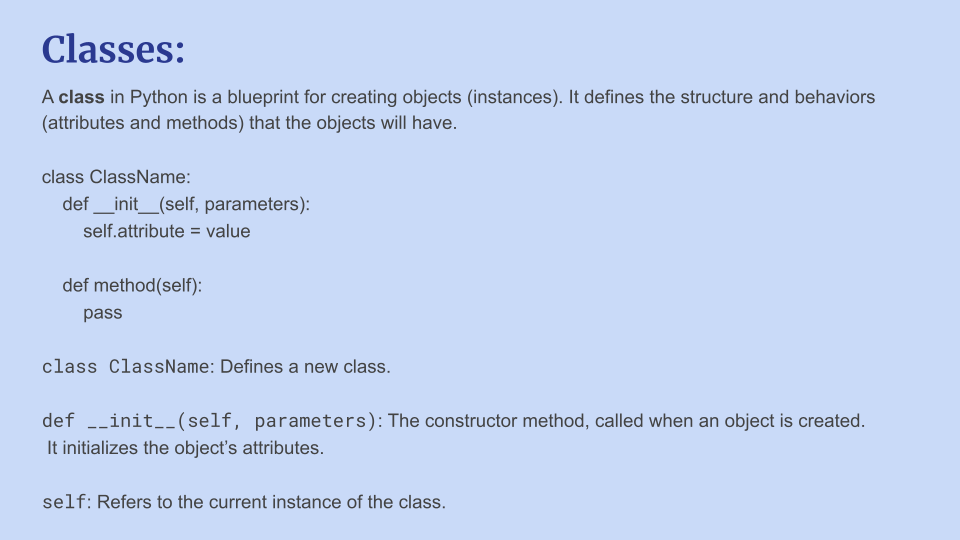
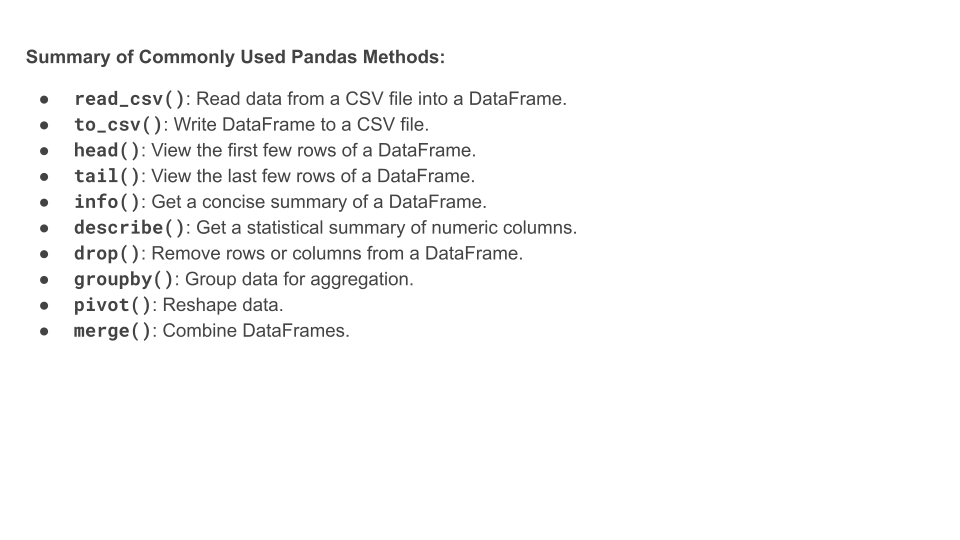
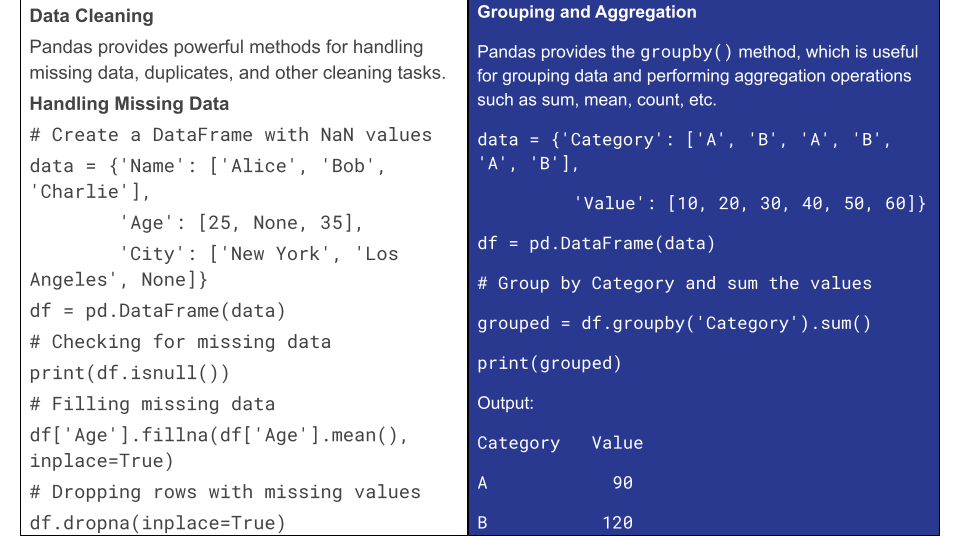
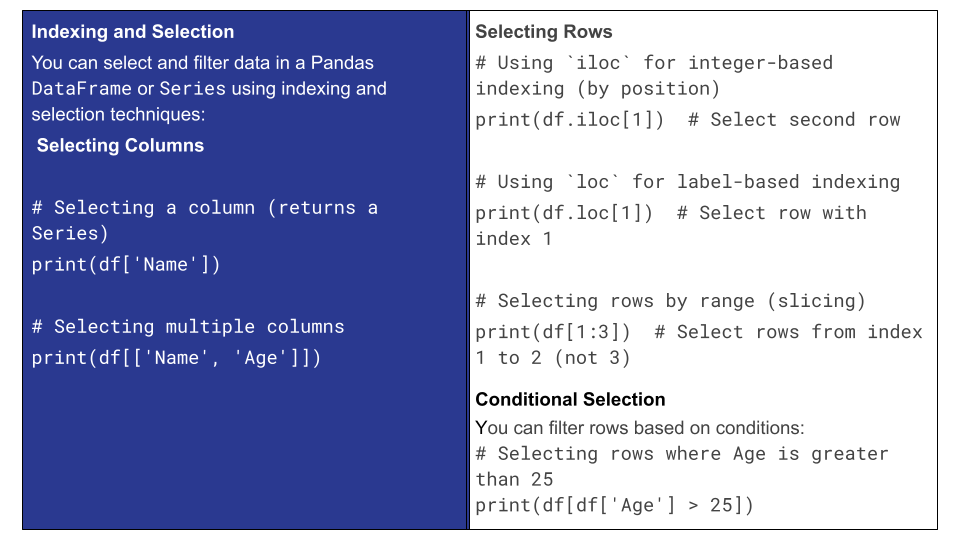
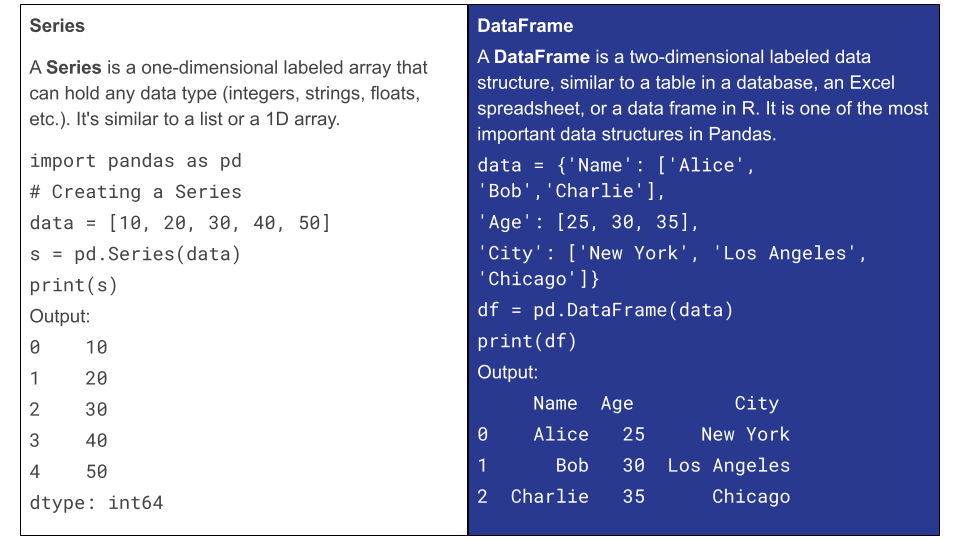
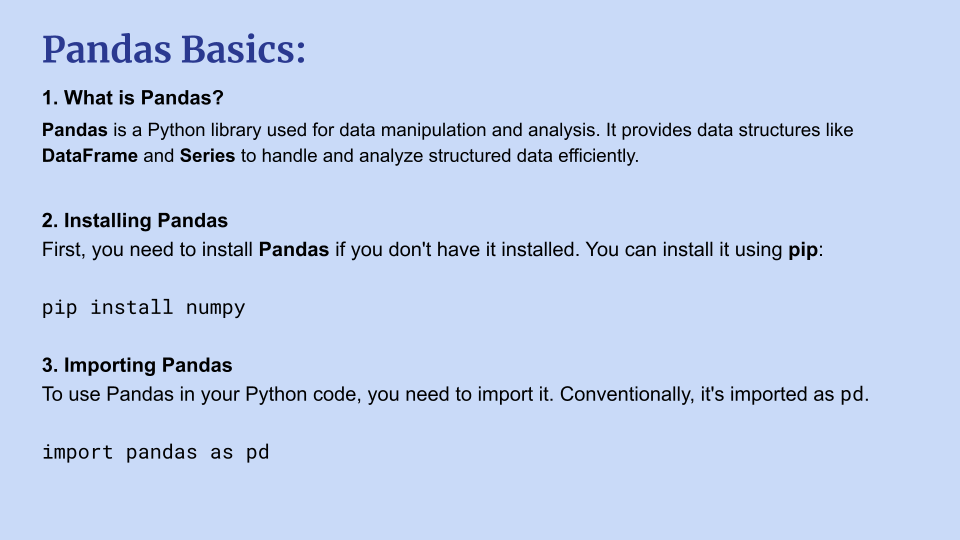
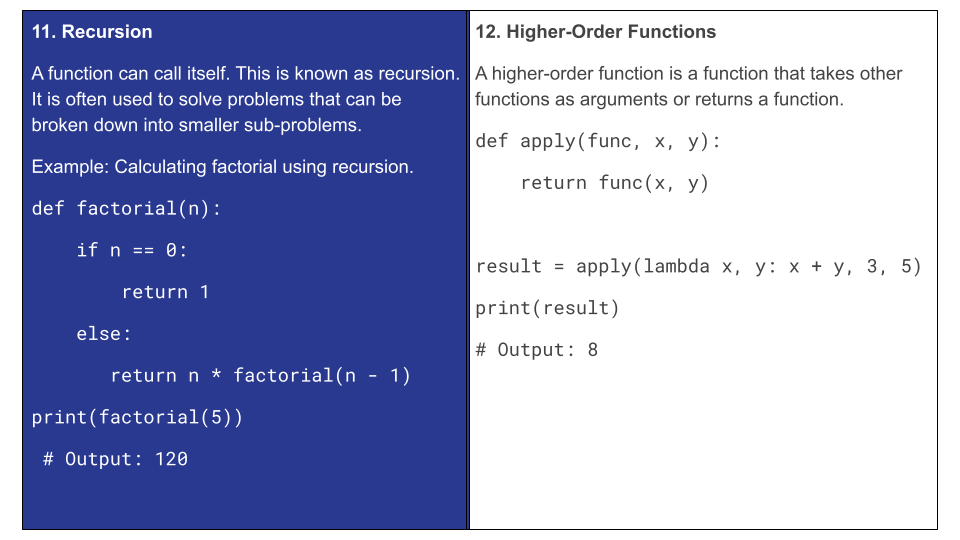
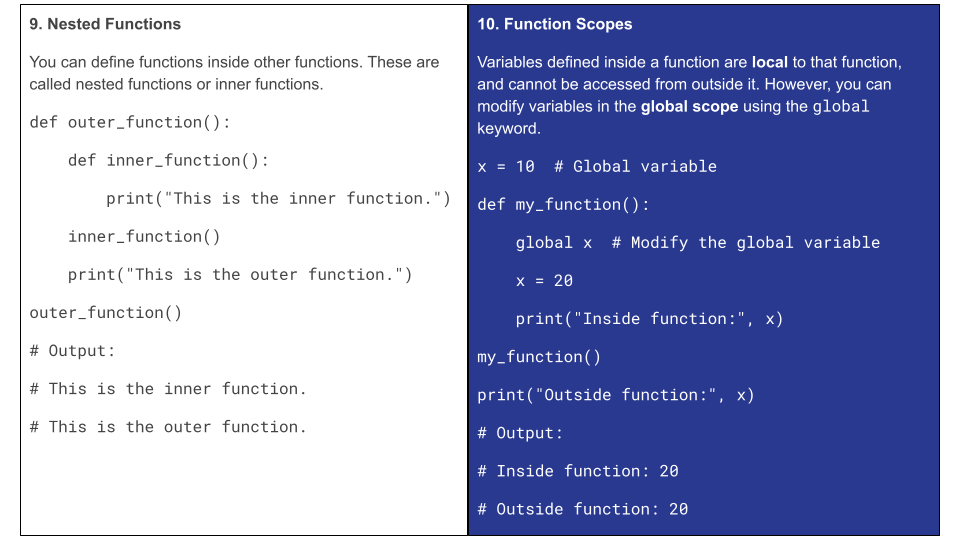
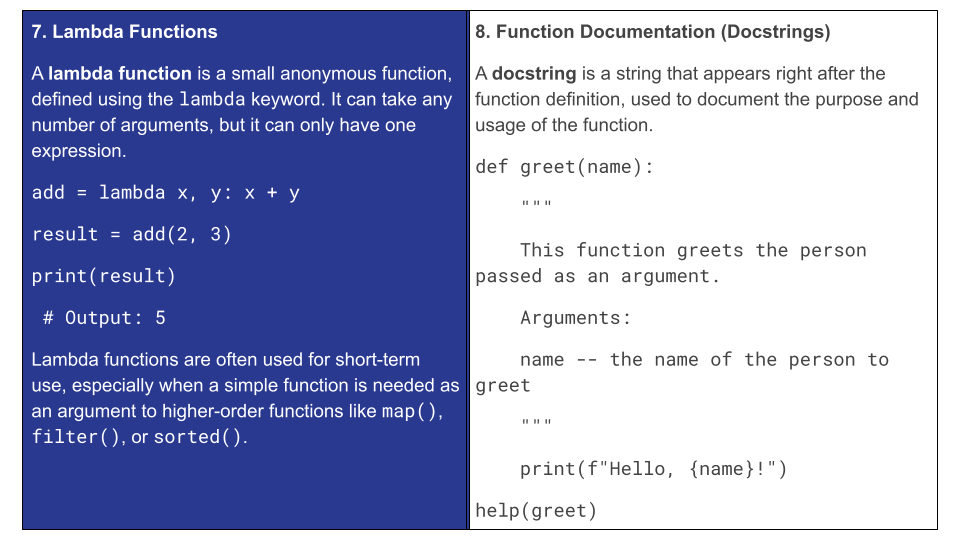
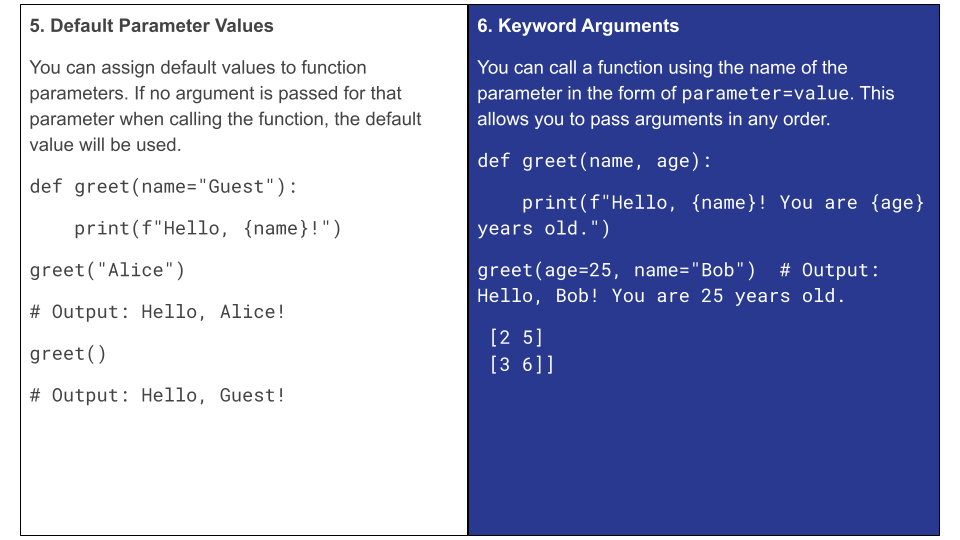
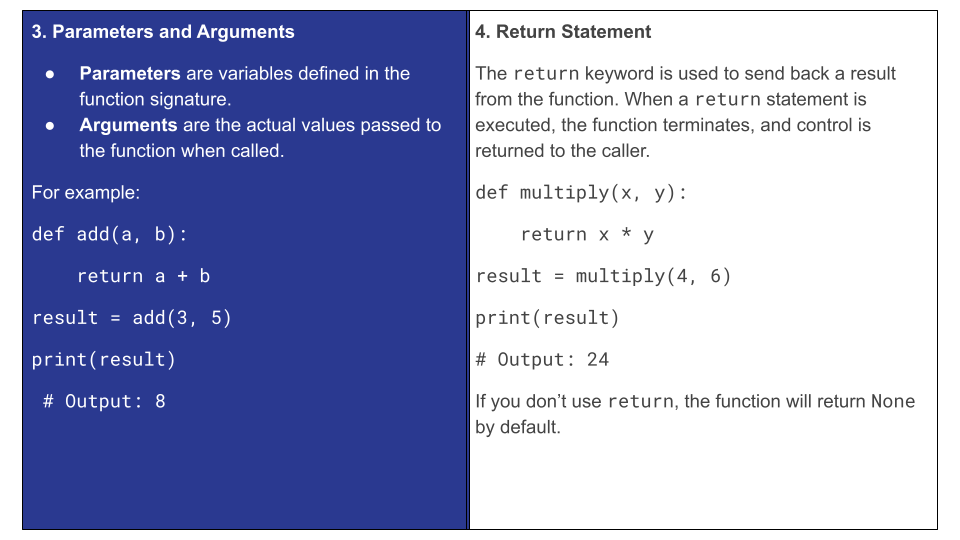
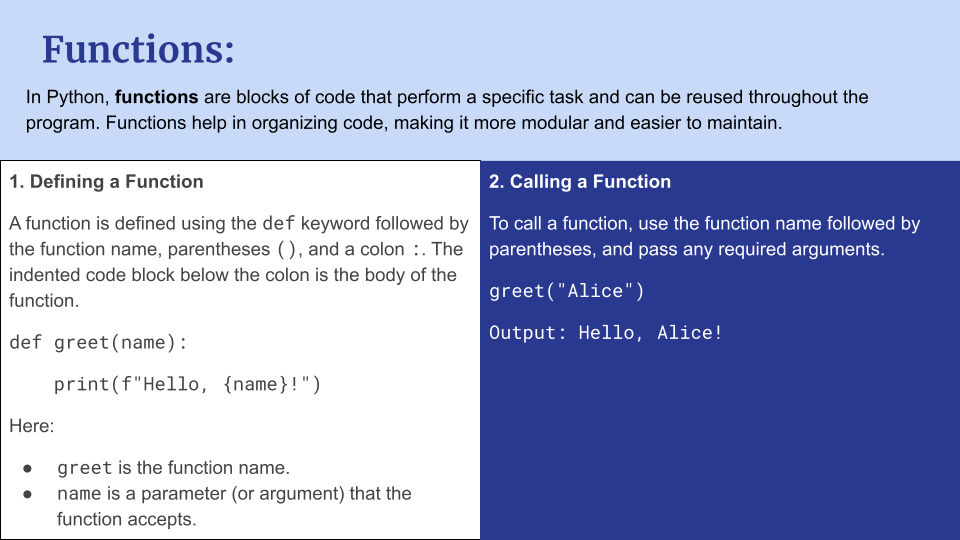
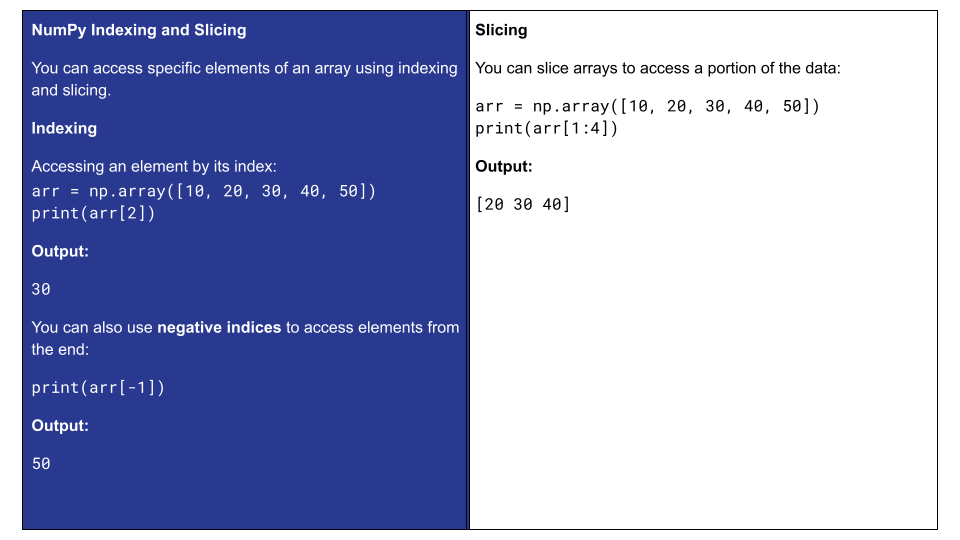
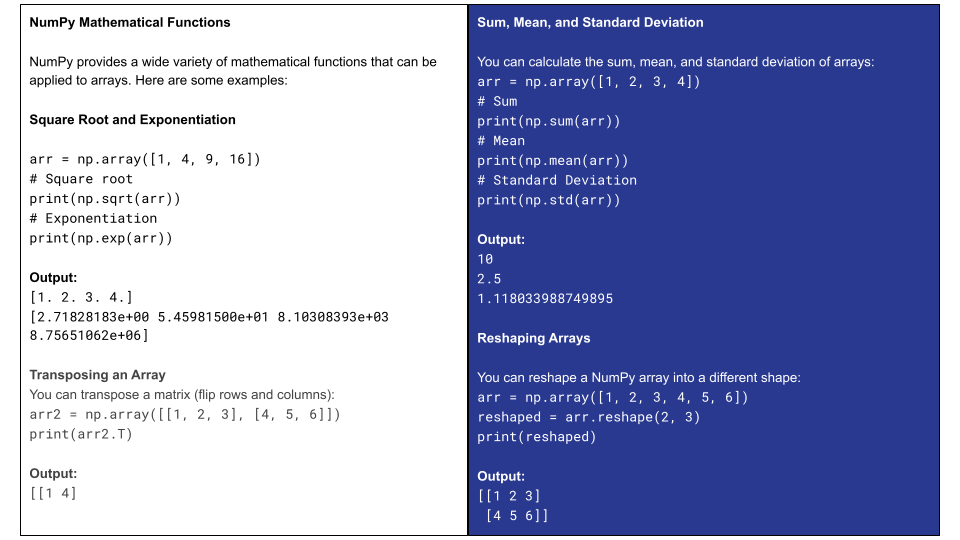
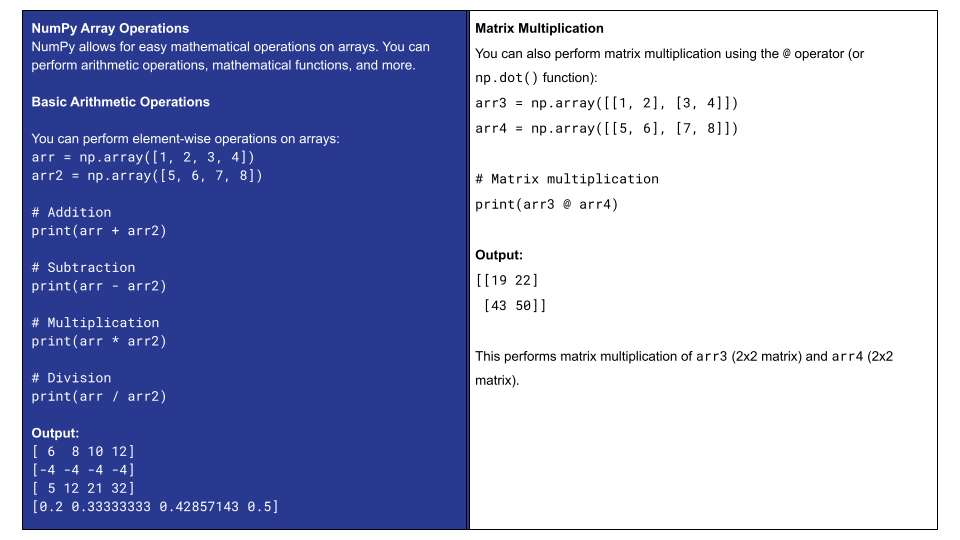
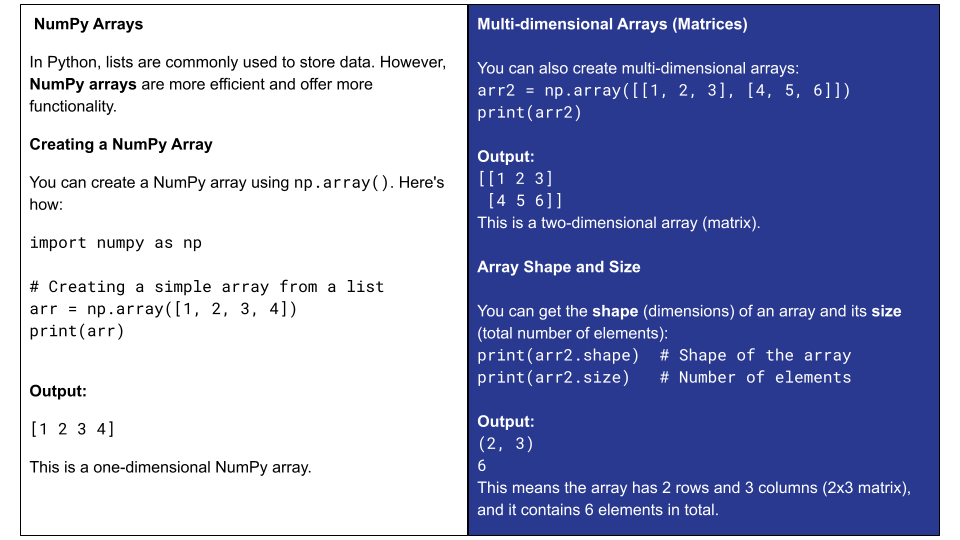
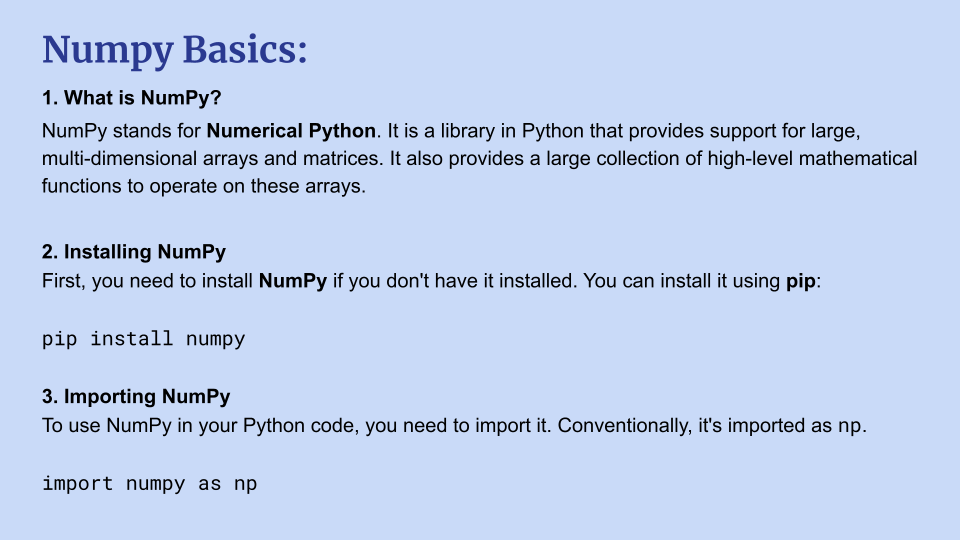
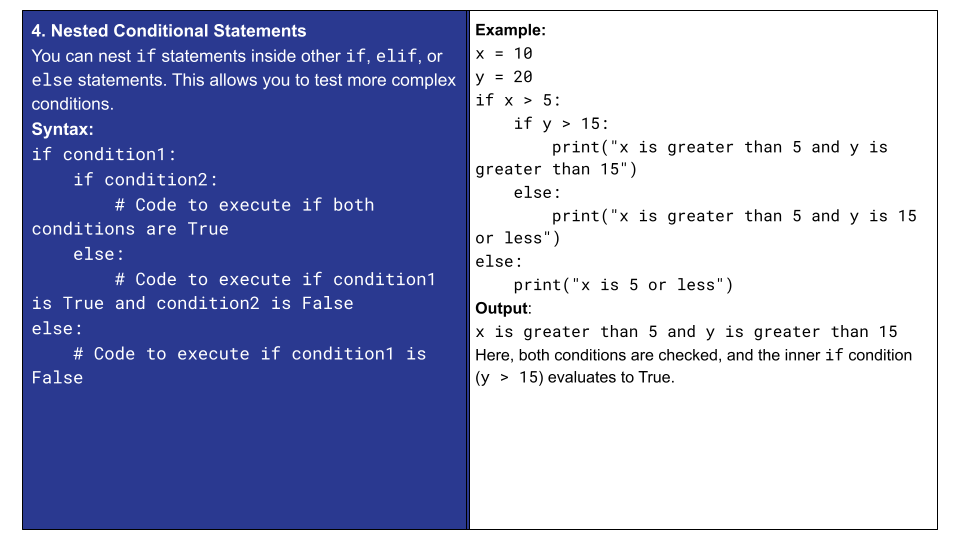
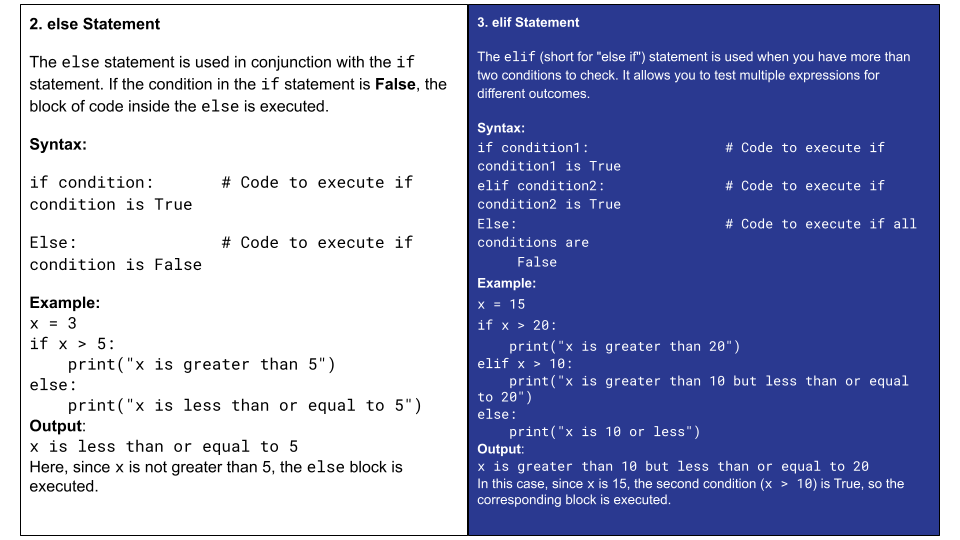
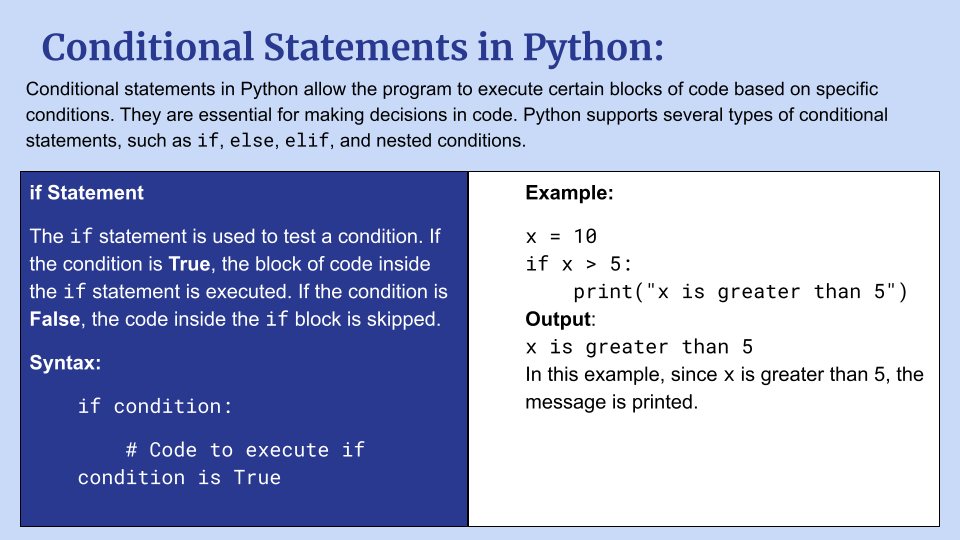
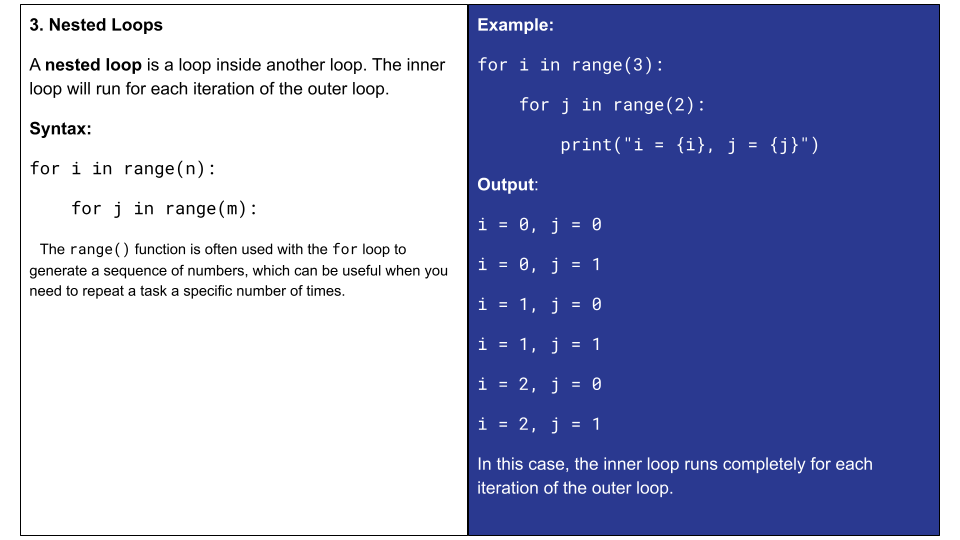
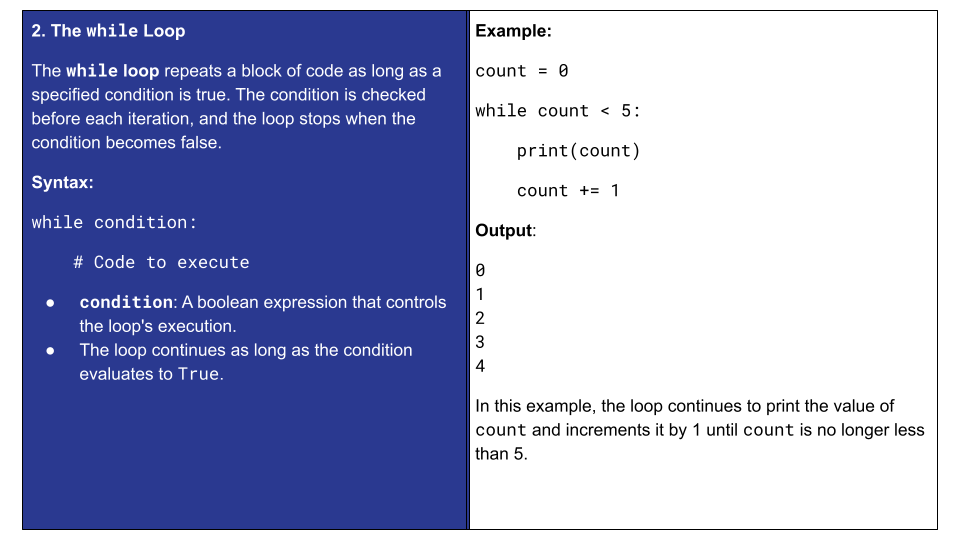
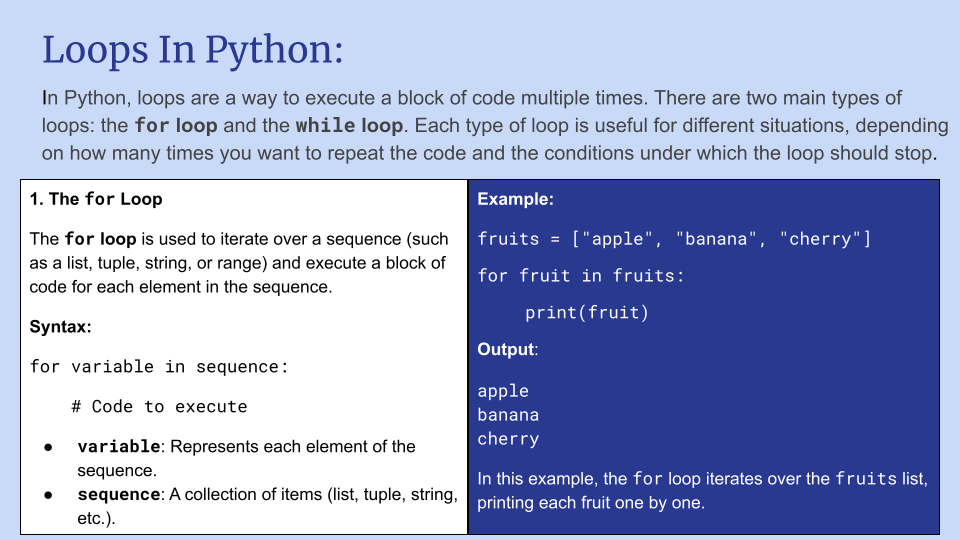
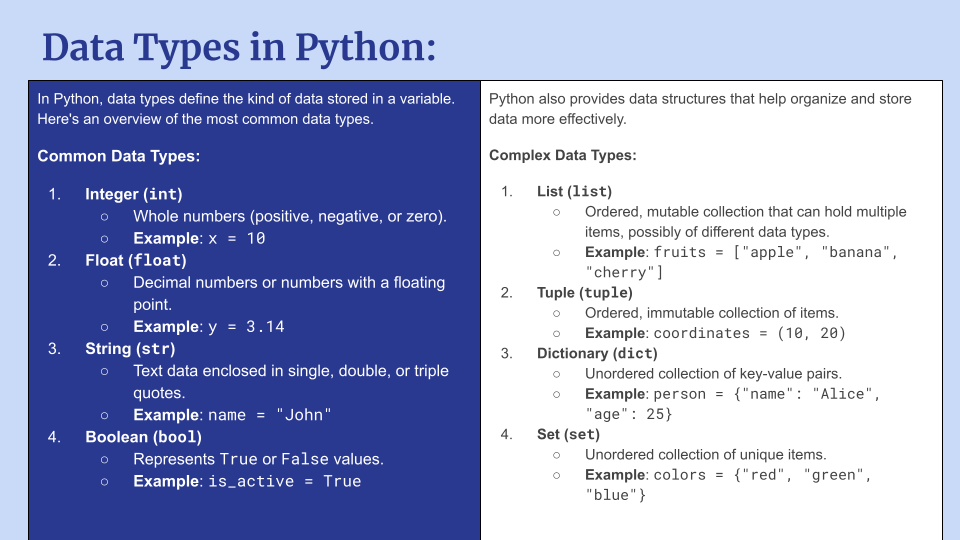
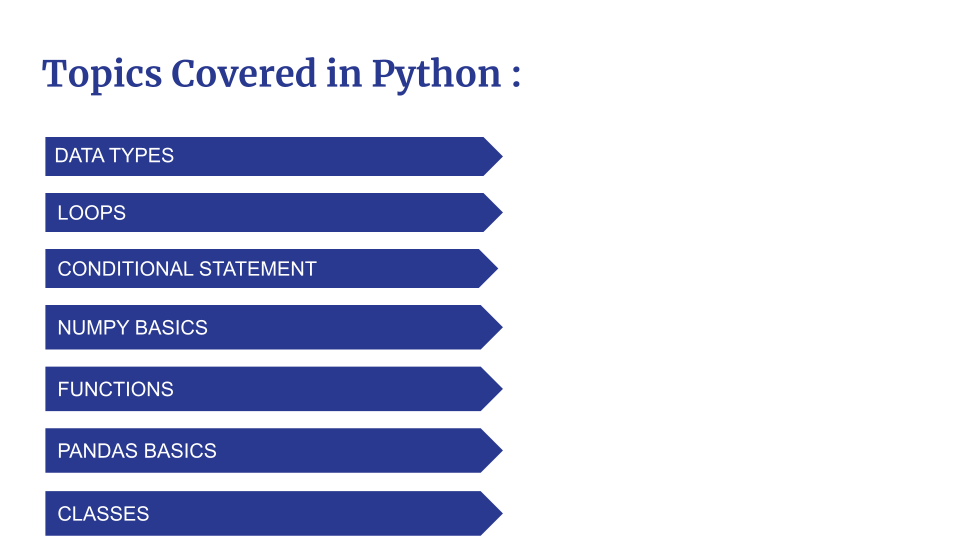
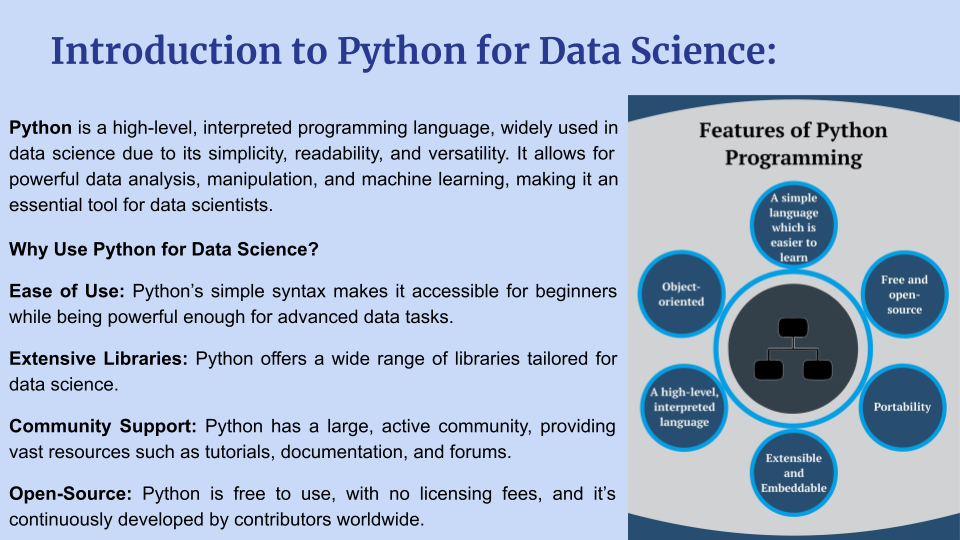
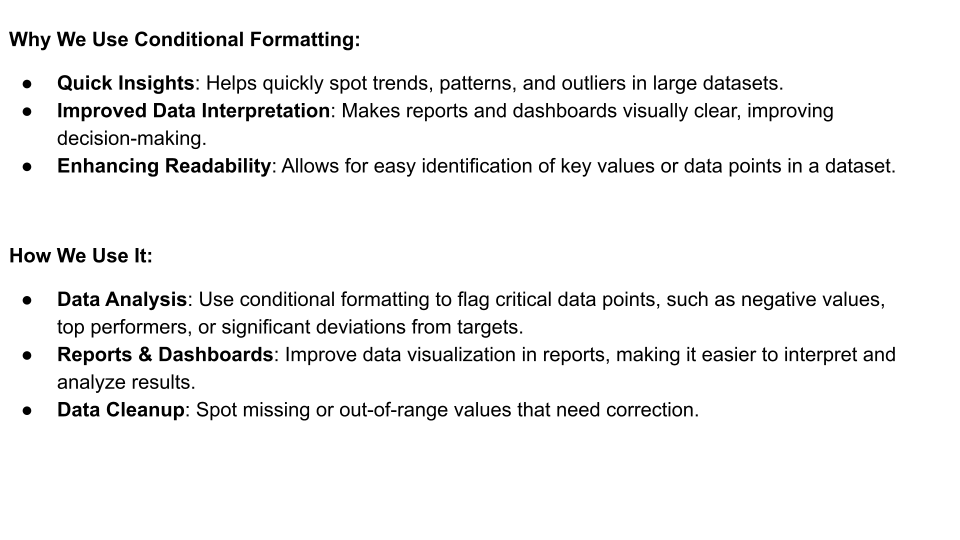
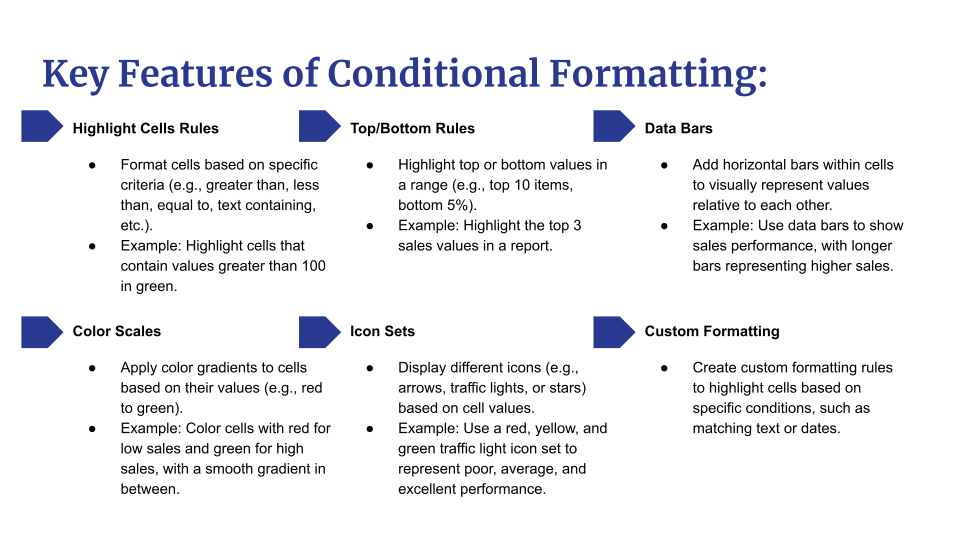
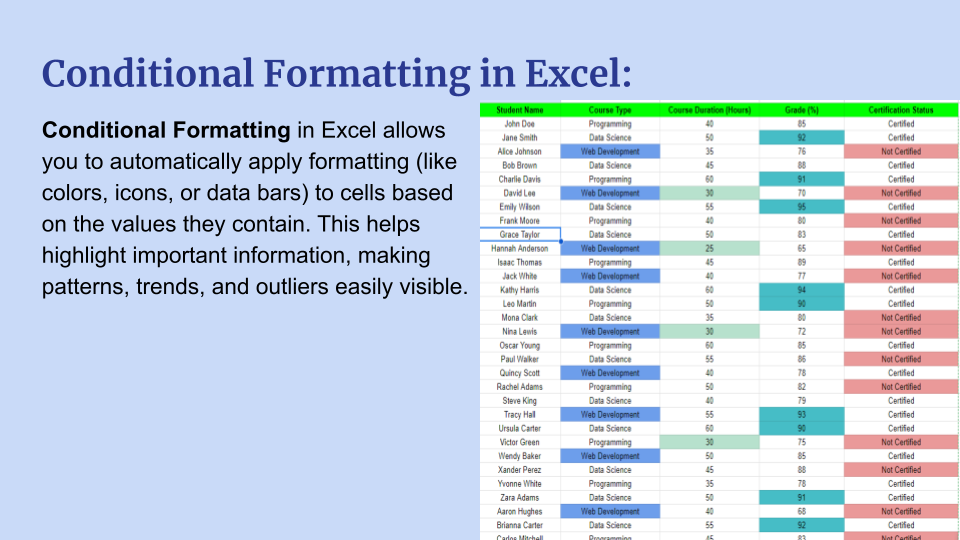
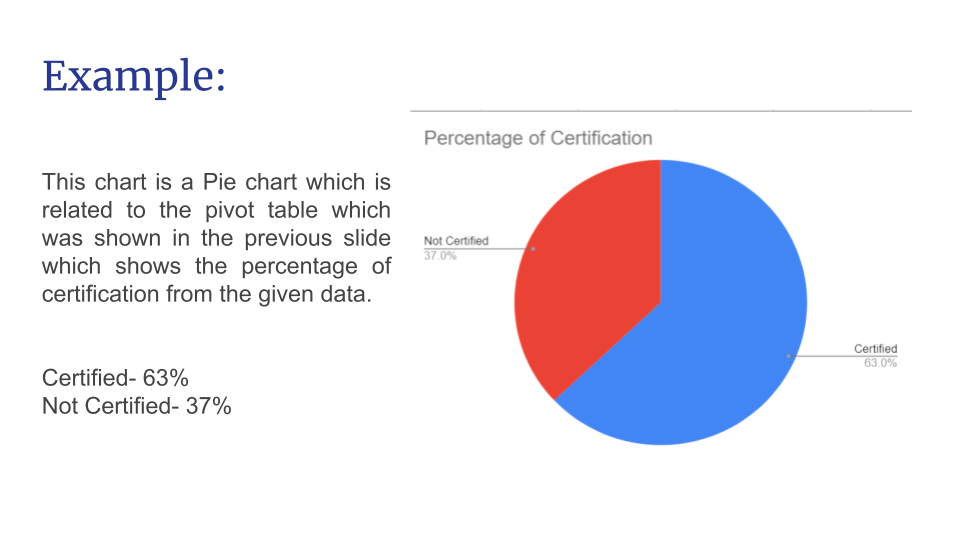
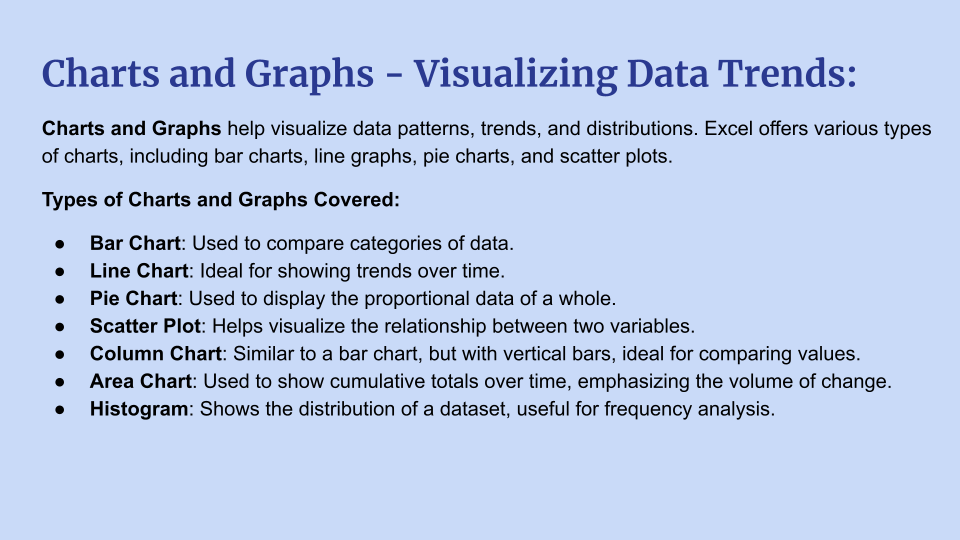
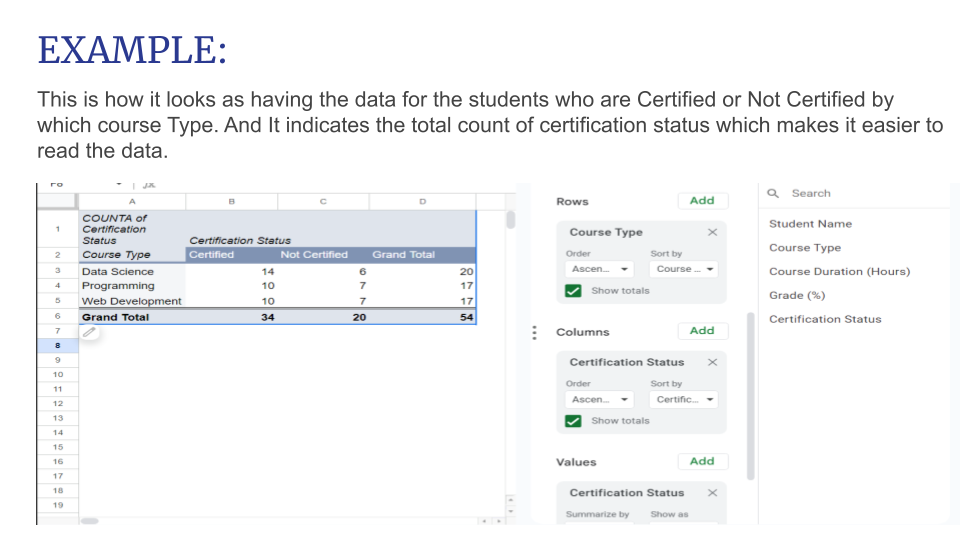
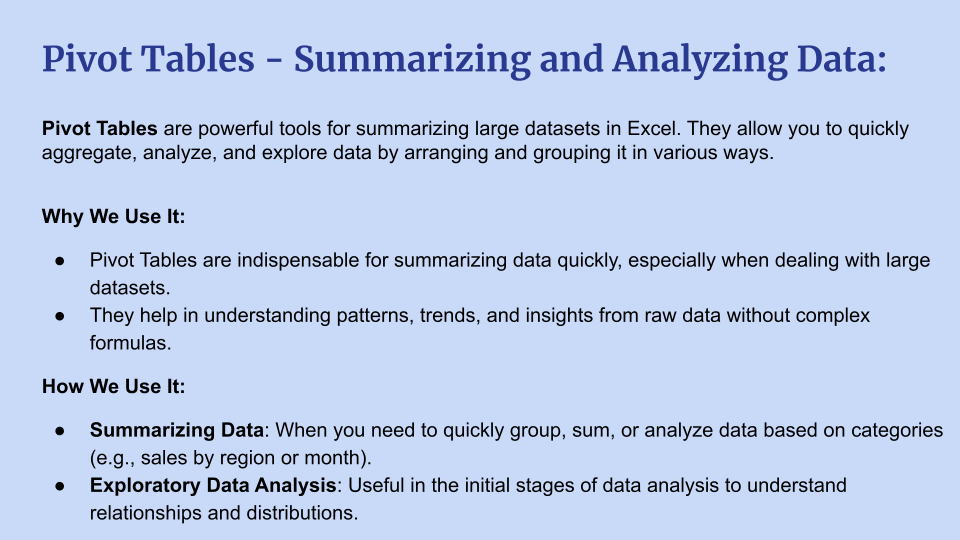
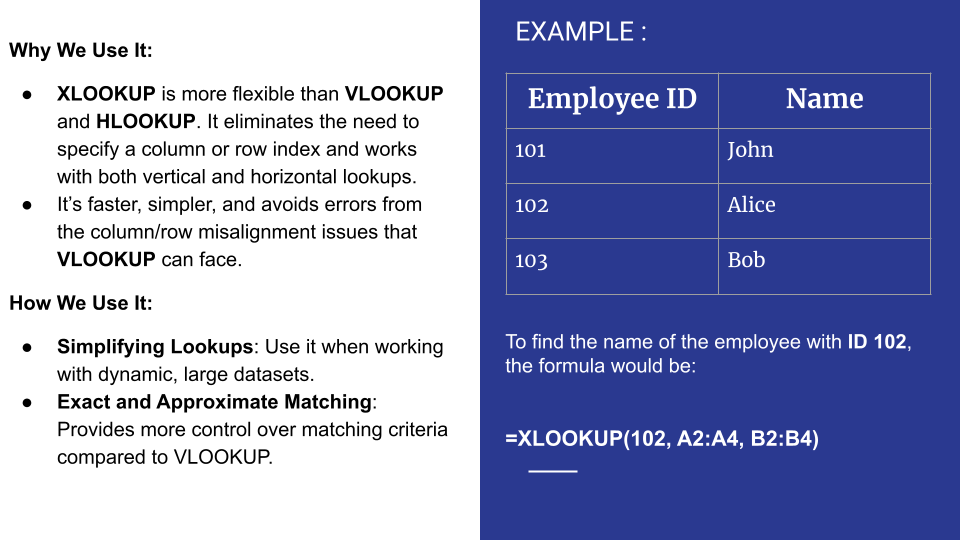
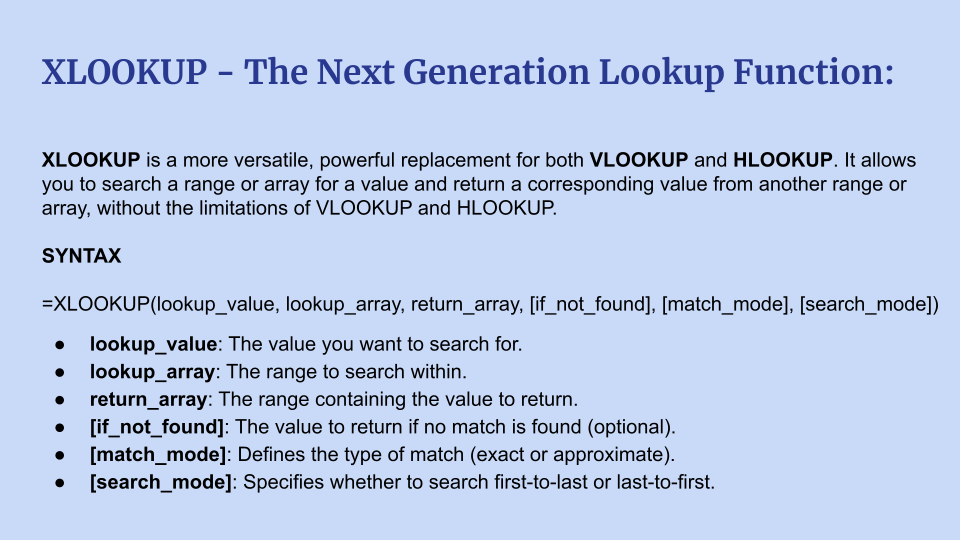
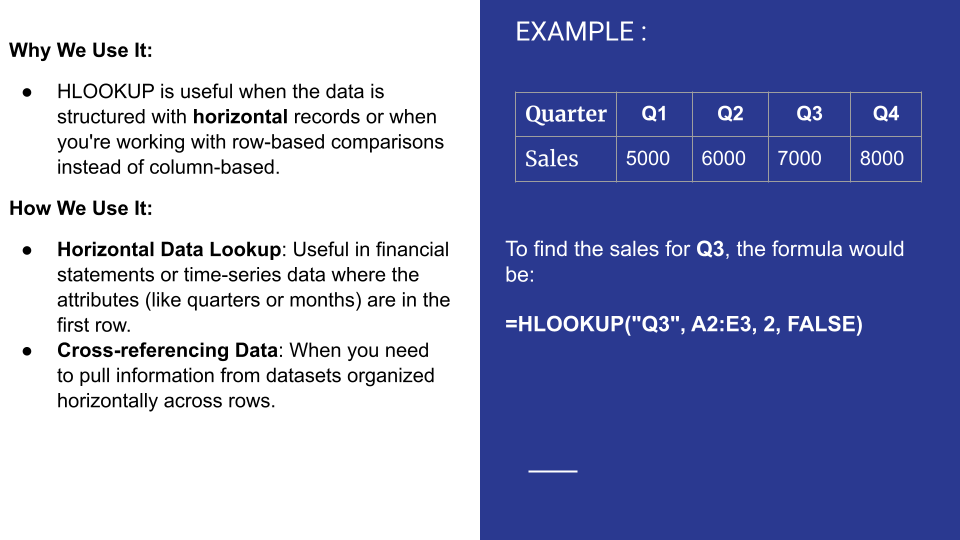
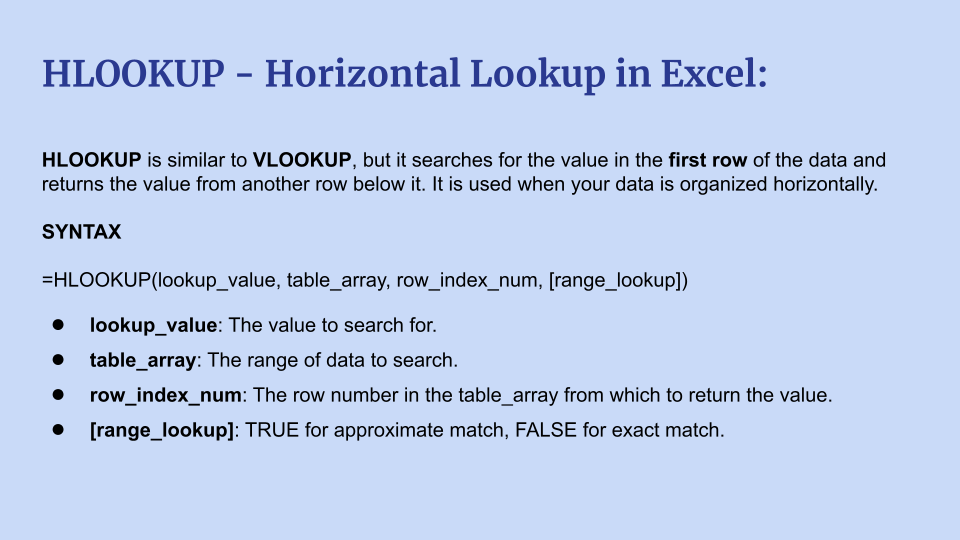
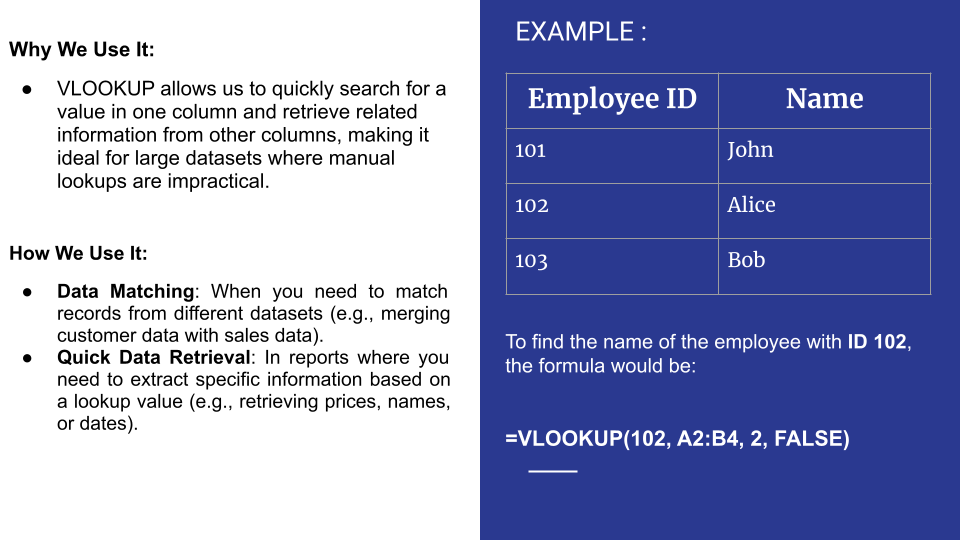
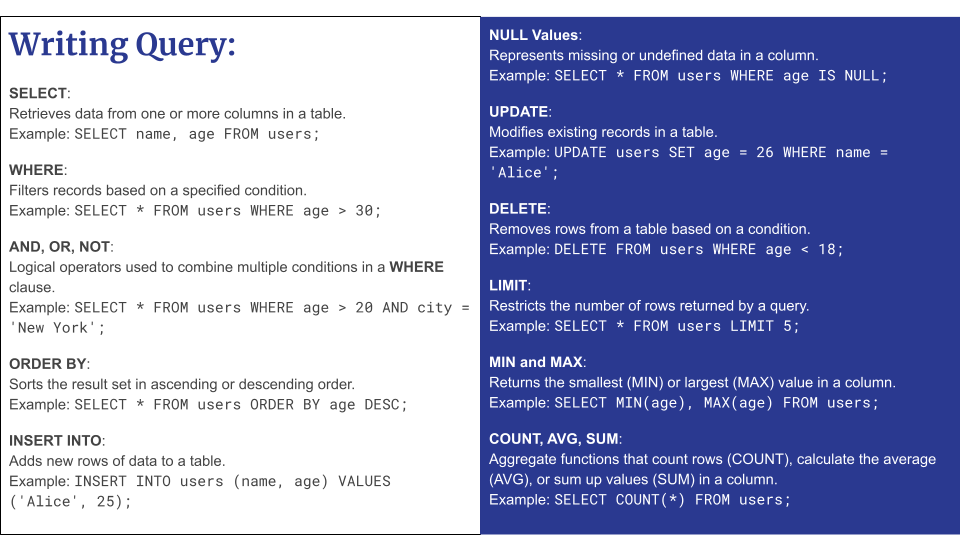
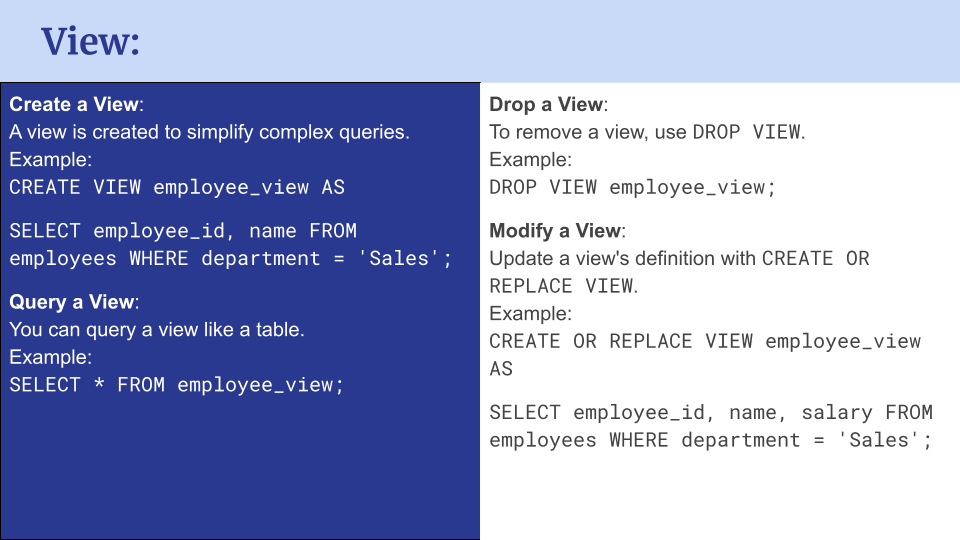
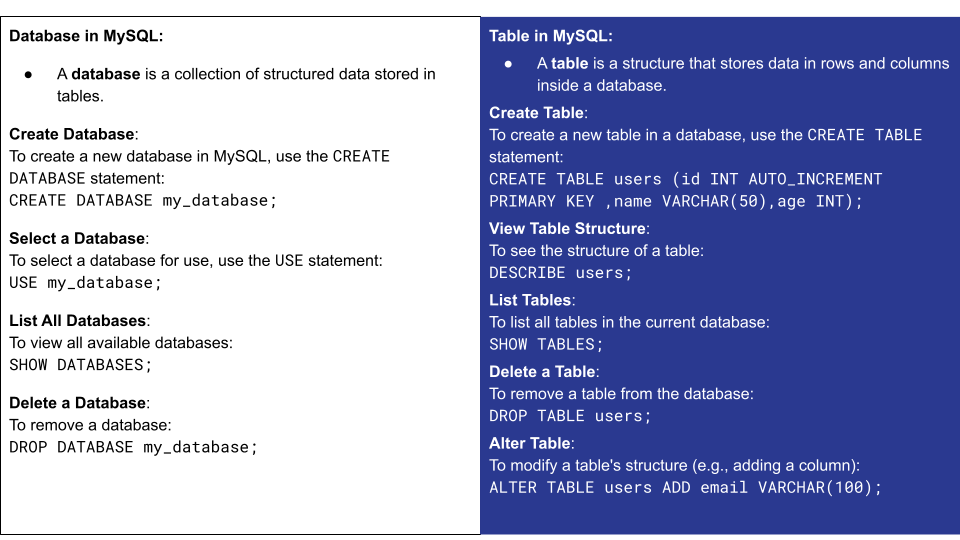
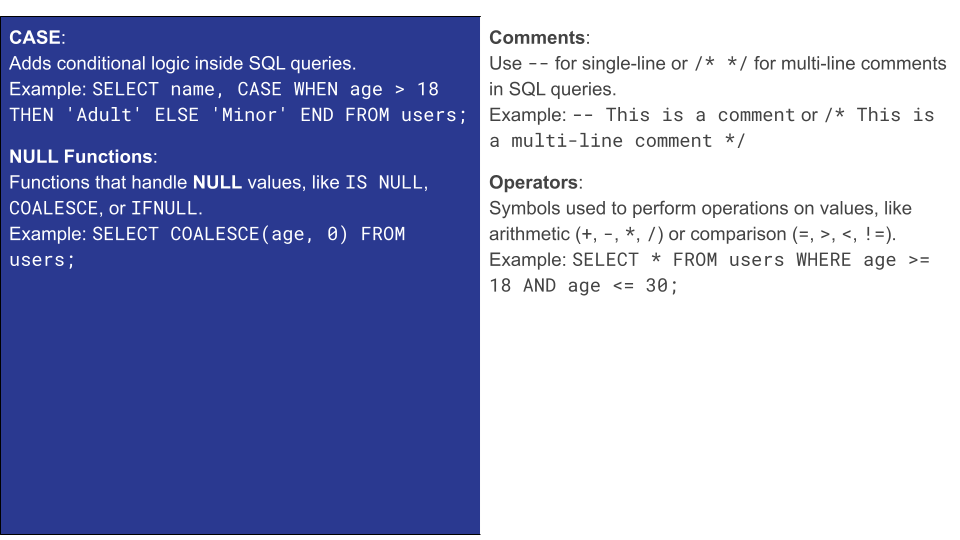
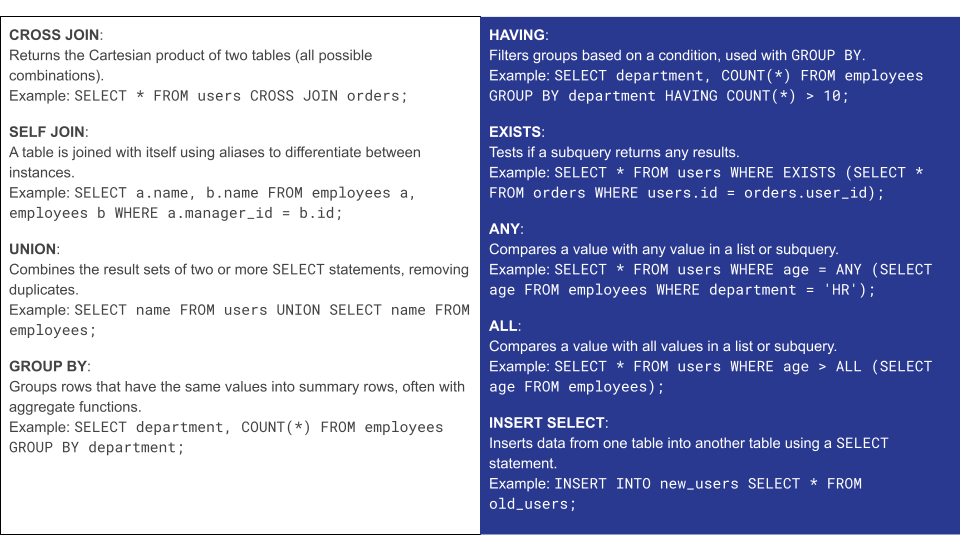
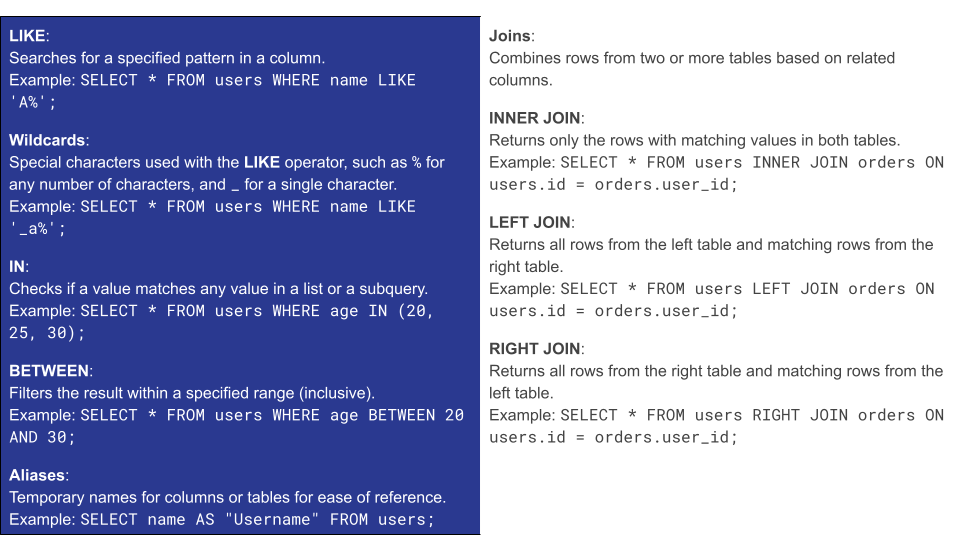
Excel,Python,MySQL









Advance SQL

### **Windows Functions With OVER**

|  |  |  |
| --- | --- | --- |
| **Window Function** | **Definition** | **Syntax** |
| **ROW\_NUMBER()** | Assigns a unique sequential integer to rows within a partition of a result set. | ROW\_NUMBER() OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **RANK()** | Assigns a rank to each row, with gaps in ranking when there are ties. | RANK() OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **DENSE\_RANK()** | Similar to RANK(), but does not leave gaps in ranking when there are ties. | DENSE\_RANK() OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **NTILE(n)** | Divides the result set into n approximately equal parts and assigns a bucket number to each row. | NTILE(n) OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **LEAD()** | Provides access to the next row's value in the result set without the need for a self-join. | LEAD(column, offset) OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **LAG()** | Provides access to the previous row's value in the result set. | LAG(column, offset) OVER (PARTITION BY partition\_column ORDER BY order\_column) |
| **SUM(), AVG(), MIN(), MAX()** | Aggregate functions used as window functions to calculate running totals, averages, or other metrics within partitions. | SUM(column) OVER (PARTITION BY partition\_column ORDER BY order\_column), AVG(column) OVER (PARTITION BY partition\_column ORDER BY order\_column), |

### **OVER() Clause:**

The OVER() clause defines the window over which the window function will operate. It can include:

* **PARTITION BY**: Divides the result set into partitions (groups of rows), with each partition getting its own set of calculations.
* **ORDER BY**: Specifies the order of rows within the window, which is often required for functions like ROW\_NUMBER(), RANK(), and others.
* **ROWS BETWEEN**: Defines a specific frame of rows within the partition to perform calculations on (e.g., a sliding window).

### **FIRST\_VALUE() Function in MySQL**

The FIRST\_VALUE() function is a window function in MySQL that retrieves the first value of a column within a specified window (group of rows). The window is determined by the PARTITION BY and ORDER BY clauses, which control how the rows are grouped and ordered.

### **Syntax:**

sql

FIRST\_VALUE(expression) OVER (PARTITION BY partition\_column ORDER BY order\_column [ROWS frame\_specification])

* **expression**: The column or expression for which you want the first value.
* **PARTITION BY**: (Optional) Divides the result set into partitions. The function operates on each partition separately.
* **ORDER BY**: (Required) Specifies the order of rows within each partition to determine which value is "first."
* **ROWS**: (Optional) Defines a specific window of rows within the partition. If omitted, the entire partition is considered.

### **Key Points:**

* **Window Function**: Operates over a window of rows, not just the entire table.
* **PARTITION BY**: Groups rows into partitions, allowing separate calculations for each group.
* **ORDER BY**: Defines the order of rows to determine the "first" value in the window.

### **SubQueries**

|  |  |  |
| --- | --- | --- |
| **Concept** | **Explanation** | **Syntax** |
| **Subqueries** | A subquery is a query embedded within another query, typically used to return data for the outer query. Can be used in SELECT, WHERE, or FROM clauses. | SELECT column1, column2 FROM table\_name WHERE column1 = (SELECT column\_name FROM table\_name WHERE condition); |
| **Types of Subqueries** | | |
| **Scalar Subquery** | Returns a single value (1 row, 1 column). | SELECT column\_name FROM table\_name WHERE condition = (SELECT single\_value FROM another\_table WHERE condition); |
| **Row Subquery** | Returns a single row with multiple columns. | SELECT column1, column2 FROM table\_name WHERE (column1, column2) = (SELECT column1, column2 FROM another\_table WHERE condition); |
| **Table Subquery** | Returns multiple rows and multiple columns. Often used in the FROM clause, treating the result as a temporary table. | SELECT column1, column2 FROM (SELECT column1, column2 FROM table\_name WHERE condition) AS derived\_table; |
| **Correlated Subquery** | References columns from the outer query and is executed for each row processed by the outer query. | SELECT column1, column2 FROM table\_name outer WHERE column1 = (SELECT column1 FROM inner\_table WHERE inner\_table.column1 = outer.column1); |
| **Subqueries in SELECT** | A subquery in the SELECT clause allows you to compute values or return additional columns. | SELECT column1, (SELECT aggregated\_column FROM table\_name WHERE condition) AS computed\_column FROM table\_name; |
| **Subqueries in WHERE** | A subquery in the WHERE clause filters results from the outer query by calculating values (e.g., averages, maximums). | SELECT column1, column2 FROM table\_name WHERE column1 = (SELECT value FROM another\_table WHERE condition); |
| **Subqueries in FROM** | A subquery in the FROM clause is treated as a temporary table or derived table, allowing complex queries. | SELECT column1, column2 FROM (SELECT column1, column2 FROM table\_name WHERE condition) AS derived\_table WHERE condition; |
| **Performance Considerations** | Subqueries can be less efficient, especially correlated subqueries, as they execute once for each row in the outer query. Joins are often more efficient. | |
| **Use with Operators** | Subqueries often work with operators like IN, EXISTS, ANY, ALL, and BETWEEN. | SELECT column1 FROM table\_name WHERE column1 IN (SELECT value FROM another\_table WHERE condition); |

### **Pivot Table in MySQL**

In MySQL, while there isn't a direct PIVOT function like in SQL Server or Oracle, you can achieve pivot-like results by using **conditional aggregation** with CASE statements and aggregate functions like SUM(), COUNT(), etc. This technique is used to transform rows into columns, allowing you to summarize and reshape your data.

### **Pivoting Concept:**

Pivoting involves transforming data so that values from a column (e.g., months) become columns, and you can perform aggregate functions (e.g., SUM(), COUNT()) over these new columns.

### **Syntax for Pivoting in MySQL:**

To pivot data in MySQL, you can use **CASE** statements inside aggregate functions. Here’s the basic syntax structure for pivoting:

sql

SELECT

column1,

SUM(CASE WHEN condition1 THEN column2 ELSE 0 END) AS column\_alias1,

SUM(CASE WHEN condition2 THEN column2 ELSE 0 END) AS column\_alias2

FROM table\_name

GROUP BY column1;

Where:

* column1: The column by which the data will be grouped (e.g., product).
* column2: The column whose values you want to aggregate (e.g., sales).
* condition1, condition2: The conditions that define how the rows are pivoted (e.g., different months).
* column\_alias1, column\_alias2: The alias names for the newly created columns (e.g., Jan\_sales, Feb\_sales).

### **Handling Missing Data:**

By default, when using aggregate functions like SUM(), MySQL treats missing data as 0. However, if you prefer to display NULL for missing values, you can modify the CASE statement to return NULL instead of 0.

sql

SELECT

column1,

SUM(CASE WHEN condition1 THEN column2 ELSE NULL END) AS column\_alias1,

SUM(CASE WHEN condition2 THEN column2 ELSE NULL END) AS column\_alias2

FROM table\_name

GROUP BY column1;

### **Dynamic Pivot:**

In cases where you do not know the values in advance (e.g., dynamic months or categories), you can create a dynamic pivot query by:

1. **Identifying unique values** that should become columns (e.g., distinct months).
2. **Generating dynamic CASE statements** in your application code or script based on the unique values retrieved.

For example, if you have months Jan, Feb, and Mar as distinct values, you can dynamically generate a query with CASE statements for each month.

### **Conclusion:**

* **Pivoting in MySQL** can be done using **conditional aggregation** with CASE statements inside aggregate functions.
* You can pivot data by grouping on a column and applying conditional logic with CASE.
* **Dynamic pivots** can be handled by first identifying distinct values (e.g., months) and then generating the corresponding CASE statements dynamically in your application or script.
* To handle **missing data**, you can adjust the CASE statement to return NULL instead of 0 if no value exists.

This method allows you to reshape and summarize your data effectively, even though MySQL doesn't have a built-in PIVOT function.

### **Common Table Expressions (CTEs) in MySQL:**

#### **What are CTEs?**

CTEs (Common Table Expressions) are temporary result sets that can be referenced within a query. They help in simplifying complex queries by breaking them into smaller, reusable parts.

#### **Syntax for CTE:**

sql

WITH cte\_name AS (

-- Your query here

)

SELECT column1, column2

FROM cte\_name;

**Key Components:**

* **WITH:** Introduces the CTE definition.
* **cte\_name:** The name of the CTE, used in the main query.
* **AS:** Defines the CTE.
* **( ... ):** Contains the query that generates the result set.

#### **Types of CTEs:**

1. **Non-Recursive CTEs:**
   * A CTE that is evaluated once and used by the main query.
   * Efficient for reusable subqueries or simplifying complex SELECT statements.
2. **Recursive CTEs:**
   * Designed to work with hierarchical data (e.g., organizational structures).
   * Composed of two parts:
     + **Base case:** Defines the starting point.
     + **Recursive case:** Refers to the CTE itself and executes repeatedly.

### **Recursive CTE**

Syntax:

WITH RECURSIVE cte\_name AS (

-- Base case: initial query

SELECT initial\_column(s)

FROM table

WHERE condition

UNION ALL

-- Recursive case: repeats until no more rows are returned

SELECT column(s)

FROM table

JOIN cte\_name ON condition

)

SELECT \* FROM cte\_name;

**Key Concepts of Recursive CTEs:**

* **Base case:** Defines the anchor data for recursion.
* **Recursive case:** Joins the CTE to fetch subsequent data, repeating the process until no new rows are returned.
* **Termination condition:** Stops when no more data is found.

#### **Key Considerations:**

* **Performance:** Recursive CTEs can be resource-intensive, so it's important to monitor performance, especially for large datasets.
* **Termination:** To avoid infinite loops, ensure the recursion halts when no new data is found.
* **Max Recursion Limit:** MySQL sets a default recursion limit of 1000; you can adjust this if necessary with SET MAX\_RECURSION.

#### **When to Use CTEs:**

* To **simplify complex queries** by breaking them into modular components.
* For **hierarchical data** that requires recursion.
* To **reuse logic** without repeating subqueries.

### **Temporary Table Operations in MySQL**

### 

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Syntax** | **Explanation** | **Example** |
| **Create Temporary Table** | CREATE TEMPORARY TABLE table\_name (column1 datatype, column2 datatype, ...); | Creates a temporary table that exists only within the current session. It will be dropped automatically when the session ends. | CREATE TEMPORARY TABLE temp\_orders (order\_id INT, product\_id INT, order\_amount DECIMAL(10, 2)); |
| **Insert Data** | INSERT INTO temp\_table\_name (column1, column2, ...) VALUES (value1, value2, ...); | Inserts rows of data into the temporary table. Can insert specific values or select from an existing table. | INSERT INTO temp\_orders (order\_id, product\_id, order\_amount) VALUES (1, 101, 50.00), (2, 102, 75.00); |
| **Insert from Another Table** | INSERT INTO temp\_table\_name (column1, column2, ...) SELECT column1, column2 FROM another\_table; | Populates the temporary table by selecting data from another table. | INSERT INTO temp\_orders (order\_id, product\_id, order\_amount) SELECT order\_id, product\_id, order\_amount FROM orders WHERE order\_date >= '2025-01-01'; |
| **Query Data** | SELECT \* FROM temp\_table\_name; | Queries the data stored in the temporary table. | SELECT \* FROM temp\_orders; |
| **Update Data** | UPDATE temp\_table\_name SET column1 = value1, column2 = value2 WHERE condition; | Updates specific rows in the temporary table that meet a certain condition. | UPDATE temp\_orders SET order\_amount = 60.00 WHERE order\_id = 1; |
| **Delete Data** | DELETE FROM temp\_table\_name WHERE condition; | Deletes specific rows from the temporary table based on a condition. | DELETE FROM temp\_orders WHERE order\_id = 2; |
| **Truncate Table** | TRUNCATE TABLE temp\_table\_name; | Deletes all rows in the temporary table but keeps the table structure intact (resets AUTO\_INCREMENT if applicable). | TRUNCATE TABLE temp\_orders; |
| **Drop Temporary Table** | DROP TEMPORARY TABLE IF EXISTS temp\_table\_name; | Explicitly drops the temporary table (though it is automatically dropped at the end of the session). | DROP TEMPORARY TABLE IF EXISTS temp\_orders; |

### **Key Differences between Temporary and Regular Tables:**

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Temporary Tables** | **Regular Tables** |
| **Visibility** | Visible only to the session that created it | Visible to all sessions/users |
| **Persistence** | Automatically dropped at the session's end | Remains in the database until explicitly dropped |
| **Auto-Increment** | Resets if applicable after truncation | Continues to increment as usual |
| **Triggers** | Does not activate triggers | Activates triggers if defined |

### **Stuff in MySQL**

In SQL Server, the **STUFF** function is used to insert a string into another string by deleting a specified number of characters from a starting position and then inserting a new string. The general syntax is:

STUFF(string, start\_position, length, new\_string)

* **string**: The original string to be modified.
* **start\_position**: The position where deletion begins (1-based index).
* **length**: The number of characters to delete starting from start\_position.
* **new\_string**: The string to be inserted.

Key Points:

* **Deletes characters** from the specified position.
* **Inserts a new string** at the specified position.
* If any argument is **NULL**, the result is also **NULL**.

The **STUFF** function is useful for modifying strings, concatenating values, or formatting strings within queries.

Tab 3

#### **Descriptive Statistics**

**Description**: Descriptive statistics involve summarizing and organizing data in a way that makes it easier to understand. These measures are used to describe the basic features of the data and provide simple summaries about the sample.

**Mean**: The average of all data points in the dataset. Calculated by summing all values and dividing by the number of values.

**Median**: The middle value when the data points are arranged in ascending or descending order. The median is less sensitive to outliers compared to the mean.

**Mode**: The value that appears most frequently in the dataset. A dataset may have no mode, one mode, or multiple modes (bimodal, multimodal).

**Variance and Standard Deviation**: These two measures describe the spread of the data points. Variance measures how far each data point is from the mean, while standard deviation is the square root of variance, making it easier to interpret.

Descriptive statistics help us quickly assess the distribution and spread of the data and understand key characteristics without performing complex analysis.

**Link to Example**: [Descriptive Statistics in Python Example](https://github.com/Vachas289/pythoncoding/blob/Statistics/Descriptive%20Statistics%20with%20logic.ipynb)

[Descriptive Statistics in Excel](https://docs.google.com/spreadsheets/d/1JlXEIFhawcmQzlxHKWKoOc0_tkrOdAbsMFs6-1y_lXs/edit?usp=sharing)

#### **Means**

Here are the formulas for each type of mean:

**Trimmed Mean**

* A trimmed mean is calculated by removing a certain percentage of the lowest and highest values from a dataset before calculating the average. This helps reduce the impact of outliers on the mean.

**Weighted Mean**

* A weighted mean gives different importance (weights) to each data point in the dataset. Values with higher weights contribute more to the average than those with lower weights.

**Geometric Mean**

* The geometric mean is the average of a set of products or rates. It is calculated by multiplying all the values together and then taking the n-th root (where n is the number of values).

**Harmonic Mean**

* The harmonic mean is used when the dataset involves rates or ratios. It is the reciprocal of the arithmetic mean of the reciprocals of the values.

**Moving Average**

* A moving average is a statistical method used to analyze data by creating averages of different subsets of the full dataset. It helps smooth out short-term fluctuations and highlight longer-term trends.

**Link to Example**: [Types of Mean Example](https://github.com/Vachas289/pythoncoding/blob/Statistics/Types%20of%20mean.ipynb)

#### **Inferential Statistics**

**Description**: Inferential statistics allow us to make predictions or inferences about a population based on a sample. We use techniques like hypothesis testing, regression, and confidence intervals to draw conclusions from data. These methods are essential for making decisions based on limited information.

**Sampling**: We typically take a subset (sample) from a larger population. Inferential statistics use this sample to estimate or infer characteristics of the entire population.

**Confidence Interval**: A range of values that is likely to contain the population parameter (like the population mean) with a certain level of confidence (typically 95% or 99%).

**P-Value**: In hypothesis testing, the p-value measures the strength of evidence against the null hypothesis. A low p-value (typically less than 0.05) suggests strong evidence against the null hypothesis, while a high p-value suggests weak evidence.

**Link to Example**: [Inferential Statistics Example](https://github.com/Vachas289/pythoncoding/blob/Statistics/Hypothesis%20Testing%20using%20scipy%20and%20logic%20.ipynb)

#### **Hypothesis Testing**

**Description**: Hypothesis testing is a method of statistical inference used to test an assumption (hypothesis) about a population. It helps determine whether there is enough evidence in the sample data to support or reject the hypothesis.

**Null Hypothesis (H₀)**: A default statement or assumption that there is no effect or no difference. For example, “there is no significant difference in means between group A and group B.”

**Alternative Hypothesis (H₁)**: The hypothesis that contradicts the null hypothesis, suggesting that there is an effect or difference.

**Common Tests**:

**T-Test**: Compares the means of two groups to determine if there is a significant difference between them.

**Z-Test**: Similar to the t-test but typically used when the population variance is known or for large sample sizes.

**Chi-Square Test**: Used for categorical data to assess if there is a significant association between two variables.

**ANOVA**: Analyzes the variance among multiple groups to check for significant differences in their means.

**P-Value**: The probability of obtaining results at least as extreme as the observed results, assuming the null hypothesis is true. A low p-value (below 0.05) suggests strong evidence against the null hypothesis.

**Link to Example**: [Hypothesis Testing with Case Examples](https://github.com/Vachas289/pythoncoding/blob/Statistics/Hypothesis%20Testing%20using%20scipy%20and%20logic%20.ipynb)

#### **Probability Distributions**

**Description**: Probability distributions describe how the values of a random variable are distributed and define the likelihood of different outcomes. These distributions are used to model random phenomena and calculate probabilities for various events.

**Normal Distribution**: A symmetric, bell-shaped distribution that is defined by the mean and standard deviation. The 68-95-99.7 rule states that 68% of data points lie within one standard deviation of the mean, 95% within two, and 99.7% within three.

**Binomial Distribution**: Used to model situations where there are two possible outcomes (success or failure) in a fixed number of trials. Examples include flipping a coin or passing a test.

**Poisson Distribution**: Used to model the number of events occurring within a fixed interval of time or space, typically when events happen randomly and independently.

**Exponential Distribution**: Models the time between events in a Poisson process, such as the time between customer arrivals at a store.

**Bernoulli and Geometric Distributions**: Models for binary outcomes (e.g., success/failure) and the number of trials needed to get the first success, respectively.

**Link to Example**: [Probability Distributions in Python](https://github.com/Vachas289/pythoncoding/blob/Statistics/Uniform%20Distribution.ipynb)

#### **Correlation**

**Description**: Correlation is a statistical measure that describes the strength and direction of the relationship between two variables. It is used to assess how well two variables move together.

**Pearson Correlation**: Measures the linear relationship between two continuous variables. The value ranges from -1 (perfect negative correlation) to +1 (perfect positive correlation), with 0 meaning no correlation.

**Spearman Rank Correlation**: A non-parametric measure that evaluates the monotonic relationship between two variables. It is used when data is not normally distributed or when dealing with ordinal data (e.g., ranks). It is less sensitive to outliers than Pearson correlation.

**Caution**: Correlation does not imply causation. A strong correlation between two variables means they move together, but it doesn't mean that one causes the other.

**Link to Example**: [Correlation Example in Python](https://github.com/Vachas289/pythoncoding/blob/Statistics/Correlation.ipynb)