Computer Science 105
Introduction to Databases and Data Mining

Boston University, Fall 2021

Introduction to Databases and Data Mining

Course Overview

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Welcome to CS 105!

- This course examines how collections of data are organized, stored, and processed.
- Topics include:
  - databases
  - programming
  - data mining
  - data visualization

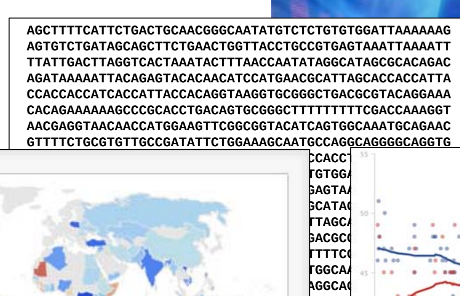
Broad Goals of the Course

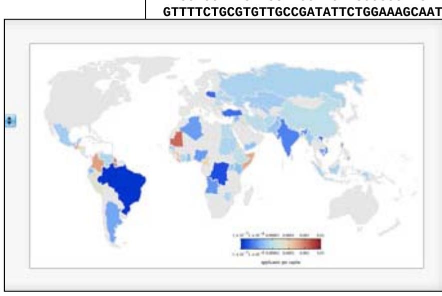
- To give you computational tools for working with data
- To give you insights into databases and data mining
  - help you to understand their increasingly important role
- To expose you to the discipline of computer science
  - to how computer scientists think and solve problems

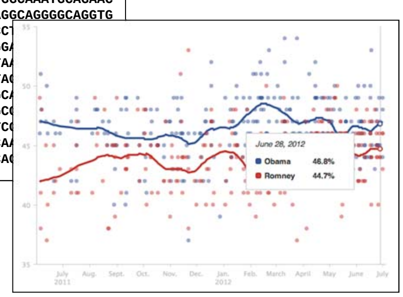
"Computer science is not so much the science of computers as it is the science of solving problems using computers."
- Eric Roberts, Stanford

Data, Data Everywhere!

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







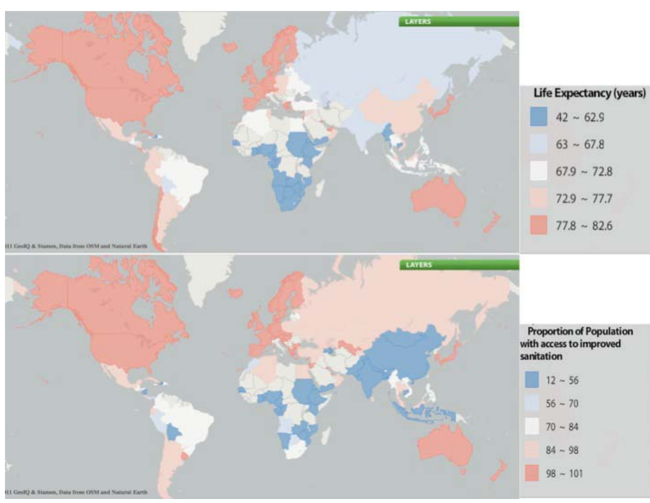
Databases

- 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 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

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

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

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 2024, 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.

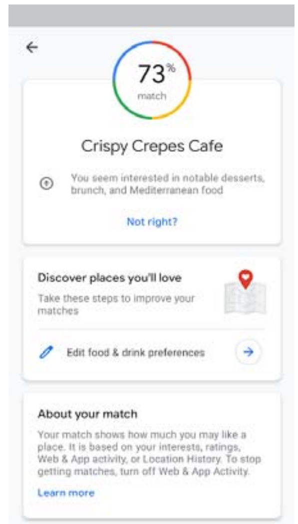
- Example: DNA sequence data

```
>gi|49175990|ref|NC_000913.2| Escherichia coli K12, complete genome
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTA
AATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAA
ACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGA
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)
  - 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



- Example: detecting credit-card fraud

Data **MINING** Everywhere!

The New York Times

February 16, 2012

How Companies Learn Your Secrets

By CHARLES DUHIGG

Andrew Pole had just started working as a statistician for Target in 2002, when two colle marketing department stopped by his desk to ask an odd question: "If we wanted to figu 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 intersection of data and human behavior most of his life. His parents were teachers in No while other kids were going to 4-H, Pole was doing algebra and writing computer progran stereotype of a math nerd is true," he told me when I spoke with him last year. "I kind of and evangelizing analytics."

TARGET

Structure of the Course

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

Course Materials

- **Required**: *The CS 105 Coursepack*
  - use it during pre-lecture and lecture – need to fill in the blanks!
  - PDF version is available on Blackboard
  - recommended: get it printed
    - one option: FedEx Office (Cummington & Comm Ave)
- **Required** in-class software: Top Hat Pro platform
  - used for pre-lecture quizzes and in-lecture exercises
  - create your account and purchase a subscription ASAP (see Lab 0 for more details)

Traditional Lecture Classes

- The instructor summarizes what you need to know.
- Readings are assigned, but may not actually be done!
- Dates back to before the printing press.

- Many technological developments since then!

Limitations of the Traditional Approach

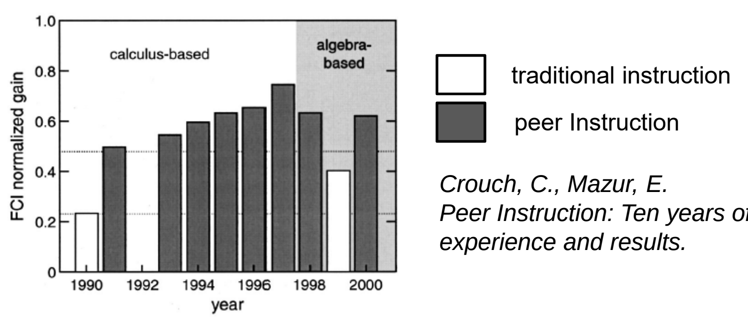
- You get little or no immediate feedback.
- Research shows that little is learned from passive listening.
  - need to actively engage with the material
- Homework provides active engagement, but in-class engagement provides added benefits.

Lectures in this Class

- Based on an approach called *peer instruction*.
  - developed by Eric Mazur at Harvard
- Basic process:
  1. Question posed (possibly after a short intro)
  2. Solo vote (no discussion yet)
  3. Small-group discussions (in teams of 3)
     - explain your thinking to each other
     - come to a consensus
  4. Group vote
     - each person in the group should enter the same answer
  5. Class-wide discussion

Benefits of Peer Instruction

- It promotes active engagement.
- You get immediate feedback about your understanding.
- I get immediate feedback about your understanding!
- It promotes increased learning.
  - explaining concepts to others benefits you!



Crouch, C., Mazur, E.
Peer Instruction: Ten years of experience and results.

Preparing for Lecture

- Short video(s) and/or readings
  - fill in the blanks as you watch the videos!
- Short online reading quiz or other exercise
  - complete **by 1 p.m.** of the day of lecture (unless noted otherwise)
  - won't typically be graded for correctness
  - your work should show that you've prepared for lecture
  - **no late submissions accepted**
- Preparing for lecture is essential!
  - gets you ready for the lecture questions and discussions
  - we won't cover everything in lecture

Course Website
http://www.cs.bu.edu/courses/cs105

- *not* the same as the Blackboard site for the course
- use the Blackboard link to access:
  - the pre-lecture readings/videos
  - the pre-lecture quizzes/exercises

  posted by 36 hours before lecture
  - the lecture notes – posted after lecture

Labs

- Attendance is required
  - begin next week
- Will help you prepare for and get started on the assignments
- Will also reinforce essential skills
- **ASAP: Complete Lab 0** (on the course website)
  - setup Top Hat account/subscription
  - setup a CS account before your first lab session
  - some other tasks to prepare you for the semester

Requirements / Grading

Preparation and participation (10%)
  - lecture preparation
  - attendance/participation – full credit if you:
    - make 85% of the votes over the entire semester
    - attend 85% of the labs
- Nine homework assignments (25%)
- Final project (10%): **done in teams of three**
- Three quizzes (25%)
- Final exam (30%)

Course Staff

- Instructor: Dave Sullivan (dgs@cs.bu.edu)
- Teaching fellow
- Office hours and contact info. will be available on the main course Web site:

  http://www.cs.bu.edu/courses/cs105
- For questions on content, homework, etc.:
  - **use Piazza**
  - send e-mail to cs105-staff@cs.bu.edu

Other Details of the Syllabus

- Collaboration / academic misconduct
- 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.

Pre-Lecture
Fundamental Facts About Data

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Measuring Data: Bits and Bytes

- All data values are stored as *binary* numbers.
  - sequences of 0s and 1s (e.g., 1001000110110100)
- A *bit* is a single 0 or 1.
- One *byte* is 8 bits.
  - example: 01101100
- Other common units:

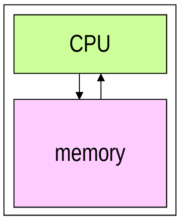
name	approximate size	exact size
kilobyte (KB)	1000 bytes	2^{10} = 1024 bytes
megabyte (MB)	1 million bytes	2^{20} bytes
gigabyte (GB)	______________	______________

Processing Data: the CPU

- At the heart of every computer is its CPU.
  - short for *central processing unit*
- Includes hardware for processing data stored in binary form.
  - example: a circuit for adding two binary numbers
- The CPU can only store a small amount of data at a time.
  - the values it is currently processing

Storing Data: Memory

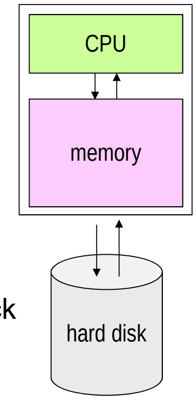
- Used to store programs and other data that are currently in use.
- Values stored in memory are read into the CPU to be processed.
- The results of operations performed by the CPU can be written back to memory.



- Advantage of memory: short *access times*
  - can read from/write to memory in ______________________
- 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
- Advantages of hard disks:
  - relatively inexpensive
  - contents are not lost when the power goes off
- Disadvantage: long access times
  - data is transferred in *blocks* (4 KB or 8 KB)
  - takes _______________ to read one block
  - 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



Database Fundamentals

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Measuring Data: Bits and Bytes

- All data values are stored as *binary* numbers.
  - sequences of 0s and 1s (e.g., 1001000110110100)
- A *bit* is a single 0 or 1.
- One *byte* is 8 bits.
  - example: 01101100
- Other common units:

name	approximate size	exact size
kilobyte (KB)	1000 bytes	2^{10} = 1024 bytes
megabyte (MB)	1 million bytes	2^{20} bytes
gigabyte (GB)	1 billion bytes	2^{30} bytes
less common, but increasingly needed:		
terabyte (TB)	1 trillion bytes	2^{40} bytes
petabyte	10^{15} bytes	2^{50} bytes

Which component of the computer contains the hardware needed to add binary numbers?

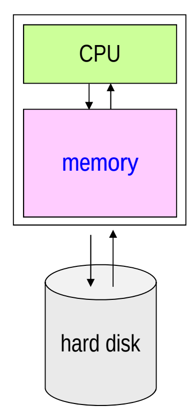
A. primary storage

B. secondary storage

C. central processing unit

D. hard disk

Which of these statements about the computer's memory is NOT true?



A. It stores the programs and other data that are currently in use.

B. It can only store a small amount of data.

C. Data stored in memory can be accessed in nanoseconds.

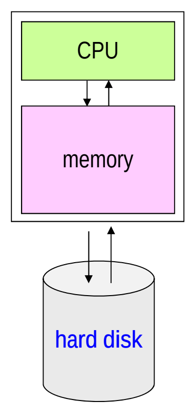
D. Data stored in memory will still be there after the computer is shut off.

E. more than one of the above

- TRS-80 Model III
  - 64 KB of memory
  - 2.03 MHz processor

- Pixel 2
  - 4 GB of memory
  - 2.35 GHz processor

Which of these statements about a hard disk is NOT true?



A. It is a type of secondary storage.

B. Its price is relatively inexpensive.

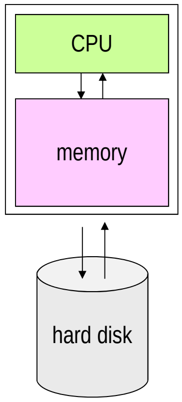
C. It takes a relatively long time to access the data stored on a disk.

D. Data is transferred to and from a disk in small blocks that are typically 4 bytes or 8 bytes in size.

E. more than one of the above

The Flow of Data

- We use a hard disk to store programs and data that we don't want to lose.
- To work with data stored on disk, we first read it from disk into memory.
- Once data is in memory, it can be read into the CPU and processed.
- Results of CPU operations are written back to memory.
- When we create new values in memory that we want to keep, we need to *eventually* write them to disk.
  - usually wait as long as possible to do this. why?



Database vs. Database Management System

- A *database* is a collection of data.
  - it is *not* a program
  - it does *not* need to be on a computer
    - example: the paper card catalogs that libraries maintained
- A *database management system* is a program that manages one or more databases.
  - abbreviation = DBMS

Key Functions of a DBMS

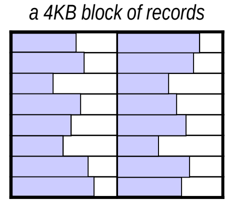
1. efficient storage
2. providing a logical view of data
3. query processing
4. transaction management

- 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 when it's shorter than that)
  - scan through the database until you find the record you're looking for
  - may require *thousands* of disk reads!

a 4KB block of records



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!

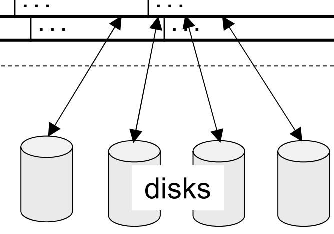
- A DBMS can also spread a database over multiple disks.
  - allows for larger collections of data
  - the disks can be accessed in parallel, which speeds things up
  - another advantage of using multiple disks?

2. Providing a Logical Representation of Data

logical representation (tables, fields, etc.)

id	*name*	*address*	*class*	*dob*
12345678	Jill Jones	Warren Towers 100	2007	3/10/85
25252525	Alan Turing	Student Village A210	2010	2/7/88
...	...	...	...	...
...	...	...	...	...

physical representation of our data



(bytes on disk blocks, index structures, etc.)

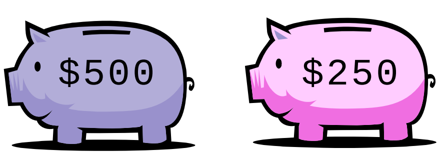
- The DBMS takes care of translating between the representations.
  - makes the user's job much easier!
- This is an example of *abstraction.*
  - hide low-level details behind a simpler representation
  - an important concept in computer science

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 performs the low-level steps needed to execute a given command.

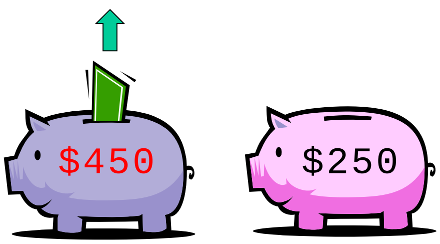
4. Transaction Management

- A *transaction* is a sequence of operations that is treated as a single logical operation.
- Example: balance transfer of $50 from blue to pink



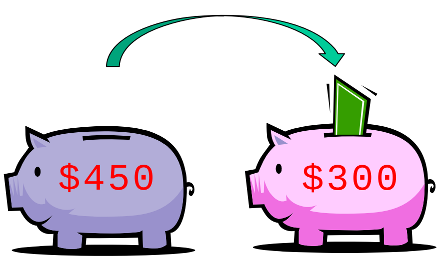
4. Transaction Management

- 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



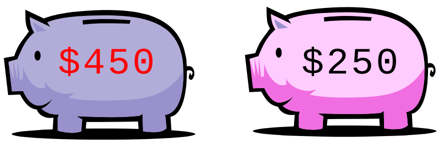
4. Transaction Management

- 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



4. Transaction Management

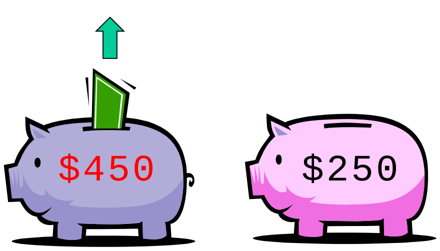
- A *transaction* is a sequence of operations that is treated as a single logical operation.
- Example: balance transfer of $50 from blue to pink
  - remove $50 from blue
  - add $50 to pink
- Without a transaction, bad things could happen!




```
remove $50 from blue
*** CRASH ***
Money is lost!
```

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!

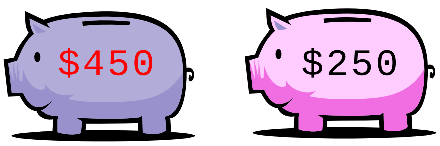


- By using a transaction for the balance transfer, we ensure that all of the steps happen, or none do.
  - ***all or nothing!***

```
remove $50 from blue
```

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!

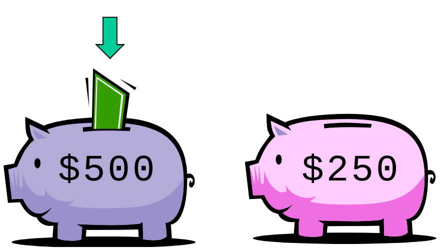


- By using a transaction for the balance transfer, we ensure that all of the steps happen, or none do.
  - ***all or nothing!***

```
remove $50 from blue
    *** CRASH ***
```

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!



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

```
remove $50 from blue
    *** CRASH ***
restore original state
```

4. Transaction Management (cont.)

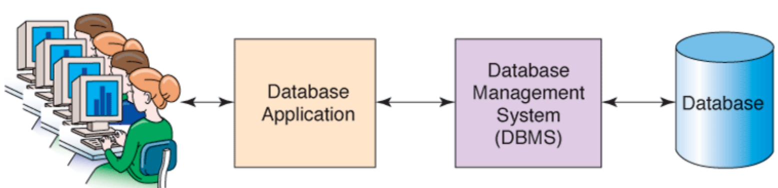
- Other examples:
  - making a flight reservation
    select flight, reserve seat, make payment
  - making an online purchase
  - making an ATM withdrawal

- Ensure that operations by different users don't overlap in problematic ways.
  - example: what's wrong with the following?

user's balance transfer *bank's check for clients below minimum balance*

```
remove 500 from blue
          read blue balance
          read pink balance
          if (blue + pink < minimum)
              charge the user a fee
add 500 to pink
```

Database Applications



(Figure 1-15 of Kroenke)

- A *database application* is a separate piece of software that interacts with a DBMS.
  - examples:
    - StudentLink
    - a web interface to a library database

- Makes it easier for users to access the database.
  - without needing to know the query language
  - the application makes queries on their behalf

Pre-Lecture
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.

The Relational Model: Basic Concepts

- A *database* consists of a collection of *tables*.
- Example of a table:

id	*name*	*address*	*email*
12345678	Jill Jones	Warren Towers 100	jjones@bu.edu
25252525	Alan Turing	Student Village A210	aturing@bu.edu
33566891	Audrey Chu	300 Main Hall	achu@bu.edu
45678900	Jose Delgado	Student Village B300	jdelgad@bu.edu
66666666	Count Dracula	The Dungeon	count@bu.edu
...	...	...	...

- 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 = ____________

  row = ____________

  column = ____________

- 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 often ignore them)
- If we name our earlier table *Student*, its schema would be:

id	*name*	*address*	*email*
12345678	Jill Jones	Warren Towers 100	jjones@bu.edu
25252525	Alan Turing	Student Village A210	aturing@bu.edu
33566891	Audrey Chu	300 Main Hall	achu@bu.edu
45678900	Jose Delgado	Student Village B300	jdelgad@bu.edu
66666666	Count Dracula	The Dungeon	count@bu.edu
...	...	...	...

Pre-Lecture
Keys, Candidate Keys, and Primary Keys

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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.

id	*name*	...	*email*
12345678	Jill Jones	...	jjones@bu.edu
25252525	Alan Turing	...	aturing@bu.edu
...	...	...	...

  - possible keys for the *Student* relation include:
    - *id*
    -
    -
    -

Candidate Key

- A *candidate key* is a *minimal* collection of attributes that is a key.
  - minimal = no unnecessary attributes are included

Candidate Key (cont.)

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

student	*course*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

Candidate Key (cont.)

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

student	*course*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

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 database 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: *Student(id, name, address, class, dob)*

Pre-Lecture
Capturing Relationships Using Foreign Keys

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Relations and Keys

- Let's say that we have the following relations:

Student(id, name, address, email)

id	name	address	email
12345678	Jill Jones	Warren Towers 100	jjones@bu.edu
25252525	Alan Turing	Student Village A210	aturing@bu.edu
...	...	...	...

Faculty(id, name, office, phone)

id	name	office	phone
11111	Ted Codd	MCS 207	617-353-1111
55555	Grace Hopper	MCS 222	617-353-5555
77777	Edgar Dijkstra	MCS 266	617-353-7777
	...	...	...

Department(name, office, phone)

name	office	phone
computer science	MCS 140	617-353-8919
english	236 Bay State Rd.	617-353-2506
mathematics	MCS 140	617-353-2560
...	...	...

Capturing Relationships

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

Capturing Relationships (cont.)

- One relationship we might want to capture is the relationship between students and their advisors.
- We can do so by expanding the *Student* relation to include an attribute called *advisor* .
  - stores the faculty ID of a student's advisor

Student

id	*name*	...	*advisor*
12345678	Jill Jones	...	11111
25252525	Alan Turing	...	
...	...	...	

Examples:
Jill Jones' advisor is Ted Codd.

Alan Turing's advisor is Edgar Dijkstra.

Faculty

id	*name*	*office*	*phone*
11111	Ted Codd	MCS 207	617-353-1111
55555	Grace Hopper	MCS 222	617-353-5555
77777	Edgar Dijkstra	MCS 266	617-353-7777
	...	...	...

Foreign Keys

Student

id	name	...	advisor
12345678	Jill Jones	...	11111
25252525	Alan Turing	...	55555
...	...	...	...

Faculty

id	name	office	phone
11111	Ted Codd	MCS 207	617-353-1111
55555	Grace Hopper	MCS 222	617-353-5555
77777	Edgar Dijkstra	MCS 266	617-353-7777
	...	...	...

- *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*
  - it takes on values from the primary key of *Department*

Student

id	name	...	major
12345678	Jill Jones	...	computer science
25252525	Alan Turing	...	mathematics
...	...	...	...

Department

name	office	phone
computer science	MCS 140	617-353-8919
english	236 Bay State Rd.	617-353-2506
mathematics	MCS 140	617-353-2560
...	...	...

More Examples of Foreign Keys (cont.)

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

Student

id	name	...	major
12345678	Jill Jones	...	computer science,english
25252525	Alan Turing	...	mathematics
...	...	...	...

Department

name	office	phone
computer science	MCS 140	617-353-8919
english	236 Bay State Rd.	617-353-2506
mathematics	MCS 140	617-353-2560
...	...	...

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*

MajorsIn

student	department
12345678	computer science
12345678	english
...	...

Student

id	name	...
12345678	Jill Jones	...
25252525	Alan Turing	...
...	...	...

Department

name	...
computer science	...
english	...
...	...

The Relational Model: Foundations; Primary and Foreign Keys

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Which of these statements about the Movie table is NOT true?

`Movie(id, name, year, rating, runtime)`

id	name	year	rating	runtime
12345	Star Wars: The Force Awakens	2015	PG-13	138
78910	Avatar	2009	PG-13	162
23232	Titanic	1997	PG-13	194
90210	Finding Dory	2016	PG	97
55555	Toy Story 3	2010	G	103
01111	Ocean's Eleven	2001	PG-13	116
...				

A. Another name for the Movie table is the Movie relation.

B. The Movie table has five tuples.

C. The primary key of Movie is the combination (name, year).

D. more than one of the above

Could We Do This?

`Movie(id, name, year, rating, runtimes)`

id	name	year	rating	runtimes
12345	Star Wars: The Force Awakens	2015	PG-13	138, 150
78910	Avatar	2009	PG-13	162, 194
23232	Titanic	1997	PG-13	194
90210	Finding Dory	2016	PG	97
55555	Toy Story 3	2010	G	103
01111	Ocean's Eleven	2001	PG-13	116
...				

- Assume that a given movie can have multiple versions.
  - e.g., the cinematic release and a special "director's cut"
- Could we capture the runtimes of all of the versions as shown above?

We Could Do This....

`Movie(id, name, year, rating, runtime1, runtime2)`

id	name	year	rating	runtime1	runtime2
12345	Star Wars: The Force Awakens	2015	PG-13	138	150
78910	Avatar	2009	PG-13	162	194
23232	Titanic	1997	PG-13	194	
90210	Finding Dory	2016	PG	97	
55555	Toy Story 3	2010	G	103	
01111	Ocean's Eleven	2001	PG-13	116	
...					

Here's a relation with info about rooms on campus...

`Room(id, building, room_num, capacity)`

id	*building*	*room_num*	*capacity*
54321	CAS	522	208
33333	CAS	224	133
24680	CAS	B12	121
11111	HAR	105	373
10101	COM	101	240
66666	CGS	129	410

Which of these are *candidate* keys of this relation?

`Room(id, building, room_num, capacity)`

id	*building*	*room_num*	*capacity*
54321	CAS	522	208
33333	CAS	224	133
24680	CAS	B12	121
11111	HAR	105	373
10101	COM	101	240
66666	CGS	129	410

candidate key:
- *can be used to uniquely identify a given row*
- *none of the attributes are unnecessary*

A. `id`
B. `(building, room_num)`
C. `(id, building)`
D. A and B, but not C
E. A, B, and C

Which of these are *keys* of this relation?

`Room(id, building, room_num, capacity)`

id	*building*	*room_num*	*capacity*
54321	CAS	522	208
33333	CAS	224	133
24680	CAS	B12	121
11111	HAR	105	373
10101	COM	101	240
66666	CGS	129	410

key:
- *can be used to uniquely identify a given row*

A. `id`

B. `(building, room_num)`

C. `(id, building)`

D. A and B, but not C

E. A, B, and C

Recall: Foreign Keys

- If students can have multiple majors, we can't just add an attribute for majors to *Student*.
  - why?
- Instead, we create a separate relation that has <u>two</u> foreign keys:
  - one with values from the primary key of *Student*
  - one with values from the primary key of *Department*

MajorsIn

student	*department*
12345678	computer science
12345678	english
...	...

- What is the primary key of *MajorsIn*?

Student

id	*name*	*...*
12345678	Jill Jones	...
25252525	Alan Turing	...
...	...	...

Department

name	*...*
computer science	...
english	...
...	...

Example of Creating a Relational Database

- Let's say that we're building a database for our new e-commerce website:

TerrierStuff.com

sportskids.com

Boston University Terriers NCAA car mat
$39.99

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?

Pre-Lecture
Constraints and Null Values

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Constraints

- In the relational model, we can specify *constraints* on the values of an attribute.
  - criteria that the values must meet
- If we attempt to add a row that would violate a constraint, the database management system (DBMS) will prevent us from doing so.

Uniqueness Constraints

Student(<u>id</u>, name, address, email)

id	*name*	*address*	*email*
12345678	Jill Jones	Warren Towers 100	jjones@bu.edu
25252525	Alan Turing	Student Village A210	aturing@bu.edu
33566891	Audrey Chu	300 Main Hall	achu@bu.edu
45678900	Jose Delgado	Student Village B300	jdelgad@bu.edu
66666666	Count Dracula	The Dungeon	count@bu.edu

- When we specify a primary key, the DBMS imposes a *uniqueness constraint* on those attribute(s).
  - each row must have unique value(s) for those attribute(s)
  - example: we <u>can't</u> add this row to Student:
    (25252525, Alex Hamilton, 45B Smith Hall, aham@bu.edu)
  - could we add this row?
    (44444444, Jill Jones, Student Village A100, jill44@bu.edu)

Uniqueness Constraints (cont.)

Movie(<u>name</u>, <u>year</u>, rating, runtime)

name	*year*	*rating*	*runtime*
Star Wars: The Force Awakens	2015	PG-13	138
Avatar	2009	PG-13	162
Titanic	1997	PG-13	194
Finding Dory	2016	PG	97
Toy Story 3	2010	G	103
Ocean's Eleven	2001	PG-13	116

- If the primary key is a combination of attributes, each row must have a unique *combination* of values for those attributes.
  - example: we <u>can</u> add these rows to Movie:
    (Ocean's Eleven, 1960, PG-13, 127)
    (American Sniper, 2015, R, 133)
  - we <u>can't</u> add this row: *why?*
    (Avatar, 2009, R, 170)

Referential Integrity Constraints

Student

id	*name*	*...*	*advisor*
12345678	Jill Jones	...	11111
25252525	Alan Turing	...	55555

Faculty

id	*name*	*office*	*phone*
11111	Ted Codd	MCS 207	617-353-1111
55555	Grace Hopper	MCS 222	617-353-5555
77777	Edgar Dijkstra	MCS 266	617-353-7777

- When we specify a foreign key, the DBMS imposes a *referential integrity constraint* on those attribute(s).
  - the foreign key attribute(s) must take on values that are already in the corresponding primary key
  - examples: can we add these rows to Student?

    (33333333, Alex Hamilton, ..., 22222)
    (33333333, Alex Hamilton, ..., 11111)

Null Values

- Recall: all values in a given column must be of the same *data type* or *domain*.
- By default, most data types include a special value called *null*.
- Null values can be used to indicate:
  - that the value of an attribute is *unknown*
  - that there is no value for that attribute in a given row
    - example:

Student

id	*name*	*...*	*major*
12345678	Jill Jones	...	computer science
25252525	Alan Turing	...	mathematics
33333333	Dan Dabbler	...	null

Null Values (cont.)

- We *can't* put a null value in a primary-key column.
- We *can* put a null value in a foreign-key column.
  - even though null is not in the corresponding primary key
  - allows us to indicate the absence of a relationship

Student

id	name	...	advisor
12345678	Jill Jones	...	11111
25252525	Alan Turing	...	77777
48484848	Alex Hamilton	...	null

Faculty

id	name	office	phone
11111	Ted Codd	MCS 207	617-353-1111
55555	Grace Hopper	MCS 222	617-353-5555
77777	Edgar Dijkstra	MCS 266	617-353-7777
	...	...	...

- We can also tell the DBMS that we don't want a given column to include any null values.

Constraints and Null Values; Designing a Database

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

One of the tables in our upcoming movie database...

`Person(id, name, dob, pob)`

id	name	dob	pob
0000007	Humphrey Bogart	1899-12-25	New York, NY, USA
0000030	Audrey Hepburn	1929-05-04	Brussels, Belgium
0000133	Geena Davis '79	1956-01-21	Wareham, MA, USA
0000151	Morgan Freeman	1937-06-01	Memphis, TN, USA
0000158	Tom Hanks	1956-07-09	Concord, CA, USA
0000194	Julianne Moore '83	1960-12-03	Fayetteville, NC, USA
...			

over 2400 people (actors and directors)!

Which of these could NOT be added to Person?

`Person(id, name, dob, pob)`

id	name	dob	pob
0000007	Humphrey Bogart	1899-12-25	New York, NY, USA
0000030	Audrey Hepburn	1929-05-04	Brussels, Belgium
0000133	Geena Davis '79	1956-01-21	Wareham, MA, USA
0000151	Morgan Freeman	1937-06-01	Memphis, TN, USA
0000158	Tom Hanks	1956-07-09	Concord, CA, USA
0000194	Julianne Moore '83	1960-12-03	Fayetteville, NC, USA

A. `(0000007, James Bond, 1920-11-11, London)`
B. `(4444444, Morgan Freeman, 1937-06-01, Memphis)`
C. `(0000030, Audrey Hepburn, 1988-10-05, Boston)`
D. A and C, but not B
E. A, B, and C

Another table in our upcoming movie database...

`Oscar(movie_id, person_id, type, year)`

movie_id	person_id	type	year
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

- movie_id takes on values from the id column in the Movie table
- person_id takes on vales from the id column in the Person table
- example: the first tuple tells us the 2016 Best Actor award went to Leonardo DiCaprio (**0000138**) for *The Revenant* (**1663202**).
- What are movie_id and person_id examples of?

Another table in our upcoming movie database...

`Oscar(movie_id, person_id, type, year)`

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

- What does NULL mean?
- Note that NULL is ***not*** a string (a piece of text)!
- NULL (or null) is a special value that means the absence of a value.

Another table in our upcoming movie database...

`Oscar(movie_id, person_id, type, year)`

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

- Would (type, year) work as the primary key?
- What about (person_id, year)?
- What about (person_id, type, year)?

Let's assume there are no NULLs...

Oscar(movie_id, person_id, type, year)

movie_id	person_id	type	year
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	1111111	BEST-PICTURE	2016
...			

Which of these could NOT be added to Oscar?

Oscar(movie_id, person_id, type, year)

movie_id	person_id	type	year
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	1111111	BEST-PICTURE	2016
...			

A. (7777777, 0000138, BEST-ACTOR, 2017)

B. (2222222, 1111111, BEST-ACTRESS, 2016)

C. (4444444, 0488953, BEST-ACTRESS, 2016)

D. A and C, but not B

E. A, B, and C

Would the DBMS allow or reject these operations?

`MajorsIn(student_id, dept_name)`

student_id	dept_name
12345678	computer science
12345678	english

Assume that these are all of the rows in each table.

`Student(id, name, ...)`

id	name	...
12345678	Jill Jones	...
25252525	Alan Turing	...

`Department(name, ...)`

name	...
computer science	...
english	...

- adding `(12345678, John Smith, …)` to `Student`
- adding `(33333333, Howdy Doody, …)` to `Student`
- adding `(12345678, physics)` to `MajorsIn`
- adding `(25252525, english)` to `MajorsIn`

Rules of Thumb for Database Design

- Give each type of entity its own relation.
- Connect related entities using foreign keys.
- Use a separate table to capture a type of relationship if a given entity can have more than one relationship of that type.
  - because you *cannot* have a multi-valued attribute

Rules of Thumb for Database Design

- Give each type of entity its own relation.
- Connect related entities using foreign keys.
- Use a separate table to capture a type of relationship if a given entity can have more than one relationship of that type.
  - because you *cannot* have a multi-valued attribute

Example Design: University Database

- Here's the full schema of a simplified university database.
  - four relations that store info. about a type of entity:

    Student(<u>id</u>, name)
    Department(<u>name</u>, office)
    Room(<u>id</u>, name, capacity)
    Course(<u>name</u>, start_time, end_time, room_id)
  - two relations that capture relationships between entities:

    MajorsIn(<u>student_id</u>, <u>dept_name</u>)
    Enrolled(student_id, course_name, credit_status)
- The *Course* relation also captures a relationship – the relationship between a course and the room in which it meets.
- We underline the primary key of each relation.
  - what would the primary key of *Enrolled* be?

Foreign Keys in the University Database

Student(<u>id</u>, name)
Department(<u>name</u>, office)
Room(<u>id</u>, name, capacity)
Course(<u>name</u>, start_time, end_time, room_id)
MajorsIn(<u>student_id</u>, <u>dept_name</u>)
Enrolled(<u>student_id</u>, <u>course_name</u>, credit_status)

- Foreign keys we've already discussed:
  - *student_id* in *MajorsIn* (takes on values from *id* in *Student*)
  - *dept_name* in *MajorsIn* (takes on values from *name* in *Department*)
- What other foreign keys make sense?
  -
  -
  -

Pre-Lecture
The SQL Query Language: Simple SELECT Commands

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Department

name	office
comp sci	MCS 140
mathematics	MCS 140
the occult	The Dungeon
english	235 Bay State Road

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

SELECT (from a single table)

- Sample query:

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

- Basic syntax:

  `SELECT` *column1*, *column2*, …
  `FROM` *table*
  `WHERE` *selection condition*;

Important notes:

- *Non-numeric column values are surrounded by single quotes.*
- *Table/column names and SQL keywords are not surrounded by quotes.*

  - 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

SELECT (from a single table) (cont.)

- Example:

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

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

`WHERE credit_status = 'grad';`

student_id	*course_name*	*credit_status*
45678900	CS 460	grad
45678900	CS 510	grad

`SELECT student_id`

student_id
45678900
45678900

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_id
FROM Enrolled;
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

SELECT student_id

student_id
12345678
25252525
45678900
33566891
45678900

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';
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

WHERE credit_status = 'grad';

student_id	course_name	credit_status
45678900	CS 460	grad
45678900	CS 510	grad

The `WHERE` Clause

`SELECT` *column1*`,` *column2*`,` …
`FROM` *table*
`WHERE` *selection condition*`;`

- The selection condition must be an expression that evaluates to either true or false.
  - example: `credit_status = 'grad'`
  - can include any column from the table(s) 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*:

operator	name
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal to
!=	not equal to

Practice

- Write a query that finds the names and capacities of all rooms that hold at least 70 people.

```
SELECT
FROM
WHERE
```

Practice

- Write a query that finds the names and capacities of all rooms that hold at least 70 people.

```
SELECT
FROM Room
WHERE
```

Practice

- Write a query that finds the names and capacities of all rooms that hold at least 70 people.

```
SELECT
FROM
WHERE
```

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
5000	CAS 314	80

name	capacity
CAS Tsai	500
CAS BigRoom	100
EDU Lecture Hall	100
CAS 314	80

The SQL Query Language:
Simple SELECT Commands

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Why Learn SQL?

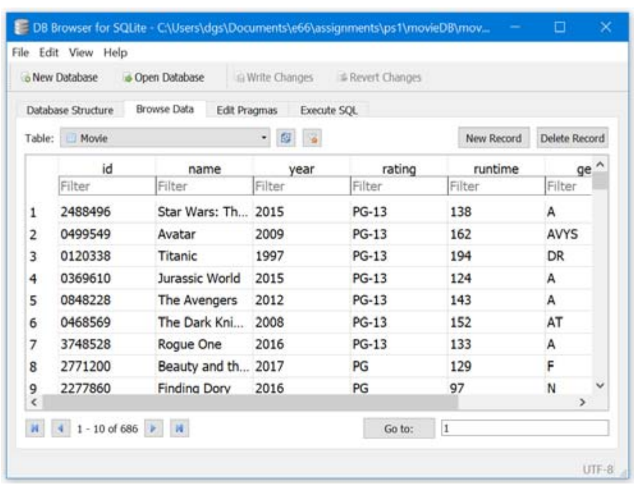
- Desktop database systems like Access provide tools for manipulating data in a database.
- However, these tools don't allow you to perform all possible types of queries.
- For more flexibility and power, we use SQL.
  - a query language
- In addition, knowledge of SQL is needed to perform queries from within a program.

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
- 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

DB Browser for SQLite

- A user-friendly program for working with a SQLite database.



- Instructions for obtaining it will be in PS 2.

How could we get all info about movies released in 2010?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

A.
```
SELECT all
FROM Movie
WHERE year = 2010;
```

B.
```
SELECT year = 2010
FROM Movie;
```

C.
```
FROM Movie
SELECT year = 2010;
```

D.
```
SELECT *
FROM Movie
WHERE year = 2010;
```

How could we get all info about movies released *before* 2010?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

How could we get the *name and runtime* of movies released *before* 2010?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

How could we get the name and runtime of of *all* movies?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

Forming More Complex Selection Conditions

- We often want to combine conditions or take the opposite of one.
- SQL provides *logical operators* for this purpose:

name	example and meaning
AND	`SELECT name, capacity` `FROM Room` `WHERE capacity >= 100 AND capacity <= 200;` *true if both parts are true, and false otherwise*
OR	`SELECT name, capacity` `FROM Room` `WHERE capacity < 50 OR capacity > 250;` *false if both parts are false, and true otherwise*
NOT	`SELECT *` `FROM COURSE` `WHERE NOT (name = 'CS 105' OR name = 'CS 111');` *true if the original 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, instead of writing:

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

we can write

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

How could we get the name and runtime of *both* Titanic *and* Toy Story 3?

Movie

id	name	year	rating	runtime
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
...				

A.
```
SELECT name, runtime
FROM Movie
WHERE name = 'Titanic' AND name = 'Toy Story 3';
```

B.
```
SELECT name, runtime
FROM Movie
WHERE name = 'Titanic' OR name = 'Toy Story 3';
```

C.
```
SELECT name, runtime
FROM Movie
WHERE name = 'Titanic' OR 'Toy Story 3';
```

D. more than one of the above

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Department

name	office
comp sci	MCS 140
mathematics	MCS 140
the occult	The Dungeon
english	235 Bay State Road

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Practice with Simple SQL Queries

- Write a query that finds all information about CAS 315.

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

- Write a query that gets the ID numbers of student(s) who are taking CS 105 for undergraduate (ugrad) credit.

Pre-Lecture
SQL: Pattern Matching, Comparisons Involving NULL

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Our simple university database...

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
CS 999	19:30:00	21:30:00	NULL

Department

name	office
comp sci	MCS 140
mathematics	MCS 140
the occult	The Dungeon
english	235 Bay State Road

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Pattern Matching

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

- Let's say we want the names and capacities of all rooms in CAS.
  - the names begin with 'CAS'
  - need to find courses with names matching this pattern

- This won't work:

```
SELECT name, capacity
FROM Room
WHERE name = 'CAS';
```

- This will:

```
SELECT name, capacity
FROM Room
WHERE name LIKE 'CAS%';
```

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

The LIKE Operator and Wildcards

- Use LIKE whenever we need to match a pattern.
- Form the pattern using one of more *wildcard characters*:
  - % stands for 0 or more arbitrary characters
  - _ stands for a single arbitrary character

More Examples of Pattern Matching

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

```
SELECT name
FROM Student
WHERE name LIKE '%u%';
```

name
Alan Turing
Audrey Chu
Count Dracula

```
SELECT name
FROM Student
WHERE name LIKE '__u%';
```

2 underscores

```
SELECT name
FROM Student
WHERE name LIKE '%u';
```

Comparisons Involving NULL

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
CS 999	19:30:00	21:30:00	NULL

- a room_id of NULL indicates the course is only offered online

- How could we find all of the online-only courses?

- This query produces no results!

```
SELECT name
FROM Course
WHERE room_id = NULL;
```

Comparisons Involving NULL

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

```
room_id = NULL
room_id != NULL
NULL = NULL
```

- SQL provides special operators:
  - IS NULL
  - IS NOT NULL
- This query will find the online-only courses:

```
SELECT name
FROM Course
WHERE room_id IS NULL;
```

Pre-Lecture
SQL: Removing Duplicates; Aggregate Functions

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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

```
SELECT student_id
FROM Enrolled;
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

student_id
12345678
25252525
45678900
33566891
45678900

Removing Duplicates (cont.)

- To eliminate duplicates, add the keyword `DISTINCT`:

```
SELECT DISTINCT student_id
FROM Enrolled;
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

student_id
12345678
25252525
45678900
33566891

- More generally:

  `SELECT DISTINCT` *column1*`,` *column2*`,` …

Aggregate Functions

- The SELECT clause can include an *aggregate function.*
  - performs a computation on a set of values
- Example: find the average capacity of rooms in CAS:

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

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

AVG

AVG(capacity)
154.0

Aggregate Functions (cont.)

- Other aggregate functions include:
  - `SUM`, `MAX`, `MIN`, and `COUNT`

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

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

SUM

Aggregate Functions (cont.)

- Other aggregate functions include:
  - `SUM`, `MAX`, `MIN`, and `COUNT`

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

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

MAX

Aggregate Functions (cont.)

- Other aggregate functions include:
  - `SUM`, `MAX`, `MIN`, and `COUNT`

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

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

MIN

Aggregate Functions (cont.)

- Other aggregate functions include:
  - `SUM`, `MAX`, `MIN`, and `COUNT`

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

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

COUNT

Aggregates and DISTINCT

- example: find the number of students enrolled for courses:

```
SELECT COUNT(student_id)
FROM Enrolled;
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

Aggregates and DISTINCT

- example: find the number of students enrolled for courses:

```
SELECT COUNT(student_id)
FROM Enrolled;
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

COUNT(student)
5

Aggregates and DISTINCT

- example: find the number of students enrolled for courses:

```
SELECT COUNT(student_id)
FROM Enrolled;
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

COUNT(student)
5

Aggregates and DISTINCT

- example: find the number of students enrolled for courses:

```
SELECT COUNT(DISTINCT student_id)
FROM Enrolled;
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

COUNT(*) vs. COUNT(*attribute*)

- `SELECT COUNT(*)` counts the number of tuples in a result.
  - example: find the total number of courses

```
SELECT COUNT(*)
FROM Course;
```

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
CS 999	19:30:00	21:30:00	NULL

COUNT(*)
5

COUNT(*) vs. COUNT(*attribute*)

- `SELECT COUNT(*)` counts the number of tuples in a result.
  - example: find the total number of courses

```
SELECT COUNT(*)
FROM Course;
```

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
CS 999	19:30:00	21:30:00	NULL

COUNT(*)
5

- `SELECT COUNT(`*attribute*`)` counts the number of non-NULL values of that attribute in a result.
  - example: find the number of courses that meet in a room

```
SELECT COUNT(room_id)
FROM Course;
```

COUNT(room_id)
4

SQL: Other Aspects of Simple SELECT Commands

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

How could we use pattern matching to get info. about movies rated PG or PG-13?

Movie

id	name	year	rating	runtime
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	12
0120338	Titanic	1997	PG-13	19
0435761	Toy Story 3	2010	G	10
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	110
...				

Assume the ratings shown here are the only ratings in the table.

A.
```
SELECT *
FROM Movie
WHERE rating LIKE 'PG%';
```

B.
```
SELECT *
FROM Movie
WHERE rating LIKE 'PG_';
```

C.
```
SELECT *
FROM Movie
WHERE rating LIKE '_G%';
```

D. two of the queries at left would work

E. all three of the queries at left would work

How could we use pattern matching to get info. about movies rated PG or PG-13?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

Assume the ratings shown here are the only ratings in the table.

A.
```
SELECT *
FROM Movie
WHERE rating LIKE 'PG%';
```
← starts with PG, followed by 0 or more arbitrary characters

B.
```
SELECT *
FROM Movie
WHERE rating LIKE 'PG_';
```
← ?

C.
```
SELECT *
FROM Movie
WHERE rating LIKE '_G%';
```
← ?

Would these patterns work for finding PG and PG-13?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0120338	Titanic	1997	PG-13	194
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0240772	Ocean's Eleven	2001	PG-13	116
...				

Assume the ratings shown here are the only ratings in the table.

```
SELECT *
FROM Movie
WHERE rating LIKE '%G%';
```

```
SELECT *
FROM Movie
WHERE rating LIKE 'PG';
```

```
SELECT *
FROM Movie
WHERE rating  =   'PG-%';
```

Pattern Matching (cont.)

- 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 do case-sensitive pattern matching
  - the = operator *is* case-sensitive

How could we find the names of all courses without a room?

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

A.
```
SELECT name
FROM Course
WHERE room_id = 'NULL';
```

B.
```
SELECT name
FROM Course
WHERE room_id = NULL;
```

C.
```
SELECT name
FROM Course
WHERE room_id IS NULL;
```

D. two or more of the queries at left would work

E. none of the queries at left would work

How could we find the names of all courses without a room?

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

A.
```
SELECT name
FROM Course
WHERE room_id = 'NULL';
```

B.
```
SELECT name
FROM Course
WHERE room_id = NULL;
```

C.
```
SELECT name
FROM Course
WHERE room_id IS NULL;
```

How could we determine how many people have won Best Actor?

Oscar

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

A.
```
SELECT COUNT(person_id)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

B.
```
SELECT TOTAL(person_id)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

C.
```
SELECT COUNT(*)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

D. two or more of the queries at left would work

E. none of the queries at left would work

How could we determine how many people have won Best Actor?

Oscar

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

A.
```
SELECT COUNT(person_id)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

B.
```
SELECT TOTAL(person_id)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

C.
```
SELECT COUNT(*)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

Would this work?

Oscar

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

```
SELECT COUNT(DISTINCT person_id)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

What about this?

Oscar

movie_id	*person_id*	*type*	*year*
1663202	0000138	BEST-ACTOR	2016
3170832	0488953	BEST-ACTRESS	2016
3682448	0753314	BEST-SUPPORTING-ACTOR	2016
0810819	2539953	BEST-SUPPORTING-ACTRESS	2016
1663202	0327944	BEST-DIRECTOR	2016
1895587	NULL	BEST-PICTURE	2016
...			

```
SELECT COUNT(DISTINCT *)
FROM Oscar
WHERE type = 'BEST-ACTOR';
```

Practice Writing Queries

1. How many CS courses are there?

2. How many rooms can hold at least 100 people?

3. What is the average capacity of the rooms from problem 2?

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Department

name	office
comp sci	MCS 140
mathematics	MCS 140
the occult	The Dungeon
english	235 Bay State Road

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Pre-Lecture
Subqueries in SQL

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Aggregate Functions

- What is the largest capacity of any room in the CAS building?

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

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

MAX

MAX(capacity)
500

A Restriction on Aggregate Functions

- What if we also wanted the *name* of the max-capacity room?

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

This does *not* work in standard SQL!

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

WHERE

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
4000	CAS 315	40
5000	CAS 314	80
6000	CAS 226	50

SELECT name

name
CAS Tsai
CAS BigRoom
CAS 315
CAS 314
CAS 226

MAX

MAX(capacity)
500

error!

A Restriction on Aggregate Functions (cont.)

- What if we also wanted the *name* of the max-capacity room?

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

This does *not* work in standard SQL!

- In general, a SELECT clause *cannot* combine:
  - an aggregate function
  - a column name that is on its own
    (and is not being operated on by an aggregate function)
- We'll see an important exception to this soon.

Subqueries

- A *subquery* allows us to use the result of one query in the evaluation of another query.

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

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

the subquery

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

name	capacity
CAS Tsai	500

Note Carefully!

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

the subquery

- if we remove the condition from the subquery, might not get the largest capacity in CAS
- if we remove the condition from the outer query, might also get …

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 <u>not</u> enrolled in CS 105

```
SELECT name
FROM Student
WHERE id NOT IN (SELECT student_id
                 FROM Enrolled
                 WHERE course_name = 'CS 105');
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

subquery

student_id
12345678
33566891

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Pre-Lecture
Queries Involving Subgroups
(GROUP BY and HAVING)

Computer Science 105
Boston University

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:

```
SELECT COUNT(*)
FROM Enrolled;
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

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:

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name;
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

⇨

course_name	COUNT(*)
CS 105	3
CS 111	2
CS 460	1

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:

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name;
```


- When you group by an attribute, you <u>can</u> include it in the SELECT clause with an aggregate function.

Evaluating a query with GROUP BY

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name;
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

student_id	course_name	credit_status
12345678	CS 105	ugrad
33566891	CS 105	non-credit
25252525	CS 105	grad
45678900	CS 111	ugrad
66666666	CS 111	ugrad
45678900	CS 460	grad

course_name	COUNT(*)
CS 105	3
CS 111	2
CS 460	1

GROUP BY + WHERE

```
SELECT course_name, COUNT(*)
FROM Enrolled
WHERE credit_status = 'ugrad'
GROUP BY course_name;
```

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

WHERE

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
66666666	CS 111	ugrad

GROUP BY

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
66666666	CS 111	ugrad

- The WHERE clause is applied *before* the GROUP BY clause.

Applying a Condition to Subgroups

- What if I only want courses with more than one student?

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

course_name	COUNT(*)
CS 105	3
CS 111	2
CS 460	1

HAVING

course_name	COUNT(*)
CS 105	3
CS 111	2

- This won't work:

```
SELECT course, COUNT(*)
FROM Enrolled
WHERE COUNT(*) > 1
GROUP BY course;
```

- This will:

```
SELECT course, COUNT(*)
FROM Enrolled
GROUP BY course
HAVING COUNT(*) > 1;
```

- WHERE is applied *before* GROUP BY.
- HAVING is applied *after* GROUP BY.
  - used for all conditions involving aggregates

SQL: Subqueries; GROUP BY and HAVING

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

How could we find the shortest PG-13 movie in the database?

Movie

id	name	year	rating	runtime
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0118998	Dr. Dolittle	1998	PG-13	85
...				

A.
```
SELECT name, MIN(runtime)
FROM Movie
WHERE rating = 'PG-13';
```

B.
```
SELECT name, runtime
FROM Movie
WHERE runtime = (SELECT MIN(runtime) FROM Movie
                 WHERE rating = 'PG-13');
```

C.
```
SELECT name, runtime
FROM Movie
WHERE rating = 'PG-13'
  AND runtime = (SELECT MIN(runtime) FROM Movie
                 WHERE rating = 'PG-13');
```

D. two of these would work

E. all three would work

A Restriction on Aggregate Functions

```
SELECT name, MIN(runtime)
FROM Movie
WHERE rating = 'PG-13';
```

This does *not* work in standard SQL!

- In general, a SELECT clause *cannot* combine:
  - an aggregate function
  - a column name that is on its own
    (and is not being operated on by an aggregate function)
- We'll see an important exception to this soon.
- ***Warning:*** SQLite lets you violate this rule, but...
  - doing so is *not* standard SQL
  - you should ***not*** do this in your work for this class!

How could we find the shortest PG-13 movie in the database?

Movie

id	*name*	*year*	*rating*	*runtime*
2488496	Star Wars: The Force Awakens	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95
0118998	Dr. Dolittle	1998	PG-13	85
...				

A.
```
SELECT name, MIN(runtime)
FROM Movie
WHERE rating = 'PG-13';
```

can't combine an aggregate with a "plain" column unless you are grouping by the column

B.
```
SELECT name, runtime
FROM Movie
WHERE runtime = (SELECT MIN(runtime) FROM Movie
                 WHERE rating = 'PG-13');
```

← why doesn't this work?

C.
```
SELECT name, runtime
FROM Movie
WHERE rating = 'PG-13'
  AND runtime = (SELECT MIN(runtime) FROM Movie
                 WHERE rating = 'PG-13');
```

How many names would this query produce?

```
SELECT name
FROM Student
WHERE id NOT IN (SELECT student_id
                 FROM Enrolled
                 WHERE credit_status = 'ugrad');
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

What was the query looking for?

What if we just wanted the IDs of those students?

```
SELECT id
FROM Student
WHERE id NOT IN (SELECT student_id
                 FROM Enrolled
                 WHERE credit_status = 'ugrad');
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

subquery

student_id
12345678
25252525
45678900

id
33566891
66666666

Is this the same thing?

```
SELECT student_id
FROM Enrolled
WHERE student_id NOT IN (SELECT student_id
                         FROM Enrolled
                         WHERE credit_status = 'ugrad');
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

omit the Student table!

What about this?

```
SELECT student_id
FROM Enrolled
WHERE credit_status != 'ugrad';
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

What about this?

```
SELECT student_id
FROM Enrolled
WHERE credit_status != 'ugrad';
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

What about this?

```
SELECT student_id
FROM Enrolled
WHERE credit_status != 'ugrad';
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

student_id
45678900
33566891

45678900 is included even though he is enrolled in a course for undergrad credit!

Need to use a subquery and NOT IN for problems like this one!

How many rows would this query produce?

```
SELECT dept_name, COUNT(*)
FROM MajorsIn
GROUP BY dept_name;
```

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult
25252525	mathematics

How could we limit this to departments with only 1 student?

```
SELECT dept_name, COUNT(*)
FROM MajorsIn
GROUP BY dept_name;
```

A.
```
SELECT dept_name, COUNT(*)
FROM MajorsIn
WHERE COUNT(*) = 1
GROUP BY dept_name;
```

B.
```
SELECT dept_name, COUNT(*)
FROM MajorsIn
GROUP BY dept_name
WHERE COUNT(*) = 1;
```

C.
```
SELECT dept_name, COUNT(*)
FROM MajorsIn
HAVING COUNT(*) = 1
GROUP BY dept_name;
```

D.
```
SELECT dept_name, COUNT(*)
FROM MajorsIn
GROUP BY dept_name
HAVING COUNT(*) = 1;
```

E. more than one of these works

GROUP BY + WHERE

```
SELECT course_name, COUNT(*)
FROM Enrolled
WHERE credit_status = 'ugrad'
GROUP BY course_name;
```

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
45678900	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
66666666	CS 111	ugrad
25252525	CS 105	grad

WHERE

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
45678900	CS 111	ugrad
66666666	CS 111	ugrad

GROUP BY

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
45678900	CS 111	ugrad
66666666	CS 111	ugrad

course_name	*COUNT(*)*
CS 105	1
CS 111	2

- The WHERE clause is applied *before* the GROUP BY clause.

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;
```

name	*capacity*
CAS Tsai	500
CAS BigRoom	100
EDU Lecture Hall	100
…	…

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. `FROM`
  2. `WHERE`
  3. `GROUP BY`
  4. `HAVING`
  5. `SELECT`
  6. `ORDER BY`

Pre-Lecture
SQL: Data Types;
Creating Tables and Inserting Rows

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Data Types

- Recall: The values in a given column must be of the same type (i.e., must come from the same domain).
- Numeric types include:
  - `INTEGER`
  - `REAL`: a real number (i.e., one with a decimal)
- Non-numeric types include:
  - `DATE` (e.g., `'2017-02-23'`)
  - `TIME` (e.g., `'15:30:30'`)
  - two types for *strings* (i.e., arbitrary sequences of characters)
    - `CHAR` for fixed-length strings
    - `VARCHAR` for variable-length strings

CHAR vs. VARCHAR

- `CHAR(`*n*`)` is for *fixed-length* strings of exactly n characters.
- `VARCHAR(`*n*`)` is for *variable-length* strings of up to n characters.
  - used for values that can have a wide range of possible lengths
- Example: types for a *Person* table:
  - `VARCHAR(64)` for the person's name
  - `VARCHAR(128)` for the street address
  - `VARCHAR(32)` for the city
  - `CHAR(2)` for the state abbreviation (`'MA'`, `'NY'`, etc.)
  - `CHAR(5)` for the zip code
  - `CHAR(8)` for the id – since every id has the same # of digits
    - example: `'00123456'`
    - a numeric type would *not* keep the leading 0s

CHAR vs. VARCHAR (cont.)

- With both `CHAR(`*n*`)` and `VARCHAR(`*n*`)`, if the user attempts to specify value with more than *n* characters, it is truncated.
  - examples:

type	user-specified value	value stored
`CHAR(5)`	`'123456'`	`'12345'`
`VARCHAR(10)`	`'computer science'`	

- If the user attempts to specify a value of less than *n* characters:
  - if the type is `CHAR(`*n*`)`, the system pads with spaces
  - if the type is `VARCHAR(`*n*`)`, the system does *not* pad
  - examples:

type	user-specified value	value stored
`CHAR(5)`	`'123'`	`'123  '`
`VARCHAR(10)`	`'math'`	

Creating a New Table

- Basic syntax:

```
CREATE TABLE table_name(
    column1_name  column1_type,
    column2_name  column2_type,
    ...
);
```

After this command, the table is initially empty!

- Examples:

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

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

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
...		

Specifying Primary Keys

- Specify a single-column primary key after the column's type:

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

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

MajorsIn

student_id	dept_name
12345678	computer science
12345678	english
...	

```
CREATE TABLE MajorsIn(
   student_id CHAR(8), dept_name VARCHAR(30),
   PRIMARY KEY (student_id, dept_name)
);
```

Specifying Foreign Keys

- Need to specify both:
  - the foreign key itself
  - the corresponding primary key in the form `Table(column)`

MajorsIn

student_id	*dept_name*
12345678	computer science
12345678	english
...	

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
...	

Department

name	*...*
computer science	...
english	...
...	

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

Adding a Single Row to an Existing Table

- Syntax:
  `INSERT INTO` *table* `VALUES (`*val1*`,` *val2*`, ...);`
- Example:
  `INSERT INTO Room VALUES ('1234', 'MCS 148', 45)`

  `id` is `CHAR(4)`, so need quotes!

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	CAS 315	40
1234	MCS 148	45

- Notes:
  - need to specify the values in the appropriate order (based on the order of the columns in `CREATE TABLE`)
  - non-numeric values are surrounded by single quotes
  - the DBMS won't allow you to insert a row if it violates a uniqueness or referential-integrity constraint

SQL: Data Types;
Creating Tables and Inserting Rows

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

SQL Data Types

- Numeric types include:
  - `INTEGER`
  - `REAL`: a real number (i.e., one that may have a fractional part)
- Non-numeric types include:
  - `DATE` (e.g., `'2017-02-23'`)
  - `TIME` (e.g., `'15:30:30'`)
  - two types for *strings* (i.e., arbitrary sequences of characters)
    - `CHAR`
    - `VARCHAR`

Given the `CREATE TABLE` command shown below, what tuple would be added by the `INSERT` command?

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

INSERT INTO Student
  VALUES ('4567', 'Robert Brown');
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

A. `('4567    ', 'Robert Brown                  ')`

B. `('4567    ', 'Robert Brown')`

C. `('4567', 'Robert Brown                  ')`

D. `('4567', 'Robert Brown')`

What if we swapped the two values in the `INSERT`?

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

INSERT INTO Student
  VALUES ('Robert Brown', '4567');
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

(____________________________, ____________________________)
would be stored.

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.

Creating the Enrolled table...

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

```
CREATE TABLE Enrolled(
  student_id CHAR(8), course_name VARCHAR(10),
  credit_status VARCHAR(10));
```

How can I specify that `student_id` is a foreign key?

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

```
CREATE TABLE Enrolled(
  student_id CHAR(8), course_name VARCHAR(10),
  credit_status VARCHAR(10),
  PRIMARY KEY (student_id, course_name),
  _____________________________________________);
```

A. `FOREIGN KEY (student_id) REFERENCES Student(id)`

B. `FOREIGN KEY (student_id) TO id IN Student`

C. `student_id FOREIGN KEY FOR Student(id)`

D. `student_id FOREIGN KEY TO id IN Student`

What about the other foreign key in Enrolled?

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

```
CREATE TABLE Enrolled(
  student_id CHAR(8), course_name VARCHAR(10),
  credit_status VARCHAR(10),
  PRIMARY KEY (student_id, course_name),
  FOREIGN KEY (student_id) REFERENCES Student(id),

  _________________________________________________);
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
CS 999	19:30:00	21:30:00	NULL

Does the order of these insertions matter?

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

1 `INSERT INTO Enrolled VALUES('4567', 'CS 105', 'grad');`

2 `INSERT INTO Student VALUES ('4567', 'Robert Brown');`

A. 1 must come before 2

B. 2 must come before 1

C. the order of the two INSERT commands doesn't matter

Writing Single-Table Queries: Rules of Thumb

- Start with the FROM clause. Which table do you need?
- 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?
  - is DISTINCT needed?

Practice Writing Queries

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

1) Find the start times of CS 105 and CS 111.

2) Find the course(s) that end latest in the day and what its/their end time is. (Use a subquery!)

3) Find the ids of all rooms that have two or more courses in them. The result should be tuples of the form (room id, # of courses).

Pre-Lecture
SQL: Cartesian Product; Joins

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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 x 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 = { apple, pear, orange }
  B = { cat, dog }

  A x B = { (apple, cat), (apple, dog), (pear, cat), (pear, dog), (orange, cat), (orange, dog) }
- Example:
  C = { 5, 10 }
  D = { 2, 4 }

  C x D = ?

Mathematical Foundation: Cartesian Product (cont.)

- We can also take the Cartesian product of three of more sets.
- A x B x 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 x D x 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 x A_2 x … x A_n is the set of all possible ordered tuples (a_{1i}, a_{2j}, …, a_{nk}), where $a_{de} \in A_d$.

Cartesian Product of Relations

- The Cartesian product of two or more relations forms all possible combinations of *rows* from the relations.
- The result is itself a relation.
  - its rows contain all of the columns from the combined relations
- Example:

Enrolled

student_id	***course_name***	***credit_status***
12345678	CS 105	ugrad
25252525	CS 111	ugrad
...		

MajorsIn

student_id	***dept_name***
12345678	comp sci
45678900	mathematics
...	

Enrolled x MajorsIn

Enrolled. student_id	***course_name***	***credit_status***	***MajorsIn. student_id***	***dept_name***
12345678	CS 105	ugrad	12345678	comp sci
12345678	CS 105	ugrad	45678900	mathematics
...				

The Cartesian Product of Two Relations (cont.)

- Example:

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Enrolled x MajorsIn

Enrolled. student_id	course_name	credit_status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
12345678	CS 105	ugrad	45678900	mathematics
12345678	CS 105	ugrad	25252525	comp sci
12345678	CS 105	ugrad	45678900	english
12345678	CS 105	ugrad	66666666	the occult

The Cartesian Product of Two Relations (cont.)

- Example:

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Enrolled x MajorsIn

Enrolled. student_id	course_name	credit_status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
12345678	CS 105	ugrad	45678900	mathematics
12345678	CS 105	ugrad	25252525	comp sci
12345678	CS 105	ugrad	45678900	english
12345678	CS 105	ugrad	66666666	the occult
25252525	CS 111	ugrad	12345678	comp sci
25252525	CS 111	ugrad	45678900	mathematics
...	...	...	...	...

Joining Multiple Tables

```
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

Joining Multiple Tables (cont.)

- Example: find Alan Turing's major.

Student

id	***name***
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	***dept_name***
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Here's a query that works:

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing'
  AND id = student_id;
```

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

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x MajorsIn

id	name	student_id	dept_name
12345678	Jill Jones	12345678	comp sci
12345678	Jill Jones	45678900	mathematics
12345678	Jill Jones	25252525	comp sci
12345678	Jill Jones	45678900	english
12345678	Jill Jones	66666666	the occult
25252525	Alan Turing	12345678	comp sci
25252525	Alan Turing	45678900	mathematics
25252525	Alan Turing	25252525	comp sci
25252525	Alan Turing	45678900	english
...	...	...	...

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x MajorsIn WHERE id = student_id

id	name	student_id	dept_name
12345678	Jill Jones	12345678	comp sci
25252525	Alan Turing	25252525	comp sci
45678900	Jose Delgado	45678900	mathematics
45678900	Jose Delgado	45678900	english
66666666	Count Dracula	66666666	the occult

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

After selecting only tuples that satisfy the WHERE clause:

id	*name*	*student_id*	*dept_name*
25252525	Alan Turing	25252525	comp sci

After extracting the attribute specified in the SELECT clause:

dept_name
comp sci

Pre-Lecture
SQL: Joins Revisited

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Another Example of Joining Tables

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Example: find the names of all students enrolled in CS 105 who are majoring in comp sci.

```
SELECT
FROM Student, Enrolled, MajorsIn
WHERE
```

3 tables, so we need

_____ join conditions!

Dealing with Ambiguous Column Names

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Example: find the names of all students enrolled in CS 105 who are majoring in comp sci.

```
SELECT name
FROM Student, Enrolled, MajorsIn
WHERE id = Enrolled.student_id
  AND Enrolled.student_id = MajorsIn.student_id
  AND course_name = 'CS 105'
  AND dept_name = 'comp sci';
```

Dealing with Ambiguous Column Names

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Example: find the names of all students enrolled in CS 105 who are majoring in comp sci.

```
SELECT Student.name
FROM Student, Enrolled, MajorsIn
WHERE Student.id = Enrolled.student_id
  AND Enrolled.student_id = MajorsIn.student_id
  AND Enrolled.course_name = 'CS 105'
  AND MajorsIn.dept_name = 'comp sci';
```

Aliases for Table Names

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Example: find the names of all students enrolled in CS 105 who are majoring in comp sci.

```
SELECT S.name
FROM Student AS S, Enrolled AS E, MajorsIn AS M
WHERE S.id = E.student_id
  AND E.student_id = M.student_id
  AND E.course_name = 'CS 105'
  AND M.dept_name = 'comp sci';
```

```
SELECT S.name
FROM Student S, Enrolled E, MajorsIn M
WHERE S.id = E.student_id
  AND E.student_id = M.student_id
  AND E.course_name = 'CS 105'
  AND M.dept_name = 'comp sci';
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x Enrolled x MajorsIn

id	name	E.student_id	course_name	credit_status	M.student_id	dept_name
12345678	Jill Jones	12345678	CS 105	ugrad	12345678	comp sci
12345678	Jill Jones	12345678	CS 105	ugrad	45678900	mathematics
12345678	Jill Jones	12345678	CS 105	ugrad	25252525	comp sci
12345678	Jill Jones	12345678	CS 105	ugrad	45678900	english
12345678	Jill Jones	12345678	CS 105	ugrad	66666666	the occult

```
SELECT S.name
FROM Student S, Enrolled E, MajorsIn M
WHERE S.id = E.student_id
  AND E.student_id = M.student_id
  AND E.course_name = 'CS 105'
  AND M.dept_name = 'comp sci';
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x Enrolled x MajorsIn **125 rows in all!**

id	name	E.student_id	course_name	credit_status	M.student_id	dept_name
12345678	Jill Jones	12345678	CS 105	ugrad	12345678	comp sci
12345678	Jill Jones	12345678	CS 105	ugrad	45678900	mathematics
12345678	Jill Jones	12345678	CS 105	ugrad	25252525	comp sci
12345678	Jill Jones	12345678	CS 105	ugrad	45678900	english
12345678	Jill Jones	12345678	CS 105	ugrad	66666666	the occult
12345678	Jill Jones	25252525	CS 111	ugrad	12345678	comp sci
12345678	Jill Jones	25252525	CS 111	ugrad	45678900	mathematics
12345678	Jill Jones	25252525	CS 111	ugrad	25252525	comp sci
12345678	Jill Jones	25252525	CS 111	ugrad	45678900	english
12345678	Jill Jones	25252525	CS 111	ugrad	66666666	the occult
...						

```
SELECT S.name
FROM Student S, Enrolled E, MajorsIn M
WHERE S.id = E.student_id
  AND E.student_id = M.student_id
  AND E.course_name = 'CS 105'
  AND M.dept_name = 'comp sci';
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x Enrolled x MajorsIn, followed by the join conditions...

id	name	E.student_id	course_name	credit_status	M.student_id	dept_name
12345678	Jill Jones	12345678	CS 105	ugrad	12345678	comp sci
25252525	Alan Turing	25252525	CS 111	ugrad	25252525	comp sci
45678900	Jose Delgado	45678900	CS 460	grad	45678900	mathematics
45678900	Jose Delgado	45678900	CS 460	grad	45678900	english
45678900	Jose Delgado	45678900	CS 510	grad	45678900	mathematics
45678900	Jose Delgado	45678900	CS 510	grad	45678900	english

```
SELECT S.name
FROM Student S, Enrolled E, MajorsIn M
WHERE S.id = E.student_id
  AND E.student_id = M.student_id
  AND E.course_name = 'CS 105'
  AND M.dept_name = 'comp sci';
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x Enrolled x MajorsIn, followed by the join conditions and the rest of the WHERE clause

id	*name*	*E.student_id*	*course_name*	*credit_status*	*M.student_id*	*dept_name*
12345678	Jill Jones	12345678	CS 105	ugrad	12345678	comp sci

after SELECT

name
Jill Jones

SQL: Cartesian Product; Joins

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

How many rows would be in the result?

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

```
SELECT name
FROM Movie, Oscar;
```

If we want the movies associated with one or more Oscar, what condition do we need?

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

```
SELECT name
FROM Movie, Oscar
WHERE _________________________;
```

Which tables do I need?

- Find the names of all rooms that CS majors have courses in.

```
SELECT
FROM ???
WHERE
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

How many join conditions do I need?

- Find the names of all rooms that CS majors have courses in.

```
SELECT
FROM Course, Room, Enrolled, MajorsIn
WHERE ???
```

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
...			

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
...		

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
...		

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
...	

Which of these is a *correctly formed* join condition for this problem?

- Find the names of all rooms that CS majors have courses in.

```
SELECT
FROM Course, Room, Enrolled, MajorsIn
WHERE ???
```

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
...			

Room

id	*name*	*capacity*
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
...		

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
...		

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
...	

A. `room_id = id`

B. `course_name = name`

C. `student_id = student_id`

D. two or more are correct

Complete the query…

- Find the names of all rooms that CS majors have courses in.

```
SELECT
FROM Course, Room, Enrolled, MajorsIn
WHERE
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
...			

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
...		

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
...		

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
...	

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
- 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?
  - is DISTINCT needed?

Practice Writing Queries

Student(id, name)   Department(name, office)   Room(id, name, capacity)
Course(name, start_time, end_time, room_id)   MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

1) Find the names of all courses taken by comp sci majors.

2) Find the number of students majoring in each department.
(The result should be tuples of the form (dept name, # students).)

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

3) Find the names and ids of all students who have a course in GCB 204.

4) Find the names of all rooms in which one or more CS courses meet.

From earlier in the lecture:
How many rows would be in this result?

```
SELECT name
FROM Movie, Oscar
WHERE id = movie_id;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

Movie x Oscar

id	name	Movie. year	rating	runtime	movie_id	person_id	type	Oscar. year
2488496	Star Wars...	2015	PG-13	138	2488496	1111111	BEST-ACTOR	2016
2488496	Star Wars...	2015	PG-13	138	1228705	2222222	BEST-ACTRESS	2011
2488496	Star Wars...	2015	PG-13	138	2488496	NULL	BEST-PICTURE	2016
1228705	Iron Man 2	2010	PG-13	124	2488496	1111111	BEST-ACTOR	2016
1228705	Iron Man 2	2010	PG-13	124	1228705	2222222	BEST-ACTRESS	2011
1228705	Iron Man 2	2010	PG-13	124	2488496	NULL	BEST-PICTURE	2016
0435761	Toy Story 3	2010	G	103	2488496	1111111	BEST-ACTOR	2016
0435761	Toy Story 3	2010	G	103	1228705	2222222	BEST-ACTRESS	2011
0435761	Toy Story 3	2010	G	103	2488496	NULL	BEST-PICTURE	2016
1323594	Despicable Me	2010	PG	95	2488496	1111111	BEST-ACTOR	2016
1323594	Despicable Me	2010	PG	95	1228705	2222222	BEST-ACTRESS	2011
1323594	Despicable Me	2010	PG	95	2488496	NULL	BEST-PICTURE	2016

From earlier in the lecture:
How many rows would be in this result?

```
SELECT name
FROM Movie, Oscar
WHERE id = movie_id;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

Movie x Oscar, followed by join condition

id	name	Movie. year	rating	runtime	movie_id	person_id	type	Oscar. year
2488496	Star Wars...	2015	PG-13	138	2488496	1111111	BEST-ACTOR	2016
2488496	Star Wars...	2015	PG-13	138	2488496	NULL	BEST-PICTURE	2016
1228705	Iron Man 2	2010	PG-13	124	1228705	2222222	BEST-ACTRESS	2011

after SELECT

name
Star Wars...
Star Wars...
Iron Man 2

Pre-Lecture
SQL: Outer Joins

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Finding the Room of Each Course

- Need a query that forms (course name, room name) pairs.

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

desired result of the query

Course.name	Room.name
CS 105	GCB 204
CS 111	CAS 314
EN 101	CAS Tsai
CS 460	MCS 205
CS 510	MCS 205
PH 101	NULL

- Will this work?

```
SELECT Course.name, Room.name
FROM Course, Room
WHERE room_id = id;
```

```
SELECT Course.name, Room.name
FROM Course, Room
WHERE room_id = id;
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course x Room **42 rows in all!**

Course.name	start_time	end_time	room_id	id	Room.name	capacity
CS 105	13:00:00	14:00:00	4000	1000	CAS Tsai	500
CS 105	13:00:00	14:00:00	4000	2000	CAS BigRoom	100
CS 105	13:00:00	14:00:00	4000	3000	EDU Lecture Hall	100
CS 105	13:00:00	14:00:00	4000	4000	GCB 204	40
CS 105	13:00:00	14:00:00	4000	5000	CAS 314	80
CS 105	13:00:00	14:00:00	4000	6000	CAS 226	50
CS 105	13:00:00	14:00:00	4000	7000	MCS 205	30
CS 111	09:30:00	11:00:00	5000	1000	CAS Tsai	500
CS 111	09:30:00	11:00:00	5000	2000	CAS BigRoom	100
CS 111	09:30:00	11:00:00	5000	3000	EDU Lecture Hall	100
CS 111	09:30:00	11:00:00	5000	4000	GCB 204	40
CS 111	09:30:00	11:00:00	5000	5000	CAS 314	80
...						

```
SELECT Course.name, Room.name
FROM Course, Room
WHERE room_id = id;
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

Course x Room, followed by the join condition

Course.name	start_time	end_time	room_id	id	Room.name	capacity
CS 105	13:00:00	14:00:00	4000	4000	GCB 204	40
CS 111	09:30:00	11:00:00	5000	5000	CAS 314	80
EN 101	11:00:00	12:30:00	1000	1000	CAS Tsai	500
CS 460	16:00:00	17:30:00	7000	7000	MCS 205	30
CS 510	12:00:00	13:30:00	7000	7000	MCS 205	30

Course.name	Room.name
CS 105	GCB 204
CS 111	CAS 314
EN 101	CAS Tsai
CS 460	MCS 205
CS 510	MCS 205

- The last row of `Course` doesn't have a match in `Room`.
  - it is an "unmatched row"
  - thus it's not in the result of the join
  - to get it, we need an *outer join*

```
SELECT Course.name, Room.name
FROM Course LEFT OUTER JOIN Room ON room_id = id;
```

the "left" table (the one to the left of `LEFT OUTER JOIN`)

the "right" table

the join condition goes in a special `ON` clause

- A left outer join includes unmatched rows from the left table in the result.

```
SELECT Course.name, Room.name
FROM Course LEFT OUTER JOIN Room ON room_id = id;
```

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

result of the LEFT OUTER JOIN

Course.name	start_time	end_time	room_id	id	Room.name	capacity
CS 105	13:00:00	14:00:00	4000	4000	GCB 204	40
CS 111	09:30:00	11:00:00	5000	5000	CAS 314	80
EN 101	11:00:00	12:30:00	1000	1000	CAS Tsai	500
CS 460	16:00:00	17:30:00	7000	7000	MCS 205	30
CS 510	12:00:00	13:30:00	7000	7000	MCS 205	30
PH 101	14:30:00	16:00:00	NULL	NULL	NULL	NULL

Course.name	Room.name
CS 105	GCB 204
CS 111	CAS 314
EN 101	CAS Tsai
CS 460	MCS 205
CS 510	MCS 205
PH 101	NULL

- A *left outer join* adds an extra row to its result for any row from the left table that doesn't have a match in the right.
  - uses `NULL`s for the right-table attributes in the extra rows

SQL: Outer Joins

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: What does this give?

```
SELECT name
FROM Movie, Oscar;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

How can we get just the movies that won Oscars?

```
SELECT name
FROM Movie, Oscar;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

What does this give?

```
SELECT name, COUNT(*)
FROM Movie, Oscar
WHERE id = movie_id
GROUP BY name;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

What if we wanted a count for each movie — including non-Oscar winners?

```
SELECT name, COUNT(*)
FROM Movie, Oscar
WHERE id = movie_id
GROUP BY name;
```

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

Movie x Oscar, followed by join condition, followed by GROUP BY

id	name	Movie. year	rating	runtime	movie_id	person_id	type	Oscar. year
2488496	Star Wars...	2015	PG-13	138	2488496	1111111	BEST-ACTOR	2016
2488496	Star Wars...	2015	PG-13	138	2488496	NULL	BEST-PICTURE	2016
1228705	Iron Man 2	2010	PG-13	124	1228705	2222222	BEST-ACTRESS	2011

after SELECT

name	COUNT(*)
Star Wars...	2
Iron Man 2	1

name	COUNT
Star Wars...	2
Iron Man 2	1
Toy Story 3	0
Despicable Me	0

Which of these would work?

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

name	COUNT
Star Wars...	2
Iron Man 2	1
Toy Story 3	0
Despicable Me	0

A.
```
SELECT name, COUNT(*)
FROM Movie, Oscar
WHERE id = movie_id
GROUP BY name;
```

B.
```
SELECT name, COUNT(type)
FROM Movie, Oscar
WHERE id = movie_id
GROUP BY name;
```

C.
```
SELECT name, COUNT(type)
FROM Movie LEFT OUTER JOIN Oscar
ON id = movie_id
GROUP BY name;
```

D.
```
SELECT name, COUNT(*)
FROM Movie LEFT OUTER JOIN Oscar
ON id = movie_id
GROUP BY name;
```

Finding the Majors of Enrolled Students

- We want the IDs and majors of every student who is enrolled in a course – including those with no major.

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Desired result:

student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
33566891	null

Which of these would work?

We want the IDs and majors of every student who is enrolled in a course – including those with no major.

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

A.
```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled, MajorsIn
WHERE Enrolled.student_id = MajorsIn.student_id;
```

B.
```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM MajorsIn LEFT OUTER JOIN Enrolled
WHERE Enrolled.student_id = MajorsIn.student_id;
```

C.
```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

D.
```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM MajorsIn LEFT OUTER JOIN Enrolled
ON Enrolled.student_id = MajorsIn.student_id;
```

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
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 rows in T1 x T2 that satisfy the join condition in the `ON` clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
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 rows in T1 x T2 that satisfy the join condition in the `ON` clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Enrolled. student_id	*course_ name*	*credit_ status*	*MajorsIn. student_id*	*dept_name*
12345678	CS 105	ugrad	12345678	comp sci
12345678	CS 105	ugrad	45678900	math...
12345678	CS 105	ugrad	25252525	comp sci
12345678	CS 105	ugrad	45678900	english
12345678	CS 105	ugrad	66666666	the occult
25252525	CS 111	ugrad	12345678	comp sci
25252525	CS 111	ugrad	45678900	math...
25252525	CS 111	ugrad	25252525	comp sci
25252525	CS 111	ugrad	45678900	english
25252525	CS 111	ugrad	66666666	the occult
45678900	CS 460	grad	12345678	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	25252525	comp sci
45678900	CS 460	grad	45678900	english
45678900	CS 460	grad	66666666	the occult
33566891	CS 105	non-cr	12345678	comp sci
33566891	CS 105	non-cr	45678900	math...
33566891	CS 105	non-cr	25252525	comp sci
33566891	CS 105	non-cr	45678900	english

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

SELECT …
FROM *T1* LEFT OUTER JOIN *T2* ON *join condition*
WHERE …

Enrolled.student_id	course_name	credit_status	MajorsIn.student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - *selecting the rows in T1 x T2 that satisfy the join condition in the ON clause*
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

SELECT …
FROM *T1* LEFT OUTER JOIN *T2* ON *join condition*
WHERE …

Enrolled.student_id	course_name	credit_status	MajorsIn.student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr		

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - *including an extra row for each unmatched row from T1 (the "left table")*
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
SELECT …
FROM T1 LEFT OUTER JOIN T2 ON join condition
WHERE …
```

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr	null	null

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - *filling the T2 attributes in the extra rows with nulls*
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
SELECT …
FROM T1 LEFT OUTER JOIN T2 ON join condition
WHERE …
```

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr	null	null

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls

  - *applying the other clauses as before*

Enrolled. student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
33566891	null

Outer Joins Can Have a WHERE Clause

- Example: find the IDs and majors of all students enrolled **in CS 105** (including those with no major):

```
SELECT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
     ON Enrolled.student_id = MajorsIn.student_id
WHERE course_name = 'CS 105';
```

  - to limit the results to students in CS 105, we need a `WHERE` clause with the appropriate condition
  - this new condition should *not* be in the `ON` clause because it's *not* being used to match up rows from the two tables

Outer Joins Can Have Extra Tables

- Example: find the **names** and majors of all students enrolled in CS 105 (including those with no major):

```
SELECT Student.name, dept_name
FROM Student, Enrolled LEFT OUTER JOIN MajorsIn
     ON Enrolled.student_id = MajorsIn.student_id
WHERE Student.id = Enrolled.student_id
  AND course_name = 'CS 105';
```

  - we need Student in the FROM clause to get the student's names
  - the extra table requires an additional join condition, which goes in the WHERE clause

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 needed? – i.e., do you want 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?
  - is DISTINCT needed?

From earlier in the lecture: Which of these would work?

Movie

id	name	year	rating	runtime
2488496	Star Wars...	2015	PG-13	138
1228705	Iron Man 2	2010	PG-13	124
0435761	Toy Story 3	2010	G	103
1323594	Despicable Me	2010	PG	95

Oscar

movie_id	person_id	type	year
2488496	1111111	BEST-ACTOR	2016
1228705	2222222	BEST-ACTRESS	2011
2488496	NULL	BEST-PICTURE	2016

`Movie LEFT OUTER JOIN Oscar ON id = movie_id GROUP BY name`

id	name	Movie. year	rating	runtime	movie_id	person_id	type	Oscar. year
2488496	Star Wars...	2015	PG-13	138	2488496	1111111	BEST-ACTOR	2016
2488496	Star Wars...	2015	PG-13	138	2488496	NULL	BEST-PICTURE	2016
1228705	Iron Man 2	2010	PG-13	124	1228705	2222222	BEST-ACTRESS	2011
0435761	Toy Story 3	2010	G	103	NULL	NULL	NULL	NULL
1323594	Despicable Me	2010	PG	95	NULL	NULL	NULL	NULL

C.
```
SELECT name, COUNT(type)
FROM Movie LEFT OUTER JOIN Oscar
ON id = movie_id
GROUP BY name;
```

name	COUNT(type)
Star Wars...	2
Iron Man 2	1
Toy Story 3	0
Despicable Me	0

D.
```
SELECT name, COUNT(*)
FROM Movie LEFT OUTER JOIN Oscar
ON id = movie_id
GROUP BY name;
```

name	COUNT(*)
Star Wars...	2
Iron Man 2	1
Toy Story 3	1
Despicable Me	1

We want the IDs and majors of every student who is enrolled in a course – including those with no major.

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled, MajorsIn
WHERE Enrolled.student_id = MajorsIn.student_id;

SELECT DISTINCT Enrolled.student_id, dept_name
FROM MajorsIn LEFT OUTER JOIN Enrolled
WHERE Enrolled.student_id = MajorsIn.student_id;
```

student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

get unmatched rows from the "left" table -- `Enrolled`

student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
33566891	null

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM MajorsIn LEFT OUTER JOIN Enrolled
ON Enrolled.student_id = MajorsIn.student_id;
```

get unmatched rows from the "left" table -- `MajorsIn`

student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
null	the occult

Pre-Lecture
SQL: Other Commands

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

DELETE: Removing Existing Rows

- syntax: `DELETE FROM` *table*
  `WHERE` *selection condition*;

```
DELETE FROM Student
WHERE id = '45678900';
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
66666666	Count Dracula

```
DELETE FROM Enrolled
WHERE student_id = '45678900';
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
33566891	CS 105	non-credit

The order of deletions can matter!

```
DELETE FROM Student
WHERE id = '45678900';
```

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
66666666	Count Dracula

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

The order of deletions can matter! (cont.)

- Before deleting a row, we must first remove all references to that row from foreign keys in other tables.

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

UPDATE: Changing Values in Existing Rows

- syntax:
```
UPDATE table
SET list of changes
WHERE selection condition;
```

```
UPDATE MajorsIn
SET dept_name = 'physics'
WHERE student_id = '45678900';
```

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	physics
25252525	comp sci
45678900	physics
66666666	the occult

UPDATE: Changing Values in Existing Rows

- syntax:
```
UPDATE table
SET list of changes
WHERE selection condition;
```


```
UPDATE MajorsIn
SET dept_name = 'physics'
WHERE student_id = '45678900'
  AND ____________________;
```

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	physics
66666666	the occult

UPDATE: Changing Values in Existing Rows (cont.)

- syntax: `UPDATE` *table*
  `SET` *list of changes*
  `WHERE` *selection condition*`;`

```
UPDATE Course
SET start_time = '13:25:00', end_time = '14:15:00',
    room_id = '6000'
WHERE name = 'CS 105';
```

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

➡

Course

name	*start_time*	*end_time*	*room_id*
CS 105	13:25:00	14:15:00	6000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

DROP TABLE: Removing an Entire Table

- syntax: `DROP TABLE` *table*`;`

```
DROP TABLE MajorsIn;
```

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

➡ no table!

- If a table is referred to by a foreign key in another table, it cannot be dropped until either:
  - the other table is dropped first

    or
  - the foreign-key constraint is removed from the other table (we won't look at how to do this)

SQL: Other Commands; Practice with Queries

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Does the order of these deletions matter?

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

1 `DELETE FROM Course WHERE name = 'CS 111';`

2 `DELETE FROM Enrolled WHERE course_name = 'CS 111';`

A. 1 must come before 2
B. 2 must come before 1
C. the order of the two DELETE commands doesn't matter

Is this deletion also needed?

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

```
DELETE FROM Enrolled WHERE course_name = 'CS 111';
DELETE FROM Room WHERE id = '5000';
DELETE FROM Course WHERE name = 'CS 111';
```

no! when deleting a row that includes a foreign key, we *don't* need to delete what the foreign key refers to.

How could I correctly remove MCS 205?

Course

name	start_time	end_time	room_id
CS 105	13:00:00	14:00:00	4000
CS 111	09:30:00	11:00:00	5000
EN 101	11:00:00	12:30:00	1000
CS 460	16:00:00	17:30:00	7000
CS 510	12:00:00	13:30:00	7000
PH 101	14:30:00	16:00:00	NULL

Room

id	name	capacity
1000	CAS Tsai	500
2000	CAS BigRoom	100
3000	EDU Lecture Hall	100
4000	GCB 204	40
5000	CAS 314	80
6000	CAS 226	50
7000	MCS 205	30

A.
```
DELETE FROM Room WHERE id = '7000';
```

B.
```
DELETE FROM Room WHERE id = '7000';
UPDATE Course SET room_id = NULL WHERE room_id = '7000';
```

C.
```
UPDATE Course SET room_id = NULL WHERE room_id = '7000';
DELETE FROM Room WHERE id = '7000';
```

D. two or more of the above would work

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 needed? – i.e., do you want 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?
  - is DISTINCT needed?

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 the names of all people in the database who acted in *Avatar.*

2) How many people in the database who were born in the state of California have won an Oscar? (assume pob = city, state, country)

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 did not act in *Avatar*?

Why won't this work?

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

What will?

SQL: More Practice with Queries

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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 needed? – i.e., do you want 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?
  - is DISTINCT needed?

Which of these problems would require a GROUP BY?

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)

A. finding the Best-Picture winner with the best/smallest earnings rank

B. finding the number of Oscars won by each movie that has won an Oscar

C. finding the number of Oscars won by each movie, including movies that have not won any Oscars

D. both B and C, but not A

E. A, B, and C

Which would require a subquery?

Which would require a LEFT OUTER JOIN?

Now Write the 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 the Best-Picture winner with the best/smallest earnings rank.
   The result should have the form (name, earnings_rank).
   Assume no two movies have the same earnings rank.

```
SELECT
FROM
WHERE                   (SELECT
                         FROM
                         WHERE

                                                    );
```

Now Write the 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)

2) Find the number of Oscars won by each movie that has won an Oscar. Produce tuples of the form (name, num Oscars).

3) Find the number of Oscars won by each movie, including movies that have not won an Oscar.

Even More Practice!

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 movie ratings have an avg runtime greater than 120 min?

Even More Practice! (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) 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.

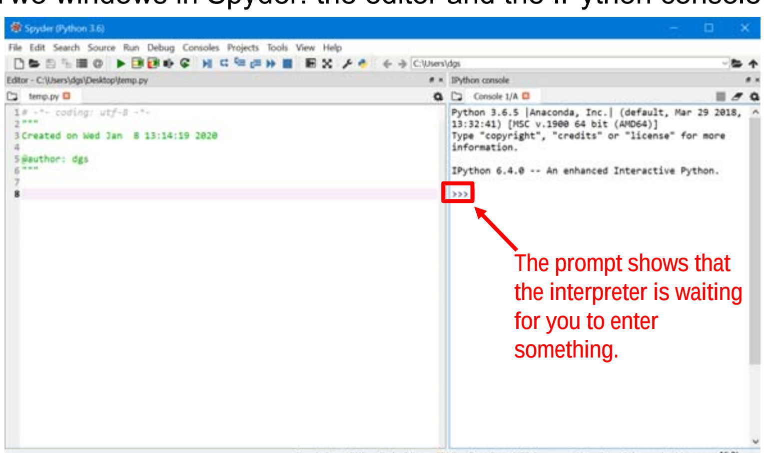
Pre-Lecture
Getting Started With Python

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Interacting with Python

- We're using Python **3** (*not* 2).
    - see Lab 0 for how to install and configure Spyder
- Two windows in Spyder: the editor and the IPython console



Arithmetic in Python

- Numeric operators include:
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `**` exponentiation
  - `%` modulus: gives the remainder of a division

Arithmetic in Python (cont.)

- The operators follow the standard order of operations.
  - example: multiplication before addition
- You can use parentheses to force a different order.

Data Types

- Different kinds of values are stored and manipulated differently.
- Python *data types* include:
  - integers
    - example: `451`
  - floating-point numbers
    - numbers that include a decimal
    - example: `3.1416`

Data Types and Operators

- There are really <u>two sets</u> of numeric operators:
  - one for integers (ints)
  - one for floating-point numbers (floats)
- 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
- One exception: division!

Two Types of Division

- The / operator *always* produces a float result.
  - examples:

```
>>> 5 / 3
1.6666666666666667
>>> 6 / 3
```

____

Two Types of Division (cont.)

- There is a separate // operator for *integer* division.

```
>>> 6 // 3
2
```

- Integer division *discards* any fractional part of the result:

```
>>> 11 // 5
2
>>> 5 // 3
```

____

- Note that it does *not* round!

Another Data Type

- A *string* is a sequence of characters/symbols
  - surrounded by single or double quotes
  - examples: `"hello"` `'Picobot'`

Pre-Lecture
Program Building Blocks: Variables, Expressions, Statements

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Variables

- Variables allow us to store a value for later use:

```
>>> temp = 77
>>> (temp - 32) * 5 / 9
25.0
```

Expressions

- *Expressions* produce a value.
  - We *evaluate* them to obtain their value.
- They include:
  - *literals* ("hard-coded" values):
    ```
    3.1416
    'Picobot'
    ```
  - variables
    ```
    temp
    ```
  - combinations of literals, variables, and operators:
    ```
    (temp - 32) * 5 / 9
    ```

Evaluating Expressions with Variables

- When an expression includes variables, they are first replaced with their current value.
- Example:

```
(temp - 32) * 5 / 9
( 77  - 32) * 5 / 9
      45    * 5 / 9
          225   / 9
              25.0
```

Statements

- A *statement* is a command that carries out an action.
- A *program* is a sequence of statements.

```
quarters = 2
dimes = 3
nickels = 1
pennies = 4
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print('you have', cents, 'cents')
```

Assignment Statements

- *Assignment statements* store a value in a variable.

```
temp = 20
```

- General syntax:

  variable = expression

= is known as the *assignment operator*

- Steps:
  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
quarters_val = 25 * quarters
```

Assignment Statements (cont.)

- We can change the value of a variable by assigning it a new value.
- Example:

`num1 = 100` `num2 = 120`	`num1` 100	`num2` 120
`num1 = 50`	`num1`	`num2` 120
`num1 = num2 * 2`	`num1`	`num2` 120
`num2 = 60`	`num1`	`num2`

Assignment Statements (cont.)

- An assignment statement does not create a permanent relationship between variables.
- ***You can only change the value of a variable by assigning it a new value!***

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!
- Example:

`sum = 13` `val = 30`	sum: 13 val: 30
`sum = sum + val` `13 + 30` `43`	sum: 43 val: 30
`val = val * 2`	sum: val:

Creating a Reusable Program

- Put the statements in a text file.

```
# a program to compute the value of some coins

quarters = 2       # number of quarters
dimes = 3
nickels = 1
pennies = 4

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

- Program file names should have the extension `.py`
  - example: `coins.py`

Print Statements

- print statements display one or more values on the screen
- Basic syntax:

  `print(`*expr*`)`

  or

  `print(`$expr_1$`, `$expr_2$`, ... `$expr_n$`)`

  where each *expr* is an expression

- Steps taken when executed:
  1) the individual expression(s) 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:

```
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)

  - second example:

```
cents = 89
print('you have', cents, 'cents')
```

Variables and Data Types

- The `type` function gives us the type of an expression:

```
>>> type('hello')
<class 'str'>
>>> type(5 / 2)
<class 'float'>
```

- Variables in Python do *not* have a fixed type.
  - examples:

```
>>> temp = 25.0
>>> type(temp)
<class 'float'>
>>> temp = 77
>>> type(temp)
```

Getting Started with Python

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Beyond Relational Databases

- While relational databases are extremely powerful, they may be inadequate/insufficient for a given problem.
- Example 1: DNA sequence data

```
>gi|49175990|ref|NC_000913.2| Escherichia coli K12, complete genome
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTA
AATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAA
ACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGA
CAGTGCGGGCTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTC
TGCGTGTTGCCGATATTCTGGAAAGCAATGCCAGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTGGCGAT
GATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAACGTATTTTTGCCGAACTTTTGACGGGACTCGCCGCCGCCCAG
CCGGGGTTCCCGCTGGCGCAA
```

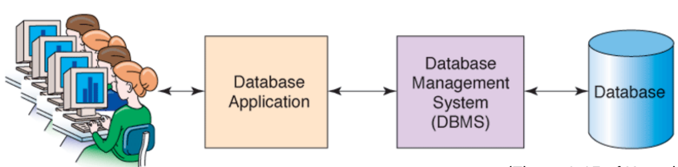
  - common queries involve looking for similarities or patterns
    - what genes in mice are similar to genes in humans?
  - need special *algorithms* (problem-solving procedures) for finding statistically significant similarities
  - biologists store this data in text files and use computer programs to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



(Figure 1-15 of Kroenke)

  - 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

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

Summary: Python Building Blocks

- The building blocks of a Python program include:
  - literals
  - variables
  - expressions
  - statements

Which of these are *expressions*?

A. `105`
B. `x`
C. `x + y`
D. both B and C, but not A
E. A, B, and C

Which of these are *expressions*?

An *expression* is anything that produces a value!

Another definition: anything you can print is an expression.

A. `105` — *literals* evaluate to themselves!

B. `x` — *variables* evaluate to their current values

C. `x + y` — in *expressions with operators, we replace variables* with their current values and then apply the operators

D. both B and C, but not A

E. A, B, and C

How a Program Flows...

- Flow of control = order in which statements are executed
- By default, a program's statements are executed sequentially, from top to bottom.

example program

```
total = 0
num1 = 5
num2 = 10
total = num1 + num2
```

variables in memory

total	
num1	5
num2	10

What is the output of the following program?

```
x = 7
name = 'Olivia'
y = x / 2
x = 11
print('name', x, y)
```

note: we do *not* print:

- commas between expressions
- quotes around string literals

What about this program?

```
x = 7
name = 'Olivia'
y = x / 2
x = 11
print(name, 'x', y * 2)
```

What are the final values of the variables after the following program runs?

```
x = 5
y = 6
x = y + 3
z = x // 2
x = x % 2
```

x	y	z
5		
5	6	

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)

Overview of the Programming Process

Analysis/Specification

↓

Design

↓

Implementation

↓

Testing/Debugging

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 from the video:

```
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!

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

Pre-Lecture
Python: Built-in Functions and User Input

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Functions

- Python comes with a number of built-in *functions* that we can use in our programs.
- `print` is one example.
- A function may take one or more *parameters.*
  - for `print`, the parameters are the expressions whose values you want to print
- Example of *calling* a function:

```
print('you have', cents, 'cents')
```

function name *parameters*

Functions (cont.)

- Some functions *return* (i.e., output) a value.
- Example: the `abs` function
  - parameter: a number *n*
  - return value (output): the absolute value of *n*
- Example: the `int` function
  - parameter: a string representing a number
  - return value (output): the number as a value of type int
  - examples:
    - `int('15')` returns `15`
    - `int('3.75')` returns ____________

The Input Function

- The `input` function allows us to get values from the user.
  - parameter: a string that serves as a prompt

    ```
    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:

  ```
  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 an integer from the user, we can combine the `input` function with the `int` function

```
quarters = int(input('number of quarters? '))
```

Getting Numeric Input

- We evaluate the functions from the inside out:

```
quarters = int(input('number of quarters? '))
           int('17')
           17
```

Pre-Lecture
Python: A First Look at Lists; the `range()` Function

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Lists

- Recall: A string is a sequence of characters.
  `'hello'`

- A list is a sequence of *arbitrary* values (the list's *elements*).
  ```
  [2, 4, 6, 8]
  ['CS', 'math', 'english', 'psych']
  ```

- A list can include values of different types:
  ```
  ['Star Wars', 1977, 'PG', [35.9, 460.9]]
  ```

Generating a Range of Integers

- `range(low, high)`: allows us to work with the range of integers from `low` to `high-1`
  - to see the result produced by `range()` use the `list()` function
  - if you omit `low`, the range will start at 0

Pre-Lecture
`for` Loops in Python

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

`for` Loops

- A `for` statement is one way to create a loop in Python.
  - allows us to *repeat* one or more statements.
- Example:

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```

the body of the loop

will output:

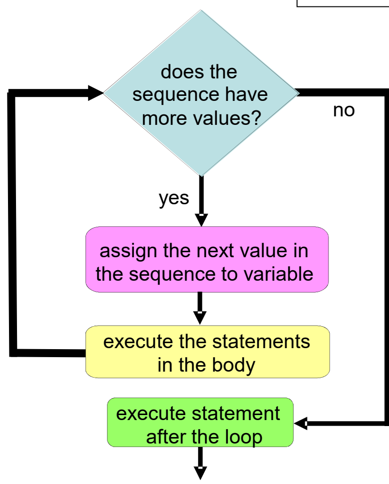
```
Warning
1
Warning
2
Warning
3
```

for Loops (cont.)

- General syntax:

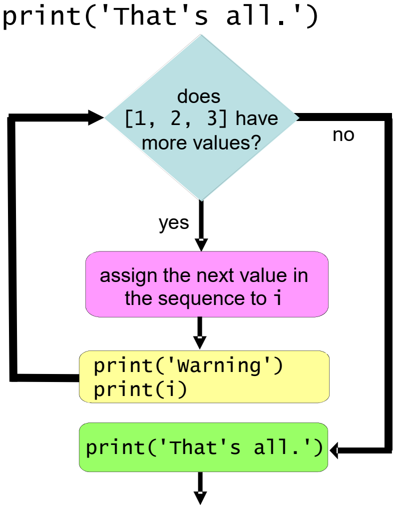
```
for variable in sequence:
    body of the loop
```

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```



Executing Our Earlier Example
(with one extra statement)

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



more?	i	output/action

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):          # [0, 1, 2, ..., N – 1]
    body of the loop
```

- Example:

```
for i in range(3):          # [0, 1, 2]
    print('I'm feeling loopy!')
```

outputs:

```
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
```

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):          # [0, 1, 2, ..., N – 1]
    body of the loop
```

- Example:

```
for i in range(5):
    print('I'm feeling loopy!')
```

outputs:

More Python Basics

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

What does this program print?

```
x = -10
y = x ** 2
abs(x)
print(x, y)
```

How could we make the program print
10 100

```
x = -10
y = x ** 2
abs(x)
print(x, y)
```

Note the Difference!

- In the Shell, entering a function call or other expression displays the value/result of the expression:

```
>>> abs(-20)
20
>>> 2 ** 10
1024
```

- In a program, you will only see a value/result if your print it!

program		*output*
`abs(-20)` `2 ** 10`	⇨	
`print(abs(-20))` `print(2 ** 10)`	⇨	`20` `1024`

Recall Our Earlier Example Program...

```
quarters = 2
dimes = 3
nickels = 1
pennies = 4

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

An Improved Version with User Input!

```
quarters = int(input('number of quarters? '))
dimes = int(input('number of dimes? '))
nickels = int(input('number of nickels? '))
pennies = int(input('number of pennies? '))

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

- Note the use of the `int()` function to convert the user's inputs to integers.

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 an integer from the user, we can combine the `input` function with the `int` function

```
quarters = int(input('number of quarters? '))
```

- To get a numeric value with a decimal from the user, we combine `input` with the `float` function

```
price = float(input('enter the price: '))
```

Identifiers

```
quarters = int(input('number of quarters? '))
dimes = int(input('number of dimes? '))
nickels = int(input('number of nickels? '))
pennies = int(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

    `quarters` `dimes` `nickels` `pennies` `cents`
  - function names like `int`, `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
```

- Unlike SQL, Python is *case-sensitive.*
  - for both identifiers and keywords
  - example: `quarters` is not the same as `Quarters`

How Many Values Will Be Printed?

```
for val in [2, 4, 6, 8, 10]:
    print(val * 10)
print(val)
```

Tracing a for Loop

- Let's trace the execution of this code:

```
for val in [2, 4, 6, 8, 10]:
    print(val * 10)
print(val)
```

- Use a table to help you:

more?	val	output/action
yes	2	20

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):        # [0, 1, 2, ..., N – 1]
    body of the loop
```

- What would this loop do?

```
for i in range(8):
    print('I'm feeling loopy!')
```

Simple Repetition Loops (cont.)

- Another example:

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

output?

To print the warning 10 times, how could you fill in the blank?

```
for i in ________________:
    print('Warning!')
```

A. `range(10)`

B. `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`

C. `range(1, 11)`

D. either A or B would work, but not C

E. A, B or C would work

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.
  - makes the output go to the beginning of the next line
- We can use a special `end` parameter to replace the newline with a different character.
- Example:

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

will output

```
0 0
1 5
2 10
3 15
4 20
```

```
for i in range(5):
    print(i, i*5, end=' ')
```

will output

```
0 0 1 5 2 10 3 15 4 20
```

Pre-Lecture
Writing Your Own Functions

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Defining a Function

the function's name

x is the input or *parameter*

```
def triple(x):
    return 3*x
```

must indent

this line specifies what the function outputs (or *returns*) – in this case, 3 times the input

- Once we define a function, we can call it:

```
>>> triple(5)
15
>>> triple(1.5)


>>> triple('hello')
```

Multiple Lines, Multiple Parameters

```
def circle_area(diam):
    """ Computes the area of a circle
        with a diameter diam.
    """
    radius = diam / 2
    area = 3.14159 * (radius**2)
    return area

def rect_perim(l, w):
    """ Computes the perimeter of a rectangle
        with length l and width w.
    """
    return 2*l + 2*w
```

- Examples:

```
>>> circle_area(20)
314.159
>>> rect_perim(5, 7)
```

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a + b + 3


print(calculate(3, 2))
```

x	y	a	b
3	2		

(see next slide)

The values in the function call are assigned to the parameters.

In this case, it's as if we had written:

```
x = 3
y = 2
```

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a + b + 3


print(calculate(3, 2))     # print(__________)
```

x	y	a	b
3	2		

The output/return value:
- is sent back to where the function call was made
- replaces the function call

The program picks up where it left off when the function call was made.

Pre-Lecture
Cumulative Computations

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Python Shortcuts

- Consider this code:

```
age = 13
age = age + 1
      13 + 1
        14
```

- Instead of writing

```
age = age + 1
```

  we can just write

```
age += 1
```

Python Shortcuts (cont.)

shortcut	equivalent to
var += *expr*	*var* = *var* + (*expr*)
var -= *expr*	*var* = *var* - (*expr*)
var *= *expr*	*var* = *var* * (*expr*)
var /= *expr*	*var* = *var* / (*expr*)
var //= *expr*	*var* = *var* // (*expr*)
var %= *expr*	*var* = *var* % (*expr*)
var **= *expr*	*var* = *var* ** (*expr*)

where *var* is a variable
expr is an expression

- **Important:** the = must come *after* the other operator.
  += is correct
  =+ is not!

Using a Loop to Sum a List of Numbers

```
def sum(vals):
    result = 0          # the accumulator variable
    for x in vals:
        result += x     # gradually accumulates the sum
    return result

print(sum([10, 20, 30, 40, 50]))
```

x   result

Writing Your Own Functions; Cumulative Computations

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

What is the output of this code?

x	y	a	b

```
def calculate(x, y):
    a = y
    b = x + 1
    return a * b - 3


print(calculate(3, 2))
```

Practice Writing a Function

- Write a function `feet_to_cm(ft)` that takes a length in feet (`ft`) and returns the equivalent length in centimeters.
  - 1 foot = 12 inches
  - 1 inch = 2.54 centimeters
  - examples:

    ```
    >>> feet_to_cm(3)
    91.44
    >>> feet_to_cm(10)
    304.8
    ```

  - template:

    ```
    def feet_to_cm(ft):
        """ converts ft feet to centimeters """

        inches = ____________

        __________________________ # optional

        return ______________
    ```

All of these work!

```
def feet_to_cm(ft):
    """ converts ft feet to centimeters """
    inches = ft * 12
    cm = inches * 2.54
    return cm

def feet_to_cm(ft):
    """ converts ft feet to centimeters """
    inches = ft * 12
    return inches * 2.54

def feet_to_cm(ft):
    """ converts ft feet to centimeters """
    return ft * 12 * 2.54
```

These are *not* the same!

```
def feet_to_cm(ft):
    """ converts ft feet to centimeters """
    inches = ft * 12
    cm = inches * 2.54
    return cm

def feet_to_cm(ft):
    """ converts ft feet to centimeters """
    inches = ft * 12
    cm = inches * 2.54
    print(cm)
```

Recall: Using a Loop to Sum a List of Numbers

```
def sum(vals):                # vals = [10, 20, 30, 40, 50]
    result = 0
    for x in vals:
        result += x
    return result

print(sum([10, 20, 30, 40, 50]))
```

x	result

Recall: Using a Loop to Sum a List of Numbers

```
def sum(vals):                # vals = [10, 20, 30, 40, 50]
    result = 0
    for x in vals:
        result += x
    return result             # return 150

print(sum([10, 20, 30, 40, 50]))
```

x	result
	0
10	10
20	30
30	60
40	100
50	150
	no more values in vals, so we're done

Cumulative Computations

```
def sum(vals):
    result = 0         # the accumulator variable
    for x in vals:
        result += x    # gradually accumulates the sum
    return result

print(sum([10, 20, 30, 40, 50]))
```

x	result
	0
10	10
20	30
30	60
40	100
50	150

no more values in `vals`, so we're done

output: 150

Summing User Inputs

- Let's trace through the code below for the inputs 7, 9, 11, 8, 6:

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

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

i	num	total
		0
0	7	

Making the Program More Flexible

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

```
total = 0
for i in range(5):
    num = int(input('enter a number: '))
    total = total + num

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

Pre-Lecture
Making Decisions: Conditional Execution

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Conditional Execution

- Conditional execution allows your code to *decide* whether to do something, based on some condition.
  - example:

```
def abs_value(x):
    """ returns the absolute value of input x """
    if x < 0:
        x = -1 * x
    return x
```

  - examples of calling this function from the Shell:

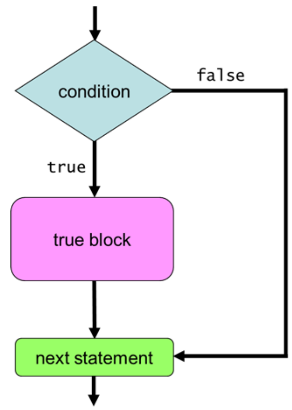
```
>>> abs_value(-5)
5
>>> abs_value(10)
```

Simple Decisions: `if` Statements

- Syntax:

```
if condition:
    true block
```

where:
  - *condition* is an expression that is true or false
  - *true block* is one or more indented statements



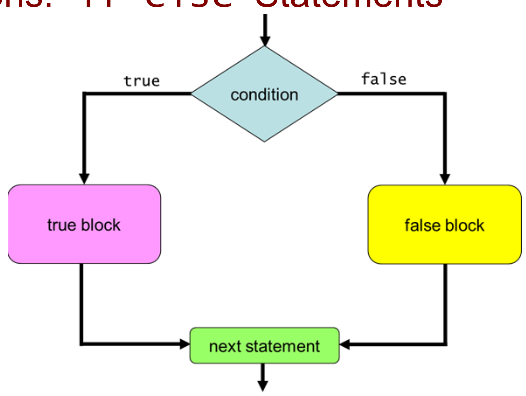
- Example:

```
def abs_value(x):
    if x < 0:
        x = -1 * x     # true block
    return x
```

Two-Way Decisions: `if-else` Statements

- Syntax:

```
if condition:
    true block
else:
    false block
```



- Example:

```
def pass_fail(avg):
    if avg >= 60:
        grade = 'pass'     # true block
    else:
        grade = 'fail'     # false block
    return grade
```

Tracing Conditional Execution: Example 1

```
def pass_fail(avg):
    if avg >= 60:
        grade = 'pass'
    else:
        grade = 'fail'
    return grade
```

avg

grade

```
>>> pass_fail(80)
```

Tracing Conditional Execution: Example 2

```
def pass_fail(avg):
    if avg >= 60:
        grade = 'pass'
    else:
        grade = 'fail'
    return grade
```

avg

grade

```
>>> pass_fail(55)
```

Expressing Simple Conditions

- Python provides a set of *relational operators* for making comparisons:

operator	name	examples
`<`	less than	`val < 10` `price < 10.99`
`>`	greater than	`num > 60` `state > 'Ohio'`
`<=`	less than or equal to	`average <= 85.8`
`>=`	greater than or equal to	`name >= 'Jones'`
`==` *(don't confuse with =)*	equal to	`total == 10` `letter == 'P'`
`!=`	not equal to	`age != my_age`

Boolean Values and Expressions

- A condition has one of two values: `True` or `False`.

```
>>> 10 < 20
True
>>> "Jones" == "Baker"
False
```

- `True` and `False` are *not* strings.
  - they are literals from the `bool` data type

```
>>> type(True)
<class 'bool'>
>>> type(30 > 6)
<class 'bool'>
```

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

Forming More Complex Conditions

- Python provides *logical operators* for combining/modifying boolean expressions:

name	example and meaning
`and`	`age >= 18 and age <= 35` `True` *if both conditions are* `True`, *and* `False` *otherwise*
`or`	`age < 3 or age > 65` `True` *if one or both of the conditions are* `True`; `False` *if both conditions are* `False`
`not`	`not (grade > 80)` `True` *if the condition is* `False`, *and* `False` *if it is* `True`

A Word About Blocks

- A block can contain multiple statements.

```
def welcome(class):
    if class == 'frosh':
        print('Welcome to BU!')
        print('Have a great four years!')
    else:
        print('Welcome back!')
        print('Have a great semester!')
        print('Be nice to the frosh students.')
```

- A new block *begins* whenever we *increase* the amount of indenting.

- A block *ends* when we either:
  - reach a line with *less* indenting than the start of the block
  - reach the end of the program

Nesting

- We can "nest" one conditional statement in the true block or false block of another conditional statement.

```
def welcome(class):
    if class == 'frosh':
        print('Welcome to BU!')
        print('Have a great four years!')
    else:
        print('Welcome back!')
        if class == 'senior':
            print('Have a great last year!')
        else:
            print('Have a great semester!')
        print('Be nice to the frosh students.')
```

Multi-Way Decisions

- The following function doesn't work.

```
def letter_grade(avg):
    if avg >= 90:
       grade = 'A'
    if avg >= 80:
       grade = 'B'
    if avg >= 70:
       grade = 'C'
    if avg >= 60:
       grade = 'D'
    else:
       grade = 'F'
    return grade
```

- example:

```
>>> letter_grade(95)
```

____

Multi-Way Decisions (cont.)

- Here's a fixed version:

```
def letter_grade(avg):
    if avg >= 90:
        grade = 'A'
    elif avg >= 80:
        grade = 'B'
    elif avg >= 70:
        grade = 'C'
    elif avg >= 60:
        grade = 'D'
    else:
        grade = 'F'
    return grade
```

- example:

```
>>> letter_grade(95)
```

____

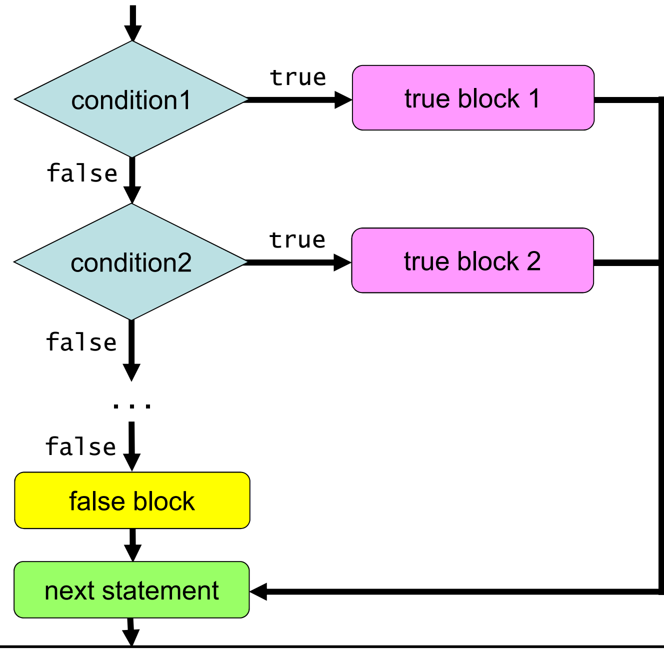
Multi-Way Decisions: `if-elif-else` Statements

- Syntax:

```
if condition1:
    true block for condition1
elif condition2:
    true block for condition2
elif condition3:
    true block for condition3
…
else:
    false block
```

- 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



Python: Working with Numbers; Making Decisions

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Numeric Operators

- Recall the operators for numbers:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	float division
`//`	integer division
`**`	exponentiation
`%`	modulus: gives the remainder of a division
	example: `11 % 3` evaluates to `2`

Using the Operators

- Recall our change-adder program:

```
quarters = int(input('number of quarters? '))
dimes = int(input('number of dimes? '))
nickels = int(input('number of nickels? '))
pennies = int(input('number of pennies? '))

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.
  - for example, 327 cents would print as 3 dollars, 27 cents

How Would Your Complete This Program?

```
quarters = int(input('number of quarters? '))
dimes = int(input('number of dimes? '))
nickels = int(input('number of nickels? '))
pennies = int(input('number of pennies? '))

cents = quarters*25 + dimes*10 + nickels*5 + pennies
dollars = ______________________
cents = ______________________

print('you have', dollars, 'dollars,', cents, 'cents')
```

	first blank	**second blank**
A.	`cents / 100`	`cents % 100`
B.	`cents // 100`	`cents % 100`
C.	`cents / 100`	`cents % dollars`
D.	`cents // 100`	`cents % dollars`

Recall: Two Types of Division

- The `/` operator *always* produces a float result.
  - examples:

```
>>> 5 / 3
1.6666666666666667

>>> 6 / 3
2.0
```

- There is a separate `//` operator for *integer* division, which *discards* (without rounding) anything after the decimal:

```
>>> 6 // 3
2
>>> 11 // 5

__________
>>> 5 // 3

__________
```

Integer Division and the Modulus Operator

- `//` and `%` are often used together, as we just did:

```
dollars = cents // 100
cents = cents % 100
```

- `//` gives the whole-number portion of a division result.
- `%` gives the whole-number remainder of that same result.
- another example: 11 divided by 4... is 2 with a remainder of 3

```
>>> 11 // 4
2

>>> 11 % 4
3
```

Other Uses of the Modulus Operator

- Determining if an integer `n` is even or odd:
  - `n % 2 == 0` if `n` is even
  - `n % 2 == 1` if `n` is odd
- Determining if an integer `n` is a multiple of another integer `m`:
  - `n % m == 0` if `n` is a multiple of `m`
  - `n % m != 0` if `n` is *not* a multiple of `m`

Recall: 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 an integer from the user, we can combine the `input` function with the `int` function

```
quarters = int(input('number of quarters? '))
```

- To get a numeric value with a decimal from the user, we combine `input` with the `float` function

```
price = float(input('enter the price: '))
```

- Entering two or more numbers!

```
a, b = eval(input('enter value 1, value 2: '))
```

  - a is assigned the first value entered
  - b is assigned the second value

Type Conversions

- `float()` and `int()` can also convert from one numeric type to another:
  - `float(n)`: converts `n` to a `float`
  - `int(n)`: converts `n` to an `int`, discarding any fractional part

- Examples:

```
>>> int(8.72532)
8
>>> float(8)
8.0
>>> 15 ** 30
191751059232884086668491363525390625
>>> float(15 ** 30)
1.9175105923288408e+35
```

scientific notation:
`1.9175105923288408 x 10`^{35}

note also: floats have less precision than integers

Type Conversions (cont.)

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

```
>>> measurement = 3.7
>>> int(measurement)
3
>>> measurement
3.7
```

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

```
>>>
```

Rounding a Number

- `round(n)` rounds the number n to an integer:

```
>>> round(7.5)
8
>>> round(7.49)
7
>>> round(2.8)
3
```

- `round(n, d)` rounds the number n to d places after the decimal.

```
>>> round(8.7583, 2)
8.76
>>> round(8.7583, 1)
8.8
>>> round(10.595, 2)
10.6
```

(note that non-essential 0s are not displayed)

What is the output of this program?

```
x = 5
if x < 15:
    if x > 8:
        print('one')
    else:
        print('two')
else:
    if x > 2:
        print('three')
```

What does this print? (note the changes!)

```
x = 5
if x < 15:
    if x > 8:
        print('one')
    else:
        print('two')
if x > 2:
    print('three')
```

What does this print? (note the new changes!)

```
x = 5
if x < 15:
    if x > 8:
        print('one')
else:
    print('two')
if x > 2:
    print('three')
```

How many lines does this print?

```
x = 5
if x == 8:
    print('how')
elif x > 1:
    print('now')
elif x < 20:
    print('wow')
print('cow')
```

How many lines does this print?

```
x = 5
if x == 8:
    print('how')
if x > 1:
    print('now')
if x < 20:
    print('wow')
print('cow')
```

Pre-Lecture
Working with Strings and Lists

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Strings: Numbering the 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

```
  0 1 2 3 4
 'Perry'
```

  - from right to left, starting from -1

```
 -5-4-3-2-1
 'Perry'
```

  - `'P'` has an index of 0 or -5
  - `'y'` has an index of …

String Operations

- Indexing: *string*[*index*]

```
>>> name = 'Picobot'
>>> name[1]
'i'
>>> name[-3]
```

____

- Slicing (extracting a substring): *string*[*start*:*end*]

from this index (*start*); up to but ***not including*** this index (*end*)

```
>>> name[0:2]
'Pi'
>>> name[1:-1]
'icobo'
>>> name[1:]
'icobot'
>>> name[:4]
'Pico'
```

String Operations (cont.)

- Concatenation: *string1* + *string2*

```
>>> word = 'program'
>>> plural = word + 's'
>>> plural
'programs'
```

- Duplication: *string* * *num_copies*

```
>>> 'ho!' * 3
'ho!ho!ho!'
```

- Determining the length: `len`(*string*)

```
>>> name = 'Perry'
>>> len(name)
```

____

```
>>> len('')    # an empty string - no characters!
0
```

List Ops == String Ops (more or less)

```
                 0       1         2          3
>>> majors = ['CS', 'math', 'english', 'psych']

>>> majors[2]
'english'

>>> majors[1:3]

______________________

>>> len(majors)
4

>>> majors + ['physics']
['CS', 'math', 'english', 'psych', 'physics']

>>>
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Terriers'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
```

Working with Strings and Lists

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Practice with Indexing and Slicing

- Given the following assignment:
```
>>> s = 'computer'
```
- What is the value of each of the following?
```
>>> s[1]

>>> s[-1]

>>> s[2:4]

>>> s[:3]

>>> s[5:]

>>> s[-4:-1]
```

What is the value of s after the following code runs?

```
s = 'def'

s = ('a' * 3) + s

s = s[2:-2]
```

What is the output of the following program?

```
mylist = [5, 4, [3, 2, 1]]
print(mylist[1], mylist[1:2])
```

- indexing a list produces a single element – which may or may not be a list
- slicing a list *always* produces a list!

How could you fill in the blank to produce `[105, 111]`?

```
intro_cs = [101, 103, 105, 108, 109, 111]

dgs_courses = ____________________
```

A. `intro_cs[2:3] + intro_cs[-1:]`

B. `intro_cs[-4] + intro_cs[5]`

C. `intro_cs[-4:3] + intro_cs[5:6]`

D. more than one of the above

E. none of the above

How could you fill in the blank to produce `[105, 111]`?

```
intro_cs = [101, 103, 105, 108, 109, 111]

dgs_courses = ____________________
```

What about this?

`intro_cs[-4] + intro_cs[-1:]`

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
```

Practice Problem: 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).
```

- To convert from centimeter to inches, divide by 2.54.

- Optional extra: If the entered height is not positive, print an error message and end the program.

One Possible Solution

```
cm = int(input('Enter your height in cm: '))
inches = cm / 2.54
inches = round(inches)

feet = inches // 12
remaining = inches % 12

print('You are', inches, 'inches tall (' + str(feet),
  'feet,', remaining, 'inches).')
```

A Solution That Handles Inputs Less Than 0

```
cm = int(input('Enter your height in cm: '))
if cm < 0:
    print('Heights must be positive')
else:
    inches = cm / 2.54
    inches = round(inches)

    feet = inches // 12
    remaining = inches % 12

    print('You are', inches, 'inches tall (' + str(feet),
     'feet,', remaining, 'inches).')
```

Extra Practice: Fill in the blank to make the code print `compute!`

```
subject = 'computer science!'
verb = _______________
print(verb)
```

A. `subject[:7] + subject[-1]`

B. `subject[:7] + subject[:-1]`

C. `subject[:8] + subject[-1]`

D. `subject[:8] + subject[:-1]`

E. none of these

Pre-Lecture
Using Objects;
Splitting and Joining Strings

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

What Is An Object?

- An object is a construct that groups together:
  - one or more data values (the object's *attributes*)
  - one or more functions that operate on those data values (known as the object's *methods*)

Strings Are Objects

- In Python, a string is an object.
  - ***attributes:***
    - the characters in the string
    - the length of the string
  - ***methods:*** functions inside the string that we can use to operate on the string

string object for `'hello'`

```
contents  'h' 'e' 'l' 'l' 'o'
  length  5

upper()     replace()
lower()     split()
find()      ...
count()
```

string object for `'bye'`

```
contents  'b' 'y' 'e'
  length  3

upper()     replace()
lower()     split()
find()      ...
count()
```

Calling a Method

- An object's methods are inside the object, so we use *dot notation* to call them.
- Example:

```
name = 'Perry'
allcaps = name.upper()
```

the dot

the object's variable

the method name

string object for `'Perry'`

```
contents  'P' 'e' 'r' 'r' 'y'
  length  5

upper()     replace()
lower()     split()
find()      ...
count()
```

- Because a method is inside the object, it is able to access the object's attributes.

String Methods (partial list)

- `s.upper()`: return a copy of `s` with all uppercase characters
- `s.lower()`: return a copy of `s` with all lowercase 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.

```
>>> name = '       Martin Luther   King   '
>>> name.split()
['Martin', 'Luther', 'King']
>>> components = name.split()
>>> components[0]
'Martin'
```

- By default, it uses *whitespace characters* (spaces, tabs, and newlines) to determine where the splits should occur.
- You can specify a different separator:

```
>>> date = '10/21/2000'
>>> date.split('/')
```

____

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:

```
>>> components = ['Martin', 'Luther', 'King']
>>> ' '.join(components)
'Martin Luther King'
>>> '/'.join(['10', '21', '2000'])

________________________
```

Using Objects;
Splitting and Joining Strings

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Examples of Using String Methods

```
>>> chant = 'We are the Terriers!'
>>> chant.upper()

>>> chant.lower()

>>> chant.replace('e', 'o')

>>> chant.replace('e', 'o').upper()

>>> chant
```

What is the output of this program?

```
s = 'Programming'
s = s.lower()
s.upper()
print(s.split('r'))
```

A. `['P', 'og', 'amming']`

B. `['p', 'og', 'amming']`

C. `['P', 'OG', 'AMMING']`

D. `['PR', 'OGR', 'AMMING']`

E. `['pr', 'ogr', 'amming']`

How could I print the string `'PROGRAMMING'`?

```
components = ['p', 'og', 'amming']
print(____________________________)
```

A. `components.join('r').upper()`

B. `join(components, 'r').upper()`

C. `'r'.join(components).upper()`

D. `components.upper().join('r')`

E. `'r'.upper().join(components)`

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.
```

Pre-Lecture
Accessing a Database from Python

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Initial Steps

- Put the database file in the same folder as the Python program that will access it.
- Import the necessary Python module:

```
import sqlite3
```

- Connect to the database as follows:

  `db = sqlite3.connect(`*name of database file*`)`

  - example:

    ```
    db = sqlite3.connect('movie.sqlite')
    ```

  - `sqlite3.connect` returns a *database handle*
    - an object that provides a connection to the database
    - we assign it to a variable

Performing a Query

- Given a database handle, we perform a query by:
  - using a method inside the database handle to create another object known as a *cursor*:

    ```
    cursor = db.cursor()
    ```

  - using a method inside the cursor object to execute the query command:

    `cursor.execute(`*command*`)`

- Example:

```
cursor = db.cursor()
cursor.execute('''SELECT name, rating
                  FROM Movie
                  WHERE year = 2012;''')
```

Obtaining the Results of a Query

- Executing a query does ***not*** automatically display the results.
- Rather, we use the cursor to access the results.
- One option: the `fetchone` method inside the cursor:

  ```
  row = cursor.fetchone()
  ```

  - gets one row at a time from the results
  - if there are no rows left to get, this function returns `None`

Format of a Row

- Each row of the query results looks something like this:

```
('Iron Man 3', 'PG-13')
```

- This is known as a *tuple* in Python.
  - another type of sequence
  - similar to a list, but it's *not* mutable
  - can access its components using indexing

```
row = ('Iron Man 3', 'PG-13')
```

row[0] row[1]

Using a `for` Loop to Display Query Results

- If we knew how many rows were in the result,
  we could do something like this:

```
for i in range(4):
    row = cursor.fetchone()
    print(row[0], '(' + row[1] + ')')
```

- However, we typically *don't* know how many rows there are!

- To solve this problem, Python lets us do this!

```
for row in cursor:
    print(row[0], '(' + row[1] + ')')
```

  - the loop variable (`row`) takes on one row at a time
    from the results of the query executed by the cursor
  - we don't need to use the `fetchone` method
  - the loop will perform as many repetitions
    as there are rows in the result!

Using a `for` Loop to Display Query Results

- General template:

```
for row in cursor:
    # code to process one row goes here
```

- Another example:

```
# preliminary steps are taken as before...

cursor = db.cursor()
cursor.execute('''SELECT name, rating, runtime
                  FROM Movie
                  WHERE year = 2013;''')
for row in cursor:
    rating = '(' + row[1] + ')'
    if row[2] > 120:
        print(row[0], rating, '**over 2 hrs**')
    else:
        print(row[0], rating)
```

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)
```

- Have the user enter the year *as a string*:

```
target_year = input('year to search for? ')
```

Parameterized Queries

- To handle user input, we use a *parameterized query*.
  - example:

```
command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''
```

    - `?` is a placeholder

- We execute the parameterized query as follows:

```
cursor.execute(command, [target_year])
```

command string with placeholders

a list containing the values that should replace the placeholders (the parameters)

- The `execute` function replaces the placeholders with the specified values.

Accessing a Database from Python

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Modules in Python

- A *module* in Python is a collection of functions.
  - can also include other things like constants
- To use a function from a module, we:
  - *import* the module
  - prepend the name of the module before the function name
- Example: the `math` module, which includes:
  - `sqrt(n)`: computes the square root of a number
  - trigonometric functions: `sin(n)`, `cos(n)`, `tan(n)`
  - constants: `pi`, `e`

```
>>> math.sqrt(25)
…NameError: name 'math' is not defined
>>> import math
>>> math.sqrt(25)
5.0
```

Recall: Connecting to a Database

- After importing `sqlite3`:

  `db = sqlite3.connect(`*name of database file*`)`

  - example:

    ```
    db = sqlite3.connect('movie.sqlite')
    ```

  - `sqlite3.connect` returns a *database handle*
    - an object that provides a connection to the database
    - we assign it to a variable

Important!

- Double-check to ensure that the database file is in the same folder as your Python program.
- If you forget to put the database file in the right place, you'll get a misleading error message like this:

  ```
  sqlite3.OperationalError: no such table: Movie
  ```

  - after the error, there will be a database file in the folder!
- When you try to connect to a database file that isn't there, SQLite creates an empty database file for you!
  - assumes that you are using the program to create a brand new database
- If you get this error:
  - delete the version of the database file in the folder
  - replace it with the actual database file

How many lines would this program output?

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

How many lines would this program output?

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')

row = cursor.fetchone()
print(row[0], row[1])
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

How could we get all of the rows of results?

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')

row = cursor.fetchone()
print(row[0], row[1])
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

One option...

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')

row = cursor.fetchone()
print(row[0], row[1])
row = cursor.fetchone()
print(row[0], row[1])
row = cursor.fetchone()
print(row[0], row[1])
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

A better option...how would you fill in the blank?

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')

for i in ___________:
    row = cursor.fetchone()
    print(row[0], row[1])
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

An even better way!

```
import sqlite3

db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT student_id, course_name
                  FROM Enrolled
                  WHERE credit_status = 'ugrad';''')

for row in cursor:
    print(row[0], row[1])
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

Accessing the Movie Database

```
import sqlite3

# connect to the database and create a cursor
db = sqlite3.connect('movie.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT name, rating
                  FROM Movie
                  WHERE year = 2013;''')

# obtain and print all results!
for row in cursor:
    print(row[0], '(' + row[1] + ')')
```

output:

```
Iron Man 3 (PG-13)
Frozen (PG)
Despicable Me 2 (PG)
Man of Steel (PG-13)
Gravity (PG-13)
Monsters University (G)
```

...plus all of the other 2013 movies!

Concluding a Database Session

- At the end of a program that accesses a database, you should use the database handle (`db`) to:
  - commit any changes that were made:
    `db.commit()`
  - close the connection to the database
    `db.close()`
- This isn't crucial if you're only executing `SELECT` commands.
- If you execute commands that make changes and don't take these steps, the changes may not take effect!
- We'll take these steps even when we're not making changes!

Revised Program

```
import sqlite3

# connect to the database and create a cursor
db = sqlite3.connect('movie.sqlite')
cursor = db.cursor()

# execute the query
cursor.execute('''SELECT name, rating
                  FROM Movie
                  WHERE year = 2013;''')

# obtain and print all results!
for row in cursor:
    print(row[0], '(' + row[1] + ')')

# conclude the database session
db.commit()
db.close()
```

output:

```
Iron Man 3 (PG-13)
Frozen (PG)
Despicable Me 2 (PG)
Man of Steel (PG-13)
Gravity (PG-13)
Monsters University (G)
```

...plus all of the other 2013 movies!

Recall: Parameterized Queries

- To handle user input, we use a *parameterized query*.
  - example:

```
command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''
```

    - ? is a placeholder

- We execute the parameterized query as follows:

```
cursor.execute(command, [target_year])
```

command string with placeholders

a list containing strings that should replace the placeholders (the parameters)

- The `execute` function replaces the placeholders with the specified values.

Recall: Example Program: Final Version

```
import sqlite3

# connect to the database and create a cursor
db = sqlite3.connect('movie.sqlite')
cursor = db.cursor()

# get the year from the user as a string
target_year = input('year to search for? ')

# execute the parameterized query
command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''
cursor.execute(command, [target_year])

# obtain and print all results!
for row in cursor:
    print(row[0], '(' + row[1] + ')')

# conclude the database session
db.commit()
db.close()
```

The Wrong Way to Incorporate User Input

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

```
target_year = input('year to search for? ')

command = '''SELECT name, rating
             FROM Movie
             WHERE year = ''' + target_year + ';'

cursor.execute(command)   # no parameters needed!

for row in cursor:
    print(row[0], '(' + row[1] + ')')
```

- Problem: this approach can lead to serious security breaches!
  - known as *SQL injections*

SQL Injection Vulnerability

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

- After showing the movie results, the program will then display the entire first two columns of `Secret`!

SQL Injection Vulnerability (cont.)

- Here's another problematic input!

```
year to search for? 1976; DROP TABLE Secret
```

- Parameterized queries eliminate this vulnerability.

```
command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''

cursor.execute(command, [target_year])
```

- When replacing a placeholder with its specified value, `execute` takes whatever steps are needed to ensure that the value is treated as a single literal value.
  - example: if the user enters `1976; SELECT * FROM Secret` the resulting command is:

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

notice the quotes!

Handling Queries with No Results

- What if the user enters a year with no movies in the database?
- 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 rows that the program processes:

```
cursor.execute(command, [target_year])

count = 0
for row in cursor:
    print(row[0], '(' + row[1] + ')')
    count = count + 1

if count == 0:
    print('There are no movies from', target_year)
```

Review: Strings and Lists; Accessing a Database

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

What is the output of the following program?

```
s = 'hello world'
values = [2, 3, 4]
print(s[1:2], values[-2:-1])
```

A. `el [3, 4]`

B. `el 3 4`

C. `e [3]`

D. `e 3`

E. none of the above

What is the output of the following program?

```
s = 'hello world'
x = s.split()
print(x[1])
```

A. `h`

B. `e`

C. `hello`

D. `world`

E. none of the above

What about this program?

```
s = 'hello world'
x = s.split('o')
print(x[1])
```

How would you fill in the blank?

```
import sqlite3
db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# get the credit status from the user
target_status = input('which credit status do you want? ')

# execute the query
command = '''SELECT student_id, course_name
             FROM Enrolled
             WHERE credit_status = ?;'''

cursor.execute(________________)
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

A. `command`

B. `command, target_status`

C. `command, [target_status]`

D. none of the above

What about this blank?

```
import sqlite3
db = sqlite3.connect('university.sqlite')
cursor = db.cursor()

# get the credit status from the user
target_status = input('which credit status do you want? ')

# execute the query
command = '''SELECT student_id, course_name
             FROM Enrolled
             WHERE credit_status = ?;'''

cursor.execute(...)   # filled in from previous slide
for row in cursor:
    print(_____________________)
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	ugrad

A. `row[0], row[1]`

B. `row.fetchone()`

C. `cursor[0], cursor[1]`

D. none of the above

Parameterized Queries: Another Example

- Here's a query that takes three parameters:

```
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:

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

cursor.execute(command, [dir_name, start, end])
                    ['Joel Coen','2000','2012'])
```

The Full Program (getMoviesByDirector.py)

```
import sqlite3

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

dir_name = 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, [dir_name, start, end])

for row in cursor:
    print(row[0], row[1])

db.commit()
db.close()
```

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:

```
...
cursor.execute(command, [dir_name, start, end])

count = 0
for row in cursor:
    print(row[0], row[1])
    count = count + 1

# print a message if there were no results
# what should go here?
```

Pre-Lecture
Working with Text Files; File Writing

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Opening a Text File

- Before working with a file, we need to *open* a connection to it.
- Example:

```
outfile = open('example.txt', 'w')
```

  where:
    - `'example.txt'` is the name of the file we want to write to
    - `'w'` indicates that we want to write to the file
      (to read from the file, we would use `'r'` instead)
- Doing so creates an object known as a *file handle*.
    - we use the file handle to perform operations on the file

Closing a File

- Here's our previous example:

```
outfile = open('example.txt', 'w')
```

- When we're done working with the file, we close its handle:

```
outfile.close()
```

- **Important:** Text that you write to a 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, the current contents will be erased!
- 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)
```

More Practice with Database Access; Writing to a Text File

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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

Sample Runs of the Program

```
name of database file: movie.sqlite
name of actor: Dave Sullivan
Dave Sullivan is not in the database.

name of database file: movie.sqlite
name of actor: Meryl Streep
Meryl Streep was born on 1949-06-22 in Summit, New
Jersey, USA.
```

A Front-End for Our Movie Database

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

# Get the actor's name.
__________________________________________

# Execute the command, get the result, and print it.
command = '''

                            '''
cursor.execute(_____________________)

for ______________________________:
    print(___________________________________________)
```

Converting the Date Format

- Here's the current output:

```
name of database file: movie.sqlite
name of actor: Meryl Streep
Meryl Streep was born on 1949-06-22 in Summit, New
Jersey, USA.
```

- Let's say that we want to change the format of the date of birth:

```
name of database file: movie.sqlite
name of actor: Meryl Streep
Meryl Streep was born on 06/22/1949 in Summit, New
Jersey, USA.
```

- What string methods would be useful here?

How would you fill in the blanks?

```
...
command = '''SELECT dob, pob
             FROM Person
             WHERE name = ?;'''
cursor.execute(command, [name])
count = 0
for row in cursor:
    comps = ________________
    dob = ______________________
    print(name, 'was born on', dob, 'in', row[1] + '.')
    count = count + 1
...
```

Example:
We want to go
from `'1949-06-22'`
to `'06/22/1949'`

	first blank	second blank
A.	`row[0].split()`	`'/'.join(comps)`
B.	`row[0].split('-')`	`'/'.join(comps)`
C.	`row[0].split()`	`'/'.join([comps[1], comps[2], comps[0]])`
D.	`row[0].split('-')`	`'/'.join([comps[1], comps[2], comps[0]])`

Revised Front-End for Our Movie Database

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

# Get the actor's name.
name = input('name of actor: ')

# Execute the command, get the result, and print it.
command = 'SELECT dob, pob FROM Person WHERE name = ?;'
cursor.execute(command, [name])

count = 0
for row in cursor:
    comps = row[0].split('-')
    dob = '/'.join([comps[1], comps[2], comps[0]])
    print(name, 'was born on', dob, 'in', row[1] + '.')
    count = count + 1
if count == 0:
    print(name, 'is not in the database.')

db.commit()
db.close()
```

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:
  `'w'` if we want to write to the file
  `'r'` if we want to read from 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 folder
- Python will open/create the file in the same directory in which the program is stored.

Recall: 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, the current contents will be erased!
- 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()

target_year = input('year to search for? ')
command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''
cursor.execute(command, [target_year])

count = 0
for row in cursor:
    print(row[0], '(' + row[1] + ')')
    count = count + 1

if count == 0:
    print('there are no movies from', target_year)
db.commit()
db.close()
```

On the slide below, modify this program so that it writes the results to a file with a name that is based on the target year (e.g., `2010.txt`)

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

target_year = input('year to search for? ')


command = '''SELECT name, rating
             FROM Movie
             WHERE year = ?;'''
cursor.execute(command, [target_year])

count = 0
for row in cursor:
    print(row[0], '(' + row[1] + ')')
    count = count + 1

if count == 0:
    print('there are no movies from', target_year)



db.commit()
db.close()
```

Pre-Lecture
Reading Text Files

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Text Files

- A text file can be thought of as one long string.
- The end of each line is stored as a newline character (`'\n'`).
- Example: the following three-line text file

```
Don't forget!

Test your code fully!
```

is equivalent to the following string:

```
'Don't forget!\n\nTest your code fully!\n'
```

Opening a Text File

- Before we can read from a text file, we need to *open* a connection to the file.
- Example:

```
f = open('reminder.txt', 'r')
```

  where:
  - `'reminder.txt'` is the name of the file we want to read
  - `'r'` indicates that we want to read from the file
- Doing so creates an object known as a *file handle*.
  - we use the file handle to perform operations on the file

Processing a File Using Methods

- A file handle is an object.
- We can use its methods to process a file.

reminder.txt

```
Don't forget!

Test your code fully!
```

```
>>> f = open('reminder.txt', 'r')
>>> f.readline()
"Don't forget!\n"
>>> f.readline()
____________
>>> f.readline()
'Test your code fully!\n'
>>> f.readline()
''
>>> f = open('reminder.txt', 'r')   # start over at top
>>> f.read()
"Don't forget!\n\nTest your code fully!\n"
```

Processing a File Using a `for` Loop

- We often want to read and process a file one line at a time.
- We could use `readline()` inside a loop, but...
  we don't know how many lines there are!
- Python makes it easy!

```
for line in file-handle:
    # code to process line goes here
```

  - reads one line at a time and assigns it to `line`
  - continues looping until there are no lines left

Processing a CSV File

- CSV = comma-separated values
  - each line is one *record*
  - the *fields* in a given record are separated by commas

`courses.txt`

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

Processing a CSV File

```
file = open('courses.txt')

count = 0
for line in file:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

line	fields	output	count
			0
'CS,111,MWF 10-11\n' 'CS,111,MWF 10-11'	['CS','111','MWF 10-11']	CS 111	1
'MA,123,TR 3-5\n' 'MA,123,TR 3-5'	['MA','123','TR 3-5']	*none*	_____
'CS,105,MWF 1-2\n' 'CS,105,MWF 1-2'	['CS','105','MWF 1-2']	CS 105	_____

...

Reading Text Files

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Text Files

- A text file can be thought of as one long string.
- The end of each line is stored as a newline character (`'\n'`).
- Example: the following three-line text file

```
Don't forget!

Test your code fully!
```

  is equivalent to the following string:

```
'Don't forget!\n\nTest your code fully!\n'
```

- Although `'\n'` has two characters when it we type it,
  it is stored as only one character:

```
>>> len('\n')
1
```

Recall: Opening a Text File

- Syntax:

  file-handle `= open(`*filename*`,` *mode*`)`

  where *file-handle* is a variable for the file handle
  filename is a string
  mode is:
  `'w'` if we want to write to the file
  `'r'` if we want to read from the file

Reading from a File

- We use the `'r'` mode when opening the file:

```
infile = open('foo.txt', 'r')
```

- If there is a file with the specified name ***in the same folder as your program***, the file handle will be connected to it.
  - if not, you will get an error

Recall: Processing a File Using Methods

- A file handle is an object.
- We can use its methods to process a file.

reminder.txt

```
Don't forget!

Test your code fully!
```

```
>>> f = open('reminder.txt', 'r')
>>> f.readline()
"Don't forget!\n"
>>> f.readline()
'\n'
>>> f.readline()
'Test your code fully!\n'
>>> f.readline()
''
>>> f = open('reminder.txt', 'r')    # start over at top
>>> f.read()
"Don't forget!\n\nTest your code fully!\n"
```

Processing a File Using a `for` Loop

- We often want to read and process a file one line at a time.
- We could use readline() inside a loop, but...
  - what's the problem we would face?
- Python makes it easy!

```
for line in file-handle:
    # code to process line goes here
```

  - reads one line at a time and assigns it to `line`
  - continues looping until there are no lines left

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:

```
filename = input('filename? ')
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]`)

Processing a Text File

```
filename = input('filename? ')
infile = open(filename, 'r')

for line in infile:
    if line != '\n':
        print(line[:-1])

infile.close()
```

info.txt

```
Reading from a file
* open the file

* use a loop

Closing the file
* use file.close()

* don't forget!
```

line	is the line printed?
'Reading from a file\n'	yes
'* open the file\n'	yes
'\n'	no
'* use a loop\n'	yes
'\n'	no
'Closing the file\n'	yes
'* use file.close()\n'	yes
'\n'	
'* don't forget\n'	

output:

```
filename? info.txt
Reading from a file
* open the file
* use a loop
Closing the file
* use file.close()
```

Processing a Text File

```
filename = input('filename? ')
infile = open(filename, 'r')

for line in infile:
    if line != '\n':
        print(line[:-1])

infile.close()
```

info.txt

```
Reading from a file
* open the file

* use a loop

Closing the file
* use file.close()

* don't forget!
```

desired output:

```
filename? info.txt
Reading from a file
Closing the file
```

What if we want the output to omit *both* blank lines *and* lines that begin with *

Fill in the blank to omit *both* blank lines *and* ones that begin with a *

```
filename = input('filename? ')
infile = open(filename, 'r')

for line in infile:
    if ___________________:
        print(line[:-1])

infile.close()
```

info.txt

```
Reading from a file
* open the file

* use a loop

Closing the file
* use file.close()

* don't forget!
```

desired output:

```
filename? info.txt
Reading from a file
Closing the file
```

A. `line != '\n' or '*'`

B. `line != '\n' or line != '*'`

C. `line != '\n' or line[0] != '*'`

D. `line != '\n' and line[0] != '*'`

E. none of the above

How Should We Fill in the Blank?

```
infile = open(________________________)

count = 0
for line in infile:
    ...
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

A. `'courses.txt', 'r'`

B. `'courses.txt', 'w'`

C. `'r', 'courses.txt'`

D. `'w', 'courses.txt'`

How Should We Fill in the Blanks?

```
infile = open('courses.txt', 'r')

count = 0
for line in infile:
    line = line[:-1]
    fields = ____________________
    if ___________ == 'CS':
        print(fields[0],fields[1])
        count += 1
...
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

	first blank	second blank
A.	`infile.split()`	`fields`
B.	`line.split()`	`fields[0]`
C.	`infile.split(',')`	`fields`
D.	`line.split(',')`	`fields[0]`
E.	none of the above	

Processing a CSV File

```
infile = open('courses.txt', 'r')

count = 0
for line in infile:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1
...
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

line	fields	output	count
			0
'CS,111,MWF 10-11\n'			
'CS,111,MWF 10-11'	['CS','111','MWF 10-11']	CS 111	1
'MA,123,TR 3-5\n'			
'MA,123,TR 3-5'	['MA','123','TR 3-5']	*none*	1
'CS,105,MWF 1-2\n'			
'CS,105,MWF 1-2'	['CS','105','MWF 1-2']	CS 105	2
...			

After the loop completes…

```
infile = open('courses.txt', 'r')

count = 0
for line in infile:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1

if count == 0:
    print('There are no CS courses in the file.')

infile.close()
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

How could we write the results to a file?

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

```
infile = open('courses.txt', 'r')

outfile =
count = 0
for line in infile:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1], ____________________)
        count += 1

if count == 0:
    print('There are no CS courses in the file.')
else:
    print('There are', count, 'CS courses in the file.')
    print('See cs_courses.txt for details.')

infile.close()

___________________
```

File-Reading Revisited

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Processing a CSV File

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

- CSV = comma-separated values
  - each line is one *record*
  - the *fields* in a given record are separated by commas
- Let's say that we want to print the names of all CS courses:

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

→

screen/console:

```
CS 111
CS 105
...
```

How Should We Fill in the Blank?

```
file = open('courses.txt', 'r')

count = 0
for line in file:
    line = ______________
    ...
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

A. `line[-1]`

B. `line[:-1]`

C. `file[-1]`

D. `file[:-1]`

E. none of the above

How Should We Fill in the Blanks?

```
file = open('courses.txt', 'r')

count = 0
for line in file:
    line = line[:-1]
    fields = ___________________
    if ___________ == 'CS':
        print(fields[0],fields[1])
        count += 1
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

screen/console:

```
CS 111
CS 105
...
```

	first blank	second blank
A.	`file.split()`	`fields`
B.	`line.split()`	`fields[0]`
C.	`file.split(',')`	`fields`
D.	`line.split(',')`	`fields[0]`
E.	none of the above	

Extracting Relevant Data from a File

- Assume that the results of a track 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 the results for a particular school, printing them to the screen.
  - sample output for BU (note: spaces, not commas):

```
Mike Mercury mile 4:50:00
Len Lightning half-mile 2:15:00
```

  - do *not* print the school name
  - print an error message if the school is not found

Extracting Relevant Data from a File

```
infilename = input('name of input file: ')
school = input('extract records of which school? ')

infile = open(infilename, 'r')

count = 0
for line in infile:
    line = line[:-1]
    fields = line.split(',')
    if fields[1] == school:
        count = count + 1
        print(fields[0], fields[2], fields[3])

if count == 0:
    print('No results for', school, 'in', infilename)

infile.close()
```

Version 2: Write the Matching Results to a CSV File

- initial file:

```
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
```

- results file for Boston University (note the commas):

```
Mike Mercury,mile,4:50:00
Len Lightning,half-mile,2:15:00
```

What changes do we need to make?

```
infilename = input('name of input file: ')
outfilename = input('name of output file: ')
school = input('extract records of which school? ')
infile = open(infilename, 'r')

outfile = _______________________________

count = 0
for line in infile:
    line = line[:-1]
    fields = line.split(',')
    if fields[1] == school:
        count = count + 1
        print(fields[0]+','+fields[2]+','+fields[3],
              ______________________)
if count == 0:
    print('No results for', school, 'in', infilename)
else:
    print(count, 'results written to', outfilename)
infile.close()
outfile.close()
```

Version 3: Handle Input Files With a Header

- CSV files often include a header line at the top:

```
athlete,school,event,time
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
```

- We typically want to either:
  - skip over the header
  - treat it differently than the other lines

Recall: Processing a File Using Methods

- A file handle is an object.

- We can use its methods to process a file.

reminder.txt

```
Don't forget!

Test your code fully!
```

```
>>> f = open('reminder.txt', 'r')
>>> f.readline()        # reads one line from the file
"Don't forget!\n"
```

Handling Input Files With a Header (cont.)

```
athlete,school,event,time
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
```

- To handle the header, we read it in using `readline()` before we start the file-processing loop:

```
f = open(filename, 'r')
header = f.readline()
# optional: do something with the header here

# the loop still processes the remaining lines
for line in f:
    line = line[:-1]
    ...
```

Pre-Lecture
Data Mining Fundamentals

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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
  - looking for products that customers tend to purchase together

Machine Learning

- In data mining, we apply an algorithm that "learns" something about the data.
- These algorithms are *machine-learning algorithms.*
- We're ultimately going to consider three different types of machine learning:
  - classification learning
  - association learning
  - numeric estimation

Classification Learning

- Classification learning involves learning how to classify objects/entities on the basis of their characteristics.
  - example: learning to determine whether a customer is likely to buy a computer in the next year (Yes/No).
- We give the algorithm a set of *training examples* that have already been classified.

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no
…					

- The algorithm produces a *model* that can be used to classify other examples.

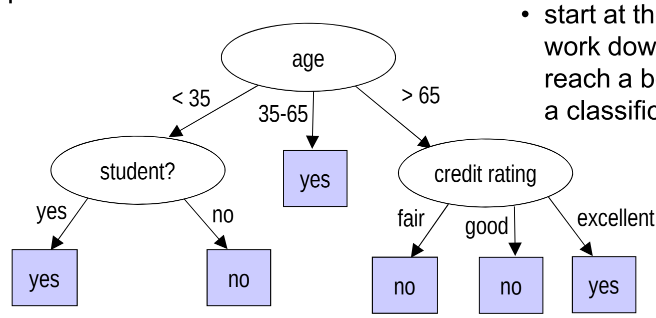
Classification Learning

- Training examples:

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no

…

- One possible model: a *decision tree*

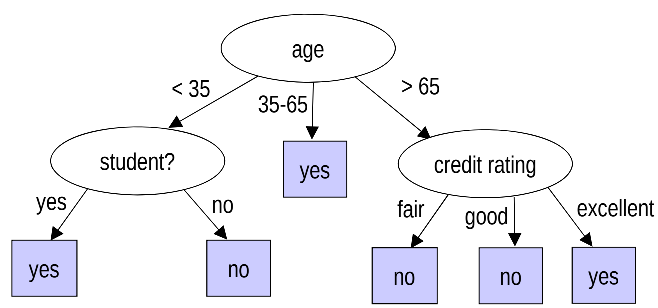


- start at the top and work down until you reach a box with a classification

Classification Learning

- Once the algorithm learns the model, we can use the model to classify new examples:

id	gender	age	student?	credit rating	buy computer?
333	female	68	no	fair	
673	male	23	yes	good	



Some Terminology

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no
…					

- Each row in the 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

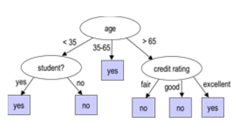
input attributes ⇨ model ⇨ output attribute

Some Terminology

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no
…					

- Each row in the 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

age
student?
credit rating
…
⇨



⇨ buy computer?

Nominal vs. Numeric

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no
…					

- *Nominal* attributes:
  - have values that are "names" of categories
- *Numeric* attributes:
  - have values that are numbers
  - it makes sense to *compare* their values using < and >
    example: we could base predictions on whether age < 35
- What about id?

Classification Learning

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no
…					

- We have a single output attribute whose value we want to determine/predict.
- That output attribute is ***nominal***.
- The input attributes can be either nominal or numeric.

Numeric Estimation

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	0.75
456	female	38	no	good	0.90
872	male	65	no	fair	0.23
222	female	28	yes	excellent	0.68
111	female	20	no	good	0.37
…					

- We have a single output attribute whose value we want to determine/predict.
- That output attribute is ***numeric***.
- The input attributes can be either nominal or numeric.

Data Mining Fundamentals, Part I

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)
- Witten & Frank, *Data Mining: Practical Machine Learning Tools and Techniques, 2nd edition* (Morgan Kaufmann, 2005)

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?

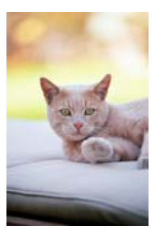
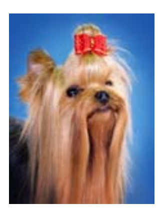
beer	cereal	diapers
milk	soft drinks	toilet paper

    - 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) using a computer program
  - the program "mines" the data
    - "sifting through" it to try find something useful/valuable

Data Mining vs. Data Query

- Database queries in SQL are *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.
  - "hidden knowledge"

Machine Learning

- In data mining, we apply an algorithm that "learns" something about the data.
  - a *machine-learning algorithm*
- We're ultimately going to consider three different types of machine learning:
  - classification learning
  - association learning
  - numeric estimation

Classification Learning

- Involves learning how to classify objects/entities on the basis of their characteristics.
  - ex: is a credit-card purchase 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.
  - can take different forms: rules, a decision tree, etc.

Example: Medical Diagnosis

- Given a set of symptoms, we want to diagnose a patient.
  - possible diagnoses: cold, allergy, strep throat

- Sample training data (table 1-1 of Roiger & Geatz):

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- Can you see any patterns that would help you diagnose patients with one or more of these symptoms?

Example: Medical Diagnosis (cont.)

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

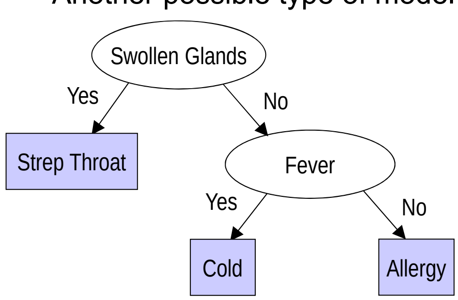
- One possible model that could be used for classifying other patients is a set of rules like the following:

```
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
```


Example: Medical Diagnosis (cont.)

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- Another possible type of model is known as a *decision tree*:

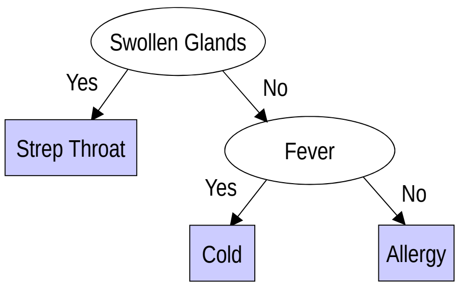


  - start at the top and work down until you reach a box containing a classification

What diagnosis would the tree give for patient 11?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold
11	**No**	**No**	**No**	**No**	**Yes**	**???**

- 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

Some Terminology

- In a collection of training data:
  - each row 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 ⇨ model ⇨ output attribute

- In our example:

fever
swollen glands
headache
…
⇨ rules *or* tree *or…* ⇨ 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 are single numbers
  - **it makes sense to *compare* their values using < and >**
  - example: Body Temp
    - each value is a single number like 98.0 or 101.5
    - it could make sense to base our predictions on comparisons like Body Temp > 98.6

Types of Attributes (cont.)

- What about this one?

attribute	possible values
Product Type	{0, 1, 2, 3}

Types of Attributes (cont.)

- What about this one?

attribute	possible values
Product Type	{0, 1, 2, 3}

it is nominal!

- the numbers are serving as *names* of categories
- comparisons like product type > 2 don't provide useful info

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
…
⇨ model ⇨ probability of response
(a number between 0 and 1)

- The model often takes the form of an equation.
  probability_of_reply = 0.424*attr1* – 0.072*attr2* + …
  where *attr1*, *attr2*, … are attributes

- Linear regression is a form of numeric estimation.

Association Learning

- Involves looking for relationships between sets of attributes in the training examples.
  - produces a set of rules
  - for example:

```
if Congestion = Yes
then Headache = Yes

if Sore Throat = Yes and Swollen Glands = No
then Congestion = Yes and Fever = No
```

- It does *not* focus on predicting a particular attribute.
  - unlike classification learning and numeric estimation
  - no distinction between input and output attributes

Association Learning (cont.)

- One form of association learning is *market-basket analysis*.
  - 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?

What approach is needed?

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

- Problem **1**: We want to determine the average trades/month for each transaction method.
- The best approach is:
  - A. database queries
  - B. data mining using classification learning
  - C. data mining using numeric estimation
  - D. data mining using association learning

What approach is needed? (cont.)

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

- Problem **2**: If we don't know a customer's favorite recreation, what other factors can we used to predict it?
- The best approach is:
  - A. database queries
  - B. data mining using classification learning
  - C. data mining using numeric estimation
  - D. data mining using association learning

What approach is needed? (cont.)

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

- Problem **2**: If we don't know a customer's **favorite recreation**, what other factors can we used to **predict it**?

output is nominal , so classification

predicting a single attribute, so data mining using either classification or estimation

- The best approach is:
  - A. database queries
  - B. **data mining using classification learning**
  - C. data mining using numeric estimation
  - D. data mining using association learning

Pre-Lecture
Evaluating a Model Learned in Data Mining

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

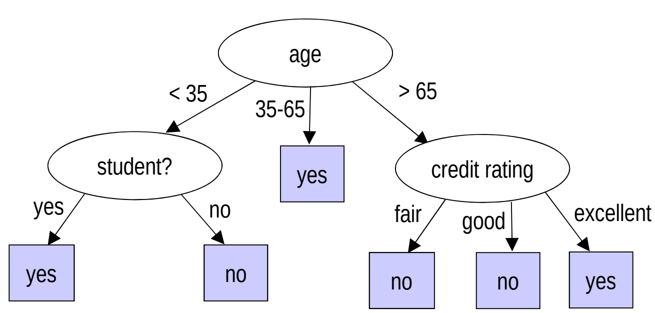
Recall: Classification Learning

- Training examples:

id	gender	age	student?	credit rating	buy computer?
123	male	15	yes	fair	yes
456	female	38	no	good	yes
872	male	65	no	fair	no
222	female	28	yes	excellent	yes
111	female	20	no	good	no

…

- One possible model: a *decision tree*



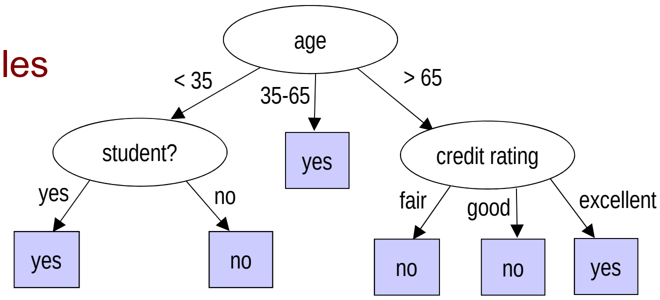
Evaluating the Model

- For most non-trivial, real-world data sets, no learned model is likely to work perfectly on all possible examples.
- 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*

Test Examples

- To see how well a model generalizes, we typically withhold some of the available data as *test examples*.
  - these examples are *not* used to train the model
- Let's assume we have data for 100 customers.
  - all of the data is already classified
  - use 90 examples to learn the model (the training data)
  - use 10 examples to test the model (the test data)

Using the Test Examples



					buy computer?	
id	gender	age	student?	credit rat.	actual	predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	______
747	female	47	no	excellent	no	______

Summarizing the Results

					buy computer?	
id	gender	age	student?	credit rat.	actual	predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes
747	female	47	no	excellent	no	yes

- accuracy of the model = **6**/10 = 60%
- error rate = **4**/10 = 40%

- Problem: these metrics treat all misclassifications as being equally bad.

Using a Confusion Matrix

id	gender	age	student?	credit rat.	buy computer? actual	buy computer? predicted
954	male	45	no	good	yes	yes ←
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes ←
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes ←
747	female	47	no	excellent	no	yes

	predicted class: yes	predicted class: no
actual class: **yes**	**3**	
no		

Using a Confusion Matrix

id	gender	age	student?	credit rat.	buy computer? actual	buy computer? predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no ←
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes
747	female	47	no	excellent	no	yes

	predicted class: yes	predicted class: no
actual class: **yes**	3	**1**
no		

Using a Confusion Matrix

					buy computer?	
id	gender	age	student?	credit rat.	actual	predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes ←
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes ←
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes
747	female	47	no	excellent	no	yes ←

		predicted class	
		yes	no
actual class:	yes	3	1
	no	**3**	

Using a Confusion Matrix

					buy computer?	
id	gender	age	student?	credit rat.	actual	predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes
747	female	47	no	excellent	no	yes

		predicted class	
		yes	no
actual class:	yes	3	1
	no	3	____

Using a Confusion Matrix

					buy computer?	
id	gender	age	student?	credit rat.	actual	predicted
954	male	45	no	good	yes	yes
888	female	22	yes	good	no	yes
357	male	25	yes	fair	yes	yes
245	female	28	no	excellent	no	no
177	female	80	no	good	no	no
523	male	68	no	good	yes	no
999	male	37	no	good	no	yes
126	female	70	yes	fair	no	no
443	male	19	yes	fair	yes	yes
747	female	47	no	excellent	no	yes

		predicted class	
		yes	no
actual class:	yes	3	1
	no	3	3

the diagonal of the matrix

- the diagonal shows the correctly classified examples

Data Mining Fundamentals, Part II

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Types of Attributes

- *Nominal* attributes have values that are "names" of categories.
  - there is typically a small set of possible values

attribute	possible values
Fever	{Yes, No}
Diagnosis	{Allergy, Cold, Strep Throat}
Body Temp	{below 96, 96-99, 99-102, above 102}

- *Numeric* attributes have values that are **single** numbers.
  - there is typically a wide range of possible values

attribute	possible values
Body Temp	any single real number in 96.0-106.0
Salary	any single integer in $15,000-250,000

  - it makes sense to *order/compare* their values

    $210,000 > $125,000

    98.6 < 101.3

Which of the attributes are numeric?

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

A. customer ID

B. trades/month

C. age

D. two of the above

E. all of the above

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

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	*nominal output attribute* Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
…	…	…	…	…	…	…

```
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

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
…	…	…	…	…	…	…

```
if Congestion = Yes
then Headache = Yes

if Sore Throat = Yes and Swollen Glands = No
then Congestion = Yes and Fever = No
...
```

What approach is needed? (cont.)

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

- Problem **3**: We want to know which attributes tend to affect the number of trades per month.

- The best approach is:

  A. database queries

  B. data mining using classification learning

  C. data mining using numeric estimation

  D. data mining using association learning

What approach is needed? (cont.)

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

- Problem **4**: We want to discover relationships between account type, transaction method, and age.
- The best approach is:
  - A. database queries
  - B. data mining using classification learning
  - C. data mining using numeric estimation
  - D. data mining using association learning

What approach is needed? (cont.)

Customer ID	Account Type	Margin Account	Transaction Method	Trades/ Month	Sex	Age	Favorite Recreation	Annual Income
1005	Joint	No	Online	12.5	F	30–39	Tennis	40–59K
1013	Custodial	No	Broker	0.5	F	50–59	Skiing	80–99K
1245	Joint	No	Online	3.6	M	20–29	Golf	20–39K
2110	Individual	Yes	Broker	22.3	M	30–39	Fishing	40–59K
1001	Individual	Yes	Online	5.0	M	40–49	Golf	60–79K

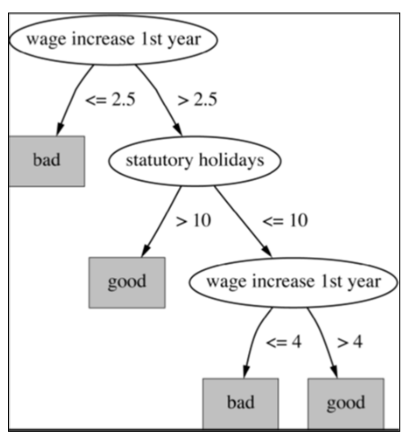
- Problem **5**: We want to know which attributes tend to affect the **annual income**.
- The best approach is:
  - A. database queries
  - B. data mining using classification learning
  - C. data mining using numeric estimation
  - D. data mining using association learning

Another Example: Labor Negotiations

- Goal: to be able to predict whether a proposed labor contract will be acceptable to members of the union.
- Source of this case study: Witten and Frank
- Training data = examples from actual labor negotiations
  - 17 attributes
  - how many of them are input attributes?
  - what are the possible values of the output attribute?
  - what type of machine learning is this?

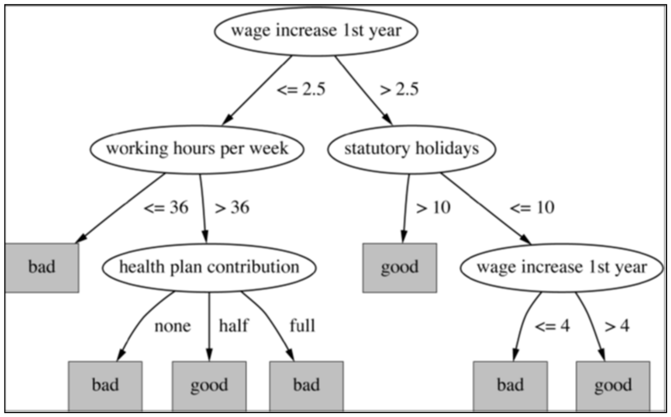
Another Example: Labor Negotiations (cont.)

- Here's one possible decision tree based on the training data:
  - simple model
  - makes intuitive sense
  - misclassifies some of the training examples



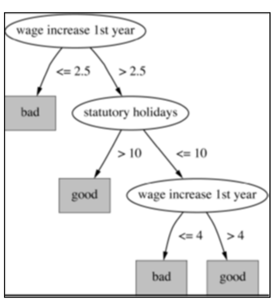
Another Example: Labor Negotiations (cont.)

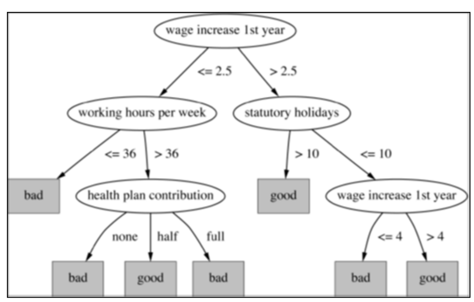
- Here's another possible decision tree from the same data:



- It does a better job classifying the training examples.

Which Model Is Better?





model A
simpler
more intuitive

model B
more accurate on
the training examples

- We need more info.!
  - even though Model B does well on the training data, it may not ***generalize*** – it may *not* do well on previously unseen examples.

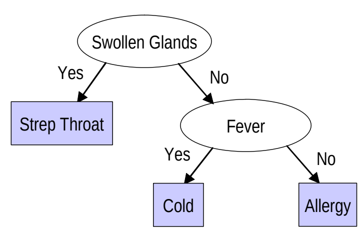
Overfitting

- In general, working too hard to match the training examples can lead to a model that:
  - is overly complicated
  - that doesn't generalize well
- This is known as *overfitting* the training data.
- Extreme overfitting: memorize the training examples!
  - example: medical diagnosis
    - store the training data in a table
    - to diagnose a new patient, find a training example with the same symptoms and use that diagnosis
    - why won't this work?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

Using Test Examples

- Example: decision-tree model for medical diagnosis
  - trained using the 10 earlier examples
  - we can evaluate it using the test examples shown below
- What is its accuracy?



Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis	Model's Diagnosis
12	Yes	Yes	Yes	No	No	Strep throat	
13	No	No	No	Yes	No	Cold	
14	No	Yes	No	Yes	Yes	Cold	
15	Yes	No	Yes	No	Yes	Strep throat	
16	No	Yes	No	Yes	No	Allergy	
17	Yes	No	No	No	Yes	Allergy	

Evaluating Classification Learning Models

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis	Model's Diagnosis
12	Yes	Yes	Yes	No	No	Strep throat	Strep throat
13	No	No	No	Yes	No	Cold	***Allergy***
14	No	Yes	No	Yes	Yes	Cold	Cold
15	Yes	No	Yes	No	Yes	Strep throat	Strep throat
16	No	Yes	No	Yes	No	Allergy	***Cold***
17	Yes	No	No	No	Yes	Allergy	Allergy

- The *error rate* of a model is the percentage of test examples that it misclassifies.
  - in our example, the error rate = ___________
  - error rate = 100 – accuracy
- 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 Models

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis	Model's Diagnosis
12	Yes	Yes	Yes	No	No	Strep throat	Strep throat
13	No	No	No	Yes	No	Cold	***Allergy***
14	No	Yes	No	Yes	Yes	Cold	Cold
15	Yes	No	Yes	No	Yes	Strep throat	Strep throat
16	No	Yes	No	Yes	No	Allergy	***Cold***
17	Yes	No	No	No	Yes	Allergy	Allergy

- To provide a more detailed picture of the model's accuracy, we can use a *confusion matrix*:

		predicted class		
		cold	allergy	strep throat
actual class:	cold	1	1	0
	allergy	1	1	0
	strep throat	0	0	2

the diagonal of the matrix

  - 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:

		predicted class: cold	allergy	strep throat
actual class:	cold	25	8	7
	allergy	6	15	3
	strep throat	5	4	33

  - what is the accuracy of the model?
    total # of test cases = 106
  - what is its error rate?

Interpreting a Confusion Matrix (cont.)

		predicted class: cold	allergy	strep throat
actual class:	cold	25	8	7
	allergy	6	15	3
	strep throat	5	4	33

- how many test cases of strep throat are there?
- how many actual colds were misdiagnosed?
- what percentage of actual colds were correctly diagnosed?

How many actual allergy cases does the model misdiagnose?

		predicted class		
		cold	allergy	strep throat
actual class:	cold	5	3	2
	allergy	2	6	5
	strep throat	1	2	8

Pre-Lecture
Classification Learning
Using the 1R Algorithm

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Classification Learning

- Classification-learning algorithms:
  - take a set of already classified training examples
  - learn a model that can classify previously unseen examples

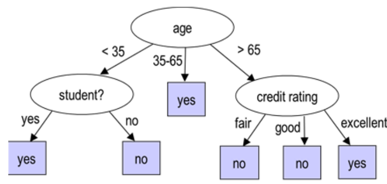
input attributes ⇨ model ⇨ output attribute (the class)

Recall: Classification Learning

- Classification-learning algorithms:
  - take a set of already classified training examples
  - learn a model that can classify previously unseen examples

- Example:

<u>inputs</u>
age
student?
credit rating
…



<u>output</u>
buy computer?

Example Problem: Credit-Card Promotions

- A credit-card company wants to determine which customers should be sent promotional materials for a life insurance offer.

age
gender
income range
…

model

Yes (will accept the offer)
or
No (will not accept the offer)

Example Problem: Credit-Card Promotions

- 15 training examples:

Age	Gender	Income Range	Credit Card Insurance	*class/ output attribute* Life Insurance Promotion
45	Male	40–50K	No	**No**
40	Female	30–40K	No	**Yes**
42	Male	40–50K	No	**No**
43	Male	30–40K	Yes	**Yes**
38	Female	50–60K	No	**Yes**
55	Female	20–30K	No	**No**
35	Male	30–40K	Yes	**Yes**
27	Male	20–30K	No	**No**
43	Male	30–40K	No	**No**
41	Female	30–40K	No	**Yes**
43	Female	40–50K	No	**Yes**
29	Male	20–30K	No	**Yes**
39	Female	50–60K	No	**Yes**
55	Male	40–50K	No	**No**
19	Female	20–30K	Yes	**Yes**

1R: Learning Simple Classification Rules

- Developed by R.C. Holte
- Why 1R?
  - R because the algorithm learns a set of Rules
  - 1 because the rules are based on only 1 input attribute
- Basic idea:
  - Determine a separate set of rules for each input attribute.
  - Pick the set of rules with the highest accuracy on the training data.

Applying 1R to the Credit-Card Promotion Data

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	No	No
40	**Female**	30–40K	No	**Yes**
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	**Female**	50–60K	No	**Yes**
55	**Female**	20–30K	No	**No**
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	**Female**	30–40K	No	**Yes**
43	**Female**	40–50K	No	**Yes**
29	Male	20–30K	No	Yes
39	**Female**	50–60K	No	**Yes**
55	Male	40–50K	No	No
19	**Female**	20–30K	Yes	**Yes**

- Let's start by determining the rules based on Gender.

```
Gender:  Female → Yes

         Male →
```

because Yes is the class of the majority (6 out of 7) of the examples in which Gender is Female

Applying 1R to the Credit-Card Promotion Data

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	**Male**	40–50K	No	**No**
40	Female	30–40K	No	Yes
42	**Male**	40–50K	No	**No**
43	**Male**	30–40K	Yes	**Yes**
38	Female	50–60K	No	Yes
55	Female	20–30K	No	No
35	**Male**	30–40K	Yes	**Yes**
27	**Male**	20–30K	No	**No**
43	**Male**	30–40K	No	**No**
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	**Male**	20–30K	No	**Yes**
39	Female	50–60K	No	Yes
55	**Male**	40–50K	No	**No**
19	Female	20–30K	Yes	Yes

- Let's start by determining the rules based on Gender.

```
Gender:  Female → Yes
         Male → No
```

because No is the class of the majority (5 out of 8) of the examples in which Gender is Male

Applying 1R to the Credit-Card Promotion Data (cont.)

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	No	No
40	Female	30–40K	No	Yes
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	Female	50–60K	No	Yes
55	Female	20–30K	No	No
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	Male	20–30K	No	Yes
39	Female	50–60K	No	Yes
55	Male	40–50K	No	No
19	Female	20–30K	Yes	Yes

- Thus, we end up with the following rules based on Gender:

```
Gender:  Female → Yes      (6 out of 7)
         Male → No         (5 out of 8)
```

overall accuracy = $\frac{6 + 5}{15} = \frac{11}{15}$ = 73%

Applying 1R to the Credit-Card Promotion Data (cont.)

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	**No**	**No**
40	Female	30–40K	**No**	**Yes**
42	Male	40–50K	**No**	**No**
43	Male	30–40K	Yes	Yes
38	Female	50–60K	**No**	**Yes**
55	Female	20–30K	**No**	**No**
35	Male	30–40K	Yes	Yes
27	Male	20–30K	**No**	**No**
43	Male	30–40K	**No**	**No**
41	Female	30–40K	**No**	**Yes**
43	Female	40–50K	**No**	**Yes**
29	Male	20–30K	**No**	**Yes**
39	Female	50–60K	**No**	**Yes**
55	Male	40–50K	**No**	**No**
19	Female	20–30K	Yes	Yes

- What rules would be produced for Credit Card Insurance?

```
Credit Card Insurance:  Yes →
                        No →
```

overall accuracy =

Applying 1R to the Credit-Card Promotion Data (cont.)

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	No	No
40	Female	30–40K	No	Yes
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	Female	50–60K	No	Yes
55	Female	20–30K	No	No
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	Male	20–30K	No	Yes
39	Female	50–60K	No	Yes
55	Male	40–50K	No	No
19	Female	20–30K	Yes	Yes

- What rules would be produced for Income Range?

```
Income Range:  20-30K → No/Yes  (2 out of 4)
               30-40K → Yes     (4 out of 5)
               40-50K → No      (3 out of 4)
               50-60K → Yes     (2 out of 2)
```

Applying 1R to the Credit-Card Promotion Data (cont.)

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	No	No
40	Female	30–40K	No	Yes
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	Female	50–60K	No	Yes
55	Female	20–30K	No	No
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	Male	20–30K	No	Yes
39	Female	50–60K	No	Yes
55	Male	40–50K	No	No
19	Female	20–30K	Yes	Yes

- What rules would be produced for Income Range?

```
Income Range:  20-30K → No      (2 out of 4)
               30-40K → Yes     (4 out of 5)
               40-50K → No      (3 out of 4)
               50-60K → Yes     (2 out of 2)
```

overall accuracy = $\frac{2 + 4 + 3 + 2}{15} = \frac{11}{15} = 73\%$

Handing Numeric Attributes

- To handle numeric attributes, we need to *discretize* the range of possible values into subranges called *bins* or *buckets.*
- One approach: (1) sort the training instances by age
  (2) find the most accurate *binary* (2-way) split

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
45	Male	40–50K	No	No
40	Female	30–40K	No	Yes
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	Female	50–60K	No	Yes
55	Female	20–30K	No	No
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	Male	20–30K	No	Yes
39	Female	50–60K	No	Yes
55	Male	40–50K	No	No
19	Female	20–30K	Yes	Yes

sort by age

Age:	19	27	29	35	38	39	40	41	42	43	43	43	45	55	55
Life Ins:	Y	N	Y	Y	Y	Y	Y	Y	N	Y	Y	N	N	N	N

Handling Numeric Attributes (cont.)

- Here's one possible binary split for age:

Age:	19	27	29	35	38	39	40	41	42	43	43	43	45	55	55
Life Ins:	Y	N	Y	Y	Y	Y	Y	Y	N	Y	Y	N	N	N	N

  - the corresponding rules are:

```
Age: <= 39 → Yes
      > 39 → No
```
(5 out of 6)
(5 out of 9)

overall accuracy: 10/15 = 67%

- The following is one of the splits with the best overall accuracy:

Age:	19	27	29	35	38	39	40	41	42	43	43	43	45	55	55
Life Ins:	Y	N	Y	Y	Y	Y	Y	Y	N	Y	Y	N	N	N	N

  - the corresponding rules are:

```
Age: <= 43 → Yes
      > 43 → No
```
(9 out of 12)
(3 out of 3)

overall accuracy: 12/15 = 80%

Summary of 1R Results

Gender: Female	→	Yes	(6 out of 7)	overall accuracy: 11/15 = 73%
Male	→	No	(5 out of 8)	
Cred.Card Ins: Yes	→	Yes	(3 out of 3)	overall accuracy: 9/15 = 60%
No	→	No*	(6 out of 12)	
Income Range: 20-30K	→	No*	(2 out of 4)	overall accuracy: 11/15 = 73%
30-40K	→	Yes	(4 out of 5)	
40-50K	→	No	(3 out of 4)	
50-60K	→	Yes	(2 out of 2)	
Age: <= 43	→	Yes	(9 out of 12)	overall accuracy: 12/15 = 80%
> 43	→	No	(3 out of 3)	

- Because the rules based on Age have the highest overall accuracy on the training data, 1R selects them as the model.

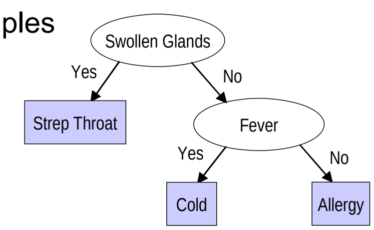
Evaluating Models (cont.); Classification Learning, Part I

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Evaluating Classification Learning Models

- 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



Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis	Model's Diagnosis
12	Yes	Yes	Yes	No	No	Strep throat	Strep throat
13	No	No	No	Yes	No	Cold	***Allergy***
14	No	Yes	No	Yes	Yes	Cold	Cold
15	Yes	No	Yes	No	Yes	Strep throat	Strep throat
16	No	Yes	No	Yes	No	Allergy	***Cold***
17	Yes	No	No	No	Yes	Allergy	Allergy

Recall: Evaluating Classification Learning Models

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis	Model's Diagnosis
12	Yes	Yes	Yes	No	No	Strep throat	Strep throat
13	No	No	No	Yes	No	Cold	***Allergy***
14	No	Yes	No	Yes	Yes	Cold	Cold
15	Yes	No	Yes	No	Yes	Strep throat	Strep throat
16	No	Yes	No	Yes	No	Allergy	***Cold***
17	Yes	No	No	No	Yes	Allergy	Allergy

- To provide a more detailed picture of the model's accuracy, we can use a *confusion matrix*:

		predicted class: cold	allergy	strep throat
actual class:	cold	1	1	0
	allergy	1	1	0
	strep throat	0	0	2

the diagonal of the matrix

- the diagonal of the matrix shows cases that were correctly classified

How many of the predictions of allergy are incorrect?

		predicted class: cold	allergy	strep throat
actual class:	cold	5	3	2
	allergy	2	6	5
	strep throat	1	2	8

What is the overall accuracy of the model?

		predicted class: cold	allergy	strep throat
actual class:	cold	5	3	2
	allergy	2	6	5
	strep throat	1	2	8

total # of test examples = 34

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"

		predicted: yes	no
actual:	yes	TP	FN
	no	FP	TN

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:

		predicted by model A: fraud	predicted by model A: not fraud
actual:	fraud	100	10
	not fraud	40	250

overall accuracy = (100+250)/400 = .875

		predicted by model B: fraud	predicted by model B: not fraud
actual:	fraud	80	30
	not fraud	20	270

overall accuracy = (80+270)/400 = .875

  - 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"!

		predicted: fraud	predicted: not fraud
actual:	fraud	0	10
	not fraud	0	990

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.

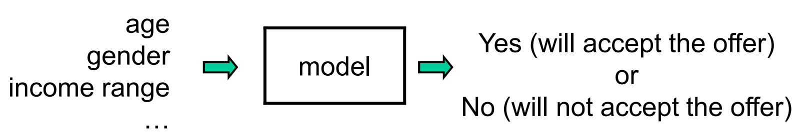
Recall: Classification Learning

- Classification-learning algorithms:
  - take a set of already classified training examples
  - learn a model that can classify previously unseen examples
- The resulting model works like this:

input attributes (everything but the class) ⇒ model ⇒ **nominal** output attribute (the class)

Recall: Credit-Card Promotions

- A credit-card company wants to determine which customers should be sent promotional materials for a life insurance offer.



Recall: Credit-Card Promotions

- 15 training examples:

Age	Gender	Income Range	Credit Card Insurance*	*class/ output attribute* **Life Insurance Promotion**
45	Male	40–50K	No	**No**
40	Female	30–40K	No	**Yes**
42	Male	40–50K	No	**No**
43	Male	30–40K	Yes	**Yes**
38	Female	50–60K	No	**Yes**
55	Female	20–30K	No	**No**
35	Male	30–40K	Yes	**Yes**
27	Male	20–30K	No	**No**
43	Male	30–40K	No	**No**
41	Female	30–40K	No	**Yes**
43	Female	40–50K	No	**Yes**
29	Male	20–30K	No	**Yes**
39	Female	50–60K	No	**Yes**
55	Male	40–50K	No	**No**
19	Female	20–30K	Yes	**Yes**

* note: credit-card insurance is a Yes/No attribute specifying whether the customer accepted a similar offer for insurance on their credit card

Recall: Summary of 1R Results

`Gender: Female → Yes` `Male → No`	(6 out of 7) (5 out of 8)	overall accuracy: 11/15 = 73%
`Cred.Card Ins: Yes → Yes` `No → No*`	(3 out of 3) (6 out of 12)	overall accuracy: 9/15 = 60%
`Income Range: 20-30K → No*` `30-40K → Yes` `40-50K → No` `50-60K → Yes`	(2 out of 4) (4 out of 5) (3 out of 4) (2 out of 2)	overall accuracy: 11/15 = 73%
`Age: <= 43 → Yes` `> 43 → No`	(9 out of 12) (3 out of 3)	overall accuracy: 12/15 = **80%**

- 1R learned the above set of *candidate models.*
- Because the rules based on Age have the highest overall accuracy on the training data, 1R selects them as the *final model*.

Returning to our medical-diagnosis dataset...

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- We want to be able to diagnose new patients...
- What is the output attribute?

A. Patient ID#

B. Swollen Glands

C. Fever

D. Diagnosis

E. more than one of the above

Returning to our medical-diagnosis dataset...

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- We want to be able to diagnose new patients...
- What is the output attribute?
- Because it is nominal:
  - we need classification learning
  - 1R is one possible algorithm we could use

What rules would 1R learn based on Congestion?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

A.
```
Congestion:
  Yes  → Cold
  No   → Strep throat
```

B.
```
Congestion:
  Yes  → Allergy
  No   → Strep throat
```

C.
```
Congestion:
  Yes  → Cold
  No   → Allergy
```

D. more than one of these could be learned

What rules would 1R learn based on Swollen Glands?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

A.
```
Swollen Glands:
  Yes  → Strep Throat
  No   → Allergy
```

B.
```
Swollen Glands:
  Yes  → Strep Throat
  No   → Cold
```

C.
```
Swollen Glands:
  Yes  → Cold
  No   → Strep Throat
```

D. more than one of these could be learned

What rules would 1R learn based on Headache?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

A.
```
Headache:
  Yes  → Cold
  No   → Strep throat
```

B.
```
Headache:
  Yes  → Allergy
  No   → Strep throat
```

C.
```
Headache:
  Yes  → Allergy
  No   → Cold
```

D. more than one of these could be learned

1R Results for Medical-Diagnosis Data Set

Rules	Accuracy
`Congestion: Yes → Cold (4/8)` `No → Strep throat (2/2)`	overall accuracy: 6/10 = 60%
`Swollen Glands: Yes → Strep Throat (3/3)` `No → Cold (4/7)`	overall accuracy: 7/10 = **70%**
`Headache: Yes → Allergy (2/5)` `No → Strep throat (2/5)`	overall accuracy: 4/10 = 40%
`Sore Throat: Yes → Strep Throat (2/5)` `No → Allergy (2/5)`	overall accuracy: 4/10 = 40%
`Fever: Yes → Cold (4/5)` `No → Allergy (3/5)`	overall accuracy: 7/10 = **70%**

- 1R learns the above set of *candidate models.*
- Two models are tied for overall accuracy:
  - the rules based on Swollen Glands
  - the rules based on Fever
- 1R can select either of them as the *final model*.

What About Patient ID?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- If we learned rules based on Patient ID, what accuracy would they have?

What About Patient ID?

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- If we learned rules based on Patient ID, what accuracy would they have? 100%!

```
Patient ID:  1 → Strep throat (1/1)
             2 → Allergy (1/1)
             3 → Cold (1/1)
             ...
```

  - these rules just memorize the training examples!
  - they are an extreme example of overfitting!

Special Case: Many-Valued Inputs

- In general, 1R doesn't tend to work well with a *nominal* input attribute that has many possible values.
  - 1R often ends up selecting its rules
  - the rules overfit the training data
- Thus, we may need to remove such attributes before we start data mining.

Another Limitation of 1R

Patient ID#	Sore Throat	Fever	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	Yes	Yes	Yes	Yes	Strep throat
2	No	No	No	Yes	Yes	Allergy
3	Yes	Yes	No	Yes	No	Cold
4	Yes	No	Yes	No	No	Strep throat
5	No	Yes	No	Yes	No	Cold
6	No	No	No	Yes	No	Allergy
7	No	No	Yes	No	No	Strep throat
8	Yes	No	No	Yes	Yes	Allergy
9	No	Yes	No	Yes	Yes	Cold
10	Yes	Yes	No	Yes	Yes	Cold

- There are three possible classes: Strep Throat, Cold, Allergy
- Binary attributes like Fever produce rules that predict at most two of these classes:

```
Fever:   Yes → Cold
          No → Allergy
```

- When this happens, a 1R model alone is not sufficient.
- We'll see next time how to build models with multiple inputs.

Pre-Lecture
Classification Learning: Learning a Decision Tree

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Credit-Card Promotions Problem

- A credit-card company wants to determine which customers should be sent promotional materials for a life insurance offer.

inputs
age
gender
income range
…

→ model →

output
Yes (will accept the offer)
or
No (will not accept the offer)

Recall: Credit-Card Promotions Problem

- 15 training examples:

Age	Gender	Income Range	Credit Card Insurance	*class/ output attribute* **Life Insurance Promotion**
45	Male	40–50K	No	**No**
40	Female	30–40K	No	**Yes**
42	Male	40–50K	No	**No**
43	Male	30–40K	Yes	**Yes**
38	Female	50–60K	No	**Yes**
55	Female	20–30K	No	**No**
35	Male	30–40K	Yes	**Yes**
27	Male	20–30K	No	**No**
43	Male	30–40K	No	**No**
41	Female	30–40K	No	**Yes**
43	Female	40–50K	No	**Yes**
29	Male	20–30K	No	**Yes**
39	Female	50–60K	No	**Yes**
55	Male	40–50K	No	**No**
19	Female	20–30K	Yes	**Yes**

Recall: Candidate Rules for 1R

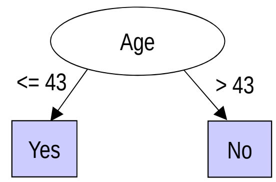
`Gender: Female → Yes`	(6 out of 7)	overall accuracy: 11/15 = 73%
`Male → No`	(5 out of 8)	
`Cred.Card Ins: Yes → Yes`	(3 out of 3)	overall accuracy: 9/15 = 60%
`No → No`	(6 out of 12)	
`Income Range: 20-30K → No`	(2 out of 4)	overall accuracy: 11/15 = 73%
`30-40K → Yes`	(4 out of 5)	
`40-50K → No`	(3 out of 4)	
`50-60K → Yes`	(2 out of 2)	
`Age: <= 43 → Yes`	(9 out of 12)	overall accuracy: 12/15 = **80%**
`> 43 → No`	(3 out of 3)	

- 1R learns the above set of *candidate rules/models.*
- 1R chooses the Age model, because its overall accuracy is best.
- When building a decision tree, we need to consider other factors.

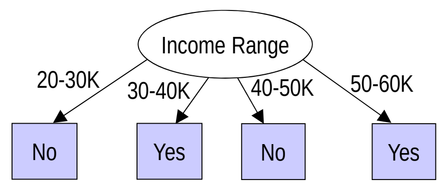
1R and Decision Trees

- We can view a set of rules learned by 1R as a simple decision tree with only one decision.

```
Age: <= 43 → Yes
      > 43 → No
```




```
Income Rng: 20-30K → No
            30-40K → Yes
            40-50K → No
            50-60K → Yes
```



Building Decision Trees

- Here's the basic algorithm:
  1. apply 1R, but choose the candidate rules that "best divide" the examples into subgroups
  2. create a decision tree based on those rules
  3. for each subgroup created by the new decision tree:
     - if its classifications are "accurate enough," do nothing
     - otherwise, build a tree for the examples in the subgroup

Choosing the Rules that "Best Divide"

- We compute a *goodness score* for each set of candidate rules:

goodness = overall accuracy / N

N = # of subgroups the rules don't classify "accurately enough"

```
Gender: Female → Yes (6 / 7)
          Male → No  (5 / 8)
```

overall accuracy = 11/15 = 73%
N = 2
goodness = 73/2 = 36.5

```
Income Rng: 20-30K → No  (2 / 4)
            30-40K → Yes (4 / 5)
            40-50K → No  (3 / 4)
            50-60K → Yes (2 / 2)
```

overall accuracy = 11/15 = 73%

N = _____

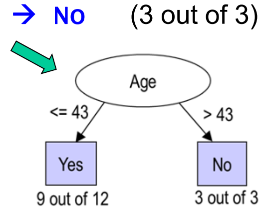
goodness = _____________

- larger trees tend to overfit the data
- by dividing by N, our goodness score favors smaller trees
- special case: if N == 0 for an attribute, choose that attribute!

Building a Decision Tree for the Credit-Card Data

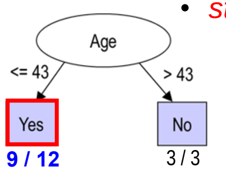
- Here are the rules we obtained for each attribute using 1R:

`Gender: Female → Yes`	(6 out of 7)	accuracy: 11/15 = 73%	
`Male → No`	(5 out of 8)	*goodness: 73/2 = 36.5*	
`Cred.Card Ins: Yes → Yes`	(3 out of 3)	accuracy: 9/15 = 60%	
`No → No*`	(6 out of 12)	*goodness: 60/1 = 60*	
`Income Rng: 20-30K → No*`	(2 out of 4)		
`30-40K → Yes`	(4 out of 5)	accuracy: 11/15 = 73%	
`40-50K → No`	(3 out of 4)	*goodness: 73/3 = 24.3*	
`50-60K → Yes`	(2 out of 2)		
`Age: <= 43 → Yes`	(9 out of 12)	accuracy: 12/15 = 80%	
`> 43 → No`	(3 out of 3)	*goodness: 80/1 =* ***80***	



Building Decision Trees

- Here's the basic algorithm:
  1. apply 1R, but choose the candidate rules that "best divide" the examples into subgroups
  2. create a decision tree based on those rules
  3. for each subgroup created by the new decision tree:
     - if its classifications are "accurate enough," do nothing
     - otherwise, build a tree for the examples in the subgroup
       - *start the algorithm over again on just those examples!*



initial tree for
all training examples

Repeating the Algorithm on a Subgroup

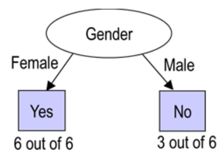
- Here are the 12 examples in the Age <= 43 subgroup:

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
40	Female	30–40K	No	Yes
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
38	Female	50–60K	No	Yes
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
41	Female	30–40K	No	Yes
43	Female	40–50K	No	Yes
29	Male	20–30K	No	Yes
39	Female	50–60K	No	Yes
19	Female	20–30K	Yes	Yes

Building a Decision Tree for the Credit-Card Data (cont.)

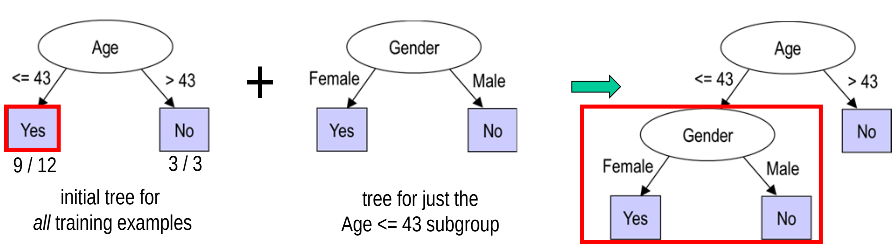
- Here are the rules obtained for these 12 examples:

Gender: Female → Yes Male → No	(6 out of 6) (3 out of 6)	accuracy: 9/12 = 75% *goodness: 75/1 = **75***
Cred.Card Ins: Yes → Yes No → Yes	(3 out of 3) (6 out of 9)	accuracy: 9/12 = 75% *goodness: 75/1 = **75***
Income Rng: 20-30K → No 30-40K → Yes 40-50K → No 50-60K → Yes	(2 out of 3) (4 out of 5) (1 out of 2) (2 out of 2)	accuracy: 9/12 = 75% *goodness: 75/3 = 25*
Age: <= 41 → Yes > 41 → No	(7 out of 8) (2 out of 4)	accuracy: 9/12 = 75% *goodness: ____________*



Recall: Building Decision Trees

- Here's the basic algorithm:
  1. apply 1R, but choose the candidate rules that "best divide" the examples into subgroups
  2. create a decision tree based on those rules
     - if we already have an existing tree, put the new tree in the appropriate place for its subgroup
  3. for each subgroup created by the new decision tree:
     - if its classifications are "accurate enough," do nothing
     - otherwise, build a tree for the examples in the subgroup



Classification Learning, Part II

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Credit-Card Promotions

- A credit-card company wants to determine which customers should be sent promotional materials for a life insurance offer.

age
gender
income range
…

→ model →

Yes (will accept the offer)
or
No (will not accept the offer)

Recall: Credit-Card Promotions

- 15 training examples:

Age	Gender	Income Range	Credit Card Insurance*	*class/ output attribute* Life Insurance Promotion
45	Male	40–50K	No	**No**
40	Female	30–40K	No	**Yes**
42	Male	40–50K	No	**No**
43	Male	30–40K	Yes	**Yes**
38	Female	50–60K	No	**Yes**
55	Female	20–30K	No	**No**
35	Male	30–40K	Yes	**Yes**
27	Male	20–30K	No	**No**
43	Male	30–40K	No	**No**
41	Female	30–40K	No	**Yes**
43	Female	40–50K	No	**Yes**
29	Male	20–30K	No	**Yes**
39	Female	50–60K	No	**Yes**
55	Male	40–50K	No	**No**
19	Female	20–30K	Yes	**Yes**

* note: credit-card insurance is a Yes/No attribute specifying whether the customer accepted a similar offer for insurance on their credit card

Recall: Building Decision Trees

- Here's the basic algorithm:
  1. apply 1R, but choose the candidate rules that **"best divide"** the examples into subgroups
  2. create a decision tree based on those rules
  3. for each subgroup created by the new decision tree:
     - if its classifications are "accurate enough," do nothing
     - otherwise, build a tree for the examples in the subgroup

Choosing the Rules that "Best Divide"

```
Gender: Female → Yes (6/7)
          Male → No  (5/8)
```

overall accuracy = 11/15 = **73%**

```
Income Rng: 20-30K → No  (2/4)
            30-40K → Yes (4/5)
            40-50K → No  (3/4)
            50-60K → Yes (2/2)
```

overall accuracy = 11/15 = **73%**

Recall: Overfitting

- In general, working too hard to match the training examples can lead to a model that:
  - is overly complicated
  - that doesn't generalize well
- This is known as *overfitting* the training data.
- The larger a decision tree gets, the more likely it is to overfit.
  - its rules/decisions are based on smaller and smaller subgroups of training data

Choosing the Rules that "Best Divide"

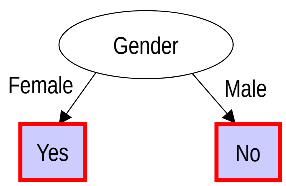
```
Gender: Female → Yes (6 / 7)
        Male   → No  (5 / 8)
```

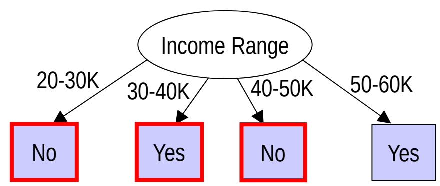
overall accuracy = 11/15 = **73%**

```
Income Rng: 20-30K → No  (2 / 4)
            30-40K → Yes (4 / 5)
            40-50K → No  (3 / 4)
            50-60K → Yes (2 / 2)
```

overall accuracy = 11/15 = **73%**

- If we choose `Income Rng`, the final tree is likely to be larger.
  - it has more subgroups that we need to expand, because they are not classified "accurately enough"





Choosing the Rules that "Best Divide"

```
Gender: Female → Yes (6 / 7)
        Male   → No  (5 / 8)
```

overall accuracy = 11/15 = **73%**
N = **2**
goodness = **73**/**2** = 36.5

```
Income Rng: 20-30K → No  (2 / 4)
            30-40K → Yes (4 / 5)
            40-50K → No  (3 / 4)
            50-60K → Yes (2 / 2)
```

overall accuracy = 11/15 = **73%**
N = **3**
goodness = **73**/**3** = 24.3

- We compute a *goodness score* for each set of candidate rules:

  goodness = overall accuracy / N

  N = # of subgroups the rules don't classify "accurately enough" (perfectly)
  - by dividing by N, we get a score that favors smaller trees that are less likely to overfit
  - special case: if N == 0 for an attribute, choose that attribute!

Consider these rules based on different training data. Which set of rules has the highest goodness score?

A.

`Gender: Female → Yes`	(6 out of 7)	accuracy: 8/10 = 80%	
`Male → No`	(2 out of 3)	*goodness: 80/2 = 40*	

B.

`Cred.Card Ins: Yes → Yes`	(3 out of 3)	accuracy: 7/10 = 70%
`No → No`	(4 out of 7)	*goodness: ?*

C.

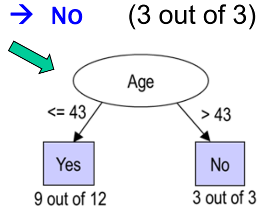
`Income Rng: 20-30K → No`	(3 out of 3)	
`30-40K → Yes`	(2 out of 2)	accuracy: 8/10 = 80%
`40-50K → No*`	(1 out of 2)	*goodness: ?*
`50-60K → Yes`	(2 out of 3)	

D. more than one of the above

Candidate rules based on our original training data…

- Here are the rules we obtained for each attribute using 1R:

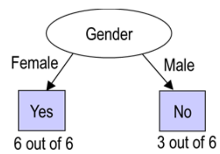
`Gender: Female → Yes`	(6 out of 7)	accuracy: 11/15 = 73%
`Male → No`	(5 out of 8)	*goodness: 73/2 = 36.5*
`Cred.Card Ins: Yes → Yes`	(3 out of 3)	accuracy: 9/15 = 60%
`No → No*`	(6 out of 12)	*goodness: 60/1 = 60*
`Income Rng: 20-30K → No*`	(2 out of 4)	
`30-40K → Yes`	(4 out of 5)	accuracy: 11/15 = 73%
`40-50K → No`	(3 out of 4)	*goodness: 73/3 = 24.3*
`50-60K → Yes`	(2 out of 2)	
`Age: <= 43 → Yes`	(9 out of 12)	accuracy: 12/15 = 80%
`> 43 → No`	(3 out of 3)	*goodness: 80/1 = **80***



Repeating the Algorithm on a Subgroup

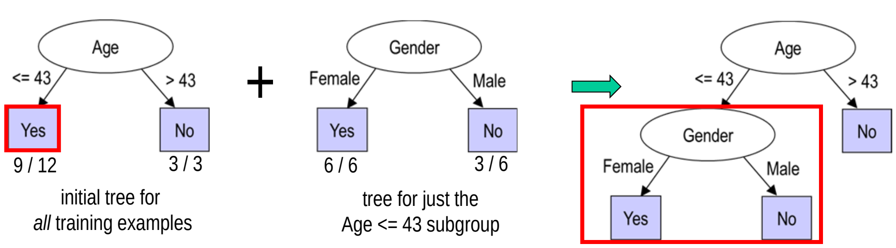
- Here are the rules obtained for the 12 examples with Age <= 43:

`Gender: Female → Yes`		(6 out of 6)	accuracy: 9/12 = 75%
`Male → No`		(3 out of 6)	*goodness: 75/1 =* ***75***
`Cred.Card Ins: Yes → Yes`		(3 out of 3)	accuracy: 9/12 = 75%
`No → Yes`		(6 out of 9)	*goodness: 75/1 =* ***75***
`Income Rng: 20-30K → No`		(2 out of 3)	
`30-40K → Yes`		(4 out of 5)	accuracy: 9/12 = 75%
`40-50K → No`		(1 out of 2)	*goodness: 75/3 = 25*
`50-60K → Yes`		(2 out of 2)	
`Age: <= 41 → Yes`		(7 out of 8)	accuracy: 9/12 = 75%
`> 41 → No`		(2 out of 4)	*goodness: 75/2 = 37.5*

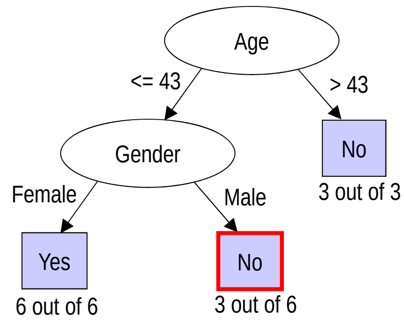


Recall: Building Decision Trees

- Here's the basic algorithm:
  1. apply 1R, but choose the candidate rules that "best divide" the examples into subgroups
  2. create a decision tree based on those rules
     - if we already have an existing tree, put the new tree in the appropriate place for its subgroup
  3. for each subgroup created by the new decision tree:
     - if its classifications are "accurate enough," do nothing
     - otherwise, build a tree for the examples in the subgroup



Repeating the Algorithm on a Subgroup (cont.)



- The subgroup (Age <= 43, Gender == Female) is already perfectly classified.
- The subgroup (Age <= 43, Gender == Male) is not, so we build a tree for that subgroup!

Repeating the Algorithm on a Subgroup (cont.)

- Here are the 6 examples in that subgroup:

Age	Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
42	Male	40–50K	No	No
43	Male	30–40K	Yes	Yes
35	Male	30–40K	Yes	Yes
27	Male	20–30K	No	No
43	Male	30–40K	No	No
29	Male	20–30K	No	Yes

sort by Age

Age:	27	29	35	42	43	43
Life Ins:	N	Y	Y	N	Y	N

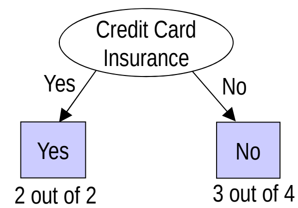
- We no longer consider Gender. Why?

Repeating the Algorithm on a Subgroup (cont.)

- Here are the rules obtained for these 6 examples:

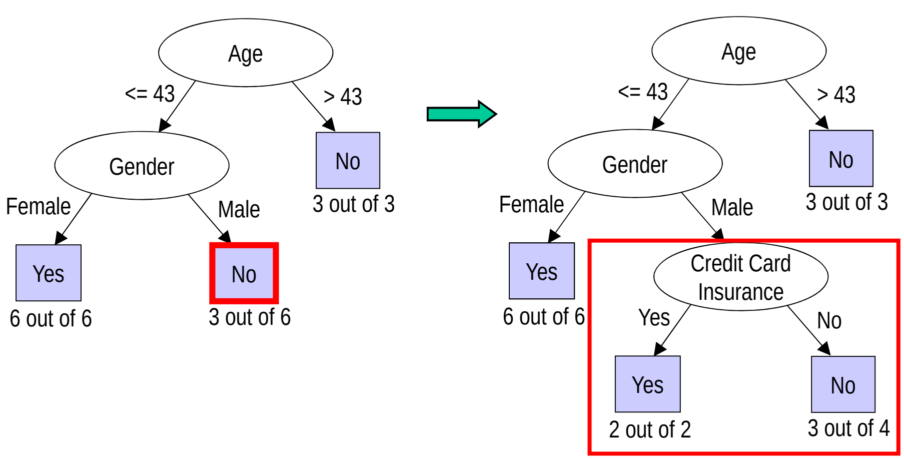
`Cred.Card Ins: Yes → Yes`	(2 out of 2)	accuracy: 5/6 = 83.3%	
`No → No`	(3 out of 4)	*goodness: 83.3/1 =* ***83.3***	
`Income Rng: 20-30K → No*`	(1 out of 2)		
`30-40K → Yes`	(2 out of 3)	accuracy: 4/6 = 66.7%	
`40-50K → No`	(1 out of 1)	*goodness: 66.7/2 = 33.3*	
`50-60K → ?`	(none)		
`Age: <= 35 → Yes`	(2 out of 3)	accuracy: 4/6 = 66.7%	
`> 35 → No`	(2 out of 3)	*goodness: 66.7/2 = 33.3*	

- Credit Card Insurance has the highest goodness score, so we pick it and create the partial tree at right:



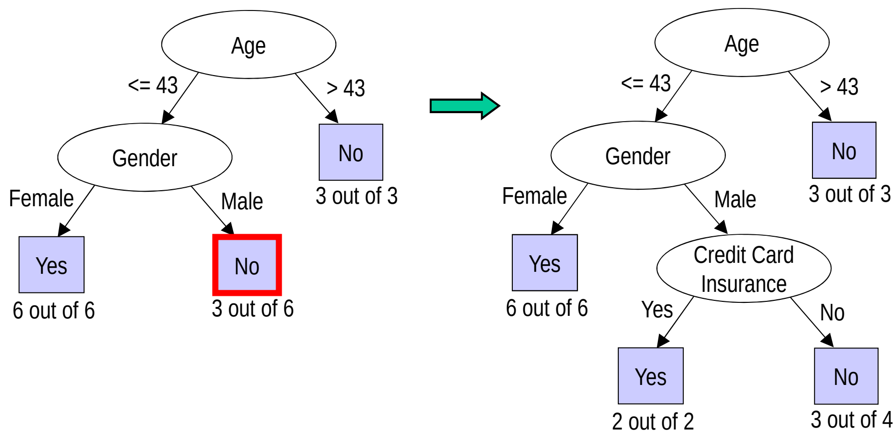
Repeating the Algorithm on a Subgroup (cont.)

- This new tree replaces the classification for the (Age <= 43, Gender = Male) subgroup in the previous tree:



Repeating the Algorithm on a Subgroup (cont.)

- This new tree replaces the classification for the (Age <= 43, Gender = Male) subgroup in the previous tree:



- It turns out that we can't improve this model any further.

Practice Building a Decision Tree

- Consider the following dataset:
  - similar to one we saw previously, but Fever (Yes/No) is replaced with Temp – the person's body temperature

Patient ID#	**Sore Throat**	**Temp**	**Swollen Glands**	**Congestion**	**Headache**	**Diagnosis**
1	Yes	100.4	Yes	Yes	Yes	Strep throat
2	No	97.8	No	Yes	Yes	Allergy
3	Yes	101.2	No	Yes	No	Cold
4	Yes	98.6	Yes	No	No	Strep throat
5	No	102.0	No	Yes	No	Cold
6	No	99.2	No	Yes	No	Allergy
7	No	98.1	Yes	No	No	Strep throat
8	Yes	98.0	No	Yes	Yes	Allergy
9	No	102.5	No	Yes	Yes	Cold
10	Yes	100.7	No	Yes	Yes	Cold

- We again want to learn a model that allows us to diagnose new patients.

Many-Valued Nominal Attributes

Patient ID#	Sore Throat	Temp	Swollen Glands	Congestion	Headache	Diagnosis
1	Yes	100.4	Yes	Yes	Yes	Strep throat
2	No	97.8	No	Yes	Yes	Allergy
3	Yes	101.2	No	Yes	No	Cold
4	Yes	98.6	Yes	No	No	Strep throat
5	No	102.0	No	Yes	No	Cold
6	No	99.2	No	Yes	No	Allergy
7	No	98.1	Yes	No	No	Strep throat
8	Yes	98.0	No	Yes	Yes	Allergy
9	No	102.5	No	Yes	Yes	Cold
10	Yes	100.7	No	Yes	Yes	Cold

- Is Patient ID# numeric or nominal?
- What is the accuracy of rules based on Patient ID#?
  - we get one rule for each ID, which correctly classifies its example. This is overfitting!
- In general, we should:
  - avoid nominal attributes with many possible values
  - remove any attributes that are unique identifiers

Decision-Tree Algorithm: Candidate Rules

- Here are some of the candidate rules for the initial tree:

```
Temp:
   <= 100.4 → Allergy (3/6)
    > 100.4 → Cold (4/4)
```

accuracy = 7/10 = 70%
goodness = 70 / 1 = 70

```
Swollen Glands:
   Yes → Strep throat (3/3)
   No  → Cold (4/7)
```

accuracy = 7/10 = 70%
goodness = ?

```
Congestion:
   Yes → Cold (4/8)
   No  → Strep throat (2/2)
```

accuracy = 6/10 = 60%
goodness = ?

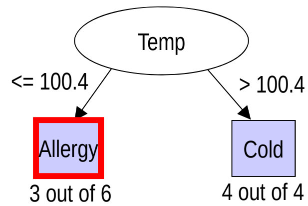
- The other candidates are not as good as these three.
- The decision-tree algorithm could pick which of these?

Decision-Tree Algorithm: Initial Tree

- Assume that it picks the first set:

```
Temp:
    <= 100.4 → Allergy (3/6)
     > 100.4 → Cold (4/4)
```

- Here's the corresponding initial tree:



- We don't need to go further with the right subgroup.
- We *do* need to go further with the left subgroup.
  - the 6 training examples for which Temp <= 100.4

Decision-Tree Algorithm: Processing a Subgroup

- Here are the 6 training examples in the left subgroup:

Sore Throat	Temp	Swollen Glands	Congestion	Headache	Diagnosis
Yes	100.4	Yes	Yes	Yes	Strep throat
No	97.8	No	Yes	Yes	Allergy
Yes	98.6	Yes	No	No	Strep throat
No	99.2	No	Yes	No	Allergy
No	98.1	Yes	No	No	Strep throat
Yes	98.0	No	Yes	Yes	Allergy

- The algorithm develops candidate rules for them, including:

```
Sore Throat:
   Yes →
    No →
```

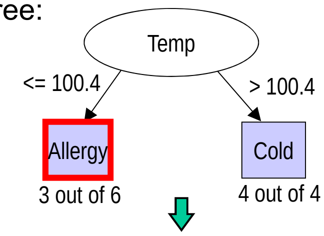
accuracy =
goodness =

```
Swollen Glands:
   Yes →
    No →
```

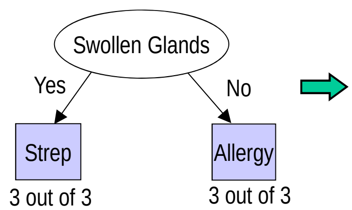
accuracy =
goodness =

Decision-Tree Algorithm: Final Tree

initial tree:



tree we just learned for
Temp <= 100.4 subgroup:



final tree:

Classification Learning, Part III

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Determining if a Mushroom is Poisonous

- 11 training examples:

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Red	Short	Yes	Hairy	No
Purple	Short	No	Smooth	No
Blue	Short	Yes	Hairy	Yes
Red	Tall	No	Rough	No
Blue	Tall	Yes	Smooth	Yes
Blue	Short	Yes	Rough	Yes
Red	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

- What is our output attribute?
- Why is this classification learning?

What rules would 1R learn based on Color?

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Red	Short	Yes	Hairy	No
Purple	Short	No	Smooth	No
Blue	Short	Yes	Hairy	Yes
Red	Tall	No	Rough	No
Blue	Tall	Yes	Smooth	Yes
Blue	Short	Yes	Rough	Yes
Red	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

A.
```
Color:
  Purple → Yes
  Red    → Yes
  Blue   → No
```

B.
```
Color:
  Purple → Yes
  Red    → No
  Blue   → No
```

C.
```
Color:
  Purple → Yes
  Red    → No
  Blue   → Yes
```

D. more than one of these could be learned

What would the final 1R model be?

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Red	Short	Yes	Hairy	No
Purple	Short	No	Smooth	No
Blue	Short	Yes	Hairy	Yes
Red	Tall	No	Rough	No
Blue	Tall	Yes	Smooth	Yes
Blue	Short	Yes	Rough	Yes
Red	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

A.
```
Color:
  Purple → Yes (3/5) }
  Red    → No  (3/3) } (9/11)
  Blue   → Yes (3/3) }
```

B.
```
Height:
  Tall   → ?
  Short  → ?
```

C.
```
Texture:
  Rough  → ?
  Smooth → ?
  Hairy  → ?
```

D.
```
Stripes:
  Yes → ?
  No  → ?
```

E. two or more of the models are tied

Which set of rules would be chosen by our decision-tree algorithm?

A.
```
Color:
  Purple → Yes (3/5)
  Red    → No  (3/3)
  Blue   → Yes (3/3)
```
accuracy: 9/11 = 82%
goodness:

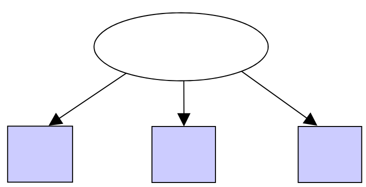
B.
```
Height:
  Tall   → Yes (3/5)
  Short  → No* (3/6)
```
accuracy: 6/11 = 55%
goodness:

C.
```
Texture:
  Rough  → Yes (2/3)
  Smooth → No  (3/5)
  Hairy  → Yes (2/3)
```
accuracy: 7/11 = 64%
goodness:

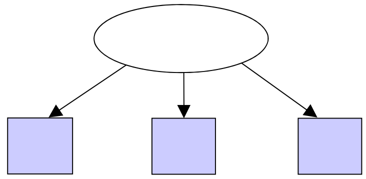
D.
```
Stripes:
  Yes → Yes (5/6)
  No  → No  (4/5)
```
accuracy: 9/11 = 82%
goodness:

E. more than one of the above could be chosen

What is the corresponding initial tree?



What (if anything) will our algorithm do next?



A. learn subtrees for all three subgroups

B. learn a subtree for just one subgroup (which one?)

C. learn subtrees for two of the subgroups

D. nothing – this is the final model

Applying the Algorithm to a Subgroup

- all 11 training examples:

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Red	Short	Yes	Hairy	No
Purple	Short	No	Smooth	No
Blue	Short	Yes	Hairy	Yes
Red	Tall	No	Rough	No
Blue	Tall	Yes	Smooth	Yes
Blue	Short	Yes	Rough	Yes
Red	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

Applying the Algorithm to a Subgroup

- just the Purple subgroup:

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Purple	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

How many candidate models will be considered?

- just the Purple subgroup:

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Purple	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

What rules will be selected for this subgroup?

- just the Purple subgroup:

Color	Height	Stripes	Texture	Poisonous
Purple	Tall	Yes	Rough	Yes
Purple	Tall	Yes	Smooth	Yes
Purple	Short	No	Smooth	No
Purple	Short	No	Hairy	Yes
Purple	Tall	No	Smooth	No

A.
```
Height:
  Tall   →
  Short  →
```
accuracy:
goodness:

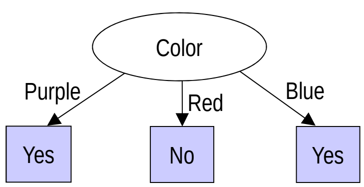
B.
```
Texture:
  Rough  →
  Smooth →
  Hairy  →
```
accuracy:
goodness:

C.
```
Stripes:
  Yes →
  No  →
```
accuracy:
goodness:

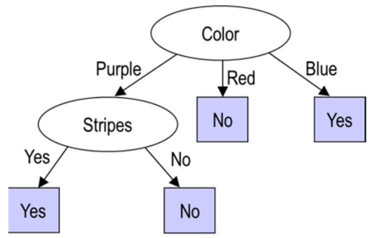
Generate all three sets of candidate rules, and then determine which will be selected!

D. more than one could be selected

How should we incorporate the new rules?

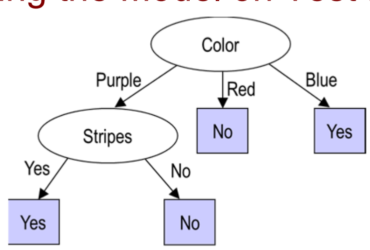


Let's assume the algorithm stops here…



- This model correctly classifies 10/11 of the training examples.
- Do we know if it is a good model?

Evaluating the Model on Test Examples



5 test examples *(not* used to train the model):

				actual	predicted
Color	**Height**	**Stripes**	**Texture**	**Poisonous**	**Poisonous**
Blue	Short	Yes	Smooth	Yes	
Blue	Tall	No	Hairy	No	
Red	Tall	Yes	Hairy	No	
Purple	Tall	Yes	Hairy	Yes	
Purple	Tall	No	Rough	No	

confusion matrix:

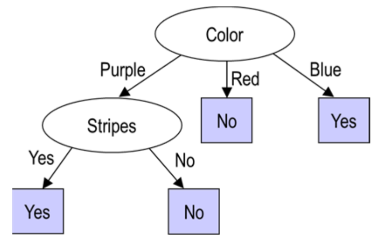
	Yes	**No**
Yes		
No		

accuracy =

error rate =

From Decision Trees to Classification Rules

- What if we wanted to use this model in the context of a program?

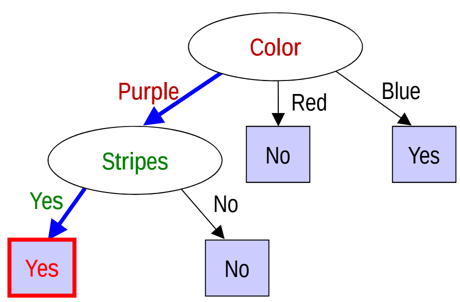


- Any decision tree can be turned into a set of rules of the following form:

```
if condition1 [and condition2 and ...]:
     class = value1
elif ...
     class = value2
...
```

- For each classification node in the tree (each blue square node), we include a rule based on the conditions on the path from the top of the tree to that node.

From Decision Trees to Classification Rules (cont.)



- What are the rules for this tree? (finish the rest!)

```
if Color == Purple and Stripes == Yes:
     Poisonous = Yes
elif

elif

elif
```

Other Algorithms for Learning Decision Trees

- ID3 – uses a different goodness score based on a field of study known as information theory
  - can't handle numeric attributes
- C4.5 – makes a series of improvements to ID3:
  - can handle numeric input attributes
  - can handle missing values
  - *prunes* 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.

Numeric Estimation; Using Weka

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Review: Numeric Estimation

- *Numeric estimation* is like classification learning.
  - it involves learning a model that works like this:

    input attributes → model → output attribute

  - 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.
- The inputs include:
  - 2 attributes that describe the CPU:
    - CTIME: the processor's cycle time (in nanosec)
    - CACHE: cache size (in KB)
  - 4 attributes that describe the task:
    - MMIN: minimum amount of main memory used (in KB)
    - MMAX: maximum amount of main memory used (in KB)
    - plus two others
- We need a model that works like this:

CTIME
MMIN, MMAX
CACHE
CHMIN, CHMAX → **model** → performance
(PERF)

Example Problem: CPU Performance (cont.)

- There are 209 training examples. Here are five of them:

input attributes: *class/ output attribute*

CTIME	MMIN	MMAX	CACHE	CHMIN	CHMAX	PERF
125	256	6000	256	16	128	198
29	8000	32000	32	8	32	269
29	8000	32000	32	8	32	172
125	2000	8000	0	2	14	52
480	512	8000	32	0	0	67

Linear Regression

- The classic approach to numeric estimation is *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:

```
PERF = 0.066CTIME + 0.0143MMIN + 0.0066MMAX +
       0.4945CACHE – 0.1723CHMIN + 1.2012CHMAX – 66.48
```

  - this type of model is known as a *regression equation*
- The general format of a linear regression equation is:

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n + c$$

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

linear regression learns these values

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

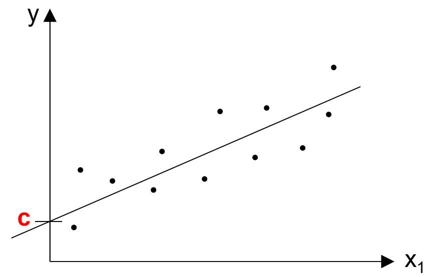
CTIME	MMIN	MMAX	CACHE	CHMIN	CHMAX	PERF
480	1000	4000	0	0	0	**5.9**

we plug the attribute values into the regression equation:

```
PERF = 0.066CTIME + 0.0143MMIN + 0.0066MMAX +
       0.4945CACHE – 0.1723CHMIN + 1.2012CHMAX – 66.48

     = 0.066 * 480 + 0.0143 * 1000 + 0.0066 * 4000 +
       0.4945 * 0 – 0.1723 * 0 + 1.2012 * 0 – 66.48

     = 5.9
```

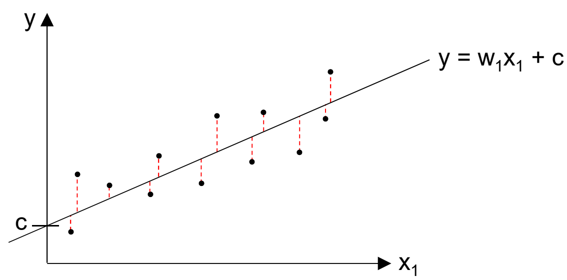
Linear Regression with One Input Attribute

- 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 $y = w_1x_1 + \mathbf{c}$
    - 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

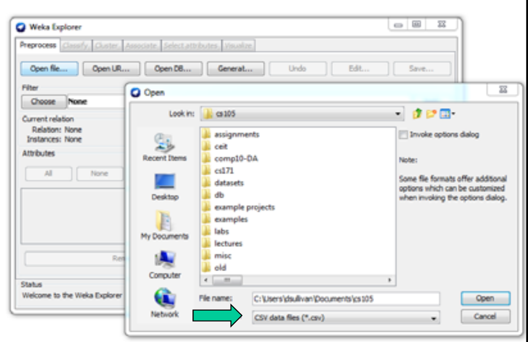
- When there are n input attributes, linear regression finds the equation of a line in (n+1) dimensions.
  - here again, it is the line that "best fits" the training examples
- The equation has the form we mentioned earlier:

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n + c$$

Using Weka

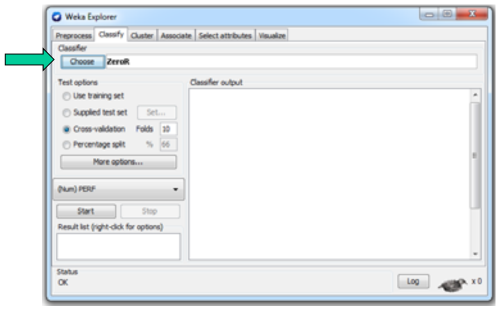
- http://www.cs.waikato.ac.nz/~ml/weka/index.html
- Choose the Explorer tool.

- Click *Open file...*
- Tell Weka to look for CSV files.
- Find/open your data file.
- Perform the necessary pre-processing steps.
  - more on this soon!



Using Weka (cont.)

- After pre-processing, go to the *Classify* tab.
  - used for both classification learning *and* numeric estimation
- Click the *Choose* button to change the algorithm.
  - you will see many folders of algorithms



- Examples:
  - *rules* folder: 1R
  - *trees* folder: J48
  - *functions* folder: Linear Regression
- Feel free to try algorithms that we haven't discussed in lecture!

Linear Regression in Weka

- By default, Weka employs *attribute selection*.
  - it may not include all of the input attributes in the equation
- On the CPU dataset, Weka learns the following equation:

```
PERF = 0.0661CTIME + 0.0142MMIN + 0.0066MMAX +
       0.4871CACHE + 1.1868CHMAX – 66.60
```

  - it does not include the CHMIN attribute
- To force Linear Regression to use all attributes:
  - click on the name of the algorithm
  - change the *attributeSelectionMethod* parameter to *No attribute selection*
  - doing so produces our earlier equation:

```
PERF = 0.066CTIME + 0.0143MMIN + 0.0066MMAX +
       0.4945CACHE – 0.1723CHMIN + 1.2012CHMAX – 66.48
```

The Coefficients in Linear Regression

```
PERF = 0.066CTIME + 0.0143MMIN + 0.0066MMAX +
       0.4945CACHE - 0.1723CHMIN + 1.2012CHMAX - 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 test examples when learning the equation
  - use the equation on the test examples and see how well it does
- 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, but for numeric estimation
- We can use it as a baseline.
  - determine the correlation coefficient of its model
  - if a more complex model has a lower correlation coefficient, don't use it!
  - (we can use 1R in a similar way when doing classification learning)
- It also gives insight into which of the input attributes has the largest impact on the output.

Which of these statements is NOT true?

A. Numeric estimation produces a model that predicts the value of a single output attribute.

B. The model produced by numeric estimation does not need to use all of the input attributes.

C. In order to perform numeric estimation, the input attributes must be numeric.

D. A numeric-estimation model is learned from a set of training examples that include values for the output attribute.

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.
  - ex: `Gender` with possible values `{Female, Male}`
    ➔ `Female?` with possible values `{1, 0}`
    where 1 means Yes, 0 means No

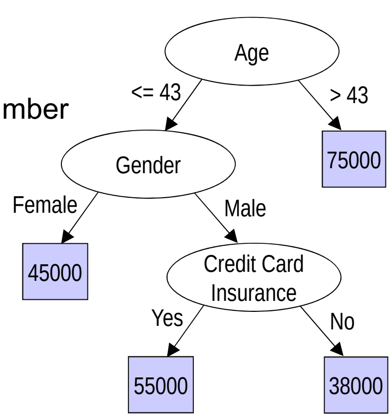
- In Weka, many algorithms – including linear regression – will automatically adapt to non-numeric inputs.

Handling Non-Numeric Input Attributes (cont.)

- There are algorithms for numeric estimation that are designed to handle both numeric and nominal attributes.

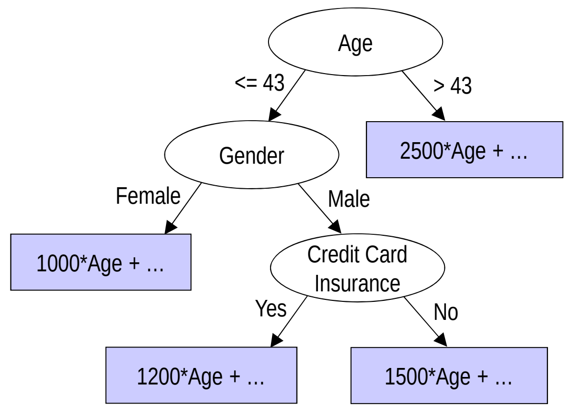
- Example problem: predict a customer's salary based on Age, Gender, and whether they purchased Credit Card Insurance

- One option: a *regression tree*
  - build a decision tree
  - each classification is a single number
    - the average output value for the training examples in that subgroup



Handling Non-Numeric Input Attributes (cont.)

- Another option: a *model tree*
  - each classification is a regression equation
    - based on just the training examples in that subgroup



Regression and Model Trees 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

Association Learning

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

Recall: Association Learning

- Algorithms for association learning:
  - take a set of training examples
  - discover associations/relationships among attributes
    - example: products that people tend to purchase together
- It does *not* single out a single attribute for special treatment.
  - there is no distinction between input and output attributes

Association Rules

- The learned associations are usually expressed as rules known as *association rules*. Examples:

```
if PurchaseDiapers = Yes
then PurchaseBeer = Yes

if PurchaseMilk = Yes and PurchaseJuice = Yes
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:

  `if` *precondition*
  `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:

```
if PurchaseDiapers = Yes
then PurchaseBeer = Yes
```

  its converse is:

```
if PurchaseBeer = Yes
then PurchaseDiapers = Yes
```

- The converse of a rule is <u>not</u> necessarily true.
  - example: this rule is true:

```
if name = 'Perry Sullivan'
then yearBorn = 2000
```

  its converse is not!

```
if yearBorn = 2000
then name = 'Perry Sullivan'
```

Example Problem: Credit-Card Promotions

- We'll use these training examples, which omit the Age attribute:

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

- Possible association rules include:

```
if Gender = Male and IncomeRange = 40-50K
then CreditCardIns = No and LifeIns = No

if CreditCardIns = Yes and LifeIns = Yes
then Gender = Male
```

Metric #1: Support

- The *support* of a rule is the number of training examples with the values in both the rule's precondition *and* conclusion.
  - the number of examples that the rule gets right
- This metric can also be expressed as a percentage of the total number of training examples.

Metric #1: Support

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	**40–50K**	**No**	**No**
Female	30–40K	No	Yes
Male	**40–50K**	**No**	**No**
Male	30–40K	**Yes**	**Yes**
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	**Yes**	**Yes**
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	**40–50K**	**No**	**No**
Female	20–30K	Yes	Yes

```
if Gender = 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 Gender = 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:

$$\text{confidence} = \frac{\underbrace{\text{\# examples with the values in the precondition } \textit{and} \text{ the conclusion}}_{\text{the support}}}{\text{\# examples with the values in just the precondition}}$$

Metric #2: Confidence

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

```
if Gender = Male and IncomeRange = 40-50K
then CreditCardIns = No and LifeIns = No
```

- confidence = $\frac{\text{\# examples with all four values}}{\text{\# examples with Gender=Male and IncRange=40-50K}}$

  $= \frac{3}{3}$

  = 1 or 100% (perfect accuracy on training examples)

Metric #2: Confidence

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

```
if CreditCardIns = Yes and LifeIns = Yes
then Gender = Male
```

- confidence = $\frac{\text{\# examples with all three values}}{\text{\# examples with CreditCardIns=Yes, LifeIns=Yes}}$

  $= \frac{2}{3}$

  = 0.667 or 66.7%

Practice: Support and Confidence

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

```
if LifeIns = Yes
then Gender = Female and CreditCardIns = No
```

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 Gender = 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

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	**Yes**	**Yes**
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	**Yes**	**Yes**
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	**Yes**	**Yes**

- An *item set* is a collection of attribute values that appears in one or more training examples.
  - example: the item set `CreditCardIns=Yes, LifeIns=Yes` appears in 3 training examples
  - it could be used to form two different rules with support = 3:
    - `if CreditCardIns = Yes then LifeIns = Yes`
    - `if LifeIns = Yes then CreditCardIns = Yes`

Apriori Algorithm

- The standard algorithm for learning association rules is called the *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
     - allows us to ensure that the rules formed from those item sets will have sufficient ***support***

  2) form rules from the item sets
     - keeping the ones with a high enough ***confidence*** value

First Stage: Building Item Sets

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

- Assume we want item sets that appear in >= 3 examples.
- We get 9 one-item sets that meet this criterion:

```
Gender=Male              CreditCardIns=Yes
Gender=Female            CreditCardIns=No
IncomeRange=20-30K       LifeIns=Yes
IncomeRange=30-40K       LifeIns=No
IncomeRange=40-50K
```

  - everything but `IncomeRange=50-60K`, which is in only 2 examples

From One-Item Sets to Two-Item Sets

- When considering two-item sets, we limit ourselves to ones that can be formed by *combining one-item sets with enough support.*
  - recall: here are the one-item sets with enough support:

```
Gender=Male              CreditCardIns=Yes
Gender=Female            CreditCardIns=No
IncomeRange=20-30K       LifeIns=Yes
IncomeRange=30-40K       LifeIns=No
IncomeRange=40-50K
```

  - we would combine them to get the possible two-item sets:

```
Gender=Male, IncomeRange=20-30K
Gender=Male, IncomeRange=30-40K
Gender=Male, IncomeRange=40-50K
Gender=Male,CreditCardIns=Yes
Gender=Male,CreditCardIns=No
Gender=Male,LifeIns=Yes
Gender=Male,LifeIns=No
Gender=Female,LifeIns=20-30K
Gender=Female,LifeIns=30-40K
... (30 possible 2-item sets in all!)
```

From One-Item Sets to Two-Item Sets (cont.)

- We *don't* consider two-item sets that include a one-item set that doesn't have enough support on its own.
- Example:
  - `IncomeRange=50-60K` doesn't have enough support
    - it only appears in 2 training examples
  - thus, we *don't* need to consider two-item sets like
    `Gender=Male, IncomeRange=50-60K`
    that include it.
  - these two-item sets can't possibly appear more often than the one-item set does!

From One-Item Sets to Two-Item Sets (cont.)

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	**30–40K**	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	**30–40K**	Yes	Yes
Male	**20–30K**	No	No
Male	**30–40K**	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	**20–30K**	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

- We test each possible two-item set and only keep the ones with enough support.
  - example: we *don't* keep `Gender=Male, IncomeRange=20-30K` because it only appears in 2 examples
  - example: we *do* keep `Gender=Male, IncomeRange=30-40K` because it appears in at least 3 examples

From One-Item Sets to Two-Item Sets (cont.)

- We end up with 15 two-item sets with enough support:

```
Gender=Male, IncomeRange=30-40K
Gender=Male, IncomeRange=40-50K
Gender=Male, CreditCardIns=No
Gender=Male, LifeIns=Yes
Gender=Male, LifeIns=No
Gender=Female, CreditCardIns=No
Gender=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.***
  - we'll see why later!

From Two-Item Sets to Three-Item Sets

- To form candidate three-item sets, we take the union of pairs of two-item sets that have one item in common:

  `Gender=Male, IncomeRange=30-40K` ∪ `Gender=Male, LifeIns=Yes`
  = `Gender=Male, IncomeRange=30-40K, LifeIns=Yes`

- Once again, we limit ourselves to combinations of two-item sets *with enough support*.

- We *don't* need to consider combinations of two-item sets that don't have enough support on their own.
  - example:
    - `Gender=Male, CreditCardIns=Yes` only appears in 2 training examples
    - thus, we *don't* need to consider three-item sets like `Gender=Male, CreditCardIns=Yes, LifeIns=Yes` that include it.

From Two-Item Sets to Three-Item Sets (cont.)

- In addition, we can limit the possible three-item sets even further by only combining two-item sets with the same *first item*.

  `Gender=Male, IncomeRange=30-40K` ∪ `Gender=Male, LifeIns=Yes`
  `= Gender=Male, IncomeRange=30-40K, LifeIns=Yes`

- It isn't necessary to consider other pairs of two-item sets.
  - example: although we could do

    `Gender=Male, LifeIns=Yes` ∪ `CreditCardIns=No, LifeIns=Yes`
    `= Gender=Male, CreditCardIns=No, LifeIns=Yes`

    we don't need to, because either:

    1) the resulting item set will already be generated by two other item sets, S1 and S2, with the same first item:

    `Gender=Male, CreditCardIns=No` ∪ `Gender=Male, 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?

```
Gender=Male, IncomeRange=30-40K
Gender=Male, IncomeRange=40-50K
Gender=Male, CreditCardIns=No
IncomeRange=30-40K, CreditCardIns=No
IncomeRange=30-40K, LifeIns=Yes
```

A. `Gender=Male, IncomeRange=30-40K, CreditCardIns=No`

B. `Gender=Male, IncomeRange=30-40K, LifeIns=Yes`

C. `IncomeRange=30-40K, CreditCardIns=No, LifeIns=Yes`

D. two of the above

E. all three of the above

Note The Savings!

- For our dataset, there are 56 three-item sets in all:

Gender=Female, Income=20-30K, CredCardIns=Yes
Gender=Female, Income=20-30K, CredCardIns=No
Gender=Female, Income=20-30K, LifeIns=Yes
Gender=Female, Income=20-30K, LifeIns=No
Gender=Female, Income=30-40K, CredCardIns=Yes
Gender=Female, Income=30-40K, CredCardIns=No
Gender=Female, Income=30-40K, LifeIns=Yes
Gender=Female, Income=30-40K, LifeIns=No
Gender=Female, Income=40-50K, CredCardIns=Yes
Gender=Female, Income=40-50K, CredCardIns=No
Gender=Female, Income=40-50K, LifeIns=Yes
Gender=Female, Income=40-50K, LifeIns=No
Gender=Female, Income=50-60K, CredCardIns=Yes
Gender=Female, Income=50-60K, CredCardIns=No
Gender=Female, Income=50-60K, LifeIns=Yes
Gender=Female, Income=50-60K, LifeIns=No
Gender=Female, CredCardIns=Yes, LifeIns=Yes
Gender=Female, CredCardIns=Yes, LifeIns=No
Gender=Female, CredCardIns=No, LifeIns=Yes
Gender=Female, CredCardIns=No, LifeIns=No
Income=20-30K, CredCardIns=Yes, LifeIns=Yes
Income=20-30K, CredCardIns=Yes, LifeIns=No
Income=20-30K, CredCardIns=No, LifeIns=Yes
Income=20-30K, CredCardIns=No, LifeIns=No
Income=30-40K, CredCardIns=Yes, LifeIns=Yes
Income=30-40K, CredCardIns=Yes, LifeIns=No
Income=30-40K, CredCardIns=No, LifeIns=Yes
Income=30-40K, CredCardIns=No, LifeIns=No
Gender=Male, Income=20-30K, CredCardIns=Yes
Gender=Male, Income=20-30K, CredCardIns=No
Gender=Male, Income=20-30K, LifeIns=Yes
Gender=Male, Income=20-30K, LifeIns=No
Gender=Male, Income=30-40K, CredCardIns=Yes
Gender=Male, Income=30-40K, CredCardIns=No
Gender=Male, Income=30-40K, LifeIns=Yes
Gender=Male, Income=30-40K, LifeIns=No
Gender=Male, Income=40-50K, CredCardIns=Yes
Gender=Male, Income=40-50K, CredCardIns=No
Gender=Male, Income=40-50K, LifeIns=Yes
Gender=Male, Income=40-50K, LifeIns=No
Gender=Male, Income=50-60K, CredCardIns=Yes
Gender=Male, Income=50-60K, CredCardIns=No
Gender=Male, Income=50-60K, LifeIns=Yes
Gender=Male, Income=50-60K, LifeIns=No
Gender=Male, CredCardIns=Yes, LifeIns=Yes
Gender=Male, CredCardIns=Yes, LifeIns=No
Gender=Male, CredCardIns=No, LifeIns=Yes
Gender=Male, CredCardIns=No, LifeIns=No
Income=40-50K, CredCardIns=Yes, LifeIns=Yes
Income=40-50K, CredCardIns=Yes, LifeIns=No
Income=40-50K, CredCardIns=No, LifeIns=Yes
Income=40-50K, CredCardIns=No, LifeIns=No
Income=50-60K, CredCardIns=Yes, LifeIns=Yes
Income=50-60K, CredCardIns=Yes, LifeIns=No
Income=50-60K, CredCardIns=No, LifeIns=Yes
Income=50-60K, CredCardIns=No, LifeIns=No

- By limiting ourselves to ones that can be formed by combining two-item sets *with enough support* and *the same first item*, we only need to test 11 of the 56!

From Two-Item Sets to Three-Item Sets (cont.)

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

- Out of the 11 potential three-items sets, only 5 have sufficient support – appearing in at least 3 examples:

```
Gender=Male, IncomeRange=40-50K, CreditCardIns=No
Gender=Male, IncomeRange=40-50K, LifeIns=No
Gender=Male, CreditCardIns=No, LifeIns=No
Gender=Female, CreditCardIns=No, LifeIns=Yes
IncomeRange=40-50K, CreditCardIns=No, LifeIns=No
```

From Three-Item Sets to Four-Item Sets

```
Gender=Male, IncomeRange=40-50K, CreditCardIns=No
Gender=Male, IncomeRange=40-50K, LifeIns=No
Gender=Male, CreditCardIns=No, LifeIns=No
Gender=Female, CreditCardIns=No, LifeIns=Yes
IncomeRange=40-50K, CreditCardIns=No, LifeIns=No
```

- To form potential four-item sets, we take the union of pairs of surviving three-item sets <u>with the same first two items</u>.
  - 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:

```
Gender=Male, IncomeRange=40-50K, CreditCardIns=No, LifeIns=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 item sets!

Results of the First Stage

- Here are all item sets with two or more items and support >= 3:

```
Gender=Male, IncRange=30-40K        Gender=Female, CreditCardIns=No
Gender=Male, IncRange=40-50K        Gender=Female, LifeIns=Yes
Gender=Male, CredCardIns=No         CredCardIns=Yes, LifeIns=Yes
Gender=Male, LifeIns=Yes            CredCardIns=No, LifeIns=Yes
Gender=Male, LifeIns=No             CredCardIns=No, LifeIns=No
IncRange=20-30K, CredCardIns=No
IncRange=30-40K, CredCardIns=No
IncRange=30-40K, LifeIns=Yes
IncRange=40-50K, CredCardIns=No
IncRange=40-50K, LifeIns=No

Gender=Male, IncRange=40-50K, CredCardIns=No
Gender=Male, IncRange=40-50K, LifeIns=No
Gender=Male, CredCardIns=No, LifeIns=No
Gender=Female, CredCardIns=No, LifeIns=Yes
IncRange=40-50K, CredCardIns=No, LifeIns=No

Gender=Male, IncRange=40-50K, CredCardIns=No, LifeIns=No
```

Second Stage: Forming the Rules

- A given item set can produce a number of potential rules.
  - example: the item set
    `Gender=Male, IncomeRange=40-50K, CreditCardIns=No`
    produces the following potential rules:

```
a) if Gender=Male and IncomeRange=40-50K
   then CreditCardIns=No
b) if Gender=Male and CreditCardIns=No
   then IncomeRange=40-50K
c) if IncomeRange=40-50K and CreditCardIns=No
   then Gender=Male
d) if Gender=Male
   then IncomeRange=40-50K and CreditCardIns=No
e) if IncomeRange=40-50K
   then Gender=Male and CreditCardIns=No
f) if CreditCardIns=No
   then Gender=Male and IncomeRange=40-50K
```

- We keep only the ones with confidence >= some min value.

Practice: Taking the Union of Item Sets

- What possible three-item sets could we form from the following two-item sets?

```
Gender=Female, IncomeRange=20-30K
Gender=Male, IncomeRange=40-50K
Gender=Male, LifeIns=No
IncomeRange=40-50K, CreditCardIns=Yes
IncomeRange=20-30K, LifeIns=No
IncomeRange=20-30K, LifeIns=Yes
```

A. `Gender=Female, IncomeRange=20-30K, LifeIns=No`

B. `Gender=Male, IncomeRange=40-50K, CreditCardIns=Yes`

C. `Gender=Male, IncomeRange=40-50K, LifeIns=No`

D. `IncomeRange=20-30K, LifeIns=No, LifeIns=Yes`

E. two or more of the above

Which of these rules would be kept?

Gender	Income Range	Credit Card Insurance	Life Insurance Promotion
Male	40–50K	No	No
Female	30–40K	No	Yes
Male	40–50K	No	No
Male	30–40K	Yes	Yes
Female	50–60K	No	Yes
Female	20–30K	No	No
Male	30–40K	Yes	Yes
Male	20–30K	No	No
Male	30–40K	No	No
Female	30–40K	No	Yes
Female	40–50K	No	Yes
Male	20–30K	No	Yes
Female	50–60K	No	Yes
Male	40–50K	No	No
Female	20–30K	Yes	Yes

Assume that we require a minimum confidence of 1.0

A. `if Gender=Male and IncRange=40-50K`
`then CreditCardIns=No`
support = ?
confidence = ?

B. `if Gender=Male and CreditCardIns=No`
`then IncomeRange=40-50K`
support = ?
confidence = ?

C. both rules would be kept

D. neither rule would be kept

Second Stage: Forming the Rules (cont.)

- In our example, there are 13 rules with conf = 1.0:

```
1) if LifeIns=No
   then CredCardIns=No
2) if Gender=Male and LifeIns=No
   then CredCardIns
3) if IncomeRange=40-50K
   then CredCardIns=No
4) if Gender=Male and IncRange=40-50K
   then CredCardIns=No and LifeIns=No
5) if IncRange=40-50K and LifeIns=No
   then Gender=Male and CredCardIns=No
6) if Gender=Male and IncRange=40-50K and CredCardIns=No
   then LifeIns=No
7) if Gender=Male and IncRange=40-50K and LifeIns=No
   then CredCardIns=No
8) if IncRange=40-50K and CredCardIns=No and LifeIns=No
   then Gender=Male
```

(continued)

Second Stage: Forming the Rules (cont.)

- 13 rules (cont.)

```
 9. if Income Range=40-50K and LifeIns=No
    then CreditCardIns=No
10. if Gender=Male and IncomeRange=40-50K
    then LifeInsPromo=No
11. if IncomeRange=40-50K and LifeInsPromo=No
    then Gender=Male
12. if Gender=Male and IncomeRange=40-50K
    then CreditCardIns=No
13. if CreditCardIns=Yes
    then LifeIns=Yes
```

Managing the Efficiency of the Algorithm

- The apriori algorithm tries to generate the rules *efficiently* – i.e., taking as few steps as possible.
- We've already seem 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

Simple Discretization Methods

Computer Science 105
Boston University

David G. Sullivan, Ph.D.

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 (35 + 65)/2 = 50

    final bins: (-inf, 9.5], (9.5, 50], (50, 83], (83, 89], (89, inf)
- Problems?

Discretization Example

- Let's say we have 8 training examples with the following values for Age:

  17, 23, 35, 41, 51, 58, 70, 89

  We want to discretize Age into 4 bins.

Which bins would be given by equal-height disc.?

- Let's say we have 8 training examples with the following values for Age:

  17, 23, 35, 41, 51, 58, 70, 89

  We want to discretize Age into 4 bins.

A. (-infinity, 29], (29, 46], (46, 64], (64, infinity)

B. [17, 29], (29, 46], (46, 64], (64, 89]

C. (-infinity, 35], (35, 53], (53, 71], (71, infinity)

D. [17, 35], (35, 53], (53, 71], (71, 89]

Preparing Your 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

- The data for a given entity (e.g., a customer) may be:
  - spread over multiple tables
  - spread over multiple rows within a given table
- To prepare for data mining, we need to *denormalize* the data.
  - multiple rows for a given entity → a single row

Denormalization

- Example: finding associations between courses students take.

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Course

name	*start_time*	*end_time*	*…*
CS 105	13:00:00	14:00:00	…
CS 111	09:30:00	11:00:00	…
EN 101	11:00:00	12:30:00	…
CS 460	16:00:00	17:30:00	…
CS 510	12:00:00	13:30:00	…
PH 101	14:30:00	16:00:00	…

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

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

Recall: Simple Discretization Methods

- We've discussed two methods for discretization.
- *Equal-width binning* divides the range of possible values into N subranges of the same size.
- *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.

Discretization Example

- Let's say we have 8 training examples with the following values for Age:

  17, 23, 35, 41, 51, 58, 70, 89

  We want to discretize Age into 4 bins.

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 *useEqualFrequency* to *True*
- Another option: *Discretize* in *filters/supervised/attribute* folder
  - attempts to learn *meaningful* cutoffs, based on your output

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: using numeric codes for Diagnosis
    - 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

Solutions to earlier discretization exercise

- Let's say we have 8 training examples with the following values for Age:

  17, 23, | 35, 41, | 51, 58, | 70, 89

  We want to discretize Age into 4 bins.

- To select a boundary cutoff for equal-height, choose the value halfway between the training examples on either side.
  - examples: (23 + 35)/2 = 29 (41 + 51)/2 = 46

A. **(-infinity, 29], (29, 46], (46, 64], (64, infinity)**

B. [17, 29], (29, 46], (46, 64], (64, 89]

C. (-infinity, 35], (35, 53], (53, 71], (71, infinity) ←

D. [17, 35], (35, 53], (53, 71], (71, 89]

equal-width bins
max – min
= 89 – 17 = 72
72 / 4 = 18
17 + 18 = 35
35 + 18 = 53
53 + 18 = 71

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 3 months after surgery
        - including whether he/she has been able to return to work

Overview of the Data-Mining Task

- Goal: to develop insights into factors that influence patient outcomes.
  - in particular, whether someone can return to work (yes/no)
  - in other words, to determine factors will allow us to predict whether or not the patient will return to work
- What type of data mining is most appropriate?
- What will the data mining produce?

Preparing the Data: Using a Spreadsheet

- Excel/Google Sheets/etc. can be used for several purposes:
  - removing unnecessary/problematic columns
  - making sure the output column is the last column
  - giving each column a simple, descriptive name
    `Gross Domestic Product` → `GDP`
    `Code` → `ReturnToWork`
  - ***removing problematic characters***
    - do a search-and-replace for:
      single quotes (`'`)
      double quotes (`"`)
      commas (`,`)
      and replace each type of character with nothing
- Use *File -> Save As* to save the file as CSV.

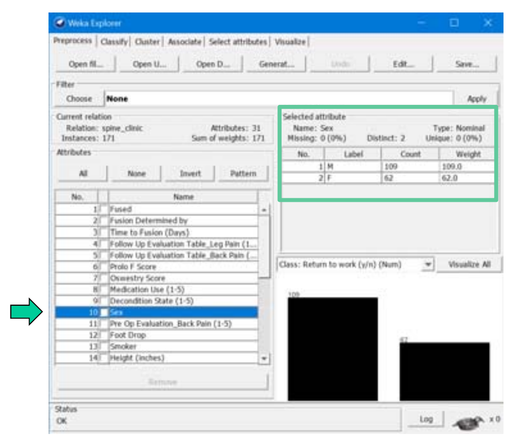
Review: Preparing the Data

- Other possible steps include:
  - denormalization
    several rows for a given entity → single training example
  - discretization
    numeric → nominal
  - nominal → numeric
  - force Weka to realize that a seemingly numeric attribute is really nominal

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 to find formatting issues that must be addressed



How many attributes should be removed/transformed?

Patient ID	Sex	# of Levels	Smoker (y/n)	Patient Type	Age		Return to Work (y/n)
1005	M	1	0	3100	30–39		1
1013	F	2	1	1400	50–59		0
1245	M	1	1	3100	20–29	. . .	1
2110	F	3	0	2500	30–39		0
1001	F	2	1	1400	40–49		1

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
- 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: 1R
  - you can also use 0R to determine what % of your training data has the most common class value
- 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
- We use cross validation when exploring possible models, because it gives a sense of how well the model will generalize.
- Note: the model reported in the output window is learned from <u>all</u> of the training examples.
  - the cross-validation results do not actually evaluate it

Reporting the Results

- You should <u>**not**</u> report the cross-validation results.
- Once you find the models with the best cross-validation results, you should evaluate them two ways.
- **On the full set of training data**:
  - select *Using training set* in the *Test* box of the *Classify* tab
  - rerun the algorithm
- **On the reserved test 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 from both training and test results:
  - 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?
  - for numeric estimation:
    - which attributes have positive coefficients?
    - which have negative?
    - remember: the *magnitude* of the coefficients may <u>not</u> be significant
  - **are the models intuitive? why or why not?**