Vahé Karamian

Python Programming

CS-110

CHAPTER 7

Decision Structures

OBJECTIVES

- To understand the simple decision programming pattern and its implementation using a Python if statement.
- To understand the two-way decision programming pattern and its implementation using a Python if-else statement.
- To understand multi-way decision programming pattern and its implementation using a Python if-elif-else statement.
- To understand the idea of exception handling and be able to write simple exception handling code that catches standard Python run-time errors.

OBJECTIVES

- To understand the concept of Boolean expressions and the bool data type.
- To be able to read, write, and implement algorithms that employ decision structures, including those that employ sequences of decisions and nested decision structures.

Simple Decisions

- So far, we've viewed programs as sequences of instructions that are followed one after the other.
- While this is a fundamental programming concept, it is not sufficient in itself to solve every problem. We need to be able to alter the sequential flow of a program to suit a particular situation.

Simple Decisions

- *Control structures* allow us to alter this sequential program flow.
- In this chapter, we'll learn about *decision structures*, which are statements that allow a program to execute different sequences of instructions for different cases, allowing the program to "choose" an appropriate course of action.

Example: Temperature Warnings

- Let's return to our Celsius to Fahrenheit temperature conversion program from Chapter 2.

```
# convert.py
#      A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = input("What is the Celsius temperature? ")
    fahrenheit = 9.0 / 5.0 * celsius + 32
    print "The temperature is", fahrenheit, "degrees Fahrenheit."

main()
```

Example: Temperature Warnings

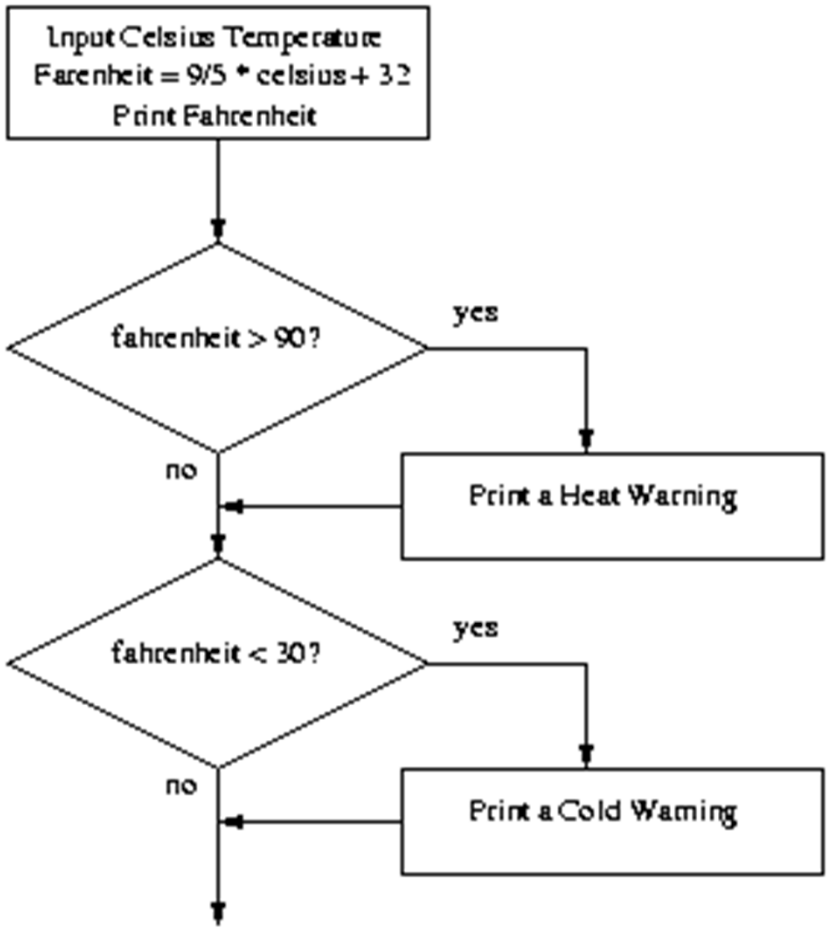
- Let's say we want to modify that program to print a warning when the weather is extreme.
- Any temperature over 90 degrees Fahrenheit and lower than 30 degrees Fahrenheit will cause a hot and cold weather warning, respectively.

Example: Temperature Warnings

- Input the temperature in degrees Celsius (call it celsius)
- Calculate fahrenheit as 9/5 celsius + 32
- Output fahrenheit
- If fahrenheit > 90
  print a heat warning
- If fahrenheit > 30
  print a cold warning

Example: Temperature Warnings

- This new algorithm has two *decisions* at the end. The indentation indicates that a step should be performed only if the condition listed in the previous line is true.


Input Celsius Temperature
Farenheit = 9/5 * celsius + 32
Print Fahrenheit
fahrenheit > 90?
yes
no
Print a Heat Warning
fahrenheit < 30?
yes
no
Print a Cold Warning

Example: Temperature Warnings

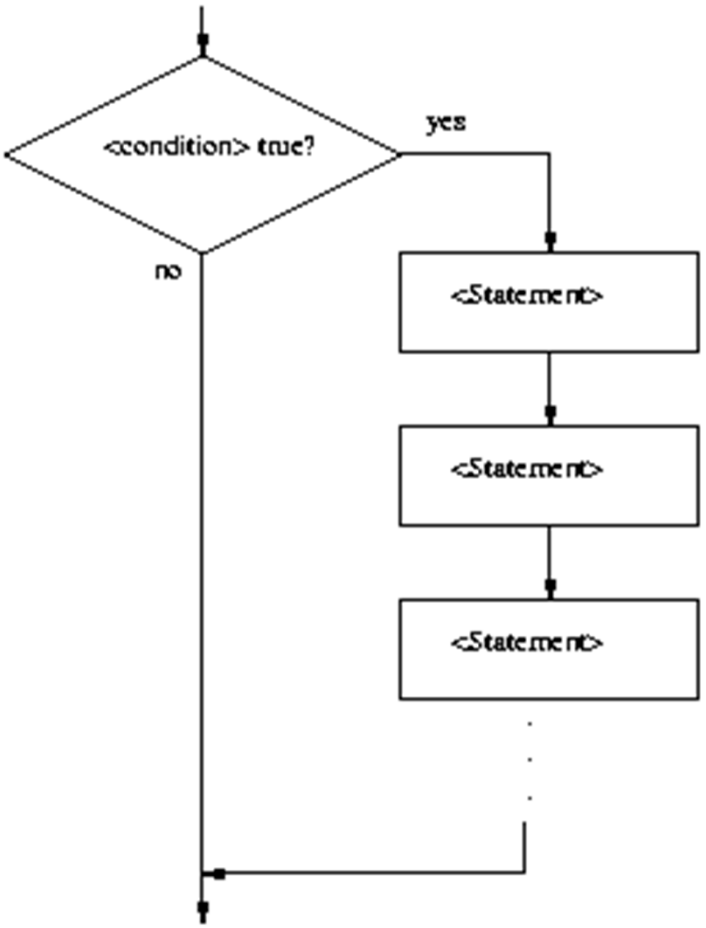
```
# convert2.py
#      A program to convert Celsius temps to Fahrenheit.
#      This version issues heat and cold warnings.

def main():
    celsius = input("What is the Celsius temperature? ")
    fahrenheit = 9.0 / 5.0 * celsius + 32
    print "The temperature is", fahrenheit, "degrees fahrenheit."
    if fahrenheit >= 90:
        print "It's really hot out there, be careful!"
    if fahrenheit <= 30:
        print "Brrrrr. Be sure to dress warmly"

main()
```

Example: Temperature Warnings

- The Python `if` statement is used to implement the decision.
- `if <condition>:`
  `<body>`
- The body is a sequence of one or more statements indented under the `if` heading.

Example: Temperature Warnings

- The semantics of the `if` should be clear.
  - First, the condition in the heading is evaluated.
  - If the condition is true, the sequence of statements in the body is executed, and then control passes to the next statement in the program.
  - If the condition is false, the statements in the body are skipped, and control passes to the next statement in the program.


<condition> true?
yes
no
<Statement>
<Statement>
<Statement>

Example: Temperature Warnings

- The body of the `if` either executes or not depending on the condition. In any case, control then passes to the next statement after the `if`.
- This is a *one-way* or *simple* decision.

Forming Simple Conditions

- What does a condition look like?
- At this point, let's use simple comparisons.
- `<expr> <relop> <expr>`
- `<relop>` is short for *relational operator*

Forming Simple Conditions

Python	Mathematics	Meaning
<	<	Less than
<=	≤	Less than or equal to
==	=	Equal to
>=	≥	Greater than or equal to
>	>	Greater than
!=	≠	Not equal to

Forming Simple Conditions

- Notice the use of == for equality. Since Python uses = to indicate assignment, a different symbol is required for the concept of equality.
- A common mistake is using = in conditions!

Forming Simple Conditions

- Conditions may compare either numbers or strings.
- When comparing strings, the ordering is *lexigraphic*, meaning that the strings are sorted based on the underlying ASCII codes. Because of this, all upper-case letters come before lower-case letters. (“Bbbb” comes before “aaaa”)

Forming Simple Conditions

- Conditions are based on *Boolean* expressions, named for the English mathematician George Boole.
- When a Boolean expression is evaluated, it produces either a value of *true* (meaning the condition holds), or it produces *false* (it does not hold).
- Some computer languages use 1 and 0 to represent "true" and "false".

Forming Simple Conditions

- Boolean conditions are of type `bool` and the Boolean values of true and false are represented by the literals `True` and `False`.

```
>>> 3 < 4
True
>>> 3 * 4 < 3 + 4
False
>>> "hello" == "hello"
True
>>> "Hello" < "hello"
True
```

Example: Conditional Program Execution

- There are several ways of running Python programs.
  - Some modules are designed to be run directly. These are referred to as programs or scripts.
  - Others are made to be imported and used by other programs. These are referred to as libraries.
  - Sometimes we want to create a hybrid that can be used both as a stand-alone program and as a library.

Example: Conditional Program Execution

- When we want to start a program once it's loaded, we include the line `main()` at the bottom of the code.
- Since Python evaluates the lines of the program during the import process, our current programs also run when they are imported into an interactive Python session or into another Python program.

Example: Conditional Program Execution

- Generally, when we **import** a module, we don't want it to execute!
- In a program that can be either run stand-alone or loaded as a library, the call to `main` at the bottom should be made conditional, e.g.

```
if <condition>:
   main()
```

Example: Conditional Program Execution

- Whenever a module is imported, Python creates a special variable in the module called `__name__` to be the name of the imported module.
- Example:

```
>>> import math
>>> math.__name__
'math'
```

Example: Conditional Program Execution

- When imported, the `__name__` variable inside the math module is assigned the string `'math'`.
- When Python code is run directly and *not* imported, the value of `__name__` is `'__main__'`. E.g.:

```
>>> __name__
'__main__'
```

Example: Conditional Program Execution

- To recap: if a module is imported, the code in the module will see a variable called `__name__` whose value is the name of the module.
- When a file is run directly, the code will see the value `'__main__'`.
- We can change the final lines of our programs to:

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

- Virtually every Python module ends this way!

Two-Way Decisions

- Let's look at the quadratic program as we left it.

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

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

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

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print
    print "The solutions are:", root1, root2

main()
```

Two-Way Decisions

- As the comment implies, when $b^2-4ac < 0$, the program tries to take the square root of a negative number, and then crashes.

```
This program finds the real solutions to a quadratic

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

Traceback (most recent call last):
  File "C:\Documents and Settings\Terry\My Documents\Teaching\W04\CS
    120\Textbook\code\chapter3\quadratic.py", line 21, in -toplevel-
    main()
  File "C:\Documents and Settings\Terry\My Documents\Teaching\W04\CS
    120\Textbook\code\chapter3\quadratic.py", line 14, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

Two-Way Decisions

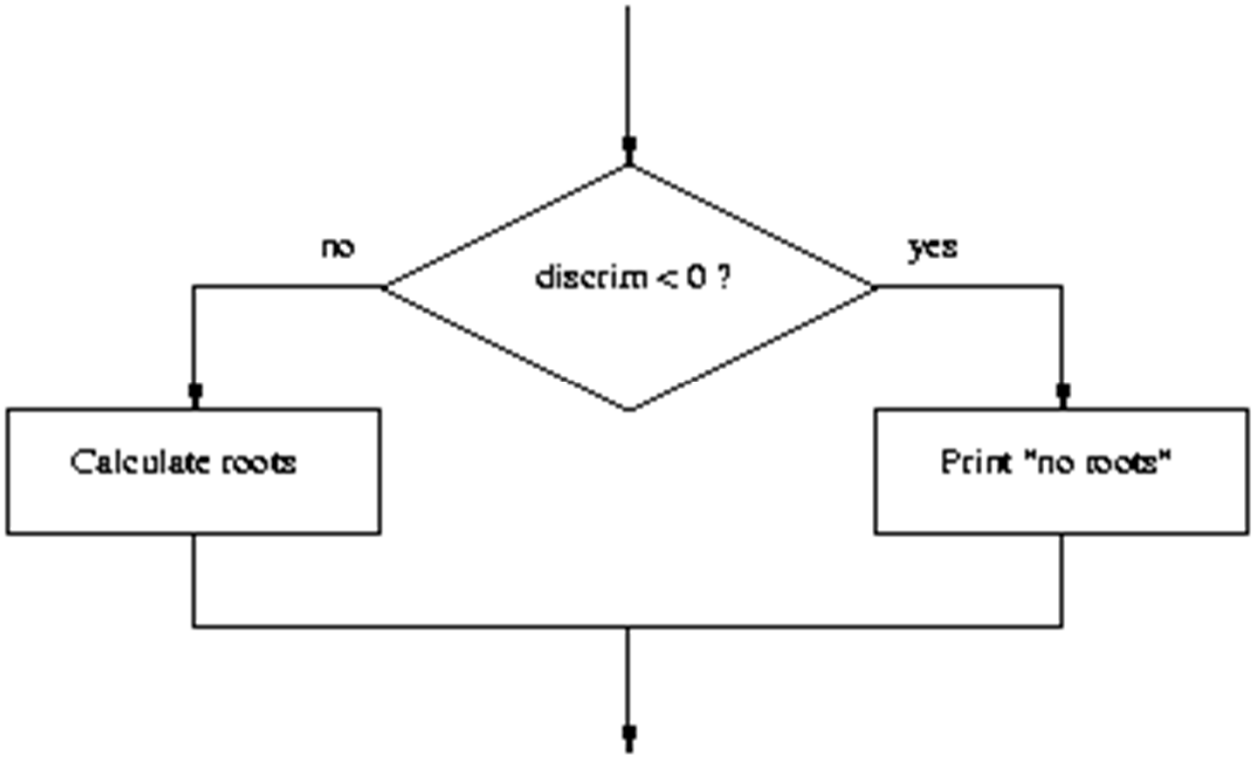
- We can check for this situation. Here's our first attempt.

```
# quadratic2.py
#    A program that computes the real roots of a quadratic equation.
#    Bad version using a simple if to avoid program crash

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:
        discRoot = math.sqrt(discrim)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print "\nThe solutions are:", root1, root2
```

Two-Way Decisions

- We first calculate the discriminant (b^2-$4ac$) and then check to make sure it's nonnegative. If it is, the program proceeds and we calculate the roots.
- Look carefully at the program. What's wrong with it? Hint: What happens when there are no real roots?

Two-Way Decisions

- ```
  This program finds the real solutions to a
  quadratic

  Please enter the coefficients (a, b, c): 1,1,1
  >>>
  ```
- This is almost worse than the version that crashes, because we don't know what went wrong!
  ```

Two-Way Decisions

- We could add another `if` to the end:

```
if discrim < 0:
    print "The equation has no real roots!"
```

- This works, but feels wrong. We have two decisions, with *mutually exclusive* outcomes (if `discrim >= 0` then `discrim < 0` must be false, and vice versa).

Two-Way Decisions



Two-Way Decisions

- In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause.
- This is called an `if-else` statement:

```
if <condition>:
    <statements>
else:
    <statements>
```

Two-Way Decisions

- When Python first encounters this structure, it first evaluates the condition. If the condition is true, the statements under the `if` are executed.
- If the condition is false, the statements under the `else` are executed.
- In either case, the statements following the `if-else` are executed after either set of statements are executed.

Two-Way Decisions

```
# quadratic3.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of a two-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!"
    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

main()
```

Two-Way Decisions

```
>>>
This program finds the real solutions to a quadratic

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

The equation has no real roots!
>>>
This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 2, 5, 2

The solutions are: -0.5 -2.0
```

Multi-Way Decisions

- The newest program is great, but it still has some quirks!

```
This program finds the real solutions to a
quadratic

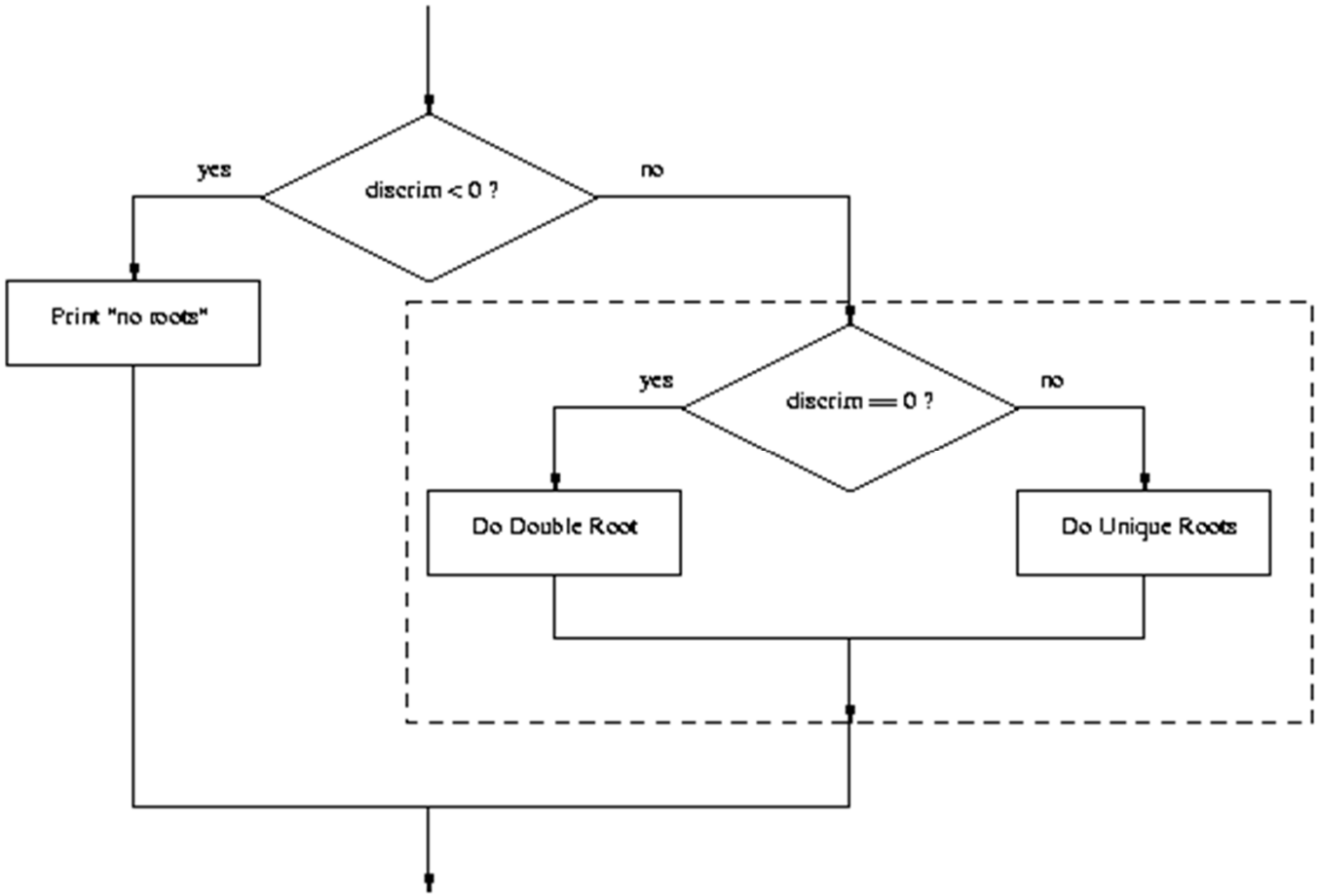
Please enter the coefficients (a, b, c):
1,2,1

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 $-b/2a$.
- 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*.
  ```

Multi-Way Decisions

```
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



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>
…
else:
    <default 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
```

Exception Handling

- In the quadratic program we used decision structures to avoid taking the square root of a negative number, thus avoiding a run-time error.
- This is true for many programs: decision structures are used to protect against rare but possible errors.

Exception Handling

- In the quadratic example, we checked the data *before* calling `sqrt`. Sometimes functions will check for errors and return a special value to indicate the operation was unsuccessful.
- E.g., a different square root operation might return a –1 to indicate an error (since square roots are never negative, we know this value will be unique).

Exception Handling

- ```
  discRt = otherSqrt(b*b - 4*a*c)
  if discRt < 0:
      print "No real roots.“
  else:
      ...
  ```
- Sometimes programs get so many checks for special cases that the algorithm becomes hard to follow.
- Programming language designers have come up with a mechanism to handle *exception handling* to solve this design problem.
  ```

Exception Handling

- The programmer can write code that catches and deals with errors that arise while the program is running, I.e., “Do these steps, and if any problem crops up, handle it this way.”
- This approach obviates the need to do explicit checking at each step in the algorithm.

Exception Handling

```
# quadratic5.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates exception handling to avoid crash on bad inputs

import math

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

    try:
        a, b, c = input("Please enter the coefficients (a, b, c): ")
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print "\nThe solutions are:", root1, root2
    except ValueError:
        print "\nNo real roots"
```

Exception Handling

- The `try` statement has the following form:

```
try:
   <body>
except <ErrorType>:
   <handler>
```

- When Python encounters a `try` statement, it attempts to execute the statements inside the body.
- If there is no error, control passes to the next statement after the `try`…`except`.

Exception Handling

- If an error occurs while executing the body, Python looks for an except clause with a matching error type. If one is found, the handler code is executed.
- The original program generated this error with a negative discriminant:

```
Traceback (most recent call last):
  File "C:\Documents and Settings\Terry\My
Documents\Teaching\W04\CS120\Textbook\code\chapter3\quadratic.py", line 21, in -toplevel-
    main()
  File "C:\Documents and Settings\Terry\My Documents\Teaching\W04\CS
120\Textbook\code\chapter3\quadratic.py", line 14, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

Exception Handling

- The last line, "`ValueError: math domain error`", indicates the specific type of error.
- Here's the new code in action:

```
This program finds the real solutions to a quadratic

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

No real roots
```

- Instead of crashing, the exception handler prints a message indicating that there are no real roots.

Exception Handling

- The `try…except` can be used to catch *any* kind of error and provide for a graceful exit.
- In the case of the quadratic program, other possible errors include not entering the right number of parameters (“`unpack tuple of wrong size`”), entering an identifier instead of a number (`NameError`), entering an invalid Python expression (`TypeError`).
- A single `try` statement can have multiple `except` clauses.

Exception Handling

```
# quadratic6.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates robust exception handling to avoid crash on bad inputs

import math

def main():
    print "This program finds the real solutions to a quadratic\n"
    try:
        a, b, c = input("Please enter the coefficients (a, b, c): ")
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print "\nThe solutions are:", root1, root2
    except ValueError, excObj:
        msg = str(excObj)
        if msg == "math domain error":
            print "\nNo Real Roots"
        elif msg == "unpack tuple of wrong size":
            print "\nYou didn't give me the right number of coefficients."
        else:
            print "\nSomething went wrong, sorry!"
```

Exception Handling

```
    except NameError:
        print "\nYou didn't enter three numbers."
    except TypeError:
        print "\nYour inputs were not all numbers."
    except SyntaxError:
        print "\nYour input was not in the correct form. Missing comma(s),
   perhaps?"
    except:
        print "\nSomething went wrong, sorry!"

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

Exception Handling

- The multiple `excepts` act like `elifs`. If an error occurs, Python will try each `except` looking for one that matches the type of error.
- The bare `except` at the bottom acts like an `else` and catches any errors without a specific match.
- If there was no bare `except` at the end and none of the `except` clauses match, the program would still crash and report an error.

Exception Handling

- Exceptions themselves are a type of object.
- If you follow the error type with an identifier in an `except` clause, Python will assign that identifier the actual exception object.

Exception Handling

```
except ValueError, excObj:
       msg = str(excObj)
       if msg == "math domain error":
           print "\nNo Real Roots"
       elif msg == "unpack tuple of wrong size":
           print "\nYou didn't give me the right number of coefficients."
       else:
           print "\nSomething went wrong, sorry!"
```

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.

Study in Design: Max of Three

```
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 x1 is the largest.
- ```
  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 x1 ≥ x2 ≥ x3.
  ```

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*!

```
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.

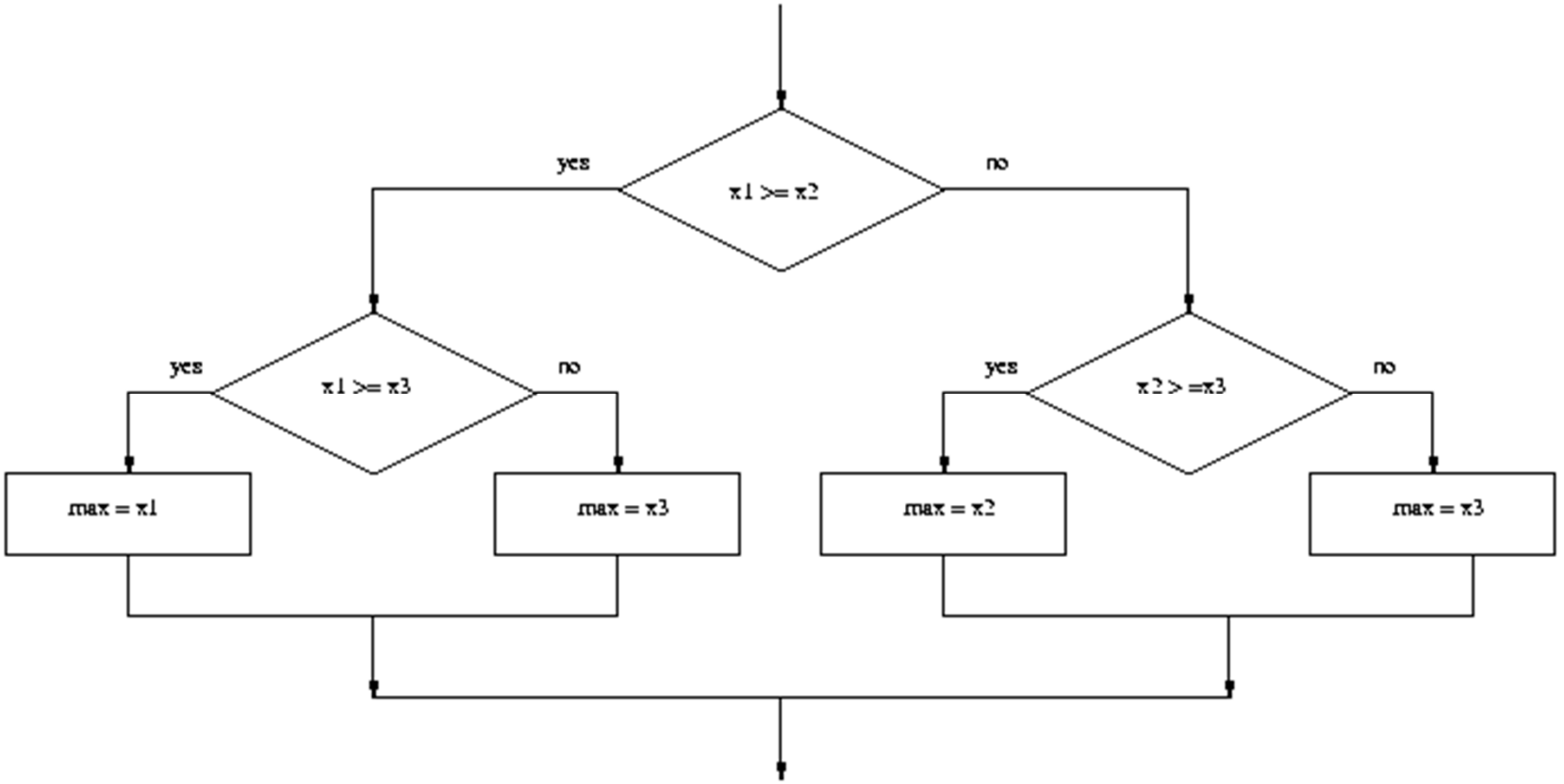
Strategy 1: Compare Each to All

- 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 *and*ed 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 `x1 >= x2`. This knocks either `x1` or `x2` out of contention to be the max.
- If the conidition is true, we need to see which is larger, `x1` or `x3`.

Strategy 2: Decision Tree



Strategy 2: Decision Tree

```
if x1 >= x2:
    if x1 >= x3:
        max = x1
    else:
        max = x3
else:
    if x2 >= x3:
        max = x2
    else
        max = x3
```

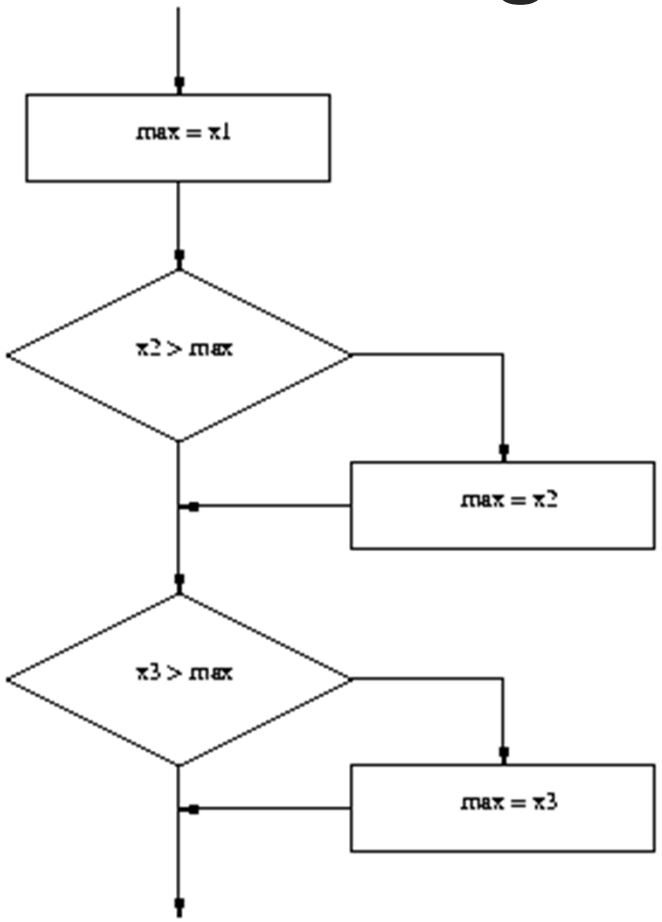
Strategy 2: Decision Tree

- 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`s 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.

```
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.

```
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.