



UMBC
TRAINING CENTERS

SQL Development in Python

TCDPRG0003-2021-05-06

Table of Contents

1. Database Design	1
Objectives	1
Introduction to Databases	2
The Relational Data Model	3
Introductory Relational Database Terminology	4
Codd's 12 Rules	5
Normalization	8
First Normal Form	9
Second Normal Form	12
Third Normal Forms	14
Exercises	15
2. Data Modeling	17
Objectives	17
The Entity Relationship Model	18
Entity Relationship Diagrams	19
Entity Relationship Diagrams (1:1)	22
Entity Relationship Diagrams (1:M)	23
Entity Relationship Diagrams (M:N)	24
A Sample Database	25
Exercises	26
3. Using MySQL	27
Objectives	27
Introduction to MySQL	28
SQL	29
The MySQL Command Line Interface	30
Getting Help	31
MySQL Workbench	36
Navigating MySQL Workbench	37
Creating a Database and Sample Tables	38
Displaying Information About Tables	40
Exercises	43
4. Data Definition Language (DDL)	45
Objectives	45
Categories of SQL Statements	46
SQL Data Types	47
The CREATE Statement	48
The DROP Statement	49
The ENUM Data Type	50
The ALTER Statement	51
Integrity Constraints	54

Domain Integrity Constraints	55
Creating a Table with Domain Constraints	56
Entity Integrity Constraints	57
Referential Integrity Constraints	59
Altering a Table's Constraints	61
Exercises	66
5. Data Manipulation Language (DML)	69
Objectives	69
DML Statements	70
The SELECT Statement	71
The ORDER BY Clause	73
The INSERT Statement	77
The DELETE Statement	80
The UPDATE Statement	81
Exercises	84
6. Transaction Control	85
Objectives	85
Transactions	86
The ROLLBACK Statement	87
The COMMIT Statement	88
The SAVEPOINT Statement	89
Exercises	92
7. SQL Operators	93
Objectives	93
Comparison Operators	94
IN and NOT IN Operators	97
BETWEEN Operator	98
The LIKE Operator	99
Logical Operators	102
IS NULL and IS NOT NULL	104
CASE Statements	106
Exercises	108
8. SQL Functions	109
Objectives	109
Introduction	110
The DISTINCT Keyword	111
Aliases	112
Miscellaneous Functions	113
Mathematical Functions	115
String Functions	118
Date Functions	121
Exercises	124
More Exercises	125

9. Joining Tables	127
Objectives	127
Joins	128
Cross Join	129
Inner joins	132
Equi-Join	133
Non-Equi Join	137
Non-Key Join	138
Self Join	140
Natural Join	141
The USING Clause	142
Outer Joins	144
Right Outer Join	145
Left Outer Join	146
Exercises	147
More Exercises	148
10. Set Operators	149
Objectives	149
Introduction	150
Selection Criteria	151
Creating Sample Data	152
Union	154
Union All	157
Exercises	158
11. SQL Sub-queries	159
Objectives	159
Introduction	160
Using a Sub-query with a DML Statement	161
Typical Sub-queries	165
Sub-query Operators	169
Standard vs. Correlated Sub-queries	170
Correlated Sub-query Example	171
Predicate Operators	173
Exercises	174
12. Groups	177
Objectives	177
SQL Statements	178
GROUP BY Clause	179
HAVING Clause	183
Order of Clauses in a SELECT Statement	187
Exercises	188
More Exercises	189
13. Stored Procedures	191

Objectives	191
Creating Stored Procedures	192
Executing Stored Procedures	194
Stored Procedures with Parameters	195
Dropping Stored Procedures	197
Exercises	198
14. More Database Objects (DML)	199
Objectives	199
More Database Objects	200
Relational Views	201
Updating a View	202
Indexes	205
Exercises	207

Chapter 1. Database Design

Objectives

- Understand the relational data model.
- Introduce relational database terminology.
- Discuss database normalization

Introduction to Databases

Using a database to deal with and organize data is part of everyday life.

- Profiles we enter on our favorite websites are stored in databases.
- Our play lists are saved in a database.
- Businesses store and retrieve customer information from databases.

A database is simply an organized collection of information that can be easily accessed and updated if necessary.

- There are several different well-established database models.
 - Network model
 - Hierarchical model
 - NoSQL model
 - Relational model

Most current and popular databases follow the relational model.

- Oracle
- MySQL
- PostgreSQL
- Microsoft SQL Server

This chapter introduces some of the important concepts behind the design of a relational database.

The Relational Data Model

Relational databases were first described by E. F. Codd in 1969.

- Articles by Codd throughout the 1970s and 1980s are still considered perfection for relational database implementations.
- His famous "Twelve Rules for Relational Databases" was published in two *Computerworld* articles in October of 1985.



- Codd's twelve rules define the principles that should be used to define an ideal database.
 - These 12 rules will be discussed in more detail shortly.

An early adopter of the relational data model was the research division of IBM.

- Their implementation of a relational database was named SEQUEL (for the Sequential English Query Language)
 - The acronym was later shortened to SQL (for the Structured Query Language.)
 - SQL, pronounced either as "S-Q-L" or "sequel", has been adopted as an ANSI/ISO standard since 1986.

Introductory Relational Database Terminology

A table is a common term used when describing the relational database model.

- A table is a collection of related items that can be thought of as a grid of columns and rows, much like a spreadsheet.
 - Rows within the table represent the records being stored such as users, profiles, or products.
 - Columns within the table describe the data within a row, such as user's name, age, and address in the user record.

For example, in a table named *Employee*, there may be any number of records.

- However, each record will have the same number of columns.
- Column names might be items like *name*, *department*, *city*, *state*, and *zipcode*.

Table 1. Employee

name	department	city	state	zipcode
Sally Baker	Finance	Alexandria	VA	22314
Josh Jacobs	Marketing	Harrisburg	PA	17101
Pat Smith	Sales	Baltimore	MD	21230

A database contains any number of tables depending upon the nature and the relationship of the data.

- The names of the tables and the names and characteristics of the columns are decided upon when the table is created.

Some alternate terminology follows.

- A table is also referred to as a relation or an entity.
- A row can be called a record, a tuple or an instance of an entity.
- A column can be called a field or an attribute.
- A Primary Key can be used to uniquely identify a row in a table.

Codd's 12 Rules

Codd's 12 rules, referred to earlier, can be found in two articles published in Computerworld on October 14th and 21st of 1985.

- The following URLs link to archived copies of the two articles.
 - ▶ https://archive.org/stream/computerworld1940unse_0#page/n234/mode/1up
 - ▶ https://archive.org/stream/computerworld1940unse_0#page/n485/mode/1up
- Although Codd's original paper consisted of 12 rules, he has since enriched the original set with many more.

The 12 rules are listed below:

- Rule 1: The Information Rule
- Rule 2: The Guaranteed Access Rule
- Rule 3: Systematic Treatment of Null Values
- Rule 4: Dynamic Online Catalog Based on the Relational Model
- Rule 5: The Comprehensive Data Sublanguage Rule
- Rule 6: The View Updating Rule
- Rule 7: High-Level Insert Update, Delete
- Rule 8: Physical Data Independence
- Rule 9: Logical Data Independence
- Rule 10: Integrity Independence
- Rule 11: Distribution Independence
- Rule 12: Nonsubversion Rule

Commercial databases typically regard the rules more as guidelines and do not implement all 12 rules.

- The focus here will be on describing the first few and leave the rest for your further research if desired.

Codd's First Rule: The Information Rule

- *"All information in a relational database is represented explicitly at the logical level in exactly one way - by values in tables."*
 - ▶ Relations can be stored in tables as matching values within several tables, no hard-coded links are necessary.

Codd's Second Rule: The Guaranteed Access Rule

- *"Each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name."*
 - ▶ This rule requires you only need to know three things to be guaranteed to locate a specific piece of data from a database.
 - ▶ None of the 12 rules explicitly requires that a row in a relation have a unique primary key, but without one you risk not getting the exact row that is desired.

Codd's Third Rule: Systematic Treatment of Null Values

- *"Null values (distinct from the empty character string or a string of blank characters or any other number) are supported in the fully relational DBMS for representing missing information in a systematic way, independent of data type."*
 - ▶ A null representation for a null numeric field should be the same representation for a null character field.
 - ▶ The handling of null values needs to be consistent when performing queries.

Codd's Fourth Rule: Dynamic Online Catalog Based on the Relation Model

- *"The database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to the interrogation as they apply to regular data."*
 - ▶ The metadata about the tables and the data store inside of them are handled in the exact same way

Codd's Fifth Rule: The Comprehensive Data Sublanguage Rule

- *"A relational system may support several languages and various modes of terminal use (for example, fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items"*
 - ▶ *Data definition*
 - ▶ *View definition*
 - ▶ *Data manipulation (interactive, and by program)*
 - ▶ *Integrity constraints*
 - ▶ *Transaction boundaries (begin, commit, rollback)"*
- The current SQL specification meets all the bullet point requirements listed as part of the fifth rule above.

Several of Codd's rules have been detailed here so that you can get a sense of motivation behind a language like SQL.

- IBM spent much research both in developing SQL and in having it be consistent with Codd's rules.

Normalization

Because data is in constant flux (especially over time), and since data in a database tables is likewise subject to change, a method was needed to avoid problems due to duplication of data values and modification of structure and content.

- This method is called normalization.
- It applies to how data is organized in a relational database.
- A database is normalized in order to:
 - Ensure data consistency;
 - Minimize redundancy;
 - Ensure consistent updating of the data; and
 - Avoid updating and deleting anomalies potentially arising when data is changed in some, rather than all expected places.

Normalization is the process of making a database compliant with the concept of a Normal Form.

- The normalization process is based on collecting an exhaustive list of all data items to be maintained in the database and starting the design with a few "superset" tables.
 - Theoretically, it may be possible (although not very practical) to begin by placing all the attributes in a single table.

There are several levels of Normal Form, and each level requires that the previous level be satisfied.

First Normal Form

Entities are reduced to first normal form (1NF) by removing repeating or multi-valued attributes to another, child entity.

- That is, make sure that the data is represented as a (proper) table that meets the following properties.
 - Property 1: Entries in columns are single-valued (atomic)
 - Property 2: Entries in columns are of the same kind
 - Property 3: Each row is unique
 - Property 4: Sequence of columns is insignificant
 - Property 5: Sequence of rows is insignificant
 - Property 6: Each column has a unique name

The most common violation of 1NF is the use of repeating columns.

- This is where multiple values of the same type are stored in multiple columns, that would violate the 6th property listed above as shown below.

Table 2. Instructor

id	instructor	course	course	course
1	Emily	C	Python	
2	Nate	Java	C++	C
3	Pat	Ruby	Haskell	Perl
4	Daniel	Python		

One might try to fix the problem by reorganizing the table, as shown next.

Table 3. Instructor

id	instructor	course
1	Emily	C, Python
2	Nate	Java, C++, C
3	Pat	Ruby, Haskell, Perl
4	Daniel	Python

While the previous table now adheres to the 6th property of 1NF, it is now violating the 1st property.

- Redesigning the table, as shown below, to adhere to the 1st property makes the table meet the 1NF requirements.
- Each row within the table is unique from all other rows, which is also a requirement of 1NF according to the 3rd property.

Table 4. Instructor

id	instructor	course
1	Emily	C
1	Emily	Python
2	Nate	Java
2	Nate	C++
2	Nate	C
3	Pat	Ruby
3	Pat	Haskell
3	Pat	Perl
4	Daniel	Python

However, we are now faced with the problem of having redundant data.

- Redundant data is repetitive column data, which is tied to other column data, such as multiple *Pat* and 3 values above
 - ▶ If the instructor name is updated, it must be updated in every row that it exists.

Redundancy can cause errors when an insert, update, or delete occurs.

Inserting a row:

- An instructor and the instructor's corresponding ID must be added together each time a course is inserted.

Deleting a row:

- If a row is deleted and it is the last remaining row for that instructor, all information concerning the instructor is lost.

Updating rows:

- If an instructor's name changes, then all rows for the instructor must be updated.

Separating the data into two separate tables can be done to remove the redundant column data as shown in the following tables.

Table 5. Instructor

id	instructor
1	Emily
2	Nate
3	Pat
4	Daniel

Table 6. Course

iid	course
1	C
1	Python
2	Java
2	C++
2	C
2	Java
3	Ruby
3	C
3	Haskell
4	Python

- The *id* can be thought of as a primary key in the *Instructor* table and the *iid* in the *Course* table would be referred to as a foreign key.

Second Normal Form

Whereas 1NF eliminates redundancies within rows, Second Normal Form (2NF) reduces redundancies within columns.

Consider the remaining redundancies in the previous *Course* table.

- Notice the repetitions of the courses Java and Perl.
 - This leaves chances for inconsistent data as it is updated.

We can eliminate these repetitions by creating two tables (*Course* and *Subject*) from the original *Course* table as shown below.

Table 7. Instructor

id	instructor
1	Emily
2	Nate
3	Pat
4	Daniel

Table 8. Course

iid	sid
1	1
1	2
2	3
2	4
2	1
3	5
3	1
3	6
4	2

Table 9. Subject

id	subject
1	C
2	Python
3	Java
4	C++
5	Ruby
6	Haskell
7	Perl

Third Normal Forms

Third Normal Form (3NF) is concerned with removing columns that are not fully dependent upon the primary key.

Suppose we have the following *Order* table.

Table 10. Order

id	customer_id	unit_price	quantity	total
23	36178	20.00	10	200.00
5	97864	9.56	5	47.80
98	69554	1.00	13	13.00
42	20701	10.00	2	20.00

Notice how the **total** field is independent of the primary key, the order number.

- In this case, we can achieve 3NF by simply eliminating the total field.
- If an application needs the total for an order, it can be computed.

There are higher order normal forms beyond 1NF, 2NF and 3NF.

- However, in most cases, achieving high levels of normalization is neither necessary nor desired.
- Most real-world databases adhere to 3NF, and few databases use anything beyond that.

Exercises

Exercise 1

Is the following table for suppliers and parts in 1NF?

- Why or why not?
- If not, how would you convert it into 1NF?

id	name	city	part	quantity
s1	Acme Suppliers	Bowie	wrench	200
s1	Acme Suppliers	Bowie	nail	1000
s1	Acme Suppliers	Bowie	screw	1000
s1	Acme Suppliers	Bowie	bolt	750
s1	Acme Suppliers	Bowie	nut	750
s1	Acme Suppliers	Bowie	hammer	500
s2	PartCo	Dundalk	drill	50
s2	PartCo	Dundalk	nail	70
s2	PartCo	Dundalk	hammer	3
s3	Suppliers-R-Us	Columbia	screw	300
s3	Suppliers-R-Us	Columbia	nail	400
s4	ABC Parts	Towson	screw	100
s4	ABC Parts	Towson	nut	75
s4	ABC Parts	Towson	bolt	120

Exercise 2

Convert your table in 1NF from problem 1 into a table that adheres to 2NF.

Exercise 3

Convert your table in 2NF from problem 2 into a table that adheres to 3NF.

Chapter 2. Data Modeling

Objectives

- Understand the entity relationship model.
- Build entity relationship diagrams.
- Define relationship cardinalities.
- Review the sample database.

The Entity Relationship Model

The entity relationship model is a conceptual data model that portrays the content of a database as entities and establishes relationships between them.

- The model is visually represented in Entity Relationship Diagrams.

The building blocks of the entity relationship model are entities, attributes, and relationships.

- Entities are conceptual objects from which data is being collected.
 - In a database, the entities correlate to the tables.
 - An instance of an entity is analogous to a row of a table.
- Attributes are properties that describe an entity and are analogous to the columns of a table.
 - Every instance of an entity must have an attribute or a collection of attributes that uniquely identifies it.
- Relationships are the associations between the entities.
 - Relationships are defined by a primary key field so that every entry is unique and can be identified using the primary key.
 - The primary key is then referenced in a second entity, at which point it is termed a foreign key, thereby creating the relationship.

Based on the two tables shown below.

- Can you identify the entities, attributes, primary and/or foreign keys?

Table 11. Course

course_id	name	unit_price	num_days	courseware_id
-----------	------	------------	----------	---------------

Table 12. Courseware

courseware_id	name	expired	in_revision	curr_rev_date	last_rev_date
---------------	------	---------	-------------	---------------	---------------

Entity Relationship Diagrams

An Entity Relationship Diagram (ERD) is a graphical data-modeling representation that helps organize a database into entities and attributes and define the relationships between the entities.

- While there are several styles of ERDs, the one that will be described here is the Information Engineering style.
- The Information Engineering style is also sometimes referred to as the "crow's feet" style.
- Entities or tables within a database and are represented by the name of the table inside a box in an ERD.

Course

- Attributes or columns within a table are commonly displayed as a list inside the entity box separated by a line from the entity they describe as shown below.

Course
*id
name
unit_price
num_days
courseware_id

- id is the primary key because it uniquely identifies an instance in the Course entity and as such is typically marked with an asterisk.

While the previous diagrams can be used to model entities and its attributes, they currently do not model the relationships between entities.

Before the symbols for diagramming the relationships are discussed, we will first look at the various types of relationships that may exist between the data within the tables of a database.

There are three basic types of relationships that may exist between two entities.

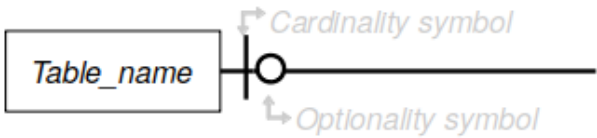

- One-to-One (1:1)
 - ▶ An example of this is an airport exists in only one city, and that city has only one airport.
 - ▶ Another example is a person has a single social security number, and each social security number belongs to a single person.
- One-to-Many (1:M)
 - ▶ An example of this is a mother that has many biological children, and each child has a single biological mother.
 - ▶ Another example is a job description that is assigned to many employees, while each employee typically has a single job title.
- Many-to-Many (M:M) or (M:N)
 - ▶ An example of this is an instructor that may be capable of teaching many courses, and each course can be taught by multiple instructors.
 - ▶ Another example of this is a tool that can be sold by many vendors, and each vendor can sell many different types of tools.

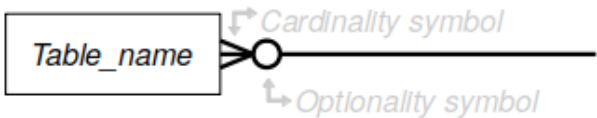
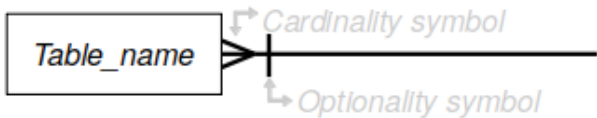
While the maximum requirement of one or many in the relationship is referred to as the cardinality, the minimum requirement of zero or one can be referred to as the optionality.

Defining a relationship between two entities starts with a solid line drawn between the two entities.

- Both the cardinality and optionality of the relationship is then able to be defined as symbols placed on the line and next to the entity that it applies to
- Each side of the relationship will typically have two symbols, for a total of 4 symbols on the line defining the relationship.
 - The symbol closest to the entity is used to define the cardinality.
 - The symbol used to define the optionality is then defined next to the cardinality on the side opposite to the entity.

The tables below show the various cardinality and optionality symbols that can be used to describe one side of the relationship.

<u>Cardinality (max) of One</u>	
<u>Optionality(min) of Zero</u>	<u>Optionality (min) of One</u>
	

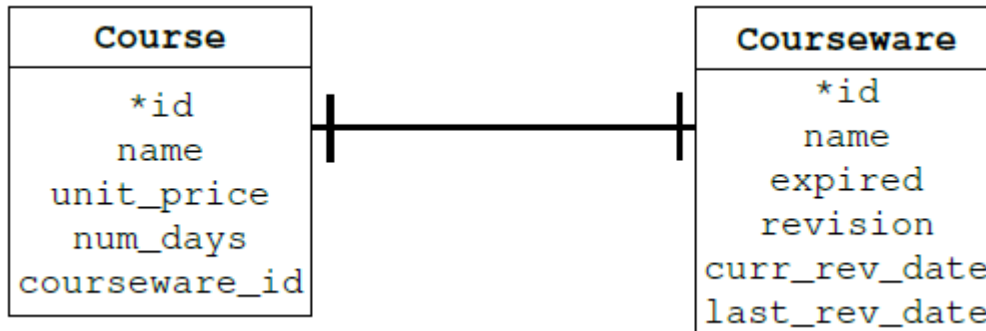
<u>Cardinality (max) of Many</u>	
<u>Optionality(min) of Zero</u>	<u>Optionality (min) of One</u>
	

- The above symbols will be used in the following examples to describe the one-to-one, one-to-many, and many-to-many relationships that one might find in a database.

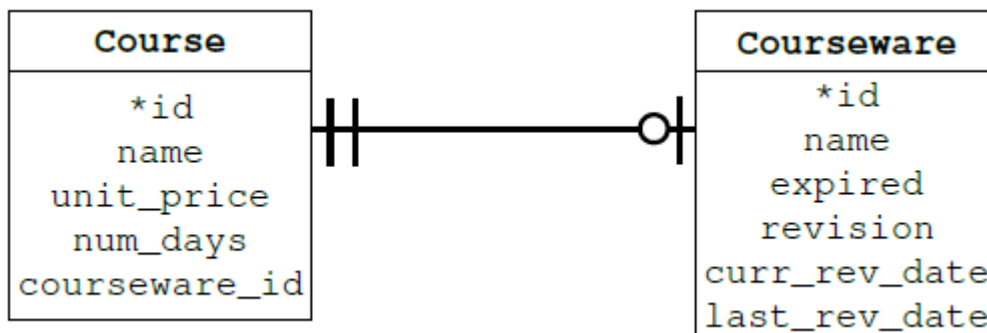
Entity Relationship Diagrams (1:1)

In the example below, the relationship between the two tables is a one-to-one relationship.

- The cardinality of the relationship is that one course entry is associated with one courseware entry.



- When the optionality of the relationship is defined, the minimum number of occurrences (zero or one) is added to the diagram as shown below:



- A **Courseware** entry is required to have a minimum and maximum of one **Course** associated with it.
- A **Course** entry may or may not have an associated **Courseware** entry (an example of this may be when the client/instructor may be providing the courseware).

Below are a few other examples of one-to-one relationships.

- Each employee is assigned his/her own desk.
- Every class consists of one, and only one, course.

Entity Relationship Diagrams (1:M)

A one-to-many relationship (1:M) is typically the most common relationship in a database.

- The cardinality of the relationship between an *Employee* and a *Team* is shown below.
 - A *Team* consists of many employees.
 - Whereas an *Employee* belongs to exactly one *Team*.



- Once again, when the optionality of the relationship is defined, the minimum number of occurrences (zero or one) is added to the diagram as shown below:



- An *Employee* entry is required to have a minimum and maximum of one *Team* associated with it.
- A *Team* entry may have 1 or more *Employee* entries associated with it.

Below are a few other examples of one-to-many relationships.

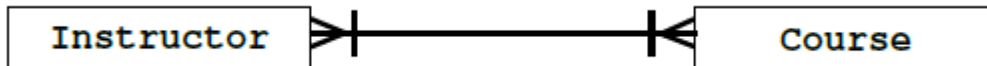
- A customer can have many orders placed within a company, but each order belongs to a single customer.
- A manager manages many employees, while each employee has a single manager.
- A street may have many houses on it, while each house only exists on a single street.

Entity Relationship Diagrams (M:N)

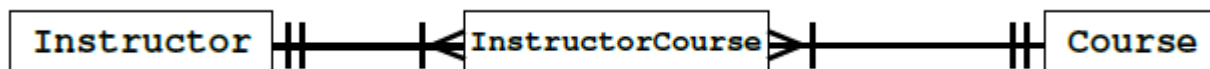
Another common relationship between entities in a database is the many-to-many relationship (M:N).

- This relationship exists when one instance of an entity has many instances of a second entity and vice versa.

The example below shows a many-to-many relationship between a table of instructors and a table of courses that they teach.

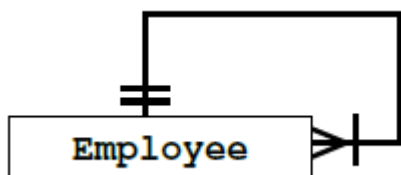


- A many-to-many relationship between two tables will typically not adhere to the rules of 2NF.
 - This type of relationship is often implemented using an intermediate table to adhere to 2NF as shown below.



Another type of relationship is a reflexive relationship where one instance in an entity refers to another instance in the same entity.

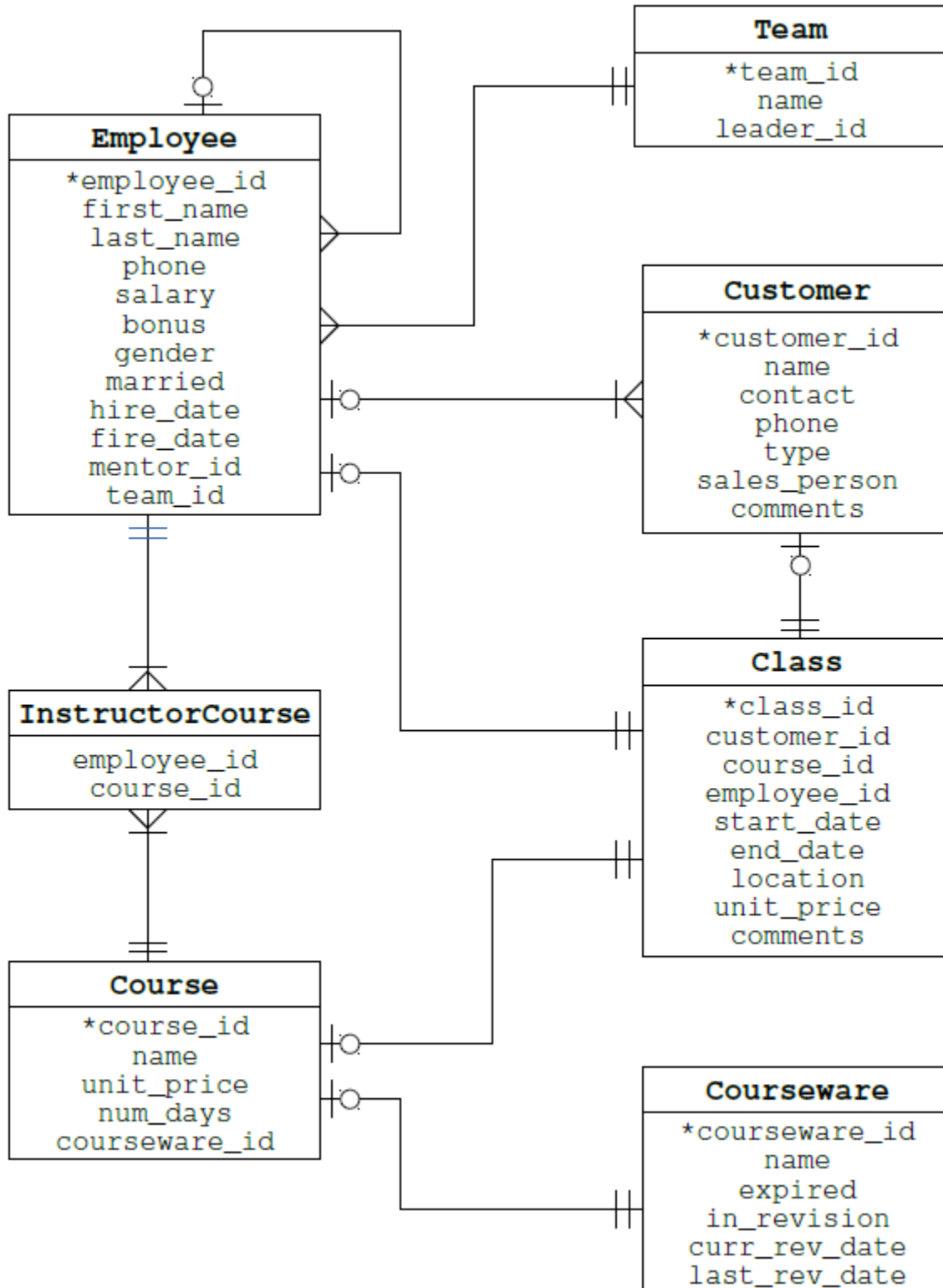
- For example, an employee may serve as a mentor for many other employees.
 - The reflexive relationship may be 1:N or M:N.



A Sample Database

The database used in this course will have seven tables representing a small training company.

- The following ERD shows the relationships between the entities.



Exercises

Exercise 1

Create an E/R Diagram for the following scenario.

- The event coordinator in our company wants to schedule events.
- Invitations to these events will be sent to our customers.
- Customers can register for events and submit full or partial payments

Chapter 3. Using MySQL

Objectives

- An Introduction to MySQL
- Defining SQL
- Getting Help in MySQL
- Navigating SQL Workbench
- Creating the Sample Database and Tables
- Describing Database and Database Objects

Introduction to MySQL

- MySQL is a **relational database management system** (RDBMS).
 - Like most RDBMSs, the language used to communicate with the database is the **Structured Query Language** (SQL).
- MySQL is open-source software that was initially released in 1995.
 - It was originally developed in Sweden by David Axmark, Allen Larsson, and Michael Widenius.
 - It was purchased by Sun Microsystems in 2008, which was acquired by Oracle in 2010.
 - ◆ The open-source version is referred to as the MySQL community server, though Oracle also provides paid versions for enterprise and cloud-based system.
- MySQL can be downloaded and installed easily on various operating systems
 - The open-source version can be downloaded from the following URL: <https://dev.mysql.com/downloads/mysql/>
 - The machines used for this course will be running the Ubuntu operating system, that will have MySQL 5.7.x installed on them.
- MySQL is a key part of the **LAMP** open source web services software stack.
 - **L**inux, **A**pache, **M**ySQL, **P**HP / **P**erl / **P**ython
- Complete documentation for MySQL can be found at the following URL:
 - <https://dev.mysql.com/doc/refman/5.7/en/>

SQL

- As mentioned in the first chapter, SQL is an ANSI/ISO standard.
 - As of this writing, the current version of the standard is SQL:2016
 - ◆ The formal name for the standard is "ISO/IEC 9075:2016 Information technology – Database languages – SQL"
- SQL is considered a declarative, not a procedural language.
 - In a procedural language, a programmer declares and initializes data and then writes procedures or functions to manipulate or extract the data.
 - In a declarative language, a programmer accesses and modifies data, but without describing how to go about accomplishing that.
 - ◆ In SQL, the user declares what to initialize, manipulate, or extract and allows the database server to determine how to execute the request most efficiently.
- While SQL is a standard, most RDBMSs have their own proprietary version of SQL that supports the standard to varying levels.
 - Additional information pertaining to MySQL's standards compliance can be found at the following URL:
 - ◆ <https://dev.mysql.com/doc/refman/5.7/en/compatibility.html>

The MySQL Command Line Interface

- Interacting with MySQL can occur using a GUI, if available, or a terminal window.
 - ▶ Within the interactive *mysql*> prompt **exit**, **quit** or **\q** can be typed to exit back to the terminal window.

```
$ mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.21-0ubuntu0.16.04.1 (Ubuntu)
Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
Affiliates. Other names may be trademarks of their respective owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

- Providing the name of a database when starting *mysql* will connect to the database if it exists.
 - ▶ Keep in mind that each student has their own database already created, whose name is the same as your username.
 - ▶ The following example uses the username of *student*.

```
$ mysql student
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.21-0ubuntu0.16.04.1 (Ubuntu)
Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

Getting Help

- At the MySQL prompt, the `help` or `\h` command can be used to access a list of the available MySQL specific commands.
- Online help documentation for MySQL is available at: <https://dev.mysql.com/doc/refman/8.0/en/>

```
mysql> help
```

```
For information about MySQL products and services, visit:
```

```
http://www.mysql.com/
```

```
For developer information, including the MySQL Reference Manual, visit:
```

```
http://dev.mysql.com/
```

```
To buy MySQL Enterprise support, training, or other products, visit:
```

```
https://shop.mysql.com/
```

```
List of all MySQL commands:
```

```
Note that all text commands must be first on line and end with ';'.
```

```
? (\?) Synonym for 'help'.
```

```
clear (\c) Clear the current input statement.
```

```
connect (\r) Reconnect to the server. Optional arguments are db and host.
```

```
delimiter (\d) Set statement delimiter.
```

```
edit (\e) Edit command with $EDITOR.
```

```
ego (\G) Send command to mysql server, display result vertically.
```

```
exit (\q) Exit mysql. Same as quit.
```

```
go (\g) Send command to mysql server.
```

```
help (\h) Display this help.
```

```
nopager (\n) Disable pager, print to stdout.
```

```
notee (\t) Don't write into outfile.
```

```
pager (\P) Set PAGER [to_pager]. Print the query results via PAGER.
```

```
print (\p) Print current command.
```

```
prompt (\R) Change your mysql prompt.
```

```
quit (\q) Quit mysql.
```

```
rehash (\#) Rebuild completion hash.
```

```
source (\.) Execute an SQL scriptfile. Takes a file name as an argument.
```

```
status (\s) Get status information from the server.
```

```
system (\!) Execute a system shell command.
```

```
tee (\T) Set outfile [to_outfile]. Append everything into given outfile.
```

```
use (\u) Use another database. Takes database name as argument.
```

```
charset (\C) Switch to another charset. Might be needed for processing binlog with multi-byte charsets.
```

```
warnings (\W) Show warnings after every statement.
```

```
nowarning (\w) Don't show warnings after every statement.
```

```
resetconnection (\x) Clean session context.
```

```
For server side help, type 'help contents'
```

```
mysql>
```

- The `help contents` command is used to show help categories as shown on the following page.

```
mysql> help contents
You asked for help about help category: "Contents"
For more information, type 'help <item>', where <item> is one of the following
categories:
  Account Management
  Administration
  Compound Statements
  Data Definition
  Data Manipulation
  Data Types
  Functions
  Functions and Modifiers for Use with GROUP BY
  Geographic Features
  Help Metadata
  Language Structure
  Plugins
  Procedures
  Storage Engines
  Table Maintenance
  Transactions
  User-Defined Functions
  Utility
mysql>
```

- As indicated above, `help Data Types` would provide more help on that category.

```
mysql> help Data Types
```

```
You asked for help about help category: "Data Types"
```

```
For more information, type 'help <item>', where <item> is one of the following topics:
```

```
AUTO_INCREMENT  
BIGINT  
BINARY  
BIT  
BLOB  
BLOB DATA TYPE  
BOOLEAN  
CHAR  
CHAR BYTE  
DATE  
DATETIME  
DEC  
DECIMAL  
DOUBLE  
DOUBLE PRECISION  
ENUM  
FLOAT  
INT  
...
```


- Additional help for any of the specific topics listed within a category can be achieved in a similar fashion.
 - ▶ The example below shows the help for the **INT**, **INTEGER**, and **TINYINT** data types listed under the category of *Data Types* from the following page.

```
mysql> help INT
Name: 'INT'
Description:
INT[(M)] [UNSIGNED] [ZEROFILL]
A normal-size integer. The signed range is -2147483648 to 2147483647.
The unsigned range is 0 to 4294967295.
URL: http://dev.mysql.com/doc/refman/5.7/en/numeric-type-overview.html
mysql> help INTEGER
Name: 'INTEGER'
Description:
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
This type is a synonym for INT.
URL: http://dev.mysql.com/doc/refman/5.7/en/numeric-type-overview.html
mysql> help TINYINT
Name: 'TINYINT'
Description:
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
A very small integer. The signed range is -128 to 127. The unsigned
range is 0 to 255.
URL: http://dev.mysql.com/doc/refman/5.7/en/numeric-type-overview.html
mysql> exit
Bye
$
```

MySQL Workbench

- There are many Graphical User Interfaces (GUIs) available for working with MySQL.
 - MySQL Workbench is the official GUI for MySQL
 - MySQL Workbench provides many capabilities
 - ◆ It is a unified visual tool for database architects, developers, and DBAs
 - ◆ It provides data modeling, SQL development, and comprehensive administration tools
 - ◆ It is available on Windows, Linux and Mac OS X
- We will focus on the SQL development features, mainly creating and executing SQL statements in the SQL Editor window
- Online help documentation for MySQL Workbench is available at: <https://dev.mysql.com/doc/workbench/en/>

Navigating MySQL Workbench

MySQL Workbench

Local instance MySQL80 x

File Edit View Query Database Server Tools Scripting Help

Navigator: SCHEMAS

Filter objects

sakila

sql_class

Tables

- class
- course
- courseware
- customer
- employee
- instructorcourse
- team

Views

Stored Procedures

Functions

sys

world

Administration Schemas

Information

Schema: sql_class

Object Info Session

Query 1 x

Limit to 1000 rows

1 • SELECT * FROM Employee;

Result Grid

employee_id	first_name	last_name	phone	salary	bonus	gender	married	hire_date	first_name
500	Morgan	Wilson	2015556298	50000	10000	M	Y	1990-06-10	NULL
501	Emily	Wilson	2015556298	50000	10000	F	Y	1995-01-01	NULL
502	Kim	Perkins	4105557985	40000	10000	F	N	1997-05-21	NULL
503	Alex	Hardin	4105557648	35000	7500	M	N	1998-04-09	NULL
508	Tina	Johnson	4105553274	35000	10000	F	N	2008-11-23	NULL
504	Rachel	Fitzgerald	3015559466	35000	5000	F	N	2000-10-30	NULL
505	Maria	Higgins	4105552105	35000	5000	F	N	2001-03-21	NULL
506	Brian	Perkins	2145554425	35000	5000	M	N	2001-03-21	NULL
507	Connie	McKinney	4105552544	35000	5000	F	Y	2001-03-21	NULL
509	Travis	Train	4105553512	25000	NULL	M	N	2004-05-31	200

Form Editor

Field Types

Query Stats

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Read Only Context Help Snippets

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	12:48:18	SELECT * FROM class LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec
2	12:48:43	SELECT * FROM Class LIMIT 0, 1000	10 row(s) returned	0.000 sec / 0.000 sec
3	12:48:58	SELECT * FROM Employee LIMIT 0, 1...	10 row(s) returned	0.000 sec / 0.000 sec

Creating a Database and Sample Tables

- The **CREATE DATABASE** *database_name* command is used to create a new database named *database_name*.
 - As mentioned earlier, In the lab environment a database with the same name as your system user name has already been created for you.
 - The **SHOW DATABASES** command can be used to show a list of available databases.
 - ◆ Note that most commands require a semicolon at the end of the command.
 - ◆ This provides for the ability of multi-line commands as we will see in various places throughout the course.

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| student |
+-----+
2 rows in set (0.00 sec)
mysql> USE student
Database changed
mysql>
```

- The **USE** command can be used to make a database the active database.
 - If *mysql* was started up with the database name as an argument, that database is automatically made the active database.
 - ◆ Next, we will use the **SOURCE** command to execute a set of SQL statements stored in several files.
- The first script will contain the SQL statements to create the sample tables whose ERDs were discussed in the previous chapter.
- The second script will contain the SQL statements to load them with some data.
 - Note that the following pages show running the commands to set up the database from a terminal session.
- All commands to create, fill, and describe the tables can be executed in the SQL Editor within MySQL Workbench.

- Your home directory contains a folder named *sqllabs* that contains all the files that pertain to this course.
 - The *sqllabs* directory is divided into several sub-directories, one of which is named *examples*.
 - ◆ The *examples* directory contains a sub-directory named *sqlscripts* that contain the scripts that will be used.
- The terminal window shown below shows the following steps:
 - Changing from the home directory to the *sqllabs* directory.
 - Starting the MySQL client with the database to connect to.
 - ◆ You will replace the use of *student* with your user name.
 - Running the script named *definetables.sql* to create the tables by sourcing it.

```
$ cd sqllabs/examples/sqlscripts/
$ mysql student
Welcome to the MySQL monitor. Commands end with ; or \g.
...
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql> SOURCE definetables.sql;
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.02 sec)
mysql>
```

- The actual commands within the script will be looked at in more detail in the coming chapters.

Displaying Information About Tables

- The **SHOW TABLES** command can be used to list the tables that are in the active database.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_student |
+-----+
| Class              |
| Course             |
| Courseware         |
| Customer           |
| Employee           |
| InstructorCourse   |
| Team               |
+-----+
7 rows in set (0.00 sec)
```

- The schema of a particular table can be displayed by using the **DESCRIBE** command.

```
mysql> DESCRIBE Employee;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| employee_id | int(11)   | YES  |     | NULL    |       |
| first_name  | char(10)  | NO   |     | NULL    |       |
| last_name   | char(10)  | YES  |     | NULL    |       |
| last_name   | char(10)  | YES  |     | NULL    |       |
| salary      | int(11)   | YES  |     | NULL    |       |
| bonus       | int(11)   | YES  |     | NULL    |       |
| gender      | char(1)   | YES  |     | NULL    |       |
| married     | char(1)   | YES  |     | NULL    |       |
| hire_date   | date      | YES  |     | NULL    |       |
| fire_date   | date      | YES  |     | NULL    |       |
| mentor_id   | int(11)   | YES  |     | NULL    |       |
| team_id     | int(11)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

- The following SQL statement displays the contents of the *Team* table (which is currently empty).

```
mysql> SELECT * FROM Team;  
Empty set (0.00 sec)
```

- The terminal window shown below runs the script named *filltables.sql* to fill all the sample tables with data.
 - The complete output is more than what is shown below.

```
mysql> SOURCE filltables.sql;
Query OK, 0 rows affected (0.00 sec)
Query OK, 5 rows affected (0.00 sec)
Records: 5 Duplicates: 0 Warnings: 0
Query OK, 4 rows affected (0.01 sec)
Records: 4 Duplicates: 0 Warnings: 0
.
.
.
Query OK, 0 rows affected (0.00 sec)
Query OK, 10 rows affected (0.00 sec)
Records: 10 Duplicates: 0 Warnings: 0
```

- The **SELECT** statement is shown once again now that the tables have data in them.

```
mysql> SELECT * FROM Team;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400 | Education | 500 |
| 401 | Management | 501 |
| 402 | Systems   | 502 |
| 403 | Sales     | 508 |
+-----+-----+-----+
4 rows in set (0.00 sec)
mysql> quit;
Bye
```

Exercises

Exercise 1

From the MySQL prompt, display the following.

- The help contents for the *Data Definition* category.
- The help contents for the *SELECT* topic.

Exercise 2

From MySQL Workbench, display the following.

- The help contents for the *Data Manipulation* category.
- The help contents for the *INSERT* topic.

Exercise 3

From the MySQL prompt, run the *definetables.sql* script to create the tables.

- From the MySQL Workbench SQL Editor window, re-run the *definetables.sql* script to show that the tables are deleted and re-created.
 - This can be verified by browsing the database in MySQL Workbench.

Exercise 4

From the MySQL Workbench SQL Editor window, execute the *filltables.sql* script to populate the tables.

- Once again, this can be verified by browsing the database in MySQL Workbench and opening a table.

Exercise 5

From the MySQL Workbench SQL Editor window, view the contents of each table in the database.

Exercise 6

Create a script, that when sourced, displays the contents of just the *Employee* and *Class* tables.

Chapter 4. Data Definition Language (DDL)

Objectives

- Categories of SQL Statements
- Datatypes
- The CREATE Statement
- The DROP Command
- The ALTER Command
- Integrity Constraints
- Entity Integrity Constraints
- Referential Integrity Constraints
- Modifying Table to Use Constraints
- Checking Constraints
- Altering Constraints

Categories of SQL Statements

- SQL statements can be divided into two types.
 - Data Definition Language (DDL) statements.
 - ◆ DDL statements provide the commands used to create, modify, or destroy databases and database objects, such as tables.
 - Data Manipulation Language (DML) statements.
 - ◆ DML statements provide the commands to insert, retrieve, and modify the data contained within the structures created by the DDL statements.
- This chapter will concentrate on the DDL statements, while the next chapter will discuss the DML statements.
- The DDL statements consists of the following three types.
 - **CREATE**, **DROP**, and **ALTER**
 - ◆ These are the types of statements used in the *definetables.sql* script from the previous chapter.

SQL Data Types

- When creating a table within a database it is important to keep in mind the following.
 - Each table is composed of a set of columns.
 - Each column is of a particular data type.
 - ◆ The types are important because of the type of data being stored and the amount of storage they use and the operations that can be performed on them.
- We saw earlier that when the `help Data Types` command is issued, a list of available data types is listed as shown in the table below.

Table 13. MySQL Data Types

AUTO_INCREMENT	DATE	INTEGER	TIME
BIGINT	DATETIME	LONGBLOB	TIMESTAMP
BINARY	DEC	LONGTEXT	TINYBLOB
BIT	DECIMAL	MEDIUMBLOB	TINYINT
BLOB	DOUBLE	MEDIUMINT	TINYTEXT
BLOB DATA TYPE	DOUBLE PRECISION	MEDIUMTEXT	VARBINARY
BOOLEAN	ENUM	SET DATA TYPE	VARCHAR
CHAR	FLOAT	SMALLINT	YEAR DATA TYPE
CHAR BYTE	INT	TEXT	—

- The use of some of the above data types are more common than others from the list.
- Some of the more commonly used are data types are:
 - `CHAR`, `VARCHAR` for character data - Strings.
 - `TINYINT`, `INT`, `FLOAT`, `DOUBLE` - Numeric Data
 - `DECIMAL` - Financial/Monetary numerical values
 - `DATE`, `TIME`, `DATETIME` -Date and Time values
 - `ENUM` - a list of allowed values

The CREATE Statement

- There are many uses of the **CREATE** statement.
 - ▶ However, for our purposes, the most common use is to construct tables.
 - ◆ For example, a simple table can be created using a **CREATE** statement as shown in the following script.
 - ◆ The **DESCRIBE** command will also be included to show information about the created table.

partner_create.sql

```
1 CREATE TABLE Partner(
2     company VARCHAR(20),
3     sales INT,
4     inception DATE
5 );
6
7 DESCRIBE Partner;
```

- The above script, along with the others from this chapter can be found in the *ddlscripts* folder within the labfiles.

```
mysql> SOURCE partner_create.sql
```

```
Query OK, 0 rows affected (0.01 sec)
```

Field	Type	Null	Key	Default	Extra
company	varchar(20)	YES		NULL	
sales	int(11)	YES		NULL	
inception	date	YES		NULL	

```
3 rows in set (0.00 sec)
```

- Attempting to run the above script twice will result in the following error.

ERROR 1050 (42S01): Table 'Partner' already exists

The following page has a rewrite of the above script that relies on the **DROP** statement to delete the table prior to trying to create it.

The DROP Statement

- The following script deletes the table before trying to create it.

partner_drop.sql

```
1 DROP TABLE Partner;  
2  
3 CREATE TABLE Partner(  
4     company VARCHAR(20),  
5     sales INT,  
6     inception DATE  
7 );  
8  
9 DESCRIBE Partner;
```

- The problem with the above approach is that if the table does not exist yet, the **DROP** statement will generate its own error.

ERROR 1051 (42S02): Unknown table 'student.Partner'

- Where *student* is the name of the database.
 - The easiest way to avoid the above error is to only drop the table if the table exists as shown in the example below.

partner_drop_if.sql

```
1 DROP TABLE IF EXISTS Partner;  
2  
3 CREATE TABLE Partner(  
4     company VARCHAR(20),  
5     sales INT,  
6     inception DATE  
7 );  
8  
9 DESCRIBE Partner;
```

The ENUM Data Type

- An **ENUM** is a string that has a value chosen from a list of permitted values that are enumerated explicitly in the column specification at table creation time.

tshirts_enum.sql

```
1 DROP TABLE IF EXISTS Tshirt;
2
3 CREATE TABLE Tshirt(
4     logo VARCHAR(60),
5     size ENUM('small', 'medium', 'large', 'extra-large')
6 );
7
8
9 INSERT INTO Tshirt values('Baltimore Ravens', 'extra-large');
10 INSERT INTO Tshirt values('Washington Redskins', 'big');
11
12 SELECT * FROM Tshirt;
```

- The example above relies on **INSERT** and **SELECT** statements that will be discussed in more detail in the next chapter.
 - They are being used here to help understand the **ENUM** data type.
 - ◆ The results of running the script are shown below.

```
mysql> SOURCE tshirts_enum.sql;
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 1 row affected (0.00 sec)
ERROR 1265 (01000): Data truncated for column 'size' at row 1
+-----+-----+
| logo          | size          |
+-----+-----+
| Baltimore Ravens | extra-large |
+-----+-----+
1 row in set (0.00 sec)
```

- Notice it is an error to try to insert a value that is not in the list of permitted values.

ERROR 1265 (01000): Data truncated for column 'size' at row 1

The ALTER Statement

- The structure of an existing table can be modified using an **ALTER** statement.
 - It can be used to do any of the following.
 - ◆ Remove existing columns,
 - ◆ Add columns, or
 - ◆ Modify the data type of an existing column or columns.
- The example below adds a single additional column to the *Tshirt* table, followed by multiple columns to the table.
 - It also introduces the use of comments within an SQL script.
 - ◆ Comments are used on lines 1, 5, and 10 in the script below.
 - ◆ More information about MySQL comments can be found at the following URL:

<https://dev.mysql.com/doc/refman/5.7/en/comments.html>

tshirts_alter.sql

```
1 /* Add Single Column to Table */
2 ALTER TABLE Tshirt
3     ADD (color VARCHAR(16));
4
5 # Add Multiple columns to the table
6 ALTER TABLE Tshirt
7     ADD (sleeves ENUM('Long', 'Short'),
8         pocket CHAR(1));
9
10 -- Describe the table to see the new columns
11 DESCRIBE Tshirt;
```

- When columns are added to a table, you would probably follow it up with an **INSERT** or **UPDATE** statement to add or modify data.
 - Both statements will be discussed in the next chapter.

- The following example demonstrates how to delete an existing column from a table.

tshirts_delete.sql

```
1 DESCRIBE Tshirt;
2
3 ALTER TABLE Tshirt DROP COLUMN color;
4
5 DESCRIBE Tshirt;
```

```
mysql> SOURCE tshirts_delete.sql;
```

Field	Type	Null	Key	Default	Extra
logo	varchar(60)	YES		NULL	
size	enum('small','medium','large','extra-large')	YES		NULL	
color	varchar(16)	YES		NULL	
sleeves	enum('Long','Short')	YES		NULL	
pocket	char(1)	YES		NULL	

5 rows in set (0.00 sec)

Query OK, 0 rows affected (0.05 sec)

Records: 0 Duplicates: 0 Warnings: 0

Field	Type	Null	Key	Default	Extra
logo	varchar(60)	YES		NULL	
size	enum('small','medium','large','extra-large')	YES		NULL	
sleeves	enum('Long','Short')	YES		NULL	
pocket	char(1)	YES		NULL	

4 rows in set (0.00 sec)

- Trying to drop a column that does not exist will result in an error like the one shown below.

ERROR 1091 (42000): Can't DROP 'color'; check that column/key exists

- The next example demonstrates using an **ALTER** statement to change the following.
 - The first statement is used to change the data type of an existing column.
 - The second statement is used to change the name of an existing column.

tshirts_changes.sql

```

1 DESCRIBE Tshirt;
2
3 # Change the data type of a column
4 ALTER TABLE Tshirt CHANGE pocket pocket ENUM('Y' , 'N');
5
6 # Change the name of a column
7 ALTER TABLE Tshirt CHANGE sleeves sleeve ENUM('Long', 'Short');
8
9 DESCRIBE Tshirt;
```

- Running the above script displays the following.

```
mysql> SOURCE tshirts_changes.sql;
```

Field	Type	Null	Key	Default	Extra
logo	varchar(60)	YES		NULL	
size	enum('small','medium','large','extra-large')	YES		NULL	
sleeves	enum('Long','Short')	YES		NULL	
pocket	char(1)	YES		NULL	

4 rows in set (0.00 sec)

Query OK, 1 row affected (0.01 sec)

Records: 1 Duplicates: 0 Warnings: 0

Query OK, 0 rows affected (0.01 sec)

Records: 0 Duplicates: 0 Warnings: 0

Field	Type	Null	Key	Default	Extra
logo	varchar(60)	YES		NULL	
size	enum('small','medium','large','extra-large')	YES		NULL	
sleeve	enum('Long','Short')	YES		NULL	
pocket	enum('Y','N')	YES		NULL	

4 rows in set (0.00 sec)

Integrity Constraints

- A constraint is a rule or restriction that a database is not permitted to violate.
 - A constraint may apply to an individual column or to a relationship between two tables.
 - ◆ The constraints ensure that updates, deletions, and or insertions to the tables in a database does not result in data corruption or loss in data consistency.
 - ◆ The constraints are placed on a column when a table is created or altered.
 - One of two methods can be used declare the constraints.
 - ◆ In-line can be used when they are defined in the column definition.
 - ◆ Out-of-line can be used when they are defined in the table definition. Out-of-line constraints typically involve multiple columns.
- Constraints may be created without a name, in which case the database provides a name for it behind the scenes.
 - It is recommended to name each constraint with a descriptive name so that when an error occurs, it is easier to identify.
- There are several types of Integrity Constraints.
 - Domain Integrity Constraints
 - Entity Integrity Constraints
 - Referential Integrity Constraints
- While details of these constraints will be discussed next, additional information on how MySQL deals with constraints can be found at the following URL.

<https://dev.mysql.com/doc/refman/5.7/en/constraints.html>

Domain Integrity Constraints

- Domain integrity constraints restrict the range of valid values of a column.
 - ▶ These types of constraints applied to a column may consist of the following.

Data Type	Length	Size
Null Values	Uniqueness	Defaults
Range of Values		

- Entity integrity constraints deal specifically with primary keys.
 - ▶ These constraints ensure that a primary key is not null and unique.
- Referential integrity constraints apply to foreign keys.
 - ▶ If a foreign key exists in a relation then its value must either match a primary key in its related table or it must be **NULL**.
 - ▶ The following rules are maintained by the database when dealing with foreign keys.
 - ◆ A record from a primary table cannot be deleted if it matches a record in a table related to it by a foreign key.
 - ◆ A primary key cannot be changed if that record has related records via a foreign key.
 - ◆ A foreign key value must be that of an existing value of a primary key from the related table or **NULL** if allowed.

Creating a Table with Domain Constraints

- The example below demonstrates various domain constraints being placed on the columns within a table when the table is being created.

sales_create.sql

```
1 DROP TABLE IF EXISTS Sale;
2
3 CREATE TABLE Sale (
4     invoice INT NOT NULL, /* in-line constraint */
5     discount VARCHAR(10),
6     amount decimal(7, 2) NOT NULL,
7     class_id INT,
8     customer_id INT,
9     description VARCHAR(20),
10    CONSTRAINT inv_num UNIQUE(invoice) /* out-of-line constraint */
11 );
```

- The following three scripts will be used to test the constraints defined in the table above.

sales_insert_a.sql

```
1 # This one should succeed
2 INSERT INTO Sale VALUES (23456, 'Gold', 10500, 5000, 1000, 'XHTML');
```

sales_insert_b.sql

```
1 # This one should fail NULL constraint on amount
2 INSERT INTO Sale VALUES (54, 'Gold', NULL, 5000, 1000, 'XHTML');
```

sales_insert_c.sql

```
1 # This one should fail UNIQUE constraint on invoice
2 INSERT INTO Sale VALUES (23456, 'Platinum', 10250, 6000, 2000, 'Java');
```

- The output below shows the running of the *sales_insert_c.sql* script.

```
mysql> SOURCE sales_insert_c.sql;
Query OK, 1 row affected (0.00 sec)
ERROR 1062 (23000): Duplicate entry '23456' for key 'inv_num'
```

Entity Integrity Constraints

- Recall that entity integrity constraints deal specifically with primary keys, ensuring that a primary key is not null and unique.
 - While this could be achieved using the **NOT NULL** and **UNIQUE** constraints, it can be simplified with the **PRIMARY KEY** constraint.
- If the primary key of a table consists of a single field, then the constraint can be defined as an in-line constraint during its column definition.
 - The in-line syntax would be as follows.

```
CREATE TABLE _table_name_ (_column_name_ _data_type_ PRIMARY KEY);
```

- If the primary key consists of a combination of fields, then the out-of-line constraint must be used.
 - The out-of-line syntax would be as follows.

```
CREATE TABLE table_name (  
    column_definitions...,  
    CONSTRAINT constraint_name  
    PRIMARY KEY (column_name1, column_name2, ...);
```

- The example on the following page redesigns the **Sale** table so that the invoice is declared using a **PRIMARY KEY** constraint.
 - The script creates the table in two different ways.
 - ◆ The first **CREATE** statement uses the in-line syntax for the primary key.
 - ◆ The second **CREATE** statement uses the out-of-line syntax.

pk_sales_create.sql

```

1 DROP TABLE IF EXISTS Sale;
2
3 /* In-line PRIMARY KEY example */
4 CREATE TABLE Sale (
5     invoice INT PRIMARY KEY,
6     discount VARCHAR(10),
7     amount decimal(7, 2) NOT NULL,
8     class_id INT,
9     customer_id INT,
10    description VARCHAR(20)
11 );
12
13
14 DROP TABLE IF EXISTS Sale;
15
16 /* Out-of-line PRIMARY KEY example */
17 CREATE TABLE Sale (
18     invoice INT,
19     discount VARCHAR(10),
20     amount decimal(7, 2) NOT NULL,
21     class_id INT,
22     customer_id INT,
23     description VARCHAR(20),
24     CONSTRAINT pk_invoice PRIMARY KEY(invoice)
25 );

```

- The output below shows the running of the same three attempts to insert into the **Sale** table, this time based on the above table definition.

```

mysql> SOURCE pk_sales_create.sql
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
mysql> SOURCE sales_insert_a.sql;
Query OK, 1 row affected (0.00 sec)
mysql> SOURCE sales_insert_b.sql;
ERROR 1048 (23000): Column 'amount' cannot be null
mysql> SOURCE sales_insert_c.sql;
Query OK, 1 row affected (0.01 sec)
ERROR 1062 (23000): Duplicate entry '767' for key 'PRIMARY'

```


Referential Integrity Constraints

- As stated earlier, referential entity integrity constraints deal specifically foreign keys.
 - A valid foreign key value is required to reference an existing primary key in another specified table or be set to **NULL**.
 - MySQL does not support any form of in-line syntax for defining a foreign key.

```
CREATE TABLE table_name (column_name data_type PRIMARY KEY);
```

- The out-of-line syntax is supported and would be as follows.

```
CREATE TABLE table_name(
    column_name data_type,
    CONSTRAINT constraint_name
        FOREIGN KEY (column_name1)
            REFERENCES other_table_name(other_column_name);
```

- The example below demonstrates the creation of a table with a foreign key that references a primary key in another table using the above syntax.

foreign_key_example.sql

```
1 DROP TABLE IF EXISTS Person;
2 DROP TABLE IF EXISTS Tshirt;
3
4 CREATE TABLE Person (
5     id INT PRIMARY KEY,
6     name CHAR(60) NOT NULL
7 );
8
9 CREATE TABLE Tshirt(
10    person_id INT,
11    logo VARCHAR(60),
12    size ENUM('small', 'medium', 'large', 'extra-large'),
13    FOREIGN KEY(person_id)
14        REFERENCES Person(id)
15 );
```

- The script on the following page attempts to execute several statements to insert data into the above two tables to show how MySQL maintains referential integrity.

foreign_key_testing.sql

```

1 INSERT INTO Person VALUES (123, 'Beverly Richards');
2 INSERT INTO Tshirt VALUES (123, 'I Can Relate', 'medium');
3 INSERT INTO Tshirt VALUES (123, 'Little Bobby Tables', 'medium');
4
5 # This one will fail - No id of 789 exists as a primary key in Person
6 INSERT INTO Tshirt VALUES (789, 'Join us', 'medium')

```

```

mysql> SOURCE foreign_key_example.sql
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
mysql> SOURCE foreign_key_testing.sql
Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected (0.01 sec)
Query OK, 1 row affected (0.00 sec)
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails
(student.Tshirt, CONSTRAINT Tshirt_ibfk_1 FOREIGN KEY (person_id) REFERENCES Person (id))

```

- The last **INSERT** statement fails due to a foreign key constraint.
 - When dealing with primary keys and foreign keys, the order in which tables are dropped becomes important in that the one with the foreign keys would have to be dropped first.
- This is yet another way in which referential integrity is maintained.

```

mysql> DROP TABLE Person;
ERROR 1217 (23000): Cannot delete or update a parent row: a foreign key constraint fails
mysql> DROP TABLE Tshirt;
Query OK, 0 rows affected (0.01 sec)
mysql> DROP TABLE Person;
Query OK, 0 rows affected (0.01 sec)

```

Altering a Table's Constraints

- Several restrictions exist when altering table constraints.
 - A field with a **NOT NULL** constraint can only be added no rows exist.
 - An existing field may only be changed to **NOT NULL** if every row has a value that is not **NULL**.
- The general syntax for altering a constraint is as follows.

```
ALTER TABLE table_name
  ADD(CONSTRAINT constraint_name constraint_type
      constraint_parameters);
```

- The **NOT NULL** constraint is added as follows.

```
ALTER TABLE table_name
  CHANGE col_name col_name data_type NOT NULL;
```

- A constraint can be dropped by using the following syntax.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

- The following script will re-create the *Sale* and *Customer* tables before altering various constraints within the tables.

customers_and_sales.sql

```
1 DROP TABLE IF EXISTS Sale;
2 DROP TABLE IF EXISTS Customer;
3
4 CREATE TABLE Sale (
5     invoice INT NOT NULL, /* in-line constraint */
6     discount VARCHAR(10),
7     amount decimal(7, 2) NOT NULL,
8     class_id INT,
9     customer_id INT,
10    description VARCHAR(20),
11    CONSTRAINT inv_num UNIQUE(invoice) /* out-of-line constraint */
12 );
13
14 CREATE TABLE Customer (
15     customer_id INT,
16     name        VARCHAR(30),
17     contact     VARCHAR(20),
18     phone       CHAR(10),
19     type        VARCHAR(15),
20     sales_person INT,
21     comments    VARCHAR(15)
22 );
```

- The script below is used to describe the two tables created above.

describing.sql

```
1 DESCRIBE Sale;
2 DESCRIBE Customer;
```

- The output below describes both the *Sale* and *Customer* tables before altering the constraints in various ways.

```
mysql> SOURCE describing.sql
```

Field	Type	Null	Key	Default	Extra
invoice	int(11)	NO	PRI	NULL	
discount	varchar(10)	YES		NULL	
amount	decimal(7,2)	NO		NULL	
class_id	int(11)	YES		NULL	
customer_id	int(11)	YES		NULL	
description	varchar(20)	YES		NULL	

```
6 rows in set (0.00 sec)
```

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	YES		NULL	
name	varchar(30)	YES		NULL	
contact	varchar(20)	YES		NULL	
phone	char(10)	YES		NULL	
type	varchar(15)	YES		NULL	
sales_person	int(11)	YES		NULL	
comments	varchar(15)	YES		NULL	

```
7 rows in set (0.00 sec)
```

The following script will add a **NOT NULL** constraint to the description field of the *Sale* table.

modifying_constraints_01.sql

```
1 ALTER TABLE Sale
2   CHANGE description description varchar(20) NOT NULL;
```

The script below shows how to handle dropping a **UNIQUE** constraint properly in MySQL.

modifying_constraints_02.sql

```
1 # MySQL does not allow dropping of a UNIQUE constraint as shown below
2 ALTER TABLE Sale DROP CONSTRAINT inv_num;
3
4 # It must be dropped by index instead
5 ALTER TABLE Sale DROP INDEX inv_num;
```

With the **UNIQUE** constraint dropped on the *Sale* table, the next example will set both **PRIMARY KEY** and **FOREIGN KEY** constraints on the *Sale* and *Customer* tables.

modifying_constraints_03.sql

```

1 ALTER TABLE Sale
2   ADD (CONSTRAINT pk_invoice
3       PRIMARY KEY(invoice, customer_id, class_id));
4
5 ALTER TABLE Customer
6   ADD (CONSTRAINT pk_cust_id PRIMARY KEY(customer_id));
7
8 ALTER TABLE Sale
9   ADD (CONSTRAINT fk_cust_id FOREIGN KEY(customer_id)
10      REFERENCES Customer(customer_id));

```

- Sourcing the tables once again will show the results of the changes to the constraints on the fields within the tables.

mysql> SOURCE describing.sql;

Field	Type	Null	Key	Default	Extra
invoice	int(11)	NO	PRI	NULL	
discount	varchar(10)	YES		NULL	
amount	decimal(7,2)	NO		NULL	
class_id	int(11)	NO	PRI	NULL	
customer_id	int(11)	NO	PRI	NULL	
description	varchar(20)	NO		NULL	

6 rows in set (0.00 sec)

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	NO	PRI	NULL	
name	varchar(30)	YES		NULL	
contact	varchar(20)	YES		NULL	
phone	char(10)	YES		NULL	
type	varchar(15)	YES		NULL	
sales_person	int(11)	YES		NULL	
comments	varchar(15)	YES		NULL	

7 rows in set (0.00 sec)

Exercises

Exercise 1

Write a script that creates a table named *Worker* that has the following fields.

- *fname* of type **VARCHAR**
 - A maximum of 10 characters allowed.
 - No nulls allowed.
- *lname* of type **VARCHAR**
 - A maximum of 10 characters allowed.
 - No nulls allowed.
- *id* of type **INT**
 - This field should be designated as the primary key.
- *hiredate* of type **DATE**.
 - No nulls allowed.
- *gender* of type **ENUM**
 - Containing only the values 'F', or 'M'.
- *position* of type **VARCHAR**
 - A maximum of 10 characters allowed.

Exercise 2

Display the fields and the data types of the *Worker* table.

Exercise 3

Change the name of the *position* field to *title*.

- Make the field not allow nulls.

Exercise 4

Add a foreign key constraint to the *class_id* field in the *Sale* table you created in this chapter.

- If you have not created it, please do so.

- If you receive an error, correct the error and try again.

Chapter 5. Data Manipulation Language (DML)

Objectives

- DML Statements
- The SELECT Statement
- The INSERT Statement
- The DELETE Statement
- The UPDATE Statement

DML Statements

The Data Manipulation Language (DML) does exactly what it states in that it manipulates data.

- In this chapter, we will focus on how it manipulates the data in the tables of a database.

The statements we will focus on in this chapter are the following.

- **SELECT**
 - A **SELECT** statement, also called a query, is typically used to retrieve data from a database table or view.
- **INSERT**
 - An **INSERT** statement is used to insert a row or rows into a table.
- **DELETE**
 - A **DELETE** statement is used to delete a row or rows from a table or view.
- **UPDATE**
 - An Update statement is used to update existing data within a table

The most powerful of these statements is probably the **SELECT** statement.

- Its full capabilities will be examined in more detail in subsequent chapters.

The **SELECT** Statement

As mentioned, the **SELECT** statement is used to query the database.

- It may query the database about one or more tables and return a collection of rows, known as the result set, which satisfies the query.

We have already used the **SELECT** statement, in its simplest form, to examine the contents of the tables from previous chapters.

```
mysql> SELECT * FROM Team;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

- The above statement contains the key words, **SELECT** and **FROM**.
 - ▶ The asterisk (*) is a wild card symbol, which matches all columns in the *Team* table.
- Since no additional restrictions have been placed on the query, the query will match every row within the table.

A **WHERE** clause can be used to place restrictions on a query.

- The **WHERE** clause will limit the returned result set to a specific set of rows which match a given restriction.

select_where.sql

```
1 SELECT * FROM Course WHERE num_days = 5;
```

```
mysql> SOURCE select_where.sql;
```

course_id	name	unit_price	num_days	courseware_id
100	Java Programming	500.00	5	TE100
220	Perl Programming	400.00	5	TE220

2 rows in set (0.00 sec)

The next **SELECT** queries limit the result set to a set of columns by explicitly specifying a comma separated list of desired fields instead of a wild card.

select_columns.sql

```
1 SELECT name, courseware_id, unit_price FROM Course WHERE num_days = 5;
2
3 SELECT name, team_id FROM Team WHERE team_id >= 402;
```

```
mysql> SOURCE select_columns.sql;
```

name	courseware_id	unit_price
Java Programming	TE100	500.00
Perl Programming	TE220	400.00

2 rows in set (0.00 sec)

name	team_id
Systems	402
Sales	403

2 rows in set (0.00 sec)

The ORDER BY Clause

The **ORDER BY** clause can be used in a **SELECT** query to specify the order in which the rows of a result set returned.

- It provides the ability to specify what field(s) the rows should be by.
 - By default, they are placed in ascending order.
 - The keywords **ASC** and **DESC** can be used to specify ascending and descending order, respectively.
- This clause is placed at the end of a **SELECT** statement.

order_by_single.sql

```
1 SELECT last_name, first_name, phone, gender, hire_date FROM Employee;  
2  
3 SELECT last_name, first_name, phone, gender, hire_date FROM Employee  
4     ORDER BY last_name;
```

```
mysql> SOURCE order_by_single.sql;
```

last_name	first_name	phone	gender	hire_date	salary
Wilson	Morgan	2015556298	M	1990-06-10	50000
Wilson	Emily	2015556298	F	1995-01-01	50000
Perkins	Kim	4105557985	F	1997-05-21	40000
Hardin	Alex	4105557648	M	1998-04-09	35000
Johnson	Tina	4105553274	F	2008-11-23	35000
Fitzgerald	Rachel	3015559466	F	2005-03-01	35000
Higgins	Maria	4105552105	F	2001-03-21	35000
Perkins	Brian	2145554425	M	2001-03-21	35000
McKinney	Connie	4105552544	F	2001-03-21	35000
Train	Travis	4105553512	M	2004-05-31	25000

```
10 rows in set (0.00 sec)
```

last_name	first_name	phone	gender	hire_date	salary
Fitzgerald	Rachel	3015559466	F	2005-03-01	35000
Hardin	Alex	4105557648	M	1998-04-09	35000
Higgins	Maria	4105552105	F	2001-03-21	35000
Johnson	Tina	4105553274	F	2008-11-23	35000
McKinney	Connie	4105552544	F	2001-03-21	35000
Perkins	Kim	4105557985	F	1997-05-21	40000
Perkins	Brian	2145554425	M	2001-03-21	35000
Train	Travis	4105553512	M	2004-05-31	25000
Wilson	Morgan	2015556298	M	1990-06-10	50000
Wilson	Emily	2015556298	F	1995-01-01	50000

```
10 rows in set (0.00 sec)
```


A query may also specify multiple columns to sort on.

- The first column listed would be considered the primary sort field.
- The second the secondary sort, and so on.

order_by_multiple.sql

```
1 SELECT last_name, first_name, phone, gender, hire_date FROM Employee;
2
3 SELECT last_name, first_name, phone, gender, hire_date FROM Employee
4     ORDER BY last_name DESC, first_name;
```

```
mysql> SOURCE order_by_multiple.sql;
```

last_name	first_name	phone	gender	hire_date
Wilson	Morgan	2015556298	M	1990-06-10
Wilson	Emily	2015556298	F	1995-01-01
Perkins	Kim	4105557985	F	1997-05-21
Hardin	Alex	4105557648	M	1998-04-09
Johnson	Tina	4105553274	F	2008-11-23
Fitzgerald	Rachel	3015559466	F	2005-03-01
Higgins	Maria	4105552105	F	2001-03-21
Perkins	Brian	2145554425	M	2001-03-21
McKinney	Connie	4105552544	F	2001-03-21
Train	Travis	4105553512	M	2004-05-31

10 rows in set (0.00 sec)

last_name	first_name	phone	gender	hire_date
Wilson	Emily	2015556298	F	1995-01-01
Wilson	Morgan	2015556298	M	1990-06-10
Train	Travis	4105553512	M	2004-05-31
Perkins	Brian	2145554425	M	2001-03-21
Perkins	Kim	4105557985	F	1997-05-21
McKinney	Connie	4105552544	F	2001-03-21
Johnson	Tina	4105553274	F	2008-11-23
Higgins	Maria	4105552105	F	2001-03-21
Hardin	Alex	4105557648	M	1998-04-09
Fitzgerald	Rachel	3015559466	F	2005-03-01

10 rows in set (0.00 sec)

The default way in which **NULL** values are included in sorts is RDBMS specific.

- MySQL will sort them first in ascending.
- **NULL** values may be omitted by adding a **WHERE column_name IS NOT NULL** clause.

order_with_nulls.sql

```
1 SELECT last_name, first_name, employee_id, mentor_id FROM Employee
2   ORDER BY mentor_id;
3
4 SELECT last_name, first_name, employee_id, mentor_id FROM Employee
5   WHERE mentor_id IS NOT NULL
6   ORDER BY mentor_id;
```

```
mysql> SOURCE order_with_nulls.sql;
```

last_name	first_name	employee_id	mentor_id
Wilson	Morgan	500	NULL
Wilson	Emily	501	NULL
Perkins	Kim	502	NULL
Hardin	Alex	503	NULL
Johnson	Tina	508	NULL
Fitzgerald	Rachel	504	500
McKinney	Connie	507	501
Train	Travis	509	502
Perkins	Brian	506	503
Higgins	Maria	505	508

10 rows in set (0.00 sec)

last_name	first_name	employee_id	mentor_id
Fitzgerald	Rachel	504	500
McKinney	Connie	507	501
Train	Travis	509	502
Perkins	Brian	506	503
Higgins	Maria	505	508

5 rows in set (0.00 sec)

The INSERT Statement

The **INSERT** statement will insert a new row(s) into a table. The first form of this statement uses explicit column names to assign values to the row.

explicit_inserts.sql

```
1 INSERT INTO Team (name, team_id) VALUES ('Production', 404);
```

- The order in which the values are listed does not matter, but they must correspond to the order in which the column names are specified.
 - ▶ In the example above, since only two values were supplied, the third column is filled with the **NULL** value.
 - ▶ If a column has a **NOT NULL** constraint and the **INSERT** statement does not enter a value into that column, the statement would fail with an error message.

An **INSERT** statement may also use implicit column names if all values for the columns are provided in the order the columns were created for the table.

implicit_inserts.sql

```
1 # This one does not provide enough values and will error out
2 INSERT INTO Team VALUES (405, 'Recreation');
3
4 # This one will be ok in the number and order of values matches table
5 INSERT INTO Team VALUES (405, 'Recreation', 500);
```

```
mysql> SOURCE explicit_inserts.sql;
Query OK, 1 row affected (0.00 sec)
mysql> SOURCE implicit_inserts.sql;
ERROR 1136 (21S01): Column count doesn't match value count at row 1
Query OK, 1 row affected (0.00 sec)
```

The results of running the previous two scripts is shown in the output of the **SELECT** statement shown below.

```
mysql> SELECT * FROM Team;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 404     | Production | NULL      |
| 405     | Recreation | 500       |
+-----+-----+-----+
6 rows in set (0.00 sec)
```

When more than a single row of data needs to be inserted into a table, one statement for each insert would create a new connection for each insert.

- A more efficient way is to supply all the rows can be supplied as a comma separated list.
 - This results in a single connection to insert all the rows.
 - This technique is the one that was used to populate all the sample tables.

The code below is a snippet of the *filltables.sql* script in the *sqlscripts* directory used earlier to populate the sample tables in the database

filltables.sql ~ *snippet*

```
1. ...
2. INSERT INTO Team
3. VALUES (400, 'Education', 500),
4. (401, 'Management', 501),
5. (402, 'Systems', 502),
6. (403, 'Sales', 508);
7. ...
```

There may be instances where data from one table needs to be inserted into another table.

- A **SELECT** clause can be used within an **INSERT** statement to include the information from one table and insert it into another.
 - If the **SELECT** statement is querying the first table for information in the order that the fields are defined in the second table, implicit column names may be used.

insert_with_select.sql

```
1 INSERT INTO Team
2     SELECT 406, 'Entertainment', employee_id FROM Employee
3     WHERE last_name = 'Johnson';
4
5 SELECT * FROM Team;
```

```
mysql> SOURCE insert_with_select.sql;
```

```
Query OK, 1 row affected (0.00 sec)
```

```
Records: 1 Duplicates: 0 Warnings: 0
```

team_id	name	leader_id
400	Education	500
401	Management	501
402	Systems	502
403	Sales	508
404	Production	NULL
405	Recreation	500
406	Entertainment	508

```
7 rows in set (0.00 sec)
```

Notice that *406* and *Entertainment* are constants that are selected and while the *employee_id* associated with *last_name* of *Johnson* from the *Employee* table is selected for the *leader_id* field.

The DELETE Statement

DELETE statements are used to remove one or more rows from a table.

- Once again, the code below is a snippet of the *filltables.sql* script in the *sqlscripts* directory used earlier to populate the sample tables in the database.
 - The use of the DELETE statement immediately precedes the INSERT statement.

filltables.sql ~ *snippet*

```

1. ...
2. DELETE FROM Team;
3. INSERT INTO Team
4. VALUES (400, 'Education', 500),
5. (401, 'Management', 501),
6. (402, 'Systems', 502),
7. (403, 'Sales', 508);
8. ...

```

A WHERE clause can be used to constrain the row or rows to be deleted as shown in the following example.

delete_where.sql

```

1 DELETE FROM Team WHERE team_id > 404;
2
3 SELECT * FROM Team;

```

```

mysql> SOURCE delete_where.sql;
Query OK, 2 rows affected (0.00 sec)
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 404     | Production | NULL      |
+-----+-----+-----+
5 rows in set (0.00 sec)

```

The UPDATE Statement

The **UPDATE** statement is used to modify the contents of one or more rows in a table.

- Once again, a **WHERE** clause is used to target the rows which are to be affected.
- A **SET** clause is used to modify a column or columns within the targeted row or rows.

updating.sql

```
1 SELECT * FROM Team;
2
3 UPDATE Team SET team_id = 405 WHERE team_id = 404;
4
5 SELECT * FROM Team;
```

```
mysql> SOURCE updating.sql;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 404     | Production | NULL      |
+-----+-----+-----+
5 rows in set (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 405     | Production | NULL      |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

Arithmetic operators may also be used in numeric fields to update the content.

- The following **UPDATE** statement increases every employee's current salary by 5%.

calculated_update.sql

```
1 SELECT first_name, hire_date, fire_date, salary FROM Employee;  
2  
3 UPDATE Employee SET salary = salary * 1.05 WHERE fire_date IS NULL;  
4  
5 SELECT first_name, hire_date, fire_date, salary FROM Employee;
```



```
mysql> SOURCE calculated_update.sql;
```

```
+-----+-----+-----+-----+
| first_name | hire_date | fire_date | salary |
+-----+-----+-----+-----+
| Morgan     | 1990-06-10 | NULL      | 55125  |
| Emily      | 1995-01-01 | NULL      | 55125  |
| Kim        | 1997-05-21 | NULL      | 44100  |
| Alex       | 1998-04-09 | NULL      | 38588  |
| Tina       | 2008-11-23 | NULL      | 38588  |
| Rachel     | 2005-03-01 | NULL      | 38588  |
| Maria      | 2001-03-21 | NULL      | 38588  |
| Brian      | 2001-03-21 | NULL      | 38588  |
| Connie     | 2001-03-21 | NULL      | 38588  |
| Travis     | 2004-05-31 | 2004-09-30 | 25000  |
+-----+-----+-----+-----+
```

```
10 rows in set (0.00 sec)
```

```
Query OK, 9 rows affected (0.00 sec)
```

```
Rows matched: 9 Changed: 9 Warnings: 0
```

```
+-----+-----+-----+-----+
| first_name | hire_date | fire_date | salary |
+-----+-----+-----+-----+
| Morgan     | 1990-06-10 | NULL      | 57881  |
| Emily      | 1995-01-01 | NULL      | 57881  |
| Kim        | 1997-05-21 | NULL      | 46305  |
| Alex       | 1998-04-09 | NULL      | 40517  |
| Tina       | 2008-11-23 | NULL      | 40517  |
| Rachel     | 2005-03-01 | NULL      | 40517  |
| Maria      | 2001-03-21 | NULL      | 40517  |
| Brian      | 2001-03-21 | NULL      | 40517  |
| Connie     | 2001-03-21 | NULL      | 40517  |
| Travis     | 2004-05-31 | 2004-09-30 | 25000  |
+-----+-----+-----+-----+
```

```
10 rows in set (0.00 sec)
```

Exercises

Exercise 1

Create a script that will have statements to list the following.

- The employee ID, first name, and last name of all employees.
- The course ID and name of all courses.
- The team ID and name of all teams.

Exercise 2

Modify a copy of the above script so that the employees will be listed in alphabetical order by last name, and then execute the script.

Exercise 3

Building upon the *Worker* table created as an exercise in the previous chapter.

- Create a script that inserts several workers into the *Worker* table.
- Use the **DESCRIBE** command if you need to see the fields and their data types.

Exercise 4

Write a script to change the last name of one of the workers that you just inserted into the *Worker* table.

- Display the contents of the *Worker* table.

Exercise 5

Make the following updates to the employee in the *Employee* table with the last name Fitzgerald.

- Increase the salary by 10% of the current salary.
- Modify the bonus to be 20% of the current salary.
- Modify the hire date to May 5, 2005.

Chapter 6. Transaction Control

Objectives

- Transactions
- Command Classification

Transactions

A transaction is a collection of SQL statements that are executed as one unit.

- The main purpose of a transaction is to ensure that the database remains in a consistent state.

An RDBMS will perform the following when a DML statement that changes a table is executed.

- The statement is saved in memory as being part of the current transaction.
 - ▶ The RDBMS records the state of the row or rows being changed in a rollback (or undo) segment.
 - ▶ The table is locked to other users until the transaction is completed, in order to maintain consistency.

Transaction control is available as a subset of DML statements

- The **COMMIT** statement makes any **INSERT**, **UPDATE**, and/or **DELETE** statements permanent.
- The **ROLLBACK** statement undoes all modifications from the last **COMMIT**.
- The **SAVEPOINT** statement creates a check point within a transaction.

By default, MySQL performs a **COMMIT** after each SQL statement.

- To disable auto commit, use the following command.

```
SET AUTOCOMMIT = 0;
```

- To enable auto commit, use the following command.

```
SET AUTOCOMMIT = 1;
```

The ROLLBACK Statement

The following statements from the interactive MySQL prompt will demonstrate the **ROLLBACK** statement.

```
mysql> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM Team;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 405     | Production | NULL      |
+-----+-----+-----+
5 rows in set (0.00 sec)
mysql> DELETE FROM Team;
Query OK, 5 rows affected (0.00 sec)
mysql> SELECT * FROM Team;
Empty set (0.00 sec)
mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)
mysql> SELECT * FROM Team;
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
| 405     | Production | NULL      |
+-----+-----+-----+
5 rows in set (0.00 sec)
mysql> exit
Bye
```

Exiting out of the MySQL prompt will result in the **AUTOCOMMIT** feature being defaulted back to a value of **1** when the MySQL prompt is started up again.

- The very first statement above that sets it equal to **0** only sets it for the current interactive MySQL session.

The COMMIT Statement

The following statements from the interactive MySQL prompt will demonstrate the COMMIT statement.

```
mysql> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)
mysql> DELETE FROM Team;
Query OK, 5 rows affected (0.00 sec)
mysql> SELECT * FROM Team;
Empty set (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
mysql> SELECT * FROM Team;
Empty set (0.00 sec)
mysql> exit
Bye
```

The SAVEPOINT Statement

A save point enables a transaction to be rolled back to a certain point within the transaction instead of rolling back to the beginning of transaction.

- This can help if several related changes are going to be made to the database and the user wants to make the changes in phases.

For example, a new employee, Janice, joins a company and replaces another employee, Morgan, because he is about to retire.

- Janice is not only entered in the *Employee* table but also in the *Team* table (to replace Morgan as leader of his department). Janice must also replace Morgan as a mentor for several employees in the *Employee* table.
 - ▶ Phase 1 would be to enter Janice into the *Employee* table.
 - ▶ Phase 2 would be to replace Morgan with Janice as the leader of the education department in the *Team* table.
 - ▶ Phase 3 would be to make Janice the mentor of all employees who have Morgan as a mentor in the *Employee* table.
- Save points would be used to save the changes at each phase.
 - ▶ That way if an error occurred in Phase 3, instead of rolling back the entire transaction, the rollback could be to Phase 2 and then a redo could be done to Phase 3.

The example below begins the process of adding a new employee as described on the previous page.

- It begins by running the *filltables.sql* script to reset all the data in the sample tables.

savepoints.sql

```
1 SOURCE ~/sqllabs/examples/sqlscripts/filltables.sql;
2 SET AUTOCOMMIT = 0;
3
4 INSERT INTO Employee(employee_id, first_name, last_name, salary)
5     VALUES(510, 'Janice', 'Johnson', 52000);
6
7 SAVEPOINT phase1;
8
9
10 UPDATE Team SET leader_id = 510;
11
12 SELECT * FROM Team;
13
14 /* This "Accidentally" updated more rows than was desired
15     but the INSERT statement was no a problem so it only needs
16     to be rolled back to the phase1 save point */
17 ROLLBACK TO phase1;
18
19 UPDATE Team SET leader_id = 510 where leader_id = 500;
20
21 COMMIT;
22
23 SELECT * FROM Team;
```

The *Team* table is shown below, before and after the rollback.


```
mysql> **SOURCE savepoints.sql;**
Query OK, 4 rows affected (0.00 sec)
Rows matched: 4 Changed: 4 Warnings: 0
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 510       |
| 401     | Management | 510       |
| 402     | Systems   | 510       |
| 403     | Sales     | 510       |
+-----+-----+-----+
4 rows in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
Query OK, 0 rows affected (0.00 sec)
+-----+-----+-----+
| team_id | name      | leader_id |
+-----+-----+-----+
| 400     | Education | 500       |
| 401     | Management | 501       |
| 402     | Systems   | 502       |
| 403     | Sales     | 508       |
+-----+-----+-----+
4 rows in set (0.00 sec)
mysql> **exit;**
Bye
```

Exercises

Exercise 1

Complete the `savepoints.sql` example in this chapter where Janice is replacing Morgan in all her capacities because she is retiring.

- Update Janice in the *Employee* table with the following.
 - A hire date of July 15, 2015
 - A phone number of your liking.
 - She will belong to the Education team.
 - She is a married female.
 - Create the first savepoint.
- Replace Morgan with Janice as the leader of the Education department in the *Team* table.
 - Create a second savepoint.
- Make Janice the mentor of all employees who have Morgan as a mentor in the *Employee* table.
 - Create a third savepoint.
- Rollback changes to the first savepoint, so that Janice's information has been updated but has not taken over Morgan's responsibilities.
- Commit the changes to the database.

Exercise 2

With **AUTO COMMIT** turned off:

- Remove the record for the only employee in the *Employee* table with a fire date.
- Display the contents of the *Employee* table.
- Rollback so that the employee still exists in the table.

Chapter 7. SQL Operators

Objectives

- Comparison Operators
- IN and NOT IN Operators
- BETWEEN Operator
- The LIKE Operator
- Logical Operators
- IS NULL and IS NOT NULL
- Case statements

Comparison Operators

- Earlier chapters demonstrated many examples of simple **SELECT** statements.
 - Several demonstrated being more specific about records using a **WHERE** clause.
 - ◆ The **WHERE** clause used the equality operator (=) to specify the conditions to for the **SELECT** query to match.
 - This can be seen in the *select_where.sql* script from the previous chapter.
 - ◆ The file from that chapter shown again here.

select_where.sql

```
1 SELECT * FROM Course WHERE num_days = 5;
```

```
mysql> SOURCE ~/sqllabs/examples/dmlscripts/select_where.sql;
```

course_id	name	unit_price	num_days	courseware_id
100	Java Programming	500.00	5	TE100
220	Perl Programming	400.00	5	TE220

2 rows in set (0.00 sec)

- The equality operator is a comparison operator.
 - It tests to see if the values on each side of the operator are equal.
- There is also an inequality operator to test for values that are not equal.
 - MySQL supports the following two variations of the inequality operator.
 - ◆ **!=**
 - ◆ **<>**

- The table below lists some of the common SQL comparison operators.
 - A complete list of comparison operators and functions supported by MySQL can be found at the following URL.

<https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html>

Table 14. Comparison Operators

Operator	Description
=	Equal
!=, <>	Not equal
>	Greater than
>=	Greater Than or equal
<	Less than
<=	Less than or equal

- These operators work for numbers, strings and dates.
- Strings are automatically converted to numbers and numbers to strings as necessary.
- By default, string comparisons are not case-sensitive.
 - You can use these operators for numeric, string, and date data types.
- The example on the following page demonstrates the use of comparison operators on numbers, strings, and dates.

strings_dates_numbers.sql

```

1 SELECT * FROM Course WHERE courseware_id > 'TE120';
2
3 SELECT first_name, last_name, hire_date
4     FROM Employee WHERE hire_date >= '2000-01-01';
5
6 SELECT * FROM Course WHERE unit_price > 250.00 AND unit_price < 500.00;

```

```
mysql> SOURCE strings_dates_numbers.sql;
```

course_id	name	unit_price	num_days	courseware_id
200	XHTML	250.00	2	TE200
210	JavaScript	300.00	3	TE210
220	Perl Programming	400.00	5	TE220
300	Oracle SQL	500.00	3	TE300

4 rows in set (0.00 sec)

first_name	last_name	hire_date
Tina	Johnson	2008-11-23
Rachel	Fitzgerald	2005-03-01
Maria	Higgins	2001-03-21
Brian	Perkins	2001-03-21
Connie	McKinney	2001-03-21
Travis	Train	2004-05-31

6 rows in set (0.00 sec)

course_id	name	unit_price	num_days	courseware_id
110	JSPs	400.00	2	TE110
120	Servlets	400.00	2	TE120
210	JavaScript	300.00	3	TE210
220	Perl Programming	400.00	5	TE220

4 rows in set (0.00 sec)

IN and NOT IN Operators

- The **IN** operator selects rows from a set of alternatives.
 - The **NOT IN** operator selects all those records that are not in the set of alternatives.
 - ◆ The set of alternatives is a comma separated list of values inside of parenthesis after the **IN** or **NOT IN** operator.

in_not_in.sql

```

1 SELECT employee_id, start_date, course_id, customer_id, location
2   FROM Class
3   WHERE location IN ('NYC', 'Las Vegas, NV');
4
5 SELECT employee_id, start_date, course_id, customer_id, location
6   FROM Class
7   WHERE location NOT IN ('NYC', 'Las Vegas, NV');
```

mysql> SOURCE in_not_in.sql;

employee_id	start_date	course_id	customer_id	location
507	2005-02-15	200	1000	Las Vegas, NV
503	2005-02-14	100	1010	NYC
507	2005-02-21	210	1000	Las Vegas, NV
503	2005-02-28	100	1000	Las Vegas, NV
506	2005-02-28	110	1010	NYC

5 rows in set (0.00 sec)

employee_id	start_date	course_id	customer_id	location
506	2005-02-14	100	1020	Manchester, CT
500	2005-02-21	220	1030	Columbia, MD
504	2005-02-21	300	1020	Manchester, CT
500	2005-02-28	220	1040	Columbia, MD
504	2005-03-07	300	1040	Columbia, MD

5 rows in set (0.00 sec)

BETWEEN Operator

- The **BETWEEN** operator values within a specified range.
 - ▶ The complimentary **NOT BETWEEN** operator selects those rows that are not within the specified range.
 - ◆ The range should always be specified from low to high for both the **BETWEEN** and **NOT BETWEEN** operators.

between.sql

```

1 /* The following finds all of the classes that did not end
2    in the month of February 2005 */
3
4 SELECT class_id, course_id, start_date, end_date FROM Class
5    WHERE end_date NOT BETWEEN '2005-02-01' AND '2005-02-28';
6
7 /* Rewrite of the previous SELECT statement:
8    SELECT * FROM Course WHERE unit_price > 250.00 && unit_price < 500.00;
9    To use BETWEEN to get similar type of results */
10 SELECT * FROM Course WHERE unit_price BETWEEN 251.00 AND 499.00;
```

mysql> SOURCE between.sql;

class_id	course_id	start_date	end_date
5060	100	2005-02-28	2005-03-04
5070	110	2005-02-28	2005-03-01
5080	220	2005-02-28	2005-03-04
5090	300	2005-03-07	2005-03-09

4 rows in set (0.00 sec)

course_id	name	unit_price	num_days	courseware_id
110	JSPs	400.00	2	TE110
120	Servlets	400.00	2	TE120
210	JavaScript	300.00	3	TE210
220	Perl Programming	400.00	5	TE220

4 rows in set (0.00 sec)

The LIKE Operator

- One of the nice features of SQL is its ability to perform simple and complex pattern matching.
 - The need for pattern matching arises with the need to often retrieve some information where an exact match is either impossible or undesirable.
 - ◆ For example, you might want to list those rows whose names are either Smith, Smyth, Smythe, or possibly any names having "Sm" at the beginning.
- In this case, you can use the special pattern matching abilities of SQL.
- Simple pattern matching is accomplished by using the LIKE operator.
 - More complex matching operations can be performed using regular expressions, but we will concentrate on basic pattern matching with the LIKE operator.
 - ◆ A more in-depth discussion of matching with regular expressions can be found in the MySQL documentation at the following URL.

<https://dev.mysql.com/doc/refman/5.7/en/pattern-matching.html>

- When the LIKE operator is used, one of two special characters must be used as placeholders within the defined expression.
 - %
 - ◆ The % character implies zero or more characters.
 - _
 - ◆ The _ character (underscore) means exactly one character.
- All pattern matches with the LIKE operator are case-insensitive by default.
- The following example contains several SELECT queries to demonstrate various patterns that can be matched using the LIKE operator.

pattern_matching.sql

```

1 /* List the employees whose phone number starts with 410 */
2 SELECT first_name, last_name, phone FROM Employee
3     WHERE phone LIKE '410%';
4
5 /* List all courseware that has Java anywhere in its name */
6 SELECT courseware_id, name FROM Courseware
7     WHERE name LIKE '%Java%';
8
9 /* List only the Employees that have exactly 5 characters in their
10    first name.
11       Note: The string after LIKE below has exactly five
12          underscores in a row */
13 SELECT first_name, last_name FROM Employee
14     WHERE first_name LIKE '_____';

```

```
mysql> SOURCE pattern_matching.sql;
```

```

+-----+-----+-----+
| first_name | last_name | phone      |
+-----+-----+-----+
| Kim        | Perkins   | 4105557985 |
| Alex       | Hardin    | 4105557648 |
| Tina       | Johnson   | 4105553274 |
| Maria      | Higgins   | 4105552105 |
| Connie     | McKinney  | 4105552544 |
| Travis     | Train     | 4105553512 |
+-----+-----+-----+
6 rows in set (0.00 sec)

+-----+-----+
| courseware_id | name                |
+-----+-----+
| TE100         | Introduction to Java |
| TE210         | JavaScript           |
+-----+-----+
2 rows in set (0.00 sec)

+-----+-----+
| first_name | last_name |
+-----+-----+
| Emily      | Wilson    |
| Maria      | Higgins   |
| Brian      | Perkins   |
+-----+-----+
3 rows in set (0.00 sec)

```

- One way to perform a case sensitive pattern match is to convert the data being matched to binary as shown in the following example.

case_sensitivity.sql

```

1 SELECT first_name, last_name FROM Employee
2   WHERE last_name LIKE '%h%';
3
4 SELECT first_name, last_name FROM Employee
5   WHERE BINARY last_name LIKE '%h%';
6
7 SELECT first_name, last_name FROM Employee
8   WHERE BINARY last_name LIKE '%H%';

```

```
mysql> SOURCE case_sensitivity.sql;
```

```

+-----+-----+
| first_name | last_name |
+-----+-----+
| Alex      | Hardin   |
| Tina      | Johnson  |
| Maria     | Higgins  |
+-----+-----+
3 rows in set (0.00 sec)

```

```

+-----+-----+
| first_name | last_name |
+-----+-----+
| Tina      | Johnson  |
+-----+-----+
1 row in set (0.00 sec)

```

```

+-----+-----+
| first_name | last_name |
+-----+-----+
| Alex      | Hardin   |
| Maria     | Higgins  |
+-----+-----+
2 rows in set (0.00 sec)

```

Logical Operators

- In addition to the comparison operators and some of the special operators that we have seen so far, MySQL also has logical operators.
 - The table below lists the logical operators supported by MySQL.

Operator	Description
AND, &&	Logical AND
NOT, !	Negates Value
OR,	Logical OR
XOR	Logical XOR

- The following examples demonstrate the above logical operators.
 - The **AND** operator.
 - ◆ This operator was seen earlier in the chapter in the last statement in the *strings_dates_numbers.sql* example.
 - ◆ Here it is rewritten using the **&&** operator.

and_operator.sql

```
1 SELECT * FROM Course WHERE unit_price > 250.00 && unit_price < 500.00;
```

```
mysql> SOURCE and_operator.sql;
```

```
+-----+-----+-----+-----+-----+
| course_id | name           | unit_price | num_days | courseware_id |
+-----+-----+-----+-----+-----+
| 110       | JSPs           | 400.00     | 2        | TE110          |
| 120       | Servlets       | 400.00     | 2        | TE120          |
| 210       | JavaScript     | 300.00     | 3        | TE210          |
| 220       | Perl Programming | 400.00     | 5        | TE220          |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

- The **OR** operator.

or_operator.sql

```
1 SELECT location, unit_price
2    FROM Class WHERE location = 'NYC' OR unit_price < 400.00;
```

```
mysql> SOURCE or_operator.sql;
```

```
+-----+-----+
| location | unit_price |
+-----+-----+
| Las Vegas, NV | 250.00 |
| NYC | 500.00 |
| Las Vegas, NV | 250.00 |
| NYC | 350.00 |
+-----+-----+
4 rows in set (0.00 sec)
```

- The **NOT** operator.

not_operator.sql

```
1 SELECT customer_id, location, comments
2    FROM Class WHERE comments NOT LIKE '__class%';
```

```
mysql> SOURCE not_operator.sql;*
```

```
+-----+-----+-----+
| customer_id | location | comments |
+-----+-----+-----+
| 1000 | Las Vegas, NV | New client |
| 1010 | NYC | Repeat cust. |
| 1020 | Manchester, CT | New client |
| 1030 | Columbia, MD | Repeat cust. |
| 1040 | Columbia, MD | Repeat cust. |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

IS NULL and IS NOT NULL

- If a column is not constrained by a **NOT NULL** constraint, then the column value can be assigned a value of **NULL**.
 - This special value is not the same as the numeric value **0**.
 - ◆ A side effect of the **NULL** value is that if you try to select all non-zero values from a column, you will not get the **NULL** values as part of the result set.

nulls_missing.sql

```
1 SELECT * FROM Course WHERE unit_price != 500;
```

```
mysql> SOURCE nulls_missing.sql;
```

course_id	name	unit_price	num_days	courseware_id
110	JSPs	400.00	2	TE110
120	Servlets	400.00	2	TE120
200	XHTML	250.00	2	TE200
210	JavaScript	300.00	3	TE210
220	Perl Programming	400.00	5	TE220

5 rows in set (0.00 sec)

- In order to retrieve all the records whose *unit_price* value is not **500.00** (including **NULL**), an additional **IS NULL** test is needed.

nulls_included.sql

```
1 SELECT * FROM Course WHERE unit_price != 500 or unit_price IS NULL;
```

```
mysql> SOURCE nulls_included.sql;
```

course_id	name	unit_price	num_days	courseware_id
110	JSPs	400.00	2	TE110
120	Servlets	400.00	2	TE120
200	XHTML	250.00	2	TE200
210	JavaScript	300.00	3	TE210
220	Perl Programming	400.00	5	TE220
400	Fortran	NULL	NULL	NULL
410	PL1	NULL	NULL	NULL

```
7 rows in set (0.00 sec)
```

CASE Statements

- **CASE** statements provide the ability to construct complex conditional statements.
 - ▶ Simple **CASE** statements shown here can only compare equality of values
 - ▶ The generic syntax of a simple **CASE** statement is:

```
CASE some_value
  WHEN value1 THEN statements
  WHEN value2 THEN other_statements
  ...
  ELSE else_statements
END ;
```

- Searched **CASE** statements can compare the result of a conditional statement.
 - ▶ The generic syntax of a searched **CASE** statement is:

```
CASE
  WHEN some_condition THEN statements
  WHEN some_condition2 THEN other_statements
  ...
  ELSE else_statements
END CASE;
```

- The first value that is met will execute, all other **WHEN** values are ignored.
- If no values are met, the **ELSE** part executes.

case.sql

```

1 SELECT name,
2 CASE num_days
3     WHEN 5 THEN "All week"
4     WHEN 3 THEN "Most of the week"
5     WHEN 2 THEN "Couple of days"
6     ELSE "No idea"
7 END AS 'Duration Description'
8 FROM Course;
9
10 SELECT name,
11 CASE
12     WHEN num_days = 5 THEN "All week"
13     WHEN num_days = 3 THEN "Most of the week"
14     WHEN num_days = 2 THEN "Couple of days"
15     ELSE "No idea"
16 END AS 'Duration Description'
17 FROM Course;

```

- Both queries would yield the following result.

name	Duration Description
Java Programming	All week
JSPs	Couple of days
Servlets	Couple of days
XHTML	Couple of days
JavaScript	Most of the week
Perl Programming	All week
Oracle SQL	Most of the week
Fortran	No idea
PL1	No idea

9 rows in set (0.00 sec)

Exercises

Exercise 1

List the last and first name of all employees in ascending alphabetical order by last name.

Exercise 2

List the full name and phone number of all employees who are within the 410 area code and are currently employed.

Exercise 3

List the name of all sub-contractors in the customers database and the employee id of the salesperson in charge of their account.

Exercise 4

List the salary and bonus of the employee named Alex Hardin.

Exercise 5

List the start date, class id, and course id of classes that are being offered in 'NYC' or 'Las Vegas, NV'.

- Use the **IN** operator in your SQL statement.

Exercise 6

List the start date, class id, and course id of classes that are being offered in 'NYC' or 'Las Vegas, NV'.

- Use a logical operator in combination with the equality operator in your SQL statement.

Exercise 7

List the first and last names of the employees who have been assigned a mentor.

Chapter 8. SQL Functions

Objectives

- Introduction
- The DISTINCT Keyword
- Aliases
- Miscellaneous Functions
- Mathematical Functions
- String Functions
- Date Functions

Introduction

Most of the queries that we have seen have extracted row or column information FROM various tables.

- However, many applications need other information such as:
 - How many rows are in a table?
 - What is the largest element in this column?

Since these and similar queries are common over all applications, the designers of SQL were careful to include many functions to solve problems like these.

- These functions can be categorized as follows.
 - Miscellaneous Functions
 - Mathematical Functions
 - String Functions
 - Date Functions
 - Conversion Functions

Before we examine these functions, we need to examine some preliminary concepts that will help us in the discussions that follow.

- The **DISTINCT** Keyword
- Aliases
- The **DUAL** Table

The **DISTINCT** Keyword

Notice the list of locations FROM the *Class* table.

locations.sql

```
1 SELECT location FROM Class;
```

```
mysql> SOURCE locations.sql;
```

```
+-----+
| location |
+-----+
| Las Vegas, NV |
| NYC |
| Manchester, CT |
| Las Vegas, NV |
| Columbia, MD |
| Manchester, CT |
| Las Vegas, NV |
| NYC |
| Columbia, MD |
| Columbia, MD |
+-----+
10 rows in set (0.00 sec)
```

To list only the unique job names, use the **DISTINCT** keyword.

locations_distinct.sql

```
1 SELECT DISTINCT location FROM Class;
```

```
mysql> SOURCE locations_distinct.sql;
```

```
+-----+
| location |
+-----+
| Las Vegas, NV |
| NYC |
| Manchester, CT |
| Columbia, MD |
+-----+
4 rows in set (0.00 sec)
```

Aliases

As you have seen, each time a query is displayed, the column head matches the fields that were requested in the query.

without_aliases.sql

```
1 SELECT first_name, last_name, hire_date
2   FROM Employee
3   WHERE last_name = 'Wilson';
```

```
mysql> SOURCE without_aliases.sql;
+-----+-----+-----+
| first_name | last_name | hire_date |
+-----+-----+-----+
| Morgan     | Wilson    | 1990-06-10 |
| Emily      | Wilson    | 1995-01-01 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

Using the alias feature of SQL, you can change the names of the displayed fields.

- The alias feature optionally uses the keyword **AS**.

with_aliases.sql

```
1 SELECT first_name AS 'First Name', last_name AS 'Last Name', hire_date AS 'Start Date'
2   FROM Employee
3   WHERE last_name = 'Wilson';
```

```
mysql> SOURCE with_aliases.sql;
+-----+-----+-----+
| First Name | Last Name | Start Date |
+-----+-----+-----+
| Morgan     | Wilson    | 1990-06-10 |
| Emily      | Wilson    | 1995-01-01 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

Miscellaneous Functions

The `NOW()` function yields the current system date and time.

- Even though the result of this function is not stored in a table, a `SELECT` is still needed to see the result.
- Moreover, since you normally cannot use `SELECT` without using a table, the output might be a little deceiving.

now_function.sql

```
1 SELECT NOW() FROM Team;
```

```
mysql> SOURCE now_function.sql;
```

```
+-----+
| NOW() |
+-----+
| 2019-11-15 02:52:56 |
| 2019-11-15 02:52:56 |
| 2019-11-15 02:52:56 |
| 2019-11-15 02:52:56 |
+-----+
4 rows in set (0.00 sec)
```

MySQL maintains a convenience table named `DUAL`.

- `DUAL` is used with various simple SQL functions to streamline the output display.
- The previous query would be cleaner if the `DUAL` table was used.
 - Recent versions of MySQL make the explicit use of `DUAL` optional.

dual_table.sql

```
1 SELECT NOW() FROM DUAL;
2
3 SELECT NOW() as 'Date and Time' FROM DUAL;
4
5 SELECT NOW() as 'Date and Time'
```

The output of the above script is shown on the following page

```
mysql> SOURCE dual_table.sql;
```

```
+-----+
| NOW()          |
+-----+
| 2019-11-15 03:00:20 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+
| Date and Time    |
+-----+
| 2019-11-15 03:00:20 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+
| Date and Time    |
+-----+
| 2019-11-15 03:00:20 |
+-----+
1 row in set (0.00 sec)
```

Other simple functions can be used in this way as well.

For example, notice the results from the **USER** function and the arbitrary calculation in the example below.

user_function.sql

```
1 SELECT USER() as 'User', 4 * 5 as 'Result';
```

```
mysql> SOURCE user_function.sql;
```

```
+-----+-----+
| User          | Result |
+-----+-----+
| student@localhost | 20 |
+-----+-----+
1 row in set (0.00 sec)
```


Mathematical Functions

The simplest mathematical function is `COUNT`, which returns the number of rows in a query.

math_functions.sql

```
1 # Count all rows of a table
2 SELECT COUNT(*) AS '# of Employees' FROM Employee;
3
4 # Count all non-null values in a column
5 SELECT COUNT(bonus) AS '# of Bonuses' FROM Employee;
6
7 # Count all records that meet a certain criteria
8 SELECT COUNT(gender) AS '# of Female Employees'
9 FROM Employee
10 WHERE gender = 'F';
11
12 # Count Distinct occurrences
13 SELECT COUNT(DISTINCT location) AS '# of Locations' FROM Class;
14
15 # The MAX and MIN functions are easy to describe and are typically applied to a
    column.
16 SELECT MIN(salary) AS 'Min', MAX(salary) AS 'Max' FROM Employee;
```

The output from the example above is shown on the following page.

```
mysql> SOURCE math_functions.sql;
```

```
+-----+
| # of Employees |
+-----+
|           10 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+
| # of Bonuses |
+-----+
|           9 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+
| # of Female Employees |
+-----+
|           6 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+
| # of Locations |
+-----+
|           4 |
+-----+
1 row in set (0.00 sec)
```

```
+-----+-----+
| Min  | Max  |
+-----+-----+
| 25000 | 50000 |
+-----+-----+
1 row in set (0.00 sec)
```

The **SUM** function can be used to SUM the elements of a column.

The **AVG** function can be used to average the elements of a column.

- These two functions are demonstrated in the example on the following page.

sums_and_averages.sql

```
1 SELECT COUNT(*) 'Employees', SUM(salary) 'Total Salaries' FROM Employee;
2
3 SELECT COUNT(*) 'Employees', AVG(salary) 'Average Salary' FROM Employee;
4
5 SELECT SUM(salary) 'Total Salaries', AVG(salary) 'Average Salary' FROM Employee;
```

```
mysql> SOURCE sums_and_averages.sql;
```

```
+-----+-----+
| Employees | Total Salaries |
+-----+-----+
|          10 |          375000 |
+-----+-----+
1 row in set (0.00 sec)
```

```
+-----+-----+
| Employees | Average Salary |
+-----+-----+
|          10 |      37500.0000 |
+-----+-----+
1 row in set (0.00 sec)
```

```
+-----+-----+
| Total Salaries | Average Salary |
+-----+-----+
|          375000 |      37500.0000 |
+-----+-----+
1 row in set (0.00 sec)
```

String Functions

Most programming languages offer a variety of functions that operate on strings.

- Many of these same functions are offered by SQL.

The **LENGTH** function can be used to discover the length of any field.

length_function.sql

```
1 SELECT first_name, LENGTH(first_name) AS '# of letters',
2      last_name, LENGTH(last_name) AS '# of letters',
3      bonus, LENGTH(bonus)
4 FROM Employee;
```

```
mysql> SOURCE length_function.sql;
```

first_name	# of letters	last_name	# of letters	bonus	LENGTH(bonus)
Morgan	6	Wilson	6	10000	5
Emily	5	Wilson	6	10000	5
Kim	3	Perkins	7	10000	5
Alex	4	Hardin	6	7500	4
Tina	4	Johnson	7	10000	5
Rachel	6	Fitzgerald	10	5000	4
Maria	5	Higgins	7	5000	4
Brian	5	Perkins	7	5000	4
Connie	6	McKinney	8	5000	4
Travis	6	Train	5	NULL	NULL

10 rows in set (0.00 sec)

The **SUBSTR** function returns a subset of its first argument as a string.

- The second argument to the function is the starting index.
- The third argument to the function is how many characters to return.
 - ▶ If the third argument is not supplied all remaining characters from the starting index are returned.

substrings.sql

```

1 SELECT first_name, last_name, SUBSTR(hire_date, 1, 4) '4 Digit Year',
2     SUBSTR(hire_date, 3, 2) '2 Digit Year'
3 FROM Employee WHERE last_name = 'Wilson';
4
5 SELECT SUBSTR(first_name, 1, 1) 'First Initial', SUBSTR(last_name, 1, 1) 'Last Initial'
6 FROM Employee WHERE last_name = 'Wilson';
7
8 SELECT first_name, last_name, SUBSTR(phone, 4) 'Area Code 410'
9 FROM Employee WHERE SUBSTR(phone, 1, 3) = '410';

```

The output of the above example is shown on the following page.

```
mysql> SOURCE substrings.sql;
```

```

+-----+-----+-----+-----+
| first_name | last_name | 4 Digit Year | 2 Digit Year |
+-----+-----+-----+-----+
| Morgan     | Wilson    | 1990         | 90           |
| Emily      | Wilson    | 1995         | 95           |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

```

+-----+-----+
| First Initial | Last Initial |
+-----+-----+
| M             | W            |
| E             | W            |
+-----+-----+
2 rows in set (0.00 sec)

```

```

+-----+-----+-----+
| first_name | last_name | Area Code 410 |
+-----+-----+-----+
| Kim        | Perkins  | 5557985       |
| Alex       | Hardin   | 5557648       |
| Tina       | Johnson  | 5553274       |
| Maria      | Higgins  | 5552105       |
| Connie     | McKinney | 5552544       |
| Travis     | Train    | 5553512       |
+-----+-----+-----+
6 rows in set (0.00 sec)

```

The **CONCAT** function accepts multiple arguments and "glues" them together as a single string.

concatenation.sql

```
1 SELECT CONCAT(first_name, ' ', last_name) AS 'Name',  
2      CONCAT(SUBSTR(first_name, 1, 1), '.', SUBSTR(last_name, 1, 1), '.') AS  
   'Initials'  
3      FROM Employee;
```

```
mysql> SOURCE dual_tables.sql;
```

Name	Initials
Morgan Wilson	M.W.
Emily Wilson	E.W.
Kim Perkins	K.P.
Alex Hardin	A.H.
Tina Johnson	T.J.
Rachel Fitzgerald	R.F.
Maria Higgins	M.H.
Brian Perkins	B.P.
Connie McKinney	C.M.
Travis Train	T.T.

10 rows in set (0.00 sec)

Date Functions

There are various ways to operate on the **DATE** data type.

- All of the relational operators can be used to compare dates.
- Likewise, the **MIN** and **MAX** functions can be used to compare dates.
- In addition, you can add or subtract from the date type as shown below.

dates.sql

```
1 SELECT CURRENT_DATE AS Today,
2     DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) AS Yesterday,
3     DATE_ADD(CURRENT_DATE, INTERVAL 1 DAY) AS Tomorrow,
4     DATE_ADD(CURRENT_DATE, INTERVAL 6 MONTH) AS Future;
```

```
mysql> SOURCE dates.sql;
```

```
+-----+-----+-----+-----+
| Today      | Yesterday | Tomorrow | Future      |
+-----+-----+-----+-----+
| 2019-11-15 | 2019-11-14 | 2019-11-16 | 2020-05-15 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

The **EXTRACT** function can be used to extract various components (year, month, etc.) from a **DATE** data type.

extracting.sql

```
1 SELECT first_name, last_name, EXTRACT(YEAR FROM hire_date) AS Year,
2     EXTRACT(MONTH FROM hire_date) AS Month, EXTRACT(DAY FROM hire_date) AS Day
3 FROM Employee
4 WHERE EXTRACT(YEAR FROM hire_date) < 2000;
```

```
mysql> SOURCE extracting.sql;
```

```
+-----+-----+-----+-----+-----+
| first_name | last_name | Year | Month | Day |
+-----+-----+-----+-----+-----+
| Morgan    | Wilson   | 1990 | 6     | 10  |
| Emily     | Wilson   | 1995 | 1     | 1   |
| Kim       | Perkins  | 1997 | 5     | 21  |
| Alex      | Hardin   | 1998 | 4     | 9   |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```


The **LAST_DAY** function returns the last day of the month of the given argument as a **DATE**.

The **DATEDIFF** function returns the number of days between two dates.

more_dates.sql

```
1 SELECT LAST_DAY(CURRENT_DATE) AS "LAST DAY OF MONTH", CURRENT_DATE as TODAY;
2
3 SELECT DATEDIFF(LAST_DAY(CURRENT_DATE), CURRENT_DATE) as "DAYS REMAINING";
```

```
mysql> SOURCE more_dates.sql;
```

```
+-----+-----+
| LAST DAY OF MONTH | TODAY      |
+-----+-----+
| 2019-11-30        | 2019-11-15 |
+-----+-----+
1 row in set (0.00 sec)
```

```
+-----+
| DAYS REMAINING |
+-----+
|              15 |
+-----+
1 row in set (0.01 sec)
```

Exercises

Exercise 1

Use the **DUAL** table to display the current system date and your user name.

Exercise 2

List the id, name, and phone number of all the direct clients in the **Customer** table.

- Limit the search to the names of the clients who are no longer than 20 characters.

Exercise 3

What is the average salary of all married persons in the **Employee** table?

Exercise 4

In how many different cities are classes being offered?

Exercise 5

How many women are listed in the **Employee** table?

Exercise 6

List the start date, end date, class id, and total cost of each class offered in the week of February 14th, 2005.

- This unit price is the price per day of the class.

Exercise 7

List the start date, end date, class id and total cost of each class whose total is greater than \$2000.

More Exercises

The following exercises will use the MySQL Sample Database (**classicmodels**), not the training database that had been previously used.

Exercise 1

From the *orderDetails* table, create a query that shows the total of the number of products ordered and the total price of all the orders.

Exercise 2

From the *payments* table, calculate the total and the count of all payments made.

Chapter 9. Joining Tables

Objectives

- Pull data from multiple tables using joins
- Use non-equi joins
- Use inner and outer joins

Joins

Relational databases consist of data stored in multiple tables.

Because some requests require information FROM more than one table, it is a common occurrence to reference more than one table within the same request.

The next three chapters deal with different methods used to display data FROM multiple tables.

This chapter focuses on using joins to display data FROM various tables. There are many different types of joins, but they all fall into the following two categories.

- Inner joins
- Outer joins

Joins are accomplished using a **SELECT** statement, the main difference being that the **FROM** clause will contain more than one table name.

- We will begin by examining inner joins, which are the most common.

An **INNER JOIN** combines column values from the rows in two tables based on some condition (or **JOIN predicate**).

- There are two different ways of defining an inner join:
 - ▶ Traditionally an inner join can be created by defining multiple tables separated by a comma in the **FROM** clause.
 - ◆ Any relationships between the tables are specified in the **WHERE** clause.
 - ▶ More recently, the standardized way of defining an inner join is to use **INNER JOIN** between the table names in the **FROM** clause.
 - ◆ Any relationships between the tables are typically specified on the **ON** clause after the **FROM**.

Cross Join

In the case where two or more tables are listed in the **FROM** or **INNER JOIN** clause and there is no corresponding **WHERE** or **ON** clause respectively, the tables are "multiplied."

- This operation is called a **cross join** (or Cartesian product).
 - A cross join results in rows that contain each row of one table concatenated with each row of another table.
 - Therefore, if one table has four rows as Team does, and another has ten rows as Employee does, then the Cartesian product of teams and courses would have 40 rows. (4 rows x 10 rows).

The queries below are terminated with a **\G** instead of a semicolon.

- This displays the results vertically which sometimes can make the result more readable.
 - The use of **\G** as the terminator is specific to MySQL and will not work with other RDBMS.
 - It also will not work with other tools necessarily, such as MySQL Workbench.

The output below shows the number of rows in the **Team** table, **Employee** table, and their cross join.

```
mysql> SELECT COUNT(*) AS 'Team Records' FROM Team\G
***** 1. row *****
Team Records: 4
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) AS 'Employee Records' FROM Employee\G
***** 1. row *****
Employee Records: 10
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) AS 'Cross Joined Records' FROM Team, Employee\G
***** 1. row *****
Cross Joined Records: 40
1 row in set (0.00 sec)

mysql> exit
Bye
```

The following example shows several fields from the tables in the resulting cross join

large_cross_join.sql

```
1 SELECT name, employee_id, first_name, last_name, hire_date
2 FROM Team INNER JOIN Employee;
```

A sampling of the output is shown below.

```
mysql> SOURCE large_cross_join.sql;
+-----+-----+-----+-----+-----+
| name      | employee_id | first_name | last_name | hire_date |
+-----+-----+-----+-----+-----+
| Education | 500         | Morgan    | Wilson    | 1990-06-10 |
| Management | 500         | Morgan    | Wilson    | 1990-06-10 |
| Systems    | 500         | Morgan    | Wilson    | 1990-06-10 |
| Sales      | 500         | Morgan    | Wilson    | 1990-06-10 |
| Education | 501         | Emily     | Wilson    | 1995-01-01 |
| Management | 501         | Emily     | Wilson    | 1995-01-01 |
| Systems    | 501         | Emily     | Wilson    | 1995-01-01 |
| Sales      | 501         | Emily     | Wilson    | 1995-01-01 |
| Education | 502         | Kim       | Perkins   | 1997-05-21 |
|           |             |           |           |           |
|           |             |           |           |           |
|           |             |           |           |           |
| Education | 507         | Connie    | McKinney  | 2001-03-21 |
| Management | 507         | Connie    | McKinney  | 2001-03-21 |
| Systems    | 507         | Connie    | McKinney  | 2001-03-21 |
| Sales      | 507         | Connie    | McKinney  | 2001-03-21 |
| Education | 509         | Travis    | Train     | 2004-05-31 |
| Management | 509         | Travis    | Train     | 2004-05-31 |
| Systems    | 509         | Travis    | Train     | 2004-05-31 |
| Sales      | 509         | Travis    | Train     | 2004-05-31 |
+-----+-----+-----+-----+-----+
40 rows in set (0.00 sec)
```

- The above results are only a sampling of the 40 rows from the previous query.

Technically since it is a cross join the standard syntax would be to use **CROSS JOIN** instead of **INNER JOIN**

large_cross_join_again.sql

```
1 SELECT name, employee_id, first_name, last_name, hire_date
2 FROM Team CROSS JOIN Employee;
```

Inner joins

While a cross join is rarely useful, it reveals the way that SQL processes inner joins.

- A more useful inner join would have a **ON** or **WHERE** clause to conditionally display certain data *based on values common to each table* in the join.

Inner joins are used primarily to display data based on the relationships between tables.

- Inner joins typically use foreign keys and primary keys, which are used to implement the relationships between business entities.

Inner joins can be categorized as follows.

- Equi-join
- Non-equi join
- Non-key join
- Self-join
- Natural join

We will begin by examining the equi-join.

Equi-Join

An **equi-join** requires all the conditional expressions in the **ON** or **WHERE** clause use only the *equality* operator.

- A simple example of an inner equi-join would be to list the first name, last name, and team name of all the employees in the *Employee* table.
 - An examination of the *Employee* table reveals a *team_id* field.
 - The *team_id* is a foreign key in *Employee* that links to the *team_id* primary key in *Team*.
- Note that if N tables are being compared, then there must exist a minimum of N - 1 expressions of equality in the **ON** or **WHERE** clause.
 - Since two tables are being compared, we need at least one equality condition for an equi-join.

equi_join_01.sql

```
1 SELECT first_name, last_name, name AS 'Team Name'
2 FROM Employee INNER JOIN Team ON Employee.team_id = Team.team_id;
```

```
mysql> SOURCE equi_join_01.sql;
```

```
+-----+-----+-----+
| first_name | last_name | Team Name |
+-----+-----+-----+
| Morgan    | Wilson   | Education |
| Emily     | Wilson   | Management |
| Kim       | Perkins  | Systems   |
| Alex      | Hardin   | Education |
| Tina      | Johnson  | Sales     |
| Rachel    | Fitzgerald | Education |
| Maria     | Higgins  | Sales     |
| Brian     | Perkins  | Education |
| Connie    | McKinney | Management |
| Travis    | Train    | Systems   |
+-----+-----+-----+
10 rows in set (0.00 sec)
```

The example below demonstrates the use of **FROM ... WHERE** as opposed to **FROM ... INNER JOIN ... ON**

equi_join_02.sql

```
1 SELECT first_name, last_name, name AS 'Team Name'  
2 FROM Employee, Team WHERE Employee.team_id = Team.team_id;
```

Another example of an equi-join would be to list the start date, end date, and course name of all the classes in the *Class* table.

- An examination of the *Class* table reveals a *course_id* field.
- The *course_id* is a foreign key in *Class* that links to the *course_id* primary key in *Course*.
 - Since the *course_id* field is the same name in both tables, the table name is required in front of the field to distinguish between them in the **ON**.

dates_and_names.sql

```
1 SELECT name, start_date, end_date
2 FROM Class INNER JOIN Course ON Class.course_id = Course.course_id;
```

```
mysql> SOURCE dates_and_names.sql;
```

name	start_date	end_date
XHTML	2005-02-15	2005-02-16
Java Programming	2005-02-14	2005-02-18
Java Programming	2005-02-14	2005-02-18
JavaScript	2005-02-21	2005-02-23
Perl Programming	2005-02-21	2005-02-25
Oracle SQL	2005-02-21	2005-02-23
Java Programming	2005-02-28	2005-03-04
JSPs	2005-02-28	2005-03-01
Perl Programming	2005-02-28	2005-03-04
Oracle SQL	2005-03-07	2005-03-09

10 rows in set (0.00 sec)

Now suppose we wish to add the name of the instructor for that class to the query.

- The instructor name appears in the *Employee* table.
 - Therefore a relationship between the *Employee* table and the *Class* table is needed.
 - The *employee_id* field of both tables fits this need.

The query and the results are shown below.

dates_names_instructors.sql

```
1 SELECT name, start_date, end_date,
2       CONCAT(first_name, ' ', last_name) AS Instructor
3 FROM Class INNER JOIN Course ON Class.course_id = Course.course_id
4       INNER JOIN Employee ON Class.employee_id = Employee.employee_id;
```

```
mysql> SOURCE dates_names_instructors.sql;
```

name	start_date	end_date	Instructor
Perl Programming	2005-02-21	2005-02-25	Morgan Wilson
Perl Programming	2005-02-28	2005-03-04	Morgan Wilson
Java Programming	2005-02-14	2005-02-18	Alex Hardin
Java Programming	2005-02-28	2005-03-04	Alex Hardin
Oracle SQL	2005-02-21	2005-02-23	Rachel Fitzgerald
Oracle SQL	2005-03-07	2005-03-09	Rachel Fitzgerald
Java Programming	2005-02-14	2005-02-18	Brian Perkins
JSPs	2005-02-28	2005-03-01	Brian Perkins
XHTML	2005-02-15	2005-02-16	Connie McKinney
JavaScript	2005-02-21	2005-02-23	Connie McKinney

10 rows in set (0.00 sec)

Non-Equi Join

A **non-equi join** is one in which at least one part of the condition for the join is *not* an equality operation.

- For example, listing the start date and end date of all classes that are not named 'Java Programming'.

non_equi_join.sql

```
1 SELECT start_date, end_date, name
2 FROM Class INNER JOIN Course
3 ON Course.course_id = Class.course_id AND
4    name <> 'Java Programming';
```

```
mysql> SOURCE non_equi_join.sql;
```

```
+-----+-----+-----+
| start_date | end_date | name          |
+-----+-----+-----+
| 2005-02-15 | 2005-02-16 | XHTML        |
| 2005-02-21 | 2005-02-23 | JavaScript     |
| 2005-02-21 | 2005-02-25 | Perl Programming |
| 2005-02-21 | 2005-02-23 | Oracle SQL     |
| 2005-02-28 | 2005-03-01 | JSPs           |
| 2005-02-28 | 2005-03-04 | Perl Programming |
| 2005-03-07 | 2005-03-09 | Oracle SQL     |
+-----+-----+-----+
7 rows in set (0.00 sec)
```

Non-Key Join

A **non-key join** compares non-key fields between tables to create its result set.

- An example is a query that finds all classes that start after an instructor's hire date.
 - The example selects classes that have a start date after Rachel's hire date.

non_key_join.sql

```
1 SELECT class_id, start_date
2 FROM Class INNER JOIN Employee
3 ON hire_date <= start_date AND
4    first_name = 'Rachel' AND
5    last_name = 'Fitzgerald';
```

```
mysql> SOURCE non_key_join.sql;
```

```
+-----+-----+
| class_id | start_date |
+-----+-----+
|      5090 | 2005-03-07 |
+-----+-----+
1 row in set (0.00 sec)
```

The following example includes the name of the class, requiring a second join with another **ON** to avoid a cross join.

non_key_join_again.sql

```
1 SELECT class_id, name, start_date
2 FROM Class
3 INNER JOIN Employee ON hire_date <= start_date AND
4                     first_name = 'Rachel' AND
5                     last_name = 'Fitzgerald'
6 INNER JOIN Course ON Class.course_id = Course.course_id;
```



```
mysql> SOURCE non_key_join_again.sql;
+-----+-----+-----+
| class_id | name      | start_date |
+-----+-----+-----+
|      5090 | Oracle SQL | 2005-03-07 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Self Join

A **self-join** (or **reflexive join**) is joining a table to itself.

- A self-join may only occur in a table where a foreign key refers to a primary key.
 - The only table in the sample database that fits that description is the *Employee* table.
 - The *Employee* table contains a *mentor_id* field that stores the *employee_id* of another employee.
- To list all the employees and their mentors, we can use a reflexive join.

Since a reflexive join contains two or more references to the same table it requires the use of table aliases.

- A table alias is defined after the table names in a similar way to the field aliases shown in previous examples.
- The next example creates a table alias named *Mentor* for the *Employee* table.
 - This permits the *Employee* table to be referenced as if it were two different tables.

self_join.sql

```
1 SELECT CONCAT(Employee.first_name, ' ', Employee.last_name) 'Employee',
2        CONCAT(Mentor.first_name, ' ', Mentor.last_name) 'Mentor'
3 FROM Employee AS Mentor INNER JOIN Employee
4 WHERE Employee.mentor_id = Mentor.employee_id;
```

```
mysql> SOURCE self_join.sql;
```

```
+-----+-----+
| Employee      | Mentor      |
+-----+-----+
| Rachel Fitzgerald | Morgan Wilson |
| Maria Higgins   | Tina Johnson |
| Brian Perkins   | Alex Hardin  |
| Connie McKinney | Emily Wilson |
| Travis Train    | Kim Perkins  |
+-----+-----+
5 rows in set (0.00 sec)
```

Natural Join

A special kind of equi-join is a **NATURAL JOIN**.

- A **NATURAL JOIN** is an operation that creates an implicit join clause based on the common columns in the two tables being joined.
 - Common columns are columns that have the exact same name in both tables.

The following example does a **NATURAL JOIN** between the *Employee* table and the *Team* table.

- The natural join is based on both tables having a column called *team_id*.
 - It returns a list of employees and the name of their team.

natural_join.sql

```
1 SELECT CONCAT (first_name, ' ', last_name) Employee, name "Team Name"
2 FROM Employee NATURAL JOIN Team;
```

```
mysql> SOURCE natural_join.sql;
+-----+-----+
| Employee      | Team Name |
+-----+-----+
| Morgan Wilson | Education |
| Emily Wilson  | Management |
| Kim Perkins   | Systems   |
| Alex Hardin   | Education |
| Tina Johnson  | Sales     |
| Rachel Fitzgerald | Education |
| Maria Higgins | Sales     |
| Brian Perkins | Education |
| Connie McKinney | Management |
| Travis Train  | Systems   |
+-----+-----+
10 rows in set (0.00 sec)
```

Suppose a natural join on the *Class* and *Course* tables is desired, based on the shared *course_id* field name.

natural_problem.sql

```
1 SELECT name, start_date FROM Class NATURAL JOIN Course;
```

```
mysql> SOURCE natural_problem.sql;
```

```
+-----+-----+
| name          | start_date |
+-----+-----+
| XHTML        | 2005-02-15 |
| Java Programming | 2005-02-14 |
| Java Programming | 2005-02-14 |
| Perl Programming | 2005-02-21 |
| Perl Programming | 2005-02-28 |
+-----+-----+
5 rows in set (0.00 sec)
```

The problem with the above results is that not all the courses and their start dates are listed.

The reason for this result is that the tables have more than one field in common.

- The both have *unit_price* and *course_id* fields.
- Therefore, a natural join will only return the rows where both the *course_id* and the *unit_price* are equal.
 - This is where the **USING** clause can be used to fix the issue. <<<

The USING Clause

To specify that you want the **NATURAL JOIN** to occur only on one of the common fields, the **NATURAL JOIN** need to specify the common field upon which the equality operator will be used.

- A **NATURAL JOIN** is assumed when only the keyword **JOIN** is used.
- The field to join on is specified using a **USING** clause as follows.

natural_fix.sql

```
1 SELECT name, start_date
2 FROM Class JOIN Course
3 USING(course_id);
```

```
mysql> SOURCE natural_fix.sql;
```

name	start_date
XHTML	2005-02-15
Java Programming	2005-02-14
Java Programming	2005-02-14
JavaScript	2005-02-21
Perl Programming	2005-02-21
Oracle SQL	2005-02-21
Java Programming	2005-02-28
JSPs	2005-02-28
Perl Programming	2005-02-28
Oracle SQL	2005-03-07

10 rows in set (0.00 sec)

Outer Joins

With an inner join, all rows that do not match the criteria in the **ON** clauses are discarded.

In an outer join, some of these rows are kept.

There are three types of outer joins.

- **RIGHT OUTER JOIN**
 - A Right outer join guarantees that it includes all rows from the table specified on the right-hand side of the join
- **LEFT OUTER JOIN**
 - A Left outer join guarantees that it includes all rows from the table specified on the left-hand side of the join
- **FULL OUTER JOIN**
 - A Full outer join guarantees that it includes all rows from both tables.
 - ◆ Note that MySQL does not support full outer joins

Right Outer Join

- The syntax for the **RIGHT OUTER JOIN** is as follows:

right_outer.sql

```
1 SELECT name, location, start_date, end_date, num_days
2 FROM Class RIGHT OUTER JOIN Course
3 ON Class.course_id = Course.course_id;
```

```
mysql> SOURCE right_outer.sql;
```

name	location	start_date	end_date	num_days
XHTML	Las Vegas, NV	2005-02-15	2005-02-16	2
Java Programming	NYC	2005-02-14	2005-02-18	5
Java Programming	Manchester, CT	2005-02-14	2005-02-18	5
JavaScript	Las Vegas, NV	2005-02-21	2005-02-23	3
Perl Programming	Columbia, MD	2005-02-21	2005-02-25	5
Oracle SQL	Manchester, CT	2005-02-21	2005-02-23	3
Java Programming	Las Vegas, NV	2005-02-28	2005-03-04	5
JSPs	NYC	2005-02-28	2005-03-01	2
Perl Programming	Columbia, MD	2005-02-28	2005-03-04	5
Oracle SQL	Columbia, MD	2005-03-07	2005-03-09	3
Servlets	NULL	NULL	NULL	2
Fortran	NULL	NULL	NULL	NULL
PL1	NULL	NULL	NULL	NULL

13 rows in set (0.00 sec)

The courses with names 'Servlets', 'Fortran', and 'PL1' are displayed from the **Course** table even though there are no classes scheduled with them in the **Class** table.

- Because the **Course** table is listed to the right of the **RIGHT OUTER JOIN**, all entries in the **Course** table are required.
- All entries in the **Course** table are matched with a row of **NULL** values in the **Class** table if there is no match with an existing row in the **Class** table.

Left Outer Join

By changing the syntax to read **LEFT OUTER JOIN** as opposed to **RIGHT OUTER JOIN**, the rows in the table on the left *Class* become the required rows.

- Therefore, the result would only show all the classes that are listed in the *Class* table that have a corresponding *course_id* in the *Course* table.

left_outer.sql

```
1 SELECT name, location, start_date, end_date, num_days
2 FROM Class LEFT OUTER JOIN Course
3 ON Class.course_id = Course.course_id;
```

```
mysql> SOURCE left_outer.sql;
```

name	location	start_date	end_date	num_days
Java Programming	NYC	2005-02-14	2005-02-18	5
Java Programming	Manchester, CT	2005-02-14	2005-02-18	5
Java Programming	Las Vegas, NV	2005-02-28	2005-03-04	5
JSPs	NYC	2005-02-28	2005-03-01	2
XHTML	Las Vegas, NV	2005-02-15	2005-02-16	2
JavaScript	Las Vegas, NV	2005-02-21	2005-02-23	3
Perl Programming	Columbia, MD	2005-02-21	2005-02-25	5
Perl Programming	Columbia, MD	2005-02-28	2005-03-04	5
Oracle SQL	Manchester, CT	2005-02-21	2005-02-23	3
Oracle SQL	Columbia, MD	2005-03-07	2005-03-09	3

10 rows in set (0.00 sec)

Exercises

Exercise 1

List the members of the "Education" department FROM the *Team* table.

Exercise 2

List the classes ordered by the customer "Green Acres Farm."

Exercise 3

List the class id and name of each class, the customer ordering the class, and the name and phone number of the salesperson in charge of the customer's account. The listing should be in ascending order by class id.

Exercise 4

List the employees who share the same last name. (Hint: You must search the table for each person's last name and find matches that do not include the person for whom you are querying.)

Exercise 5

List the full name of all employees who have mentors and list the full name of their respective mentors. Provide appropriate headings for each column.

Exercise 6

Assuming that you have added the Recreation team to the *Team* table (as suggested in the Left Outer Join section in this chapter), list the names of the teams that have members.

- Remove any redundant rows from the list.

Exercise 7

List the names of the employees that do not belong to a team.

Exercise 8

Who is the only instructor that does not belong to the Education team?

More Exercises

The following exercises will use the MySQL Sample Database, not the training database that had been previously used.

Exercise 1

Write a query that will retrieve a list of employees first and last names and the city and state their office is located in.

Exercise 2

Create a query that will get each customers payments sorted by customer and then by the payment date.

Exercise 3

Create a query that shows only the customers that have not placed any orders.

Chapter 10. Set Operators

Objectives

- Introduce set concepts
- Specifying selection criteria
- Using Unions

Introduction

Earlier in the course, we introduced various SQL operators.

- In this section, we introduce a special group of operators known as the set operators.

The set operators realize mathematical set operations.

- There are two set operators supported by MySQL.
 - ▶ Union
 - ▶ Union All

When two queries are issued using one of the set operators, there must be same number of expressions in both SELECT statements of a Union

- The corresponding expressions should also have the same data type.

Selection Criteria

The **UNION** operator takes the results of one query and combines this result with the result of a second query in such a way that only one row of all duplicate rows is kept.

- Two rows are duplicates if the fields in the first row respectively match the fields in the second row.
- Here is an *incorrect* example of a **UNION**. We are using the *Employee* and the *Team* tables.
 - The column counts for the first **SELECT** are different from those of the second **SELECT** and will cause an error in the **UNION**.

incorrect_union.sql

```
1 SELECT first_name, team_id FROM Employee
2 UNION
3 SELECT name FROM Team;
```

```
mysql> SOURCE incorrect_union.sql;
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

Another restriction to the set operators is that the individual component queries cannot have an **ORDER BY** clause.

- If you do wish to sort the merged result set, use the **ORDER BY** clause at the end of the compound query.

As we go through the next few examples, notice that the column headings are derived from the first **SELECT** component.

Creating Sample Data

The next example creates two new tables that will be used to demonstrate the **UNION** and **UNION ALL** operators in subsequent examples.

golf_handicaps.sql

```
1 DROP TABLE IF EXISTS GolfCourse1;
2
3 CREATE TABLE GolfCourse1 (name VARCHAR(20), handicap INT);
4
5 INSERT INTO GolfCourse1
6 VALUES ('Smith', 10), ('Jones', 20), ('Miller', 15), ('Bradock', 10);
7
8
9 DROP TABLE IF EXISTS GolfCourse2;
10
11 CREATE TABLE GolfCourse2 (golfer VARCHAR(20), handicap INT,
12                             favorite_hole TINYINT );
13
14 INSERT INTO GolfCourse2
15 VALUES ('Adams', 5, 3), ('Smith', 15, 18), ('Brock', 10, 7), ('Jones', 20, 12);
16
17 SELECT * FROM GolfCourse1;
18
19 SELECT * FROM GolfCourse2;
```

The two tables generated by the above script are shown on the following page.

```
mysql> SOURCE golf_handicaps.sql;
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
Query OK, 4 rows affected (0.00 sec)
Records: 4 Duplicates: 0 Warnings: 0
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
Query OK, 4 rows affected (0.00 sec)
Records: 4 Duplicates: 0 Warnings: 0
```

```
+-----+-----+
| name   | handicap |
+-----+-----+
| Smith  | 10       |
| Jones  | 20       |
| Miller | 15       |
| Bradock| 10       |
+-----+-----+
4 rows in set (0.00 sec)
```

```
+-----+-----+-----+
| golfer | handicap | favorite_hole |
+-----+-----+-----+
| Adams  | 5        | 3             |
| Smith  | 15       | 18            |
| Brock  | 10       | 7             |
| Jones  | 20       | 12            |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

Union

The **UNION** operator is used to combine the result sets of multiple **SELECT** statements.

- It removes duplicate rows between the various **SELECT** statements.
- It selects all those rows that are in either the first result set, the second result set, or both result sets.
- The column names are determined by those of the first select.
 - Note that duplicate rows are eliminated.

simple_union.sql

```
1 SELECT name FROM GolfCourse1
2 UNION
3 SELECT golfer FROM GolfCourse2;
```

```
mysql> SOURCE simple_union.sql;
```

```
+-----+
| name  |
+-----+
| Smith |
| Jones |
| Miller|
| Bradock|
| Adams |
| Brock |
+-----+
6 rows in set (0.00 sec)
```

Multiple fields could have been selected rather than just the **name** field.

simple_union_revised.sql

```
1 SELECT name, handicap FROM GolfCourse1
2 UNION
3 SELECT golfer, handicap FROM GolfCourse2;
```

The output of the example above is shown on the following page.


```
mysql> SOURCE simple_union_revised.sql;
```

```
+-----+-----+
| name   | handicap |
+-----+-----+
| Smith  | 10       |
| Jones  | 20       |
| Miller | 15       |
| Bradock| 10       |
| Adams  | 5        |
| Smith  | 15       |
| Brock  | 10       |
+-----+-----+
7 rows in set (0.00 sec)
```

Notice now that there is an extra row in the above output because the two Smiths differ in the *handicap* column.

Sorting by the *handicap* field can be accomplished by adding the **ORDER BY** at the end of the query.

ordering_union.sql

```
1 SELECT name, handicap FROM GolfCourse1
2 UNION
3 SELECT golfer, handicap FROM GolfCourse2
4 ORDER BY handicap;
```

```
mysql> SOURCE ordering_union.sql;
```

```
+-----+-----+
| name   | handicap |
+-----+-----+
| Adams  | 5        |
| Bradock| 10       |
| Brock  | 10       |
| Smith  | 10       |
| Miller | 15       |
| Smith  | 15       |
| Jones  | 20       |
+-----+-----+
7 rows in set (0.00 sec)
```

- The **ORDER BY** above could have referenced the field by index of 2 instead of by *handicap*.
 - ▶ If the SELECT uses an ***** to include all fields the **ORDER BY** would be required to be specified by the

index of the desired field.

Union All

The **UNION ALL** operator does not remove duplicate rows from the result set like the **UNION** does.

union_all.sql

```
1 SELECT golfer, handicap FROM GolfCourse2
2 UNION ALL
3 SELECT name, handicap FROM GolfCourse1
4 ORDER BY golfer;
```

```
mysql> SOURCE union_all.sql;
```

```
+-----+-----+
| golfer | handicap |
+-----+-----+
| Adams  | 5        |
| Bradock| 10       |
| Brock  | 10       |
| Jones  | 20       |
| Jones  | 20       |
| Miller | 15       |
| Smith  | 15       |
| Smith  | 10       |
+-----+-----+
8 rows in set (0.00 sec)
```

Exercises

Exercise 1

Use a set operator to list all the classes by name that either begin in March or are taken by the client "Green Acres Farm."

Chapter 11. SQL Sub-queries

Objectives

- Sub-queries with DML statements
- Typical sub-queries
- Standard and correlated sub-queries
- Predicate operators

Introduction

A **sub-query** is a query that is contained within another SQL statement.

- Most RDBMS require that a sub-query must always appear within parentheses.

A sub-query can be used in several scenarios including the following.

- SELECT
- UPDATE
- DELETE
- INSERT

A sub-query is usually added within the **ON** or **WHERE** Clause of another **SELECT** statement.

- You can use the comparison operators, such as **>**, **<**, or **=**.
- The comparison operator can also be a multiple-row operator, such as **IN**, **ANY**, or **ALL**.
 - These operators are sometimes called sub-query operators.

A sub-query is also called an inner query or inner select, while the statement containing a sub-query is also called an outer query or outer select.

Some of the possible advantages of sub-queries are:

- They can be used to isolate each part of a complex statement.
- They can be an alternative way of querying that would otherwise require complex joins and unions.
- Sub-queries are often easier to read than complex joins or unions.

Using a Sub-query with a DML Statement

It is possible to insert data into a table using a sub-query.

- The idea here is to use the results from a **SELECT** and "plug" them into a table.
 - ▶ The example below uses a sub-query in the second **INSERT** to populate a newly created table *FullName* with data from another table *Employee*.

insert_subquery.sql

```
1 DROP TABLE IF EXISTS FullName;
2
3 CREATE TABLE FullName (emp_id VARCHAR(10), name VARCHAR(20));
4
5 INSERT INTO FullName VALUES(678, 'Kyle Jones');
6
7 INSERT INTO FullName
8     (SELECT employee_id, CONCAT(first_name, ' ', last_name) FROM Employee);
9
10 SELECT * FROM FullName;
```

The output from running the above script is shown on the following page.

```
mysql> SOURCE insert_subquery.sql;
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.03 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 10 rows affected (0.00 sec)
Records: 10  Duplicates: 0  Warnings: 0
```

```
+-----+-----+
| emp_id | name          |
+-----+-----+
| 678    | Kyle Jones    |
| 500    | Morgan Wilson |
| 501    | Emily Wilson  |
| 502    | Kim Perkins   |
| 503    | Alex Hardin   |
| 508    | Tina Johnson  |
| 504    | Rachel Fitzgerald |
| 505    | Maria Higgins |
| 506    | Brian Perkins |
| 507    | Connie McKinney |
| 509    | Travis Train  |
+-----+-----+
11 rows in set (0.00 sec)
```


You can also use a sub-query with a **DELETE** statement as shown here.

delete_subquery.sql

```
1 DELETE FROM FullName
2 WHERE emp_id IN
3     (SELECT employee_id FROM Employee WHERE last_name = 'Wilson');
4
5 SELECT * FROM FullName;
```

```
mysql> SOURCE delete_subquery.sql;
Query OK, 2 rows affected (0.01 sec)
```

emp_id	name
678	Kyle Jones
502	Kim Perkins
503	Alex Hardin
508	Tina Johnson
504	Rachel Fitzgerald
505	Maria Higgins
506	Brian Perkins
507	Connie McKinney
509	Travis Train

9 rows in set (0.00 sec)

Here is an example showing the use of a sub-query in an **UPDATE** statement.

update_subquery.sql

```
1 UPDATE FullName
2 SET emp_id = emp_id + 200
3 WHERE SUBSTR(name, LOCATE(' ', name) + 1) IN
4     (SELECT last_name FROM Employee);
5
6 SELECT * FROM FullName;
```

- The above **UPDATE** will only update the *emp_id* of those records from the *FullName* that have the same last name as those in the *Employee* table.
 - ▶ Kyle Jones does not exist in the *Employee* table and therefore will not have his *emp_id* updated.

```
mysql> SOURCE update_subquery.sql;
Query OK, 8 rows affected (0.01 sec)
Rows matched: 8  Changed: 8  Warnings: 0
```

```
+-----+-----+
| emp_id | name          |
+-----+-----+
| 678    | Kyle Jones    |
| 702    | Kim Perkins   |
| 703    | Alex Hardin   |
| 708    | Tina Johnson  |
| 704    | Rachel Fitzgerald |
| 705    | Maria Higgins |
| 706    | Brian Perkins |
| 707    | Connie McKinney |
| 709    | Travis Train  |
+-----+-----+
9 rows in set (0.00 sec)
```

Typical Sub-queries

Suppose the names of all those employees whose salary is greater than the average salary is desired.

- The following attempt would generate an error due to an invalid use of the group function named **AVG**.

bad_average.sql

```
1 SELECT first_name, last_name, salary
2 FROM Employee
3 WHERE salary > AVG(salary);
```

```
mysql> SOURCE bad_average.sql;
ERROR 1111 (HY000): Invalid use of group function
```

A sub-query can be used to solve the above problem.

good_average.sql

```
1 SELECT first_name, last_name, salary
2 FROM Employee
3 WHERE salary > (SELECT AVG(salary) FROM Employee);
```

```
mysql> SOURCE good_average.sql;
+-----+-----+-----+
| first_name | last_name | salary |
+-----+-----+-----+
| Morgan    | Wilson   | 50000  |
| Emily     | Wilson   | 50000  |
| Kim       | Perkins  | 40000  |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Now, suppose you wish to count all the people who make more than the average salary.

- The previous query yields the employees who make more than the average salary.
 - All that is needed now is to count them. <<<

A first attempt at using the **COUNT** function to solve the problem is shown below.

- It uses what is called a derived table.
 - A derived table is a subquery that is defined as part of the **FROM** clause as shown below.

bad_count.sql

```
1 SELECT COUNT(*) FROM
2   (SELECT salary FROM Employee
3    WHERE salary > (SELECT AVG(salary) FROM Employee));
```

```
mysql> SOURCE bad_count.sql;
ERROR 1248 (42000): Every derived table must have its own alias
```

The derived table will need to be defined using a table alias.

good_count.sql

```
1 SELECT COUNT(*) AS '# Above Average' FROM
2   (SELECT salary
3    FROM Employee
4    WHERE salary > (SELECT AVG(salary) FROM Employee)) as Salary;
```

```
mysql> SOURCE good_count.sql;
```

```
+-----+
| # Above Average |
+-----+
|                3 |
+-----+
1 row in set (0.00 sec)
```

While the table alias of *Salary* was not used in the outer **SELECT**, it is still required to be defined for the derived table the **COUNT** function is operating on.

A common use for the **IN**, **ANY**, and **ALL** operators is in a sub-query.

- The following query displays information about the employees and their mentors from the *Employee* table

mentor_info.sql

```
1 SELECT employee_id, first_name, last_name, mentor_id FROM Employee;
```

```
mysql> SOURCE mentor_info.sql;
```

employee_id	first_name	last_name	mentor_id
500	Morgan	Wilson	NULL
501	Emily	Wilson	NULL
502	Kim	Perkins	NULL
503	Alex	Hardin	NULL
508	Tina	Johnson	NULL
504	Rachel	Fitzgerald	500
505	Maria	Higgins	508
506	Brian	Perkins	503
507	Connie	McKinney	501
509	Travis	Train	502

10 rows in set (0.00 sec)

- In the table above, there are five mentors - those that have a *mentor_id* of 500, 501, 502, 503, and 508.

Suppose we wish to list the first and last names of these mentors.

- Those employees whose *employee_id* appears in the *mentor_id* column).
 - The query is shown below.

mentor_details.sql

```
1 SELECT employee_id, first_name, last_name
2 FROM Employee
3 WHERE employee_id IN (SELECT DISTINCT mentor_id FROM Employee);
```

The results of the previous query are shown below.

```
mysql> SOURCE mentor_details.sql;
+-----+-----+-----+
| employee_id | first_name | last_name |
+-----+-----+-----+
|          500 | Morgan    | Wilson    |
|          501 | Emily     | Wilson    |
|          502 | Kim       | Perkins   |
|          503 | Alex      | Hardin    |
|          508 | Tina      | Johnson   |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

The sub-query from the previous code is:

```
SELECT DISTINCT mentor_id FROM Employee
```

- The values of the *mentor_id* are 500, 501, 502, 503, and 508.

Therefore, it could have been written as:

```
SELECT employee_id, first_name, last_name
FROM Employee WHERE employee_id IN(500,501,502,503,508);
```

Sub-query Operators

ANY and **ALL** are commonly used with sub-queries.

- The following shows the starting and ending date of all 5-day courses.

five_day_courses.sql

```
1 SELECT start_date, end_date
2 FROM Class
3 WHERE course_id = ANY (SELECT course_id FROM Course WHERE num_days = 5);
```

```
mysql> SOURCE five_day_courses.sql;
```

```
+-----+-----+
| start_date | end_date |
+-----+-----+
| 2005-02-14 | 2005-02-18 |
| 2005-02-14 | 2005-02-18 |
| 2005-02-21 | 2005-02-25 |
| 2005-02-28 | 2005-03-04 |
| 2005-02-28 | 2005-03-04 |
+-----+-----+
5 rows in set (0.01 sec)
```

The next example demonstrates building a query that displays information about all courses for a given city.

courses_for_city.sql

```
1 SELECT *
2 FROM Course
3 WHERE course_id = ANY (SELECT course_id FROM Class WHERE location LIKE '%Col%')
```

```
mysql> SOURCE courses_for_city.sql;
```

```
+-----+-----+-----+-----+-----+
| course_id | name           | unit_price | num_days | courseware_id |
+-----+-----+-----+-----+-----+
|      220 | Perl Programming |    400.00 |      5 | TE220          |
|      300 | Oracle SQL      |    500.00 |      3 | TE300          |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Standard vs. Correlated Sub-queries

All the sub-queries shown so far have been **standard sub-queries**.

- Standard sub-queries deal only with sub-queries where the inner query is completed in its entirety before the outer query is executed.

Correlated sub-queries are sub-queries where the inner query is processed every time a row is processed in the outer query.

- Correlated sub-queries make references from the inner query to one or more tables in the outer query.
- The outer query must run first for the inner query to acquire the data it needs to determine whether the row should be added to the resultant table.

The next example demonstrates how to find the employees whose salaries were less than the average salary for their teams.

- This type of query is a prime candidate for a correlated sub-query.

Correlated Sub-query Example

The first query lists each team and their average salary.

- The **GROUP BY** clause, which groups all the entries by team, will be further discussed in the next chapter.

The second query then shows those employees whose salaries are lower than that of their team's average.

- Every time the outer query is evaluated for an employee, the average salary for that employee's team must be calculated.

correlated_subquery.sql

```
1 SELECT Team.team_id, AVG(salary)
2 FROM Employee INNER JOIN Team
3 ON Employee.team_id = Team.team_id
4 GROUP BY Team.team_id;
5
6
7 SELECT team_id, last_name, salary
8 FROM Employee AS E
9 WHERE salary <
10     (SELECT AVG(salary)
11      FROM Employee INNER JOIN Team
12      ON Employee.team_id = Team.team_id
13      WHERE Employee.team_id = E.team_id);
```

```
mysql> SOURCE correlated_subquery.sql;
```

```
+-----+-----+
| team_id | AVG(salary) |
+-----+-----+
|      400 | 38750.0000 |
|      401 | 42500.0000 |
|      402 | 32500.0000 |
|      403 | 35000.0000 |
+-----+-----+
4 rows in set (0.00 sec)
```

```
+-----+-----+-----+
| team_id | last_name  | salary |
+-----+-----+-----+
|      400 | Hardin     | 35000 |
|      400 | Fitzgerald | 35000 |
|      400 | Perkins    | 35000 |
|      401 | McKinney   | 35000 |
|      402 | Train      | 25000 |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

Predicate Operators

A predicate is an expression that evaluates to TRUE, FALSE, or UNKNOWN.

- Predicates are used in the search condition of **WHERE** clauses and **HAVING** clauses, the join conditions of **FROM** clauses, and other constructs where a Boolean value is required.
- The following operators are called predicate operators.
 - **BETWEEN EXISTS IN LIKE**

These operators are typically used with a correlated sub-query where the outer query returns a row to the resultant table if the inner query has at least one row of output.

- The result of the inner query is simply a Boolean and as such has no available data.

predicates.sql

```
1 SELECT name
2 FROM Team
3 WHERE EXISTS
4     (SELECT team_id FROM Employee WHERE Employee.team_id = Team.team_id);
```

```
mysql> SOURCE predicates.sql;
```

```
+-----+
| name   |
+-----+
| Education |
| Management |
| Systems  |
| Sales    |
+-----+
4 rows in set (0.00 sec)
```

Exercises

Exercise 1

Display the name and the team of the highest paid individual(s) in the employees table.

Exercise 2

What is the name of the client(s) who has the first scheduled class?

Exercise 3

What is the name and team of the most recently hired individual who is currently employed?

Exercise 4

Add a new team to the teams table. Write a query that will display only the teams to which there is no one assigned.

Exercise 5

List the name, salary, and team ID of the employees who earn more than the average salary of all the employees.

Exercise 6

List the courses that have not been ordered by any customers.

Exercise 7

Michael Saltzman has had an unexpected emergency. (He won free tickets to a golf tournament in Scotland.)

- He managed to talk Alan Baumgarten into covering his class the week of February 21st and Patti Ordonez to cover his class the week of February 28th.
 - Update the classes table to reflect these changes.

Exercise 8

List the names of instructors who are currently not scheduled to teach any classes.

Exercise 9

List the name, salary, and team name of the employees who earn more than the average salary for their team.

Chapter 12. Groups

Objectives

- Grouping query results

SQL Statements

Until now, the SQL statements that we have examined have displayed detailed information about the data in a table or provided summary information for the entire table.

- The following is an example of a query that returns detailed information.
 - It also introduces the LIMIT statement to limit the number of records returned

basic_details.sql

```
1 SELECT first_name, last_name, salary, name
2 FROM Team, Employee
3 WHERE Team.team_id = Employee.team_id
4 LIMIT 3;
```

```
mysql> SOURCE basic_details.sql;
```

```
+-----+-----+-----+-----+
| first_name | last_name | salary | name      |
+-----+-----+-----+-----+
| Morgan     | Wilson    | 50000  | Education |
| Emily      | Wilson    | 50000  | Management|
| Kim        | Perkins   | 40000  | Systems   |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Summaries or statistics can be provided for a column in a table using group functions such as **SUM**, **COUNT**, **AVG**, **MIN**, **MAX**, **STDDEV** and **VARIANCE**.

summaries.sql

```
1 SELECT COUNT(*) AS Count, SUM(salary) AS Sum, AVG(salary) AS Avg
2 FROM Employee;
```

```
mysql> SOURCE summaries.sql;
```

```
+-----+-----+-----+
| Count | Sum    | Avg      |
+-----+-----+-----+
| 10    | 375000 | 37500.0000 |
+-----+-----+-----+
1 row in set (0.00 sec)
```


GROUP BY Clause

The queries displayed so far have had either group functions or field names in a query, but not both.

- Suppose we are interested in listing the first name, last name, and salary of each employee by team
- We may also want to gather some statistics, such as the average salary for each team.

A table is separated into groups using the **GROUP BY** clause.

- It is placed in a query after the **FROM** clause and the **WHERE** clause if it exists.
 - Note that the following query does *not* produce the desired results.

bad_grouping.sql

```
1 SELECT last_name, team_id
2 FROM Employee
3 GROUP BY team_id
```

- The ANSI standard for SQL states everything in the **SELECT** clause must be included in the **GROUP BY** expression or be a group function.
 - If MySQL is configured to adhere to the ANSI standard with `sql_mode = 'ONLY_FULL_GROUP_BY'`, the following error will occur.

```
mysql> SOURCE bad_grouping.sql;
ERROR 1055 (42000): Expression #1 of SELECT list is not in GROUP BY clause and contains
nonaggregated column 'student.Employee.last_name' which is not functionally dependent on
columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by
```

- Even if MySQL is not configured to be ANSI compliant the result will only show one employee per group as shown below.

```
mysql> SOURCE bad_grouping.sql;
```

```
+-----+-----+
| last_name | team_id |
+-----+-----+
| Wilson    | 400     |
| Wilson    | 401     |
| Perkins   | 402     |
| Johnson   | 403     |
+-----+-----+
4 rows in set (0.00 sec)
```

More information about MySQL's `sql_mode` can be found at the following URL:

- <https://dev.mysql.com/doc/refman/8.0/en/sql-mode.html>

The next example attempts to correct the grouping issue by including both the `last_name` and `team_id` fields in the `GROUP BY` since they are both listed in the `SELECT`.

ansi_grouping.sql

```
1 SELECT last_name, team_id
2 FROM Employee
3 GROUP BY team_id, last_name
```

```
mysql> SOURCE ansi_grouping.sql;
```

```
+-----+-----+
| last_name | team_id |
+-----+-----+
| Wilson    | 400     |
| Wilson    | 401     |
| Perkins   | 402     |
| Hardin    | 400     |
| Johnson   | 403     |
| Fitzgerald | 400     |
| Higgins   | 403     |
| Perkins   | 400     |
| McKinney  | 401     |
| Train     | 402     |
+-----+-----+
10 rows in set (0.00 sec)
```

While MySQL used to perform an implicit `ORDER BY` when a `GROUP BY` was used, MySQL 8 no longer supports this.

- This means that the above query is as if it were as follows:

```
SELECT DISTINCT last_name, team_id FROM Employee;
```

For a **GROUP BY** to be meaningful it needs to have an aggregate function (also called a group function) as at least one of the fields in the **SELECT**.

The following query lists the number of members in each team and the average salary of all the members of a team.

group_with_aggregates.sql

```
1 SELECT team_id, count(*), avg(salary) FROM Employee
2 GROUP BY team_id;
```

```
mysql> SOURCE group_with_aggregates.sql;
```

```
+-----+-----+-----+
| team_id | count(*) | avg(salary) |
+-----+-----+-----+
|      400 |         4 | 38750.0000 |
|      401 |         2 | 42500.0000 |
|      402 |         2 | 32500.0000 |
|      403 |         2 | 35000.0000 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

Adding an **ORDER BY** on the AVG(salary) can be done by aliasing that column as follows.

group_with_order.sql

```
1 SELECT team_id, count(*), avg(salary) AS 'Average Salary' FROM Employee
2 GROUP BY team_id
3 ORDER BY 'Average Salary';
```

```
mysql> SOURCE group_with_order.sql;
```

```
+-----+-----+-----+
| team_id | count(*) | Average Salary |
+-----+-----+-----+
|      402 |         2 | 32500.0000 |
|      403 |         2 | 35000.0000 |
|      400 |         4 | 38750.0000 |
|      401 |         2 | 42500.0000 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

HAVING Clause

The **WHERE** clause can be used to limit the result set returned by the **SELECT**.

- Similarly, the **HAVING** clause follows the **GROUP BY** clause and further restricts the resultant set that it bases the grouping on.
 - You can think of the **HAVING** clause as limiting the grouped resultant set whereas the **WHERE** clause limits the resultant set of the initial query.

Consider a query that finds the teams that have an average salary greater than \$35,000.

having.sql

```
1 SELECT name, COUNT(*), AVG(salary)
2 FROM Team INNER JOIN Employee
3 ON Team.team_id = Employee.team_id
4 GROUP BY Team.team_id, name
5 HAVING AVG(salary) > 35000;
```

```
mysql> SOURCE having.sql;
```

```
+-----+-----+-----+
| name      | COUNT(*) | AVG(salary) |
+-----+-----+-----+
| Education |         4 | 38750.0000 |
| Management |         2 | 42500.0000 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

The inner join is performed before grouping, so individuals not belonging to a group are ignored.

If any of the individuals did not have a salary, then the average salary figure would be skewed.

- This can be seen in the example below that adds two employees with a NULL for the salary.

having_distorted.sql

```

1 INSERT INTO Employee (employee_id, first_name, last_name, team_id, salary)
2 VALUES (9000, 'Emma', 'Rogers', 400, NULL), (9001, 'Joe', 'Baxter', 401, NULL);
3
4 SELECT name, COUNT(*), AVG(salary)
5 FROM Team INNER JOIN Employee
6 ON Team.team_id = Employee.team_id
7 GROUP BY Team.team_id, name
8 HAVING AVG(salary) > 35000;
9
10 DELETE FROM Employee WHERE employee_id >= 9000;
```

```

mysql> SOURCE having_distorted.sql;
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```

+-----+-----+-----+
| name      | COUNT(*) | AVG(salary) |
+-----+-----+-----+
| Education |         5 | 38750.0000 |
| Management |         3 | 42500.0000 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
Query OK, 2 rows affected (0.00 sec)
```

In the above output it is apparent that the number of employees in a department increased but the average did not.

- The ANSI standard requires that aggregate function ignore NULL in the field they are aggregating on.
 - So the additional employee with NULL for the salary is not figured into the average for the salary, but they do get counted.

Changing the **ON** clause to exclude salaries of NULL would improve the resultant count in the query.

- This can be seen in the example on the following page.

having_more.sql

```

1 INSERT INTO Employee (employee_id, first_name, last_name, team_id, salary)
2 VALUES (9000, 'Emma', "Rogers", 400, NULL), (9001, 'Joe', "Baxter", 401, NULL);
3
4 SELECT name, COUNT(*), AVG(salary)
5 FROM Team INNER JOIN Employee
6 ON Team.team_id = Employee.team_id AND Employee.salary IS NOT NULL
7 GROUP BY Team.team_id, name
8 HAVING AVG(salary) > 35000;
9
10 DELETE FROM Employee WHERE employee_id >= 9000;

```

mysql> SOURCE having_more.sql;

```

+-----+-----+-----+
| name      | COUNT(*) | AVG(salary) |
+-----+-----+-----+
| Education |         4 | 38750.0000 |
| Management |         2 | 42500.0000 |
+-----+-----+-----+
2 rows in set (0.00 sec)

```

A different approach to improving the results would be to `COUNT(salary)` instead of `COUNT(*)`.

having_count_salary.sql

```

1 INSERT INTO Employee (employee_id, first_name, last_name, team_id, salary)
2 VALUES (9000, 'Emma', "Rogers", 400, NULL), (9001, 'Joe', "Baxter", 401, NULL);
3
4 SELECT name, COUNT(salary), AVG(salary)
5 FROM Team INNER JOIN Employee
6 ON Team.team_id = Employee.team_id
7 GROUP BY Team.team_id, name
8 HAVING AVG(salary) > 35000;
9
10 DELETE FROM Employee WHERE employee_id >= 9000;

```

```
mysql> SOURCE having_count_salary.sql;
```

```
+-----+-----+-----+
| name      | COUNT(*) | AVG(salary) |
+-----+-----+-----+
| Education |         4 | 38750.0000 |
| Management |         2 | 42500.0000 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```


Order of Clauses in a **SELECT** Statement

- Considering all the clauses we have examined for a **SELECT** statement, here is an example that demonstrates the ordering of the clauses in a **SELECT** statement.

select_ordering.sql

```
1 SELECT name, COUNT(salary), AVG(salary)
2 FROM Team INNER JOIN Employee
3 ON Team.team_id = Employee.team_id
4 GROUP BY Team.team_id, name
5 HAVING AVG(salary) > 35000
6 ORDER BY AVG(salary) DESC;
```

```
mysql> SOURCE select_ordering.sql;
+-----+-----+-----+
| name      | COUNT(salary) | AVG(salary) |
+-----+-----+-----+
| Management |          2 | 42500.0000 |
| Education  |          4 | 38750.0000 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

When using a **GROUP BY** clause in conjunction with the **ORDER BY** clause, the **ORDER BY** criteria must contain an aggregate(group) function.

- The **AVG** is the group function used within the **ORDER BY** in the above example.

Exercises

Exercise 1

List the average salaries of the Employee by gender. Be sure to include only the salaries of the currently employed.

Exercise 2

List the average salaries of the Employee by marital status. Again, only include the salaries of the currently employed.

Exercise 3

List all mentors and the number of Employee they are mentoring. Show the mentors with the largest number of Employee first.

Exercise 4

Update the *Employee* table to reflect that Alan Baumgarten will assume Michael Saltzman's mentoring responsibilities. Then, repeat the query FROM the previous exercise.

Exercise 5

List the classes by customer and course name. Include in the list the customer name, the name of the class, and the total cost for the class.

More Exercises

The following exercises will use the MySQL Sample Database, not the training database that had been previously used.

Exercise 1

Create a query that shows the total of the number of products ordered and the total price of all the orders per customer.

Exercise 2

Create a query to calculate the total and the count of all payments made per customer.

Exercise 3

Create a query that calculates the total number of products sold and total amount for each product line.

Exercise 4

Create a query that will calculate the sum of all payments for each customer.

Exercise 5

Create a query that will count all the orders each customer has made for all product lines.

- The results of this query should include all product lines, even if the customer hasn't ordered any products that belong to that product line.

Chapter 13. Stored Procedures

Objectives

- Create stored procedures
- Execute stored procedures
- Delete stored procedures

Creating Stored Procedures

A stored procedure is a prepared SQL code that is saved and re-usable.

- One benefit of using stored procedures is the ability to execute the same set of SQL statements without having to re-write the query.
 - Other benefits include less network traffic and more secure query execution.

The generic syntax for creating stored procedures is

```
DELIMITER //  
CREATE PROCEDURE proc_name  
BEGIN  
    sql_statements  
END //  
DELIMITER ;
```

The **DELIMITER** statements are not part of the stored procedure.

- The first **DELIMITER** statement changes the default delimiter from **;** to ****
- The last **DELIMITER** statement changes the default delimiter back to **;**
 - They are required so that the SQL commands in the stored procedure can include the **;** character.

The following code will create a stored procedure to retrieve a count of team members who have a salary greater than the average salary.

get_costly_teams.sql

```
1 DELIMITER //  
2 CREATE PROCEDURE GetCostlyTeams()  
3 BEGIN  
4 SELECT t.name, COUNT(*), AVG(e.salary)  
5 FROM Team t, Employee e  
6 WHERE t.team_id = e.team_id  
7 GROUP BY t.team_id, t.name  
8 HAVING AVG(e.salary)>35000;  
9 END //  
10 DELIMITER ;
```

```
mysql> SOURCE get_costly_teams.sql;  
Query OK, 0 rows affected (0.01 sec)
```

Executing Stored Procedures

Once created, stored procedures can be executed using the CALL command.

call_get_costly_teams.sql

```
1 CALL GetCostlyTeams();
```

```
mysql> SOURCE call_get_costly_teams.sql;
```

```
+-----+-----+-----+
| name      | COUNT(*) | AVG(e.salary) |
+-----+-----+-----+
| Education |         4 | 38750.0000    |
| Management |         2 | 42500.0000    |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

```
Query OK, 0 rows affected (0.03 sec)
```


Stored Procedures with Parameters

Stored procedures can include one or more parameters.

- When parameters are defined, data must be sent to the procedure when it is called.
- Multiple parameters are separated by commas.
- Parameters must also define their data type.

get_courses_at.sql

```
1 DELIMITER //
2
3 CREATE PROCEDURE GetCoursesAt(place varchar(15))
4 BEGIN
5     SELECT Course.name, Class.start_date, Class.end_date, Class.location
6     FROM Course, Class
7     WHERE Course.course_id = Class.course_id
8     AND Class.location = place;
9 END //
10
11 DELIMITER ;
```

```
mysql> SOURCE get_courses_at.sql;
Query OK, 0 rows affected (0.01 sec)
```

Values for the parameters are passed to the procedure when it is called.

call_get_courses_at.sql

```
1 CALL GetCoursesAt('NYC');
```

```
mysql> SOURCE call_get_courses_at.sql;
```

```
+-----+-----+-----+
| name      | COUNT(*) | AVG(e.salary) |
+-----+-----+-----+
| Education |         4 | 38750.0000    |
| Management |         2 | 42500.0000    |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
Query OK, 0 rows affected (0.01 sec)
```

Dropping Stored Procedures

The **DROP** command is used to remove stored procedures from the database.

```
DROP PROCEDURE proc_name;
```

drop_both_procedures.sql

```
1 DROP PROCEDURE GetCostlyTeams;  
2 DROP PROCEDURE GetCoursesAt;
```

```
mysql> SOURCE drop_both_procedures.sql;  
Query OK, 0 rows affected (0.01 sec)  
  
Query OK, 0 rows affected (0.01 sec)
```

Exercises

Exercise 1

Create a stored procedure that retrieves an instructor and all the courses they are teaching.

- The procedure should have a parameter for the instructors last name.
- Call the stored procedure a couple of times with different last names.

Exercise 2

Remove the stored procedure created in the previous exercise.

Exercise 3

Recreate the same stored procedure created in the first exercise to be the same as before but with a second parameter that is for the location of the course.

- Call the stored procedure a couple of times with different last names and locations.

Chapter 14. More Database Objects (DML)

Objectives

- Using relational views
- updating relational views
- Creating indexes

More Database Objects

Until now, this course has dealt exclusively with tables.

- Other database objects include views and indexes.

Views are ways to access the data in a table, but they do not store data.

- A view provides a lot of the functionality of a table, such as allowing a user to query and update data while restricting the user's accessibility to the data.

Indexes are supportive objects used to improve the performance of table searches.

- Query performance for random access is improved with indexing.

Relational Views

Relational views store SQL queries or statements.

- Typically, these statements are complex, and the views are used to hide the query's complexity.
- Relational views can also be used for security in cases where some of the information in a database may be confidential and only available to a limited number of users.

A view is created with the **CREATE VIEW** statement as follows.

simple_view.sql

```
1 CREATE VIEW Coworker AS
2 SELECT last_name, first_name, name
3 FROM Employee INNER JOIN Team ON Employee.team_id = Team.team_id;
4
5 # Display the contents of the view
6 SELECT * FROM Coworker
```

```
mysql> SOURCE simple_view.sql;
Query OK, 0 rows affected (0.02 sec)
```

last_name	first_name	name
Wilson	Morgan	Education
Wilson	Emily	Management
Perkins	Kim	Systems
Hardin	Alex	Education
Johnson	Tina	Sales
Fitzgerald	Rachel	Education
Higgins	Maria	Sales
Perkins	Brian	Education
McKinney	Connie	Management
Train	Travis	Systems

10 rows in set (0.00 sec)

Updating a View

A view can also be used for updates.

update_view.sql

```
1 # Updating the Coworker View also updates the Employee table
2
3 UPDATE Coworker SET first_name = 'Declan' WHERE last_name = 'Hardin';
4
5 SELECT * FROM Coworker LIMIT 5;
6
7 SELECT first_name, last_name FROM Employee LIMIT 5;
```

```
mysql> SOURCE update_view.sql;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

last_name	first_name	name
Wilson	Morgan	Education
Wilson	Emily	Management
Perkins	Kim	Systems
Hardin	Declan	Education
Johnson	Tina	Sales

5 rows in set (0.00 sec)

first_name	last_name
Morgan	Wilson
Emily	Wilson
Kim	Perkins
Declan	Hardin
Tina	Johnson

5 rows in set (0.00 sec)

Restrictions can also be added to a view so that when an update occurs via the view.

- It will adhere to the specifications of the select statement used to create the view by using the **WITH CHECK OPTION** clause. :filename: restricted_view.sql .update_view.sql

Updating the Coworker **View** also updates the Employee **table**

```
UPDATE Coworker SET first_name = 'Declan' WHERE last_name = 'Hardin';
```

```
SELECT * FROM Coworker LIMIT 5;
```

```
SELECT first_name, last_name FROM Employee LIMIT 5;
```

```
mysql> SOURCE restricted_view.sql;
Query OK, 0 rows affected (0.01 sec)
```

```
+-----+-----+-----+
| first_name | last_name | salary |
+-----+-----+-----+
| Morgan     | Wilson    | 50000  |
| Emily      | Wilson    | 50000  |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

restricted_access.sql

```
1 INSERT INTO BigSpender VALUES('John', 'Doe', 35000); # Fails
2
3 INSERT INTO BigSpender VALUES('Jane', 'Doe', 45000); # Succeeds
4
5 SELECT * FROM BigSpender;
6
7 SELECT employee_id, first_name, last_name, salary
8 FROM Employee ORDER BY salary DESC LIMIT 5;
```

```
mysql> SOURCE restricted_access.sql;
ERROR 1369 (HY000): CHECK OPTION failed 'student.bigspender'
Query OK, 1 row affected (0.01 sec)
```

```
+-----+-----+-----+
| first_name | last_name | salary |
+-----+-----+-----+
| Morgan     | Wilson    | 50000  |
| Emily      | Wilson    | 50000  |
| Jane       | Doe       | 45000  |
+-----+-----+-----+
```

3 rows in set (0.00 sec)

```
+-----+-----+-----+-----+
| employee_id | first_name | last_name | salary |
+-----+-----+-----+-----+
|          500 | Morgan     | Wilson    | 50000  |
|          501 | Emily      | Wilson    | 50000  |
|          NULL | Jane       | Doe       | 45000  |
|          502 | Kim        | Perkins   | 40000  |
|          503 | Declan     | Hardin    | 35000  |
+-----+-----+-----+-----+
```

5 rows in set (0.00 sec)

Indexes

Indexes are objects that improve the efficiency of accessing the data in tables.

- The most common type of index is a B-tree index.
 - A B-tree is a generalization of a binary search tree WHERE more than two paths can diverge from a single node.
 - The database automatically generates a B-tree index to enforce primary key and unique constraints.
- Other candidates for indexing are foreign key fields or any field that is frequently referenced in join operations or in conditional expressions of **WHERE** clauses in a query.
- In general, indexes improve query performance for random access but can degrade update performance for delete, insert, and update, because the indexes must be updated as well as the row.
- Therefore, it may be favorable to use an index for periods of heavy querying and drop the index in periods of heavy updating.

- To create a single column index, use the following syntax.

```
CREATE INDEX index_name  
ON table_name(column_name);
```

- To create a multi-column index, use the following syntax.

```
CREATE INDEX index_name  
ON table_name(column1_name, column2_name, ...);
```

- An index may be created to enforce unique values and would be created as follows.

```
CREATE UNIQUE INDEX index_name  
ON table_name(column_name);
```

- An index may be dropped without affecting the underlying base table data as follows.

```
DROP INDEX index_name ON table_name;
```

Exercises

Exercise 1

Create a view for the *employee* table named *coworkers* that contains all the information from the *employee* table except for the following columns *salary*, *bonus*, and *fire_date*. Include only the current Employee in your view.

Exercise 2

Show all records in the *Employee* table by using the *coworkers* view.

Exercise 3

Create an index named *crswr_index* for the *Courseware* table using the *courseware_id* column.

Exercise 4

Create a second view of the *Employee* table named *managers* that displays all the names and salaries of the Employee with a salary greater than \$50,000.

- Make it such that the user cannot insert an employee with a salary less than \$50,000 using this view.

Exercise 5

Drop the objects that you have created in the exercises for this chapter.

- *coworkers*, *crswr_indx*, and *managers*.

UMBC

Training Centers

6996 Columbia Gateway Drive

Suite 100

Columbia, MD 21046

Tel: 443-692-6600

<http://www.umbctraining.com>

SQL DEVELOPMENT IN PYTHON PART II

Course # TCPRG0003

Rev. 05/06/2021

This Page Intentionally Left Blank

Course Objectives

- At the conclusion of this course, students will be able to:
 - ▶ Write Python Programs that use the Python Database API to communicate with a database.
 - ▶ Write HTML documents that can use a form to collect data from a user.
 - ▶ Incorporate Cascading Style Sheets (CSS) into an HTML document.
 - ▶ Use the CGI module Cookies modules in Python to send and receive HTML requests.
 - ▶ Incorporate JavaScript into an HTML document to make it more dynamic.

This Page Intentionally Left Blank

Table of Contents

Chapter 15: Python and Databases.....	7
Introduction.....	8
MySQL.....	9
Connecting to the Database.....	10
Creating a Table.....	11
Inserting Records.....	12
Selecting Records.....	14
Additional SQL Queries.....	17
Updates.....	19
Deletes.....	20
Table Descriptions.....	21
Writing Database Scripts.....	24
Chapter 16: HTML.....	29
A Brief History of the World Wide Web.....	30
HTML Basics.....	31
HTML Tags.....	32
The HTML Root Element.....	33
HTML Sections.....	34
Grouping Content.....	36
Text Level Elements.....	39
Links and Anchors.....	41
Images.....	43
Tables.....	44
HTML Forms.....	45
Submitting HTML Forms.....	49
Character References.....	51
Chapter 16: Cascading Style Sheets.....	53
Introduction.....	54
CSS Syntax.....	56
CSS Property Names and Values.....	57
Creating CSS Style Sheets.....	58
Grouping Selectors.....	60
Chapter 18: Python and WebServers.....	61
Introduction.....	62
HTTPServer.....	63
CGI Scripts.....	65
HTML Forms and CGI Scripts.....	67
CGI Scripts and Cookies.....	71
Chapter 19: JavaScript.....	73
The Core JavaScript Language.....	74
A Simple JavaScript Example.....	75
Language Structure.....	76
Strings.....	79

Arithmetic Operators.....	81
Comparison and Logical Operators.....	82
The if Statement.....	83
The switch Statement.....	84
The for Statement.....	85
Arrays.....	86
Associative Arrays.....	88
Functions.....	90
Core JavaScript Functions.....	91
The Document Object Model (DOM).....	93
Trees and Nodes.....	94
The Node Interface.....	95
The NodeList and NamedNodeMap Interfaces.....	96
Node Traversal Example.....	97
The DOM Event Model.....	101

Chapter 15:

Python and Databases

Introduction

- Python's PEP 249 defines the Python Database API Specification.
 - ▶ The API is designed to encourage similarity between the Python modules that are used to access databases.
 - The API attempts to achieve consistency between different vendor's modules, permitting more portability across databases.
- Access to the database is made available through `Connection` objects.
 - ▶ The module must provide a method named `connect()` that returns a `Connection` object.
- `Connection` objects should have the following four methods.
 - ▶ `close()`
 - Used to close the connection.
 - Note that closing a connection without committing the changes first will cause an implicit rollback to be performed.
 - ▶ `commit()`
 - Commits any pending transaction to the database.
 - If the an auto-commit feature is supported, it must be initially off.
 - Database modules that do not support transactions should implement this method with void functionality.
 - ▶ `rollback()`
 - In case a database does provide transactions this method causes the database to roll back to the start of any pending transaction.
 - ▶ `cursor()`
 - Returns a new `Cursor` object using the connection.

MySQL

- Relational databases are widely used to store information as tables containing a number of rows.
 - ▶ While there are many choices of database engines, the lab environment here has been configured for the MySQL database.
 - The examples in this chapter will focus on the "MySQL Connector/Python" implementation of the Python Database API Specification.
 - ▶ A complete list of database interfaces available for Python can be found at the following URL.
 - <https://wiki.python.org/moin/DatabaseInterfaces>
- The various steps required to connect to a database and to create, read, update and delete tables and their contents will be discussed before putting them all together in a finished application.
- To begin the process, the first step will be to import the Python module necessary to connect to the database.
 - ▶ The `mysql.connector` module for the MySQL driver has already been installed on your system.
 - There might be slight differences in the calls in the example on the next page if a different driver or a database other than MySQL is used.
 - On the other hand, once a connection to the database is obtained, the remainder of the code that will follow in this chapter is similar to all database vendors that implement the Python Database API.

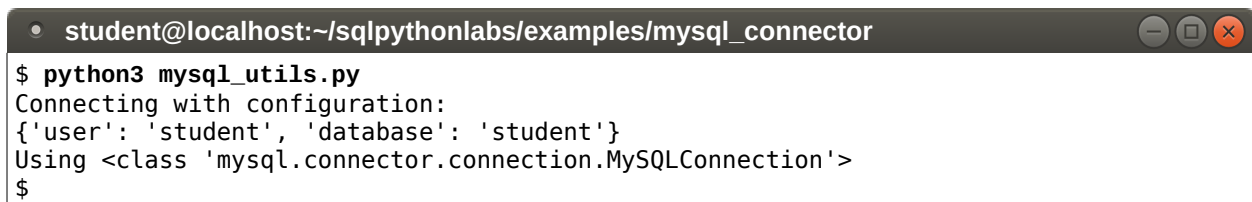
Connecting to the Database

- The following module will be used throughout the chapter to use the `mysql.connector` module to obtain a `Connection` object to connect to a MySQL database.

`mysql_utils.py`

```
1. import getpass
2. import mysql.connector
3.
4. def get_connection(verbose=False):
5.     login_name = getpass.getuser() # Obtain user login name
6.     config = {"user": login_name, "database": login_name}
7.     connection = mysql.connector.connect(**config)
8.     if verbose:
9.         print("Connecting with configuration:", config, sep="\n")
10.        print("Using", type(connection))
11.    return connection
12.
13. if __name__ == "__main__":
14.     connection = get_connection(True)
15.     connection.close()
```

- ▶ The `getpass` module from the Python Standard Library is used to call the `getpass.getuser()` function to obtain the login name of the current user in a portable manner from the underlying operating system.
 - The resulting `login_name` returned from the function is then used as the value to both the "user" and "database" keys used as key/value pair arguments to the `connect` function of the `mysql.connector` module.

A terminal window titled 'student@localhost:~/sqlpythonlabs/examples/mysql_connector' showing the execution of the script. The prompt is '\$', followed by the command 'python3 mysql_utils.py'. The output shows the connection configuration and the type of the connection object.

```
student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 mysql_utils.py
Connecting with configuration:
{'user': 'student', 'database': 'student'}
Using <class 'mysql.connector.connection.MySQLConnection'>
$
```

- ▶ The value of "*student*" in the output above will be replaced by the actual login name of the student.
- Once a connection to the database is obtained, the next step is to create a table within the database.

Creating a Table

- The example that follows connects to the database and creates a table.
 - ▶ Many of the operations (queries and commands) on the database and its tables will be made through a Cursor object.
 - ▶ It starts by creating a CREATE TABLE statement that defines the schema for a table named people.
 - The SQL statement is then passed as an argument to the execute method of the Cursor object.

create_a_table.py

```
1. from mysql_utils import get_connection
2.
3. def create_table():
4.     connection = get_connection()
5.     sql = ("CREATE TABLE people ("
6.           "    name VARCHAR(64), job VARCHAR(32), pay INTEGER"
7.           ")")
8.     cursor = connection.cursor()
9.     cursor.execute(sql)
10.    cursor.close()
11.    connection.close()
12.
13. def main():
14.     create_table()
15.
16. if __name__ == "__main__":
17.     main()
```

- ▶ Execution of the above statement will fail if the table already exists in the database.
 - If desired to only create a table if it does not exist already, "if not exists" can be added to the statement as shown below.
sql = "CREATE TABLE IF NOT EXISTS people (...)"

Inserting Records

- INSERT statements allow data to be inserted into an SQL table.

insert_a_record.py

```
1. from mysql_utils import get_connection
2.
3. def insert_record():
4.     connection = get_connection()
5.     sql = "INSERT INTO people VALUES (%s, %s, %s)"
6.     cursor = connection.cursor()
7.     cursor.execute(sql, ("Bob", "developer", 50000))
8.     connection.commit()
9.     cursor.close()
10.    connection.close()
11.
12. def main():
13.     insert_record()
14.
15. if __name__ == "__main__":
16.     main()
```

- ▶ The INSERT statement defined above uses placeholder parameters.
 - The placeholder parameters, or variables, are defined using %s.
 - Other database vendors and drivers may use a different syntax for placeholder parameters such as SQLite3 that uses "?" instead of "%s".
 - An added benefit of using the placeholder parameters is that it prevents certain SQL injection vulnerabilities by automatically escaping the input parameters.
- ▶ The execute() method takes an optional second argument of a tuple, or dictionary.
 - The contents of the tuple or dictionary are bound to the placeholder parameters of the first argument to the method.
- ▶ The commit() method is called on the Connection object to end the transaction and commit the changes to the database.
- ▶ close() is then called on both the Cursor and the Connection objects to release the resources before exiting.

Inserting Records

- There are two techniques for inserting multiple records.
 - ▶ The first technique is to simply include the call to `execute()` inside of a Python looping construct.
 - ▶ The other would be to call `executemany()` on the `Cursor` object as opposed to `execute()`.
 - The `executemany()` is optimized when using the `INSERT` statement.
- The code below demonstrates both ways of inserting multiple records into a table.

multiple_inserts.py

```
1. from mysql_utils import get_connection
2.
3. def multi_inserts():
4.     connection = get_connection()
5.     cursor = connection.cursor()
6.     sql = "INSERT INTO people VALUES (%s, %s, %s)"
7.
8.     mgr, adm, dev = ("manager", "administrator", "developer")
9.     records = [("Kim", mgr, 95000), ("Tom", adm, 45000),
10.               ("Pat", dev, 85000)]
11.     for record in records:
12.         cursor.execute(sql, record)
13.     connection.commit()
14.
15.     records = [("Kimmy", mgr, 95000), ("Tommy", adm, 45000),
16.               ("Patty", dev, 85000)]
17.     cursor.executemany(sql, records)
18.     connection.commit()
19.
20.     cursor.close()
21.     connection.close()
22.
23. def main():
24.     multi_inserts()
25.
26. if __name__ == "__main__":
27.     main()
```

- ▶ The `executemany()` takes a sequence of tuples as its second argument which are batched using multiple-row syntax for optimization.

Selecting Records

- With the creation of the table and insertion of records into the table, the selecting of existing records can now be studied.
 - ▶ In the next several examples, various SELECT statements will be used to obtain information from the database table.
 - From the results of the query, various "fetch" methods will be used to process the records retrieved.

selecting_records.py

```
1. from mysql_utils import get_connection
2.
3. def select_records():
4.     connection = get_connection()
5.     sql = "SELECT * FROM people"
6.     cursor = connection.cursor()
7.     cursor.execute(sql)
8.     for index, record in enumerate(cursor.fetchall()):
9.         print(record, end = " ")
10.        if index % 2 == 1:
11.            print()
12.
13.    # Loop again by unpacking tuples
14.    cursor.execute(sql)
15.    for name, job, pay in cursor.fetchall():
16.        print(name, ":", pay)
17.
18.    cursor.close()
19.    connection.close()
20.
21. def main():
22.     select_records()
23.
24. if __name__ == "__main__":
25.     main()
```

- ▶ The `fetchall()` method fetches all remaining rows from the Cursor, and returns it as a nested list of tuple objects.
 - The second for loop in the example above unpacks each tuple into three variables.

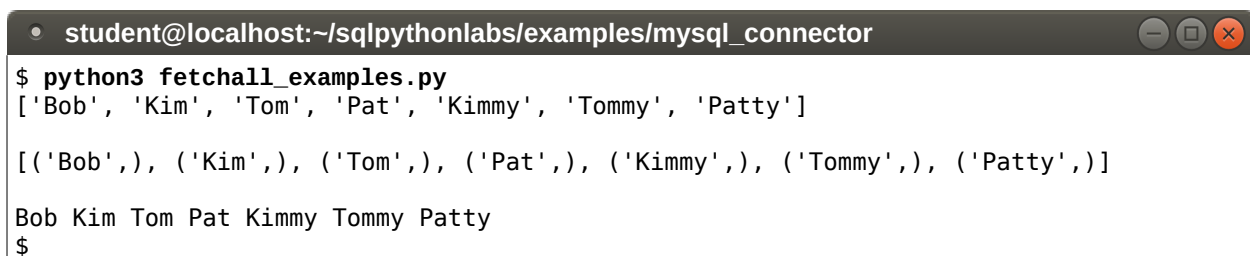
Selecting Records

- The next example uses a list comprehension to build a list of the names from the results returned from `fetchall()`.
 - ▶ It is then followed by a modification to the select query to limit the data in each record returned from the database.
 - Since only the name column is part of the `SELECT` query, much less data is retrieved from the table into the cursor object.

`fetchall_examples.py`

```
1. from mysql_utils import get_connection
2.
3. def select_records():
4.     connection = get_connection()
5.     sql = "SELECT * FROM people"
6.     cursor = connection.cursor()
7.     cursor.execute(sql)
8.     names = [row[0] for row in cursor.fetchall()]
9.     print(names, end="\n\n")
10.
11.     sql = "SELECT NAME FROM people"
12.     cursor.execute(sql)
13.     names = cursor.fetchall()
14.     print(names, end="\n\n")
15.
16.     for name in names:
17.         print(name[0], end=" ")
18.     print()
19.
20.     cursor.close()
21.     connection.close()
22.
23. def main():
24.     select_records()
25.
26. if __name__ == "__main__":
27.     main()
```

- The output of the above program is shown below.



```
student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 fetchall_examples.py
['Bob', 'Kim', 'Tom', 'Pat', 'Kimmy', 'Tommy', 'Patty']

[('Bob',), ('Kim',), ('Tom',), ('Pat',), ('Kimmy',), ('Tommy',), ('Patty',)]

Bob Kim Tom Pat Kimmy Tommy Patty
$
```

Selecting Records

- The `fetchone()` and `fetchmany()` methods can be called to better control the number of records fetched at a time from the database.
 - ▶ The following example demonstrates these two methods.

`fetchone_fetchmany.py`

```

1. from mysql_utils import get_connection
2.
3. def select_records():
4.     connection = get_connection()
5.     sql = "SELECT * FROM people"
6.     cursor = connection.cursor()
7.     cursor.execute(sql)
8.     for i in range(3):
9.         row = cursor.fetchone()
10.        if not row:
11.            break # fetchone() returns None when no more data exists
12.        print(row)
13.
14.    separator = "*" * 50
15.    print(separator, cursor.fetchone(), sep="\n")
16.    print(separator, cursor.fetchmany(2), sep="\n")
17.    print(separator, cursor.fetchmany(2), sep="\n")
18.    cursor.close()
19.    connection.close()
20.
21. def main():
22.     select_records()
23.
24. if __name__ == "__main__":
25.     main()

```

- The output of the above program is shown below.

```

• student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 fetchone_fetchmany.py
('Bob', 'developer', 50000)
('Kim', 'manager', 95000)
('Tom', 'administrator', 45000)
*****
('Pat', 'developer', 85000)
*****
[('Kimmy', 'manager', 95000), ('Tommy', 'administrator', 45000)]
*****
[('Patty', 'developer', 85000)]
$

```

Additional SQL Queries

- The next several examples demonstrate various other types of SQL statements such as using a WHERE clause, UPDATE and DELETE statements.

where.py

```
1. from mysql_utils import get_connection
2.
3. def select_records_where():
4.     connection = get_connection()
5.     sql = "SELECT job, name FROM people WHERE pay > 60000"
6.     cursor = connection.cursor()
7.     cursor.execute(sql)
8.     for job, name in cursor.fetchall():
9.         print(job, name, sep=" ")
10.
11.     cursor.close()
12.     connection.close()
13.
14. def main():
15.     select_records_where()
16.
17. if __name__ == "__main__":
18.     main()
```

- ▶ The output of the above program is shown below.



```
student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 where.py
manager:Kim
developer:Pat
manager:Kimmy
developer:Patty
$
```

- The above sql command could have also been written using a placeholder parameter in the statement as shown in the next example.
 - ▶ The example also incorporates an ORDER BY into the SQL statement.

Additional SQL Queries

order_by.py

```
1. from mysql_utils import get_connection
2.
3. def orderby():
4.     connection = get_connection()
5.     sql = "SELECT job, name FROM people WHERE pay > %s ORDER BY name"
6.     cursor = connection.cursor()
7.     cursor.execute(sql, [60000])
8.     for job, name in cursor.fetchall():
9.         print(job, name, sep=":")
10.
11.     cursor.close()
12.     connection.close()
13.
14. def main():
15.     orderby()
16.
17. if __name__ == "__main__":
18.     main()
```

► The output of the above program is shown below.



A terminal window titled "student@localhost:~/sqlpythonlabs/examples/mysql_connector" displays the output of running the script. The prompt is "\$ python3 order_by.py". The output shows four lines of data: "manager:Kim", "manager:Kimmy", "developer:Pat", and "developer:Patty". The prompt "\$" appears again at the end.

```
student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 order_by.py
manager:Kim
manager:Kimmy
developer:Pat
developer:Patty
$
```

Updates

- In the example below, an UPDATE is performed on the records inside of the people table.

update_records.py

```
1. from mysql_utils import get_connection
2.
3. def update_records():
4.     salaries = (70000, 65000)
5.     connection = get_connection()
6.     sql = "SELECT * FROM people WHERE pay <= %s"
7.     cursor = connection.cursor()
8.     cursor.execute(sql, [salaries[1]])
9.     for name, job, pay in cursor.fetchall():
10.         print(name, job, pay)
11.
12.     print("'" * 50)
13.     sql = "UPDATE people SET pay = %s WHERE pay <= %s"
14.     cursor = connection.cursor()
15.     cursor.execute(sql, salaries)
16.     connection.commit()
17.
18.     sql = "SELECT * FROM people"
19.     cursor = connection.cursor()
20.     cursor.execute(sql)
21.     for name, job, pay in cursor.fetchall():
22.         print(name, job, pay)
23.
24.     cursor.close()
25.     connection.close()
26.
27. def main():
28.     update_records()
29.
30. if __name__ == "__main__":
31.     main()
```

• student@localhost:~/sqlpythonlabs/examples/mysql_connector

```
$ python3 update_records.py
Bob developer 50000
Tom administrator 45000
Tommy administrator 45000
*****
Bob developer 70000
Kim manager 95000
Tom administrator 70000
Pat developer 85000
Kimmy manager 95000
Tommy administrator 70000
Patty developer 85000
$
```


Deletes

- The following example uses a DELETE statement to remove all records from the table where the name field is LIKE to "Pat%".
 - ▶ The % acts as a wild card and as such applies to all names that start with "Pat"

delete_records.py

```
1. from mysql_utils import get_connection
2.
3. def delete_records():
4.     connection = get_connection()
5.     sql = "DELETE FROM people WHERE name LIKE 'Pat%'"
6.     cursor = connection.cursor()
7.     cursor.execute(sql)
8.     connection.commit()
9.
10.    sql = "SELECT * FROM people"
11.    cursor = connection.cursor()
12.    cursor.execute(sql)
13.    for name, job, pay in cursor.fetchall():
14.        print(name, job, pay)
15.
16.    cursor.close()
17.    connection.close()
18.
19. def main():
20.     delete_records()
21.
22. if __name__ == "__main__":
23.     main()
```

- ▶ Running the above program removes both Pat and Patty from the people table as seen in the output below.



```
• student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 delete_records.py
Bob developer 70000
Kim manager 95000
Tom administrator 70000
Kimmy manager 95000
Tommy administrator 70000
$
```

Table Descriptions

- The Python Database API defines a `description` attribute for the `Cursor` object that after a select query has been executed, the `Cursor` holds information about the data in this attribute.
 - ▶ The specification defines the attribute as the following sequence of tuples for each column in the result:
`(name, type, display_size, internal_size, precision, scale, null_ok)`
 - The first two items are mandatory, the other five are optional and set to `None` if not meaningful values are provided.
 - ▶ The MySQL driver provides valid information about the name, type and `null_ok` and adds an additional `column_flags` to the end of the tuple.
`(name, type, None, None, None, None, null_ok, column_flags)`
 - The `column_flags` value is an instance of the `mysql.connector.constants.FieldFlag` class
 - The following function can aid in interpreting the `column_flags` value.

fieldflag_descriptions.py

```
1. from mysql.connector import FieldFlag
2.
3. def print_descriptions():
4.     flag_keys = list(FieldFlag.desc.keys())
5.     flag_keys.sort(key=lambda x:FieldFlag.desc.get(x)[0])
6.
7.     longest = len(max(flag_keys, key=len))
8.     fmt_string = "{:>{}} : {}".format
9.     for flag_key in flag_keys:
10.         flag_value = FieldFlag.desc.get(flag_key)
11.         print(fmt_string.format(flag_key, longest, flag_value))
12.
13.
14. if __name__ == "__main__":
15.     print_descriptions()
```

Table Descriptions

- The following example uses the `cursor.description` to describe each of the columns in the `people` table.
 - ▶ It also relies on the `FieldType.get_info()` function to display type of the column in a more meaningful way than just the number returned in the tuple.

`table_descriptions.py`

```
1. from mysql_utils import get_connection
2. from mysql.connector import FieldType
3. import fieldflag_descriptions
4.
5. def table_description():
6.     connection = get_connection()
7.     sql = "SELECT * FROM people"
8.     cursor = connection.cursor()
9.     cursor.execute(sql)
10.    for column_description in cursor.description:
11.        print("Name: ", column_description[0])
12.        field_type = column_description[1]
13.        print("Type: ", field_type, FieldType.get_info(field_type))
14.        print("Flag: ", column_description[7])
15.        print()
16.
17.    print("Flag Descriptions:")
18.    fieldflag_descriptions.print_descriptions()
19.
20.
21. def main():
22.     table_description()
23.
24. if __name__ == "__main__":
25.     main()
```

- ▶ The output from the above program is shown on the following page.

Table Descriptions

```

• student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 table_descriptions.py
Name: name
Type: 253 VAR_STRING
Flag: 0

Name: job
Type: 253 VAR_STRING
Flag: 0

Name: pay
Type: 3 LONG
Flag: 0

Flag Descriptions:
    NOT_NULL : (1, "Field can't be NULL")
    PRI_KEY : (2, 'Field is part of a primary key')
    UNIQUE_KEY : (4, 'Field is part of a unique key')
    MULTIPLE_KEY : (8, 'Field is part of a key')
    BLOB : (16, 'Field is a blob')
    UNSIGNED : (32, 'Field is unsigned')
    ZEROFILL : (64, 'Field is zerofill')
    BINARY : (128, 'Field is binary ')
    ENUM : (256, 'field is an enum')
    AUTO_INCREMENT : (512, 'field is a autoincrement field')
    TIMESTAMP : (1024, 'Field is a timestamp')
    SET : (2048, 'field is a set')
    NO_DEFAULT_VALUE : (4096, "Field doesn't have default value")
    ON_UPDATE_NOW : (8192, 'Field is set to NOW on UPDATE')
    GROUP : (16384, 'Intern: Group field')
    NUM : (16384, 'Field is num (for clients)')
    PART_KEY : (32768, 'Intern; Part of some key')
    UNIQUE : (65536, 'Intern: Used by sql_yacc')
    BINCMP : (131072, 'Intern: Used by sql_yacc')
    GET_FIXED_FIELDS : (262144, 'Used to get fields in item tree')
    FIELD_IN_PART_FUNC : (524288, 'Field part of partition func')
    FIELD_IN_ADD_INDEX : (1048576, 'Intern: Field used in ADD INDEX')
    FIELD_IS_RENAMED : (2097152, 'Intern: Field is being renamed')
$

```

- A table with a more intricate schema would have more meaningful information available in the `FieldFlag` value.

Writing Database Scripts

- The following application combines many of the SQL statements and Cursor method calls seen throughout this chapter.

complete.py

```
1. #!/usr/bin/env python3
2. from mysql_utils import get_connection
3. from mysql.connector import FieldType, FieldFlag
4.
5. def drop_table():
6.     sql = "DROP TABLE IF EXISTS peopleII"
7.     cursor = connection.cursor()
8.     cursor.execute(sql)
9.     cursor.close()
10.
11. def create_table():
12.     sql = ("CREATE TABLE IF NOT EXISTS peopleII ("
13.           " id MEDIUMINT NOT NULL AUTO_INCREMENT,"
14.           " name VARCHAR(64) NOT NULL,"
15.           " job ENUM('admin', 'manager', 'developer') NOT NULL,"
16.           " pay INTEGER,"
17.           " PRIMARY KEY (id)"
18.           ")")
19.     cursor = connection.cursor()
20.     cursor.execute(sql)
21.     cursor.close()
22.
23.
24. def populate_table():
25.     sql_cmd = 'INSERT INTO peopleII VALUES (null, %s, %s, %s)'
26.     rows = [
27.         ['mike', 'developer', 50000], ['joan', 'manager', 70000],
28.         ['mike', 'admin', 40000], ['peter', 'admin', 40000],
29.         ['larry', 'developer', 55000], ['moe', 'manager', 60000],
30.         ['curley', 'admin', 45000]
31.     ]
32.     cursor = connection.cursor()
33.     cursor.executemany(sql_cmd, rows)
34.     connection.commit()
35.     cursor.close()
36.
```

- Code continued on the following page

Writing Database Scripts

complete.py ~ *continued*

```
37. def from_user_input():
38.     sql_cmd = 'INSERT INTO peopleII VALUES(null, %s, %s, %s)'
39.     prompt = "Enter: <name> <job> <pay> (Or <enter> when done)"
40.     cursor = connection.cursor()
41.     while True:
42.         line = input(prompt)
43.         if not line:
44.             break
45.         cursor.execute(sql_cmd, line.split())
46.     connection.commit()
47.     cursor.close()
48.
49. def display_table():
50.     sql = "SELECT * FROM peopleII"
51.     cursor = connection.cursor()
52.     cursor.execute(sql)
53.     for row in cursor.fetchall():
54.         print(row)
55.     print()
56.
57.     for column_description in cursor.description:
58.         print("Name: ", column_description[0])
59.         field_type = column_description[1]
60.         print("Type: ", field_type, FieldType.get_info(field_type))
61.         field_flag = column_description[7]
62.         print("Flag: ", field_flag, FieldFlag.get_bit_info(field_flag))
63.         print()
64.
65.     cursor.close()
66.
67.
68. def main():
69.     global connection
70.     connection = get_connection()
71.     drop_table()
72.     create_table()
73.     populate_table()
74.     from_user_input()
75.     display_table()
76.     connection.close()
77.
78. if __name__ == "__main__":
79.     main()
```

► The output of the above program is shown on the following page.

Writing Database Scripts

```
student@localhost:~/sqlpythonlabs/examples/mysql_connector
$ python3 complete.py
Enter: <name> <job> <pay> (Or <enter> when done)Dave developer 75000
Enter: <name> <job> <pay> (Or <enter> when done)
(1, 'mike', 'developer', 50000)
(2, 'joan', 'manager', 70000)
(3, 'mike', 'admin', 40000)
(4, 'peter', 'admin', 40000)
(5, 'larry', 'developer', 55000)
(6, 'moe', 'manager', 60000)
(7, 'curley', 'admin', 45000)
(8, 'Dave', 'developer', 75000)

Name: id
Type: 9 INT24
Flag: 16899 ['NOT_NULL', 'PRI_KEY', 'AUTO_INCREMENT', 'NUM', 'GROUP']

Name: name
Type: 253 VAR_STRING
Flag: 4097 ['NOT_NULL', 'NO_DEFAULT_VALUE']

Name: job
Type: 254 STRING
Flag: 4353 ['NOT_NULL', 'NO_DEFAULT_VALUE', 'ENUM']

Name: pay
Type: 3 LONG
Flag: 0 []

$
```

Exercises

1. Write a script that reads from the comma-separated customers file used earlier in the course.
 - ▶ It should insert the data into a sqlite3 database.
 - ▶ It should then prompt for a state as input and print just those customers who live in the state specified.

This Page Intentionally Left Blank

Chapter 16:

HTML

A Brief History of the World Wide Web

- In 1980 Tim Berners-Lee with the European Laboratory for Particle Physics in Geneva, Switzerland (CERN) developed hypertext, links within pages that linked to other pages.
 - ▶ The purpose of creating hypertext was for sharing scientific information across their network.
- In 1990 Lynx, the first hypertext browser, was developed at the University of Kansas running under UNIX and VMS.
- In 1990 Tim Berners-Lee coined the name "World Wide Web".
- In 1992 CERN made software available to research centers and universities to access the Internet for research purposes.
 - ▶ Mosaic was created in 1992 by Marc Andreessen and Eric Bina with the National Center for Supercomputing Applications (NCSA). Running on the X Windows system, it was the first graphical browser.
- In 1993 the Mosaic browser was released for the Apple Macintosh and Microsoft Windows.
- In 1994 the W3C (World Wide Web Consortium) was founded at MIT to oversee the development of web standards, including the HTML standard.
 - ▶ The current standard is HTML 5.1.
- The prevalence of Web browsers on computers lends itself to a very common interface to many programs in many programming languages.
 - ▶ The next several chapters will introduce several of the fundamental pieces of building a web based application that provides an HTML front end to a server running Python on the back end.
 - Specifically the following building blocks will be discussed: HTML, Cascading Style Sheets (CSS), and JavaScript.
 - This chapter begins with the introduction to HTML.

HTML Basics

- Web Pages are written in the HyperText Markup Language (HTML).
 - ▶ HTML was primarily designed for formatting text, but has evolved to create dynamic web pages allowing clients to view images, link to other web pages or sites, submit forms, and access databases.
 - ▶ HTML is compatible on all computer platforms.
- With HTML 5, richer and more complex Web pages can be created on an increasing range of browser platforms including cell phones, televisions, cars, kiosks, and desktops.
 - ▶ Most current browsers support HTML 5 to some extent but might not be 100% compliant.
 - The HTML5 specification can be found at the following URL:
<https://www.w3.org/TR/html5/>
 - ▶ Most of the HTML discussed in this chapter have been around prior to version 5 of HTML and as such work more reliably in all browsers.
- HTML documents are simple text files created in any text editor such as Notepad or gedit and are typically saved with a .html extension.
- HTML is not a programming language.
 - ▶ It doesn't use variables, sub-routines, loops, or other constructs normally associated with programming languages.
 - ▶ A compiler is not used to display the document and the only "interpreter" is the web browser.
- Basic website structure:
 - ▶ A web page is a single HTML document within a website.
 - ▶ Websites are comprised of one or more web pages.
 - ▶ A homepage is the uppermost web page in a website.

HTML Tags

- HTML code is comprised of tags and their associated attributes.
 - ▶ These tags and attributes identify the structural components within a web page.
 - ▶ HTML tags can be used to manipulate the size, color, and location of text as well as incorporate images and hyperlinks into web pages.
- HTML tags are typically paired with start and end tags.
 - ▶ Starting tags are enclosed within brackets `<tag>`.
 - ▶ Ending tags are enclosed within brackets with a `/` preceding the tag name `</tag>`.
 - ▶ Any text or content placed between the starting and closing tags will be formatted in accordance to that tag specification.
- Some HTML tags are unpaired or self closing.
 - ▶ This means that they do not have an end tag.
 - This also implies that the tag cannot enclose anything.
 - ▶ The HTML specification refers to these as "void elements".
 - ▶ Unpaired tags follow the rules of a start tag, followed by an optional slash as the last character in the tag.
 - The break tag is a good example of this; `
` or `
`
- Tags may be nested within other tags to manipulate content placed between both tags.

`<u><i>This text will be underlined and italicized.</i></u>`
- Browsers collapse contiguous sequences of whitespace into a single space within HTML documents.
 - ▶ Long lines of text are typically wrapped in the Browser display.
- Note: Different browsers may display formatting differently.
 - ▶ Always test your web design on various web browsers.

The HTML Root Element

- The `html` element, represented by the `<html></html>` start and end tags respectively act as the root element of every HTML document.
 - ▶ At a bare minimum its content model is composed of a head element followed by a body element.
 - The head element will typically be composed of a nested `title` element for the document.
- The HTML document below shows the skeletal tags that every HTML document is typically composed of.
 - ▶ The `<!doctype html>` tag is used to indicate to browsers that the document uses HTML version 5.

skeletal.html

```
1. <!doctype html>
2. <html>
3.     <head>
4.         <title>Skeletal Page</title>
5.     </head>
6.     <body>
7.         This page is simply a template
8.         for a basic HTML document
9.     </body>
10. </html>
```

- ▶ Opening the above document in a browser displays the file as follows.



- ▶ Note how the whitespace is collapsed and the text is all on one line.

HTML Sections

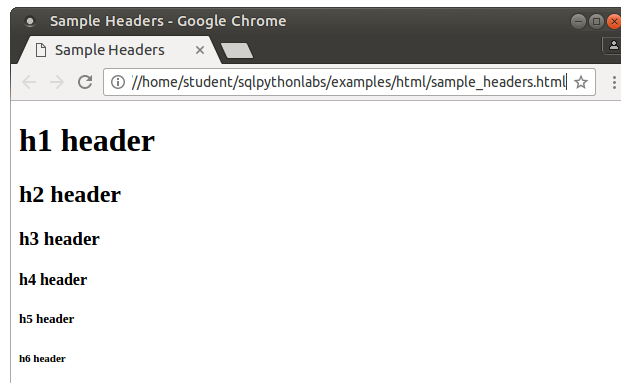
- The head element represents a collection of metadata for the entire document.
 - ▶ The `title` element, can serve several purposes within the head of the document.
 - It is used as the title of the browser window and should be descriptive of what the page represents.
 - It is also used as the default text of a bookmark if the page is bookmarked by the user.
 - ▶ Two other common elements found in the head are the `link` and `script` elements that indicate references to Cascading Style Sheets and JavaScript respectively.
 - The use of these elements within the head will be seen in the coming chapters that deal with each of the topics they represent.
- The body element, represented by the `<body></body>` start and end tags respectively represents the body of the document.
 - ▶ It is within this set of tags where all of the content to display on the page is nested.
- The `h1`, `h2`, `h3`, `h4`, `h5`, and `h6` elements are often referred to as heading elements.
 - ▶ The elements have a rank given by their number in their name.
 - `h1` is the highest rank and is usually rendered as the largest text of the six headers.
 - `h6` is the lowest rank and is usually rendered as the smallest text of the six headers.
- All of the section elements defined within the specification can be found at the following URL:
 - ▶ <https://www.w3.org/TR/html5/sections.html>

HTML Sections

- The example below demonstrates the header elements defined on the previous page.

sample_headers.html

```
1. <!doctype html>
2. <html>
3.     <head>
4.         <title>Sample Headers</title>
5.     </head>
6.     <body>
7.         <!-- This is an HTML comment -->
8.         <h1>h1 header</h1><h2>h2 header</h2>
9.         <h3>h3 header</h3><h4>h4 header</h4>
10.        <h5>h5 header</h5><h6>h6 header</h6>
11.    </body>
12. </html>
```



- ▶ Although there are multiple elements defined on a single line in the HTML file, each element is displayed on its own line in the resulting web page.
 - This is because the header elements are "block level elements".
 - Browsers typically display block level elements with the equivalent of a newline both before and after the element.
- ▶ The above code also introduces HTML comments.
 - An HTML comment is everything between the <!-- and --> and is not displayed in anyway by the browser as seen above

Grouping Content

- Grouping elements are those elements that can be used to group related content together in various ways.
 - ▶ The following are some of the available grouping elements.

Grouping Elements	
Element	Representation
p	A paragraph.
hr	A paragraph level break as a horizontal rule.
pre	A block of preformatted text.
blockquote	Content that is quoted from another source.
ol	An order list of items
ul	An unordered list of items
li	An item within an ordered or unordered list element.
dl	A descriptive list of term/description items.
dt	A term within a descriptive list.
dd	A description within a descriptive list.
div	Used solely as a grouping mechanism.

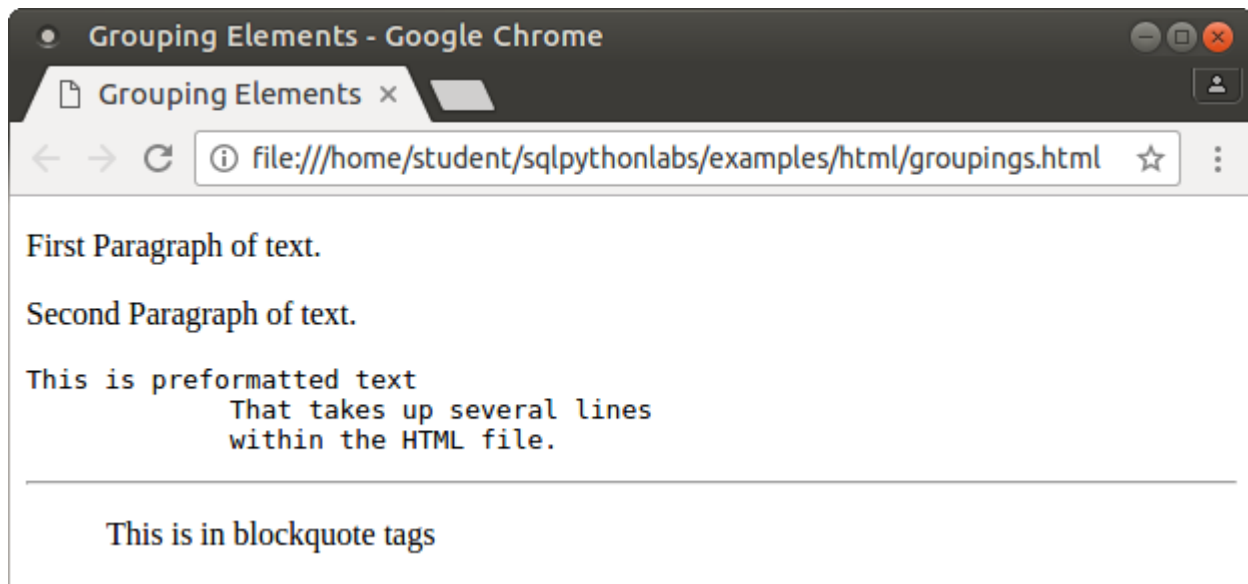
- The next two examples demonstrate the above grouping elements.

Grouping Content

groupings.html

```
1. <!doctype html>
2. <html>
3.     <head><title>Grouping Elements</title></head>
4.     <body>
5.         <p>First Paragraph of text.</p>
6.         <p>Second Paragraph of text.</p>
7.         <pre>This is preformatted text
8.             That takes up several lines
9.             within the HTML file.</pre>
10.        <hr/>
11.        <blockquote>This is in blockquote tags</blockquote>
12.    </body>
13. </html>
```

- ▶ The above HTML is shown in a browser as seen below.



- The next example demonstrates the elements that group content into lists.

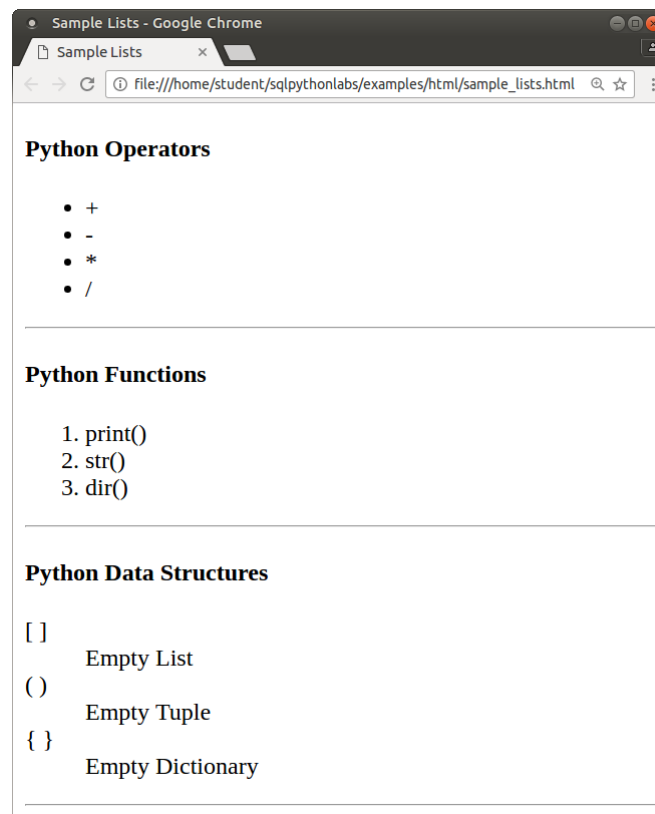
Grouping Content

lists.html

```

1. <!doctype html>
2. <html>
3.     <head><title>Sample Lists</title></head>
4.     <body>
5.         <h4>Python Operators</h4>
6.         <ul><li>+</li><li>-</li><li>*</li><li>/</li></ul>
7.         <hr/>
8.         <h4>Python Functions</h4>
9.         <ol><li>print()</li><li>str()</li><li>dir()</li></ol>
10.        <hr/>
11.        <h4>Python Data Structures</h4>
12.        <dl>
13.            <dt>[ ]</dt><dd>Empty List</dd>
14.            <dt>( )</dt><dd>Empty Tuple</dd>
15.            <dt>{ }</dt><dd>Empty Dictionary</dd>
16.        </dl>
17.        <hr/>
18.    </body>
19. </html>

```



- Changing the bullets and numbering in the lists will be shown in the chapter on Cascading Style Sheets.

Text Level Elements

- Text level elements are those elements that are used to add structure and meaning to the text that they surround.
 - ▶ The following are some of the available text level elements.

Text Level Elements	
Element	Representation
i	Historically meant simply to display in italics. Represents an alternative voice or mood in HTML5
em	Stresses emphasis of its contents
b	historically meant simply to display in bold. Represents a mechanism to draw attention to the text.
strong	Stresses an urgency to its contents
u	Historically meant to display text as underlined.
small	Conveys the text as small print.
code	A fragment of computer code.
kbd	Keyboard I/O
br	A line break
span	Nothing on its own, but helpful with Cascading Style Sheets.

- The next example demonstrates several of the above text level elements.

text_level_elements.html

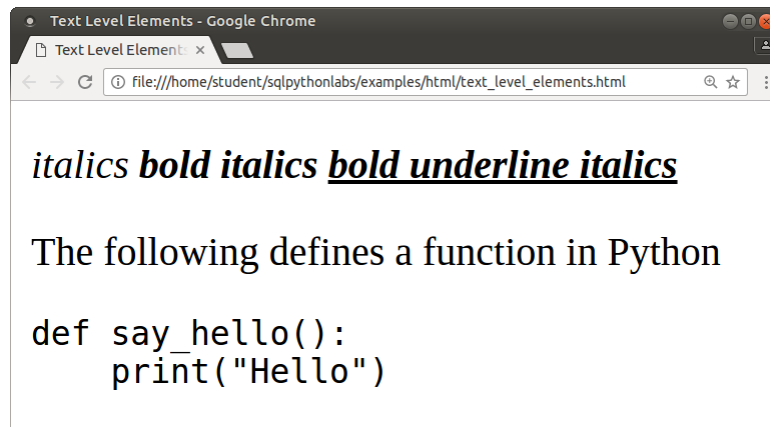
```

1. <!doctype html>
2. <html>
3.   <head><title>Text Level Elements</title></head>
4.   <body>
5.     <p><i>italics <b>bold italics
6.     <u>bold underline italics</u></b></i></p>
7.     <p>The following defines a function in Python<br/>
8.     <pre><code>def say_hello():<br/>    print("Hello")</code>
9.     </pre></p>
10.   </body>
11. </html>

```

- ▶ The above HTML is shown in a browser on the following page.

Text Level Elements



- As shown in the previous example, the tags can be nested inside of each other to apply several styles to text.
- The majority of the text level elements are "inline" elements as opposed to the earlier discussed block level elements.
 - ▶ An inline element does not have newlines before or after the element like block level elements do.

Links and Anchors

- Another text level element that is commonly used is the `a` element.
 - ▶ The `a` element is used to create what is referred to as a link or hyperlink.
 - ▶ A link or anchor is a connection from one Web resource to another.
 - The resource may be any Web resource (e.g., an image, a video clip, a sound bite, a program, an HTML document, an element within an HTML document, etc.).
 - ▶ The default behavior associated with a link is the retrieval of another Web resource.
 - This behavior is commonly obtained by selecting the link.
 - ▶ The link relies on an `href` attribute that specifies the address of the destination with a URI.
 - Although not seen in any of the start tags of the elements discussed so far, attributes are very common within elements.
 - An attribute is a name/value pair separated with an `=` sign that provides additional information for the element is defined for.
- The example that follows uses several `a` elements to define several links and anchors within the document.

`links_and_anchors.html`

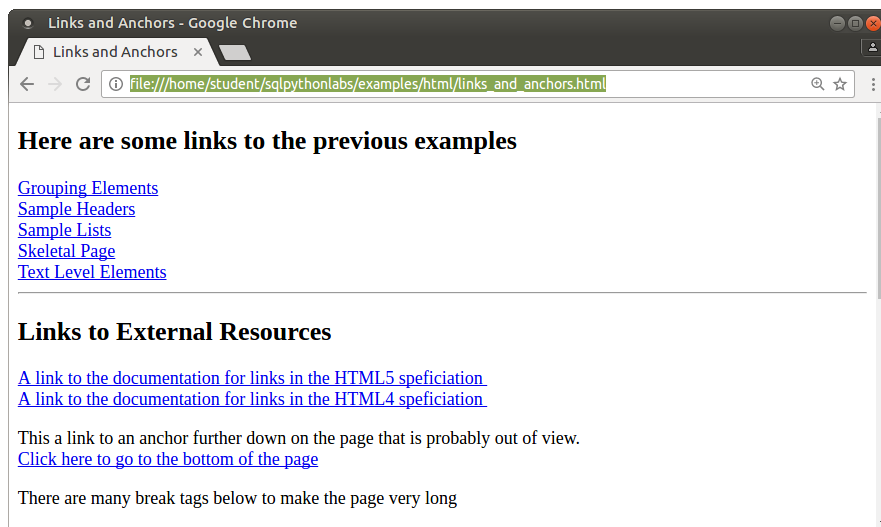
```
1. <!doctype html>
2. <html>
3.     <head><title>Links and Anchors</title></head>
4.     <body>
5.         <a name="top"/>
6.         <h2>Here are some links to the previous examples</h2>
7.         <a href="groupings.html">Grouping Elements</a><br/>
8.         <a href="sample_headers.html">Sample Headers</a><br/>
9.         <a href="sample_lists.html">Sample Lists</a><br/>
10.        <a href="skeletal.html">Skeletal Page</a><br/>
11.        <a href="text_level_elements.html">Text Level Elements</a><br/>
12.        <hr/>
13.        <h2>Links to External Resources</h2>
14.        <a href="https://www.w3.org/TR/html5">
15.            A link to the documentation for links in HTML5</a><br/>
```

Links and Anchors

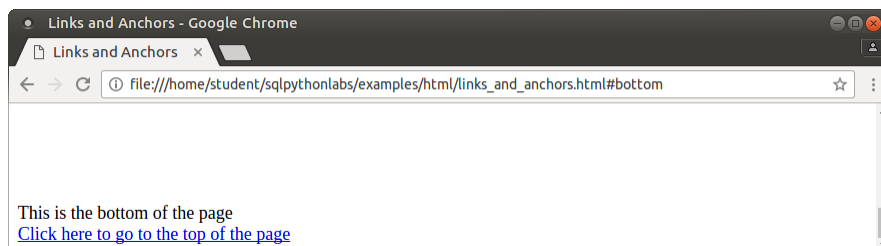
links_and_anchors.html ~ *continued*

```
16.      <a href="https://www.w3.org/TR/html4/struct">
17.          A link to the documentation for links in HTML4</a><br/>
18.      <p>This a link to an anchor further down on the page
19.          that is probably out of view.<br>
20.      <a href="#bottom">Click here to go to the bottom of the page</a>
21.      </p>
22.      There are many break tags below to make the page very long
23.      <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
24.      <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
25.      <a name="bottom">This is the bottom of the page</a><br/>
26.      <a href="#top">Click here to go to the top of the page</a>
27.  </body>
28. </html>
```

- ▶ The display of the above web page is shown below.



- ▶ Clicking on the link "Click here to go to the bottom of the page" scrolls to the bottom of the page where the a element acts as an anchor within the same web page.



Images

- An `img` element represents an image.
 - ▶ The image given as the `src` attribute is the embedded content of the element.
 - ▶ The `alt` attribute provides a mechanism for making visual information accessible by providing a text alternative to the information being rendered.
- The example below demonstrates the use of the `img` element within a web page to display graphics.

`images.html`

```
1. <!doctype html>
2. <html>
3.     <head><title>Images</title></head>
4.     <body>
5.         <div></div>
7.         <hr />
8.         <h1>Buttons
9.         
10.        
11.        </h1>
12.        <hr />
13.        <h1>A Clickable image</h1>
14.        <a href="http://www.umbctraining.com">
15.            <a/>
17.        </body>
18. </html>
```



Tables

- The table element represents data typically as a two dimensional table.
 - ▶ Tables have rows, columns and cells that form the grid of the table.
 - The tr element represents a row of cells in a table.
 - The td element represents a cell in a table.
 - The th element can be used in place of a tr where the row acts as a header row for the cells below it.
 - ▶ The td and th elements may use colspan and/or rowspan attributes to indicate the columns and rows to span in the table.

tables.html

```

1. <!doctype html>
2. <html>
3.     <head><title>Table Elements</title></head>
4.     <body>
5.         <table border="1">
6.             <tr>
7.                 <td colspan="2" rowspan="2">A</td><td>B</td>
8.                 <td colspan="2">C</td>
9.             </tr>
10.            <tr><td>D</td><td>E</td><td>F</td></tr>
11.            <tr><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td></tr>
12.            <tr>
13.                <td rowspan="2">L</td>
14.                <td>M</td><td>N</td><td>O</td><td>P</td>
15.            </tr>
16.            <tr><td>Q</td><td colspan="3">R</td></tr>
17.        </table>
18.    </body>
19. </html>

```

A		B	C	
		D	E	F
G	H	I	J	K
L	M	N	O	P
	Q	R		

HTML Forms

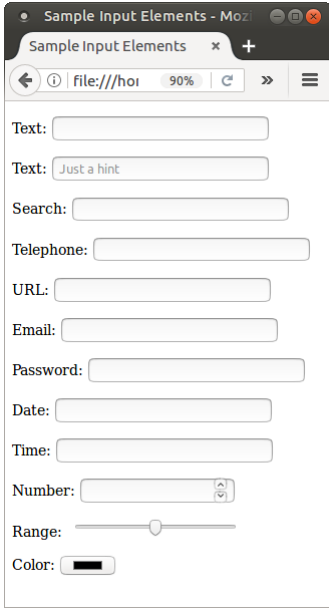
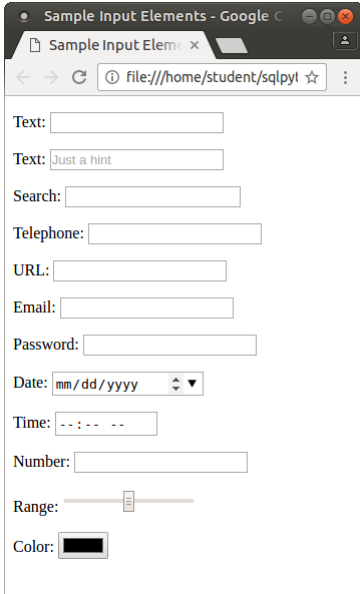
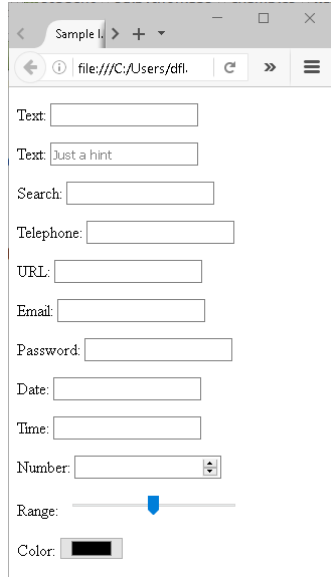
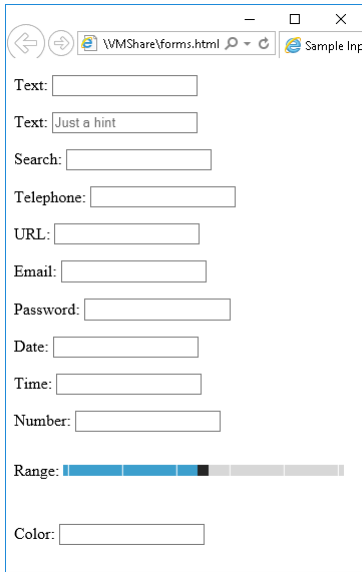
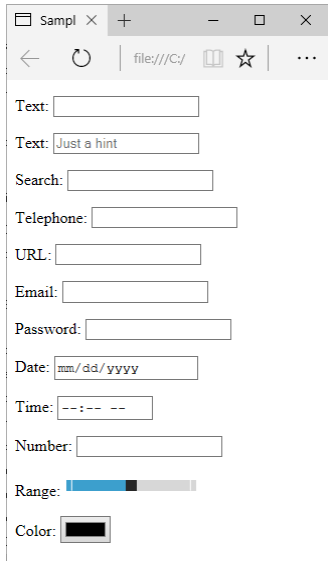
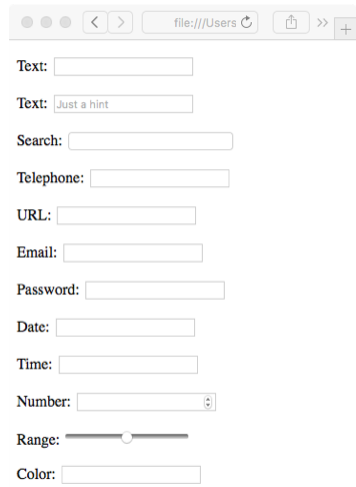
- A form is an element of a Web page that has controls, such as text fields, buttons, checkboxes, range controls, or color pickers.
 - ▶ A user typically interacts with a form by providing data that is sent to the server for processing.
- An input element provides a data field that typically allows a user to enter and edit the data it contains.
 - ▶ The type attribute on an input element can be used to specify the type of input to present.
 - Several of the different types of input are shown in the example below.

sample_inputs.html

```
1. <!doctype html>
2. <html>
3.     <head><title>Sample Input Elements</title></head>
4.     <body>
5.         <form>
6.             <p><label>Text: <input type="text"/></label></p>
7.             <p><label>Text:
8.                 <input type="text" placeholder="Just a hint"/></label></p>
9.             <p><label>Search: <input type="search"/></label></p>
10.            <p><label>Telephone: <input type="tel"/></label></p>
11.            <p><label>URL: <input type="url"/></label></p>
12.            <p><label>Email: <input type="email"/></label></p>
13.            <p><label>Password: <input type="password"/></label></p>
14.            <p><label>Date: <input type="date"/></label></p>
15.            <p><label>Time: <input type="time"/></label></p>
16.            <p><label>Number: <input type="number"/></label></p>
17.            <p><label>Range: <input type="range"/></label></p>
18.            <p><label>Color: <input type="color"/></label></p>
19.        </form>
20.    </body>
21. </html>
```

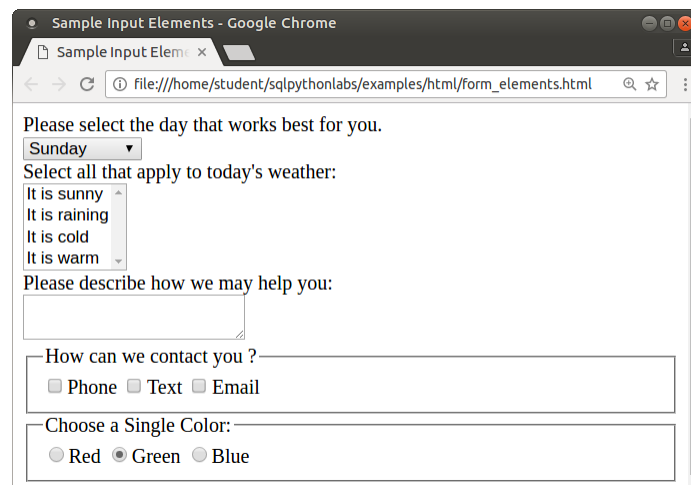
- ▶ Currently, there is no mechanism to submit the data entered in the above web page, which will be shown shortly in additional examples.
 - The presentation of the above web page is shown on the following page as it appears in various browsers.

HTML Forms

HTML Forms in Various Browsers and Operating Systems		
Firefox on Ubuntu 16.04	Chrome on Ubuntu 16.04	Firefox on Windows 10
		
Internet Explorer 11 on Windows 10	Microsoft Edge on Windows 10	Safari on Mac OS X
		

HTML Forms

- Several attributes are common to many of the input elements.
 - ▶ Some of the common attributes are listed below.
 - minlength
 - maxlength
 - size
 - readonly
 - required
 - multiple
 - pattern
 - min
 - max
 - step
 - list
 - placeholder
 - ▶ Complete documentation for each of the above attributes can be found within the HTML documentation for forms at the following URL.
 - <https://www.w3.org/TR/html5/forms.html>
- In addition to the input element, there are various other types of elements that can be used within a form for the purpose of collecting data from a user.
 - ▶ Some of these additional types of elements are used in the following example.
 - Once again the example does not currently have the ability to submit the data, but that will be discussed shortly.
 - The code to produce the web page below is on the next page.



The screenshot shows a Google Chrome browser window titled "Sample Input Elements - Google Chrome". The address bar shows the file path: `file:///home/student/sqlpythonlabs/examples/html/form_elements.html`. The form content is as follows:

Please select the day that works best for you.

Select all that apply to today's weather:
☐ It is sunny
☐ It is raining
☐ It is cold
☐ It is warm

Please describe how we may help you:

How can we contact you ?
☐ Phone ☐ Text ☐ Email

Choose a Single Color:
☐ Red ☒ Green ☐ Blue

HTML Forms

form_elements.html

```

1. <!doctype html>
2. <html>
3.     <head><title>Sample Input Elements</title></head>
4.     <body>
5.         <form>
6.             <div><label>Please select your preferred day<br>
7.             <select>
8.                 <option value="Sun">Sunday</option>
9.                 <option value="Mon">Monday</option>
10.                <option value="Tue">Tuesday</option>
11.                <option value="Wed">Wednesday</option>
12.                <option value="Thu">Thursday</option>
13.                <option value="Fri">Friday</option>
14.                <option value="Sat">Saturday</option>
15.            </select></div>
16.
17.            <div><label>Select all that apply to today's weather:<br />
18.            <select multiple>
19.                <option value="sunny">It is sunny</option>
20.                <option value="raining">It is raining</option>
21.                <option value="cold">It is cold</option>
22.                <option value="warm">It is warm</option>
23.            </select>
24.            </label></div>
25.
26.            <div><label>Please describe how we may help you:<br />
27.            <textarea></textarea></label></div>
28.
29.            <fieldset>
30.                <legend>How can we contact you ?</legend>
31.                <label><input type="checkbox" name="contact"
32.                value="phone">Phone</label>
33.                <label><input type="checkbox" name="contact"
34.                value="text">Text</label>
35.                <label><input type="checkbox" name="contact"
36.                value="email">Email</label>
37.            </fieldset>
38.            <fieldset>
39.                <legend>Choose a Single Color:</legend>
40.                <label><input type="radio" name="color"
41.                value="red">Red</label>
42.                <label><input type="radio" name="color"
43.                value="green" checked>Green</label>
44.                <label><input type="radio" name="color"
45.                value="blue">Blue</label>
46.            </fieldset>
47.        </form>
48.    </body>
49. </html>

```

Submitting HTML Forms

- The forms in the previous examples lacked several important necessities to be able to submit the data in the form for further processing to a server.
 - ▶ The first thing that was missing in the forms was a `input` element of type `submit` that acts as the submit button for the form..
 - ▶ The other thing that was missing in almost all of the form elements was a `name` attribute.
 - It is the `name` attribute that allows the data or selection in the form to be submitted to the server as key/value pairs.
- While the following examples demonstrate the submitting of the form data, the actual processing of the submitted data requires a server that knows how to process the incoming HTML request.
 - ▶ For now the data entered will be visible in the query string of the request.
 - The ability to process the request using Python and sending an HTML response back to the user will be discussed in a coming chapter.
- The `name` attribute, in addition to acting as the key to a value in a form, can have other behavior depending on the element it is used with.
 - ▶ When used on radio buttons, all buttons with the same `name` are able to act as being mutually exclusive amongst the group.
 - This makes sure that only one at a time is allowed to be selected.
 - ▶ The `name` attribute, when acting as a key in a key/value pair, does not have to be unique.
 - Checkboxes often share the same `name` allowing each of them to be submitted as a collection of values with that `name`.

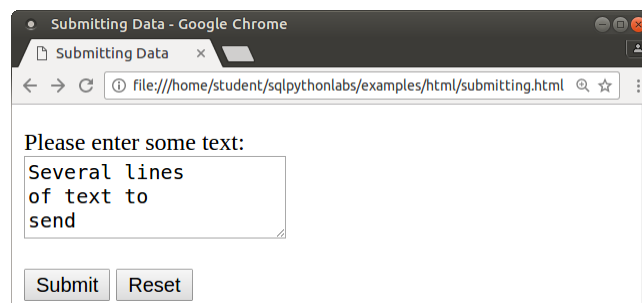
Submitting HTML Forms

- The example below includes a simple `textarea` element where data can be entered, and two `input` elements of type `submit` and `reset`.
 - ▶ Note the addition of the `name` attribute to the `textarea` element allowing the data to be included as the value associated with the name when the form is submitted.

submitting_data.html

```
1. <!doctype html>
2. <html>
3.     <head><title>Submitting Data</title></head>
4.     <body>
5.         <form>
6.             <p><label>Please enter some text:<br />
7.             <textarea name="txt"></textarea></label></p>
8.             <p><input type="submit" /> <input type="reset" />
9.         </form>
10.    </body>
11. </html>
```

- The initial view of the form is presented as follows.
 - ▶ The text typed into the text area was typed after the page loaded.



- ▶ When the submit button is clicked the request to the server includes additional information in the URL.
 - Everything past the question mark "?" in the URL is referred to as the query string and looks as follows:
`submitting?txt=Several+lines%0D%0Aof+text+to%0D%0Aend`
 - The query string is automatically URL encoded to replace characters that are not legal characters in a URL.

Character References

- Character references are numeric or symbolic names for characters that may be included in an HTML document.
 - ▶ They are useful for referring to rarely used characters, or those that authoring tools make it difficult or impossible to enter.
 - ▶ Character references begin with an ampersand "&" sign and end with a semi-colon ";".
- Four character entity references deserve special mention since they are frequently used to escape special characters:
 - ▶ "<" represents the < sign.
 - The "<" character in text should use "<" (ASCII decimal 60) to avoid possible confusion with the beginning of a tag.
 - ▶ ">" represents the > sign.
 - The ">" character in text should use ">" (ASCII decimal 62) to avoid possible confusion with the ending of a tag.
 - ▶ "&" represents the & sign.
 - The "&" character in text should use "&" (ASCII decimal 38) to avoid confusion with the beginning of a character reference.
 - ▶ """ represents the " mark.
 - The """ (ASCII decimal 38) should be used instead since the double quote mark character may be used to delimit attribute values.
- A complete list of character references can be found in the documentation at the following URL.

<https://www.w3.org/TR/html5/syntax.html#named-character-references>

This Page Intentionally Left Blank

Chapter 17:

Cascading Style Sheets

Introduction

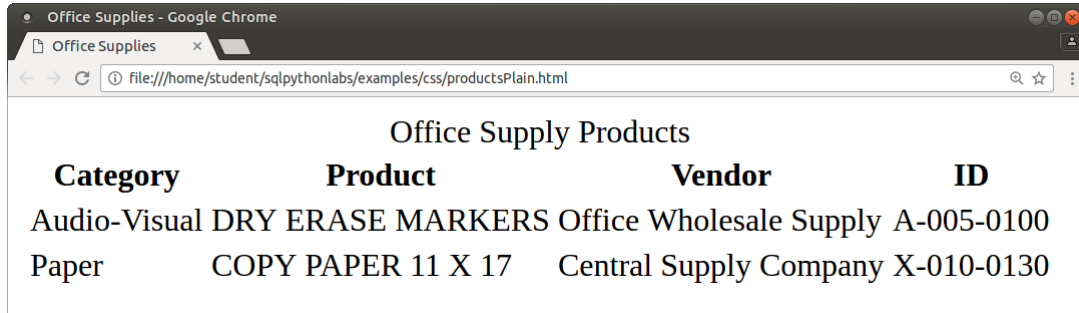
- Cascading Style Sheets (CSS) is a style sheet language that provides a mechanism to attach styles to structured documents.
 - ▶ By separating the presentation style of documents from the content of documents, CSS simplifies Web authoring and site maintenance.
 - ▶ The current World Wide Web Consortium (W3C) specification for CSS can be found at the following URL.
https://www.w3.org/standards/techs/css#w3c_all
 - CSS3 builds on the previous specifications; CSS2 and CSS1.
 - The level in which the specification is supported by the various browsers varies by browser and browser version.
 - ▶ This chapter will focus on those features of CSS that are currently implemented by most browsers.
- The example that follows contains no CSS formatting and will be used as the base file to which formatting will be applied.

products_plain.html

```
1. <!doctype html>
2. <html>
3. <head><title>Office Supplies</title></head>
4. <body>
5. <table>
6.   <caption>Office Supply Products</caption>
7.   <thead>
8.     <tr>
9.       <th>Category</th><th>Product</th><th>Vendor</th><th>ID</th>
10.    </tr>
11.  </thead>
12.  <tbody>
13.    <tr>
14.      <td>Audio-Visual</td><td>DRY ERASE MARKERS</td>
15.      <td>Office Wholesale Supply</td><td>A-005-0100</td></tr>
16.    <tr>
17.      <td>Paper</td><td>COPY PAPER 11 X 17</td>
18.      <td>Central Supply Company</td><td>X-010-0130</td></tr>
19.    </tbody>
20.  </table>
21. </body>
22. </html>
```

Introduction

- The HTML document from the previous page is shown below as it appears in a browser.



A screenshot of a Google Chrome browser window displaying a table titled "Office Supply Products". The table has four columns: "Category", "Product", "Vendor", and "ID". The first row contains "Audio-Visual", "DRY ERASE MARKERS", "Office Wholesale Supply", and "A-005-0100". The second row contains "Paper", "COPY PAPER 11 X 17", "Central Supply Company", and "X-010-0130".

Office Supply Products			
Category	Product	Vendor	ID
Audio-Visual	DRY ERASE MARKERS	Office Wholesale Supply	A-005-0100
Paper	COPY PAPER 11 X 17	Central Supply Company	X-010-0130

- Prior the introduction of CSS, in order to apply any formatting to the document, it would have required adding multiple font tags and or attributes for the table and/or td tags.
 - ▶ This technique makes it difficult to separate the presentation from the content of the document.
 - ▶ It also requires the size of the file to increase to accommodate all of the additional tags that need to be added.
- With CSS, formatting can be applied to the above output without modifying the actual content of the document.
 - ▶ The only modification to the document needed is the addition of the following link element as child of the head element in the document.
`<link href="styling.css" type="text/css" rel="stylesheet" />`
 - While the above technique is not the only way to provide css styles to a document, it is the cleanest way of separating the presentation from the content of the document.
 - `<style></style>` tags and/or style attributes within a tag can also be used to specify CSS formatting.
- The next page takes a look at the syntax required in creating the actual style sheet "styling.css" referenced in the above tag.

CSS Syntax

- A CSS style sheet is a list of rules that specify how the content of a document should be formatted.
- A CSS rule consists of a selector and a declaration block.
 - ▶ The declaration block consists of a semicolon-separated list of declarations.
 - Each declaration has a property and a value separated by a colon.
- The general syntax for a CSS rule is as follows:

```
selector {  
    propertyA: valueA;  
    propertyB: valueB;  
    propertyC: valueC  
}
```

- ▶ The *selector* indicates which content from the source document the formatting will apply to.
 - ▶ The declaration block is a list of declarations placed inside a set of curly braces { }.
 - ▶ *propertyA: valueA;* is one of the declarations that will apply to the *selector*.
 - When there is more than one declaration in a declaration block, they are separated by a semicolon.
- A partial list of properties and associated values is shown on the following page.
 - ▶ The complete list of available property names and associated values are defined in the CSS specification.

CSS Property Names and Values

Font Properties		
Name	Sample of Possible Values	*
font-family	serif, sans-serif, cursive, fantasy, monospace, "Arial", "Verdana"	Y
font-size	12pt, 90%, larger, smaller, xx-small, x-small, small, medium, large, x-large, xx-large	Y
font-style	normal, italic, oblique	Y
font-weight	normal, bold, bolder, lighter, 100, 200, 300, 400, 500, 600, 700, 800, 900	Y
color	aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow, #00FF00, rgb(0, 255, 0), rgb(25%, 50%, 25%)	Y
background-color	<i>same as color above</i>	N
text-align	left, right, center, justify	Y
text-indent	0.5in, 2.5cm, 10%	Y
line-height	normal, .5in, 200%	Y
vertical-align	baseline, sub, super, top, text-top, middle, bottom, text-bottom	N
text-decoration	none, underline, overline, line-through	N
display	none, block, inline, list-item	N
white-space	normal, pre, nowrap	Y
border-style	none, dotted, dashed, solid, double, groove, ridge, inset, outset	N
margin	0.5in, 10%, auto	N
padding	0.5in, 10%	N

- ▶ The 3rd column (*) indicates whether the property is inherited or not.
 - Inheritance of properties by elements will be discussed later in this chapter.

Creating CSS Style Sheets

- The style sheet rules below define the formatting for certain elements in any document it is associated with.

styling.css

```
1. table {  
2.     border-style: solid;  
3.     font-size: 10pt;  
4. }  
5. thead{  
6.     font-family:"Arial";  
7.     font-size: 14pt;  
8.     background-color: #cccccc;  
9. }  
10. tbody {  
11.     font-family:"Arial";  
12.     background-color: #eeeeee;  
13. }  
14. caption {  
15.     font-weight:bold;  
16.     font-size: 18pt;  
17.     text-decoration: underline;  
18. }
```

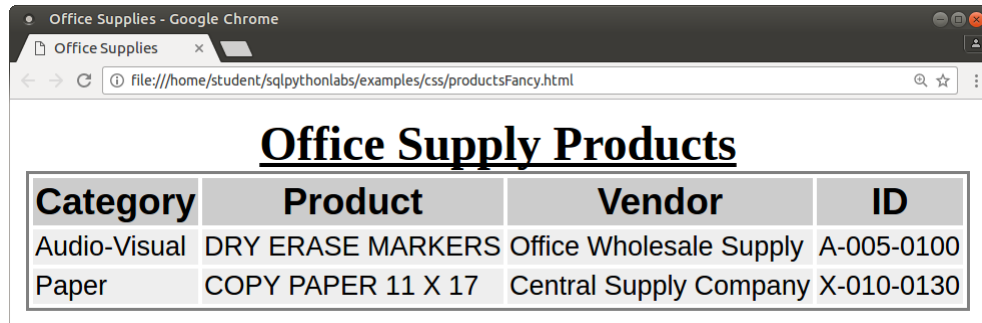
- Each selector above; table, thead, tbody, and caption is followed by a declaration block that consists of a semicolon separated list of declarations.
 - ▶ The thead selector above for example has three declarations, each consisting of the following syntax - *property:value*;
 - Multiple declarations are separated by semicolons.
- The productsFancy.html file is a rewrite of productsPlain.html with the following line added to the head element.

```
<link href="styling.css" type="text/css" rel="stylesheet" />
```

 - ▶ The images on the following page show the formatting applied by the style sheet to the productsFancy.html document.

Creating CSS Style Sheets

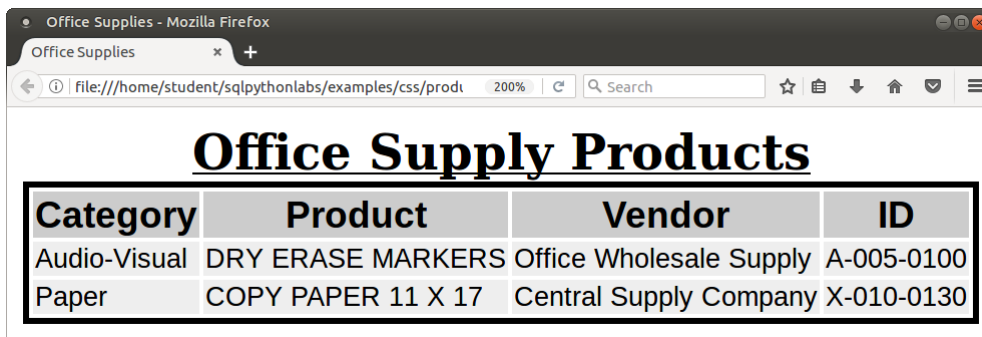
- productsFancy.html as displayed in Google Chrome



A screenshot of a Google Chrome browser window. The title bar says "Office Supplies - Google Chrome". The address bar shows the file path: file:///home/student/sqlpythonlabs/examples/css/productsFancy.html. The page content features a heading "Office Supply Products" in a large, bold, serif font. Below the heading is a table with four columns: Category, Product, Vendor, and ID. The table has two data rows. The first row shows "Audio-Visual", "DRY ERASE MARKERS", "Office Wholesale Supply", and "A-005-0100". The second row shows "Paper", "COPY PAPER 11 X 17", "Central Supply Company", and "X-010-0130". The table has a thin border.

Category	Product	Vendor	ID
Audio-Visual	DRY ERASE MARKERS	Office Wholesale Supply	A-005-0100
Paper	COPY PAPER 11 X 17	Central Supply Company	X-010-0130

- productsFancy.html as displayed in Mozilla Firefox



A screenshot of a Mozilla Firefox browser window. The title bar says "Office Supplies - Mozilla Firefox". The address bar shows the file path: file:///home/student/sqlpythonlabs/examples/css/prodi. The page content is identical to the Chrome screenshot, featuring the heading "Office Supply Products" and the same table with two data rows. However, the table in this screenshot has a thicker border.

Category	Product	Vendor	ID
Audio-Visual	DRY ERASE MARKERS	Office Wholesale Supply	A-005-0100
Paper	COPY PAPER 11 X 17	Central Supply Company	X-010-0130

- Note the subtle differences in the views above:
 - ▶ Not all CSS tags are necessarily supported by all browsers.
 - ▶ Of those supported, they are not necessarily implemented in the same way.
 - For instance, notice the differences in border around the table.

Grouping Selectors

- When several selectors share the same declarations, they may be grouped into a comma-separated list.

- ▶ Consider the two declarations from the preceding style sheet shown below.

```
thead{
    font-family:"Nimbus Sans L", "Arial";
    font-size: 14pt;
    background-color: #cccccc;
}
tbody {
    font-family:"Arial";
    background-color: #eeeeee;
}
```

- Both selectors share the font-family: "Arial" declaration and therefore can be grouped into a comma-separated list as follows.

```
thead, tbody {
    font-family:"Arial";
}
thead{
    font-size: 14pt;
    background-color: #cccccc;
}
tbody {
    background-color: #eeeeee;
}
```

Chapter 18:

Python and WebServers

Introduction

- The Python Standard Library includes an `http.server` module that defines classes for implementing HTTP servers in Python.
 - ▶ The `http.server.HTTPServer` class listens for and dispatches HTTP requests to a configured handler.
 - ▶ The `http.server.CGIHTTPRequestHandler` class handles either files or output of CGI scripts.
- The Common Gateway Interface (CGI) is a standard that defines how external programs interface with information servers such as a web server.
 - ▶ CGI allows web servers to execute programs (often referred to as scripts) and incorporate their output into the response sent to the client.
 - ▶ A CGI script is executed in real-time, providing the capability of a dynamic response being sent to the client.
 - This includes, but is not limited to languages such as; Python, Perl, C, and Unix/Linux shells.
- The entire CGI process typically involves a programming language such as Python, HTML pages and responses, a web server and a browser.
 - ▶ The Python program, acting as the CGI script, is generally executed when it is called from a web page that is typically written using HTML.
 - The HTML page will typically incorporate the use of Cascading Style Sheets (CSS) and possibly some JavaScript that is executed on the clients computer, as opposed to the CGI Script that is executed on the server.
 - The `action` attribute of a form element on an HTML page usually has a link that when submitted executes the CGI Script on the server.

HTTPServer

- As mentioned, the `http.server.HTTPServer` class listens for and dispatches HTTP requests to a configured handler.
 - ▶ The examples in this chapter will rely on this class as the HTTP Server that will be used to serve up both static HTML pages and Dynamic HTTP responses from the CGI script being executed.
 - ▶ In order to handle the CGI requests, the server will rely on another built-in data type, `http.server.CGIHTTPRequestHandler`.
- The code below will be used as the HTTP server for all of the examples in this chapter.

server.py

```
1. #!/usr/bin/env python3
2. import inspect, os, sys
3. from http.server import HTTPServer
4. from http.server import CGIHTTPRequestHandler as CGIHandler
5.
6. def display_server_info(port):
7.     main_module_dirname = os.path.dirname(os.path.abspath(sys.argv[0]))
8.     print("Server's root directory:", main_module_dirname)
9.     print("Server's CGI directories:", CGIHandler.cgi_directories)
10.    print("Starting HTTP Server on port:", port)
11.
12. def configure_server():
13.     port = 8000 + os.getuid()
14.     return HTTPServer(('localhost', port), CGIHandler)
15.
16. def main():
17.     try:
18.         server = configure_server()
19.     except Exception as e:
20.         print("Unable to start server:", e)
21.         return
22.     try:
23.         display_server_info(server.server_port)
24.         server.serve_forever()
25.     except BaseException as base:
26.         print()
27.         print("Exception:", base, "Shutting Down Server", sep="\n")
28.         server.shutdown()
29.
30. if __name__ == "__main__":
31.     main()
```

- ▶ The above application starts the server as shown on the next page.

HTTPServer

```
student@localhost:~/sqlpythonlabs/examples/webserver
$ python3 server.py
Server's root directory: /home/student/sqlpythonlabs/examples/webserver
Server's CGI directories: ['/cgi-bin', '/htbin']
Starting HTTP Server on port: 8000
$
```

- ▶ The actual root directory path and port # will vary by student.
- Accessing the server can be done by supplying the following URL to a browser: `http://localhost:8000`
 - ▶ The port# `8000` above will need to be replaced with the actual port number shown in the output of your server above.



- The above page is served up by default from a file named `index.html` located in the root directory of the server.
 - ▶ The server's root directory is:
 - `~/sqlpythonlabs/examples/webserver`
 - Any files within this directory are publicly available to anyone having access to the web server.
 - The only exception to this is the `cgi-bin` directory whose contents are only ever executed by the server as opposed to offered up as a file by the server.

CGI Scripts

- A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML form element.
 - ▶ The HTTP server places information about the incoming request in the script's shell environment, executes the script, and sends the script's output back to the client.
 - ▶ The output of a CGI script, the response, should consist of two sections, separated by a blank line.
 - The first section contains a number of headers.
 - The second section (the body of the response) is usually HTML.
- The following example demonstrates how the CGI Script has access to the script's shell environment through the `os.environ` dictionary like object.
 - ▶ Accessing the script through a web browser is shown on the following page.

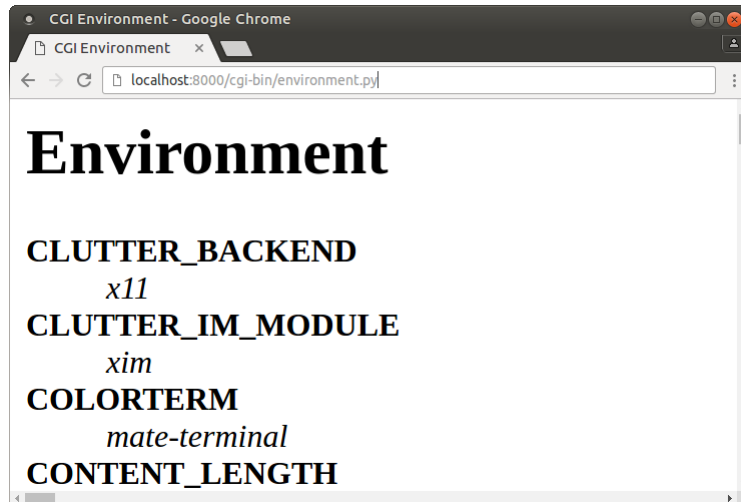
environment.py

```
1. #!/usr/bin/env python3
2. import cgi
3. import os
4.
5. # This is the Head of the HTTP response
6. print("Content-type: text/html\n")
7.
8. #This begins the Body of the HTTP Response
9. print("<html><head><title>CGI Environment</title></head>")
10. print("<body>")
11. print("<h1>Environment</h1>")
12. print("<dl>")
13. keys = list(os.environ.keys())
14. keys.sort()
15. for key in keys:
16.     print("<dt><strong>", key, "</strong></dt>", sep="")
17.     print("<dd><em>", os.environ[key], "</em></dd>", sep="")
18. print("</dl>")
19. print("</body></html>")
```

- Note that all print statements in the above script are automatically routed back to the client as a stream of bytes by the server.

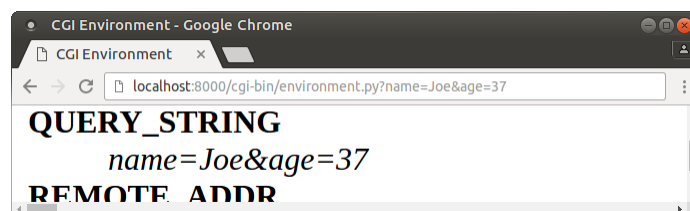
CGI Scripts

- The script from the previous page is access using the following URL:
 - ▶ `http://localhost:8000/cgi-bin/environment.py`
 - Once again, as throughout the chapter, the actual port number to use in the URL above will vary by student.



- ▶ Two of the environment properties that will be of the most use are QUERY_STRING and HTTP_COOKIE properties that are commonly used in CGI scripts.
 - This chapter includes several examples where the use of these two properties will be used within the CGI scripts.
 - For now, note how the following modification to the previous URL will result in a value being supplied for the QUERY_STRING property.

`localhost:8000/cgi-bin/environment.py?name=Joe&age=37`



HTML Forms and CGI Scripts

- Typically it is an HTML `<form>` element's `action` attribute and a corresponding `<input type="submit">` element that sends a request to the server that results in the running of a CGI Script on the server side.
 - ▶ The example below is an simple form in an HTML page consisting of a simple submit button.
 - The `action` attribute of the `form` element contains the URL to the script to run when the submit button is clicked.

basic_form.html

```
1. <!doctype html>
2. <html>
3. <head><title>A Very Basic Form</title></head>
4. <body>
5.     <form action="/cgi-bin/basic_form_response.py">
6.         <input type="submit">
7.     </form>
8. </body>
9. </html>
```

- ▶ The associated script is shown below.

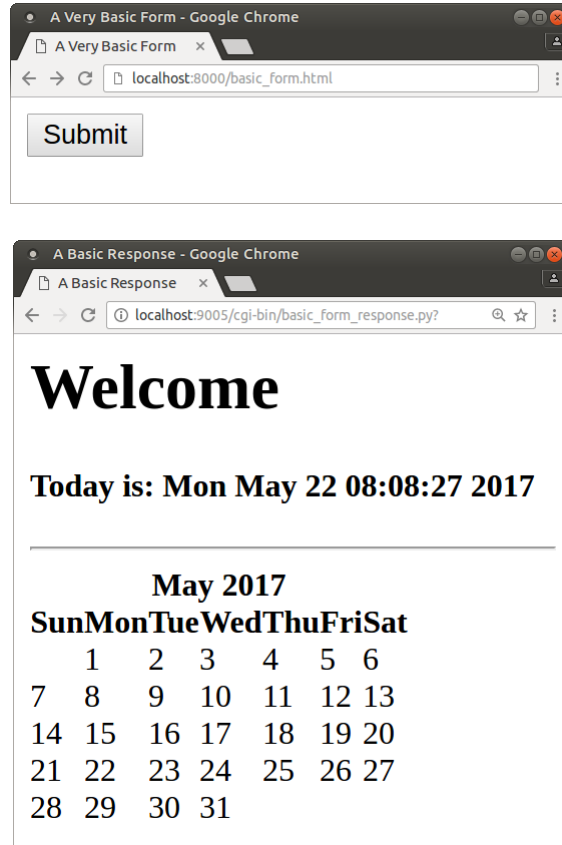
basic_form_response.py

```
1. #!/usr/bin/env python3
2. import cgi, calendar, time
3.
4. # This is the Head of the HTTP response
5. print("Content-type: text/html\n")
6.
7. #This begins the Body of hte HTTP Response
8. print("<html><head><title>A Basic Response</title></head>")
9. print("<body>")
10. print("<h1>Welcome</h1>")
11. print("<h4>Today is: ", time.ctime(), "</h4>")
12. print("<hr>")
13. cal = calendar.HTMLCalendar(firstweekday = calendar.SUNDAY)
14. print(cal.formatmonth(2017, 5, withyear=True))
15. print("</body></html>")
```

- Both the HTML form and the HTML response generated by the script are shown on the following page.

HTML Forms and CGI Scripts

- The images below show the HTML page and resulting HTML response from the code on the previous page.



- The previous example has demonstrated how a CGI Script can be used to provide a dynamic HTTP response to a request made by a client.
 - ▶ Another advantage that CGI brings is the ability to obtain input from the client via the data entered in an HTML form.
 - The next example designs a `guestbook.html` page that has a form to collect information from the user that for now will simply be used in the response.

HTML Forms and CGI Scripts

guestbook.html

```

1. <!doctype html>
2. <html>
3. <head>
4.     <title>Guest Book</title>
5.     <style>
6.         .txt { width:20em; }
7.     </style>
8. </head>
9. <body>
10.    <h3>Welcome to my Guestbook</h3>
11.    <p>Please fill in the following form</p>
12.    <form method="get" action="/cgi-bin/guestbook.py">
13.        <div>First Name:</div>
14.        <div><input class="txt" type="text" name="first_name" /></div>
15.        <div>Last Name:</div>
16.        <div><input class="txt" type="text" name="last_name" /></div>
17.        <div>Comments:</div>
18.        <textarea class="txt" name="comments"></textarea>
19.        <div><input type="submit" /> <input type="reset"/></div>
20.    </form>
21. </body>
22. </html>

```

- ▶ The above document is rendered in a browser as shown below.
 - The image on the left is as it first loads.
 - The one on the right has been filled out prior to hitting submit.

Guest Book - Google Chrome

Guest Book

localhost:8000/guestbook.html

Welcome to my Guestbook

Please fill in the following form

First Name:

Last Name:

Comments:

Submit Reset

Guest Book - Google Chrome

Guest Book

localhost:8000/guestbook.html

Welcome to my Guestbook

Please fill in the following form

First Name:

Sally

Last Name:

Morgan

Comments:

My first time here

Submit Reset

- ▶ The Python CGI script referenced in the action attribute is shown on the following page.

HTML Forms and CGI Scripts

guestbook.py

```
1. #!/usr/bin/env python3
2. import cgi
3. fields = cgi.FieldStorage()
4. default = ""
5. first_name = fields.getvalue("first_name", default)
6. last_name = fields.getvalue("last_name", default)
7. comments = fields.getvalue("comments", default)
8. # This is the Head of the HTTP response
9. print("Content-type: text/html\n")
10.
11. #This begins the Body of the HTTP Response
12. print("<html><head><title>Thank You</title></head>")
13. print("<body>")
14. print("<h1>Thank you", first_name, last_name, "</h1>")
15. print("Your comments below are greatly appreciated<br>")
16. print("<em>", comments, "</em>")
17. print("</body></html>")
```

- ▶ The rendered output of the above script based on the user input from the following page is shown below.



- The python code above could have used the `os.environ["QUERY_STRING"]` to obtain the query string and manually parse it to be used in the output.
 - ▶ Instead, it uses a higher level interface to the form data through the use of the `FieldStorage` class.
 - The first argument to each of the `getValue()` method calls is the name of one of the form elements in the `.html` document.
 - If the form field key does not exist, the optional second argument to the `getValue()` method is used instead.

CGI Scripts and Cookies

- The `http.cookies` module defines classes for abstracting the concept of cookies.
 - ▶ Cookies are an HTTP state management mechanism, since the HTTP protocol itself is a stateless protocol.
 - ▶ Cookies basically contain key/value pairs that a web server and a client can pass back and forth to maintain state across multiple requests made by the browser.
- The `http.cookies.SimpleCookie` class can be used to store cookie information as a dictionary-like object.
 - ▶ The keys are strings, and the values are `http.cookies.Morsel` objects.
 - ▶ The `Morsel` object is a dictionary-like object itself which holds key value pairs describing various attributes of a cookie.
 - ▶ The keys for a `Morsel` are one of the following:
 - `expires` • `path` • `comment` • `domain`
 - `max-age` • `secure` • `version` • `httponly`
- The example on the next page uses a cookie to track the number of times the user has visited the page.
 - ▶ It also set an expiration date for the cookie of 30 days from the current date.
 - Keep in mind this is reset to 30 days each time the page is visited.
 - ▶ In order for the whole site to use the customizations stored in the cookies, it would require that the entire website be dynamically driven with no static `.html` files.

CGI Scripts and Cookies

cookies.py

```
1. #!/usr/bin/env python3
2.
3. import cgi
4. import os
5. import datetime
6. from http.cookies import SimpleCookie, Morsel
7.
8. cookie_ = SimpleCookie(os.environ["HTTP_COOKIE"])
9. expiration = datetime.datetime.now() + datetime.timedelta(days=30)
10. morsel = Morsel()
11. morsel.key="visits"
12. morsel.value="0"
13. visits = int(cookie_.get("visits", morsel).value) + 1
14. cookie_["visits"] = str(visits)
15. date_format = "%a, %d-%b-%Y %H:%M:%S %Z"
16. cookie_["visits"]["expires"] = expiration.strftime(date_format)
17.
18.
19. print("Content-type: text/html")
20. print(cookie_.output(), "\n")
21. print("<html><head><title>Cookies</title></head>")
22. print("<body>")
23. print("<h1>Cookie Example</h1>")
24. print("<b>The Following header was sent:<br />",
25.       cookie_.output(), "</b><br/>")
26. print("<div><h3># of visits: ", visits, "</h3></div><hr />")
27. print("</body></html>")
```



Chapter 19:

JavaScript

The Core JavaScript Language

- JavaScript is a scripting language developed by Netscape in 1995.
 - ▶ JavaScript is not a programming language that can stand on its own.
 - It needs a JavaScript capable browser that can interpret and execute the JavaScript code.
- As the JavaScript language evolved, Microsoft and Netscape started adding proprietary extensions to the language.
 - ▶ The European Computer Manufacturers Association developed a standard called ECMAScript-262 to provide a standard for vendors to follow when implementing JavaScript.
- The JavaScript language is divided into two basic components.
 - ▶ The **core** part of the language consists primarily of the following components:
 - Language structure
 - Operators
 - Data types
 - Statements
 - Variables
 - Functions
 - Expressions
 - ▶ The **object-oriented** part of the language consists of more complex things that have properties, methods and events associated with them.

A Simple JavaScript Example

- Below is a simple example of using JavaScript within HTML.

HelloWorld1.html

```
1. <!doctype html>
2. <html>
3.   <head><title>Hello World - JavaScript Example</title></head>
4.   <body>
5.     <script type="text/javascript">
6.       document.write("<h1>Hello World</h1>");
7.     </script>
8.   </body>
9. </html>
```

- ▶ When the `document.write()` method is called, whatever string is placed within the parentheses becomes part of the output.
 - This will be explained in more detail in the chapter that discusses the object-oriented components of JavaScript.
- ▶ One issue with JavaScript is how browsers behave when JavaScript is turned off in the browser.
 - The possible complications posed by this scenario is best handled by always placing the actual JavaScript code in an external file (as demonstrated below).

- The next example is a rewrite of the one above.

- ▶ This example places the actual JavaScript code in an external file, and references the file with the `src` attribute in the `script` tag where it is to be used.

HelloWorld2.js

```
1. document.write("<h1>Hello World</h1>");
```

HelloWorld2.html

```
1. <!doctype html>
2. <html>
3.   <head><title>Hello World 2 - JavaScript Example</title></head>
4.   <body>
5.     <script type="text/javascript" src="HelloWorld2.js"></script>
6.   </body>
7. </html>
```


Language Structure

- The JavaScript language is case sensitive.
 - ▶ Below are examples that address the issue of case sensitivity.
 - Function cannot be used in place of the correct usage of the keyword function.
 - A variable named amount and one named Amount are two completely different variables.
 - A function named compute () and one named Compute () are two completely different functions.
- Whitespace is ignored except when used inside a literal string.
 - ▶ Whitespace is often used to make code more readable by indenting or nesting lines of code.
- Comments are represented using either of the following forms.

```
// This is a single line JavaScript comment
/* This represents the syntax used to comment
   lines that span more than one line in the
   JavaScript code
*/
```
- Semicolons are used to separate statements.
 - ▶ Semicolons are optional when there is only one statement per line in the code, but it is a good programming practice to always use them.
- Literals, as opposed to variables, are data values that appear directly in a program.
 - ▶ Some examples of literals follow.
 - 12
 - 1.23
 - "Susan"
 - 'Hi'
 - true
 - false
 - null

Language Structure

- Identifiers are used as names of variables, functions and statement labels.
 - ▶ The first character of an identifier must be a letter of the alphabet, an underscore, or a dollar sign.
 - ▶ Subsequent characters may be any combination of letters, digits, underscores, and/or dollar signs.
- Reserved words are those in the language that have a special meaning within the language.
 - ▶ It is illegal to use a reserved word as the name of an identifier.

break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	try	
do	instanceof	typeof	

- ▶ The following keywords are reserved for future use within the JavaScript language.

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

- A variable is a name associated with a data value, allowing for the storage and manipulation of the data in a program.
 - ▶ JavaScript is an dynamically typed programming language allowing a variable to hold a value of any data type.
 - It is perfectly legal to store a number in a variable and later overwrite it with a string.

Language Structure

- Before a variable is used in JavaScript, it should always be declared using the var keyword.
 - ▶ Although the use of var is optional, it is highly recommended that all variables be declared in order to head off potential confusion within an application as to the scope of the variable.

- Below are several examples of declaring variables.

```
var i;           // declare a single variable i
var sum;         // declare a single variable sum
var i, sum;      // declare multiple variables
var i=0;         // declare and initialize a variable i

// declare and initialize multiple variables
var i=0, j=5, k=10;
```

- ▶ Variables are not automatically initialized to a default value.
 - Variables that are declared but not initialized have a value of undefined until a value is stored in it.
- Javascript has two fundamental data types.
 - ▶ **Primitive** data types such as numbers, strings, and Booleans
 - ▶ **Reference** data types reference a collection of primitives and possibly other references represented as an object.
- All numbers in JavaScript are stored as floating point values.
 - ▶ For the most part, there is no distinction within the language between integer and floating point values.
 - Floating point numbers: var a = .10, b = -123.456;
 - Integers: var a = 0, b = -10, c = 10;
 - Octal values begin with the digit zero: var a = 0777;
 - Hexadecimal values begin with the digit zero followed by the letter x: var a = 0xFF, b = 0x9A;

Strings

- A string is a sequence of zero or more characters enclosed in a matching pair of either single or double quotes.
 - Strings in JavaScript are primitive data types that rely on the object syntax for accessing methods and properties.
 - The + operator acts as a concatenation operator when used in conjunction with strings.
- In addition to a `length` property, there are many methods to operate on string values.
 - These methods fall into one of two sets:
 - A set that wraps the string in a variety of HTML tags.
 - A set that allows generalized string manipulation.
- Some of the methods that allow generalized string manipulation are shown in the table below.

String Methods	
Method Name	Description
<code>charAt</code>	Returns the character at the specified index.
<code>indexOf</code>	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
<code>match</code>	Used to match a regular expression against a string.
<code>replace</code>	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
<code>split</code>	Splits a String object into an array of strings by separating the string into substrings.
<code>substr</code>	Returns the characters in a string beginning at the specified location through the specified number of characters.
<code>substring</code>	Returns the characters in a string between two indexes into the string.

- The example on the next page demonstrates several of the above methods.

Strings

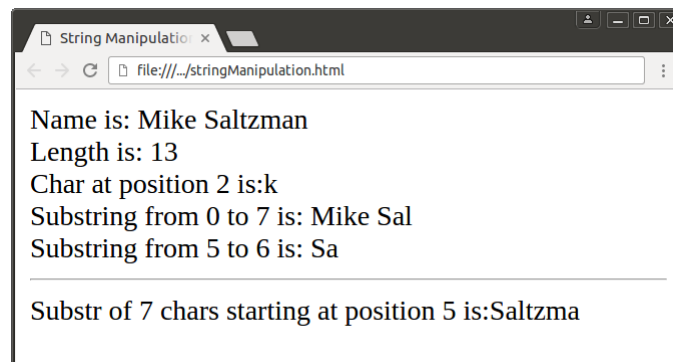
StringManipulation.js

```
1. var firstName = "Mike", lastName = "Saltzman";
2. var theName = firstName.concat(" ", lastName);
3. var br = "<br />";
4. var output = "Name is: " + theName + br
5.   + "Length is: " + theName.length + br
6.   + "Char at position 2 is:" + theName.charAt(2) + br
7.   + "Substring from 0 to 7 is: "
8.   + theName.substring(0,8) + br
9.   + "Substring from 5 to 6 is: "
10.  + theName.substring(5,7) + "<hr />"
11.  + "Substr of 7 chars starting at position 5 is:"
12.  + theName.substr(5,7);
13. document.write(output);
```

StringManipulation.html

```
1. <!doctype html>
2. <html>
3.   <head><title>String Manipulation</title></head>
4.   <body>
5.     <script type="text/javascript" src="stringManipulation.js"></script>
6.   </body>
7. </html>
```

- The output from the above script is shown below.



Arithmetic Operators

- The JavaScript operators can be grouped into three categories.
 - ▶ Arithmetic Operators
 - ▶ Comparison Operators
 - ▶ Logical Operators
- The following are some of the commonly used Arithmetic operators.

Arithmetic Operators		
Operator		Description
+	Addition	Adds numeric operands or concatenates string operands.
-	Subtraction	Subtracts the second operand from the first.
*	Multiplication	Multiplies its two operands.
/	Division	Divides its first operand by its second.
%	Modulo	Returns the remainder when the first operand is divided by the second operand.
++	Increment	Adds 1 to the operand.
--	Decrement	Subtracts 1 from the operand.

- ▶ When used with non-numeric operands, these operators attempt to convert the operands to numbers to perform the necessary calculation.
- ▶ The behavior of the ++ and -- operators depend on whether they are before or after the operand.
 - The value of the operand will be incremented or decremented in both instances but evaluates to a different value based on its position.
 - For example:

```
i = 1; j = ++i;    // i and j equal to 2
i = 1; j = i++;    // i equal to 2 but j equal to 1
```

Comparison and Logical Operators

- Comparison and Logical operators evaluate to a Boolean value.

Comparison Operators		
Operator		Description
==	Equality	true if the values are equal, false otherwise.
!=	Inequality	true if the values are not equal, false otherwise.
<	Less than	true if the left operand is less than the right, false otherwise.
>	Greater Than	true if the left operand is greater than the right, false otherwise.
<=	Less than or equal to	true if the left operand is less than or equal to the right, false otherwise.
>=	Greater than or equal to	true if the left operand is greater than or equal to the right, false otherwise.

Logical Operators		
Operator		Description
&&	AND	true && true evaluates to true.
		true && false evaluates to false.
		false && true evaluates to false.
		false && false evaluates to false.
	OR	true true evaluates to true.
		true false evaluates to true.
		false true evaluates to true.
		false false evaluates to false.
!	NOT	!true evaluates to false.
		!false evaluates to true.

- ▶ Logical operators are commonly used to create more complex comparisons.

The if Statement

- The if statement allows JavaScript to make a decision or to conditionally execute a statement.
 - ▶ The general syntax for an if else statement is as follows (where the else is optional):

```
if (expression) {  
    statement1;  
}  
else {  
    statement2;  
}
```

ifExamples.js

```
1. var x = 3, y = 10;  
2.  
3. // if statement 1  
4. document.write("<hr />Output of statement 1: ");  
5. if(x == 3)  
6.     document.writeln("<em>x equals 3</em>");  
7.  
8. // if statement 2  
9. document.write("<hr />Output of statement 2: ");  
10. document.write("<em>");  
11. if(x > y){  
12.     document.write("x is greater than y");  
13. } else if (x < y){  
14.     document.write("x is less than y");  
15. } else{  
16.     document.write("x is equal to y");  
17. }  
18. document.writeln("</em>");
```

- ▶ The JavaScript file shown above is referenced in the file named `ifExamples.html`.

The switch Statement

- JavaScript also has a `switch` statement that may be used rather than a set of `if else` statements.
 - ▶ It is often easier to use when testing for various values of a single variable.
 - The `case` clauses define where execution begins inside the `switch` for a given value of `n`.
 - The `break` statement is used to cause execution to jump to the end of the `switch`. The `break` statement is optional, if the desired affect is to "fall through" the `switch` executing several statements.
 - The `default` clause is also optional. When omitted, only those cases tested for will have code executed. This similar to omitting the `else` clause at the end of several `else ifs`.

switchExample.js

```
1. var s = "Please enter a day\nof the week";
2. var choice = window.prompt(s, "");
3. var choiceLax = choice.toLowerCase();
4. var response = choice + " is ";
5. switch(choiceLax) {
6.     case "monday":
7.         response += "the first day of the week and ";
8.     case "tuesday":
9.         response += "in the beginning of the week";
10.        break;
11.    case "wednesday":
12.        response += "the middle of the week";
13.        break;
14.    case "thursday":
15.    case "friday":
16.        response += "towards the end of the week";
17.        break;
18.    default:
19.        response += "either the weekend or is invalid";
20. }
21. document.write("<h1>" + response + "</h1>");
22. document.write("<h3>Refresh to try new value</h3>");
```

- ▶ The JavaScript file shown above is referenced in the file named `switchExamples.html`.

The for Statement

- The for statement is a looping construct for repetitive execution of statements.

- ▶ The general syntax of a for loop is as follows:

```
for(initialization; test; increment) {  
    statement(s);  
}
```

- The example below demonstrates the use of a for loop .

forLoop.js

```
1. var i = 1, sum = 0;  
2. while( i <= 5) {  
3.     document.write("i = " + i);  
4.     sum += i;  
5.     document.write(" sum = " + sum);  
6.     document.write("<br />");  
7.     i = i + 1;  
8. }  
9. document.write("Total is " + sum + "<br />");  
10. document.write(" Value of i is " + i);
```

- ▶ The JavaScript file shown above is referenced in the file named `forLoop.html`.

- Nesting of loops can add additional functionality to a program.

nestedForLoop.js

```
1. var numRows=5, numCols=10, cnt=1, row, col;  
2. document.write("<table border = '1'>");  
3. for(var row = 0; row < numRows; row++){  
4.     document.write("<tr>");  
5.     for(var col = 0; col < numCols; col++){  
6.         document.write("<td>" + cnt++ + "</td>");  
7.     } //end of inner for loop  
8.     document.write("</tr>");  
9. } //end of outer for loop  
10. document.write("</table>");
```

- ▶ The JavaScript file shown above is referenced in the file named `nestedForLoop.html`.

Arrays

- An array represents an ordered collection of data, any element of which can be retrieved by number called its index.
 - ▶ The number of elements in an array can be changed at any time.
 - ▶ A value is retrieved from an array by enclosing the index inside a set of square brackets.
 - ▶ An array in JavaScript may contain any data type or combination of data types.
- Since an array is an object (reference data type), creating one requires use of the new operator and a special function called a constructor.
 - ▶ The Array constructor can be invoked in several different ways.
 - `var a = new Array();`
 - `var a = new Array(5, 10, 15, 20);`
 - `var a = new Array(10);`
 - `var a = ["Mon", "Tue", "Wed", "Thu", "Fri"];`
 - ▶ Once an array is created, its `length` property can be used to determine the number of elements in the array.
- Arrays are typically processed with for loops to step through the indices of the array.
 - ▶ The example on the following page demonstrates various techniques for processing arrays.

Arrays

simpleArrays.js

```
1. var longNames = new Array(7);
2. longNames[0] = "Sunday";
3. longNames[1] = "Monday";
4. longNames[2] = "Tuesday";
5. longNames[3] = "Wednesday";
6. longNames[4] = "Thursday";
7. longNames[5] = "Friday";
8. longNames[6] = "Saturday";
9.
10. var shortNames = new Array("Sun", "Mon", "Tue",
11.    "Wed", "Thu", "Fri", "Sat");
12.
13. var monthNames = ["Jan", "Feb", "Mar", "Apr", "May",
14.    "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
15.
16. var days = "31|28|31|30|31|30|31|31|30|31|30|31";
17. var monthDays = days.split("|");
18.
19. var i;
20. document.writeln("<h1>Days of the week:</h1>");
21. for(i = 0; i < longNames.length; i++){
22.    document.write(longNames[i] + "&nbsp;");
23. }
24.
25. document.writeln("<hr />");
26. for(i = 0; i < shortNames.length; i++){
27.    document.write(shortNames[i] + "&nbsp;");
28. }
29. document.writeln("<hr />");
30. document.writeln("<h1>Months of the year:</h1>");
31. for(i = 0; i < monthNames.length; i++){
32.    document.write(monthNames[i] + " has ");
33.    document.writeln(monthDays[i] + " days<br />");
34. }
```

- ▶ The JavaScript file shown above is referenced in the file named `simpleArrays.html`.
- In addition to the `length` property of an array, there are many methods defined for an `Array` object.
 - ▶ Everything about JavaScript arrays can be found within the specification at the following URL:
ecma-international.org/ecma-262/8.0/index.html#sec-array-objects

Associative Arrays

- Associative arrays (sometimes referred to as dictionaries) are arrays whose element indices are strings rather than numbers.
 - ▶ An associative array can be thought of as a collection of key/value pairs, where the key is unique within the array.
 - Each key acts as an indices, and each is associated with a value.
- The following example demonstrates the use of associative arrays to deal with states and their capitals.

statesAndCapitals.js

```
1. var states = {  
2.     "Alabama" : "Montgomery",  
3.     "Alaska" : "Juneau",  
4.     ...  
5.     "Wisconsin" : "Madison",  
6.     "Wyoming" : "Cheyenne"  
7. };
```

- ▶ The above data is a snippet of the full 50 states and capitals defined in the actual file.

associativeArray.js

```
1. var currentState = "Maryland";  
2. document.write("The capital of " + currentState +  
3.     " is " + states[currentState] + "<br />");  
4. document.write("<table border='1'>");  
5. document.write("<caption>States and Capitals</caption>");  
6. document.write("<thead><tr>");  
7. document.write("<th class='key'>State</th>");  
8. document.write("<th class='val'>Capital</th>");  
9. document.write("<th class='key'>State</th>");  
10. document.write("<th class='val'>Capital</th>");  
11. document.writeln("</tr></thead><tbody>");  
12. var column = 0;  
13. for (s in states){  
14.     if(column % 2 == 0)  
15.         document.writeln("<tr>");  
16.         document.write("<td class='key'>" + s + "</td>");  
17.         document.writeln("<td class='val'>" + states[s] + "</td>");  
18.         if(column % 2 == 1)  
19.             document.writeln("</tr>");  
20.         column++;  
21. }  
22. document.writeln("</tbody></table>");
```

Associative Arrays

- The JavaScript files shown on the previous page, along with a style sheet named `associativeArray.css`, are referenced in the file named `associativeArray.html`.
 - ▶ The output of `associativeArray.html` is shown below.

The capital of Maryland is Annapolis

States and Capitals

State	Capital	State	Capital
Alabama	Montgomery	Alaska	Juneau
Arizona	Phoenix	Arkansas	Little Rock
California	Sacramento	Colorado	Denver
Connecticut	Hartford	Delaware	Dover
Florida	Tallahassee	Georgia	Atlanta
Hawaii	Honolulu	Idaho	Boise
Illinois	Springfield	Indiana	Indianapolis
Iowa	Des Moines	Kansas	Topeka
Kentucky	Frankfort	Louisiana	Baton Rouge
Maine	Augusta	Maryland	Annapolis
Massachusetts	Boston	Michigan	Lansing
Minnesota	Saint Paul	Mississippi	Jackson
Missouri	Jefferson City	Montana	Helena
Nebraska	Lincoln	Nevada	Carson City
New Hampshire	Concord	New Jersey	Trenton
New Mexico	Santa Fe	New York	Albany
North Carolina	Raleigh	North Dakota	Bismarck
Ohio	Columbus	Oklahoma	Oklahoma City
Oregon	Salem	Pennsylvania	Harrisburg
Rhode Island	Providence	South Carolina	Columbia
South Dakota	Pierre	Tennessee	Nashville
Texas	Austin	Utah	Salt Lake City
Vermont	Montpelier	Virginia	Richmond
Washington	Olympia	West Virginia	Charleston
Wisconsin	Madison	Wyoming	Cheyenne

Functions

- In order to create a user defined function, the `function` keyword is used followed by:
 - ▶ The name of the function followed by an optional list of comma-separated parameter names inside a set of parenthesis; and
 - ▶ The body of the function, consisting of statements inside curly braces.
 - The body may contain a `return` statement that returns a value to the part of the script that called the function.
- Below are some examples of user-defined functions.
 - ▶ Note that the functions themselves are defined in one file, and a reference to it is placed in the head element of the HTML file.
 - ▶ The code that calls the functions is in a separate file and is placed in the body element of the HTML where it is needed.

userFunctionsDefined.js

```
1. // function that takes a single parameter and appends
2. // an XHTML break tag to it as its output.
3. function println(text) {
4.     document.write(text + "<br />");
5. }
6.
7. // function that takes a single parameter and returns
8. // a string of that many non-breaking spaces.
9. function padding(padWidth){
10.     var pad = "";
11.     for(var i = 0; i < padWidth; i++)
12.         pad += "&nbsp;";
13.     return pad;
14. }
```

callFunctions.js

```
1. var days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
2. for(var i=0; i < days.length; i++)
3.     println(days[i]);
4.
5. for(i = 0; i < days.length; i++)
6.     println(padding(25) + days[i]);
```

- ▶ The JavaScript files shown above are referenced in the file named `callFunctions.html`.

Core JavaScript Functions

- JavaScript provides several core (global) functions to simplify some common programming tasks.
 - ▶ Several of the available functions deal with converting and working with numbers and are demonstrated in the following example.
- The display below is from `stringsToNumbers.html`.

Converting Strings to Numbers - Google Chrome

Converting Strings x

file:///home/student/sqlpythonlabs/examples/javascript/stringsToNumbers.html

Converting Numbers

String str	parseFloat (str)	parseInt (str)	parseInt (str, 10)	parseInt (str, 2)	parseInt (str, 16)	isFinite (str)	isNaN (str)	Number (str)
"5"	5	5	5	NaN	5	true	false	5
"4.00"	4	4	4	NaN	4	true	false	4
"3.09"	3.09	3	3	NaN	3	true	false	3.09
"02"	2	2	2	0	2	true	false	2
"6,789"	6	6	6	NaN	6	false	true	NaN
"10"	10	10	10	2	16	true	false	10
"101"	101	101	101	5	257	true	false	101
"010"	10	10	10	2	16	true	false	10
"3A"	3	3	3	NaN	58	false	true	NaN
"A4"	NaN	NaN	NaN	NaN	164	false	true	NaN

- ▶ The actual JavaScript code is on the following page below.

Core JavaScript Functions

stringsToNumbers.js

```
1. var nums = ["5","4.00","3.09","02","6,789",
2.           "10","101","010","3A","A4"];
3. function asRow(text) {
4.     document.writeln("<td>" + text + "</td>");
5. }
6. function asRowHeader(text){
7.     document.writeln("<th>" + text + "</th>");
8. }
9.
10. document.writeln("<table><caption>Converting Numbers"
11.                  + "</caption>");
12.
13. document.writeln("<tr>");
14. asRowHeader("String str");
15. asRowHeader("parseFloat (str)");
16. asRowHeader("parseInt (str)");
17. asRowHeader("parseInt (str, 10)");
18. asRowHeader("parseInt (str, 2)");
19. asRowHeader("parseInt (str, 16)");asRowHeader("isFinite (str)");
20. asRowHeader("isNaN (str)");
21. asRowHeader("Number (str)");
22.
23. document.writeln("</tr>");
24. for(var i = 0; i < nums.length; i++) {
25.     document.writeln("<tr>");
26.     asRow("<strong>\\"" + nums[i] + "\"</strong>");
27.     asRow(parseFloat(nums[i]));
28.     asRow(parseInt(nums[i]));
29.     asRow(parseInt(nums[i], 10));
30.     asRow(parseInt(nums[i], 2));
31.     asRow(parseInt(nums[i], 16));
32.     asRow(isFinite(nums[i]));
33.     asRow(isNaN(nums[i]));
34.     asRow(Number(nums[i]));
35.     document.writeln("</tr>");
36. }
37. document.writeln("</table>");
```

The Document Object Model (DOM)

- When a browser loads an HTML document into memory to be displayed on the screen, it first relies on a parser to read the actual contents of the document.
 - ▶ A parser is a low level piece of software that reads and processes data from a document.
- One of the primary tasks of any web browser is to present an HTML document in a window to display the various elements such as images, hyperlinks, and forms to name a few.
 - ▶ JavaScript provides an interface to the window and all of its elements by representing each of the elements as an object.
 - In order for scripts to be able to access the necessary objects within the window, a model of all of the objects is provided.
- The Document Object Model (DOM) is the model representing the organization of the objects within the page in memory.
 - ▶ The World Wide Web Consortium (W3C) has defined a standard DOM that is both platform-independent and language-independent.
 - ▶ The DOM is an API (application programming interface) that exposes a document (a web page for the purposes of this course) to the JavaScript engine.
 - Using the DOM, the structure of the document can be manipulated programmatically.
 - ▶ A DOM representation of a web page is also organized into a tree structure where each piece of information is represented as an object of type `Node`.
 - The root of a DOM tree is a `Node` object, specifically of type `Document`, and is referred to as the Document node.
 - The `document` variable is implicitly defined in JavaScript as a reference to the Document node (root) of the DOM tree.
 - `document` represents the parent of the `html` element.

Trees and Nodes

- Before looking at the DOM interface in detail, it is important to get a better understanding of the structure of an HTML document.
 - ▶ Consider the following XHTML document.

basic.html

```
1. <!doctype html>
2. <html>
3.   <head><title>Basic Document</title></head>
4.   <body><div>Hello</div></body>
5. </html>
```

- ▶ The DOM API represents the information in the document above as a tree of Node objects.
 - For example the body tag is represented as a Node object.
 - The div tag is represented as a Node.
 - The text Hello is represented as a Node object.
- ▶ It is important to realize there are several Node objects in the DOM representation of the above document that may not be immediately apparent.
 - The part of the document often overlooked is the white-space characters between the various start and end tags.

The Node Interface

- The Node interface has the following properties and methods.

Node Types (Read-Only)				
Node Type	Value		Node Type	Value
ELEMENT_NODE	1		PROCESSING_INSTRUCTION_NODE	7
ATTRIBUTE_NODE	2		COMMENT_NODE	8
TEXT_NODE	3		DOCUMENT_NODE	9
CDATA_SECTION_NODE	4		DOCUMENT_TYPE_NODE	10
ENTITY_REFERENCE_NODE	5		DOCUMENT_FRAGMENT_NODE	11
ENTITY_NODE	6		NOTATION_NODE	12

Node Properties (Read-Only)				
Node Type	Value		Node Type	Value
nodeName	String		previousSibling	Node
nodeValue	String		nextSibling	Node
nodeType	Number		attributes	NamedNodeMap
parentNode	Node		ownerDocument	Document
childNodes	NodeList		namespaceURI	String
firstChild	Node		prefix	String
lastChild	Node		localName	String

Node Methods for an object n	
Method	Description
n.hasChildNodes()	Returns a Boolean.
n.hasAttributes()	Returns a Boolean.
n.insertBefore(newChild, oldChild)	Inserts <i>newChild</i> Node before <i>oldChild</i> Node in the children of n.
n.replaceChild(newChild, oldChild)	Replaces the <i>oldChild</i> Node with the <i>newChild</i> Node in the children of n.
removeChild(oldChild)	Removes the <i>oldChild</i> Node from the children of n.
appendChild(newChild)	Appends the <i>newChild</i> Node at the end of the children of n.

The NodeList and NamedNodeMap Interfaces

- A **NodeList** is an ordered collection of **Node** objects that can be referenced by index.
 - ▶ A **NodeList** has a **length** property that returns the number of **Node** objects in the list.
 - ▶ The **item(index)** method returns a **Node** from by its index.
 - The **length** property and **item** method are often used together to loop through the **Node** objects in the **NodeList**.
- A **NamedNodeMap** is an unordered collection of **Node** objects that holds a set of name/value pairs.
 - ▶ Although unordered, the **NamedNodeMap** has a **length** property and **item** method that allow traversal much like a **NodeList**.
- The example that follows incorporates properties and methods of the **Node**, **NodeList**, and **NamedNodeMap** interfaces to iterate through all of the information in the HTML document.

traverseSource.html

```
1. <!doctype html>
2. <html>
3.   <head><title>Node Traversal Example</title>
4.     <script type="text/javascript" src="utilities.js">
5.     </script>
6.     <script type="text/javascript" src="traverseSource.js">
7.     </script>
8.   </head>
9.   <body onload="traverse();"><div>Hello</div></body>
10. </html>
```

- ▶ **utilities.js** is a set of utility functions for generating an HTML document.
- ▶ **traverseSource.js** is the code that relies on the various DOM interfaces to traverse the document.
- ▶ The **onload** attribute of the **body** tag calls the function named **traverse** in the **traverseSource.js** file when the browser finishes loading the HTML document into memory.

Node Traversal Example

utilities.js

```
1. function defaultHTMLStartTag(){
2.     return "<!doctype html><html>\n";
3. }
4.
5. function defaultHEADTag(aTitle, styleSheetName){
6.     return "<head><title>" + aTitle + "</title>\n" +
7.           cssTag(styleSheetName) + "</head>\n";
8. }
9.
10. function cssTag(styleSheetName){
11.     return "<link href=\"\" + styleSheetName +
12.           "\" type=\"text/css\" rel=\"stylesheet\" />";
13. }
14.
15. function asTableRow(){
16.     var txt = "<tr>";
17.     for (var i=0; i < arguments.length; i++)
18.         txt += "<td>" + arguments[i] + "</td>";
19.     txt += "</tr>";
20.     return txt;
21. }
22.
23. function writeToNewWindow(stringOfHTML){
24.     var theWin = window.open();
25.     theWin.document.write(stringOfHTML);
26.     theWin.document.close();
27. }
```

- ▶ The `writeToNewWindow(stringOfHTML)` function uses several methods from the built-in window object to open and write to a new browser window.
 - The `close` does not actually close the window but rather indicates that the output to the window has finished being sent.
- The `traverseSource.js` file is shown on the following page.

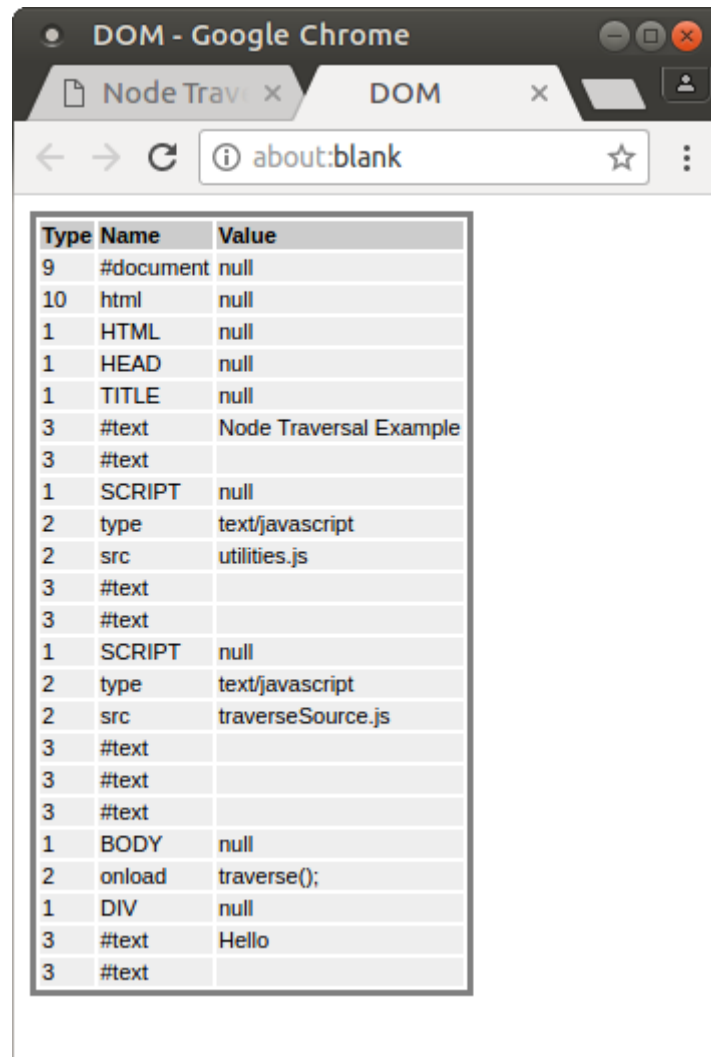
Node Traversal Example

traverseSource.js

```
1. var buffer;
2.
3. function traverse(){
4.     buffer = defaultHTMLStartTag() +
5.         defaultHEADTag("DOM", "traverse.css") +
6.         "<body>\n<table>\n<thead><tr><th>Type</th>" +
7.         "<th>Name</th><th>Value</th></tr></thead>" +
8.         "\n<tbody>";
9.     processNode(document);
10.    buffer += "</tbody>\n</table>\n</body>\n</html>";
11.    writeToNewWindow(buffer);
12. }
13.
14. function processNode(aNode){
15.     getNodeInfo(aNode);
16.     getAttrInfo(aNode);
17.     checkForChildren(aNode);
18. }
19.
20. function getNodeInfo(aNode){
21.     buffer += asTableRow(aNode.nodeType,
22.         aNode.nodeName, aNode.nodeValue);
23. }
24.
25. function getAttrInfo(aNode){
26.     if(aNode.attributes){
27.         var nodeMap = aNode.attributes;
28.         for(var i = 0; i < nodeMap.length; i++){
29.             getNodeInfo(nodeMap.item(i));
30.         }
31.     }
32. }
33.
34. function checkForChildren(aNode){
35.     if(aNode.hasChildNodes()){
36.         var children = aNode.childNodes;
37.         for(var i = 0; i < children.length; i++){
38.             processNode(children.item(i));
39.         }
40.     }
41. }
```

- The output from the previous HTML and JavaScript is shown on the following page.

Node Traversal Example



Type	Name	Value
9	#document	null
10	html	null
1	HTML	null
1	HEAD	null
1	TITLE	null
3	#text	Node Traversal Example
3	#text	
1	SCRIPT	null
2	type	text/javascript
2	src	utilities.js
3	#text	
3	#text	
1	SCRIPT	null
2	type	text/javascript
2	src	traverseSource.js
3	#text	
3	#text	
3	#text	
1	BODY	null
2	onload	traverse();
1	DIV	null
3	#text	Hello
3	#text	

- The contents of the `traverseSource.js` are described in more detail below
 - ▶ `buffer`
 - This variable is used to append all of the strings representing the new HTML document that is being built.
 - ▶ `traverse()`
 - This function relies on several of the utility functions defined in the `utilities.js` file.

Node Traversal Example

- ▶ `processNode(aNode)`
 - This function is a recursive function, in that a statement inside of the function calls the function itself.
 - The function relies on three other functions to get the actual work done.
 - Breaking a function into multiple functions calls often makes the code easier to follow and maintain over time.
- ▶ `getNodeInfo(aNode)`
 - This function relies on properties of the Node interface to collect information about the `aNode` object.
- ▶ `getAttrInfo(aNode)`
 - This function first determines if there are any attributes associated with `aNode`.
- ▶ `checkForChildren(aNode)`
 - This function first determines if there are any child nodes associated with `aNode`.
 - Inside of the `if` statement `children` (a reference to a `NodeList` object) is obtained from `aNode.childNodes`.
 - Each child node (`children.item(i)`) is then passed recursively to the `processNode` function until all nodes have been traversed.

The DOM Event Model

- We have seen that objects can have properties and methods associated with them. However, an object can possess a third trait, called an event.
 - ▶ An event is typically an action that occurs when a user interacts with a web page. Below are some examples.
 - Clicking on a hyperlink
 - Clicking on a button
 - Entering data in a form
 - ▶ Each event in JavaScript is represented as an Event object.
 - Like other objects we have studied, an Event object will have properties associated with it.
- The DOM provides methods for capturing events so that customized actions can be performed in response to them.
 - ▶ In order for a customized action to be performed when an event is triggered you must:
 - define the code that will get executed when the event occurs; and
 - connect the defined code to the particular event.
- JavaScript defines different types of events that are pre-defined for certain nodes in the DOM hierarchy.
 - ▶ The table on the following page lists each of the events and the conditions under which they are triggered.

The DOM Event Model

JavaScript Events		
Event Name	Triggered By;	Applies To
onload	The window finishes loading.	body, frameset
onunload	A document is removed from a window.	body, frameset
onclick	The mouse is clicked over an element.	Most Elements
ondblclick	The mouse is double clicked over an element.	Most Elements
onmousedown	The mouse is pressed over an element.	Most Elements
onmouseup	The mouse is released over an element.	Most Elements
onmouseover	The mouse is moved onto an element.	Most Elements
onmousemove	The mouse is moved while over an element.	Most Elements
onmouseout	The mouse is moved away from an element.	Most Elements
onfocus	An element receives focus either by the mouse or by tabbing navigation.	a, area, button, input, label, select, textarea
onblur	An element loses focus either by the mouse or by tabbing navigation.	a, area, button, input, label, select, textarea
onkeypress	A key is pressed and released over an element	Most Elements
onkeydown	A key is pressed down over an element.	Most Elements
onkeyup	A key is released over an element.	Most Elements
onsubmit	A form is submitted.	Form
onreset	A form is reset.	Form
onselect	A user selects some text in a text field.	input, textarea
onchange	A control loses the focus and its value has been modified since gaining focus.	input, select, textarea

- Using the event names from above as attributes to the HTML tags, code can then be executed when one of the events occur.
 - ▶ The value for each of the attributes is a string consisting of JavaScript code. The code may be one of the following.
 - A single JavaScript statement.
 - Multiple JavaScript statements separated by semicolons.
 - A call to a JavaScript function.

The DOM Event Model

- The example that follows demonstrates the handling of various mouse events using element attributes.

mouseEvents.js

```
1. var count = 0
2. function generateText(){
3.     var txtNode =
4.         document.getElementById("magic").firstChild;
5.     txtNode.insertData(0, " hello" + count++);
6. }
7. function resetText(){
8.     var txtNode =
9.         document.getElementById("magic").firstChild;
10.    txtNode.replaceData(0, txtNode.length, "hello");
11. }
```

- ▶ The HTML document that calls the functions is shown below.

mouseEvents.html

```
1. <!doctype html>
2. <html>
3.     <head><title>Mouse Event Examples</title>
4.         <script type="text/javascript" src="mouseEvents.js">
5.         </script>
6.     </head>
7.     <body>
8.         <div onmouseover="alert(new Date());">
9.             Move your mouse over me for the time and day
10.        </div>
11.        <hr />
12.        <div>
13.            </img>
16.            Click the image for a roll of the dice
17.        </div>
18.        <hr />
19.        <div>
20.            <p onmousemove="generateText();" onmouseout="resetText();">
21.                Moving the mouse over me will cause additional text to begin
22.                to appear below<br />Moving the mouse off will reset the text
23.            </p>
24.            <hr />
25.            <p id="magic">hello</p>
26.        </div>
27.    </body>
28. </html>
```

The DOM Event Model

- The page that is presented from the previous example is shown below.

