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 ML2, 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.
  
  ![Diagram](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

Algorithms

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

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

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

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

Don Knuth
Is This An Algorithm?

- Recipe for preparing a meat roast:

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

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

Here’s the Algorithm…

- Recipe for preparing a meat roast:

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

  (also from Pohl and McDowell)
Python

- We'll learn to write programs in Python.
- Python is a freely available language that makes it easy to write small- and medium-sized programs.
- You can download it from [http://www.python.org/download/](http://www.python.org/download/)
  - use any version of Python 3
- Python is also available on the lab machines.

Overview of the Programming Process

1. Analysis/Specification
2. Design
3. Implementation
4. Testing/Debugging
Example Problem: Adding Up Your Change

• Let's say that we have a bunch of coins of various types, and we want to figure out how much money we have.

• Let's begin the process of developing a program that does this.

Step 1: Analysis and Specification

• Analyze the problem (making sure that you understand it), and specify the problem requirements clearly and unambiguously.

• Describe exactly what the program will do, without worrying about how it will do it.

• Ask questions like the following:
  • what are the inputs to the program?
  • what are the desired outputs?
  • what needs to be done to go from the inputs to the outputs?
Step 2: Design

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

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

- Algorithms are often designed using pseudocode.
  - more informal than an actual programming language
  - allows us to avoid worrying about the syntax of the language
  - example for our change-adder problem:

    ```
    get the number of quarters
    get the number of dimes
    get the number of nickels
    get the number of pennies
    compute the total value of the coins
    output the total value
    ```

Step 3: Implementation

- Translate your design into the programming language.

  pseudocode → code

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

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

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

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

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

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

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

Overview of the Programming Process

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

- Python is an example of a high-level language.
  - others include: Java, C, C++, Fortran, Perl, and LISP
- They are designed to make it easier for programmers to write and read programs.
- However, a computer cannot run a high-level program directly.
- Instead, each type of CPU has its own machine language—its own set of low-level instructions that it understands.
  - they perform simple operations like adding two numbers
  - the instructions are written as binary numbers
    00000010100100011000000000100000

Converting from High to Low

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

```
high-level program

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

- The interpreter itself has been converted to machine language and is run directly by the CPU.
- Most interpreters – including the one for Python – allow you to execute commands from the keyboard, one at a time.
- Using an interpreter makes it easier to develop programs quickly.
Interacting with the Python Interpreter

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

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

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

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

  >>>

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

    >>> print('Hello world!')
    Hello world!
    >>> print(5 + 10)
    15
    >>> print(2 ** 10)
    1024
    >>> print(2 ** 100)
    1267650600228229401496703205376

Interacting with the Python Interpreter (cont.)
Creating a Reusable Program

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

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

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

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

Running a Program

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

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

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

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

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

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

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

Comments

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

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

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

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

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

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

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

**Identifiers**

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

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

- **Identifiers** are words that are used to name components of a Python program.
- They include:
  - variables, which give a name to a value
    quarters dimes nickels pennies cents
  - function names like `eval`, `input` and `print`
Identifiers (cont.)

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

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

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

Expressions

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

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

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

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

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

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

Expressions (cont.)

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

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

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

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

- Example: consider this code fragment:

```python
quarters = 10
dimes = 3
nickels = 7
pennies = 6

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

Statements

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

- A statement is an individual command in a program.

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

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

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

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

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

Print Statements (cont.)

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

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

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

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

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

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

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

Functions (cont.)

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

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

    ```python
    >>> abs(-20)
    20
    >>> abs(35)
    35
    ```

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

• We often need to combine functions in our programs.

• Example: to print the absolute value of a number

```python
num = -15
print("The absolute value is: ", abs(num))
```

• the value returned by abs is one of the parameters of print

• In such cases, the statement is executed "from the inside out":

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

```
abs(-15)
```

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

output: The absolute value is: 15

Assignment Statements

• We use an assignment statement to give a value to a variable.

• Syntax:

```
<variable> = <expression>
```

<variable> is a valid identifier.

= is the assignment operator; it does not compare the values!

• Examples:

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

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

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

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

Assignment Statements (cont.)

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

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

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

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

#### Examples:

<table>
<thead>
<tr>
<th>num1 = 100</th>
<th>num1</th>
<th>num2 = 120</th>
<th>num2</th>
</tr>
</thead>
<tbody>
<tr>
<td>num1 = 50</td>
<td>num1</td>
<td>num2 = 120</td>
<td>num2</td>
</tr>
<tr>
<td>num1 = num2 * 2</td>
<td>num1</td>
<td>num2 = 120</td>
<td>num2</td>
</tr>
<tr>
<td>sum = 13</td>
<td>sum</td>
<td>val = 30</td>
<td>val</td>
</tr>
<tr>
<td>sum = sum + val</td>
<td>sum</td>
<td>val</td>
<td></td>
</tr>
<tr>
<td>val = val * 2</td>
<td>sum</td>
<td>val</td>
<td></td>
</tr>
</tbody>
</table>

The value of **num1** is unchanged!
The Input Function

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

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

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

Getting Numeric Input

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

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

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

• Once again, we work from the inside out:
  \[
  \text{quarters} = \text{eval}(\text{input}(\text{"number of quarters? "}))  \\
  \text{eval}(\text{"17"})  \\
  17
  \]

Summary: Input Statements

• Getting a \textit{string value} from the user:
  \[
  \langle\text{variable}\rangle = \text{input}(\langle\text{prompt}\rangle)  \\
  \text{where } \langle\text{prompt}\rangle \text{ is a string}
  \]

• Getting a \textit{numeric value} from the user:
  \[
  \langle\text{variable}\rangle = \text{eval}(\text{input}(\langle\text{prompt}\rangle))  \\
  \text{where } \langle\text{prompt}\rangle \text{ is a string}
  \]

• Note: it looks better if you put a space at the end of the prompt.
  • example: if we changed the line from changeAdder.py to omit the space
    \[
    \text{quarters} = \text{eval}(\text{input}(\text{"number of quarters?"}))
    \]
    we will get the following when the user enters a value
    \[
    \text{number of quarters?}17
    \]
Review

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

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

for Statements

• A for statement allows us to repeat one or more statements.
  • example:
    ```python
    for i in [1, 2, 3]:
        print('Warning')
        print(i)
    ```
    the body of the loop

  will output:
  Warning
  1
  Warning
  2
  Warning
  3

• A for statement is often referred to as a for loop.
• The repeated statement(s) are known as the body of the loop.
  • they must be indented the same amount
for Statements (cont.)

• Syntax:

```python
for <variable> in <sequence>:
    <body>
```

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

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

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

Executing a for Loop

```python
for <variable> in <sequence>:
    <body of the loop>
```

- does the sequence have more values?
  - yes: assign the next value in the sequence to variable
    - execute the statements in the body
    - execute statement after the loop
  - no: continue with the statement that comes after the loop
Executing Our Earlier Example
(with one extra statement)

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

```
yes
no
```

Does [1, 2, 3] have more values?

- yes
  - assign the next value in the sequence to i
  - print('Warning')
  - print(i)
  - print('That's all.')
- no

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

Another Example

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

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

- \( \text{range}(n) \) generates the sequence 0, 1, 2, \ldots, \( n-1 \)

- Examples:
  - \( \text{range}(4) \) generates the sequence 0, 1, 2, 3
  - \( \text{range}(7) \) generates the sequence 0, 1, 2, 3, 4, 5, 6
  - \( \text{range}(2) \) generates the sequence 0, 1

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

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

- Example:

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

  output:
  
  Ho!
  Ho!
  Ho!

range Function and for Loops (cont.)

- Another example:

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

  output?
Printing Separate Values on the Same Line

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

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

- Example:
  ```python
  for i in range(7):
    print(i * 5, end=" ")
  ```
  will output
  ```
  0 5 10 15 20 25 30
  ```

Using a for Loop to Compute a Sum

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

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

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

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

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

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

Using a for Loop to Compute a Sum

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

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

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