Multi-Way Decisions

- The newest program is great, but it still has some quirks!
  This program finds the real solutions to a quadratic

  Please enter the coefficients (a, b, c): 1,2,1

  The solutions are: -1.0 -1.0
Multi-Way Decisions

- While correct, this method might be confusing for some people. It looks like it has mistakenly printed the same number twice!
- Double roots occur when the discriminant is exactly 0, and then the roots are \(-\frac{b}{2a}\).
- It looks like we need a three-way decision!
Multi-Way Decisions

- Check the value of discrim
  - when < 0: handle the case of no roots
  - when = 0: handle the case of a double root
  - when > 0: handle the case of two distinct roots

- We can do this with two if-else statements, one inside the other.
- Putting one compound statement inside of another is called nesting.
if discrim < 0:
    print "Equation has no real roots"
else:
    if discrim == 0:
        root = -b / (2 * a)
        print "There is a double root at", root
    else:
        # Do stuff for two roots
Multi-Way Decisions

- If `discrim < 0`, print "no roots".
- If `discrim >= 0`:
  - If `discrim == 0`, do double root.
  - If `discrim > 0`, do unique roots.
Multi-Way Decisions

- Imagine if we needed to make a five-way decision using nesting. The `if-else` statements would be nested four levels deep!
- There is a construct in Python that achieves this, combining an `else` followed immediately by an `if` into a single `elif`.
Multi-Way Decisions

- if `<condition1>`:
  `<case1 statements>`
  
elif `<condition2>`:
  `<case2 statements>`
  
elif `<condition3>`:
  `<case3 statements>`

...
Multi-Way Decisions

- This form sets of any number of mutually exclusive code blocks.
- Python evaluates each condition in turn looking for the first one that is true. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire `if-elif-else`.
- If none are true, the statements under `else` are performed.
Multi-Way Decisions

- The `else` is optional. If there is no `else`, it’s possible no indented block would be executed.
multi-way decisions

# quadratic4.py
# A program that computes the real roots of a quadratic equation.
# Illustrates use of a multi-way decision

import math

def main():
    print "This program finds the real solutions to a quadratic\n"

    a, b, c = input("Please enter the coefficients (a, b, c): ")

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print "\nThe equation has no real roots!"
    elif discrim == 0:
        root = -b / (2 * a)
        print "\nThere is a double root at", root
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print "\nThe solutions are: ", root1, root2
Study in Design: Max of Three

- Now that we have decision structures, we can solve more complicated programming problems. The negative is that writing these programs becomes harder!

- Suppose we need an algorithm to find the largest of three numbers.
def main():
    x1, x2, x3 = input("Please enter three values: ")

    # missing code sets max to the value of the largest
    print "The largest value is", max
Strategy 1: Compare Each to All

- This looks like a three-way decision, where we need to execute one of the following:
  
  ```
  max = x1
  max = x2
  max = x3
  ```

- All we need to do now is preface each one of these with the right condition!
Strategy 1: Compare Each to All

- Let’s look at the case where \( x_1 \) is the largest.
  
  ```python
  if x1 >= x2 >= x3:
    max = x1
  ```

- Is this syntactically correct?
  - Many languages would not allow this compound condition
  - Python does allow it, though. It’s equivalent to \( x_1 \geq x_2 \geq x_3 \).
Strategy 1: Compare Each to All

Whenever you write a decision, there are two crucial questions:

- When the condition is true, is executing the body of the decision the right action to take?
  - x1 is at least as large as x2 and x3, so assigning max to x1 is OK.
  - Always pay attention to borderline values!!
Strategy 1: Compare Each to All

- Secondly, ask the converse of the first question, namely, are we certain that this condition is true in all cases where x1 is the max?
  - Suppose the values are 5, 2, and 4.
  - Clearly, x1 is the largest, but does x1 ≥ x2 ≥ x3 hold?
  - We don’t really care about the relative ordering of x2 and x3, so we can make two separate tests: x1 >= x2 and x1 >= x3.
Strategy 1: Compare Each to All

- We can separate these conditions with and!

```python
if x1 >= x2 and x1 >= x3:
    max = x1
elif x2 >= x1 and x2 >= x3:
    max = x2
else:
    max = x3
```

- We’re comparing each possible value against all the others to determine which one is largest.
Strategy 1: Compare Each to All

- What would happen if we were trying to find the max of five values?
- We would need four Boolean expressions, each consisting of four conditions anded together.
- Yuck!
Strategy 2: Decision Tree

- We can avoid the redundant tests of the previous algorithm using a decision tree approach.
- Suppose we start with $x_1 \geq x_2$. This knocks either $x_1$ or $x_2$ out of contention to be the max.
- If the condition is true, we need to see which is larger, $x_1$ or $x_3$. 
Strategy 2: Decision Tree

```
1. Is \( \pi_1 \geq \pi_2 \)?
   - Yes: Set \( \text{max} = \pi_1 \)
   - No: Set \( \text{max} = \pi_3 \)

2. Is \( \pi_1 \geq \pi_3 \)?
   - Yes: Set \( \text{max} = \pi_1 \)
   - No: Set \( \text{max} = \pi_3 \)

3. Is \( \pi_2 \geq \pi_3 \)?
   - Yes: Set \( \text{max} = \pi_2 \)
   - No: Set \( \text{max} = \pi_3 \)
```
Strategy 2: Decision Tree

- if \( x_1 \geq x_2 \):
  - if \( x_1 \geq x_3 \):
    - \( \text{max} = x_1 \)
  - else:
    - \( \text{max} = x_3 \)
- else:
  - if \( x_2 \geq x_3 \):
    - \( \text{max} = x_2 \)
  - else
    - \( \text{max} = x_3 \)
Strategy 2: Decision Tree

- This approach makes exactly two comparisons, regardless of the ordering of the original three variables.

- However, this approach is more complicated than the first. To find the max of four values you’d need `if-else` nested three levels deep with eight assignment statements.
Strategy 3: Sequential Processing

- How would you solve the problem?
- You could probably look at three numbers and just know which is the largest. But what if you were given a list of a hundred numbers?
- One strategy is to scan through the list looking for a big number. When one is found, mark it, and continue looking. If you find a larger value, mark it, erase the previous mark, and continue looking.
Strategy 3: Sequential Processing
Strategy 3: Sequential Processing

- This idea can easily be translated into Python.

```python
max = x1
if x2 > max:
    max = x2
if x3 > max:
    max = x3
```
Strategy 3: Sequential Programming

- This process is repetitive and lends itself to using a loop.
- We prompt the user for a number, we compare it to our current max, if it is larger, we update the max value, repeat.
Strategy 3: Sequential Programming

# maxn.py
# Finds the maximum of a series of numbers

def main():
    n = input("How many numbers are there? ")

    # Set max to be the first value
    max = input("Enter a number >> ")

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = input("Enter a number >> ")
        if x > max:
            max = x

    print "The largest value is", max
Strategy 4: Use Python

- Python has a built-in function called `max` that returns the largest of its parameters.

```python
def main():
    x1, x2, x3 = input("Please enter three values: ")
    print "The largest value is", max(x1, x2, x3)
```
Some Lessons

- There’s usually more than one way to solve a problem.
  - Don’t rush to code the first idea that pops out of your head. Think about the design and ask if there’s a better way to approach the problem.
  - Your first task is to find a correct algorithm. After that, strive for clarity, simplicity, efficiency, scalability, and elegance.
Some Lessons

- Be the computer.
  - One of the best ways to formulate an algorithm is to ask yourself how you would solve the problem.
  - This straightforward approach is often simple, clear, and efficient enough.
Some Lessons

- Generality is good.
  - Consideration of a more general problem can lead to a better solution for a special case.
  - If the max of n program is just as easy to write as the max of three, write the more general program because it’s more likely to be useful in other situations.
Some Lessons

- Don’t reinvent the wheel.
  - If the problem you’re trying to solve is one that lots of other people have encountered, find out if there’s already a solution for it!
  - As you learn to program, designing programs from scratch is a great experience!
  - Truly expert programmers know when to borrow.