Introduction and Course Overview

Unit 1: Databases
Fundamentals
The Relational Model
The SQL Query Language
Case Study: An Astronomical Database

Unit 2: Programming in Python
Programming Basics
Working with Numbers
Making Decisions
Strings/Text and Lists
Accessing a Database
Working with Files

Unit 3: Data Visualization and Information Design
notes to be provided

Unit 4: Data Mining
Introduction
Classification Learning
Numeric Estimation
Association Learning
Preparing the Data
Case Study: Predicting Patient Outcomes

Case Study: A Biological Database

David G. Sullivan, Ph.D.
Boston University, Spring 2016
Welcome to CS 105!

- This course examines how collections of data are organized, stored, and processed.

- Topics include:
  - databases
  - programming
  - data mining
  - data visualization

- We'll consider applications from a variety of domains.
  - business, the arts, the life sciences, the social sciences
Broad Goals of the Course

- To give you computational tools for working with data
- To give you insights into databases and data mining
  - help you to understand their increasingly important role
- To expose you to the discipline of computer science
  - to how computer scientists think and solve problems

“Computer science is not so much the science of computers as it is the science of solving problems using computers.”
- Eric Roberts, Stanford

Data, Data Everywhere!

- financial data
- commercial data
- scientific data
- socioeconomic data
- etc.
Databases

• A database is a collection of related data.
  • example: the database behind StudentLink
  • other examples?

• Managed by some type of database management system (DBMS)
  • a piece of software (a program) that allows users to store, retrieve, and update collections of data

The Amount of Data Is Exploding!

• Example: the GenBank database of genetic sequences

  • on this graph, the data doubles every 12-14 months
  • as of May 2006, the doubling time was less than a year and getting shorter!

from: NCBI Field Guide presentation
(ftp://ftp.ncbi.nih.gov/pub/FieldGuide/Slides/Current/MtHolyoke.05.10.06/)
The Amount of Data Is Exploding!

- Example: the UN Database (data.un.org)

from "An Analysis of Factors Relating to Energy and Environment in Predicting Life Expectancy", CS 105 Final Project by Valerie Belding '12

The Amount of Data Is Exploding!

- Example: the Google Ngrams Corpus

books.google.com/ngrams
Relational Databases

- Most data collections are managed by a DBMS that employs a way of organizing data known as the relational model.
  - examples: IBM DB2, Oracle, Microsoft SQL Server, Microsoft Access
- In the relational model, data is organized into tables of records.
  - each record consists of one or more fields
  - example: a table of information about students

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>class</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>Warren Towers 100</td>
<td>2007</td>
<td>3/10/85</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>Student Village A210</td>
<td>2010</td>
<td>2/7/88</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>300 Main Hall</td>
<td>2008</td>
<td>10/2/86</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>Student Village B300</td>
<td>2009</td>
<td>7/13/88</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>The Dungeon</td>
<td>2007</td>
<td>11/1431</td>
</tr>
</tbody>
</table>
SQL

• A relational DBMS has an associated **query language** called SQL that is used to:
  • define the tables
  • add records to a table
  • modify or delete existing records
  • retrieve data according to some criteria
    • example: get the names of all students who live in Warren Towers
    • example: get the names of all students in the class of 2017, and the number of courses they are taking
  • perform computations on the data
    • example: compute the average age of all students who live in Warren Towers

Example Database

• A relational database containing data obtained from imdb.com

• We'll use SQL to answer (or at least explore) questions like:
  • How many of the top-grossing films of all time have won one or more Oscars?
  • Does the Academy discriminate against older women?
Beyond Relational Databases

• While relational databases are extremely powerful, they are sometimes inadequate/insufficient for a given problem.

• Example: DNA sequence data

GI|49175990|ref|NC_000913.2| Escherichia coli K12, complete genome
AGCTTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTA
AATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAA
ACGCCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGA
CAGTGCGGGCTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTC
TGCGTGTTGCCGATATTCTGGAAAGCAATGCCAGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTGGCGATGATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAACGTATTTTTGCCGAACTTTTGACGGGACTCGCCGCCGCCCAGCCGGGGTTCCCGCTGGCGCAA

• common queries involve looking for similarities or patterns
  • what genes in mice are similar to genes in humans?
• need special algorithms (problem-solving procedures)
• biologists store this data in text files and use simple computer programs to process it
• we'll learn to write simple programs using Python

Data MINING Everywhere!

• Informally, data mining is the process of finding patterns in data.

• Example: customized recommendations

Drama SUGGESTIONS (about 82) See all +

The Wrestler
Because you enjoyed: 
Mir City
Reservoir Dogs
The Big Lebowski
Add

The Visitor
Because you enjoyed: 
Gandhi
The Motorcycle Dance
The Queen
Add

Brick
Because you enjoyed: 
The Big Lebowski
Rushmore
Fight Club
Add

netflix.com

• Example: detecting credit-card fraud
Data MINING Everywhere!

The New York Times

How Companies Learn Your Secrets

By CHARLES DHING

Andrew Pole had just started working as a statistician for Target in 2002, when two college marketing department friends by his desk to ask an odd question: "If we wanted to figure out if a customer is pregnant, even if she didn't want us to know, can you do that?"

Pole has a master's degree in statistics and another in economics, and has been obsessed with the intersection of data and human behavior most of his life. His parents were teachers in New York while other kids were going to a H. Pole was doing algebra and writing computer programs, and "I kind of and evangelizing analytics."

- Target may know that your friend is pregnant before you do!

Data MINING Everywhere!

The New York Times

How the U.S. Uses Technology to Mine More Data More Quickly

WASHINGTON — When American analysts hunting terrorists sought new ways to comb through the troves of phone records, emails and other data piling up as digital communications exploded
Structure of the Course

- databases (4 weeks)
- programming in Python (4 weeks)
- data graphics/visualization (1 week)
- data mining (4 weeks)

Requirements

- Attendance at and participation in lectures and labs (10%)
  - everyone has an allowance of 3 missed classes; do not email unless extreme circumstances
  - labs begin next week
  - held in the CS teaching lab, EMA 304
  - complete Lab 0 sometime this week (see course website)
- Nine homework assignments (30%)
- Final project (10%): done in teams of three
  - use the techniques covered in the course to explore a dataset that interests you
- Three quizzes (20%)
- Final exam (30%)
Textbooks

- **optional:** *Database Concepts, 7th edition* by Kroenke & Auer (Prentice Hall, 2015)


- **required:** *The CS 105 Coursepack*
  - contains all of the lecture notes
  - will be available at Fedex Office at the corner of Comm Ave and Cummington Mall (the Warren Towers building)

Course Staff

- Instructor: Dave Sullivan (dgs@cs.bu.edu)

- Teaching fellow: Baichuan Zhou (baichuan@bu.edu)

- Office hours and contact info. will be available on the course Web site:
  - http://www.cs.bu.edu/courses/cs105

- *For general course-related questions, email: cs105-staff@cs.bu.edu which will forward your question to the full course staff.*
Algorithm for Finding My Office

1. Go to the entrance to the MCS (math/CS) building at 111 Cummington Mall – behind Warren Towers. Do not enter this building!

2. Turn around and cross the street to the doors across from MCS.

3. Enter those doors and take an immediate right. (continued on next slide)

Algorithm for Finding My Office (cont.)

4. As you turn right, you should see the door below. Open it and go up the stairs to the second floor.

5. As you leave the stairs, turn right and then go left into a small hallway. My office is the first door on the left (PSY 228D).
Other Details of the Syllabus

• Collaboration

• Policies:
  • lateness
  • please don't request an extension unless it's an emergency!
  • grading

• Please read the syllabus carefully and make sure that you understand the policies and follow them carefully.

• Let us know if you have any questions.
Database Fundamentals

Measuring Data: Bits and Bytes

- Bit = 0 or 1
- One byte is 8 bits.
  - example: 01101100
- Other common units:
  
<table>
<thead>
<tr>
<th>Name</th>
<th>Approximate Size</th>
<th>Exact Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>kilobyte (KB)</td>
<td>1000 bytes</td>
<td>$2^{10} = 1024$ bytes</td>
</tr>
<tr>
<td>megabyte (MB)</td>
<td>1 million bytes</td>
<td>$2^{20}$ bytes</td>
</tr>
<tr>
<td>gigabyte (GB)</td>
<td>1 billion bytes</td>
<td>$2^{30}$ bytes</td>
</tr>
</tbody>
</table>

- Scientists are starting to generate data collections measured in:
  - terabytes: $2^{40}$ or approx. $10^{12}$ bytes
  - petabytes: $2^{50}$ or approx. $10^{15}$ bytes
    - equivalent to the text in one billion books!
Storing Data: Memory

- Used to store programs and other data that are currently in use.
- Values stored in memory are read into the CPU to be operated on.
- The results of operations performed by the CPU can be written back to memory.
- Advantage of memory: short *access times*
  - can read from/write to memory in nanoseconds (10^{-9} sec)
- Disadvantages:
  - relatively expensive
  - contents are lost when the power is turned off

Storing Data: Secondary Storage

- Used to store programs and other data for later use.
  - examples: hard disks, floppy disks, CD/DVD drives, tape drives
- Advantages of hard disks:
  - relatively inexpensive
  - contents are retained when the power goes off
- Disadvantage: long access times
  - roughly 10 ms (10^{-3} sec)
  - in that time, a modern CPU can perform *millions* of operations!
  - it's important to minimize the number of times that the disk is accessed
What is a Database?

- A collection of data
  - it does not need to be on a computer.
    - example: the paper card catalogs that libraries maintained
- A given database may be divided into subcollections (tables)
  - should be related in some way
  - example: a university database
    - possible subcollections?

Database vs. Database Management System

- A database is a collection of data. It is not a piece of software.
- A database management system (DBMS) is the software that manages one or more databases.
Key Functions of a DBMS

1. efficient storage
2. providing a logical view of data
3. query processing
4. transaction management

• Let's look at each of them in turn.

1. Efficient Storage

• Recall: accessing the disk is very inefficient.

• A DBMS organizes the data on disk in ways that allow it to reduce the number of disk accesses.

• Example:
  • a database with 100,000 records
  • a given record is between 64-256 bytes long

• An inefficient approach:
  • give each record 256 bytes, even though it may not need it
  • scan through the database to find a record
  • may require thousands of disk reads!
1. Efficient Storage (cont.)

- A more efficient approach:
  - give each record only as much space as it needs
  - use a special *index structure*
    - allows the DBMS to locate a particular record *without* looking at every record
  - can dramatically reduce the number of disk accesses
    - as few as 1-3!

- A DBMS can also spread a database over multiple disks.
  - allows for larger collections of data
  - the disks can be accessed in parallel, which speeds things up
  - another advantage of using multiple disks?

2. Providing a Logical Representation of Data

Logical representation (tables, fields, etc.)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>class</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>Warren Towers 100</td>
<td>2007</td>
<td>3/10/85</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>Student Village A210</td>
<td>2010</td>
<td>2/7/88</td>
</tr>
</tbody>
</table>

- The DBMS takes care of translating between the representations.
  - makes the user's job much easier!

- This is an example of abstraction.
  - hide low-level details behind a simpler representation
  - an important concept in computer science
3. Query Processing

- A DBMS has some type of query language.
  - example: SQL
  - includes commands for:
    - adding new records
    - modifying or deleting existing records
    - retrieving data according to some criteria

- The DBMS determines the best way to execute a query-language command.
  - which index structures to use
  - if multiple operations are needed, the order in which they should be carried out

4. Transaction Management

- A transaction is a sequence of operations that is treated as a single logical operation.

- Example: balance transfer of $50 from blue to pink
  - remove $50 from blue
  - add $50 to pink

![Diagram of piggy banks before and after the transfer with $450 and $300 respectively.](image-url)
4. Transaction Management

• A transaction is a sequence of operations that is treated as a single logical operation.

• Example: balance transfer of $50 from blue to pink
  • remove $50 from blue
  • add $50 to pink

• Without a transaction, bad things could happen!

![Diagram](image_url)

• By using a transaction for the balance transfer, we ensure that all of the steps happen, or none do.
  • all or nothing!

![Diagram](image_url)
4. Transaction Management (cont.)

- Other examples:
  - making a flight reservation
    select flight, reserve seat, make payment
  - making an online purchase
  - making an ATM withdrawal

- Ensure that operations by different users don’t overlap in problematic ways.
  - example: what’s wrong with the following?
    
    ```
    user's balance transfer
    remove 500 from blue
    bank's check for clients below minimum balance
    read blue balance
    read pink balance
    if (blue + pink < minimum)
      charge the user a fee
    add 500 to pink
    ```

Database Applications

- Users often use a database application.
  - a separate piece of software that interacts with the DBMS

- Provide easier access to the database.
  - don’t need to know the query language

- Examples:
  - the software that runs on ATMs for a bank
  - a web interface to a library database
Desktop Database System

- Combines the functions of a database application and a DBMS.
  - examples: Microsoft Access, Filemaker Pro

- Includes tools/wizards for building the databases, forms, etc.

- Less flexible and less powerful than a full-fledged DBMS.
  - doesn’t support all possible operations
  - doesn’t support multi-user applications
  - doesn’t scale well to very large databases

Looking Ahead

- The logical representation that a DBMS uses is based on some type of data model.

- There are a number of different models that can be used for this purpose.

- The most prevalent one is the relational model.

- We’ll look next at the key features of this model.

- Reminder: complete Lab 0 by the first lab
The Relational Model

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

What Is a Data Model?

- A formal way of describing:
  - pieces of data (data items)
  - relationships between data items
  - constraints on the values of data items

- We'll focus on the relational model – the dominant data model in current database systems.

- To understand the benefits of the relational model, it helps to briefly consider earlier models.
Earlier Data Models

• Before the relational model, the data models were closely tied to the physical representation of the data.

• To access data records, users had to write programs that navigated from one record to another.
  • difficult to write
  • adding new fields required modifying the programs – even if the programs were not accessing the new fields!

The Relational Model: A Brief History


• The model was revolutionary because it provided data independence – separating the logical model of the data from its underlying physical representation.

• Allows users to access the data without understanding how it is stored on disk.
The Relational Model: A Brief History (cont.)

"Codd had a bunch of ... fairly complicated queries.... I could imagine how those queries would have been represented ... by programs that were five pages long.... Codd would sort of write them down as one-liners ... they weren't complicated at all. I said, 'Wow.' This was kind of a conversion experience for me."

— Don Chamberlin, describing a seminar that Codd gave at IBM about the new model

• Codd won the Turing Award (computer science's Nobel Prize) in 1981 for his work.

The Relational Model: Basic Concepts

• A database consists of a collection of tables.

• Example of a table:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>class</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>Warren Towers 100</td>
<td>2007</td>
<td>3/10/85</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>Student Village A210</td>
<td>2010</td>
<td>2/7/88</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>300 Main Hall</td>
<td>2008</td>
<td>10/2/86</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>Student Village B300</td>
<td>2009</td>
<td>7/13/88</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>The Dungeon</td>
<td>2007</td>
<td>11/1431</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• Each row in a table holds data that describes either:
  • an entity (a person, place, or thing!)
  • a relationship between two or more entities

• Each column in a table represents one attribute of an entity.
Relational Model: Terminology

- Two sets of terminology:
  - table = relation
  - row = tuple
  - column = attribute

- We'll use both sets of terms.

Requirements of a Relation

- Each column must have a unique name.

- The values in a column must be of the same type (i.e., must come from the same domain).

- Each cell must contain a single value.
  - example: we can’t do something like this:

    | id    | name     | ... | phones             |
    |-------|----------|-----|--------------------|
    | 12345678 | Jill Jones | ... | 123-456-5678, 234-666-7890 |
    | 25252525 | Alan Turing | ... | 777-777-7777, 111-111-1111 |
    | ...     | ...      | ... | ...                |

- No two rows can be identical.
  - identical rows are known as duplicates
Schema of a Relation

- The schema of a relation consists of:
  - the name of the relation
  - the names of its attributes
  - the domains (possible values) of the attributes (although we’ll ignore them for now)

If we name our earlier table Student, its schema would be:

\[
\text{Student}(id, \text{name}, \text{address}, \text{class}, \text{dob})
\]

Keys

- A key is an attribute or collection of attributes that can be used to uniquely identify a row in a relation.
- allows us to distinguish one row from another

- A relation may have more than one possible key.
  - example:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>...</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>jill jones</td>
<td>...</td>
<td><a href="mailto:jones@bu.edu">jones@bu.edu</a></td>
</tr>
<tr>
<td>25252525</td>
<td>alan turing</td>
<td>...</td>
<td><a href="mailto:aturing@bu.edu">aturing@bu.edu</a></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- possible keys include:
  - 
  - 
  - 
  -
Candidate Key

- A candidate key is a minimal collection of attributes that is a key.
  - minimal = no unnecessary attributes are included
    - not the same as minimum

- Example: consider again our Student table
  - (name, dob) is a minimal key if we need both attributes to uniquely identify a student
  - (id, email) is a key, but it is not minimal, because just one of these attributes is sufficient

Candidate Key (cont.)

- Consider a table describing the courses in which students are enrolled:

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>222566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

key? candidate key?

• student
• student, course
• student, course, credit status
Primary Key

• When defining a relation, we typically choose one of the candidate keys as the primary key.

• The records are arranged on disk to allow for quick retrieval using the value of the primary key.

• In a schema, we underline the primary key attribute(s).
  • example: \textit{Student}(id, name, address, class, dob)

Capturing Relationships

• In addition to storing info. about entities, we also use relations to capture relationships between two or more entities.

• Let's say that we have the following relations:
  • \textit{Student}\text{(id, name, address, class, dob)}: see earlier slides
  • \textit{Faculty}(id, name, office, phone)

\begin{tabular}{|l|l|l|l|}
\hline
\text{Id} & \text{name} & \text{office} & \text{phone} \\
\hline
11111 & Ted Codd & MCS 207 & 617-353-1111 \\
55555 & Grace Hopper & MCS 222 & 617-353-5555 \\
77777 & Edgar Dijkstra & MCS 266 & 617-353-7777 \\
\ldots & \ldots & \ldots & \ldots \\
\hline
\end{tabular}

• \textit{Department}(name, office, phone)

\begin{tabular}{|l|l|l|l|}
\hline
\text{name} & \text{office} & \text{phone} \\
\hline
\text{computer science} & MCS 140 & 617-353-8919 \\
\text{english} & 236 Bay State Rd. & 617-353-2506 \\
\text{mathematics} & MCS 140 & 617-353-2560 \\
\ldots & \ldots & \ldots \\
\hline
\end{tabular}
Capturing Relationships (cont.)

- One relationship among these three entities is the relationship between students and their advisors.
- We can capture this relationship by expanding the Student relation to include an attribute called `advisor` that stores the faculty ID of a student's advisor.

<table>
<thead>
<tr>
<th>Student</th>
<th>Faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td><code>id</code></td>
</tr>
<tr>
<td><code>name</code></td>
<td><code>name</code></td>
</tr>
<tr>
<td><code>advisor</code></td>
<td><code>office</code></td>
</tr>
<tr>
<td>Jill Jones</td>
<td>Ted Codd</td>
</tr>
<tr>
<td>Alan Turing</td>
<td>Grace Hopper</td>
</tr>
</tbody>
</table>

Examples:
- Jill Jones' advisor is Ted Codd.
- Alan Turing's advisor is Grace Hopper.

Foreign Keys

- `advisor` is an example of a foreign key – an attribute that takes on values from the primary-key column of another relation
  - the name of a foreign key does not need to match the name of the corresponding primary key
  - each value in a foreign-key column must match one of the values in the corresponding primary-key column
More Examples of Foreign Keys

- We can view students' majors as a relationship between students and departments.

- If students can have at most one major, we can capture the relationship by making the major part of the Student relation.
  - add a foreign-key attribute called major that takes on values from the primary key of Department

<table>
<thead>
<tr>
<th>Student</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>...</td>
<td>major</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>...</td>
<td>computer science</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>...</td>
<td>mathematics</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th>office</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer science</td>
<td>MCS 140</td>
<td>617-353-8919</td>
</tr>
<tr>
<td>english</td>
<td>236 Bay State Rd.</td>
<td>617-353-2506</td>
</tr>
<tr>
<td>mathematics</td>
<td>MCS 140</td>
<td>617-353-2560</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

More Examples of Foreign Keys (cont.)

- If students can have multiple majors, we can't just add an attribute for it to Student.
  - why?

- Instead, we create a separate relation that has two foreign keys:
  - one with values from the primary key of Student
  - one with values from the primary key of Department

<table>
<thead>
<tr>
<th>MajorsIn</th>
<th>student</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>computer science</td>
<td>...</td>
</tr>
<tr>
<td>12345678</td>
<td>English</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
<th></th>
<th>ID</th>
<th>name</th>
<th>...</th>
<th></th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>name</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td></td>
<td>...</td>
<td></td>
<td>computer science</td>
<td>...</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td></td>
<td>...</td>
<td></td>
<td>English</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

CS 105, Boston University, Spring 2016
Example of Creating a Relational Database

• Let's say that we're building a database for our new e-commerce website, TerrierStuff.com

• What relations might it make sense to include? (Give a partial schema for each.)

• What are possible primary keys for each relation? (underline the attributes in the schema)

• Where could foreign keys be used?

Constraints

• In the relational model, we can specify constraints on the values of attributes.

• The types of constraints include:
  • uniqueness constraints: specify that a given attribute or combination of attributes must have different values in each row (i.e., that no value or combination of values can appear more than once)
    • specifying a primary key imposes this type of constraint
  • referential integrity constraints: specify that a given attribute or combination of attributes must take on values that appear in another column or combination of columns
    • specifying a foreign key imposes this type of constraint

• If we attempt to add/insert a row that would violate a constraint, the DBMS prevents us from doing so.
Constraints (cont.)

• Example: assume that the tables below show all of their tuples.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>computer science</td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td>english</td>
</tr>
</tbody>
</table>

• Which of the following operations would the DBMS allow?
  • adding (12345678, "John Smith", ...) to Student
  • adding (33333333, "Howdy Doody", ...) to Student
  • adding (12345678, "physics") to MajorsIn
  • adding (25252525, "english") to MajorsIn

Surrogate Keys

• Problem: what if a relation doesn't have any candidate keys – or if the available candidate keys have values that take up a lot of space?
  • example: a relation capturing info. about actors, directors, etc.

<table>
<thead>
<tr>
<th>name</th>
<th>birthplace</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>James Cameron</td>
<td>Ontario, Canada</td>
<td>8/16/54</td>
</tr>
<tr>
<td>James Cameron</td>
<td>London, England</td>
<td>6/17/11</td>
</tr>
<tr>
<td>James Cameron</td>
<td>LaCrosse, WI</td>
<td>3/23/14</td>
</tr>
<tr>
<td>Morgan Freeman</td>
<td>Long Beach, CA</td>
<td>12/5/69</td>
</tr>
<tr>
<td>Morgan Freeman</td>
<td>Memphis, TN</td>
<td>6/1/1937</td>
</tr>
</tbody>
</table>

• schema: Person(name, birthplace, dob)
• why not use (name, birthplace) or (name, dob)?
Surrogate Keys (cont.)

- Solution: create a **surrogate key** – an attribute that is added to a relation for the purpose of serving as its primary key.

- Database systems typically allow you to indicate that an attribute of a relation is a surrogate key.
  - in Microsoft Access, this is known as an *AutoNumber attribute*
  - the DBMS takes care of assigning a value to this attribute whenever a new tuple is added to the table
    - example: assign 1 to the first tuple, 2 to the second tuple, etc.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>birthplace</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>James Cameron</td>
<td>Ontario, Canada</td>
<td>8/16/54</td>
</tr>
<tr>
<td>25252525</td>
<td>James Cameron</td>
<td>London, England</td>
<td>6/17/11</td>
</tr>
<tr>
<td>335666691</td>
<td>James Cameron</td>
<td>LaCrosse, WI</td>
<td>3/23/14</td>
</tr>
<tr>
<td>45678900</td>
<td>Morgan Freeman</td>
<td>Long Beach, CA</td>
<td>12/5/69</td>
</tr>
<tr>
<td>66666666</td>
<td>Morgan Freeman</td>
<td>Memphis, TN</td>
<td>6/1/1937</td>
</tr>
</tbody>
</table>

Null Values

- Recall: all values in a given column of a table must come from the same domain.

- By default, the domains of most columns include a special value called **null**.

- Null values can be used to indicate one of the following:
  - the value of an attribute is unknown for a particular tuple
  - the attribute doesn't apply to a particular tuple. example:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>...</th>
<th>major</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td></td>
<td>computer science</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td></td>
<td>mathematics</td>
</tr>
<tr>
<td>33333333</td>
<td>Dan Dabbler</td>
<td></td>
<td>null</td>
</tr>
</tbody>
</table>
Null Values (cont.)

- If we tell the DBMS that an attribute is serving as the primary key, it won’t allow us to assign a null value to that attribute.
  - why does this make sense?
- Foreign-key attributes can be assigned null values.
- We can also tell the DBMS that we don’t want a column to be assigned a null value.

Anomalies

- Problems can result if you try to put too much data in one table. Example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
<th>Adviser</th>
<th>AdviserEmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews, Matthew</td>
<td><a href="mailto:Math@ourcampus.edu">Math@ourcampus.edu</a></td>
<td>Baker</td>
<td><a href="mailto:Baker@ourcampus.edu">Baker@ourcampus.edu</a></td>
</tr>
<tr>
<td>Bivins, Lisa</td>
<td><a href="mailto:Lisa@ourcampus.edu">Lisa@ourcampus.edu</a></td>
<td>Valdez</td>
<td><a href="mailto:Valdez@ourcampus.edu">Valdez@ourcampus.edu</a></td>
</tr>
<tr>
<td>Fieser, Douglas</td>
<td><a href="mailto:Doug@ourcampus.edu">Doug@ourcampus.edu</a></td>
<td>Baker</td>
<td><a href="mailto:Baker@ourcampus.edu">Baker@ourcampus.edu</a></td>
</tr>
<tr>
<td>Haas, Terry</td>
<td><a href="mailto:Terry@ourcampus.edu">Terry@ourcampus.edu</a></td>
<td>Targ</td>
<td><a href="mailto:Targ@ourcampus.edu">Targ@ourcampus.edu</a></td>
</tr>
<tr>
<td>Manso, Chip</td>
<td><a href="mailto:Chip@mysamer.com">Chip@mysamer.com</a></td>
<td>Tran</td>
<td><a href="mailto:Tran@ourcampus.edu">Tran@ourcampus.edu</a></td>
</tr>
<tr>
<td>Le, Tru</td>
<td><a href="mailto:Tru@ourcampus.edu">Tru@ourcampus.edu</a></td>
<td>Valzer</td>
<td><a href="mailto:Valzer@ourcampus.edu">Valzer@ourcampus.edu</a></td>
</tr>
<tr>
<td>Thompson, James</td>
<td><a href="mailto:James@mysamer.com">James@mysamer.com</a></td>
<td>Targ</td>
<td><a href="mailto:Targ@ourcampus.edu">Targ@ourcampus.edu</a></td>
</tr>
</tbody>
</table>

  These problems are known as anomalies. They include:
  - redundancy: data is repeated unnecessarily
  - update anomalies: inconsistencies result if we fail to update all copies of a piece of repeated data
  - deletion anomalies: deleting one type of data causes us to lose other, unrelated data
  - insertion anomalies: in order to insert one type of data, we need in to include other, unrelated info.
Avoiding Anomalies

- Rules of thumb for avoiding anomalies:
  - give each type of entity its own relation
  - connect related entities using foreign keys – and possibly a separate relation

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>11111</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>55555</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>office</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111</td>
<td>Ted Codd</td>
<td>MCS 207</td>
<td>617-353-1111</td>
</tr>
<tr>
<td>55555</td>
<td>Grace Hopper</td>
<td>MCS 222</td>
<td>617-353-5555</td>
</tr>
<tr>
<td>77777</td>
<td>Edgar Dijkstra</td>
<td>MCS 266</td>
<td>617-353-7777</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Normalization

- Normalization is the process of taking a relational schema and revising it as needed to eliminate anomalies.

- Informally, we can:
  - look for cases in which a relation stores information about two or more types of entities
  - break it up into multiple relations, each of which represents either:
    - a single type of entity
    - a relationship between two or more types of entities
Normalization (cont.)

• Example (based on one from the textbook):
a landscaping company wants to store info. about the properties
  that it maintains and the service calls that it makes.

• They are thinking of using the following schema:
  \[
  \text{Property(name, type, street, city, zip, service\_date, description, amount)}
  \]

• Here is some sample data:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Street</th>
<th>City</th>
<th>Zip</th>
<th>Service_Date</th>
<th>Description</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eastlake Building</td>
<td>Office</td>
<td>123 Eastlake</td>
<td>Seattle</td>
<td>98119</td>
<td>5/5/02</td>
<td>Lawn Mow</td>
<td>$42.50</td>
</tr>
<tr>
<td>Elm St Apts</td>
<td>Apartment</td>
<td>4 East Elm</td>
<td>Lynnwood</td>
<td>98023</td>
<td>5/7/00</td>
<td>Lawn Mow</td>
<td>$125.00</td>
</tr>
<tr>
<td>Jefferson Hill</td>
<td>Office</td>
<td>42 West 7th St</td>
<td>Bellevue</td>
<td>98040</td>
<td>5/7/02</td>
<td>Garden Service</td>
<td>$53.00</td>
</tr>
<tr>
<td>Eastlake Building</td>
<td>Office</td>
<td>123 Eastlake</td>
<td>Seattle</td>
<td>98119</td>
<td>5/7/02</td>
<td>Lawn Mow</td>
<td>$42.50</td>
</tr>
<tr>
<td>Elm St Apts</td>
<td>Apartment</td>
<td>4 East Elm</td>
<td>Lynnwood</td>
<td>98023</td>
<td>5/14/02</td>
<td>Lawn Mow</td>
<td>$125.00</td>
</tr>
<tr>
<td>Eastlake Building</td>
<td>Office</td>
<td>144 Eastlake</td>
<td>Bellevue</td>
<td>98040</td>
<td>5/10/02</td>
<td>Lawn Mow</td>
<td>$63.00</td>
</tr>
</tbody>
</table>

• What changes (if any) would you make to this schema?
The SQL Query Language

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

Why Learn SQL?

- Desktop database systems like Access provide tools for manipulating data in a database.
  - use for queries that specify what data you want to extract
    - e.g., give me the records of all students who are sophomores and are majoring in computer science
    - can also use to add, modify, or remove data
    - are fairly intuitive to use
  - However, these tools don’t allow you to perform all possible types of queries.
  - For more flexibility and power, we use SQL.
    - a query language
  - In addition, knowledge of SQL is needed to perform queries from within a program.
Example Domain: a University

- We'll continue to use relations from a university database.
  - four relations that store info. about a type of entity:
    - Student(id, name)
    - Department(name, office)
    - Room(id, name, capacity)
    - Course(name, start_time, end_time, room)
  - two relations that capture relationships between entities:
    - MajorsIn(student, dept)
    - Enrolled(student, course, credit_status)
- The Course relation also captures a relationship – the relationship between a course and the room in which it meets.
- What would you specify as the primary key of each relation?

Foreign Keys in the University Database

- Student(id, name)
- Department(name, office)
- Room(id, name, capacity)
- Course(name, start_time, end_time, room)
- MajorsIn(student, dept)
- Enrolled(student, course, credit_status)
- Which of the attributes should be foreign keys, and what are the associated referential-integrity constraints?
### Student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

### Room

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CAS Tsiai</td>
<td>500</td>
</tr>
<tr>
<td>2000</td>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>3000</td>
<td>EDU Lecture Hall</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>CAS 315</td>
<td>40</td>
</tr>
<tr>
<td>5000</td>
<td>CAS 314</td>
<td>80</td>
</tr>
<tr>
<td>6000</td>
<td>CAS 226</td>
<td>50</td>
</tr>
<tr>
<td>7000</td>
<td>MCS 205</td>
<td>30</td>
</tr>
</tbody>
</table>

### Course

<table>
<thead>
<tr>
<th>name</th>
<th>start_time</th>
<th>end_time</th>
<th>room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 105</td>
<td>13:00:00</td>
<td>14:00:00</td>
<td>4000</td>
</tr>
<tr>
<td>CS 111</td>
<td>09:30:00</td>
<td>11:00:00</td>
<td>5000</td>
</tr>
<tr>
<td>CS 460</td>
<td>16:00:00</td>
<td>17:30:00</td>
<td>7000</td>
</tr>
<tr>
<td>CS 510</td>
<td>12:00:00</td>
<td>13:30:00</td>
<td>7000</td>
</tr>
</tbody>
</table>

### Department

<table>
<thead>
<tr>
<th>name</th>
<th>office</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp sci</td>
<td>MCS 140</td>
</tr>
<tr>
<td>math</td>
<td>MCS 140</td>
</tr>
<tr>
<td>the occult</td>
<td>The Dungeon</td>
</tr>
<tr>
<td>english</td>
<td>235 Bay State Road</td>
</tr>
</tbody>
</table>

### Enrolled

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

### Majors

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

---

**SQLite**

- An open-source relational DBMS (RDBMS)
- It can be easily downloaded and used on any common type of platform (Windows, Mac, Linux).
  - including the machines in the lab
Traditional DBMS Structure

- Most DBMSs use a client-server approach.
  - the DBMS runs as a server program
  - commands are sent to it from separate client programs
  - multiple client programs can use the same server
  - client programs don’t have to be on the same computer as the server program

- Can you think of another type of application that uses this approach?

SQLite’s Structure

- SQLite does not use a client-server approach.
  - it is serverless

- It is a library that can be used by programs that want to maintain a database.
  - no separate server program is needed
  - still allows multiple users to access the same database, provided they have access to the same files

- Because it is serverless, it is much easier to set up.

- A SQLite database (i.e., a collection of tables) is stored in a single file.
  - cross-platform: can create the file on one machine/OS, and use it on a different OS
SQLite Manager

- One user-friendly way to use SQLite is to use the SQLite Manager add-on for Firefox.

- To download and use it, follow the instructions in PS 2.

Querying an Existing Database

- In Problem Set 2, we will give you the database file that you should use.

- Thus, we will focus first on how to perform queries.
  - the SELECT command
  - initially, we will just query a single table

- Later, we will look at how to:
  - create a table
  - modify an existing table
  - perform queries from multiple tables
**SELECT (from a single table)**

- **Example:** to get the ID numbers of all grad-credit students, we do:
  ```sql
  SELECT student
  FROM Enrolled
  WHERE credit_status = 'grad';
  ```

- **Basic syntax:**
  ```sql
  SELECT column1, column2, ...
  FROM table
  WHERE selection condition;
  ```
  - the FROM clause specifies which table you are using
  - the WHERE clause specifies which rows should be included in the result
  - the SELECT clause specifies which columns should be included

- The result of a SELECT command is a relation!

**SELECT (from a single table) (cont.)**

- **Example:**
  ```sql
  SELECT student
  FROM Enrolled
  WHERE credit_status = 'grad';
  ```

```sql
<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>
```

WHERE credit_status = 'grad';

```sql
<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>
```

SELECT student

```sql
<table>
<thead>
<tr>
<th>student</th>
</tr>
</thead>
<tbody>
<tr>
<td>45678900</td>
</tr>
<tr>
<td>45678900</td>
</tr>
</tbody>
</table>
```
Selecting Entire Columns

• If there’s no WHERE clause, the result will consist of one or more entire columns. No rows will be excluded.

```
SELECT student
FROM Enrolled;
```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

Selecting Entire Rows

• If we want the result to include entire rows (i.e., all of the columns), we use a `*` in the SELECT clause:

```
SELECT *
FROM Enrolled
WHERE credit_status = 'grad';
```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>
The **WHERE** Clause

```sql
SELECT column1, column2, ...
FROM table
WHERE selection condition;
```

- The selection condition in the **WHERE** clause must consist of an expression that evaluates to either true or false.
  - example: a comparison like `credit_status = 'grad'`
  - it can include the name of any column from the table(s) mentioned in the **FROM** clause

- The results of the **SELECT** command will include only those tuples for which the selection condition evaluates to true.

---

**Simple Comparisons**

- The simplest selection condition is a comparison that uses one of the following *comparison operators*:

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td><code>capacity &lt; 200</code></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td><code>start_time &gt; '11:00'</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td><code>end_time &lt;= '16:00'</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td><code>id &gt;= '01000000'</code></td>
</tr>
<tr>
<td>=</td>
<td>equal to</td>
<td><code>room = id</code></td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td><code>credit_status != 'grad'</code></td>
</tr>
</tbody>
</table>
Let's practice what we've learned so far…

### Student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CAS Tsai</td>
<td>500</td>
</tr>
<tr>
<td>2000</td>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>3000</td>
<td>EDU Lecture Hall</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>CAS 315</td>
<td>40</td>
</tr>
<tr>
<td>5000</td>
<td>CAS 314</td>
<td>80</td>
</tr>
<tr>
<td>6000</td>
<td>CAS 226</td>
<td>50</td>
</tr>
<tr>
<td>7000</td>
<td>MCS 205</td>
<td>30</td>
</tr>
</tbody>
</table>

### Course

<table>
<thead>
<tr>
<th>name</th>
<th>start_time</th>
<th>end_time</th>
<th>room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 105</td>
<td>13:00:00</td>
<td>14:00:00</td>
<td>4000</td>
</tr>
<tr>
<td>CS 111</td>
<td>09:30:00</td>
<td>11:00:00</td>
<td>5000</td>
</tr>
<tr>
<td>CS 460</td>
<td>16:00:00</td>
<td>17:30:00</td>
<td>7000</td>
</tr>
<tr>
<td>CS 510</td>
<td>12:00:00</td>
<td>13:30:00</td>
<td>7000</td>
</tr>
</tbody>
</table>

### Department

<table>
<thead>
<tr>
<th>name</th>
<th>office</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 105</td>
<td>MCB 140</td>
</tr>
<tr>
<td>CS 111</td>
<td>MCB 140</td>
</tr>
<tr>
<td>CS 460</td>
<td>The Dungeon</td>
</tr>
<tr>
<td>CS 510</td>
<td>235 Bay State Road</td>
</tr>
</tbody>
</table>

### Enrolled

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 111</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

### Majors in

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Comp Sci</td>
</tr>
<tr>
<td>45678900</td>
<td>Comp Sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Comp Sci</td>
</tr>
<tr>
<td>33566891</td>
<td>English</td>
</tr>
<tr>
<td>45678900</td>
<td>The Occult</td>
</tr>
</tbody>
</table>

---

**Practice**

- Write a query that finds the capacity of CAS 315.

- Write a query that lists the names and start times of all courses.

- Write a query that gets all information for rooms with ID numbers no greater than 4000.
Comparisons Involving Pattern Matching

• Let's say we want the names and rooms of all CS courses.
  • we know the names all begin with 'CS'
  • we need to find courses with names matching this pattern

• Here's how to do it:

  ```sql
  SELECT name, room
  FROM Course
  WHERE name LIKE 'CS%';
  ```

• We use the LIKE operator, which performs pattern matching.

• The pattern is specified using wildcard characters:
  • % stands for 0 or more arbitrary characters
  • _ stands for a single arbitrary character

Comparisons Involving Pattern Matching (cont.)

• More examples: let's say that you have the comparison
  
  ```sql
  word LIKE '_at%'
  ```

  • what is the value of this comparison (true or false) for each of the following values of word?
    - 'batter' ?
    - 'at' ?
    - 'sat' ?
    - 'attribute' ?
    - 'atomic matter' ?

• DBMSs typically have an operator that performs case-insensitive pattern matching.
  • not part of the SQL standard
  • different implementations use different names for it

• In SQLite, the LIKE operator itself is case-insensitive.
  There's no easy way to perform case-sensitive pattern matching.
  • the = operator is case-sensitive.
Comparisons Involving **NULL**

- Because **NULL** is a special value, any comparison involving **NULL** that uses the standard operators is always false.

- For example, all of the following will always be false:
  
  ```
  room = NULL  
  room != NULL  
  NULL != 10  
  NULL = NULL  
  ```

- SQL provides special operators:
  
  - **IS NULL**
  - **IS NOT NULL**

- Example:
  
  ```
  SELECT name  
  FROM Course  
  WHERE room IS NULL;
  ```

---

**Forming More Complex Selection Conditions**

- We often want to select rows based on more than one condition – or based on the opposite of a condition.

  - examples:
    
    - if the course is a CS course and it meets before noon
    - if the course does not meet between 9 and noon

- SQL provides three **logical operators** for this purpose:

  **name** | **example and meaning**
  --- | ---
  **AND** | `name LIKE 'CS%' AND end_time < '12:00'`
  | *true if both conditions are true, and false otherwise*
  **OR** | `name LIKE 'CS%' OR name LIKE 'Math%'`
  | *true if one or both of the conditions are true; false if both conditions are false*
  **NOT** | `NOT(start_time >='9:00' AND end_time <='12:00')`
  | *true if the condition is false, and false if it is true*
Range Comparisons

- SQL also provides a special operator called `BETWEEN` for checking if a value falls within a range of values.

- For example:

  ```sql
  SELECT id
  FROM Room
  WHERE capacity BETWEEN 100 AND 200;
  ```

  which is equivalent to

  ```sql
  SELECT id
  FROM Room
  WHERE capacity >= 100 AND capacity <= 200;
  ```

Removing Duplicates

- By default, the relation produced by a `SELECT` command may include duplicate tuples.
- example: find the IDs of all students enrolled in a course

  ```sql
  SELECT student
  FROM Enrolled;
  ```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

  Enrolled

  ```sql
  SELECT student
  FROM Enrolled;
  ```

<table>
<thead>
<tr>
<th>student</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>25252525</td>
</tr>
<tr>
<td>45678900</td>
</tr>
<tr>
<td>33566891</td>
</tr>
<tr>
<td>45678900</td>
</tr>
</tbody>
</table>
Removing Duplicates (cont.)

- To eliminate duplicates, add the keyword `DISTINCT` to the `SELECT` clause:

  ```sql
  SELECT DISTINCT student
  FROM Enrolled;
  ```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

- More generally:

  ```sql
  SELECT DISTINCT column1, column2, ...
  ```

Aggregate Functions

- The SELECT clause can include an *aggregate function*, which performs a computation on a collection of values of an attribute.

- Example: find the average capacity of rooms in CAS:

  ```sql
  SELECT AVG(capacity)
  FROM Room
  WHERE name LIKE 'CAS%';
  ```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CAS Tsai</td>
<td>500</td>
</tr>
<tr>
<td>2000</td>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>3000</td>
<td>EDU Lecture Hall</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>CAS 315</td>
<td>40</td>
</tr>
<tr>
<td>5000</td>
<td>CAS 314</td>
<td>80</td>
</tr>
<tr>
<td>6000</td>
<td>CAS 226</td>
<td>50</td>
</tr>
<tr>
<td>7000</td>
<td>MCS 205</td>
<td>30</td>
</tr>
</tbody>
</table>

WHERE

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CAS Tsai</td>
<td>500</td>
</tr>
<tr>
<td>2000</td>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>CAS 315</td>
<td>40</td>
</tr>
<tr>
<td>5000</td>
<td>CAS 314</td>
<td>80</td>
</tr>
<tr>
<td>6000</td>
<td>CAS 226</td>
<td>50</td>
</tr>
<tr>
<td>7000</td>
<td>MCS 205</td>
<td>30</td>
</tr>
</tbody>
</table>

AVG(capacity)

154.0
Aggregate Functions (cont.)

- Possible functions include:
  - MIN, MAX: find the minimum/maximum of a value
  - AVG, SUM: compute the average/sum of numeric values
  - COUNT: count the number of values

- For AVG, SUM, and COUNT, we can add the keyword DISTINCT to perform the computation on all distinct values.
  - example: find the number of students enrolled for courses:
    ```sql
    SELECT COUNT(DISTINCT student) FROM Enrolled;
    ```

- SELECT COUNT(*) will count the number of tuples in the result of the select command.
  - example: find the number of CS courses
    ```sql
    SELECT COUNT(*)
    FROM Course
    WHERE name LIKE 'CS%';
    ```
  - COUNT(attribute) counts the number of non-NULL values of attribute, so it won't always be equivalent to COUNT(*)

- Aggregate functions cannot be used in the WHERE clause.

- Practice with aggregate functions: write a query to find the largest capacity of any room in CAS:
Aggregate Functions (cont.)

• What if we wanted the name of the room with the max. capacity?

• The following will not work!

```sql
SELECT name, MAX(capacity)
FROM Room
WHERE name LIKE 'CAS%';
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CAS Tsai</td>
<td>500</td>
</tr>
<tr>
<td>2000</td>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>3000</td>
<td>EDU Lecture Hall</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>CAS 315</td>
<td>40</td>
</tr>
<tr>
<td>5000</td>
<td>CAS 314</td>
<td>80</td>
</tr>
<tr>
<td>6000</td>
<td>CAS 226</td>
<td>50</td>
</tr>
<tr>
<td>7000</td>
<td>MCS 205</td>
<td>30</td>
</tr>
</tbody>
</table>

Aggregate Functions (cont.)

• In general, you can't mix aggregate functions with column names in the SELECT clause.
  • SQLite doesn't complain, but the results it gives you won't necessarily make sense.
Subqueries

- A *subquery* allows us to use the result of one query in the evaluation of another query.
  - the queries can involve the same table or different tables

- We can use a subquery to solve the previous problem:

```sql
SELECT name, capacity
FROM Room
WHERE name LIKE 'CAS%' 
  AND capacity = (SELECT MAX(capacity) 
                  FROM Room 
                  WHERE name LIKE 'CAS%');
```

```sql
SELECT name, capacity
FROM Room
WHERE name LIKE 'CAS%' 
  AND capacity = 500;
```

Subqueries and Set Membership

- Subqueries can be used to test for *set membership* in conjunction with the *IN* and *NOT IN* operators.
- example: find all students who are enrolled in CS 105

```sql
SELECT name
FROM Student
WHERE id IN (SELECT student 
              FROM Enrolled 
              WHERE course = 'CS 105');
```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

```sql
SELECT id, name
FROM Student
WHERE id IN (SELECT id 
              FROM Enrolled 
              WHERE course = 'CS 105');
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

---

CS 105, Boston University, Spring 2016

David G. Sullivan, Ph.D
Applying an Aggregate Function to Subgroups

- A `GROUP BY` clause allows us to:
  - group together tuples that have a common value
  - apply an aggregate function to the tuples in each subgroup

- Example: find the enrollment of each course:
  ```sql
  SELECT course, COUNT(*)
  FROM Enrolled
  GROUP BY course;
  ```

- When you group by an attribute, you can include it in the `SELECT` clause with an aggregate function.
  - because we're grouping by that attribute, every tuple in a given group will have the same value for it

---

Evaluating a query with `GROUP BY`

```sql
SELECT course, COUNT(*)
FROM Enrolled
GROUP BY course;
```
Applying a Condition to Subgroups

- A **HAVING** clause allows us to apply a selection condition to the subgroups produced by a **GROUP BY** clause.
  - example: find enrollments of courses with at least 2 students
    
    ```
    SELECT course, COUNT(*)
    FROM Enrolled
    GROUP BY course
    HAVING COUNT(*) > 1;
    ```

- Important difference:
  - a **WHERE** clause is applied before grouping
  - a **HAVING** clause is applied after grouping

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>66666666</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 105</td>
<td>grad</td>
</tr>
</tbody>
</table>

---

Sorting the Results

- An **ORDER BY** clause sorts the tuples in the result of the query by one or more attributes.
  - ascending order by default, use **DESC** to get descending
  - example:
    
    ```
    SELECT name, capacity
    FROM Room
    WHERE capacity > 50
    ORDER BY capacity DESC, name;
    ```

<table>
<thead>
<tr>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS Tsai</td>
<td>500</td>
</tr>
<tr>
<td>CAS BigRoom</td>
<td>100</td>
</tr>
<tr>
<td>EDU Lecture Hall</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Summary: SELECT for a single table

```
SELECT column1, column2, ...
FROM table
WHERE condition
GROUP BY column
HAVING condition
ORDER BY one or more columns;
```

- The clauses are effectively applied in this order:
  1. WHERE
  2. GROUP BY
  3. HAVING
  4. SELECT
  5. ORDER BY
Let's practice what we've learned so far…

### Student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

### Course

<table>
<thead>
<tr>
<th>name</th>
<th>start_time</th>
<th>end_time</th>
<th>room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS 105</td>
<td>13:00:00</td>
<td>14:00:00</td>
<td>4000</td>
</tr>
<tr>
<td>CS 111</td>
<td>09:30:00</td>
<td>11:00:00</td>
<td>5000</td>
</tr>
<tr>
<td>CS 460</td>
<td>10:00:00</td>
<td>17:30:00</td>
<td>7000</td>
</tr>
<tr>
<td>CS 510</td>
<td>12:00:00</td>
<td>13:30:00</td>
<td>7000</td>
</tr>
</tbody>
</table>

### Department

<table>
<thead>
<tr>
<th>name</th>
<th>office</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp sci</td>
<td>MCS 140</td>
</tr>
<tr>
<td>mathematics</td>
<td>MCS 140</td>
</tr>
<tr>
<td>the occult</td>
<td>The Dungeon</td>
</tr>
<tr>
<td>english</td>
<td>235 Bay State Road</td>
</tr>
</tbody>
</table>

### Enrolled

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45679000</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

### Majors

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>75255525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45679000</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

---

CS 105, Boston University, Spring 2016

David G. Sullivan, Ph.D
CREATE TABLE

• What it does: creates a relation with the specified schema

• Basic syntax:

```
CREATE TABLE relation_name(
    attribute1_name attribute1_type,
    attribute2_name attribute2_type,
    ...
    attributeN_name attributeN_type
);
```

• Examples:

```
CREATE TABLE Student(id CHAR(8), name VARCHAR(30));
```

```
CREATE TABLE Room(id CHAR(4), name VARCHAR(20), capacity INTEGER);
```

Attribute Types

• An attribute’s type specifies the domain of the attribute.

• Possible types include:

  • INTEGER: a four-byte integer (-2147483648 to +2147483647)
  • CHAR(n): a fixed-length string of n characters
  • VARCHAR(n): a variable-length string of up to n characters
  • REAL: a real number (i.e., one that may have a fractional part)
  • DATE: a date (e.g., 2007-01-26 or 01/26/2007)
  • TIME: a time (e.g., 11:00 AM or 15:30:30)

• When specifying a non-numeric value, you must surround it with single quotes (e.g., 'Jill Jones' or '2007-01-26').
**CHAR vs. VARCHAR**

- **CHAR(n)** is used for fixed-length strings of exactly n characters.
  - if fewer than n characters, the DBMS should pad with spaces
  - example:
    - id has the type CHAR(6)
    - you give it the value 'hello'
    - 'hello ' should be stored

- **VARCHAR(n)** is used for variable-length strings of up to n characters.
  - the system does not pad values

- With both CHAR and VARCHAR, values with more than n characters are truncated.

- If an attribute's values can have a wide range of possible lengths, it's usually better to use VARCHAR.

---

**Types in SQLite**

- SQLite has its own types, including:
  - INTEGER
  - REAL
  - TEXT

- It also allows you to use the typical SQL types, but it converts them to one of its own types.

- As a result, the length restrictions indicated for CHAR and VARCHAR are not observed.

- It is also more lax in type checking than typical DBMSs.
Specifying a Primary Key

- If the primary key is a single attribute:
  
  ```sql
  CREATE TABLE relation_name(
      attribute_name attribute_type PRIMARY KEY, ...
  );
  ```

- Example:
  ```sql
  CREATE TABLE Student(
      id CHAR(8) PRIMARY KEY,
      name VARCHAR(30)
  );
  ```

- If the primary key is a combination of two or more attributes, it is specified separately:

  ```sql
  CREATE TABLE MajorsIn(
      student CHAR(8), dept VARCHAR(30),
      PRIMARY KEY (student, dept)
  );
  ```

Specifying a Foreign Key

- Foreign keys are typically specified after the attribute specifications.

- Syntax:
  ```sql
  CREATE TABLE relation_name(
      attribute_name attribute_type, ...
      FOREIGN KEY (attribute(s)) REFERENCES PK_TableName(PK_attribute(s))
  );
  ```

- Example:
  ```sql
  CREATE TABLE MajorsIn(
      student CHAR(8), dept VARCHAR(30),
      PRIMARY KEY (student, dept),
      FOREIGN KEY (student) REFERENCES Student(id),
      FOREIGN KEY (dept) REFERENCES Department(name)
  );
  ```
Other Terms Used to Express Constraints

- **UNIQUE**: used to specify attribute(s) that form a (non-primary) key

  ```sql
  CREATE TABLE Course(name CHAR(8) PRIMARY KEY,
    start_time TIME, end_time TIME, room CHAR(4),
    UNIQUE (start_time, end_time, room));
  ```

  unlike a primary-key attribute, a unique attribute may have a null value

- **NOT NULL**: used to specify that an attribute can never be null

  ```sql
  CREATE TABLE Student(id CHAR(8) PRIMARY KEY,
    name VARCHAR(30) NOT NULL);
  ```

DROP TABLE

- **What it does**: removes an entire relation from a database
  - including all of its existing rows

- **Syntax**:
  ```sql
  DROP TABLE relation_name;
  ```

- **Example**:
  ```sql
  DROP TABLE MajorsIn;
  ```
**INSERT**

- **What it does:** adds a tuple to a relation

- **Syntax:**
  
  ```plaintext
  INSERT INTO relation VALUES (val1, val2, ...);
  ```

  - the attribute values must be given in the order in which the attributes were specified when the table was created

- **Alternate syntax:**
  
  ```plaintext
  INSERT INTO relation (attr1, attr2, ...)
  VALUES (val1, val2, ...);
  ```

  - allows you to specify values of the attributes in a different order, or for only a subset of the attributes

---

**INSERT** (cont.)

- **Examples:**
  
  ```plaintext
  INSERT INTO MajorsIn VALUES ('10005000', 'math');
  ```

  [Recall the CREATE TABLE command:
  
  ```sql
  CREATE TABLE MajorsIn(
      student CHAR(8), dept VARCHAR(30),...);
  ```
  ]

  ```plaintext
  INSERT INTO MajorsIn(dept, student)
  VALUES ('math', '10005000');
  ```
• The DBMS checks to make sure that an INSERT statement is consistent with the constraints specified for the relation.

• Example: the *MajorsIn* relation was created as follows:

```sql
CREATE TABLE MajorsIn(
    student CHAR(8), dept VARCHAR(30),
    PRIMARY KEY (student, dept),
    FOREIGN KEY (student) REFERENCES Student(id),
    FOREIGN KEY (dept) REFERENCES Department(name)
);
```

Given the *Student* relation at left, would the following command be allowed?

```sql
INSERT INTO MajorsIn
VALUES ('98765432', 'math');
```

---

### Mathematical Foundation: Cartesian Product

• Let: A be the set of values \{ a_1, a_2, ... \}
  B be the set of values \{ b_1, b_2, ... \}
  C be the set of values \{ c_1, c_2, ... \}

• The **Cartesian product** of A and B (written $A \times B$) is the set of all possible ordered pairs $(a_i, b_j)$, where $a_i \in A$ and $b_j \in B$.

• Example:
  \[
  A = \{ \text{apple, pear, orange} \} \\
  B = \{ \text{cat, dog} \} \\
  A \times B = \{ (\text{apple, cat}), (\text{apple, dog}), (\text{pear, cat}), (\text{pear, dog}), (\text{orange, cat}), (\text{orange, dog}) \}
  \]

• Example:
  \[
  C = \{ 5, 10 \} \\
  D = \{ 2, 4 \} \\
  C \times D = ?
  \]
Mathematical Foundation: Cartesian Product (cont.)

- We can also take the Cartesian product of three or more sets.

- \( A \times B \times C \) is the set of all possible ordered triples \((a_i, b_j, c_k)\), where \( a_i \in A \), \( b_j \in B \), and \( c_k \in C \).

  - example:
    
    \[
    C = \{ 5, 10 \} \\
    D = \{ 2, 4 \} \\
    E = \{ "hi", "there" \}
    \]
    
    \[
    C \times D \times E = \{ (5, 2, "hi"), (5, 2, "there"), (5, 4, "hi"), (5, 4, "there"), (10, 2, "hi"), (10, 2, "there"), (10, 4, "hi"), (10, 4, "there") \}
    \]

- \( A_1 \times A_2 \times \ldots \times A_n \) is the set of all possible ordered tuples \((a_{1i}, a_{2j}, \ldots, a_{nk})\), where \( a_{di} \in A_d \).

---

Cartesian Product of Relations

- The Cartesian product of two or more relations forms all possible combinations of rows from the relations.

- The resulting set of combinations is itself a relation.
  - its rows include the attributes/columns from all of the combined relations

  - example:
    
    Enrolled\((\text{student, course, credit\_status})\)
    
    Course\((\text{name, start\_time, end\_time, room})\)
    
    Enrolled \times Course has rows that look like this:
    
    student, course, credit\_status, name, start\_time, end\_time, room
The Cartesian Product of Two Relations (cont.)

- **Example:**

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
<th>MajorsIn student</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrolled x MajorsIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>student</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>25252525</td>
</tr>
</tbody>
</table>

---

**SELECT (joining multiple tables)**

- We've already seen one way to form a query involving more than one table – using a subquery.

  - example:
    ```
    SELECT name
    FROM Student
    WHERE id IN (SELECT student
                  FROM Enrolled
                  WHERE course = 'CS 105');
    ```

  - limitation: all columns in the resulting relation must come from the same table.

- If we need columns from more than one table, we put all of the necessary tables in the FROM clause:

  ```
  SELECT column1, column2, ...
  FROM table1, table2, ...
  ```
**SELECT** (joining multiple tables) (cont.)

```
SELECT column1, column2, ...
FROM table1, table2, ...
... 
```

- When the `FROM` clause specifies multiple tables, the resulting operation is known as a *join*.

- The result is equivalent to:
  - forming the Cartesian product of the tables in the `FROM` clause
    
    table1 x table2 x ... 

  - applying the remaining clauses to the Cartesian product, in the same order as for a single-table command:
    
    WHERE
    GROUP BY
    HAVING
    SELECT
    ORDER BY

---

**SELECT** (joining multiple tables) (cont.)

- Example: find the major of the student Alan Turing.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>35252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

- Here's a query that will give us the result that we want:

  ```
  SELECT dept
  FROM Student, MajorsIn
  WHERE name = 'Alan Turing'
  AND id = student;
  ```

- `id = student` is a *join condition* – a condition that is used to match up "related" tuples from the two tables.
  - it selects the tuples in the Cartesian product that "make sense"
  - for N tables, you typically need N – 1 join conditions
### Student Majors

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

### Student x Majors

#### SELECT dept
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>
SELECT dept
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

After selecting only tuples that satisfy the WHERE clause:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
</tbody>
</table>

After extracting the attribute specified in the SELECT clause:

<table>
<thead>
<tr>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp sci</td>
</tr>
</tbody>
</table>

Other Examples

• Given these relations:
  Student(id, name)
  Enrolled(student, course, credit_status)
  MajorsIn(student, dept)

• Find the names of all students enrolled in CS 105 with non-credit status:

• Find the names of all students enrolled in any course:
Avoiding Ambiguous Column Names

- To avoid ambiguity, we can include the table name when specifying the name of the column.

- Example: find the name and credit status of all students enrolled in CS 105 who are majoring in computer science:
  
  ```sql
  SELECT name, credit_status
  FROM Student, Enrolled, MajorsIn
  WHERE id = Enrolled.student
  AND Enrolled.student = MajorsIn.student
  AND course = 'CS 105' AND dept = 'comp sci';
  ```

- This way of specifying columns can also be used to make a query easier to understand.

- Note: the query above involves three tables, and thus it needs two join conditions. What are they?

Renaming Columns or Tables

- We can rename a column or table using the keyword AS.

- Example:
  
  ```sql
  SELECT name AS student, credit_status
  FROM Student, Enrolled AS E, MajorsIn AS M
  WHERE id = E.student
  AND E.student = M.student
  AND course = 'CS 105' AND dept = 'comp sci';
  ```

- Renaming is necessary when taking the Cartesian product of a table with itself:
  
  ```sql
  SELECT E1.name
  FROM Employee AS E1, Employee AS E2
  WHERE E1.supervisor = E2.id
  AND E2.name = 'Teresa Lopes';
  ```
More Practice with Joins

- Let's say we want the names of all rooms in which one or more CS courses meet.

<table>
<thead>
<tr>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>start_time</td>
</tr>
<tr>
<td>CS 105</td>
<td>13:00:00</td>
</tr>
<tr>
<td>CS 111</td>
<td>09:30:00</td>
</tr>
<tr>
<td>EN 101</td>
<td>11:00:00</td>
</tr>
<tr>
<td>CS 460</td>
<td>16:00:00</td>
</tr>
<tr>
<td>CS 510</td>
<td>12:00:00</td>
</tr>
<tr>
<td>PH 101</td>
<td>14:30:00</td>
</tr>
</tbody>
</table>

- What query should we use?

Joins and Unmatched Rows

- Let's say we want the IDs and majors of everyone enrolled in a course.

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

- We begin by trying a standard join:

```sql
SELECT DISTINCT Enrolled.student, dept
FROM Enrolled, MajorsIn
WHERE Enrolled.student = MajorsIn.student;
```

- Why isn't this sufficient?
Outer Joins

- Outer joins allow us to include unmatched rows in the result.

- We will focus on left outer joins, with the following syntax:
  ```sql
  SELECT ... 
  FROM T1 LEFT OUTER JOIN T2 ON join condition 
  WHERE ... 
  ```

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the tuples in the Cartesian product that satisfy the join condition in the ON clause
  - including an extra tuple for each row from T1 that does not have a match with a row from T2
    - the T2 attributes in the extra tuples are given null values
  - applying the remaining clauses as before

Outer Joins (cont.)

- We can use a left outer join to get the IDs and majors of everyone enrolled in a course -- including those with no major.

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>CS 105</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>CS 111</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 460</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>CS 105</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>CS 510</td>
<td>grad</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

```
SELECT DISTINCT Enrolled.student, dept 
FROM Enrolled LEFT OUTER JOIN MajorsIn 
ON Enrolled.student = MajorsIn.student;
```

- Note: we don't need a WHERE clause in this case, because the join condition is now in the ON clause.

- Additional selection conditions would go in the WHERE clause.
Outer Joins (cont.)

• Another example: find the IDs and majors of all students enrolled in CS 105 (including those with no major):

```sql
SELECT Enrolled.student, dept
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student = MajorsIn.student
WHERE course = 'CS 105';
```

• in this case, there is a WHERE clause with an additional selection condition
• the additional condition belongs in the WHERE clause because it's not a join condition – i.e., it isn't used to match up tuples from the two tables

Writing Queries: Rules of Thumb

• Start with the FROM clause. Which table(s) do you need?

• If you need more than one table, determine the necessary join conditions.
  • for N tables, you typically need N – 1 join conditions
  • is an outer join is needed – i.e., do you need to include unmatched tuples?

• Determine if a GROUP BY clause is needed.
  • are you performing computations involving subgroups?

• Determine any other conditions that are needed.
  • if they rely on aggregate functions, put in a HAVING clause
  • otherwise, add to the WHERE clause
  • is a subquery needed?

• Fill in the rest of the query: SELECT, ORDER BY?
Commands for Database Modifications

- **DELETE**: remove one or more tuples from a relation
  - basic syntax:
    
    ```sql
    DELETE FROM table
    WHERE selection condition;
    ```
  - example:
    
    ```sql
    DELETE FROM Student
    WHERE id = '10005000';
    ```

Commands for Database Modifications (cont.)

- **UPDATE**: modify attributes of one or more tuples in a relation
  - basic syntax:
    
    ```sql
    UPDATE table
    SET list of assignments
    WHERE selection condition;
    ```
  - examples:
    
    ```sql
    UPDATE MajorsIn
    SET dept = 'physics'
    WHERE student = '10005000';
    ```
    ```sql
    UPDATE Course
    SET start_time = '11:00:00', end_time='12:00:00',
    room = 'STO 143'
    WHERE name = 'CS 105';
    ```
Practice Writing Queries

Student(id, name)  Department(name, office)  Room(id, name, capacity)
Course(name, start_time, end_time, room)  MajorsIn(student, dept)
Enrolled(student, course, credit_status)

1) Find all rooms that can seat at least 100 people.

2) Find the course or courses with the earliest start time.

Practice Writing Queries (cont.)

Student(id, name)  Department(name, office)  Room(id, name, capacity)
Course(name, start_time, end_time, room)  MajorsIn(student, dept)
Enrolled(student, course, credit_status)

3) Find the number of majoring students in each department.

4) Find all courses taken by CS ('comp sci') majors.
5) Create a list of all Students who are not enrolled in a course.

Why won't this work?

```sql
SELECT name
FROM Student, Enrolled
WHERE Student.id != Enrolled.student;
```

6) Find the number of CS majors enrolled in CS 105.

```sql
-- SQL query to find the number of CS majors enrolled in CS 105
```
Practice Writing Queries (cont.)

Student(id, name)      Department(name, office)      Room(id, name, capacity)
Course(name, start_time, end_time, room) MajorsIn(student, dept)
Enrolled(student, course, credit_status)

7) Find the number of majors that each student has declared.

Practice Writing Queries (cont.)

Student(id, name)      Department(name, office)      Room(id, name, capacity)
Course(name, start_time, end_time, room) MajorsIn(student, dept)
Enrolled(student, course, credit_status)

8) For each department with more than one majoring student, output the department's name and the number of majoring students.
Practice Writing Queries (cont.)

Student(id, name)      Department(name, office)      Room(id, name, capacity)
Course(name, start_time, end_time, room)     MajorsIn(student, dept)
Enrolled(student, course, credit_status)

9) Find the names of all students who have a course in GCB 204.
Extra Practice Writing Queries

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

1) Find all people in the database who acted in Avatar.

Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

2) How many people in the database did not act in Avatar?
Will this work?

```
SELECT COUNT(*)
FROM Person P, Actor A, Movie M
WHERE P.id = A.person_id AND M.id = A.movie_id
    AND M.name != 'Avatar';
```

If not, what will?
Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

3) How many people in the database who were born in California have won an Oscar?

Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

4) Which people in the database have acted in a movie directed by James Cameron?
Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

5) Which movie ratings have an average runtime that is greater than 120 minutes?

Extra Practice Writing Queries (cont.)

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)  Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

6) For each person in the database born in Boston, find the number of movies in the database (possibly 0) in which the person has acted.
SkyServer: An Astronomical Database

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

Sloan Digital Sky Survey (SDSS)

- A database of astronomical data (10s of terabytes)
  - five-year survey mapping 1/4 of the night sky
  - information on 100s of millions of objects
    - stars
    - quasars
    - galaxies
  - a collaboration involving scientists from many countries and institutions
- Part of the International Virtual Observatory
Types of Data in SDSS

- Types of data:
  - images
  - *spectrograms*: graphs of the amount of light emitted by an object at different wavelengths
    - can be used to determine:
      - object type
      - temperature
      - chemical composition
      - distance from Earth
  - *photometric data*: info. measured or computed from images
    - brightness, size, etc.
  - *spectrographic data*: info. measured or computed from spectrograms
    - redshift, spectral type, etc.

SkyServer

- Provides public access to SDSS data
  [http://skyserver.sdss.org](http://skyserver.sdss.org)

- Includes some good overviews of the relevant astronomy

- It also includes a useful set of tools
  - famous places
  - navigate: pan and zoom through the sky
  - search forms: use menus to specify what you’re looking for
  - SQL search: enter an arbitrary query in SQL
The SDSS Database

- A relational database on Microsoft SQLServer
  - Jim Gray of Microsoft Research worked on the implementation

- The schema was devised in part by:
  - coming up with 20 queries that astronomers might pose
  - refining the schema until these queries could be easily answered
  - example questions:
    - find all objects classified as galaxies in a specified region of the night sky
    - find all asteroids in the database
      - why might this be hard?

SDSS Database Schema

- Three main groups of relations:
  - photo: imaging data
  - spectro: spectroscopic data
  - meta: documentation and metadata (data about the data)

- Can use the schema browser to get info.:
Key Relations in the Photo Group

• PhotoObjAll: photometric data for all objects
• Neighbors: all pairs of objects that are "near" each other
• PhotoTag: contains copies of the most important columns from PhotoObjAll
  • how could using this table speed up queries?
  • why is this table potentially problematic?

Views of a Database

• A database view is like a virtual table.
• Rows in a database view may contain:
  • attributes drawn from different tables
  • attributes computed from the attributes stored in the tables
  • only some of the rows in a given table
• Why could views be helpful?
Views Provide Another Layer of Abstraction

views (virtual tables)

<table>
<thead>
<tr>
<th>name</th>
<th>class</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jill Jones</td>
<td>2007</td>
<td>21</td>
</tr>
<tr>
<td>Alan Turing</td>
<td>2010</td>
<td>19</td>
</tr>
</tbody>
</table>

logical representation

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>class</th>
<th>dob</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>Warren Towers 100</td>
<td>2007</td>
<td>3/10/85</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>Student Village A210</td>
<td>2010</td>
<td>2/7/88</td>
</tr>
</tbody>
</table>

physical representation

disks

Views in the Photo Group

- The photo group includes views that make it easier to limit our queries to certain types of objects.
  - stars, galaxies, etc.

- Views enable simpler queries:
  ```sql
  SELECT objID, ...
  FROM Galaxy
  WHERE ... 
  ```
  instead of:
  ```sql
  SELECT objID, ...
  FROM PhotoObjAll
  WHERE type = GALAXY AND ...
  ```

- Views don’t speed up queries.
  - the SELECT command still accesses the full table(s), and it performs the same operations
Example Query: Find All Galaxies

- Find all objects classified as galaxies within a 1-arcmin radius of the point with coordinates (185, -0.5).
  
  ```sql
  SELECT G.objID, N.distance
  FROM Galaxy AS G,
       dbo.fGetNearbyObjEq(185, -0.5, 1) AS N
  WHERE G.objID = N.objID
  ORDER BY N.distance
  ```

- Note: `dbo.fGetNearbyObjEq()` is a function that returns a table of all objects within a specified radius of a specified point.
  - if we didn't have this function, what could we use instead?

---

Seeing Images of Query Results

- To see images of objects obtained, we can use the image-list tool:

- It takes (name, ra, dec) tuples, where name is an identifier of our choosing.

- We can use a query to generate the list of requested images.

- Here's a modified version of our earlier query:
  ```sql
  SELECT G.objID AS name, G.ra, G.dec
  FROM Galaxy AS G,
       dbo.fGetNearbyObjEq(185, -0.5, 1) AS N
  WHERE G.objID = N.objID
  ORDER BY N.distance
  ```
  - do not add a semi-colon!
Finding Moving Objects

- For a given portion of the night sky, SDSS takes 5 images, one for each of five color filters.
  - red (r), green (g), ultraviolet (u), infrared (i and z)
  - taken over a 5-minute interval

- Most of the objects in the images (stars, galaxies, etc.) are far enough away that they don't appear to move.

- The movement of closer objects like asteroids is detectable.
  - they can show up as separate "dots" when the three images (r, i, g) used to make the color images are superimposed

  example asteroid:
  note: the colors used to make the images don't correspond to the colors of the filters

"Slow-Moving" Objects

- If an object moves slowly enough, the program that processes the raw image data is able to recognize that the different measurements belong to a single object.
  - this includes most asteroids

- In such cases, the movement is recorded in two velocity attributes: rowv and colv
  - total velocity = \sqrt{(rowv^2 + colv^2)}

- To find an object that might be a slow-moving asteroid, check for objects with a total velocity that is high enough.
"Fast-Moving" Objects

- Fast-moving objects appear as streaks:

- These aren't recognized as a single object, so a query to find them needs to examine objects that are close to each other.

```
select r.objID as rid, g.objID as gid,
   r.run, r.camcol,
   r.fieldID as field, g.fieldID as gfield,
   r.iz as iz, r.dec as dec r,
   g.iz as giz, g.dec as dec g,
   cosine(r.iz-giz, r.dec-g.dec) as cosang,
   power(r.iz-giz,2)+power(r.dec-g.dec,2) as distsq
from PhotoObj r, PhotoObj g
where r.run = g.run and r.camcol=g.camcol -- same run and camera column
and abs(r.field-g.field) <= 1 -- adjacent fields
   -- the raw selection criteria
and (power(r.iz,2) + power(r.dec,2) > 1.111111) -- g/s as ellipticity
and abs(r.fibermag-g.fibermag) < 0.01 and abs(r.fibermag-g.fibermag) < 0.01
and abs(r.fibermag-g.fibermag) < 0.01
and abs(r.parentID-g.parentID) = 1
and r.iz < 1.0 and g.iz < 1.0
   -- the green selection criteria
and (power(g.iz,2) + power(g.fibermag,2) > 0.111111)
and g.fibermag < g.fibermag
and g.fibermag < g.fibermag
and g.fibermag < g.fibermag
and g.fibermag < g.fibermag
and g.fibermag < g.fibermag
   and g.parentID < 1
and g.iz < 1.0
   -- the match-up of the pair -- note cosang(x) ~ x for x<1
and abs(r.fibermag-g.fibermag) < 0.01
and abs(r.fibermag-g.fibermag) < 0.01
```

"Fast-Moving" Objects (cont.)

- Here's a query by Szalay, Gray, et al. for fast-moving asteroids:
"Fast-Moving" Objects (cont.)

• Despite the complexity of the prior query, it's simpler than previous approaches to finding such asteroids.
  • a colleague of Gray and Szalay wrote a 12-page program that operated on the image data itself
    • it took 3 days to execute
  • it took about a day for G&S to develop the SQL query
    • it executed in about 10 minutes

Being able to pose questions in a few hours and get answers in a few minutes changes the way one views the data: you can experiment with it almost interactively.
When queries take 3 days and hundreds of lines of code, one asks questions cautiously.

– Gray and Szalay

References


Beyond Relational Databases

- While relational databases are extremely powerful, they may be inadequate/insufficient for a given problem.

- Example 1: DNA sequence data

  ```
  AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTA
  AATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAA
  ACGCCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGA
  CAGTGCGGGCTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTC
  TGCGTGTTGCCGATATTCTGGAAAGCAATGCCAGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTGGCGAT
  GATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAACGTATTTTTGCCGAACTTTTGACGGGACTCGCCGCCGCCCAG
  CCGGGGTTCCCGCTGGCGCAA
  ```

- common queries involve looking for similarities or patterns
  - what genes in mice are similar to genes in humans?
- need special *algorithms* (problem-solving procedures) for finding statistically significant similarities
- biologists store this data in text files and use computer programs to process it
Beyond Relational Databases (cont.)

• Example 2: data mining – the process of finding patterns in data
  • here again, special algorithms are needed
  • typical process:
    • extract data from a DBMS
    • use a separate program to apply the necessary algorithms

Other Reasons for Writing Programs

• To create a simple database application.

  • example: a program known as a CGI script that:
    • takes values entered into a form on a Web page
    • creates a query based on those values and submits it to a DBMS
    • generates a Web page to present the results

(Figure 1-15 of Kroenke)
Other Reasons for Writing Programs (cont.)

• To transform data in some way.
  • example: when an attribute has a large number of possible values, it's often necessary to divide them into subranges of values called bins.
  • example bins for an age attribute:
    - child: 0-12
    - teen: 12-17
    - young: 18-35
    - middle: 36-59
    - senior: 60-
  • use a simple program to replace the actual values with the corresponding bin names/numbers
    - 15 → teen
    - 6 → child
    - 40 → middle

Algorithms

• In order to solve a problem using a computer, you need to come up with one or more algorithms.

• An algorithm is a step-by-step description of how to accomplish a task.

• An algorithm must be:
  • precise: specified in a clear and unambiguous way
  • effective: capable of being carried out

"It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm."

Don Knuth
Is This An Algorithm?

• Recipe for preparing a meat roast:

  Sprinkle the roast with salt and pepper. Insert a meat thermometer and place in oven preheated to 150 degrees C. Cook until the thermometer registers 80-85 degrees C. Serve roast with gravy prepared from either meat stock or from pan drippings if there is sufficient amount.

  (taken from a book on programming by Pohl and McDowell)

Here’s the Algorithm…

• Recipe for preparing a meat roast:

  1. Sprinkle roast with 1/8 teaspoon salt and pepper.
  2. Turn oven on to 150 degrees C.
  3. Insert meat thermometer into center of roast.
  4. Wait a few minutes.
  5. If oven does not yet register 150 degrees, return to step 4.
  6. Place roast in oven.
  7. Wait a few minutes.
  8. Check meat thermometer. If temperature is less than 80 degrees C, go back to step 7.
  9. Remove roast from oven.
  10. If there is at least ½ cup of pan drippings, go to step 12.
  11. Prepare gravy from meat stock and go to step 13.
  12. Prepare gravy from pan drippings.
  13. Serve roast with gravy.

  (also from Pohl and McDowell)
Python

• We'll learn to write programs in Python.

• Python is a freely available language that makes it easy to write small- and medium-sized programs.

• You can download it from https://www.python.org/download/
  • use any version of Python 3

• Python is also available on the lab machines.
Example Problem: Adding Up Your Change

- Let's say that we have a bunch of coins of various types, and we want to figure out how much money we have.
- Let's begin the process of developing a program that does this.

Step 1: Analysis and Specification

- Analyze the problem (making sure that you understand it), and specify the problem requirements clearly and unambiguously.
- Describe exactly what the program will do, without worrying about how it will do it.
- Ask questions like the following:
  - what are the inputs to the program?
  - what are the desired outputs?
  - what needs to be done to go from the inputs to the outputs?
Step 2: Design

- Determine the necessary algorithms (and possibly other aspects of the program) and sketch out a design for them.

- This is where we figure out how the program will solve the problem.

- Algorithms are often designed using pseudocode.
  - more informal than an actual programming language
  - allows us to avoid worrying about the syntax of the language
  - example for our change-adder problem:
    
    ```
    get the number of quarters
    get the number of dimes
    get the number of nickels
    get the number of pennies
    compute the total value of the coins
    output the total value
    ```

Step 3: Implementation

- Translate your design into the programming language.
  pseudocode → code

- We need to learn more Python before we can do this!

- Here’s one possible implementation in Python (we’ll look at it in more detail later):

  ```python
  quarters = eval(input("number of quarters? "))
  dimes = eval(input("number of dimes? "))
  nickels = eval(input("number of nickels? "))
  pennies = eval(input("number of pennies? "))

  cents = quarters*25 + dimes*10 + nickels*5 + pennies
  print("you have", cents, "cents")
  ```
Step 4: Testing and Debugging

- A **bug** is an error in your program.
- **Debugging** involves finding and fixing the bugs.

The first program bug! Found by Grace Murray Hopper at Harvard. (http://www.hopper.navy.mil/grace/grace.htm)

- Testing – trying the programs on a variety of inputs – helps us to find the bugs.

Overview of the Programming Process

- Analysis/Specification
- Design
- Implementation
- Testing/Debugging
Types of Programming Languages

• Python is an example of a *high-level language*.
  • others include: Java, C, C++, Fortran, Perl, and LISP

• They are designed to make it easier for programmers to write and read programs.

• However, a computer *cannot* run a high-level program directly.

• Instead, each type of CPU has its own *machine language*—its own set of low-level instructions that it understands.
  • they perform simple operations like adding two numbers
  • the instructions are written as binary numbers
    00000010100100011000000000100000

Converting from High to Low

• In Python, we use an *interpreter*, a program that takes Python code and executes it.

  ```
  i = 2
  j = i + 1
  print(i)
  print(k)
  ```

• The interpreter itself has been converted to machine language and is run directly by the CPU.

• Most interpreters – including the one for Python – allow you to execute commands from the keyboard, one at a time.

• Using an interpreter makes it easier to develop programs quickly.
Interacting with the Python Interpreter

• We will use an integrated development environment (IDE) – an application that helps you develop programs.

• The standard Python IDE is called IDLE
  • after Monty Python member Eric Idle!

• When you start IDLE, you get the interpreter in a window.
  • also known as the Python Shell

![Image of Python IDE](image)

Interacting with the Python Interpreter (cont.)

• The following prompt indicates that the interpreter is waiting for you to type a command:

  >>>

• When you type a command and hit the enter key, the interpreter executes the command.

  • examples:

    >>> print('Hello world!')
    Hello world!
    >>> print(5 + 10)
    15
    >>> print(2 ** 10)
    1024
    >>> print(2 ** 100)
    "1267650600228229401496703205376"
Creating a Reusable Program

• To create programs that can be reused, we put the commands in a text file.
  • the resulting file is often called a *module* or *script*

• We can use any text editor to write a program.

• We'll typically use the one that comes with IDLE.
  • choose the *File-*>*New* menu option to start a new program file
  • choose the *File-*>*Open* menu option to open an existing program file

• Program files should be saved using a filename that has the extension *.py*
  • example:  *myProgram.py*

Running a Program

• We can tell the interpreter to run a program stored in a file.

• In IDLE, we can use the *Run -*> *Run Module* menu option.
Our Change-Adder Program

```python
# changeAdder.py
# a program to determine the value
# of a collection of coins
# Dave Sullivan

quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))

cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")    # print result
```

- We save this in a file named `changeAdder.py`

- Let's use this program to examine some of the fundamental building blocks of a Python program.

Comments

```python
# changeAdder.py
# a program to determine the value
# of a collection of coins
# Dave Sullivan

quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))

cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")    # print result
```

- Comments are text that appears after a `#` symbol.
  - used to make programs more readable
  - they are *not* executed
  - Python ignores everything from a `#` to the end of the line.
Literals

```python
quarters = eval(input("number of quarters? ""))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")
```

- **Literals** specify a particular value.

- They include:
  - numeric literals: 25 3.1416
  - commas are not allowed!
  - string literals: "you have" "a" 'hello!'
  - can be surrounded by either single or double quotes

Identifiers

```python
quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")
```

- **Identifiers** are words that are used to name components of a Python program.

- They include:
  - variables, which give a name to a value
    ```python
    quarters
dimes
nickels
pennies
cents
    ```
  - function names like `eval`, `input` and `print`
Identifiers (cont.)

• Rules:
  • must begin with a letter or _
  • can be followed by any number of letters, numbers, or _
  • spaces are not allowed
  • cannot be the same as a keyword – a word that is reserved by the language for its own use

• Which of these are not valid identifiers?
  n1 num_values 2n
  avgSalary course name

• Python is case-sensitive (for both identifiers and keywords).
  • example: quarters is not the same as Quarters

Expressions

```python
quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")
```

• Expressions are pieces of code that evaluate to a value.

• They include:
  • literals, which evaluate to themselves
  • variables, which evaluate to the value that they represent
  • combinations of literals, variables, and operators:
    quarters*25 + dimes*10 + nickels*5 + pennies
Expressions (cont.)

• Numerical operators include:
  + addition
  - subtraction
  * multiplication
  / division
  ** exponentiation
  % modulus: gives the remainder of a division
  example: 11 % 3 evaluates to 2

• They are evaluated according to the standard order of operations.
  • example: multiplication before addition
    quarters*25 + dimes*10 + nickels*5 + pennies

• You can use parentheses to force a particular evaluation order.
  • example: 3 * (4 + 5)

Expressions (cont.)

• If you enter an expression in the shell, Python evaluates it and displays the result.
  • examples from the Python shell:
    >>> 'Hello world!'
    'Hello world!'
    >>> 5 + 10
    15
    >>> 2 ** 10
    1024

• Note that it isn't necessary to use print when evaluating an expression from the shell.

• However, if we're running a module, we do need to use print if we want a value to be displayed.
Evaluating Expressions with Variables

• When an expression includes variables, they are first replaced with their current value.

• Example: consider this code fragment:

```
quarters = 10
dimes = 3
nickels = 7
pennies = 6
cents = 25*quarters + 10*dimes + 5*nickels + pennies
    = 25* 10 + 10* 3 + 5* 7 + 6
    = 250 + 10* 3 + 5* 7 + 6
    = 250 + 30 + 5* 7 + 6
    = 250 + 30 + 35 + 6
    = 280 + 35 + 6
    = 315 + 6
    = 321
```

Statements

```
quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print("you have", cents, "cents")
```

• A statement is an individual command in a program.

• In Python, statements often fit on a single line.
  • the end of the line signals the end of the statement
  • no special terminating character (e.g., a semi-colon) is needed
Print Statements

- print statements display one or more values on the screen
  - this is sometimes referred to as console output
  - console = screen and keyboard

- Basic syntax:
  ```python
  print(<expr>)
  ```
  or
  ```python
  print(<expr1>, <expr2>, ..., <exprn>)
  ```
  where each <expr> is an expression

- Steps taken when executed:
  1) the individual expressions are evaluated
  2) the resulting values are displayed on the same line, separated by spaces

- To print a blank line, omit the expressions: `print()`

Print Statements (cont.)

- Examples:
  - first example:
    ```python
    print('the results are:', 15 + 5, 15 - 5)
    ```
    ```
    'the results are:' 20 10
    ```
    output: the results are: 20 10
    (note that the quotes around the string literal are not printed)

  - example from changeAdder.py (assume a total of 321 cents)
    ```python
    print('you have', cents, 'cents')
    ```
    ```
    'you have' 321 'cents'
    ```
    output: you have 321 cents
Functions

- Python comes with a number of built-in functions that we can use in our programs.

- `print` is one example of a built-in function

- A function may take one or more parameters – values that serve as inputs to the function
  - for `print`, the parameters are the expressions whose values you want to print

- To execute or call a function, you write its name, followed by parentheses that contain any parameters, separated by commas:

  ```python
  print("you have", cents, "cents")
  ```

Functions (cont.)

- Some functions also return (i.e., output) a value.

- Example: the `abs` function
  - parameter (input): a number `n`
  - return value (output): the absolute value of `n`
  - examples from the Python shell:
    ```python
    >>> abs(-20)
    20
    >>> abs(35)
    35
    ```

- If a function returns a value, then a call to that function is a type of expression.
  - it evaluates to the value returned by the function
  - example: `abs(-10)` evaluates to 10
Combining Functions

- We often need to combine functions in our programs.
- Example: to print the absolute value of a number
  
  ```python
  num = -15
  print("The absolute value is: ", abs(num))
  ```

  - the value returned by `abs` is one of the parameters of `print`

  - In such cases, the statement is executed "from the inside out":
    
    ```python
    print("The absolute value is: ", abs(num))
    ```

    ```python
    abs(-15)
    ```

    ```python
    print("The absolute value is: ", 15)
    ```

    output: `The absolute value is: 15`

Assignment Statements

- We use an assignment statement to give a value to a variable.
- Syntax:
  
  ```python
  <variable> = <expression>
  ```

  `<variable>` is a valid identifier. `=` is the `assignment operator`; it does not compare the values!

- Examples:
  
  ```python
  pi = 3.1416
  x = 10 + 5
  y = x
  message = 'Please enter your name: '
  ```

  - It is an error to use a variable before assigning it a value.
### Assignment Statements (cont.)

- **Steps in executing an assignment statement:**
  1. evaluate the expression on the right-hand side of the `=`
  2. assign the resulting value to the variable on the left-hand side of the `=`

- **Examples:**
  ```
  quarters = 10  # 10 evaluates to itself!
  quarters = 10
  quartersValue = 25 * quarters
  quartersValue = 25 * 10
  quartersValue = 250
  ```

### Assignment Statements (cont.)

- **An assignment statement does not create a permanent relationship between variables.**

- **Example from the shell:**
  ```
  >>> x = 10
  >>> y = x + 2
  >>> y
  12
  >>> x = 20
  >>> y
  12
  ```
  - changing the value of `x` does *not* change the value of `y`!

- **You can only change the value of a variable by assigning it a new value.**
Assignment Statements (cont.)

• As the values of variables change, it can be helpful to picture what’s happening in memory.

• Examples:

<table>
<thead>
<tr>
<th>num1 = 100</th>
<th>num1</th>
<th>100</th>
<th>num2 = 120</th>
<th>num2</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>num1 = 50</td>
<td>num1</td>
<td>50</td>
<td>num2 = 120</td>
<td>num2</td>
<td>120</td>
</tr>
<tr>
<td>num1 = num2 * 2</td>
<td>120 * 2</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>num2 = 60</td>
<td>num1</td>
<td>240</td>
<td>num2 = 60</td>
<td>num2</td>
<td>60</td>
</tr>
</tbody>
</table>

The value of num1 is unchanged!

Assignment Statements (cont.)

• A variable can appear on both sides of the assignment operator!

• Example (fill in the missing values):

<table>
<thead>
<tr>
<th>sum = 13</th>
<th>sum</th>
<th>13</th>
<th>val = 30</th>
<th>val</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum = sum + val</td>
<td>sum</td>
<td></td>
<td>val</td>
<td></td>
<td></td>
</tr>
<tr>
<td>val = val * 2</td>
<td>sum</td>
<td></td>
<td>val</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Input Function

• The `input` function allows us to get values from the user.
  • optional parameter: a string that serves as a prompt
    ```python
    input("What is your name? ")
    ```
  • return value: the `string` entered by the user

• When the `input` function is called, it:
  • prints the prompt
  • waits for the user to type 0 or more characters, followed by the `Enter` key
  • returns a string containing those characters

• Typically, we use the input function as part of an assignment:
  ```python
  name = input("What is your name? ")
  ```

Getting Numeric Input

• The `input` function *always* returns a string, regardless of whether the user enters letters or numbers.
  • example: if the user enters 17, `input` will return "17"

• To get a number from the user, we can combine the `input` function with another function called `eval`.
  • example from `changeAdder.py`:
    ```python
    quarters = eval(input("number of quarters? "))
    ```

• `eval` function:
  • parameter: a string representing an expression
  • return value: the value of that expression
  • examples:
    ```python
    eval("15") returns 15
    eval("3 + 5 - 2") returns 6
    ```
Getting Numeric Input (cont.)

- Once again, we work from the inside out:

  ```python
  quarters = eval(input("number of quarters? "))
  eval("17")
  17
  ```

Summary: Input Statements

- Getting a **string value** from the user:

  ```python
  <variable> = input(<prompt>)
  ```

  where `<prompt>` is a string

- Getting a **numeric value** from the user:

  ```python
  <variable> = eval(input(<prompt>))
  ```

  where `<prompt>` is a string

- Note: it looks better if you put a space at the end of the prompt.
  
  - example: if we changed the line from changeAdder.py to omit the space

  ```python
  quarters = eval(input("number of quarters?"))
  ```

  we will get the following when the user enters a value

  ```text
  number of quarters?17
  ```
Review

- Consider the following code fragments
  1) 1000
  2) 10 ** 5
  3) print('Hello')
  4) hello
  5) num1 = 5
  6) 2*width + 2*length
  7) prompt = 'student id number: '
  8) age = eval(input('student age: '))
  9) num_values

- Which of them are examples of:
  - literals?
  - identifiers?
  - expressions?
  - statements?

for Statements

- A for statement allows us to repeat one or more statements.
  - example:
    ```python
    for i in [1, 2, 3]:
        print('Warning')
        print(i)
    ```
    will output:
    ```
    Warning
    1
    Warning
    2
    Warning
    3
    ```
  - A for statement is often referred to as a for loop.
  - The repeated statement(s) are known as the body of the loop.
    - they must be indented the same amount
for Statements (cont.)

- Syntax:
  ```python
  for <variable> in <sequence>:
      <body>
  ```
  where `<sequence>` is a sequence/list of values
  `<body>` is one or more indented statements

- For each value in the sequence:
  - the value is assigned to `<variable>`
  - all statements in the body of the loop are executed using that value

- Once all values in the sequence have been processed, the program continues with the statement that comes after the loop.

---

Executing a for Loop

```python
for <variable> in <sequence>:
    <body of the loop>
```
Executing Our Earlier Example
(with one extra statement)

```python
for i in [1, 2, 3]:
    print('Warning')
    print(i)
    print('That's all. ')
```

<table>
<thead>
<tr>
<th>more?</th>
<th>i</th>
<th>output/action</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1</td>
<td>Warning</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>2</td>
<td>Warning</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>3</td>
<td>Warning</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td></td>
<td>That's all.</td>
</tr>
</tbody>
</table>

Another Example

- What would this code output?
  ```python
  for val in [2, 4, 6, 8, 10]:
      print(val * 10)
      print(val)
  ```

- Use a table to help you:
  ```
  more?  val  output/action
  ```
range Function and for Loops

- `range(n)` generates the sequence 0, 1, 2, ..., n-1

- Examples:
  - `range(4)` generates the sequence 0, 1, 2, 3
  - `range(7)` generates the sequence 0, 1, 2, 3, 4, 5, 6
  - `range(2)` generates the sequence 0, 1

- To repeat a loop’s body \(N\) times, we can use `range` to generate a sequence of \(N\) values:

  ```
  for i in range(<N>):
      <body of the loop>
  ```

  ```
  for i in range(3):
      print("Ho!")
  output: Ho!
          Ho!
          Ho!
  ```

range Function and for Loops (cont.)

- Another example:

  ```
  for i in range(7):
      print(i * 5)
  ```

  output?
Printing Separate Values on the Same Line

- By default, the `print` function puts an invisible character called a *newline character* at the end of whatever it prints.
  - causes the console to go to the beginning of the next line
  - that's why separate print statements print on separate lines.

- To get separate print statements to print on the same line, we need to replace the newline character with a space.

- Example:

  ```python
  for i in range(7):
      print(i * 5, end=" ")
  ```

  will output

  0 5 10 15 20 25 30

Using a for Loop to Compute a Sum

- The following program reads in 5 numbers from the user and computes their sum.

  ```python
  sum = 0
  for i in range(5):
      num = eval(input('enter a number: '))
      sum = sum + num

  # output the result
  print("the sum of the numbers is", sum)
  ```
Tracing a for Loop

- Let's trace through our code for the inputs 10, 20, 30, 40, 50:

```python
sum = 0
for i in range(5):  # range(5) = 0, 1, 2, 3, 4
    num = eval(input('enter a number: '))
    sum = sum + num

# output the result
print('the sum of the numbers is', sum)
```

Using a for Loop to Compute a Sum

- How could we change the program to allow the user to specify the number of values to be summed?

```python
sum = 0
for i in range(5):
    num = eval(input('enter a number: '))
    sum = sum + num

# output the result
print('the sum of the numbers is', sum)
```
Data Types in Python

- In Python and other programming languages, different kinds of data are stored and manipulated differently.

- A data type specifies:
  - a domain: a set of possible values (as in SQL)
  - a set of operations that can be performed on values of that type

- In Python, there are two main data types for numbers:
  - integers: `int`
  - numbers that can have fractional parts (floating-point numbers): `float`
    - like the REAL type in SQL
    - used whenever a value has a decimal point
Data Types in Python (cont.)

- To determine the type of a value, we can use the built-in `type()` function.
  
  ```python
  >>> type(12)
  <class 'int'>
  >>> type(3.1495)
  <class 'float'>
  >>> type(3.0)
  <class 'float'>
  >>> type(2 ** 100)
  <class 'int'>
  >>> foo = 100
  >>> type(foo)
  <class 'int'>
  >>> type(input)
  <class 'builtin_function_or_method'>
  ```

Data Types and Variables

- Variables in Python do not have a type, so it's acceptable to change the type of the value assigned to a variable.
  
  ```python
  >>> value = 3.1495
  >>> print(value)
  3.1495
  >>> value = 5280
  >>> print(value)
  5280
  >>> value = 'Hello world!
  >>> print(value)
  Hello world!
  ```
Different Data Types Have Different Operations

- Recall the list of operators for numbers:
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `**` exponentiation
  - `%` modulus: gives the remainder of a division

  Example:
  \[ 11 \% 3 \text{ evaluates to } 2 \]

- There are really two sets of operators: one set for `int` values, and one for `float` values.

- In most cases, the following rules apply:
  - if at least one of the operands is a float, the result is a float
  - if both of the operands are ints, the result is an int

Two Types of Division

- One exception to the rules on the last slide: the regular division operator `/` always produces a float result, regardless of whether the operands are ints or floats.

  Examples:
  ```python
  >>> 5 / 3
  1.6666666666666667
  >>> 6 / 3
  2.0
  >>> 11.0 / 5
  2.2
  ```

- The `/` operator is sometimes called the `float division` operator.
Two Types of Division (cont.)

- Sometimes, it's useful to perform integer division, which discards the fractional part of the result (i.e., everything after the decimal).

- In Python 3, there is a separate // operator for integer division.
  - examples:
    ```python
    >>> 5 // 3
    1
    >>> 6 // 3
    2
    >>> 11 // 5
    2
    ```

- Note that the // operator truncates anything after the decimal. It does not round.

Using the Division Operator

- Recall our change-adder program:
  ```python
  quarters = eval(input("number of quarters? "))
dimes = eval(input("number of dimes? "))
nickels = eval(input("number of nickels? "))
pennies = eval(input("number of pennies? "))
  ```
  ```python
  cents = quarters*25 + dimes*10 + nickels*5 + pennies
  print("you have", cents, "cents")
  ```

- Let's change it to print the result in dollars and cents.
  ```python
  dollars = ／_________
  cents = ／_________
  print("you have", dollars, "dollars and", cents, "cents")
  ```
Type Conversions

- There are built-in functions for converting to any numeric type:
  - `float(n)`: converts `n` to a `float`
  - `int(n)`: converts `n` to an `int`, discarding any fractional part

Examples:
```python
>>> int(8.72532)
8
>>> float(8)
8.0
>>> float(2**60)
1.152921504606847e+018
```

Type Conversions (cont.)

- Using a type-conversion function does not change the type of the value stored in memory.

  - examples:
    ```python
    >>> measurement = 3.7
    >>> int(measurement)
    3
    >>> measurement
    3.7
    ```

- How could we change the type of the value stored in memory?

  - this works because variables do not have a type
Rounding a Number

- `round(n)` rounds the number `n` to an integer:
  ```
  >>> round(8.5)
  9
  >>> round(8.49)
  8
  >>> round(2.8)
  3
  ```

- `round(n, d)` rounds the number `n` to `d` places after the decimal. If `n` is a float, it remains a float. If `n` is an int, it remains an int.
  ```
  >>> round(8.7583, 2)
  8.76
  >>> round(8.7583, 1)
  8.8
  >>> round(8, 2)
  8
  >>> round(10.595, 2)
  10.6
  (note that non-essential 0s are not displayed)
  ```

Python's Math Module

- Python comes with a math module that contains definitions for a number of mathematical functions and constants, including:
  - `sqrt(n)` computes the square root of a number
  - `sin(n), cos(n), tan(n)` trigonometric functions
  - `pi, e` constants

- To use them, we need to:
  - `import` the module
  - Prepend the name of the module (e.g., `math.sqrt(25)`)
    ```
    >>> math.sqrt(25)  # won't work before import
    ...NameError: name 'math' is not defined
    >>> import math
    >>> math.sqrt(25)
    5.0
    >>> math.pi
    3.141592653589793
    ```
Height Converter

- Let's design and write a program that reads a height in centimeters and computes:
  - the height in inches (rounded to the nearest inch)
  - the height in feet, which any fraction of a foot expressed in inches

- Example interaction:
  
  Enter your height in cm: 172
  You are 68 inches tall (5 feet, 8 inches).

- Conversion factor: 1 cm = 0.393700787 inches

Height Converter (cont.)

- Pseudocode:

- Python code:
Flow of Control

- Flow of control = order in which instructions are executed
- By default, instructions are executed in sequential order.

Instructions |
--- |
`sum = 0` |
`num1 = 5` |
`num2 = 10` |
`sum = num1 + num2` |

Flowchart

- When we call a function like `input()`, we transfer control to the function until it completes.
Altering the Flow of Control

- To solve many types of problems, we need to be able to modify the order in which instructions are executed.

- We've already seen one example of this: for loops allow us to repeat a set of statements some number of times.

```python
# add five integers
sum = 0
for i in range(5):
    num = eval(input("Enter a number: "))
    sum = sum + num
print("the sum is ", sum)
```

- Now we'll see how to allow a program to decide whether to do something, based on some condition.

Example of Making Decisions

- The ability to make decisions allows us to handle invalid inputs.

- Let's say the user gives us a number, and we want to compute its square root.
  - invalid inputs include:
    - negative numbers
    - non-numeric values (we'll ignore these for now)

- Here's one way to handle negative numbers:

```python
import math
num = eval(input("Enter a number >= 0: "))
if num < 0:
    print("using the absolute value of the number")
    num = num * -1
root = math.sqrt(num)
print("the square root of", num, "is", root)
```
Simple Decisions: if Statements

- A simple if statement has the form
  
  ```
  if <condition>:
    <true block>
  
  where <condition> is an expression that is true or false
  <true block> is one or more statements
  ```

- If the condition is true, the statement(s) in the true block are executed.

- If the condition is false, the statement(s) in the true block are skipped.

Flowchart for a Simple if Statement

```
condition
   true
   false
true block
next statement
```
Flowchart for Our Example

```
num = eval(input("Enter…"))
if num < 0:
    print("using the absolute ...")
    num = num * -1
    root = math.sqrt(num)
    print("the square root of ")
```

Expressing Simple Conditions

- As in SQL, Python provides a set of operators called relational operators for expressing simple conditions:

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>5 &lt; 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>num &lt; 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>40 &gt; 60 (which is false!)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state &gt; 'Ohio'</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>average &lt;= 85.8</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>name &gt;= &quot;Jones&quot;</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td>sum == 10</td>
</tr>
<tr>
<td></td>
<td>(don't confuse with =)</td>
<td>firstChar == &quot;P&quot;</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>age != myAge</td>
</tr>
</tbody>
</table>
### bool Data Type

- A condition has one of two values: `True` or `False`.
  ```python
  >>> 10 != 20
  True
  >>> "Jones" < "Baker"
  False
  ```

- In Python, these two values are represented using a special data type called `bool`.
  ```python
  >>> type(10 != 20)
  <class 'bool'>
  ```

- This type is named after the 19th-century mathematician George Boole, who developed the system of logic called boolean algebra.

- An expression that evaluates to `True` or `False` is known as a boolean expression.

### Forming More Complex Conditions

- We often need to make a decision based on more than one condition – or based on the opposite of a condition.
  - examples in pseudocode:
    ```python
    if the number is even AND it is greater than 100…
    if it is NOT the case that your grade is > 80…
    ```

- Like SQL, Python provides logical operators for this purpose:
  - **name**
  - **example and meaning**
  - **and**
    ```python
    age >= 18 and age <= 35
    true if both conditions are true, and false otherwise
    ```
  - **or**
    ```python
    age < 3 or age > 65
    true if one or both of the conditions are true; false if both conditions are false
    ```
  - **not**
    ```python
    not (grade > 80)
    true if the condition is false, and false if it is true
    ```
Practice with Boolean Expressions

• Let's say that we wanted to express the following English condition in Python:
  "num is not equal to either 0 or 1"

• Which of the following boolean expression(s) would work?
  a) num != 0 or 1
  b) num != 0 or num != 1
  c) not (num == 0 or num == 1)

• Is there a different boolean expression that would work here?

Warning: Python May Not Catch Your Mistakes!

• It turns out that
  num != 0 or 1
is actually a valid boolean expression in Python!

• That's because Python allows you to use values from other types to represent True and False.

• For numeric types:
  • 0 is equivalent to False
  • any other number is equivalent to True

• Therefore:
  num != 0 or 1
is equivalent to
  num != 0 or True

What's the value of this expression for different values of num?
Sample Problem: Number Analyzer

• Read in an integer from the user, and report whether it is even or odd.
  • what operator can we use to determine if a number is even?

• One possible approach (fill in the conditions):
  
  ```python
  num = eval(input("enter an integer: "))
  # if num is even, say so.
  if ________________________:
      print(num, "is even."
  # if num is odd, say so.
  if ________________________:
      print(num, "is odd."
  
  • The second condition is redundant. Why?

Sample Problem: Number Analyzer (cont.)

• A better approach (fill in the same first condition as before):
  
  ```python
  num = eval(input("enter an integer: "))
  if num % 2 == 0:
      print(num, "is even."
  else:
      print(num, "is odd."
  ```
Two-Way Decisions: `if-else` Statements

- In general, an `if-else` statement has the form
  ```python
  if <condition>:
      <true block>
  else:
      <false block>
  ```

- If the condition is true:
  - the statement(s) in the true block are executed
  - the statement(s) in the false block are skipped

- If the condition is false:
  - the statement(s) in the false block are executed
  - the statement(s) in the true block are skipped

Flowchart for an `if-else` Statement

[Diagram of a flowchart showing the flow of control for an `if-else` statement.]
Flowchart for Our Example

Extended Number Analyzer

- Let's add code to check if the value entered is really an integer.
  - what built-in function can we use?
  - if it isn't an int, we'll print an error message
  - if it is an int, we'll say whether it's even or odd as before

- We could do something like this:
  ```python
  num = eval(input("enter an integer: "))
  if type(num) != int:
    print(num, "is not an integer.")
  else:
    if num % 2 == 0:
      print(num, "is even.")
    else:
      print(num, "is odd.")
  ```

- We've nested our previous if-else statement in the false block of another if-else statement!
Extended Number Analyzer (cont.)

```python
num = eval(input("enter an integer: "))
if type(num) != int:
    print(num, "is not an integer.")
else:
    if num % 2 == 0:
        print(num, "is even.")
    else:
        print(num, "is odd.")
```

- Instead of using nesting, Python allows us to combine an `else` followed immediately by an `if` as follows:

```python
num = eval(input("enter an integer: "))
if type(num) != int:
    print(num, "is not an integer.")
elif num % 2 == 0:
    print(num, "is even.")
else:
    print(num, "is odd.")
```
Multi-Way Decisions: if-elif-else Statements

- In general, an if-elif-else statement has the form

```python
if <condition1>:
    <true block for condition1>
elif <condition2>:
    <true block for condition2>
elif <condition3>:
    <true block for condition3>
...
else:
    <false block for all conditions>
```

- The conditions are evaluated in order. The true block of the *first* true condition is executed.
- If none of the conditions are true, the false block is executed.

Flowchart for an if-elif-else Statement

- Decision points for conditions:
  - `condition1` (true or false)
  - `condition2` (true or false)
  - ... (true or false)

- True blocks:
  - `true block 1`
  - `true block 2`

- False block:
  - `false block`

- Next statement:
  - `next statement`
Example Problem: Ticket Sales

• Rules for ticket sales:
  • persons younger than 13 are not allowed to buy a ticket
  • persons 13-24 or 65 and older pay a discounted price of $35
  • everyone else pays the regular price of $50

Example Problem: Ticket Sales (cont.)

```python
age = eval(input("Enter your age: "))
if age < 13:
    print("You're too young to buy a ticket.")
else:
    print("You may buy a ticket.")
    if ______________________:
        price = 35
    else:
        price = 50
    print("The price is", price, "dollars.")
```

• What condition should be used to fill in the blank?
• Why can't we use an if-elif-else statement here?
Example Problem: Ticket Sales (cont.)

- Here's another version of the ticket-sales program that nests an `if-else` statement in the true block:

```python
age = eval(input("Enter your age: "))
if age >= 13:
    print("You may buy a ticket.")
    if age <= 24 or age >= 65:
        price = 35
    else:
        price = 50
    print("The price is", price, "dollars.")
else:
    print("You're too young to buy a ticket.")
```

Warning: Indentation Matters!

- Consider the following two code fragments:

```python
if age > 24:
    if age < 65:
        price = 50
    else:
        price = 35
```

```
if age > 24:
    if age < 65:
        price = 50
    else:
        price = 35
```

- When `age == 18`, what will the value of `price` be:
  - when we use the first version?
  - when we use the second version?
Warning: Indentation Matters! (cont.)

- An else or elif clause must be indented the same number of spaces as the corresponding if clause.

- If you're not careful, you can get an indentation error.
  - example:
    ```python
    if age > 24:
        price = 50
    else:
        price = 35
    ```
    which gives the following error message:
    ```
    SyntaxError: unindent does not match any outer indentation level
    ```
  - this error occurs because the else is indented 2 spaces more than the corresponding if

Warning: Don't Forget the Colon!

- There must be a colon at the end of each if, elif, and else clause.

- If you forget to include it, you'll get an error.
  ```python
  if age > 24:
      price = 50
  else
      ```
      highlighting the error
  ```SyntaxError: invalid syntax```

- When you get a syntax error, the problematic code is highlighted.
  - in the code above, the space after the else is highlighted, indicating that something is missing!
Practice: Expanding Our Ticket-Sales Program

• Different prices for balcony seats and orchestra seats

• Expanded rules for ticket sales:
  • persons younger than 13 are not allowed to buy a ticket
  • persons 13-24 or 65 and older receive discounted prices:
    • $20 for balcony seats
    • $35 for orchestra seats
  • everyone else pays the regular prices:
    • $30 for balcony seats
    • $50 for orchestra seats

Expanding Our Ticket-Sales Program (cont.)

```python
age = eval(input("Enter your age: "))
if age >= 13:
    print("You may buy a ticket.")
    type = eval(input("(1) orchestra | (2) balcony? "))
    # what should go here? (assume input is 1 or 2)
    price = 0
    if type == 1:
        price = 30
    else:
        price = 50
    print("The price is", price, "dollars.")
else:
    print("You're too young to buy a ticket.")
```
Avoid Overly Complicated Code

• The following also involves decisions based on a person’s age:

```python
age = eval(input("Enter your age: "))
if age < 13:
    print('You are a child. ')
elif age >= 13 and age < 20:
    print('You are a teenager. ')
elif age >= 20 and age < 30:
    print('You are in your twenties. ')
elif age >= 30 and age < 40:
    print('You are in your thirties. ')
else:
    print('You are really old. ')
```

• How could it be simplified?

Practice: A Simple Calculator

• Ask the user to enter:
  • two numbers
  • the operation they want to perform (add, subtract, multiply)
  and compute and print the result.

• We can use a single `input` statement to get both numbers!

```python
>>> a, b = eval(input("Enter two numbers: "))
Enter two numbers: 5, 10
>>> a
5
>>> b
10
```

• This makes use of what is known as **simultaneous assignment**.
  • general form:
    `<var1>, <var2>, <var3>, ... = <expr1>, <expr2>, <expr3>, ...`
Practice: A Simple Calculator

```python
a, b = eval(input("Enter two numbers: "))

print("Choose an operation by number:"
print("  (1) addition")
print("  (2) subtraction")
print("  (3) multiplication")
choice = eval(input())

# what should go here?

print("The result is", result)
```
Strings in Python

- The `str`ing data type is used to represent text data.

- We've already been using string literals, which consist of a sequence of characters surrounded by either double or single quotes.
  
  ```python
  "Enter a number: "
  'The sum is'
  ```

- To see the data type of these values, we can use the built-in `type()` function:

  ```python
  >>> type('Enter a number: ')  
  <class 'str'>
  >>> myName = 'Dave'
  >>> type(myName)  
  <class 'str'>
  ```
Inputting String Values

• Recall: the `input` function treats any value that the user enters as a string.

• Thus, when we want the user to enter a string, we use `input` without using `eval`.

  ```
  >>> name = input("Enter your name: ")
  Enter your name: Perry
  >>> print(name)
  Perry
  ```

• Using `eval` doesn't work with strings that can't be evaluated.

  ```
  >>> name = eval(input("Enter your name: "))
  Enter your name: Perry
  Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    name = eval(input("Enter your name: "))
  File "<string>", line 1, in <module>
  NameError: name 'Perry' is not defined
  ```

Numbering the Characters in a String

• A string is just a sequence of characters.

• The position of a character within a string is known as its `index`.

• There are two ways of numbering characters in Python:
  • from left to right, starting from 0

    ```
    "Perry"
    ```

    • from right to left, starting from -1

    ```
    "Perry"
    ```

• `P` has an index of 0 or -5
• `y` has an index of 4 or -1
Accessing a Character in a String

- To access an individual character within a string, we specify its index as follows:

  `<string>[<index>]`

- examples:

  ```python
  >>> name = "Perry"
  >>> name[0]
  'P'
  >>> name[4]
  'y'
  >>> name[2 - 1]
  'e'
  >>> name[-1]
  'y'
  
  An index that is too large or too small produces an error:

  ```python
  >>> name[5]
  IndexError: string index out of range
  ```

Slicing: Extracting a Substring

- To extract a substring, we specify two index values:

  `<string>[<startIndex>:<endIndex>]`

- The resulting substring/slice:

  - begins at `<startIndex>`
  - ends at `<endIndex> - 1`

  - for a substring of length `N`, figure out `startIndex` and do:

    ```python
    <string>[startIndex:startIndex + N]
    ```

- examples:

  ```python
  >>> name = "Perry"
  >>> name[0:2]
  'Pe'
  >>> name[4:5]
  'y'
  >>> name[1:4]
  'err'
  ```
Slicing: Extracting a Substring (cont.)

- If we omit the start/end index, the substring extends to the start/end of the string:
  - examples:
    ```python
    >>> name = "Perry"
    >>> name[1:]
    'erry'
    >>> name[:3]
    'Per'
    >>> name[:]
    'Perry'
    ```

  - Given the following assignment:
    ```python
    >>> s = "computer"
    ```

  - What is the value of each of the following?
    ```python
    >>> s[1]
    >>>
    >>> s[-1]
    >>>
    >>> s[2:4]
    >>>
    >>> s[:3]
    >>>
    >>> s[5:]
    >>>
    >>> s[-4:-1]
    ```
Concatenation

- To concatenate two strings, we use the + operator.
  - examples:
    ```python
    >>> word = "database"
    >>> plural = word + "s"
    >>> plural
    'databases'
    - note: both of the values must be strings
      ```python
      >>> print("grade: " + 90)
      ...TypeError: cannot concatenate 'str' and 'int' objects
      ```
    - to convert a number to a string, use the str() function:
      ```python
      >>> print("grade: " + str(90))
      grade: 90
      ```

Other String Operations

- The * operator can be used to repeat a string.
  ```python
  >>> print("-" * 10)
  ----------
  >>> print("ho! " * 3)
  ho! ho! ho!
  ```
- The built-in len() function can be used to get the length of a string.
  ```python
  >>> len("Perry")
  5
  >>> len("")
  0
  ```
- The in operator allows us to test for the presence of a substring.
  ```python
  >>> "base" in "database"
  True
  >>> "case" in "database"
  False
  ```
for Statements and Strings

- Recall the syntax of a `for` statement:
  ```python
  for <variable> in <sequence>:
      <body>
  ```

  where `<sequence>` is a sequence of values
  `<body>` is one or more statements

- Example: what will the following `for` loop print?
  ```python
  >>> for i in [2, 4, 6, 8]:
      print(i * 3, end=" ")
  ```

- Because a string is a sequence of characters, we can use a `for` loop to iterate over the characters in a string:
  ```python
  >>> for i in "hello":
      print(i, end=" ")
  hello
  ```

Lists

- As we saw earlier, a list is another type of sequence.
  ```python
  [2, 4, 6, 8]
  ["CS", "math", "english", "psych"]
  ```

- unlike a string, a list can include values of different types:
  ```python
  ["The Godfather", 1972, "R"]
  ```

- All of the string operations are really operations that can be applied to *any* type of sequence, including lists.
  ```python
  >>> majors = ["CS", "math", "english", "psych"]
  >>> majors[2]
  'english'
  >>> majors[1:3]
  ['math', 'english']
  >>> len(majors)
  4
  >>> majors + ["physics"]
  ["CS", "math", "english", "psych", "physics"]
  ```
Mutable vs. Immutable

• A list is mutable, which means that it can be changed "in place":
  >>> majors = ['CS', 'math', 'english', 'psych']
  >>> majors
  ['CS', 'math', 'english', 'psych']
  >>> majors[2] = 'literature'
  >>> majors
  ['CS', 'math', 'literature', 'psych']

• A string is immutable, which means it can't be changed "in place."
  >>> sentence = "a string a immutable."
  >>> sentence[0] = "A"
  TypeError: 'str' object does not support item assignment

Strings Are Objects

• In programming languages, an object is a construct that allows us to group together:
  • one or more data values
  • operations that can be performed on those values (special functions known as the object's methods)

• In Python, a string is an object.
  • data values = the characters in the string
  • methods = functions that operate on the string

• To use a method that belongs to an object, we use dot notation.
  • examples:
    >>> name = "Perry"
    >>> name.upper()
    PERRY
String Methods (partial list)

- `s.lower()`: return a copy of `s` with all lowercase characters
- `s.upper()`: return a copy of `s` with all uppercase characters
- `s.find(sub)`: return the index of the first occurrence of the substring `sub` in the string `s` (-1 if not found)
- `s.count(sub)`: return the number of occurrences of the substring `sub` in the string `s` (0 if not found)
- `s.replace(target, repl)`: replace all occurrences of the substring `target` in `s` with the substring `repl`

Splitting a String

- The `split()` method breaks a string into a list of substrings.

- By default, it uses spaces to determine where the splits should occur:
  ```python
  >>> name = "Martin Luther King"
  >>> name.split()
  ['Martin', 'Luther', 'King']
  ```

- You can specify a different separator:
  ```python
  >>> date = "03/05/2013"
  >>> date.split("/")
  ['03', '05', '2013']
  >>> record = '11234,Alan Turing,comp sci'
  >>> record.split(',')
  ['11234', 'Alan Turing', 'comp sci']
  ```
Splitting a String (cont.)

- You can use simultaneous assignment to assign the slices produced by `split()` to separate variables:
  
  ```python
  >>> mon, day, year = "03/05/2013".split("/")
  >>> mon
  '03'
  >>> day
  '05'
  >>> year
  '2013'
  ```

Joining Together a List of Strings

- The `join()` method takes a list of strings and joins them together.

- `join()` is a `string` method, not a list method.
  - we call it using the string that we want to use as a separator

- Examples:
  
  ```python
  >>> components = ['Martin', 'Luther', 'King']
  >>> ".join(components)
  'Martin Luther King'
  >>> '/'.join(['03', '05', '13'])
  '03/05/13'
  ```
Practice: Analyzing a Name

• Write a program that analyzes a person's name.

• Here's a sample run of the program:

Enter your full name: George Alexander Louis Wales
Your name has 28 characters (including spaces).

Your name has 4 components.
  first name: George
  last name: Wales
  other names: Alexander Louis

Enter a letter: r
That letter occurs 2 times in your name.
The first occurrence is at position 3 in the name.
Accessing a Database from Python

• Import the necessary Python module:
  
  ```python
  import sqlite3
  ```

• Connect to the database and create an object known as a database handle, assigning it to a variable named `db`.
  
  ```python
  db = sqlite3.connect('<name of database file>')
  ```

• To connect to the database that we used for PS 2 and 3:
  • put the database file (`movie.db`) in the same folder as the Python program that will access it
  • use the following command in the program:
  
  ```python
  db = sqlite3.connect('movie.db')
  ```
Performing a Query

- Given a database handle, we can perform a query by:
  - using the database handle to create a cursor object:
    ```python
cursor = db.cursor()
```
  - using the cursor to execute the query:
    ```python
cursor.execute("<SELECT command>")
```
- Once the query has been executed, we use the cursor to access the results.
- To get one tuple from the result, we use the `fetchone` function:
  ```python
  row = cursor.fetchone()
  ```
  - if there's nothing to get, this function returns `None`

Performing a Query (cont.)

- Example for our movie database:
  ```python
  >>> cursor = db.cursor()
  >>> cursor.execute("SELECT name, year FROM Movie;")
  >>> cursor.fetchone()
  ('Avatar', 2009)
  ```
Performing a Query (cont.)

- Another example:
  >>> cursor.execute('''SELECT name, rating
FROM Movie
WHERE year = 1976;''')
  >>> cursor.fetchone()
  ('Rocky', 'PG')
  >>> cursor.fetchone()
  ('Network', 'R')
  >>> cursor.fetchone()
  ('All the President's Men', 'PG')

Using a for Loop to Display Query Results

- Python allows us to use a for loop to process the results of a query one tuple at a time.

- Syntax: 
  ```python
  for tuple in cursor:
    # code to process tuple goes here
  ```

- Example:
  ```python
  cursor = db.cursor()
  cursor.execute('''SELECT name, rating
FROM Movie
WHERE year = 1976;'')
  for tuple in cursor:
    print(tuple[0], tuple[1])
  ```

- The above code would give the following output:
  Rocky PG
  Network R
  All the President's Men PG
Executing a Query Based on User Input

- How can we execute a query that's based on user inputs?
- Example interaction:

```
year to search for? 1976
Rocky PG
Network R
All the President's Men PG
```

- In theory, we could construct the query using string concatenation:

```
searchYear = input("year to search for? ")
command = '''SELECT name, rating FROM Movie
WHERE year = '''
command = command + searchYear + '''
cursor.execute(command)
for tuple in cursor:
    print(t[0], t[1])
```

SQL Injection Vulnerability

- Problem: constructing a query using string concatenation can lead to a serious security breaches.
  - known as a SQL injection.

- Example: let's say that in addition to the movie tables, there's a table called Secret containing sensitive data.

- The user of the program shown on the previous slide can see data from Secret by entering something like this:

```
Year to search for? 1976; SELECT * FROM Secret
```

- The string concatenation will produce the following:

```
SELECT name, rating FROM Movie WHERE year = 1976;
SELECT * FROM Secret;
```

- The program will display the entire first two columns of Secret!
SQL Injection Vulnerability (cont.)

- Here's another problematic input!
  
  \textit{Year to search for? 1976; DROP TABLE Secret}

Parameterized Queries

- To avoid SQL injections, we use a \textit{parameterized query}.
- example:
  
  \begin{verbatim}
  command = '''SELECT name, rating FROM Movie
           WHERE year = ?;'''
  \end{verbatim}

  - ? is a placeholder

- We execute the parameterized query as follows:
  
  \begin{verbatim}
  cursor.execute(command, [searchYear])
  \end{verbatim}

  - The \texttt{execute} function replaces the placeholders with the specified values – taking whatever steps are needed to treat each parameter as a single literal value.
Parameterized Queries (cont.)

• Here's a query that takes three parameters:

```sql
command = '''SELECT M.name, M.year
FROM Movie M, Person P, Director D
WHERE M.id = D.movie_id
AND P.id = D.director_id
AND P.name = ?
AND M.year BETWEEN ? AND ?;'''
```

• Here's an example of using it:

```python
dirName = input("director's name: ")
start = input("start of year range: ")
end = input("end of year range: ")

cursor.execute(command, [dirName, start, end])
for tuple in cursor:
    print(tuple[0], tuple[1])
```

The Full Program (getMoviesByDirector.py)

```python
import sqlite3
filename = input("name of database file: ")
db = sqlite3.connect(filename)
cursor = db.cursor()

dirName = input("director's name: ")
start = input("start of year range: ")
end = input("end of year range: ")

command = '''SELECT M.name, M.year
FROM Movie M, Person P, Director D
WHERE M.id = D.movie_id
AND P.id = D.director_id
AND P.name = ?
AND M.year BETWEEN ? AND ?;'''
cursor.execute(command, [dirName, start, end])
for tuple in cursor:
    print(tuple[0], tuple[1])
```
Handling Queries with No Results

• What if the user enters a director who isn't in the database, or a range of years with no movies for the director?

• We'd like our program to print a message when this happens.

• One way of doing this is to maintain a count of the number of tuples that the program processes:

```python
    ...;
    cursor.execute(command, [dirName, start, end])
    count = 0
    for tuple in cursor:
        print(tuple[0], tuple[1])
        count = count + 1
    # print a message if there were no results
    # what should go here?
```

Concluding a Database Session

• At the end of a program that accesses a database, you should always do the following:
  • commit the transaction in which the commands were executed:
    db.commit()
  • close the database handle:
    db.close()
A Front-End for Our Movie Database

• Let's write a Python program that serves as a front-end to our movie database.

• For now, it will do the following:
  • get the name of a person from the user
  • use a parameterized SELECT command to retrieve the appropriate record from the Person table
  • if the specified person is in the database, print his/her information in the following format:
    <name> was born on <dob> in <pob>.
  • otherwise, print an appropriate error message

• Extension: change the format of the dob from yyyy-mm-dd to mm/dd/yyyy
Converting the Date Format

• To change the format of the dob from yyyy-mm-dd to mm/dd/yyyy, what should we do?
Escape Sequences

- Recall: we can surround strings by either single or double quotes.
  - doing so allows us to embed quotes within a string
    
    'Homer said, "Doh!"'

- We can also embed a double or single quote by preceding it with a \ character.
  
  "Homer said, \"Doh!\""

- \" is known as an escape sequence.

- The \ tells the compiler to interpret the following character differently than it ordinarily would.

- Other examples:
  - \n    a newline character (go to the next line)
  - \t    a tab
  - \\    a backslash!
Text Files

• A text file can be thought of as a multi-line string.
  • example: the following four-line text file

```
# simple.py
print(2 + 3)
print(10 - 7)
```

is equivalent to the following string:
```
# simple.py
print(2 + 3)
print(10 - 7)
```

Opening a Text File

• Before we can read from or write to a text file, we need to open a connection to the file.

• Doing so creates an object known as a file handle.
  • we use the file handle to perform operations on the file

• Syntax:

```
<file-handle> = open(<filename>, <mode>)
```

where <file-handle> is a variable for the file handle
<filename> is a string
<mode> is:
  'r' if we want to read from the file
  'w' if we want to write to the file,
  (erasing any existing contents)
  'a' if we want to append to the end of the file
Specifying Filenames

- When specifying the name of a file, we'll just give the name of the file itself.
  - example: `"myData.txt"
  - we won't specify the directory

- Python will open/create the file in the same directory in which the module file is stored.
  - the rules are a bit more complicated when you're using the interpreter from the command prompt

Closing a File

- Here's an example of opening a file for writing:
  ```python
  outfile = open("example.txt", 'w')
  ```

- When we're done reading from or writing to a file, we need to close its handle:
  ```python
  outfile.close()
  ```

- **Important:** Text that you write to file may not make it to disk until you close the file handle!
Writing to a File

- When you open a file for writing:
  - if the file doesn't already exist, it will be created
  - if the file does exist and you specify the 'w' mode, the current contents will be erased!
  - if the file does exist and you specify the 'a' mode, the text you write will be appended to the end of the file

- To write values to a file, we can use the `print()` method as usual, but with an extra parameter for the file:

  ```
  print(..., file=<file-handle>)
  ```

- Example:

  ```
  outfile = open("foo.txt", 'w')
  print("I love Python!", file=outfile)
  ```

Example: Writing Database Results to a File

- Recall our program for getting all movies from a given year:

  ```
  import sqlite3
  filename = input("name of database file: ")
  db = sqlite3.connect(filename)
  cursor = db.cursor()
  searchYear = input("year to search for? ")
  command = '''SELECT name, rating
  FROM Movie WHERE year = ?;'''
  cursor.execute(command, [searchYear])
  for tuple in cursor:
    print(tuple[0], tuple[1])
  db.commit()
  db.close()
  ```

- Let's modify it so that it writes the results to a file.
Reading from a File

- When you open a file for reading, the file must already exist, or you'll get an error:
  ```python
  >>> infile = open("noexist.txt", 'r')
  ...IOError: [Errno 2] No such file or directory
  ```

- To read one line at a time, we can use the `readline()` function.

- Syntax:
  ```python
  <variable> = <file-handle>.readline()
  ```
  example: `line = infile.readline()`

- This function returns a string containing the next line in the file – up to and including the next newline character (`\n`).

Reading from a File (cont.)

- Example: assume that we have our earlier four-line text file:
  ```python
  # simple.py
  print(2 + 3)
  print(10 - 7)
  ```

- here's one possible set of operations on that file:
  ```python
  >>> infile = open("simple.py", 'r')
  >>> line = infile.readline()
  >>> line
  '# simple.py
'
  >>> infile.readline()
  '\n'
  >>> line = infile.readline()
  >>> line
  'print(2 + 3)\n'
  ```
Closing and Reopening a File

- If we've been reading from a file and want to start over again from the beginning of the file, we need to close the file and reopen it again.

  - example:
    ```python
    >>> infile = open("simple.py", 'r')
    >>> infile.readline()
    '# simple.py
    >>> infile.readline()
    '\n
    >>> infile.close()
    >>> infile = open("simple.py", 'r')
    >>> infile.readline()
    '# simple.py
    ```

Processing a File Using a for Loop

- We often want to read and process each line in a file.

- Because we don't usually know how many lines there are in the file, we use a for loop.

- Syntax:
  ```python
  for line in <file-handle>:
      # code to process line goes here
  ```

- reads one line at a time and assigns it to line
Example of Processing a File

• Let’s say that we want a program to print a text file to the screen, omitting all blank lines.

• Here’s one possible implementation:

```python
filename = input("name of file: ")
infile = open(filename, 'r')
for line in infile:
    if line != "\n":
        print(line[:-1])
infile.close()
```

• Why do we need to use slicing? (line[:-1])

• How could we change it so that it omits full-line comments instead (i.e., lines that begin with a #)?

Extracting Relevant Data from a File

• High-school and college track teams often participate in meets involving a large number of schools.

• Assume that the results of a meet are summarized in a comma-delimited text file that looks like this:

  Mike Mercury, Boston University, mile, 4:50:00
  Steve Slug, Boston College, mile, 7:30:00
  Len Lightning, Boston University, half-mile, 2:15:00
  Tom Turtle, UMass, half-mile, 4:00:00

• Let’s write a program that reads in a results file and extracts just the results for a particular school – with the name of the school omitted from the records.
  • write the results to a file
Extracting Relevant Data from a File
Data Mining I: Introduction

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

References for This Part of the Course

• Roiger & Geatz, *Data Mining: A Tutorial-Based Primer* (Addison-Wesley, 2003)

What is Data Mining?

• Informally, it's the process of using a computer program to find patterns or relationships in data.

• Examples:
  • looking for combinations of symptoms that are reliable indicators of a given disease
  • mining a grocery store’s customer-purchase data
    • which two products below were found to be frequently purchased together?
      beer  cereal  diapers
      milk  soft drinks  toilet paper
    • how could the store make use of this result?

Finding Patterns

• Something that human beings have always done!
  • example: how do we learn to identify a dog?
Finding Patterns (cont.)

- In data mining:
  - the data is stored in electronic form
  - the process is automated (at least in part) by using a computer program
  - the program "mines" the data – "sifting through" it in an attempt to find something useful/valuable

Data Mining vs. Data Query

- Database queries in SQL are \textit{not} the same thing as data mining.
- Queries allow us to extract factual information.
  - "shallow knowledge"
- In data mining, we attempt to extract patterns and relationships that go beyond mere factual information.
  - "hidden knowledge"
Machine Learning

• In data mining, we apply an algorithm that "learns" something about the data.

• These algorithms are referred to as *machine-learning algorithms*.

• There are several different types of machine learning that are used in data mining.
  • classification learning
  • association learning
  • numeric estimation
  • clustering

• We will limit ourselves to the first three types.

Classification Learning

• Classification learning involves learning how to classify objects/entities on the basis of their characteristics.
  • example: how to classify credit-card purchases as fraudulent or non-fraudulent

• Input to the algorithm = a set of data describing objects that have already been classified.
  • known as *training data or training examples*

• Output = a *model* that can be used to classify other objects.
  • different algorithms produce different types of models
Example: Medical Diagnosis

- Given a set of symptoms, we want to be able to determine a correct diagnosis for a patient with cold-like symptoms.

- Sample training data (table 1-1 of Roiger & Geatz):

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Allergy</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
</tbody>
</table>

- Can you see any patterns that would help you diagnose patients with one or more of these symptoms?

Example: Medical Diagnosis (cont.)

- One possible model that could be used for classifying other patients is a set of rules like the following:

  if Swollen Glands == Yes then Diagnosis = Strep Throat
  if Swollen Glands == No and Fever == Yes then Diagnosis = Cold
  if Swollen Glands == No and Fever == No then Diagnosis = Allergy
• Another possible type of model is known as a decision tree:

- start at the top and work down until you reach a box containing a classification
- what diagnosis would the tree give for patient 11 above?

Some Terminology

- Each row in a collection of training data is known as an example or instance.

- Each column is referred to as an attribute.

- The attributes can be divided into two types:
  - the output attribute – the one we want to determine/predict
  - the input attributes – everything else

  input attributes $\rightarrow$ model $\rightarrow$ output attribute

- In our example:

  fever swollen glands headache … $\rightarrow$ rules or tree or… $\rightarrow$ diagnosis
### Types of Attributes

- **Nominal** attributes have values that are "names" of categories.
  - there is a small set of possible values
    | attribute      | possible values       |
    |----------------|-----------------------|
    | Fever          | {Yes, No}             |
    | Diagnosis      | {Allergy, Cold, Strep Throat} |

- In classification learning, the output attribute is always *nominal*.

- **Numeric** attributes have values that come from a range of #s.
<table>
<thead>
<tr>
<th>attribute</th>
<th>possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body Temp</td>
<td>any value in 96.0-106.0</td>
</tr>
<tr>
<td>Salary</td>
<td>any value in $15,000-250,000</td>
</tr>
</tbody>
</table>

  - you can *order* their values
    - $210,000 > $125,000
    - 98.6 < 101.3

### Types of Attributes (cont.)

- What about this one?
<table>
<thead>
<tr>
<th>attribute</th>
<th>possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Type</td>
<td>{0, 1, 2, 3}</td>
</tr>
</tbody>
</table>

- If numbers are used as IDs or names of categories, the corresponding attribute is actually nominal.

- Note that it doesn't make sense to order the values of such attributes.
  - example: product type 2 > product type 1 doesn't have any meaning
### Numeric Estimation

- Like classification learning, but for a *numeric* output attribute.
  - example: a charity that needs to decide who should be sent a fundraising appeal letter

```
| age | income | zip code | avg past donation | ... |
```

- The model learned by the algorithm often takes the form of an equation.

\[
\text{probability of reply} = 0.424\text{attr1} - 0.072\text{attr2} + \ldots
\]

where \(\text{attr1}, \text{attr2}, \ldots\) are attributes

- Linear regression is a form of numeric estimation.

### Association Learning

- In association learning, the algorithm looks for relationships between sets of attributes in the training examples.
  - produces a set of rules
  - for example:
    ```
    \text{if Congestion} = \text{Yes} \\
    \text{then Headache} = \text{Yes}
    \\
    \text{if Sore Throat} = \text{Yes and Swollen Glands} = \text{No} \\
    \text{then Congestion} = \text{Yes and Fever} = \text{No}
    ```

- Unlike classification learning and numeric estimation, association learning does *not* focus on predicting a particular attribute.
  - no distinction between input and output attributes
Association Learning (cont.)

- One form of association learning is *market-basket analysis*, which finds associations between items that people buy.
  - classic example: beer and diapers on Thursdays!

- Association learning is often more difficult than classification learning. Why do you think that is?

---

Example: Data About Investors

<table>
<thead>
<tr>
<th>Customer ID</th>
<th>Account Type</th>
<th>Margin Account</th>
<th>Transaction Method</th>
<th>Trades/ Month</th>
<th>Sex</th>
<th>Age</th>
<th>Favorite Recreation</th>
<th>Annual Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Joint</td>
<td>No</td>
<td>Online</td>
<td>12.5</td>
<td>F</td>
<td>30–39</td>
<td>Tennis</td>
<td>40–59K</td>
</tr>
<tr>
<td>1013</td>
<td>Custodial</td>
<td>No</td>
<td>Broker</td>
<td>0.5</td>
<td>F</td>
<td>50–59</td>
<td>Skiing</td>
<td>80–99K</td>
</tr>
<tr>
<td>1245</td>
<td>Joint</td>
<td>Yes</td>
<td>Online</td>
<td>3.6</td>
<td>M</td>
<td>20–29</td>
<td>Golf</td>
<td>20–39K</td>
</tr>
<tr>
<td>2110</td>
<td>Individual</td>
<td>Yes</td>
<td>Broker</td>
<td>22.3</td>
<td>M</td>
<td>30–39</td>
<td>Fishing</td>
<td>40–59K</td>
</tr>
<tr>
<td>1001</td>
<td>Individual</td>
<td>Yes</td>
<td>Online</td>
<td>5.0</td>
<td>M</td>
<td>40–49</td>
<td>Golf</td>
<td>60–78K</td>
</tr>
</tbody>
</table>

*Table 1-3 in Roiger & Geatz*

- Why might we want to perform classification learning on this type of data?
  - what are some possible class attributes?
    - 
    - 

- What about numeric estimation?
Summary of Machine-Learning Approaches

- **classification learning**: takes a set of already classified training examples and learns a model that can be used to classify previously unseen examples

  ```
  if Swollen Glands = Yes then Diagnosis = Strep Throat
  if Swollen Glands = No and Fever = Yes then Diagnosis = Cold
  ...
  ```

- **numeric estimation**: like classification learning, but the output attribute is numeric
  - the model is typically in the form of an equation

Summary of Machine-Learning Approaches (cont.)

- **association learning**: takes a set of training examples and discovers associations among attributes
  - we don't specify a single class/output attribute

  ```
  if Congestion = Yes then Headache = Yes
  if Sore Throat = Yes and Swollen Glands = No then Congestion = Yes and Fever = No
  ```
Evaluating the Results

- For most non-trivial, real-world data sets, no learned model is likely to work perfectly on all possible examples.
  - the concepts being modelled are complicated
  - they may depend on factors for which we don't have data
  - there may be noise in our training data – imprecise or inaccurate values

- Our goal is not to create a model that perfectly matches the training data.

- Instead, we want a model that performs well on previously unseen examples.
  - we say that we want the model to generalize

Another Example: Labor Negotiations

- Goal: to be able to predict whether a labor contract will be acceptable to the union and to management.
  - class = good (acceptable) or bad (unacceptable)

- Training data = 40 examples from actual labor negotiations
  - 17 attributes
  - lots of missing values

- Source of this case study: Witten and Frank
Another Example: Labor Negotiations (cont.)

• Here's one possible decision tree based on the training data:
  • simple model
  • makes intuitive sense
  • misclassifies some of the training examples

• Here's another possible decision tree from the same data:
  • It does a better job classifying the training examples.
  • However, it may not do as well on previously unseen examples.
  • it may not generalize as well as the simpler model
Overfitting

• In general, working too hard to match the training examples can lead to an overly complicated model that doesn't generalize well.

• This problem is known as overfitting the training data.

• Extreme example of overfitting: memorize the training examples!
  • example: store all 10 training examples from the diagnosis problem in a table
  • when a new patient comes in, see if his/her symptoms match any of the training examples and use that diagnosis
  • why won't this work?

Overfitting (cont.)

• Overfitting can also happen in numeric estimation.

• Example:

  • points = training examples
  • h(x) = the actual function we are trying to estimate
  • g(x) underfits the training data – it's too simple
  • j(x) overfits the training data – it's too complicated
  • neither g(x) nor j(x) will do well on unseen examples
Evaluating Classification Learning

- To test how well a model generalizes, we typically withhold some of the examples as test examples.
  - these examples are not used to train the model

- Example: decision-tree model for medical diagnosis
  - trained using the 10 earlier examples
  - we can test it using the examples shown below

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
<th>Model's Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
<td>Strep throat</td>
</tr>
<tr>
<td>13</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
<td>Cold</td>
</tr>
<tr>
<td>14</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
<td>Cold</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Strep throat</td>
<td>Cold</td>
</tr>
<tr>
<td>16</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
<td>Allergy</td>
</tr>
<tr>
<td>17</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Allergy</td>
<td>Allergy</td>
</tr>
</tbody>
</table>

Evaluating Classification Learning (cont.)

- The error rate of a model is the percentage of test examples that it misclassifies.
  - in our example, the error rate $= 2/6 = 33.3 \%$
  - the model's accuracy $= 100 – \text{error rate}$

- Problem: these metrics treat all misclassifications as being equal.
  - this isn't always the case
  - example: more problematic to misclassify strep throat than to misclassify a cold or allergy
Evaluating Classification Learning (cont.)

To provide a more detailed picture of the model's accuracy, we can use a **confusion matrix**:

<table>
<thead>
<tr>
<th>predicted class</th>
<th>cold</th>
<th>allergy</th>
<th>strep throat</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual class:</td>
<td>cold</td>
<td>allergy</td>
<td>strep throat</td>
</tr>
<tr>
<td>cold</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>allergy</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>strep throat</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- the diagonal of the matrix shows cases that were correctly classified

---

**Interpreting a Confusion Matrix**

- Let's say that we had a larger number of test examples, and that we obtained the following confusion matrix:

<table>
<thead>
<tr>
<th>predicted class</th>
<th>cold</th>
<th>allergy</th>
<th>strep throat</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual class:</td>
<td>cold</td>
<td>allergy</td>
<td>strep throat</td>
</tr>
<tr>
<td>cold</td>
<td>25</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>allergy</td>
<td>6</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>strep throat</td>
<td>5</td>
<td>4</td>
<td>33</td>
</tr>
</tbody>
</table>

- what is the accuracy of the model?

- what is its error rate?
Interpreting a Confusion Matrix (cont.)

<table>
<thead>
<tr>
<th>actual class:</th>
<th>cold</th>
<th>allergy</th>
<th>strep throat</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>25</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>allergy</td>
<td>6</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>strep throat</td>
<td>5</td>
<td>4</td>
<td>33</td>
</tr>
</tbody>
</table>

- how many test cases of strep throat are there?
- how many actual colds were misdiagnosed?
- what percentage of actual colds were correctly diagnosed?

Two-Class Confusion Matrices

- When there are only two classes, the classification problem is often framed as a yes / no judgement:
  - yes / no
  - fraudulent / not fraudulent
  - has cancer / doesn't have cancer

  The terms positive / negative are often used in place of yes / no.

- In such cases, there are four possible types of classifications:
  - true positive (TP): the model correctly predicts "yes"
  - false positive (FP): the model incorrectly predicts "yes"
  - true negative (TN): the model correctly predicts "no"
  - false negative (FN): the model incorrectly predicts "no"

<table>
<thead>
<tr>
<th>predicted</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>no</td>
<td>FP</td>
<td>TN</td>
</tr>
</tbody>
</table>
Comparing Models Using Confusion Matrices

• Let's say we're trying to detect credit-card fraud.

• We use two different classification-learning techniques and get two different models.

• Performance on 400 test examples:

<table>
<thead>
<tr>
<th></th>
<th>predicted by model A</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fraud</td>
<td>100</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>actual:</td>
<td>not fraud</td>
<td>40</td>
<td>250</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>predicted by model B</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fraud</td>
<td>80</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>actual:</td>
<td>not fraud</td>
<td>20</td>
<td>270</td>
<td></td>
</tr>
</tbody>
</table>

• which model is better?

Overall Accuracy Isn't Enough

• Someone tells you that they have a fraud-detection classifier with an overall accuracy of 99%. Should you use it?

• It depends on the test examples used to compute the accuracy!

• Example:
  • assume 1% of actual credit-card purchases are fraudulent
  • assume the test examples reflect this:
    • 10 examples of fraud, 990 examples of not fraud
  • on these examples, a model can be 99% accurate by always predicting "not fraud"!

<table>
<thead>
<tr>
<th></th>
<th>predicted</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fraud</td>
<td>0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>actual:</td>
<td>not fraud</td>
<td>0</td>
<td>990</td>
<td></td>
</tr>
</tbody>
</table>
Overall Accuracy Isn't Enough (cont.)

- Test examples should include an adequate number of all possible classifications.
  - especially ones you're most concerned about getting right
  - in our example, need to include enough examples of fraud

- It's also important that your training examples include all possible classifications.

- If you're primarily concerned about getting one of the classifications right, it may make sense to artificially inflate the number of examples of that class in the training examples.
Data Mining II: Classification Learning

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

Weka Data-Mining Software

- A free, open-source data-mining tool.
- Weka is available here:
  http://www.cs.waikato.ac.nz/~ml/weka
- Documentation is available here:
  http://www.cs.waikato.ac.nz/~ml/weka/documentation.html
  and in the Witten & Frank book mentioned last week.
Review: Classification Learning

- Classification-learning algorithms:
  - take a set of already classified training examples
  - also known as training instances
  - learn a model that can classify previously unseen examples

- The resulting model works like this:

  ![Diagram](image.png)

  **input attributes** (everything but the class)  
  **model**  
  **output attribute/class**

Review: Classification Learning (cont.)

- Recall our medical-diagnosis example:

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Allergy</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
</tbody>
</table>

  **the learned model:**
  
  \[
  \text{if Swollen Glands} = \text{Yes} \quad \Rightarrow \quad \text{Diagnosis} = \text{Strep Throat} \\
  \text{if Swollen Glands} = \text{No and Fever} = \text{Yes} \quad \Rightarrow \quad \text{Diagnosis} = \text{Cold} \\
  \text{if Swollen Glands} = \text{No and Fever} = \text{No} \quad \Rightarrow \quad \text{Diagnosis} = \text{Allergy}
  \]
Example Problem: Credit-Card Promotions

- A credit-card company wants to determine which customers should be sent promotional materials for a life insurance offer.

- It needs a model that predicts whether a customer will accept the offer:

  ![model diagram]

  age
  sex
  income range
  credit-card insurance*

  Yes (will accept the offer)
  or
  No (will not accept the offer)

* note: credit-card insurance is a Yes/No attribute specifying whether the customer accepted a similar offer for insurance on their credit card

Example Problem: Credit-Card Promotions

- 15 training examples (Table 3.1 of Roiger & Geatz):

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
1R: Learning Simple Classification Rules

- Presented by R.C. Holte in the following paper:
  

- Why is it called 1R?
  - R because the algorithm learns a set of Rules
  - 1 because the rules are based on only 1 input attribute

- The rules that 1R learns look like this:
  
  \[
  \langle\text{attribute-name}\rangle: \langle\text{attribute-val1}\rangle \rightarrow \langle\text{class value}\rangle \\
  \langle\text{attribute-val2}\rangle \rightarrow \langle\text{class value}\rangle \\
  \ldots
  \]

- To see how 1R learns the rules, let's consider an example.

---

Applying 1R to the Credit-Card Promotion Data

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Let's start by determining the rules based on Sex.

- To do so, we ask the following:
  - when Sex = Female, what is the most frequent class?
  - when Sex = Male, what is the most frequent class?
Applying 1R to the Credit-Card Promotion Data (cont.)

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• Thus, we end up with the following rules based on Sex:

Sex: Female $\rightarrow$ Yes (6 out of 7)
Male $\rightarrow$ No (5 out of 8)

Pseudocode for the 1R Algorithm

for each input attribute A:
  for each value V of A:
    count how often each class appears together with V
    find the most frequent class F
    add the rule $A = V \rightarrow F$ to the rules for A
  calculate and store the accuracy of the rules learned for A
choose the rules with the highest overall accuracy

• So far, we've learned the rules for the attribute Sex:

Sex: Female $\rightarrow$ Yes (6 out of 7)
Male $\rightarrow$ No (5 out of 8)

overall accuracy = ?

• Equivalently, we can focus on the error rate and minimize it.
  error rate of rules above = ?
Applying 1R to the Credit-Card Promotion Data (cont.)

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>35</td>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>33</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• What rules would be produced for Credit Card Insurance?

  Credit Card Insurance: Yes → No →

Applying 1R to the Credit-Card Promotion Data (cont.)

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>35</td>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>33</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• What rules would be produced for Income Range?

Handing Numeric Attributes

- To handle numeric attributes, we need to **discretize** the range of possible values into subranges called **bins** or **buckets**.

- One way is to sort the training instances by age and look for the binary (two-way) split that leads to the most accurate rules.

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance</th>
<th>Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

### Handling Numeric Attributes (cont.)

- Here’s one possible binary split for age:

  - the corresponding rules are:
    - Age: <= 39 → Yes (5 out of 6) overall accuracy: 10/15 = 67%
    - Age: > 39 → No (5 out of 9)

- The following is one of the splits with the best overall accuracy:

  - the corresponding rules are:
    - Age: <= 43 → Yes (9 out of 12) overall accuracy: 12/15 = 80%
    - Age: > 43 → No (3 out of 3)
### Summary of 1R Results

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
<th>Rule 4</th>
<th>Overall Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sex: Female</td>
<td>Male → No (5 out of 8)</td>
<td>Female → Yes (6 out of 7)</td>
<td>overall accuracy: 11/15 = 73%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credit Card Ins:</td>
<td>Yes → Yes (3 out of 3)</td>
<td>No → No* (6 out of 12)</td>
<td>overall accuracy: 9/15 = 60%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Income Range:</td>
<td>20-30K → No* (2 out of 4)</td>
<td>30-40K → Yes (4 out of 5)</td>
<td>40-50K → No (3 out of 4)</td>
<td>50-60K → Yes (2 out of 2)</td>
<td>overall accuracy: 11/15 = 73%</td>
</tr>
<tr>
<td>Age:</td>
<td>&lt;= 43 → Yes (9 out of 12)</td>
<td>&gt; 43 → No (3 out of 3)</td>
<td>overall accuracy: 12/15 = 80%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Because the rules based on Age have the highest overall accuracy on the training data, 1R selects them as the model.

### Special Case: Many-Valued Attributes

- 1R does not tend to work well with attributes that have many possible values.

- When such an attribute is present, 1R often ends up selecting its rules.
  - each rule applies to only a small number of examples, which tends to give them a high accuracy

- However, the rules learned for a many-valued attribute tend not to generalize well.
  - what is this called?
Special Case: Many-Valued Attributes (cont.)

- Example: let’s say we used 1R on this dataset.
  - what would be the accuracy of rules based on Patient ID#?

- We need to remove identifier fields before running 1R.

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Allergy</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
</tbody>
</table>

Special Case: Numeric Attributes

- The standard way of handling numeric attributes in 1R is a bit more complicated than the method we presented earlier.
  - allows for more than two bins/buckets
  - possible alternate discretization:

<table>
<thead>
<tr>
<th>Age:</th>
<th>19</th>
<th>27</th>
<th>29</th>
<th>35</th>
<th>38</th>
<th>39</th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>43</th>
<th>43</th>
<th>45</th>
<th>55</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life Ins:</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

  - what’s the problem with this discretization?

  - To avoid overfitting, you can specify a minimum bucket size – the smallest number of examples allowed in a given bucket.
Limitation of 1R

- 1R won't work well if many of the input attributes have fewer possible values than the class/output attribute does.

- Example: our medical diagnosis dataset

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Allergy</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
</tbody>
</table>

- there are three possible classes: Strep Throat, Cold, Allergy
- binary attributes like Fever produce rules that predict at most two of these classes:

\[
\text{Fever: } \begin{cases} 
\text{Yes} & \Rightarrow \text{Cold} \\
\text{No} & \Rightarrow \text{Allergy}
\end{cases}
\]

Using 1R as a Baseline

- When performing classification learning, 1R, can serve as a useful baseline.
  - compare the models from more complex algorithms to the model it produces
  - if a model has a lower accuracy than 1R, it probably isn't worth keeping

- It also gives insight into which of the input attributes has the most impact on the output attribute.
0 R: Another Useful Baseline

- The 0R algorithm learns a model that considers *none* of the input attributes!

- It simply predicts the majority class in the training data.

- Example: the credit-card training data
  - 9 examples in which the output is Yes
  - 6 examples in which the output is No
  - thus, the 0R model would always predict Yes.
    - gives an accuracy of 9/15 = 60%

- When performing classification learning, you should use the results of this algorithm to put your results in context.
  - if the 0R accuracy is high, you may want to create training data that is less skewed
  - *at the very least, you should include the class breakdown of your training and test sets in your report*

---

Review: Decision Trees for Classification

- We've already seen examples of decision-tree models.
  - example: the tree for our medical-diagnosis dataset:

```
   Swollen Glands
     Yes  No
    /    \
   Strep Throat  Fever
     Yes  No
        Yes  No
          Cold  Allergy
```

- what class would this decision tree assign to the following instance?

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Fever</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
1R and Decision Trees

- We can view the models learned by 1R as simple decision trees with only one decision.
  - here is the model that we learned for the credit-card data:

```
Age
<= 43  > 43
Yes    No
```

- here are the rules based on Income Range:

```
Income Range
20-30K 30-40K 40-50K 50-60K
No     Yes    No     Yes
```

Building Decision Trees

- How can we build decision trees that use multiple attributes?

- Here's the basic algorithm:
  1. apply 1R to the full set of attributes, but choose the attribute that "best divides" the examples into subgroups
  2. create a decision based on that attribute and put it in the appropriate place in the existing tree (if any)
  3. for each subgroup created by the new decision:
     * if the classifications of its examples are "accurate enough" or if there are no remaining attributes to use, do nothing
     * otherwise, repeat the process for the examples in the subgroup
Building Decision Trees (cont.)

- What does it mean to choose the attribute that "best divides" the training instances?
  - overall accuracy still plays a role
  - however, it's not as important, since subsequent decisions can improve the model's accuracy
  - in addition, we want to avoid letting the tree get too large, to prevent overfitting

- We'll compute a goodness score for each attribute's rules:
  \[
  \text{goodness} = \frac{\text{overall accuracy}}{N}
  \]
  where \( N \) = the number of subgroups that would need to be subdivided further if we chose this attribute.
  - dividing by \( N \) should help to create a smaller tree
- Special case: if \( N == 0 \) for an attribute, we'll select that attribute.

Building a Decision Tree for the Credit-Card Data

- Here are the rules we obtained for each attribute using 1R:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Rule</th>
<th>Accuracy</th>
<th>Goodness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sex: Female</td>
<td>Female (\rightarrow) Yes</td>
<td>11/15 = 73%</td>
<td>goodness: 73/2 = 36.5</td>
</tr>
<tr>
<td></td>
<td>Male (\rightarrow) No</td>
<td>5/8</td>
<td>goodness: 60/1 = 60</td>
</tr>
<tr>
<td>Cred.Card Ins: Yes</td>
<td>Yes (\rightarrow) Yes</td>
<td>3/3</td>
<td>goodness: 73/3 = 24.3</td>
</tr>
<tr>
<td></td>
<td>No (\rightarrow) No*</td>
<td>6/12</td>
<td></td>
</tr>
<tr>
<td>Income Rng: 20-30K</td>
<td>No*</td>
<td>2/4</td>
<td>accuracy: 11/15 = 73%</td>
</tr>
<tr>
<td></td>
<td>Yes (\rightarrow) Yes</td>
<td>4/5</td>
<td>goodness: 73/3 = 24.3</td>
</tr>
<tr>
<td></td>
<td>No (\rightarrow) No</td>
<td>3/4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30-40K (\rightarrow) Yes</td>
<td>4/5</td>
<td>accuracy: 11/15 = 73%</td>
</tr>
<tr>
<td></td>
<td>40-50K (\rightarrow) No</td>
<td>3/4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50-60K (\rightarrow) Yes</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td>Age: (\leq 43)</td>
<td>Yes (\rightarrow) Yes</td>
<td>9/12</td>
<td>accuracy: 12/15 = 80%</td>
</tr>
<tr>
<td></td>
<td>No (\rightarrow) No</td>
<td>3/3</td>
<td>goodness: 80/1 = 80</td>
</tr>
<tr>
<td></td>
<td>(&gt; 43) (\rightarrow) No</td>
<td>3/3</td>
<td></td>
</tr>
</tbody>
</table>
Because Age has the highest goodness score, we use it as
the first decision in the tree:

- Nothing further needs to be done to the Age > 43 subgroup.
- We return to step 2 and apply the same procedure to
  the Age <= 43 subgroup.
  - this is an example of recursion: applying the same algorithm
    to a smaller version of the original problem

Here are the 12 examples in the Age <= 43 subgroup:

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>38</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>41</td>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>39</td>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

As before, we sort the examples by Age and find the most
accurate binary split.
- we'll use a minimum bucket size of 3

<table>
<thead>
<tr>
<th>Age:</th>
<th>19</th>
<th>27</th>
<th>29</th>
<th>35</th>
<th>38</th>
<th>39</th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>43</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life Ins:</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
Here are the rules obtained for these 12 examples:

<table>
<thead>
<tr>
<th>Sex: Female</th>
<th>Yes (6 out of 6)</th>
<th>accuracy: 9/12 = 75%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>No* (3 out of 6)</td>
<td>goodness: 75/1 = 75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cred. Card Ins: Yes</th>
<th>Yes (3 out of 3)</th>
<th>accuracy: 9/12 = 75%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No (6 out of 9)</td>
<td>goodness: 75/1 = 75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Income Rng: 20-30K</th>
<th>No (2 out of 3)</th>
<th>accuracy: 9/12 = 75%</th>
</tr>
</thead>
<tbody>
<tr>
<td>30-40K</td>
<td>Yes (4 out of 5)</td>
<td>goodness: 75/3 = 25</td>
</tr>
<tr>
<td>40-50K</td>
<td>No* (1 out of 2)</td>
<td></td>
</tr>
<tr>
<td>50-60K</td>
<td>Yes (2 out of 2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age: &lt;= 41</th>
<th>Yes (7 out of 8)</th>
<th>accuracy: 9/12 = 75%</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 41</td>
<td>No (2 out of 4)</td>
<td>goodness: 75/2 = 37.5</td>
</tr>
</tbody>
</table>

- Sex and Credit Card Insurance are tied for the highest goodness score.
- We'll pick Sex since it has more examples in the subgroup that doesn't need to be subdivided further.

Here's the tree that splits the Age <= 43 subgroup:

- It replaces the classification for that subgroup in the earlier tree:
Building a Decision Tree for the Credit-Card Data (cont.)

- We now recursively apply the same procedure to the 6 examples in the (Age <= 43, Sex = Male) subgroup:

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>35</td>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

sort by Age: 27 29 35 42 43 43

Life Ins: N Y Y N N N

- We no longer consider Sex. Why?

Building a Decision Tree for the Credit-Card Data (cont.)

- Here are the rules obtained for these 6 examples:

<table>
<thead>
<tr>
<th>Credit Card Insurance</th>
<th>Age &lt;= 35</th>
<th>Age &gt; 35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Income Range</th>
<th>Credit Insurance</th>
<th>Age &lt;= 35</th>
<th>Income Range</th>
<th>Credit Insurance</th>
<th>Age &gt; 35</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-30K</td>
<td>No</td>
<td>Yes</td>
<td>30-40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>30-40K</td>
<td>Yes</td>
<td>No</td>
<td>40-50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>50-60K</td>
<td>?</td>
<td>(none)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Credit Card Insurance has the highest goodness score, so we pick it and create the partial tree at right:
Building a Decision Tree for the Credit-Card Data (cont.)

- This new tree replaces the classification for the (Age <= 43, Sex = Male) subgroup in the previous tree:

![Decision Tree Diagram]

- Here are the four instances in the (Age <= 43, Sex = Male, Cred.Card Ins = No) subgroup:

<table>
<thead>
<tr>
<th>Age</th>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>27</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>43</td>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>29</td>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Sorted by Age:
- Age: 27 29 42 43
- Life Ins: N Y N N

- The only remaining attributes are Age and Income Range.
  - Income Range won't help, because there are two instances with Income Range = 20-30K, one with Life Ins = Yes class and one with Life Ins = No.
  - Age won't help, because we can't make a binary split that separates the Life Ins = Yes and Life Ins = No instances.

- Thus, the algorithm stops here.
Building a Decision Tree for the Credit-Card Data (cont.)

- Here’s the final model:

- It manages to correctly classify all but one training example.

Building a Decision Tree for the Credit-Card Data (cont.)

- How would it classify the following instance?

- Age
  - <= 43
  - > 43

- Sex
  - Female
  - Male

- Credit Card Insurance
  - Yes
  - No

- Income
  - Range: 20–30K

- Life Insurance
  - Promotion: ?
Other Algorithms for Learning Decision Trees

- ID3 – uses a different goodness score based on a field of study known as information theory
  - it can't handle numeric attributes

- C4.5 – makes a series of improvements to ID3:
  - the ability to handle numeric input attributes
  - the ability to handle missing values
  - measures that prune the tree after it is built – making it smaller to improve its ability to generalize

- Both ID3 and C4.5 were developed by Ross Quinlan of the University of Sydney.

- Weka's implementation of C4.5 is called J48.

Decision Tree Results in Weka

- Weka's output window gives the tree in text form that looks something like this:

```
J48 pruned tree
---------------
Sex = Male
| Credit Card Ins. = No: No (6.0/1.0)
| Credit Card Ins. = Yes: Yes (2.0)
Sex = Female: Yes (7.0/1.0)
```

```
total # of examples in this subgroup
# that are misclassified
```
Decision Tree Results in Weka (cont.)

- Right-clicking the name of the model in the result list allows you to view the tree in graphical form.

From Decision Trees to Classification Rules

- Any decision tree can be turned into a set of rules of the following form:

  \[
  \text{if } <test1> \text{ and } <test2> \text{ and ... then } <class> = <value>
  \]

  were the condition is formed by combining the tests used to get from the top of the tree to one of the leaves.
• Here are the rules for this tree:

\[
\begin{align*}
&\text{if } \text{Age} > 43 \\
&\quad \text{then Life Ins} = \text{No} \\
&\text{if } \text{Age} \leq 43 \text{ and } \text{Sex} = \text{Female} \\
&\quad \text{then Life Ins} = \text{Yes} \\
&\text{if } \text{Age} \leq 43 \text{ and } \text{Sex} = \text{Male and Credit Card Ins} = \text{Yes} \\
&\quad \text{then Life Ins} = \text{Yes} \\
&\text{if } \text{Age} \leq 43 \text{ and } \text{Sex} = \text{Male and Credit Card Ins} = \text{No} \\
&\quad \text{then Life Ins} = \text{No}
\end{align*}
\]

Advantages and Disadvantages of Decision Trees

• Advantages:
  • easy to understand
  • can be converted to a set of rules
    • makes it easier to actually use the model for classification
    • can handle both categorical and numeric input attributes
      • except for ID3, which is limited to categorical
  • Disadvantages:
    • the class attribute must be categorical
    • slight changes in the set of training examples can produce a significantly different decision tree
      • we say that the tree-building algorithm is \textit{unstable}
Practice Building a Decision Tree

- Let's apply our decision-tree algorithm to the diagnosis dataset.
- to allow us to practice with numeric attributes, I've replaced Fever with Temp – the person's body temperature

<table>
<thead>
<tr>
<th>Patient ID#</th>
<th>Sore Throat</th>
<th>Temp</th>
<th>Swollen Glands</th>
<th>Congestion</th>
<th>Headache</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>100.4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Strep throat</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>97.8</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>101.2</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>98.6</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>102.0</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Cold</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>99.2</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Allergy</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>98.1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Strep throat</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>98.0</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Allergy</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>102.5</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>100.7</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Cold</td>
</tr>
</tbody>
</table>
Review: Numeric Estimation

- *Numeric estimation* is like classification learning.
  - it involves learning a model that works like this:

    $\text{input attributes} \rightarrow \text{model} \rightarrow \text{output attribute/class}$

  - the model is learned from a set of training examples that include the output attribute

- In numeric estimation, the output attribute is numeric.
  - we want to be able to *estimate* its value
Example Problem: CPU Performance

• We want to predict how well a CPU will perform on some task, given the following info. about the CPU and the task:
  • CTIME: the processor's cycle time (in nanosec)
  • MMIN: minimum amount of main memory used (in KB)
  • MMAX: maximum amount of main memory used (in KB)
  • CACHE: cache size (in KB)
  • CHMIN: minimum number of CPU channels used
  • CHMAX: maximum number of CPU channels used

• We need a model that will estimate a performance score for a given combination of values for these attributes.

Example Problem: CPU Performance (cont.)

• The data was originally published in a 1987 article in the Communications of the ACM by Phillip Ein-Dor and Jacob Feldmesser of Tel-Aviv University.

• There are 209 training examples. Here are five of them:

<table>
<thead>
<tr>
<th>CTIME</th>
<th>MMIN</th>
<th>MMAX</th>
<th>CACHE</th>
<th>CHMIN</th>
<th>CHMAX</th>
<th>PERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>256</td>
<td>6000</td>
<td>256</td>
<td>16</td>
<td>128</td>
<td>198</td>
</tr>
<tr>
<td>29</td>
<td>8000</td>
<td>32000</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>269</td>
</tr>
<tr>
<td>29</td>
<td>8000</td>
<td>32000</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>172</td>
</tr>
<tr>
<td>125</td>
<td>2000</td>
<td>8000</td>
<td>0</td>
<td>2</td>
<td>14</td>
<td>52</td>
</tr>
<tr>
<td>480</td>
<td>512</td>
<td>8000</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>67</td>
</tr>
</tbody>
</table>
Linear Regression

• The classic approach to numeric estimation is \textit{linear regression}.

• It produces a model that is a linear function (i.e., a weighted sum) of the input attributes.
  • example for the CPU data:
    \[
    \text{PERF} = 0.066 \times \text{CTIME} + 0.0143 \times \text{MMIN} + 0.0066 \times \text{MMAX} + 0.4945 \times \text{CACHE} - 0.1723 \times \text{CHMIN} + 1.2012 \times \text{CHMAX} - 66.48
    \]
  • this type of model is known as a \textit{regression equation}

• The general format of a linear regression equation is:
  \[
y = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + c
\]
where
  \(y\) is the output attribute
  \(x_1, \ldots, x_n\) are the input attributes
  \(w_1, \ldots, w_n\) are numeric weights
  \(c\) is an additional numeric constant}

\textit{linear regression} learns these values

\hspace{1cm}

Linear Regression (cont.)

• Once the regression equation is learned, it can estimate the output attribute for previously unseen instances.
  • example: to estimate CPU performance for the instance

\[
\begin{array}{cccccccc}
\text{CTIME} & \text{MMIN} & \text{MMAX} & \text{CACHE} & \text{CHMIN} & \text{CHMAX} & \text{PERF} \\
480 & 1000 & 4000 & 0 & 0 & 0 & ?
\end{array}
\]

we plug the attribute values into the regression equation:

\[
\text{PERF} = 0.066 \times 480 + 0.0143 \times 1000 + 0.0066 \times 4000 + 0.4945 \times 0 - 0.1723 \times 0 + 1.2012 \times 0 - 66.48
\]

\[
= 0.066 \times 480 + 0.0143 \times 1000 + 0.0066 \times 4000 + 0.4945 \times 0 - 0.1723 \times 0 + 1.2012 \times 0 - 66.48
\]

\[
= 5.9
\]
Linear Regression with One Input Attribute

- Linear regression is easier to understand when there's only one input attribute, $x_1$.

- In that case:
  - the training examples are ordered pairs of the form $(x_1, y)$ shown as points in the graph above
  - the regression equation has the form $y = w_1 x_1 + c$ shown as the line in the graph above
  - $w_1$ is the slope of the line; $c$ is the y-intercept

- Linear regression finds the line that "best fits" the training examples.

Linear Regression with One Input Attribute (cont.)

- The dotted vertical bars show the differences between:
  - the actual $y$ values (the ones from the training examples)
  - the estimated $y$ values (the ones given by the equation)

Why do these differences exist?

- Linear regression finds the parameter values ($w_1$ and $c$) that minimize the sum of the squares of these differences.
Linear Regression with Multiple Input Attributes

- When there are $k$ input attributes, linear regression finds the equation of a line in $(k+1)$ dimensions.
  - here again, it is the line that "best fits" the training examples

- The equation has the form we mentioned earlier:
  $$y = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + c$$

- Here again, linear regression finds the parameter values (for the weights $w_1, \ldots, w_n$ and constant $c$) that minimize the sum of the squares of the differences between the actual and predicted $y$ values.

Linear Regression in Weka

- Use the Classify tab in the Weka Explorer.

- Click the Choose button to change the algorithm.
  - linear regression is in the folder labelled functions

- By default, Weka employs attribute selection, which means it may not include all of the attributes in the regression equation.
Linear Regression in Weka (cont.)

- On the CPU dataset with M5 attribute selection, Weka learns the following equation:
  \[
  \text{PERF} = 0.0661 \text{CTIME} + 0.0142 \text{MMIN} + 0.0066 \text{MMAX} + \\
  0.4871 \text{CACHE} + 1.1868 \text{CHMAX} - 66.60
  \]
  - it does not include the CHMIN attribute

- To eliminate attribute selection, you can click on the name of the algorithm and change the `attributeSelectionMethod` parameter to "No attribute selection".
  - doing so produces our earlier equation:
    \[
    \text{PERF} = 0.066 \text{CTIME} + 0.0143 \text{MMIN} + 0.0066 \text{MMAX} + \\
    0.4945 \text{CACHE} - 0.1723 \text{CHMIN} + 1.2012 \text{CHMAX} - 66.48
    \]

- Notes about the coefficients:
  - what do the signs of the coefficients mean?
  - what about their magnitudes?

Evaluating a Regression Equation

- To evaluate the goodness of a regression equation, we again set aside some of the examples for testing.
  - do not use these examples when learning the equation
  - use the equation on the test examples and see how well it does

- Weka provides a variety of error measures, which are based on the differences between the actual and estimated y values.
  - we want to minimize them

- The correlation coefficient measures the degree of correlation between the input attributes and the output attribute.
  - its absolute value is between 0.0 and 1.0
  - we want to maximize its absolute value
Simple Linear Regression

- This algorithm in Weka creates a regression equation that uses only one of the input attributes.
  - even when there are multiple inputs

- Like 1R, simple linear regression can serve as a baseline.
  - compare the models from more complex algorithms to the model it produces

- It also gives insight into which of the inputs has the largest impact on the output.

Handling Non-Numeric Input Attributes

- We employ numeric estimation when the output attribute is numeric.

- Some algorithms for numeric estimation also require that the input attributes be numeric.

- If we have a non-numeric input attribute, it may be possible to convert it to a numeric one.
  - example: if we have a binary attribute (yes/no or true/false), we can convert the two values to 0 and 1

- In Weka, many algorithms – including linear regression – will automatically adapt to non-numeric inputs.
Handling Non-Numeric Input Attributes (cont.)

- There are algorithms for numeric estimation that are specifically designed to handle both numeric and nominal attributes.

- One option:
  - build a decision tree
  - have each classification be a numeric value that is the average of the values for the training examples in that subgroup
  - the result is called a **regression tree**

- Another option is to have a separate regression equation for each classification in the tree – based on the training examples in that subgroup.
  - this is called a **model tree**
Regression and Model Trees in Weka

- To build a tree for estimation in Weka, select the M5P algorithm in the trees folder.
  - by default, it builds model trees
  - you can click on the name of the algorithm and tell it to build regression trees
Review: Association Learning

- Algorithms for association learning:
  - take a set of training examples
  - discover associations among attributes
    - example: products that people tend to purchase together

- Unlike classification learning, association learning doesn't single out a single attribute for special treatment.
  - the resulting model may determine the value of any of the attributes – or even of combinations of attributes
Association Rules

- The learned associations are usually expressed as rules known as association rules. Examples:
  
  \[
  \text{if PurchaseDiapers = Yes}
  \quad \text{then PurchaseBeer = Yes}
  \]
  
  \[
  \text{if PurchaseMilk = Yes and PurchaseJuice = Yes}
  \quad \text{then PurchaseEggs = Yes and PurchaseCheese = Yes}
  \]

- The test or tests in the if clause of a rule is known as the precondition of the rule.

- The assignment in the then clause of a rule is known as the conclusion of the rule.

- General format:
  
  \[
  \text{if <precondition>}
  \quad \text{then <conclusion>}
  \]

The Converse of a Rule

- The converse of a rule is obtained by swapping the precondition and conclusion.

  - example: here's one rule:
    
    \[
    \text{if PurchaseDiapers = Yes}
    \quad \text{then PurchaseBeer = Yes}
    \]

    its converse is:

    \[
    \text{if PurchaseBeer = Yes}
    \quad \text{then PurchaseDiapers = Yes}
    \]

- The converse of a rule is not necessarily true.

  - example: this rule is true:
    
    \[
    \text{if name = 'Perry Sullivan'}
    \quad \text{then yearBorn = 2000}
    \]

    its converse is not!

    \[
    \text{if yearBorn = 2000}
    \quad \text{then name = 'Perry Sullivan'}
    \]
Example Problem: Credit-Card Promotions

• We'll use these training examples, which omit the Age attribute:

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>50–60K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• Possible association rules include:
  
  if Sex = Male and IncomeRange = 40–50K
  then CreditCardIns = No and LifeIns = No

  if CreditCardIns = Yes and LifeIns = Yes
  then Sex = Male

Metric #1: Support

• The support of a rule is the number of training examples containing the attribute values found in both the rule's precondition and its conclusion.
  • i.e., the number of examples that the rule gets right
  • these examples "support" the rule

• This metric can also be expressed as a percentage of the total number of training examples.
Metric #1: Support (cont.)

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

If Sex = Male and IncomeRange = 40-50K then CreditCardIns = No and LifeIns = No
• support = 3 instances (or 20% of the total training set)

If CreditCardIns = Yes and LifeIns = Yes then Sex = Male
• support = 2 instances (or 13.3% of the total training set)

Metric #2: Confidence

• The confidence of a rule provides a measure of a rule’s accuracy – of how well it predicts the values in the conclusion.

• It answers the question: if the precondition of the rule holds, how likely is it that the conclusion also holds?

• Here’s the formula:

$$\text{confidence} = \frac{\text{# examples with the values in the precondition and the conclusion}}{\text{# examples with the values in just the precondition}}$$

> the support
**Metric #2: Confidence (cont)**

if Sex = Male and IncomeRange = 40-50K then CreditCardIns = No and LifeIns = No

- confidence = 
  \[
  \frac{\text{# examples with Sex=Male and IncomeRange=40-50K}}{3} = \frac{3}{3} = 1 \text{ or } 100\% \quad \text{(perfect accuracy on training examples)}
  \]

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

if CreditCardIns = Yes and LifeIns = Yes then Sex = Male

- confidence = 
  \[
  \frac{\text{# examples with CreditCardIns=Yes and LifeIns=Yes}}{3} = \frac{2}{3} = .667 \text{ or } 66.7\%
  \]

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Practice: Support and Confidence

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Female</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Female</td>
</tr>
</tbody>
</table>

if LifeIns = Yes and CreditCardIns = No
then Sex = Female

• support = ?
• confidence = ?

Learning Association Rules

• For a given dataset, there are a large number of association rules that could be learned.
  • example:
    ```
    if CreditCardIns = Yes and LifeIns = Yes and IncomeRange = 20-30K
    then Sex = Female
    ```
    has a confidence of 100%, but it is only based on a single example (i.e., its support = 1)

• To cut down the number of rules that we consider, we limit ourselves to ones with sufficient support.

• Of these rules, we keep the most accurate ones – the ones with a confidence value that is above some minimum value.
Item Sets

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- An item set is a collection of attribute values that appears in one or more training examples.
- example: the item set \text{CreditCardIns=Yes, LifeIns=Yes} appears in 3 training examples
- it could be used to form two different rules with support = 3:
  - if \text{CreditCardIns = Yes} then \text{LifeIns = Yes}
  - if \text{LifeIns = Yes} then \text{CreditCardIns = Yes}

Apriori Algorithm for Learning Association Rules

- The standard algorithm for learning association rules is called the \textit{apriori algorithm}.

- It has two stages:
  1) gradually build up larger and larger item sets, keeping only the ones that appear in a sufficient number of training examples
     - this allows us to ensure that the rules formed from the item sets will have sufficient support
  2) form rules from the item sets and keep the ones with a confidence value that is \text{>=} some minimum value.
First Stage: Building Item Sets

- We'll limit ourselves to item sets that appear in >= 3 examples.

- We get 9 one-item sets that meet this criterion:
  
<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- We don't pair item sets that have different values for the same attribute, since such a pairing isn't possible.
  
  - example: Sex = Male and Sex = Female

First Stage: Building Item Sets (cont.)

- Next, we combine our one-item sets to form two-items sets, keeping those with support >= 3.

- We don't pair item sets that have different values for the same attribute, since such a pairing isn't possible.
  
  - example: Sex = Male and Sex = Female
First Stage: Building Item Sets (cont.)

• We end up with 15 two-item sets:
  Sex=Male, IncomeRange=30-40K
  Sex=Male, IncomeRange=40-50K
  Sex=Male, CreditCardIns=No
  Sex=Male, LifeIns=Yes
  Sex=Male, LifeIns=No
  Sex=Female, CreditCardIns=No
  Sex=Female, LifeIns=Yes
  IncomeRange=20-30K, CreditCardIns=No
  IncomeRange=30-40K, CreditCardIns=No
  IncomeRange=30-40K, LifeIns=Yes
  IncomeRange=40-50K, CreditCardIns=No
  IncomeRange=40-50K, LifeIns=No
  CreditCardIns=Yes, LifeIns=Yes
  CreditCardIns=No, LifeIns=Yes
  CreditCardIns=No, LifeIns=No

• Within an item set, we write the items in the order given by the columns in the dataset file.

First Stage: Building Item Sets (cont.)

• To form three-item sets, we take the union of pairs of two-item sets with the same first item. Example:
  \[\text{Sex}=\text{Male}, \text{IncomeRange}=30-40K \cup \text{Sex}=\text{Male}, \text{LifeIns}=Yes\]
  \[= \text{Sex}=\text{Male}, \text{IncomeRange}=30-40K, \text{LifeIns}=Yes\]

• It isn't necessary to consider any other pairs of two-item sets, even if they share an item in common.
  • example: although we could do
    \[\text{Sex}=\text{Male}, \text{LifeIns}=Yes \cup \text{CreditCardIns}=No, \text{LifeIns}=Yes\]
    \[= \text{Sex}=\text{Male}, \text{CreditCardIns}=No, \text{LifeIns}=Yes\]

    we don't need to, because either:
    1) the resulting item set will be generated by two other item sets, S1 and S2, with the same first item:
      \[\text{Sex}=\text{Male}, \text{CreditCardIns}=No \cup \text{Sex}=\text{Male}, \text{LifeIns}=Yes\]
    2) one or both of S1 and S2 didn't have enough support, and thus the resulting item set won't either
Practice: Taking the Union of Item Sets

- What possible three-item sets could we form from the following two-item sets?
  
  \[
  \begin{align*}
  \text{Sex} &= \text{Male}, \text{IncomeRange}=30-40K \\
  \text{Sex} &= \text{Male}, \text{IncomeRange}=40-50K \\
  \text{Sex} &= \text{Male}, \text{CreditCardIns}=\text{No} \\
  \text{IncomeRange} &= 30-40K, \text{CreditCardIns}=\text{No} \\
  \text{IncomeRange} &= 30-40K, \text{LifeIns}=\text{Yes}
  \end{align*}
  \]

First Stage: Building Item Sets (cont.)

- Out of the potential three-items sets, only 5 have sufficient support – appearing in at least 3 examples:
  
  \[
  \begin{align*}
  \text{Sex} &= \text{Male}, \text{IncomeRange}=40-50K, \text{CreditCardIns}=\text{No} \\
  \text{Sex} &= \text{Male}, \text{IncomeRange}=40-50K, \text{LifeIns}=\text{No} \\
  \text{Sex} &= \text{Male}, \text{CreditCardIns}=\text{No}, \text{LifeIns}=\text{No} \\
  \text{Sex} &= \text{Female}, \text{CreditCardIns}=\text{No}, \text{LifeIns}=\text{Yes} \\
  \text{IncomeRange} &= 40-50K, \text{CreditCardIns}=\text{No}, \text{LifeIns}=\text{No}
  \end{align*}
  \]
First Stage: Building Item Sets (cont.)

- To form potential four-item sets, we take the union of pairs of three-item sets with the same first two items.
  - more generally, to form n-item sets, we take the union of pairs of (n – 1)-item sets with the same first n – 2 items

- We get only one potential four-item set:
  \( \text{Sex=Male, IncomeRange}=40-50K, \text{Credit Card Ins}=No, \text{Life Ins}=No \)
  and it has enough support.

- There can't be any five-item sets (because there are only four attributes), so we're done building sets.

Results of the First Stage

- Here are all item sets with two or more items and support >= 3:

  \[
  \begin{align*}
  &\text{Sex=Male, IncomeRange}=30-40K, &\text{Sex=Female, Credit Card Ins}=No \\
  &\text{Sex=Male, IncomeRange}=40-50K, &\text{Sex=Female, Life Ins}=Yes \\
  &\text{Sex=Male, Credit Card Ins}=No, &\text{Credit Card Ins}=Yes, \text{Life Ins}=Yes \\
  &\text{Sex=Male, Life Ins}=Yes, &\text{Credit Card Ins}=No, \text{Life Ins}=Yes \\
  &\text{Sex=Male, Life Ins}=No, &\text{Credit Card Ins}=No, \text{Life Ins}=No \\
  &\text{IncomeRange}=20-30K, \text{Credit Card Ins}=No \\
  &\text{IncomeRange}=30-40K, \text{Credit Card Ins}=No \\
  &\text{IncomeRange}=30-40K, \text{Life Ins}=Yes \\
  &\text{IncomeRange}=40-50K, \text{Credit Card Ins}=No \\
  &\text{IncomeRange}=40-50K, \text{Life Ins}=No \\
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{Sex=Male, IncomeRange}=40-50K, \text{Credit Card Ins}=No \\
  &\text{Sex=Male, IncomeRange}=40-50K, \text{Life Ins}=No \\
  &\text{Sex=Male, Credit Card Ins}=No, \text{Life Ins}=No \\
  &\text{Sex=Female, Credit Card Ins}=No, \text{Life Ins}=Yes \\
  &\text{IncomeRange}=40-50K, \text{Credit Card Ins}=No, \text{Life Ins}=No \\
  \end{align*}
  \]
Second Stage: Forming the Rules

- A given item set can produce a number of potential rules.
  
  - example: the item set
    
    \[
    \text{Sex=}\text{Male}, \text{IncomeRange=}40-50K, \text{CreditCardIns=}\text{No}
    \]

  produces the following potential rules:

  a) if \(\text{Sex=}\text{Male and IncomeRange=}40-50K\) then \(\text{CreditCardIns=}\text{No}\)

  b) if \(\text{Sex=}\text{Male and CreditCardIns=}\text{No}\) then \(\text{IncomeRange=}40-50K\)

  c) if \(\text{IncomeRange=}40-50K\) and \(\text{CreditCardIns=}\text{No}\) then \(\text{Sex=}\text{Male}\)

  d) if \(\text{Sex=}\text{Male}\) then \(\text{IncomeRange=}40-50K\) and \(\text{CreditCardIns=}\text{No}\)

  e) if \(\text{IncomeRange=}40-50K\) then \(\text{Sex=}\text{Male and CreditCardIns=}\text{No}\)

  f) if \(\text{CreditCardins=}\text{No}\) then \(\text{Sex=}\text{Male and IncomeRange=}40-50K\)

- We keep only the ones with confidence \(\geq \) some min value.

---

Second Stage: Forming the Rules (cont.)

<table>
<thead>
<tr>
<th>Sex</th>
<th>Income Range</th>
<th>Credit Card Insurance</th>
<th>Life Insurance Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Male</td>
<td>30–40K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>30–40K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>40–50K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>20–30K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>50–60K</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Male</td>
<td>40–50K</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Female</td>
<td>20–30K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Example: assume we require confidence = 1.0 (100% accuracy). Would either of the following rules be kept?

  a) if \(\text{Sex=}\text{Male and IncomeRange=}40-50K\) then \(\text{CreditCardins=}\text{No}\) support = ? confidence = ?

  b) if \(\text{Sex=}\text{Male and CreditCardins=}\text{No}\) then \(\text{IncomeRange=}40-50K\) support = ? confidence = ?
Second Stage: Forming the Rules (cont.)

• In our example, there are 13 rules with conf = 1.0:

1) if LifeIns=No
    then CreditCardIns=No
2) if Sex=Male and LifeIns=No
    then CreditCardIns
3) if IncomeRange=40-50K
    then CreditCardIns=No
4) if Sex=Male and IncomeRange=40-50K
    then CreditCardIns=No and LifeIns=No
5) if IncomeRange=40-50K and LifeIns=No
    then Sex=Male and CreditCardIns=No
6) if Sex=Male and IncomeRange=40-50K and
    CreditCardIns=No
    then LifeIns=No
7) if Sex=Male and IncomeRange=40-50K and
    LifeInsPromo=No
    then CreditCardIns=No
8) if IncomeRange=40-50K and CreditCardIns=No and
    LifeInsPromo=No
    (continued)

Second Stage: Forming the Rules (cont.)

• 13 rules (cont.)

9. if IncomeRange=40-50K and LifeIns=No
    then CreditCardIns=No
10. if Sex=Male and IncomeRange=40-50K
    then LifeInsPromo=No
11. if IncomeRange=40-50K and LifeInsPromo=No
    then Sex=Male
12. if Sex=Male and IncomeRange=40-50K
    then CreditCardIns=No
13. if CreditCardIns=Yes
    then LifeIns=Yes
Learning Association Rules in Weka

- Use the *Associate* tab in the Weka Explorer.

- Apriori is the default algorithm, but you may want to click on its name and change some of the parameters, such as:
  - `lowerBoundMinSupport`: the minimum support that item sets must have, as a percentage of the training examples (expressed in decimal form)
    - in our example, this would be $3/15 = 0.2$
  - `minMetric`: the minimum confidence that the rules must have (expressed in decimal form)
    - in our example, this would be 1.0
  - `numRules`: the number of rules you want to find
  - `outputItemSets`: do you want to see the item sets that were formed?

Learning Association Rules in Weka (cont.)

- When it outputs item sets, Weka includes the support of each item set (as a number, not a percentage).
  - example:
    ```
    Sex=Male IncomeRange=40-50000   3
    Sex=Male IncomeRange=30-40000   3
    Sex=Male CreditCardIns=No       6
    ```

- Weka uses the following format for the rules:
  - `precondition  →  conclusion`
  - example
    ```
    LifeIns=No 6 ==> CreditCardIns=No 6  conf:(1)
    ```
    which is equivalent to
    ```
    if LifeIns=No
    then CreditCardIns=No
    ```
  - note that Weka includes both support and confidence values with the rule
Managing the Efficiency of the Algorithm

- The apriori algorithm tries to generate the rules efficiently – i.e., taking as few steps as possible.
  - this is extremely important, since the number of possible rules increases exponentially with the number of attributes and the number of possible values of each attribute.

- We've already seen some ways that it does this:
  - by only considering item sets with sufficient support
  - by building larger item sets from smaller ones that have enough support

- It also builds rules with larger conclusions (i.e., with more attributes in the then clause) from rules with smaller conclusions.

- Even with these steps, the algorithm may take too long for very large datasets.

Managing the Efficiency of the Algorithm (cont.)

- To improve the efficiency even further, we can:
  - specify a large initial support value
    - the larger the support value, the sooner the first phase will finish
  - have the algorithm gradually decrease this support value and rerun the algorithm until it has generated enough rules
    - the delta parameter in Weka specifies how much the support should be decreased each time
Data Mining V: Preparing the Data

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

The Data Mining Process

- Key steps:
  - assemble the data in the format needed for data mining
    - typically a text file
  - perform the data mining
  - interpret/evaluate the results
  - apply the results
Denormalization

- Recall: in designing a database, we try to avoid redundancies by normalizing the data.
- As a result, the data for a given entity (e.g., a customer) may be:
  - spread over multiple tables
  - spread over multiple records within a given table
- To prepare for data warehousing and/or data mining, we often need to denormalize the data.
  - multiple records for a given entity $\rightarrow$ a single record
- Example: finding associations between courses students take.
  - our university database has three relevant relations: Student, Course, and Enrolled
  - we might need to combine data from all three to create the necessary training examples

Transforming the Data

- We may also need to reformat or transform the data.
  - we can use a Python program to do the reformatting!
- One reason for transforming the data: many machine-learning algorithms can only handle certain types of data.
  - some algorithms only work with nominal attributes – attributes with a specified set of possible values
    - examples: \{yes, no\}
    - \{strep throat, cold, allergy\}
  - other algorithms only work with numeric attributes
Discretizing Numeric Attributes

- We can turn a numeric attribute into a nominal/categorical one by using some sort of discretization.

- This involves dividing the range of possible values into subranges called buckets or bins.
  - example: an age attribute could be divided into these bins:
    - child: 0-12
    - teen: 12-17
    - young: 18-35
    - middle: 36-59
    - senior: 60+

Simple Discretization Methods

- What if we don't know which subranges make sense?

- Equal-width binning divides the range of possible values into N subranges of the same size.
  - bin width = (max value – min value) / N
  - example: if the observed values are all between 0-100, we could create 5 bins as follows:
    width = (100 – 0)/5 = 20
    bins: [0-20], (20-40], (40-60], (60-80], (80-100]
      [ or ] means the endpoint is included
      ( or ) means the endpoint is not included
  - typically, the first and last bins are extended to allow for values outside the range of observed values
    (-infinity-20], (20-40], (40-60], (60-80], (80-infinity)
  - problems with this equal-width approach?
Simple Discretization Methods (cont.)

- *Equal-frequency* or *equal-height binning* divides the range of possible values into $N$ bins, each of which holds the same number of training instances.
  - example: let's say we have 10 training examples with the following values for the attribute that we're discretizing:
    
    5, 7, 12, 35, 65, 82, 84, 88, 90, 95
  
  to create 5 bins, we would divide up the range of values so that each bin holds 2 of the training examples:

  5, 7, 12, 35, 65, 82, 84, 88, 90, 95

- To select the boundary values for the bins, this method typically chooses a value halfway between the training examples on either side of the boundary.
  - examples: $(7 + 12)/2 = 9.5 \quad (35 + 65)/2 = 50$

- Problems with this approach?

Other Discretization Methods

- Ideally, we'd like to come up with bins that capture distinctions that will be useful in data mining.
  - example: if we're discretizing *body temperature*, we'd like the discretization method to learn that 98.6 F is an important boundary value
  - more generally, we want to capture distinctions that will help us to learn to predict/estimate the class of an example

- Both equal-width and equal-frequency binning are considered *unsupervised* methods, because they don't take into account the class values of the training examples.

- There are *supervised* methods for discretization that attempt to take the class values into account.
Discretization in Weka

- In Weka, you can discretize an attribute by applying the appropriate filter to it.

- After loading in the dataset in the Preprocess tab, click the Choose button in the Filter portion of the tab.

- For equal-width or equal-height, you choose the Discretize option in the filters/unsupervised/attribute folder.
  - by default, it uses equal-width binning
  - to use equal-frequency binning instead, click on the name of the filter and set the useEqualFrequency parameter to True

- For supervised discretization, choose the Discretize option in the filters/supervised/attribute folder.

Nominal Attributes with Numeric Values

- Some attributes that use numeric values may actually be nominal attributes.
  - the attribute has a small number of possible values
  - there is no ordering to the values, and you would never perform mathematical operations on them
  - example: an attribute that uses numeric codes for medical diagnoses
    - 1 = Strep Throat, 2 = Cold, 3 = Allergy

- If you load into Weka a comma-separated-value file containing such an attribute, Weka will assume that it is numeric.

- To force Weka to treat an attribute with numeric values as nominal, use the NumericToNominal option in the filters/unsupervised/attribute folder.
  - click on the name of the filter, and enter the number(s) of the attributes you want to convert
Removing Problematic Attributes

- Problematic attributes include:
  - irrelevant attributes: ones that don't help to predict the class
    - despite their irrelevance, the algorithm may erroneously include them in the model
  - attributes that cause overfitting
    - example: a unique identifier like Patient ID
  - redundant attributes: ones that offer basically the same information as another attribute
    - example: in many problems, date-of-birth and age provide the same information
    - some algorithms may end up giving the information from these attributes too much weight

- We can remove an attribute manually in Weka by clicking the checkbox next to the attribute in the Preprocess tab and then clicking the Remove button.

Undoing Preprocess Actions

- In the Preprocess tab, the Undo button allows you to undo actions that you perform, including:
  - applying a filter to a dataset
  - manually removing one or more attributes

- If you apply two filters without using Undo in between the two, the second filter will be applied to the results of the first filter.

- Undo can be pressed multiple times to undo a sequence of actions.
Dividing Up the Data File

- To allow us to validate the model(s) learned in data mining, we'll divide the examples into two files:
  - \( n\% \) for training
  - \( 100 - n\% \) for testing: these should not be touched until you have finalized your model or models
- possible splits:
  - 67/33
  - 80/20
  - 90/10
- You can use Weka to split the dataset for you after you perform whatever reformatting/editing is needed.
- If you discretize one or more attributes, you need to do so before you divide up the data file.
  - otherwise, the training and test sets will be incompatible

Dividing Up the Data File (cont.)

- Here's one way to do it in Weka:
  1) shuffle the examples by choosing the Randomize filter from the filters/unsupervised/instance folder
  2) save the entire file of shuffled examples in Arff format.
  3) use the RemovePercentage filter from the same folder to remove some percentage of the examples
    - whatever percentage you're using for the training set
    - click on the name of the filter to set the percentage
  4) save the remaining examples in a new file
    - this will be our test data
  5) load the full file of shuffled examples back into Weka
  6) use RemovePercentage again with the same percentage as before, but set invertSelection to True
  7) save the remaining examples in a new file
    - this will be our training data
Case Study:
Predicting Patient Outcomes

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

Dataset Description

• The "spine clinic dataset" from Roiger & Geatz.

• Data consists of records for 171 patients who had back surgery at a spine clinic.

• 31 attributes per record describing:
  • the patient's condition before and during surgery
  • the patient's condition three months after surgery
    • including whether he/she has been able to return to work

• Includes missing and erroneous information
Overview of the Data-Mining Task

• Goal: to develop insights into factors that influence patient outcomes – in particular, whether the patient can return to work.

• What type of data mining should we perform?

• What will the data mining produce?
Review: Preparing the Data

- Possible steps include:
  - denormalization
    - several records for a given entity → single training example
  - discretization
    - numeric → nominal
  - nominal → numeric
  - force Weka to realize that a seemingly numeric attribute is really nominal
  - remove ID attributes and other problematic attributes

Preparing the Data (cont.)

- We begin by loading the dataset (a CSV file) into Weka Explorer.

- It's helpful to examine each attribute by highlighting its name in the Attribute portion of the Preprocess tab.
  - helps us to identify missing/anomalous values
  - can also help us to discover formatting issues that should be addressed
Preparing the Data (cont.)

• Things worth noting about the attributes in this dataset:

• Steps we may want to take:

Review: Dividing Up the Data

• To allow us to validate the model(s) we learn, we'll divide the examples into two files:
  • n% for training
  • 100 – n% for testing
    • don't touch these until you've finalized your model(s)

• You can use Weka to split the dataset:
  1) filters/unsupervised/instance/Randomize
  2) save the shuffled examples in Arff format
  3) filters/unsupervised/instance/RemovePercentage
    • specify the percentage parameter to remove n%
  4) save the remaining examples as your test set
  5) load the full file of shuffled examples back into Weka
  6) use RemovePercentage with invertSelection set to True
to remove the other 100 – n%
  7) save the remaining examples as your training set
Experimenting with Different Techniques

• Use Weka to try different techniques on the training data.

• For each technique, examine:
  • the resulting model
  • the validation results
    • for classification models: overall accuracy, confusion matrix
    • for numeric estimation models: correlation coefficient, errors
    • for association-rule models: support, confidence

• If the model is something you can interpret, make sure it seems reasonable.

• Try to improve the validation results by:
  • changing the algorithm used
  • changing the algorithm’s parameters

Remember to Start with a Baseline

• For classification learning:
  • 0R
  • 1R

• For numeric estimation:
  • simple linear regression

• Include the results of these baselines to put your other results in context.
  • example: 80% accuracy isn't that impressive if 0R has 78% accuracy
  • being honest about your results is better than making exaggerated claims!
Cross Validation

• When validating classification/estimation models, Weka performs 10-fold cross validation by default:
  1) divides the training data into 10 subsets
  2) repeatedly does the following:
     a) holds out one of the 10 subsets
     b) builds a model using the other 9 subsets
     c) tests the model using the held-out subset
  3) reports results that average the 10 models together

• Note: the model reported in the output window is learned from all of the training examples.
  • the cross-validation results do not actually evaluate it

Reporting the Results

• Once you have settled on the algorithm(s) with the best cross-validation results, you should evaluate the resulting model(s) on both the training and test data.

• To see how well the reported model does on the training data, select Using training set in the Test box of the Classify tab and rerun the algorithm.

• To see how well the reported model does on the training data, select Supplied test set in the Test box of the Classify tab.
  • click the Set button to specify the file
  • rerun the algorithm

• Include appropriate metrics for each portion of your data:
  • classification learning: accuracy, confusion matrix
  • numeric estimation: correlation coefficient
Discussing the Results

- Your report should include more than just the numeric results.

- You should include an *intelligent discussion* of the results.
  - compare training vs. test results
  - how well do the models appear to generalize?
  - which attributes are included in the models?
  - for classification learning, what do the confusion matrices tell you about the types of examples that the models get right or get wrong?
  - for numeric estimation, which attributes have positive coefficients and which have negative?
    - note: the magnitude of the coefficients may *not* be significant
  - are the models intuitive? why or why not?

- Don't make overly confident claims!

Summary of Experiments

- Summary of experiments:
Summary of Experiments (cont.)
GenBank: A Database of Genetic Sequence Data

Computer Science 105
Boston University
David G. Sullivan, Ph.D.

An Explosion of Scientific Data

- Scientists are generating ever increasing amounts of data.
- Relevant units include:
  - terabyte (TB) = 1024 GB = $2^{40}$ (or approx. $10^{12}$) bytes
  - petabyte (PB) = 1024 TB = $2^{50}$ (or approx. $10^{15}$) bytes
    - equivalent to the text in one billion books!
    - it would take over 3 years to read this much info using a high-speed tape drive!
- The amount of scientific data doubles every year!
An Explosion of Scientific Data (cont.)

- Example: GenBank database of genetic sequences
  - on this graph, the data doubles every 12-14 months
  - as of May 2006, the doubling time was less than a year and getting shorter!

An Explosion of Scientific Data (cont.)

- The complexity of the data is also increasing.
  - need significant computing power to process it

- As a result, there is an increasing trend toward scientific data centers that manage large collections of data on the behalf of scientists.
  - provide tools for accessing/manipulating the data
GenBank

- A database of genetic sequence data (DNA, RNA, etc.)
  - contributed by scientists; May ‘06: ~11,000 submissions/day
- Managed by the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland
- NCBI collaborates with two other data centers:
  - Each repository has the same raw data (use a daily sync).
  - The derived data (e.g., info. on genes and proteins) is partially shared.

Genetics Primer

- Genome = the instructions for building an organism
  - recorded in DNA
- DNA is built from four fundamental units called bases:
  - adenosine (A), guanine (G), cytosine (C), and thymidine (T)
  - form a double helix: two strands connected together in the form a “twisted ladder”
    - one pair of bases = one rung of the ladder
  - each strand contains the same info.
    - A always pairs with T
    - C always pairs with G
  - the strands are tightly bundled to form chromosomes
Genetics Primer (cont.)

- A DNA sequence specifies the bases that appear in a given strand or substrand of DNA.
  - AGCTTTTCAATTCTGACTGCAACGGGCAATATGTC…
  - up to several hundred million bases per chromosome
  - in humans, this adds up to a total of ~3 billion bases

- Gene = a sequence of bases that is used to build a protein
  - example: the gene for eye color = the sequence of bases that specifies a protein that makes your eyes look blue, or brown, or …

Genomics Research

- The Human Genome Project has determined the full sequence of bases in the human genome.
  - everyone’s genome is unique (except identical twins), but 99.9% of the bases are the same in all people
  - the sequenced human genome is an average of sorts

- Researchers continue to add new sequence data:
  - from the genomes of other organisms
    - over 130,000 different organisms in GenBank
  - from the genomes of other human individuals

- Researchers compare sequences from different genomes to:
  - identify genes
  - gain insight into the role of a given gene
Data Storage in GenBank

- The data is stored in flat text files. Example record:

<table>
<thead>
<tr>
<th>LOCUS</th>
<th>NC_000913</th>
<th>4639675 bp</th>
<th>DNA</th>
<th>circular</th>
<th>BCT 15-OCT-2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINITION</td>
<td>Escherichia coli K12, complete genome.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCESSION</td>
<td>NC_000913</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION</td>
<td>NC_000913.2</td>
<td>GI: 49175990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYWORDS</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td>Escherichia coli K12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORGANISM</td>
<td>Escherichia coli K12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bacteria; Proteobacteria; Gammaproteobacteria; Enterobacteriales; Enterobacteriaceae; Escherichia.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCE</td>
<td>1 (bases 1 to 4639675)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TITLE</td>
<td>The complete genome sequence of Escherichia coli K-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOURNAL</td>
<td>Science 277 (5331), 1453-1474 (1997)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEDLINE</td>
<td>97426617</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUBMED</td>
<td>9278503</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORIGIN</td>
<td>1 agcttttcat tctgactgca acgggcaata tgcctcgggt tagtattaaa aaagagtgct</td>
<td>61 tgatagcagc ttctgaactg gttacctgcc gtgagtaaat taaaatttta ttgacttagg</td>
<td>121 tcactaaata cttaacccaa tataggcata ggcgccagc agataaaaaat tacagagtac</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stored in compressed form and available by FTP. 

Data Storage in GenBank (cont.)

- Similar records are grouped together in a file.
  - example: the code BCT above means bacteria
  - the records in this file are all for bacteria

- The accession number is a permanent and universally unique identifier for a particular sequence.

- The version number reflects changes made to a sequence.
  - updating the sequence changes the version number, but it doesn’t change the accession number

- Which of these numbers would you use as the primary key?
Why Flat Files?

- Most conventional database systems are not optimized for large strings like the ones needed for genetic sequences.
- Conventional DBMSs don’t support the types of queries that are often needed.
  - biologists use programs (often written in Python or Perl) to perform the queries
- Flat files can be viewed on any platform without specialized software.

Queries I: Text Searches

- *Entrez Nucleotide* allows you to perform a text search on the records in GenBank and other related databases.
  - can specify predicates based on one or more fields
    - examples:
      - `3000[SLEN]`  
        - (sequence length = 3000)
      - `3000:4000[SLEN]`  
        - (3000 <= sequence length <= 4000)
      - `human[orgn]`  
        - (organism type = human)
      - `3000:4000[SLEN] AND human[orgn]`

  - what aspect of SQL does this resemble?

  - if you don't specify field names, Entrez performs the equivalent of a Google search on the records
Example of Entrez Nucleotide

Queries II: Similarity Searches

- What if we want to search for sequences that "include" a particular gene or other sequence of bases?

- Text search does not work well for this problem. Why?
  - genetic sequence data is noisy.
    - errors in sequencing
    - missing data
    - mutations
    - individual variation
  - common queries involve looking for similarities, rather than exact matches
    - problem: noise may make unrelated genes appear similar!
Similarity Searches Using BLAST

- BLAST is a set of online tools for performing similarity searches.
  - nucleotide-nucleotide BLAST (blastn) is used for similarity searches involving DNA sequences
  - BLAST returns some number of hits: sequences containing subsequences whose similarity to the query is statistically significant.
  - the “goodness” of a match takes into account the number of insertions, deletions, and substitutions that would be needed to create a perfect match.

query: \[TCGTCTGTT\]

hit: AGCTTTGCTTCTGAGTCCAACGGGCAAGTC...
aligned query: TCG-TCT-GTT

- We've changed the database to "Nucleotide collection (nr/nt)" so that sequences from non-human organisms are included.
Note that both human and non-human sequences appear in the results.

Each hit is accompanied by several metrics, including:

- **max score**: measures how close the match is
  - higher max scores are better

- **E value**: measures statistical significance
  - E = the number of hits you would expect to obtain with this max score purely by chance
  - lower E scores are better
BLAST Results (cont.)

- The results screen also displays the partial match.

- Example: the query shown on the earlier slides was a 140-base sequence taken from a human sequence in the database.
  - it matches perfectly with other human ones
  - it also partially matches sequences for other organisms, including cows:

```
>Boletus 010004903.1 [SIB1] transmembrane family, member 1 (SIB1), mRNA
<table>
<thead>
<tr>
<th>E-value</th>
<th>Score</th>
<th>Expect</th>
<th>Identities</th>
<th>Similarity</th>
<th>State</th>
<th>Query</th>
<th>Identity</th>
<th>Match</th>
<th>Query</th>
<th>Expect</th>
<th>Identity</th>
<th>Match</th>
<th>Query</th>
<th>Expect</th>
<th>Identity</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5e-05</td>
<td>105</td>
<td>10.6e-09</td>
<td>111/126</td>
<td>90/126</td>
<td>A</td>
<td>CCCC</td>
<td>79</td>
<td></td>
<td></td>
<td>97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>3.8e-07</td>
<td>107/126</td>
<td>76/126</td>
<td>A</td>
<td>CCCC</td>
<td>67</td>
<td></td>
<td></td>
<td>93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Using BLAST for Cancer Research

- Scientists believe that some forms of cancer are caused by viruses.

- Such viruses may insert copies of their own DNA into the DNA of the infected person.

- Weber et al. use BLAST to look for cancer-causing viruses.
  - break an infected person's DNA into many pieces
  - run each piece through BLAST
  - use the nr/nt database as before, so that non-human sequences are included
Using BLAST for Cancer Research (cont.)

- Which of the following sequences may be from a virus?

```
GTATAAGTGAACACCTCAAGGTTGCTGGCCGCAGGACAGTGCTGCACGCCGTCGTAATCCGGCCG
ACTTTGGAGGCGAGGTTGGGTGGATCTGAGGTCAGGAGTGCAAGCCTGGCCAGGAAATACGTCC
ACTTGGGAGGCAAGTGGTGGTGCCTTATAGGAATAGGTTTTGAGGTCAGGACCCCAAT
AAATTGGTTTACCTGAGTAATAGTATTATCTGAAACACAGTTAGTGGTGCTG
```

```
TGCTGGAGTGGAAATGCGCTGTCAGCCTTAAGGCTGCTGCTGCTGCTGCTGCTGCTGCTGCTG

ACATTGGTTTCTGACAACTGTTCTCTAGCAACCTCAACACGACAGCCTGGAATC
TGACTCTGAGGAGAAGCTGTGGCGGTAGTGGGCAAGGTCAGGATGGAAG
TTGTTGAGGGCGCTTGGCGAGGCTGTGGGCTGCTACCTTGGACCAGGGTCTTGG
AGTCCTTTGGGATCTGTCACTCTGTAGCTGTTATGGGCAACCCTAAGGTAAGCTC
```

References

- Portions of these notes are based on a presentation by Griffin Weber.

- Other resources:
  - the genomics primer at www.yourgenome.org