**Lab 1 Hands On**

1. In Azure Data Studio, create a new query (you can do this from the **File** menu or on the *welcome* page)
2. In the new **SQLQuery\_…** pane, ensure that the **AdventureWorks** database is selected at the top of the query pane. If not, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

SELECT \* FROM SalesLT.Product;

1. Use the **⏵Run** button to run the query, and and after a few seconds, review the results, which includes all columns for all products.
2. In the query editor, modify the query as follows:
3. SELECT Name, StandardCost, ListPrice

FROM SalesLT.Product;

1. Use the **⏵Run** button to re-run the query, and and after a few seconds, review the results, which this time include only the **Name**, **StandardCost**, and **ListPrice** columns for all products.
2. Modify the query as shown below to include an expression that results in a calculated column, and then re-run the query:
3. SELECT Name, ListPrice - StandardCost

FROM SalesLT.Product;

1. Note that the results this time include the **Nam**
2. **e** column and an unnamed column containing the result of subtracting the **StandardCost** from the **ListPrice**.
3. Modify the query as shown below to assign names to the columns in the results, and then re-run the query.
4. SELECT Name AS ProductName, ListPrice - StandardCost AS Markup

FROM SalesLT.Product;

Note that the results now include columns named **ProductName** and **Markup**. The **AS** keyword has been used to assign an *alias* for each column in the results.

1. Replace the existing query with the following code, which also includes an expression that produces a calculated column in the results:
2. SELECT ProductNumber, Color, Size, Color + ', ' + Size AS ProductDetails

FROM SalesLT.Product;

1. Run the query, and note that the **+** operator in the calculated **ProductDetails** column is used to *concatenate* the **Color** and **Size** column values (with a literal comma between them). The behavior of this operator is determined by the data types of the columns - had they been numeric values, the **+** operator would have *added* them. Note also that some results are *NULL* - we'll explore NULL values later in this lab.
2. Run the following query, which includes searched **CASE** that uses an **IS NULL** expression to check for NULL **SellEndDate** values.
3. SELECT Name,
4. CASE
5. WHEN SellEndDate IS NULL THEN 'Currently for sale'
6. ELSE 'No longer available'
7. END AS SalesStatus

FROM SalesLT.Product;

The previous query used a searched **CASE** expression, which begins with a **CASE** keyword, and includes one or more **WHEN…THEN** expressions with the values and predicates to be checked. An **ELSE** expression provides a value to use if none of the **WHEN** conditions are matched, and the **END** keyword denotes the end of the **CASE** expression, which is aliased to a column name for the result using an **AS** expression.

In some queries, it's more appropriate to use a simple **CASE** expression that applies multiple **WHERE…THEN** predictes to the same value.

Run the following query to see an example of a simple **CASE** expression that produced different results depending on the **Size** column value.

1. SELECT Name,
2. CASE Size
3. WHEN 'S' THEN 'Small'
4. WHEN 'M' THEN 'Medium'
5. WHEN 'L' THEN 'Large'
6. WHEN 'XL' THEN 'Extra-Large'
7. ELSE ISNULL(Size, 'n/a')
8. END AS ProductSize

FROM SalesLT.Product;

**Lab 2 Hands On**

1. Modify the existing query to return the **Name** and **ListPrice** of all products:
2. SELECT Name, ListPrice

FROM SalesLT.Product;

1. Run the query and note that the results are presented in no particular order.
2. Modify the query to add an **ORDER BY** clause that sorts the results by **Name**, as shown here:
3. SELECT Name, ListPrice
4. FROM SalesLT.Product

ORDER BY Name;

1. Run the query and review the results. This time the products are listed in alphabetical order by **Name**.
2. Modify the query as shown below to sort the results by **ListPrice**.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product

ORDER BY ListPrice;

1. Run the query and note that the results are listed in ascending order of **ListPrice**. By default, the **ORDER BY** clause applies an ascending sort order to the specified field.
2. Modify the query as shown below to sort the results into descending order of **ListPrice**.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product

ORDER BY ListPrice DESC;

1. Run the query and note that the results now show the most expensive items first.
2. Modify the query as shown below to sort the results into descending order of **ListPrice**, and then into ascending order of **Name**.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product

ORDER BY ListPrice DESC, Name ASC;

1. Run the query and review the results. Note that they are sorted into descending order of **ListPrice**, and each set of products with the same price is sorted in ascending order of **Name**.

## Restrict results using TOP

Sometimes you only want to return a specific number of rows. For example, you might want to find the twenty most expensive products.

1. Modify the existing query to return the **Name** and **ListPrice** of all products:
2. SELECT TOP (20) Name, ListPrice
3. FROM SalesLT.Product

ORDER BY ListPrice DESC;

1. Run the query and note that the results contain the first twenty products in descending order of **ListPrice**. Typically, you include an **ORDER BY** clause when using the **TOP** parameter; otherwise the query just returns the first specified number of rows in an arbitrary order.
2. Modify the query to add the **WITH TIES** parameter as shown here, and re-run it.
3. SELECT TOP (20) WITH TIES Name, ListPrice
4. FROM SalesLT.Product

ORDER BY ListPrice DESC;

1. This time, there are 21 rows in the results, because there are multiple products that share the same price, one of which wasn't included when ties were ignored by the previous query.
2. Modify the query to add the **PERCENT** parameter as shown here, and re-run it.
3. SELECT TOP (20) PERCENT WITH TIES Name, ListPrice
4. FROM SalesLT.Product

ORDER BY ListPrice DESC;

1. Note that this time the results contain the 20% most expensive products.

## Retrieve pages of results with OFFSET and FETCH

User interfaces and reports often present large volumes of data as pages, you make them easier to navigate on a screen.

1. Modify the existing query to return product **Name** and **ListPrice** values:
2. SELECT Name, ListPrice
3. FROM SalesLT.Product

ORDER BY Name OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;

1. Run the query and note the effect of the **OFFSET** and **FETCH** parameters of the **ORDER BY** clause. The results start at the 0 position (the beginning of the result set) and include only the next 10 rows, essentially defining the first page of results with 10 rows per page.

## Use the ALL and DISTINCT options

Often, multiple rows in a table may contain the same values for a given subset of fields. For example, a table of products might contain a **Color** field that identifies the color of a given product. It's not unreasonable to assume that there may be multiple products of the same color. Similarly, the table might contain a **Size** field; and again it's not unreasonable to assume that there may be multiple products of the same size; or even multiple products with the same combination of size and color.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the welcome page).
2. In the new **SQLQuery\_…** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:
4. SELECT Color

FROM SalesLT.Product;

1. Use the **⏵Run** button to run the query, and and after a few seconds, review the results, which includes the color of each product in the table.
2. Modify the query as follows, and re-run it.
3. SELECT ALL Color

FROM SalesLT.Product;

The results should be the same as before. The **ALL** parameter is the default behavior, and is applied implicitly to return a row for every record that meets the query criteria.

1. Modify the query to replace **ALL** with **DISTINCT**, as shown here:
2. SELECT DISTINCT Color

FROM SalesLT.Product;

1. Run the modified query and note that the results include one row for each unique **Color** value. This ability to remove duplicates from the results can often be useful - for example to retrieve values in order to populate a drop-down list of color options in a user interface.
2. Modify the query to add the **Size** field as shown here:
3. SELECT DISTINCT Color, Size

FROM SalesLT.Product;

1. Run the modified query and note that it returns each unique combination of color and size.

## Filter results with the WHERE clause

Most queries for application development or reporting involve filtering the data to match specified criteria. You typically apply filtering criteria as a predicate in a **WHERE** clause of a query.

1. In Azure Data Studio, replace the existing query with the following code:
2. SELECT Name, Color, Size
3. FROM SalesLT.Product
4. WHERE ProductModelID = 6

ORDER BY Name;

1. Run the query and review the results, which contain the **Name**, **Color**, and **Size** for each product with a **ProductModelID** value of 6 (this is the ID for the HL Road Frame product model, of which there are multiple variants).
2. Replace the query with the following code, which uses the not equal (<>) operator, and run it.
3. SELECT Name, Color, Size
4. FROM SalesLT.Product
5. WHERE ProductModelID <> 6

ORDER BY Name;

1. Review the results, noting that they contain all products with a **ProductModelID** other than **6**.
2. Replace the query with the following code, and run it.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product
5. WHERE ListPrice > 1000.00

ORDER BY ListPrice;

1. Review the results, noting that they contain all products with a **ListPrice** greater than 1000.00.
2. Modify the query as follows, and run it.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product

WHERE Name LIKE 'HL Road Frame %';

1. Review the results, noting that the **LIKE** operator enables you to match string patterns. The **%** character in the predicate is a wildcard for one or more characters, so the query returns all rows where the **Name** is HL Road Frame followed by any string.

The **LIKE** operator can be used to define complex pattern matches based on regular expressions, which can be useful when dealing with string data that follows a predictable pattern.

1. Modify the query as follows, and run it.
2. SELECT Name, ListPrice
3. FROM SalesLT.Product

WHERE SellEndDate IS NOT NULL;

1. Note that to filter based on NULL values you must use **IS NULL** (or **IS NOT NULL**) you cannot compare a NULL value using the **=** operator.
2. Now try the following query, which uses the **BETWEEN** operator to define a filter based on values within a defined range.
3. SELECT Name
4. FROM SalesLT.Product

WHERE SellEndDate BETWEEN '2006/1/1' AND '2006/12/31';

1. Review the results, which contain products that the company stopped selling in 2006.
2. Run the following query, which retrieves products with a **ProductCategoryID** value that is in a specified list.
3. SELECT ProductCategoryID, Name, ListPrice
4. FROM SalesLT.Product

WHERE ProductCategoryID IN (5,6,7);

1. Now try the following query, which uses the **AND** operator to combine two criteria.
2. SELECT ProductCategoryID, Name, ListPrice, SellEndDate
3. FROM SalesLT.Product

WHERE ProductCategoryID IN (5,6,7) AND SellEndDate IS NULL;

1. Try the following query, which filters the results to include rows that match one (or both) of two criteria.
2. SELECT Name, ProductCategoryID, ProductNumber
3. FROM SalesLT.Product

WHERE ProductNumber LIKE 'FR%' OR ProductCategoryID IN (5,6,7);

**Lab 3 Hands On**

## Use inner joins

An inner join is used to find related data in two tables. For example, suppose you need to retrieve data about a product and its category from the **SalesLT.Product** and **SalesLT.ProductCategory** tables. You can find the relevant product category record for a product based on its **ProductCategoryID** field; which is a foreign-key in the product table that matches a primary key in the product category table.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the welcome page).
2. In the new **SQLQuery\_…** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:
4. SELECT SalesLT.Product.Name As ProductName, SalesLT.ProductCategory.Name AS Category
5. FROM SalesLT.Product
6. INNER JOIN SalesLT.ProductCategory

ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;

1. Use the **⏵Run** button to run the query, and and after a few seconds, review the results, which include the **ProductName** from the products table and the corresponding **Category** from the product category table. Because the query uses an **INNER** join, any products that do not have corresponding categories, and any categories that contain no products are omitted from the results.
2. Modify the query as follows to remove the **INNER** keyword, and re-run it.
3. SELECT SalesLT.Product.Name As ProductName, SalesLT.ProductCategory.Name AS Category
4. FROM SalesLT.Product
5. JOIN SalesLT.ProductCategory

ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;

The results should be the same as before. **INNER** joins are the default kind of join.

1. Modify the query to assign aliases to the tables in the **JOIN** clause, as shown here:
2. SELECT p.Name As ProductName, c.Name AS Category
3. FROM SalesLT.Product AS p
4. JOIN SalesLT.ProductCategory As c

ON p.ProductCategoryID = c.ProductCategoryID;

## Use outer joins

An outer join is used to retrieve all rows from one table, and any corresponding rows from a related table. In cases where a row in the outer table has no corresponding rows in the related table, NULL values are returned for the related table fields. For example, suppose you want to retrieve a list of all customers and any orders they have placed, including customers who have registered but never placed an order.

1. Replace the existing query with the following code:
2. SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
3. FROM SalesLT.Customer AS c
4. LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
5. ON c.CustomerID = oh.CustomerID

ORDER BY c.CustomerID;

1. Run the query and note that the results contain data for every customer. If a customer has placed an order, the order number is shown. Customers who have registered but not placed an order are shown with a NULL order number.

Note the use of the **LEFT** keyword. This identifies which of the tables in the join is the outer table (the one from which all rows should be preserved). In this case, the join is between the **Customer** and **SalesOrderHeader** tables, so a **LEFT** join designates **Customer** as the outer table. Had a **RIGHT** join been used, the query would have returned all records from the **SalesOrderHeader** table and only matching data from the **Customer\*\*table (in other words, all orders including those for which there was no matching customer record). You can also use a \*FULL** outer join to preserve unmatched rows from \*both sides of the join (all customers, including those who haven't placed an order; and all orders, including those with no matching customer), though in practice this is used less frequently.

1. Modify the query to remove the **OUTER** keyword, as shown here:
2. SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
3. FROM SalesLT.Customer AS c
4. LEFT JOIN SalesLT.SalesOrderHeader AS oh
5. ON c.CustomerID = oh.CustomerID

ORDER BY c.CustomerID;

1. Run the query and review the results, which should be the same as before. Using the **LEFT** (or **RIGHT**) keyword automatically identifies the join as an **OUTER** join.

**Lab 4 Hands On**

# **Get Started with Transact-SQL**

In this lab, you will use some basic SELECT queries to retrieve data from the **AdventureWorks** database.

## Explore the AdventureWorks database

We'll use the **AdventureWorks** database in this lab, so let's start by exploring it in Azure Data Studio.

1. Start Azure Data Studio, and in the **Connections** tab, select the **AdventureWorks** connection by clicking on the arrow just to the left of the name. This will connect to the SQL Server instance and show the objects in the **AdventureWorks** database.
2. Expand the **Tables** folder to see the tables that are defined in the database. Note that there are a few tables in the **dbo** schema, but most of the tables are defined in a schema named **SalesLT**.
3. Expand the **SalesLT.Product** table and then expand its **Columns** folder to see the columns in this table. Each column has a name, a data type, an indication of whether it can contain null values, and in some cases an indication that the columns is used as a primary key (PK) or foreign key (FK).
4. Right-click the **SalesLT.Product** table and use the **SELECT TOP (1000)** option to create and run a new query script that retrieves the first 1000 rows from the table.
5. Review the query results, which consist of 1000 rows - each row representing a product that is sold by the fictitious Adventure Works Cycles company.
6. Close the **SQLQuery\_1** pane that contains the query and its results.
7. Explore the other tables in the database, which contain information about product details, customers, and sales orders. The tables are related through primary and foreign keys, as shown here (you may need to resize the pane to see them clearly):

## Use SELECT queries to retrieve data

Now that you've had a chance to explore the **AdventureWorks** database, it's time to dig a little deeper into the product data it contains by querying the **Product** table.

1. In Azure Data Studio, create a new query (you can do this from the **File** menu or on the welcome page).
2. In the new **SQLQuery\_…** pane, ensure that the **AdventureWorks** database is selected at the top of the query pane. If not, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:

SELECT \* FROM SalesLT.Product;

1. Use the **⏵Run** button to run the query, and and after a few seconds, review the results, which includes all columns for all products.
2. In the query editor, modify the query as follows:
3. SELECT Name, StandardCost, ListPrice

FROM SalesLT.Product;

1. Use the **⏵Run** button to re-run the query, and and after a few seconds, review the results, which this time include only the **Name**, **StandardCost**, and **ListPrice** columns for all products.
2. Modify the query as shown below to include an expression that results in a calculated column, and then re-run the query:
3. SELECT Name, ListPrice - StandardCost

FROM SalesLT.Product;

1. Note that the results this time include the **Name** column and an unnamed column containing the result of subtracting the **StandardCost** from the **ListPrice**.
2. Modify the query as shown below to assign names to the columns in the results, and then re-run the query.
3. SELECT Name AS ProductName, ListPrice - StandardCost AS Markup

FROM SalesLT.Product;

1. Note that the results now include columns named **ProductName** and **Markup**. The **AS** keyword has been used to assign an alias for each column in the results.
2. Replace the existing query with the following code, which also includes an expression that produces a calculated column in the results:
3. SELECT ProductNumber, Color, Size, Color + ', ' + Size AS ProductDetails

FROM SalesLT.Product;

1. Run the query, and note that the **+** operator in the calculated **ProductDetails** column is used to concatenate the **Color** and **Size** column values (with a literal comma between them). The behavior of this operator is determined by the data types of the columns - had they been numeric values, the **+** operator would have added them. Note also that some results are NULL - we'll explore NULL values later in this lab.

**Lab 5 Hands On**

## Use simple subqueries

A subquery is a query that is nested within another query. The subquery is often referred to as the inner query, and the query within which it is nested is referred to as the outer query.

1. Start Azure Data Studio, and create a new query (you can do this from the **File** menu or on the welcome page).
2. In the new **SQLQuery\_…** pane, use the **Connect** button to connect the query to the **AdventureWorks** saved connection.
3. In the query editor, enter the following code:
4. SELECT MAX(UnitPrice)

FROM SalesLT.SalesOrderDetail;

1. Use the **⏵Run** button to run the query, and and after a few seconds, review the results, which consists of the maximum **UnitPrice** in the **SalesLT.SalesOrderDetail** (the highest price for which any individual product has been sold).
2. Modify the query as follows to use the query you just ran as a subquery in an outer query that retrieves products with a **ListPrice** higher than the maximum selling price.
3. SELECT Name, ListPrice
4. FROM SalesLT.Product
5. WHERE ListPrice >
6. (SELECT MAX(UnitPrice)

FROM SalesLT.SalesOrderDetail);

1. Run the query and review the results, which include all products that have a **listPrice** that is higher than the maximum price for which any product has been sold.
2. Replace the existing query with the following code:
3. SELECT DISTINCT ProductID
4. FROM SalesLT.SalesOrderDetail

WHERE OrderQty >= 20;

**Lab 6 Hands on (Powershell)**

Powershell - Create Folder

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**New-Item** cmdlet is used to create a directory by passing the path using -Path as path of the directory and -ItemType as Directory.

Example

In this example, we'll create a folder in D:\Temp\ with name "Test Folder"

Type the following command in PowerShell ISE Console

New-Item -Path 'D:\temp\Test Folder' -ItemType Directory

Output

You will see the following output.

Directory: D:\temp

Mode           LastWriteTime Length Name

----            ------------- ------ ----

d----      4/3/2018   7:06 PM        Test Folder

Powershell - Create File

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**New-Item** cmdlet is used to create a file by passing the path using -Path as path of the file and -ItemType as File.

Example

In this example, we'll create a file in D:\Temp\Test Folder with name "Test File.txt"

Type the following command in PowerShell ISE Console

New-Item -Path 'D:\temp\Test Folder\Test File.txt' -ItemType File

Output

You will see the following output.

Directory: D:\temp

Mode            LastWriteTime Length Name

----            ------------- ------ ----

-a---      4/3/2018   7:14 PM      0  Test File.txt

Powershell - Copy Folder

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Copy-Item** cmdlet is used to copy a directory by passing the path of the directory to be copied and destination path where the folder is to be copied.

Example 1

In this example, we'll copy a folder D:\Temp\Test Folder as D:\Temp\Test Folder1

Type the following command in PowerShell ISE Console

Copy-Item 'D:\temp\Test Folder' 'D:\temp\Test Folder1'

You can see the Test Folder1 in Windows Explorer created.

Example 2

In this example, we'll copy a folder recursively D:\Temp\Test Folder to D:\Temp\Test Folder1

Type the following command in PowerShell ISE Console

Copy-Item 'D:\temp\Test Folder' -Destination 'D:\temp\Test Folder1'

Powershell - Copy File

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Copy-Item** cmdlet is used to copy a file by passing the path of the file to be copied and destination path where the file is to be copied.

Example 1

In this example, we'll copy a folder D:\Temp\Test Folder\Test File.txt to D:\Temp\Test Folder1

Type the following command in PowerShell ISE Console

Copy-Item 'D:\temp\Test Folder\Test File.txt' 'D:\temp\Test Folder1\Test File1.txt'

You can see the Test File1.txt in Test Folder1 with content of Test File.txt. Test Folder1 folder should be present before running this command.

Example 2

In this example, we'll copy all text file recursively D:\Temp\Test Folder to D:\Temp\Test Folder1

Type the following command in PowerShell ISE Console

Copy-Item -Filter \*.txt -Path 'D:\temp\Test Folder' -Recurse -Destination 'D:\temp\Test Folder1'

Powershell - Delete Folder

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Remove-Item** cmdlet is used to delete a directory by passing the path of the directory to be deleted.

Example 1

In this example, we'll delete a folder D:\Temp\Test Folder1

Type the following command in PowerShell ISE Console

Remove-Item 'D:\temp\Test Folder1'

You can see the Test Folder1 in Windows Explorer is deleted now.

Example 2

In this example, we'll remove the folder D:\Temp\Test Folder1 recursively. In first example, PowerShell confirms if directory is not empty. In this case, it will simply delete the item.

Type the following command in PowerShell ISE Console

Remove-Item 'D:\temp\Test Folder' -Recurse

Powershell - Delete File

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Remove-Item** cmdlet is used to delete a file by passing the path of the file to be deleted.

Example 1

In this example, we'll delete a file D:\Temp\Test Folder\Test.txt

Type the following command in PowerShell ISE Console

Remove-Item 'D:\temp\Test Folder\test.txt'

You can see the Test Folder1 in Windows Explorer is deleted now.

Example 2

In this example, we'll remove the folder D:\Temp\Test Folder recursively deleting its all files. In first example, PowerShell confirms if directory is not empty. In this case, it will simply delete the item.

Type the following command in PowerShell ISE Console

Remove-Item 'D:\temp\Test Folder' -Recurse

Powershell - Move Folder

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Move-Item** cmdlet is used to move a directory by passing the path of the directory to be moved and destination path where the folder is to be moved.

Example 1

In this example, we'll move a folder D:\Temp\Test to D:\Temp\Test1

Type the following command in PowerShell ISE Console

Move-Item D:\temp\Test D:\temp\Test1

You can see the Test directory moved to Test1 directory in Windows Explorer.

Example 2

In this example, Create a file test.txt in Test folder in D:\Temp\ and then run the same command.

Type the following command in PowerShell ISE Console

Move-Item D:\temp\Test D:\temp\Test1

Powershell - Move File

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Move-Item** cmdlet is used to move a file by passing the path of the file to be moved and destination path where the file is to be moved.

Example 1

In this example, we'll move a folder D:\Temp\Test\Test.txt to D:\Temp\Test1

Type the following command in PowerShell ISE Console

Move-Item D:\temp\Test\Test.txt D:\temp\Test1

Powershell - Rename Folder

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Rename-Item** cmdlet is used to rename a folder by passing the path of the folder to be renamed and target name.

Example 1

In this example, we'll rename a folder D:\Temp\Test to D:\Temp\Test1

Type the following command in PowerShell ISE Console

Rename-Item "D:\temp\Test Test1"

Powershell - Rename File

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Rename-Item** cmdlet is used to rename a File by passing the path of the file to be renamed and target name.

Example 1

In this example, we'll rename a folder D:\Temp\Test\test.txt to test1.txt

Type the following command in PowerShell ISE Console

Rename-Item D:\temp\Test\test.txt test1.txt

Powershell - Retrieving Item

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Get-Content** cmdlet is used to retrieve content of a file as an array.

Example 1

In this example, we'll read a file D:\Temp\Test\Test.txt

Type the following command in PowerShell ISE Console

Get-Content D:\temp\Test\test.txt

Output

You can see following output in PowerShell console.

Get-Content D:\temp\test\test.txt

;This is a test file.

Example 2

In this example, we'll read the size of the content of the file read.

Type the following command in PowerShell ISE Console

(Get-Content D:\temp\test\test.txt).length

You can see following output in PowerShell console.

(Get-Content D:\temp\test\test.txt).length

20

Powershell - Check Folder Existence

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Test-Path** cmdlet is used to check existence of a folder.

Example 1

In this example, we're having a folder test in D:\temp directory

Type the following command in PowerShell ISE Console

Test-Path D:\temp\test

Output

You can see following output in PowerShell console.

Test-Path D:\temp\test

True

Example 2

In this example, we're not having a folder named test2 in D:\temp directory

Type the following command in PowerShell ISE Console

Test-Path D:\temp\test2

Output

You can see following output in PowerShell console.

Test-Path D:\temp\test2

False

Powershell - Check File Existence

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_files_folders.htm)

Cmdlet

**Test-Path** cmdlet is used to check existence of a file.

Example 1

In this example, we're having a file test.txt in D:\temp\test directory

Type the following command in PowerShell ISE Console

Test-Path D:\temp\test\test.txt

Output

You can see following output in PowerShell console.

Test-Path D:\temp\test\test.txt

True

Example 2

In this example, we're not having a file named test2.txt in D:\temp\test directory

Type the following command in PowerShell ISE Console

Test-Path D:\temp\test\test2.txt

Output

You can see following output in PowerShell console.

Test-Path D:\temp\test\test2.txt

False

Powershell - Set System Date

[**Previous Page**](https://www.tutorialspoint.com/powershell/powershell_dates_timers.htm)

[**Next Page**](https://www.tutorialspoint.com/powershell/powershell_dates_timers.htm)

Cmdlet

**Set-Date** cmdlet is used to set System Date.

In this example, we're using Get-Date to get current date

Type the following command in PowerShell ISE Console

Get-Date

Output

You can see following output in PowerShell console.

Get-Date

Saturday, May 05, 2018 9:58:06 AM

In this example, we're using Set-Date to add one more day to current date.