In [1]:
```
print("hello world")
```

```
hello world
```

Step 1 Data Exploration & Loading

In [29]:
```
#importing dependencies

import pandas as pd

import pymysql
from sqlalchemy import create_engine

import psycopg2
```

In [4]:
```
print(pd.__version__)
```

```
2.3.2
```

In [7]:
```
df = pd.read_csv('D:\Placement Prep\Data analysis\Projects\Walmart\walmart-10k-s

df.shape
```

```
<>:1: SyntaxWarning: invalid escape sequence '\P'
<>:1: SyntaxWarning: invalid escape sequence '\P'
C:\Users\ayush\AppData\Local\Temp\ipykernel_18212\1128941245.py:1: SyntaxWarning:
invalid escape sequence '\P'
  df = pd.read_csv('D:\Placement Prep\Data analysis\Projects\Walmart\walmart-10k-
sales-datasets\Walmart.csv', encoding_errors='ignore')
```

Out[7]:
```
(10051, 11)
```

In [8]:
```
df.head()
```

Out[8]:

	invoice_id	Branch	City	category	unit_price	quantity	date	time
0	1	WALM003	San Antonio	Health and beauty	$74.69	7.0	05/01/19	13:08:00
1	2	WALM048	Harlingen	Electronic accessories	$15.28	5.0	08/03/19	10:29:00
2	3	WALM067	Haltom City	Home and lifestyle	$46.33	7.0	03/03/19	13:23:00
3	4	WALM064	Bedford	Health and beauty	$58.22	8.0	27/01/19	20:33:00
4	5	WALM013	Irving	Sports and travel	$86.31	7.0	08/02/19	10:37:00

In [9]:
```
df.describe()
```

Out[9]:

	invoice_id	quantity	rating	profit_margin
count	10051.000000	10020.000000	10051.000000	10051.000000
mean	5025.741220	2.353493	5.825659	0.393791
std	2901.174372	1.602658	1.763991	0.090669
min	1.000000	1.000000	3.000000	0.180000
25%	2513.500000	1.000000	4.000000	0.330000
50%	5026.000000	2.000000	6.000000	0.330000
75%	7538.500000	3.000000	7.000000	0.480000
max	10000.000000	10.000000	10.000000	0.570000

In [10]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10051 entries, 0 to 10050
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   invoice_id      10051 non-null  int64  
 1   Branch          10051 non-null  object 
 2   City            10051 non-null  object 
 3   category        10051 non-null  object 
 4   unit_price      10020 non-null  object 
 5   quantity        10020 non-null  float64
 6   date            10051 non-null  object 
 7   time            10051 non-null  object 
 8   payment_method  10051 non-null  object 
 9   rating          10051 non-null  float64
 10  profit_margin   10051 non-null  float64
dtypes: float64(3), int64(1), object(7)
memory usage: 863.9+ KB
```

In [11]:

```
#all duplicates
df.duplicated().sum()
```

Out[11]:

```
np.int64(51)
```

In [12]:

```
df.drop_duplicates(inplace=True)
df.duplicated().sum()
```

Out[12]:

```
np.int64(0)
```

In [13]:

```
df.shape
```

Out[13]:

```
(10000, 11)
```

In [14]:

```
df.isnull().sum()
```

Out[14]:

```
invoice_id         0
Branch             0
City               0
category           0
unit_price        31
quantity          31
date               0
time               0
payment_method     0
rating             0
profit_margin      0
dtype: int64
```

In [15]:

```
#droppping all rows with missing records
df.dropna(inplace=True)

# verify
df.isnull().sum()
```

Out[15]:

```
invoice_id        0
Branch            0
City              0
category          0
unit_price        0
quantity          0
date              0
time              0
payment_method    0
rating            0
profit_margin     0
dtype: int64
```

In [16]:

```
df.shape
```

Out[16]:

```
(9969, 11)
```

In [17]:

```
df.dtypes
```

Out[17]:

```
invoice_id          int64
Branch             object
City               object
category           object
unit_price         object
quantity          float64
date               object
time               object
payment_method     object
rating            float64
profit_margin     float64
dtype: object
```

In [19]:

```
df['unit_price'] = df['unit_price'].str.replace('$', '').astype(float)

df.head()
```

Out[19]:

	invoice_id	Branch	City	category	unit_price	quantity	date	time
0	1	WALM003	San Antonio	Health and beauty	74.69	7.0	05/01/19	13:08:00
1	2	WALM048	Harlingen	Electronic accessories	15.28	5.0	08/03/19	10:29:00
2	3	WALM067	Haltom City	Home and lifestyle	46.33	7.0	03/03/19	13:23:00
3	4	WALM064	Bedford	Health and beauty	58.22	8.0	27/01/19	20:33:00
4	5	WALM013	Irving	Sports and travel	86.31	7.0	08/02/19	10:37:00

In [20]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 9969 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   invoice_id      9969 non-null   int64  
 1   Branch          9969 non-null   object 
 2   City            9969 non-null   object 
 3   category        9969 non-null   object 
 4   unit_price      9969 non-null   float64
 5   quantity        9969 non-null   float64
 6   date            9969 non-null   object 
 7   time            9969 non-null   object 
 8   payment_method  9969 non-null   object 
 9   rating          9969 non-null   float64
 10  profit_margin   9969 non-null   float64
dtypes: float64(4), int64(1), object(6)
memory usage: 934.6+ KB
```

In [21]:

```
df.columns
```

Out[21]:

```
Index(['invoice_id', 'Branch', 'City', 'category', 'unit_price', 'quantity',
       'date', 'time', 'payment_method', 'rating', 'profit_margin'],
      dtype='object')
```

In [22]:

```
df['total'] = df['unit_price'] * df['quantity']
df.head()
```

Out[22]:

	invoice_id	Branch	City	category	unit_price	quantity	date	time
0	1	WALM003	San Antonio	Health and beauty	74.69	7.0	05/01/19	13:08:00
1	2	WALM048	Harlingen	Electronic accessories	15.28	5.0	08/03/19	10:29:00
2	3	WALM067	Haltom City	Home and lifestyle	46.33	7.0	03/03/19	13:23:00
3	4	WALM064	Bedford	Health and beauty	58.22	8.0	27/01/19	20:33:00
4	5	WALM013	Irving	Sports and travel	86.31	7.0	08/02/19	10:37:00

Fixing the column name to lower case

In [23]:
```
df.columns
```

Out[23]:
```
Index(['invoice_id', 'Branch', 'City', 'category', 'unit_price', 'quantity',
       'date', 'time', 'payment_method', 'rating', 'profit_margin', 'total'],
      dtype='object')
```

In [24]:
```
df.columns = df.columns.str.lower()
df.columns
```

Out[24]:
```
Index(['invoice_id', 'branch', 'city', 'category', 'unit_price', 'quantity',
       'date', 'time', 'payment_method', 'rating', 'profit_margin', 'total'],
      dtype='object')
```

In [25]:
```
df.shape
```

Out[25]:
```
(9969, 12)
```

In [26]:
```
df.to_csv('walmart_clean_data.csv', index=False)
```

In [30]:
```
help(create_engine)
```

```
Help on function create_engine in module sqlalchemy.engine.create:

create_engine(url: 'Union[str, _url.URL]', **kwargs: 'Any') -> 'Engine'
    Create a new :class:`_engine.Engine` instance.

    The standard calling form is to send the :ref:`URL <database_urls>` as the
    first positional argument, usually a string
    that indicates database dialect and connection arguments::

        engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/tes
t")

    .. note::

        Please review :ref:`database_urls` for general guidelines in composing
        URL strings.  In particular, special characters, such as those often
        part of passwords, must be URL encoded to be properly parsed.

    Additional keyword arguments may then follow it which
    establish various options on the resulting :class:`_engine.Engine`
    and its underlying :class:`.Dialect` and :class:`_pool.Pool`
    constructs::

        engine = create_engine(
            "mysql+mysqldb://scott:tiger@hostname/dbname",
            pool_recycle=3600,
            echo=True,
        )

    The string form of the URL is
    ``dialect[+driver]://user:password@host/dbname[?key=value..]``, where
    ``dialect`` is a database name such as ``mysql``, ``oracle``,
    ``postgresql``, etc., and ``driver`` the name of a DBAPI, such as
    ``psycopg2``, ``pyodbc``, ``cx_oracle``, etc.  Alternatively,
    the URL can be an instance of :class:`~sqlalchemy.engine.url.URL`.

    ``**kwargs`` takes a wide variety of options which are routed
    towards their appropriate components.  Arguments may be specific to
    the :class:`_engine.Engine`, the underlying :class:`.Dialect`,
    as well as the
    :class:`_pool.Pool`.  Specific dialects also accept keyword arguments that
    are unique to that dialect.   Here, we describe the parameters
    that are common to most :func:`_sa.create_engine()` usage.

    Once established, the newly resulting :class:`_engine.Engine` will
    request a connection from the underlying :class:`_pool.Pool` once
    :meth:`_engine.Engine.connect` is called, or a method which depends on it
    such as :meth:`_engine.Engine.execute` is invoked.   The
    :class:`_pool.Pool` in turn
    will establish the first actual DBAPI connection when this request
    is received.   The :func:`_sa.create_engine` call itself does **not**
    establish any actual DBAPI connections directly.

    .. seealso::

        :doc:`/core/engines`

        :doc:`/dialects/index`

        :ref:`connections_toplevel`
```

```
:param connect_args: a dictionary of options which will be
    passed directly to the DBAPI's ``connect()`` method as
    additional keyword arguments.  See the example
    at :ref:`custom_dbapi_args`.

:param creator: a callable which returns a DBAPI connection.
    This creation function will be passed to the underlying
    connection pool and will be used to create all new database
    connections. Usage of this function causes connection
    parameters specified in the URL argument to be bypassed.

    This hook is not as flexible as the newer
    :meth:`_events.DialectEvents.do_connect` hook which allows complete
    control over how a connection is made to the database, given the full
    set of URL arguments and state beforehand.

    .. seealso::

        :meth:`_events.DialectEvents.do_connect` - event hook that allows
        full control over DBAPI connection mechanics.

        :ref:`custom_dbapi_args`

:param echo=False: if True, the Engine will log all statements
    as well as a ``repr()`` of their parameter lists to the default log
    handler, which defaults to ``sys.stdout`` for output.   If set to the
    string ``"debug"``, result rows will be printed to the standard output
    as well. The ``echo`` attribute of ``Engine`` can be modified at any
    time to turn logging on and off; direct control of logging is also
    available using the standard Python ``logging`` module.

    .. seealso::

        :ref:`dbengine_logging` - further detail on how to configure
        logging.


:param echo_pool=False: if True, the connection pool will log
    informational output such as when connections are invalidated
    as well as when connections are recycled to the default log handler,
    which defaults to ``sys.stdout`` for output.   If set to the string
    ``"debug"``, the logging will include pool checkouts and checkins.
    Direct control of logging is also available using the standard Python
    ``logging`` module.

    .. seealso::

        :ref:`dbengine_logging` - further detail on how to configure
        logging.


:param empty_in_strategy:   No longer used; SQLAlchemy now uses
    "empty set" behavior for IN in all cases.

:param enable_from_linting: defaults to True.  Will emit a warning
    if a given SELECT statement is found to have un-linked FROM elements
    which would cause a cartesian product.

    .. versionadded:: 1.4
```

```
        .. seealso::

            :ref:`change_4737`

    :param execution_options: Dictionary execution options which will
        be applied to all connections.  See
        :meth:`~sqlalchemy.engine.Connection.execution_options`

    :param future: Use the 2.0 style :class:`_engine.Engine` and
        :class:`_engine.Connection` API.

        As of SQLAlchemy 2.0, this parameter is present for backwards
        compatibility only and must remain at its default value of ``True``.

        The :paramref:`_sa.create_engine.future` parameter will be
        deprecated in a subsequent 2.x release and eventually removed.

        .. versionadded:: 1.4

        .. versionchanged:: 2.0 All :class:`_engine.Engine` objects are
           "future" style engines and there is no longer a ``future=False``
           mode of operation.

        .. seealso::

            :ref:`migration_20_toplevel`

    :param hide_parameters: Boolean, when set to True, SQL statement parameters
        will not be displayed in INFO logging nor will they be formatted into
        the string representation of :class:`.StatementError` objects.

        .. versionadded:: 1.3.8

        .. seealso::

            :ref:`dbengine_logging` - further detail on how to configure
            logging.

    :param implicit_returning=True:  Legacy parameter that may only be set
        to True. In SQLAlchemy 2.0, this parameter does nothing. In order to
        disable "implicit returning" for statements invoked by the ORM,
        configure this on a per-table basis using the
        :paramref:`.Table.implicit_returning` parameter.


    :param insertmanyvalues_page_size: number of rows to format into an
     INSERT statement when the statement uses "insertmanyvalues" mode, which is
     a paged form of bulk insert that is used for many backends when using
     :term:`executemany` execution typically in conjunction with RETURNING.
     Defaults to 1000, but may also be subject to dialect-specific limiting
     factors which may override this value on a per-statement basis.

     .. versionadded:: 2.0

     .. seealso::

        :ref:`engine_insertmanyvalues`

        :ref:`engine_insertmanyvalues_page_size`
        :paramref:`_engine.Connection.execution_options.insertmanyvalues_page_siz
e`

    :param isolation_level: optional string name of an isolation level
        which will be set on all new connections unconditionally.
        Isolation levels are typically some subset of the string names
        ``"SERIALIZABLE"``, ``"REPEATABLE READ"``,
        ``"READ COMMITTED"``, ``"READ UNCOMMITTED"`` and ``"AUTOCOMMIT"``
        based on backend.

        The :paramref:`_sa.create_engine.isolation_level` parameter is
        in contrast to the
        :paramref:`.Connection.execution_options.isolation_level`
        execution option, which may be set on an individual
        :class:`.Connection`, as well as the same parameter passed to
        :meth:`.Engine.execution_options`, where it may be used to create
        multiple engines with different isolation levels that share a common
        connection pool and dialect.

        .. versionchanged:: 2.0 The
           :paramref:`_sa.create_engine.isolation_level`
           parameter has been generalized to work on all dialects which support
           the concept of isolation level, and is provided as a more succinct,
           up front configuration switch in contrast to the execution option
           which is more of an ad-hoc programmatic option.

        .. seealso::

            :ref:`dbapi_autocommit`

    :param json_deserializer: for dialects that support the
        :class:`_types.JSON`
        datatype, this is a Python callable that will convert a JSON string
        to a Python object.  By default, the Python ``json.loads`` function is
        used.

        .. versionchanged:: 1.3.7  The SQLite dialect renamed this from
           ``_json_deserializer``.

    :param json_serializer: for dialects that support the :class:`_types.JSON`
        datatype, this is a Python callable that will render a given object
        as JSON.   By default, the Python ``json.dumps`` function is used.

        .. versionchanged:: 1.3.7  The SQLite dialect renamed this from
           ``_json_serializer``.


    :param label_length=None: optional integer value which limits
        the size of dynamically generated column labels to that many
        characters. If less than 6, labels are generated as
        "_(counter)". If ``None``, the value of
        ``dialect.max_identifier_length``, which may be affected via the
        :paramref:`_sa.create_engine.max_identifier_length` parameter,
        is used instead.   The value of
        :paramref:`_sa.create_engine.label_length`
        may not be larger than that of
        :paramref:`_sa.create_engine.max_identfier_length`.

        .. seealso::
            :paramref:`_sa.create_engine.max_identifier_length`

    :param logging_name:  String identifier which will be used within
        the "name" field of logging records generated within the
        "sqlalchemy.engine" logger. Defaults to a hexstring of the
        object's id.

        .. seealso::

            :ref:`dbengine_logging` - further detail on how to configure
            logging.

            :paramref:`_engine.Connection.execution_options.logging_token`

    :param max_identifier_length: integer; override the max_identifier_length
        determined by the dialect.  if ``None`` or zero, has no effect.  This
        is the database's configured maximum number of characters that may be
        used in a SQL identifier such as a table name, column name, or label
        name. All dialects determine this value automatically, however in the
        case of a new database version for which this value has changed but
        SQLAlchemy's dialect has not been adjusted, the value may be passed
        here.

        .. versionadded:: 1.3.9

        .. seealso::

            :paramref:`_sa.create_engine.label_length`

    :param max_overflow=10: the number of connections to allow in
        connection pool "overflow", that is connections that can be
        opened above and beyond the pool_size setting, which defaults
        to five. this is only used with :class:`~sqlalchemy.pool.QueuePool`.

    :param module=None: reference to a Python module object (the module
        itself, not its string name).  Specifies an alternate DBAPI module to
        be used by the engine's dialect.  Each sub-dialect references a
        specific DBAPI which will be imported before first connect.  This
        parameter causes the import to be bypassed, and the given module to
        be used instead. Can be used for testing of DBAPIs as well as to
        inject "mock" DBAPI implementations into the :class:`_engine.Engine`.

    :param paramstyle=None: The `paramstyle <https://legacy.python.org/dev/peps/p
ep-0249/#paramstyle>`_
        to use when rendering bound parameters.  This style defaults to the
        one recommended by the DBAPI itself, which is retrieved from the
        ``.paramstyle`` attribute of the DBAPI.  However, most DBAPIs accept
        more than one paramstyle, and in particular it may be desirable
        to change a "named" paramstyle into a "positional" one, or vice versa.
        When this attribute is passed, it should be one of the values
        ``"qmark"``, ``"numeric"``, ``"named"``, ``"format"`` or
        ``"pyformat"``, and should correspond to a parameter style known
        to be supported by the DBAPI in use.

    :param pool=None: an already-constructed instance of
        :class:`~sqlalchemy.pool.Pool`, such as a
        :class:`~sqlalchemy.pool.QueuePool` instance. If non-None, this
        pool will be used directly as the underlying connection pool
        for the engine, bypassing whatever connection parameters are
        present in the URL argument. For information on constructing
        connection pools manually, see :ref:`pooling_toplevel`.

    :param poolclass=None: a :class:`~sqlalchemy.pool.Pool`
        subclass, which will be used to create a connection pool
        instance using the connection parameters given in the URL. Note
        this differs from ``pool`` in that you don't actually
        instantiate the pool in this case, you just indicate what type
        of pool to be used.

    :param pool_logging_name:  String identifier which will be used within
       the "name" field of logging records generated within the
       "sqlalchemy.pool" logger. Defaults to a hexstring of the object's
       id.

       .. seealso::

            :ref:`dbengine_logging` - further detail on how to configure
            logging.

    :param pool_pre_ping: boolean, if True will enable the connection pool
        "pre-ping" feature that tests connections for liveness upon
        each checkout.

        .. versionadded:: 1.2

        .. seealso::

            :ref:`pool_disconnects_pessimistic`

    :param pool_size=5: the number of connections to keep open
        inside the connection pool. This used with
        :class:`~sqlalchemy.pool.QueuePool` as
        well as :class:`~sqlalchemy.pool.SingletonThreadPool`.  With
        :class:`~sqlalchemy.pool.QueuePool`, a ``pool_size`` setting
        of 0 indicates no limit; to disable pooling, set ``poolclass`` to
        :class:`~sqlalchemy.pool.NullPool` instead.

    :param pool_recycle=-1: this setting causes the pool to recycle
        connections after the given number of seconds has passed. It
        defaults to -1, or no timeout. For example, setting to 3600
        means connections will be recycled after one hour. Note that
        MySQL in particular will disconnect automatically if no
        activity is detected on a connection for eight hours (although
        this is configurable with the MySQLDB connection itself and the
        server configuration as well).

        .. seealso::

            :ref:`pool_setting_recycle`

    :param pool_reset_on_return='rollback': set the
        :paramref:`_pool.Pool.reset_on_return` parameter of the underlying
        :class:`_pool.Pool` object, which can be set to the values
        ``"rollback"``, ``"commit"``, or ``None``.

        .. seealso::

            :ref:`pool_reset_on_return`
            :ref:`dbapi_autocommit_skip_rollback` - a more modern approach
            to using connections with no transactional instructions

    :param pool_timeout=30: number of seconds to wait before giving
        up on getting a connection from the pool. This is only used
        with :class:`~sqlalchemy.pool.QueuePool`. This can be a float but is
        subject to the limitations of Python time functions which may not be
        reliable in the tens of milliseconds.

        .. note: don't use 30.0 above, it seems to break with the :param tag

    :param pool_use_lifo=False: use LIFO (last-in-first-out) when retrieving
        connections from :class:`.QueuePool` instead of FIFO
        (first-in-first-out). Using LIFO, a server-side timeout scheme can
        reduce the number of connections used during non- peak   periods of
        use.   When planning for server-side timeouts, ensure that a recycle or
        pre-ping strategy is in use to gracefully   handle stale connections.

          .. versionadded:: 1.3

          .. seealso::

            :ref:`pool_use_lifo`

            :ref:`pool_disconnects`

    :param plugins: string list of plugin names to load.  See
        :class:`.CreateEnginePlugin` for background.

        .. versionadded:: 1.2.3

    :param query_cache_size: size of the cache used to cache the SQL string
     form of queries.  Set to zero to disable caching.

     The cache is pruned of its least recently used items when its size reaches
     N * 1.5.  Defaults to 500, meaning the cache will always store at least
     500 SQL statements when filled, and will grow up to 750 items at which
     point it is pruned back down to 500 by removing the 250 least recently
     used items.

     Caching is accomplished on a per-statement basis by generating a
     cache key that represents the statement's structure, then generating
     string SQL for the current dialect only if that key is not present
     in the cache.   All statements support caching, however some features
     such as an INSERT with a large set of parameters will intentionally
     bypass the cache.   SQL logging will indicate statistics for each
     statement whether or not it were pull from the cache.

     .. note:: some ORM functions related to unit-of-work persistence as well
        as some attribute loading strategies will make use of individual
        per-mapper caches outside of the main cache.


     .. seealso::

        :ref:`sql_caching`

     .. versionadded:: 1.4

    :param skip_autocommit_rollback: When True, the dialect will
```

```
            unconditionally skip all calls to the DBAPI ``connection.rollback()``
            method if the DBAPI connection is confirmed to be in "autocommit" mode.
            The availability of this feature is dialect specific; if not available,
            a ``NotImplementedError`` is raised by the dialect when rollback occurs.

            .. seealso::

                :ref:`dbapi_autocommit_skip_rollback`

            .. versionadded:: 2.0.43

         :param use_insertmanyvalues: True by default, use the "insertmanyvalues"
          execution style for INSERT..RETURNING statements by default.

          .. versionadded:: 2.0

          .. seealso::

            :ref:`engine_insertmanyvalues`
```

In [31]:

```python
#psql connection
engine_psql = create_engine("postgresql+psycopg2://postgres:0919@localhost:5432/

try:
    engine_psql
    print("Connection Successed to PSQL")
except:
    print("Unable to connect")
```

```
Connection Successed to PSQL
```

In [32]:

```python
df.to_sql(name='walmart', con=engine_psql, if_exists='replace', index=False)
```

Out[32]:

```
969
```

In [34]:

```python
df.to_csv('walmart_clean_data.csv', index=False)
```