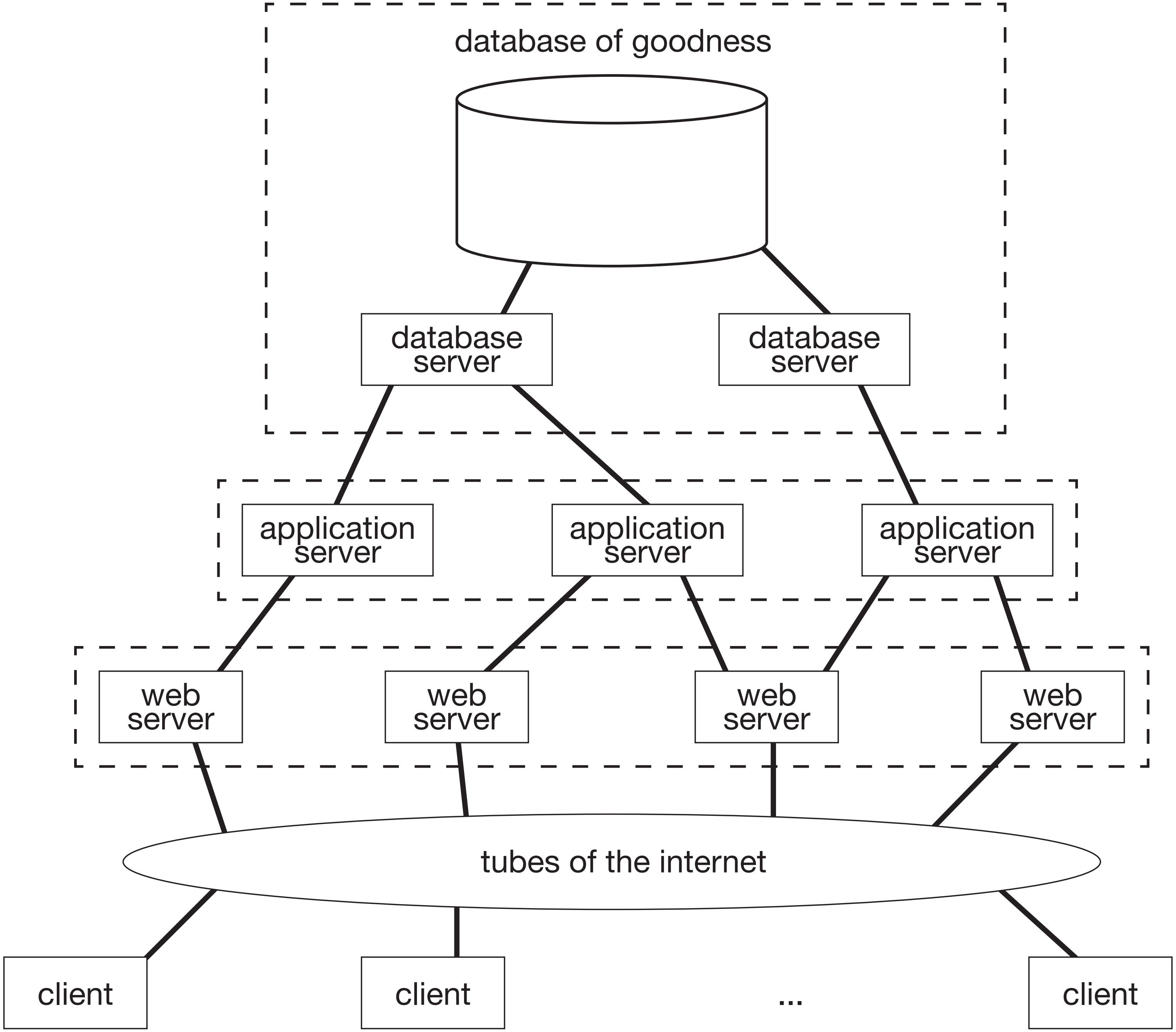
CSC 370

Programmatic access to a DBMS

Present coding reality

- Many production systems are multi-tiered
  - Front-end: web-server tier
  - Middle-tier: application tier
  - Bottom-tier: database tier
- At this point in the course we are interested in how to access a DBMS programmatically
  - The result is not necessarily sexy...
  - ... as it often involves a mix of SQL and regular code.
  - However, what we will use will hopefully look better than PHP!


database of goodness
database server
database server
application server
application server
application server
web server
web server
web server
web server
tubes of the internet
client
client
...
client

Web-server tier

- Manages interactions with the user
- Often running Apache/Tomcat
  - Helps handle details such as session management
  - Users connecting to the webserver via their browser are **clients**
- Example: www.amazon.ca
  - User opens a connection to the Amazon database by first entering the URL of the site
  - Webserver presents a view of the system (home page) which is rendered by the user's browser
  - Forms, menus, buttons enable client to express what they want accomplished
  - Client's browser transmits the information to the webserver
  - Webserver must then negotiate with the next tier (application-server tier)

Application-server tier

- Turns data (from database) into a response to a webserver request (ultimately from the client)
- A web-server process can invoke one or more application-tier processes to handle the request
  - Actions performed by application tier: sometimes called business logic
  - In essence, application server is designed to respond to a customer's request and implement the company's strategy for handling that request
- Amazon.ca again: Possible response are...
  - Show the user's wishlist
  - Fetch information about a book
  - Add book to a cart
- May also contain a subtier ("database integration")

Database tier

- Executes queries requested from the application-server tier
  - May also buffer data needed to handle some requests
- In a large system, database tier may consist of many servers
  - Query load is distributed across the servers
  - System must ensure association between application server and database-server instance is maintained
- Connections must be established to a database server...
  - ... and these are expensive (in time) to establish.
  - Therefore some management of connections (pools) is often used.

Our interest at this point

- We are concerned now with how to implement the database tier
  - Implementing the webserver and application tiers is a topic for a course such as SENG 450.
  - Note: some programming frameworks help combine work on all three tiers into one package (e.g., Django, Ruby on Rails)
- For now, we need to learn:
  1. How does a database interact with regular programs?
  2. How can we deal with data-type differences between SQL and our chosen language?
  3. How do we manage connections to short-lived processes?

psycopg2 (note spelling – no "h"!)

- Database adapter for Python
  - There are several, but this one is well use.
  - Is also available on the lab machines.
  - Chat with instructor if you need more information on how to install it on your own machines.
- Literally a wrapper...
  - ... as it is a way of accessing the official libpq Postgres client library.
- We will use it to access your database on studentdb.csc.uvic.ca
- Can of course be used to access other PostgreSQL servers
  - But remember that any connections to studentdb.csc.uvic.ca must originate from within the Faculty of Engineering's network.

psycopg2: Our "hello, world!"

```
#!/usr/bin/python

import psycopg2

def main():
    dbconn = psycopg2.connect(host='studentdb.csc.uvic.ca', user='c370_s63',
        password='fS8qPPLc')
    cursor = dbconn.cursor()

    cursor.execute("""
    select *
    from sells
    where price < 5
    """)

    for row in cursor.fetchall():
        print "%s %s %s" % (row[0], row[1], row[2])

    cursor.close()
    dbconn.close()


if __name__ == "__main__": main()
```

psycopg2: Our "hello, world!" output

```
$ ./hello_world.py
Bard & Banker        Blue                 3.5
The Hacked Library   Blue                 3.25
The Hacked Library   Bud Light            4.0
```

Python and database APIs

- Python has a mixed reputation with database folks
  - Language is great to use...
  - ... but adapter support has not been consistent.
- However:
  - There is a standard Python database API
  - (PEP 249 – now at version 2.0)
  - Standard describes what is needed in an adapter.
  - Must also be usable by all major DBMS implementation (i.e., might not be ideally tuned for Postgres)
- We will be looking at some of the simpler functionality
  - And don't forget – Postgres permits us to create stored procedures, so not everything need be done via Python.

A word about driver independence

- Often best to write code that is not specific to a DBMS or driver
  - Makes it easier when porting the program to a different platform
  - Makes it easier when porting program to use a different DBMS
- Relatively straightforward to do this in Python

Accessing driver outside of "import"

```
#!/usr/bin/python

DBTYPE = 'psycopg2'

def main():
    dbdriver = __import__(DBTYPE, globals(), locals(), [], -1)

    dbconn = dbdriver.connect(...)

# etc. etc. etc.
```

Setting up a connection

- Everything that holds for connecting to the studentdb server holds for connecting via Python
  - Must be connecting from a machine in the Faculty of Engineering network
  - Provide username and password
- There are 2.5 different ways to set up the connection object
  - Using libpq connection string
  - Using keyword arguments

Connection approaches

```
# libpq-ish string

    dbconn = psycopg2.connect("host=studentdb.csc.uvic.ca user=c370_s63 "
        + "password='fS8qPPLc'")

# libpq-ish with explicitly named argument

    dbconn = psycopg2.connect(dsn="host=studentdb.csc.uvic.ca user=c370_s63 "
        + "password='fS8qPPLc'")

# keyword arguments
# (but not everything libpq is necessarily supported by psycopg2

 dbconn = psycopg2.connect(host='studentdb.csc.uvic.ca',
                 user='c370_s63',
                 password='fS8qPPLc')


# For more information on libpq (the C interface to Postgres) go to
# http://www.postgresql.org/docs/9.1/static/libpq.html
```

Cursors

- Some mechanism is needed to permit programmatic access to the results of a query
  - The query result is a relation...
  - ... which is made up of rows with attributes ...
  - ... although there is no guarantee these rows easily map to a program data structure
- To provide consistency, database-interface languages support the concept of a **cursor**
  - A control structure the enables the code to traverse of the rows in the relation.
  - In essence it supports operations such as retrieval, addition and deletion of tuples / table rows.
  - (Acts a bit like an iterator)
  - (How much data should be transferred?)

Cursor: style 1

- A cursor is associated with a connection for the life of that connection
- If unnamed, the cursor object really isn't a cursor object
  - Rather, it is handle to data stored within the Python program (a "statement handle")
  - That is, all the data is transferred from the server to the client!

```
cursor = dbconn.cursor()
```

Cursor: style 2

- **server-side cursor**
  - Controls that amount of data transferred from server to client.
  - Very useful when there is a large amount of data.
  - For this to work, our code will need to fetch data from the cursor from time to time.
- Also called a named cursor
- (Semantics can be a bit confusing when cursors and transactions mix...)

```
cursor = dbconn.cursor(name="mycursor')
```

Executing

- Cursors permit:
  - querying
  - update modifications
  - other Postgresql operations

```
cursor.execute("""
select pub, beer
from sells
where price < 5
""")

cursor.execute("update sells set price = 3.00 where beer = 'Blue'")

cursor.execute("analyze verbose sells")
```

Fetching results

- Traversing through the cursor results is known as fetching
  - fetchall
  - fetchmany
  - fetchone
- Beware: not everything can be fetched!

```
cursor.execute("""
select pub, beer
from sells
where price < 5
""")
result = cursor.fetchall()
print result

cursor.execute("analyze verbose sells")
result = cursor.fetchall()
print result
```

fetchall output: examples

```
$ ./slide15.py
[('Bard & Banker       ', 'Blue                ')), ('The Hacked Library  ', 'Blue
'), ('The Hacked Library  ', 'Bud Light           ')]

Traceback (most recent call last):
  File "./slide15.py", line 26, in <module>
    if __name__ == "__main__": main()
  File "./slide15.py", line 23, in main
    result = cursor.fetchall()
psycopg2.ProgrammingError: no results to fetch
```

Fetching results

- Iterating through each result row is straightforward...
  - ... but we do lose some information.
  - Attributes must be accessed by position in the row (i.e., not by their name).

```
cursor.execute("select pub, price, beer from sells")
for row in cursor.fetchall():
    print "Sells data: %s %s %s" % (row[0], row[1], row[2])
```

```
$ ./slide17.py
Sells data: Bard & Banker        3.5 Blue
Sells data: The Hacked Library   3.25 Blue
Sells data: The Hacked Library   7.0 Amnesiac
Sells data: The Hacked Library   4.0 Bud Light
```

More fetching techniques

- Fetch a single row
  - Make sure, however, that there was a row returned
  - (Unlike an iterator loop, need to use an "if")

```
cursor.execute("select pub, price, beer from sells")
row = cursor.fetchone()
if row is not None:
    print "Sells data: %s %s %s" % (row[0], row[1], row[2])
```

Fetching in batches

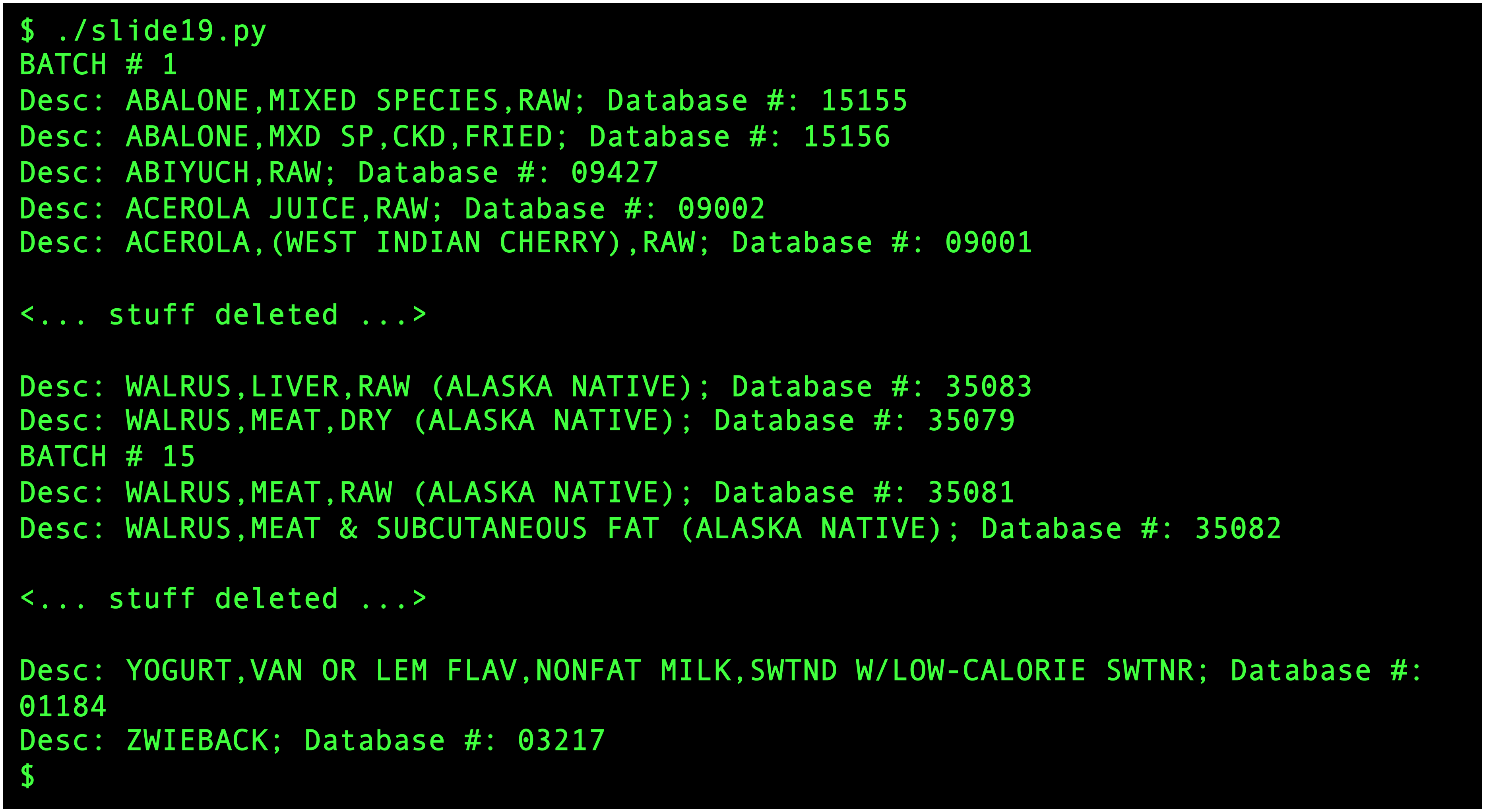
- If there is a lot of data at the server, we may only want to process a bit of it at a time
- We can control the maximum number of rows returned
  - Specify the maximum number of batch rows before we execute the SQL statement

```
cursor = dbconn.cursor()
cursor.arraysize = 500
cursor.execute("select shrt_desc, ndb_no from food_des order by shrt_desc")

batch_num = 1
while True:
    batch = cursor.fetchmany()
    if not batch:
        break
    print "BATCH # %d" % (batch_num)
    for row in batch:
        print "Desc: %s; Database #: %s" % (row[0], row[1])

    batch_num = batch_num + 1
```

fetchmany example: output



```
$ ./slide19.py
BATCH # 1
Desc: ABALONE,MIXED SPECIES,RAW; Database #: 15155
Desc: ABALONE,MXD SP,CKD,FRIED; Database #: 15156
Desc: ABIYUCH,RAW; Database #: 09427
Desc: ACEROLA JUICE,RAW; Database #: 09002
Desc: ACEROLA,(WEST INDIAN CHERRY),RAW; Database #: 09001

<... stuff deleted ...>

Desc: WALRUS,LIVER,RAW (ALASKA NATIVE); Database #: 35083
Desc: WALRUS,MEAT,DRY (ALASKA NATIVE); Database #: 35079
BATCH # 15
Desc: WALRUS,MEAT,RAW (ALASKA NATIVE); Database #: 35081
Desc: WALRUS,MEAT & SUBCUTANEOUS FAT (ALASKA NATIVE); Database #: 35082

<... stuff deleted ...>

Desc: YOGURT,VAN OR LEM FLAV,NONFAT MILK,SWTND W/LOW-CALORIE SWTNR; Database #:
01184
Desc: ZWIEBACK; Database #: 03217
$
```

Metadata

- Even though we must access attributes by row index...
  - ... we can obtain some table metadata from the cursor.
- Can use this metadata as appropriate.
  - Not a lot we can do with type_code besides compare for equality.

```
cursor.execute("select price, pub, beer from sells")
print cursor.description[0].name
print cursor.description[0].type_code
print cursor.description[2].name
print cursor.description[2].type_code
```

```
$ ./slide21.py
price
700
beer
1042
```