Quick review

- **one-way or simple decision**
  - `if <condition>:`
    `<body>`
- **two-way decision**
  - This is called an `if-else` statement:

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

Quick review

- **Multi-Way Decisions**
  -
```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
…
else:
    <default statements>
```

Cont'd…
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

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.

- **Drawback:**
  - Sometimes programs get so many checks for special cases that the algorithm becomes hard to follow.

Exception Handling

- 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 = eval(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
import math

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

    try:
        a, b, c = eval(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 as excObj:
        if str(excObj) == "math domain error":
            print("No Real Roots")
        else:
            print("You didn't give me the right number of coefficients.")
    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?")
    except:
        print("\nSomething went wrong, sorry!")

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.

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 = eval(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:

  case 1: `max = x1`
  case 2: `max = x2`
  case 3: `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?

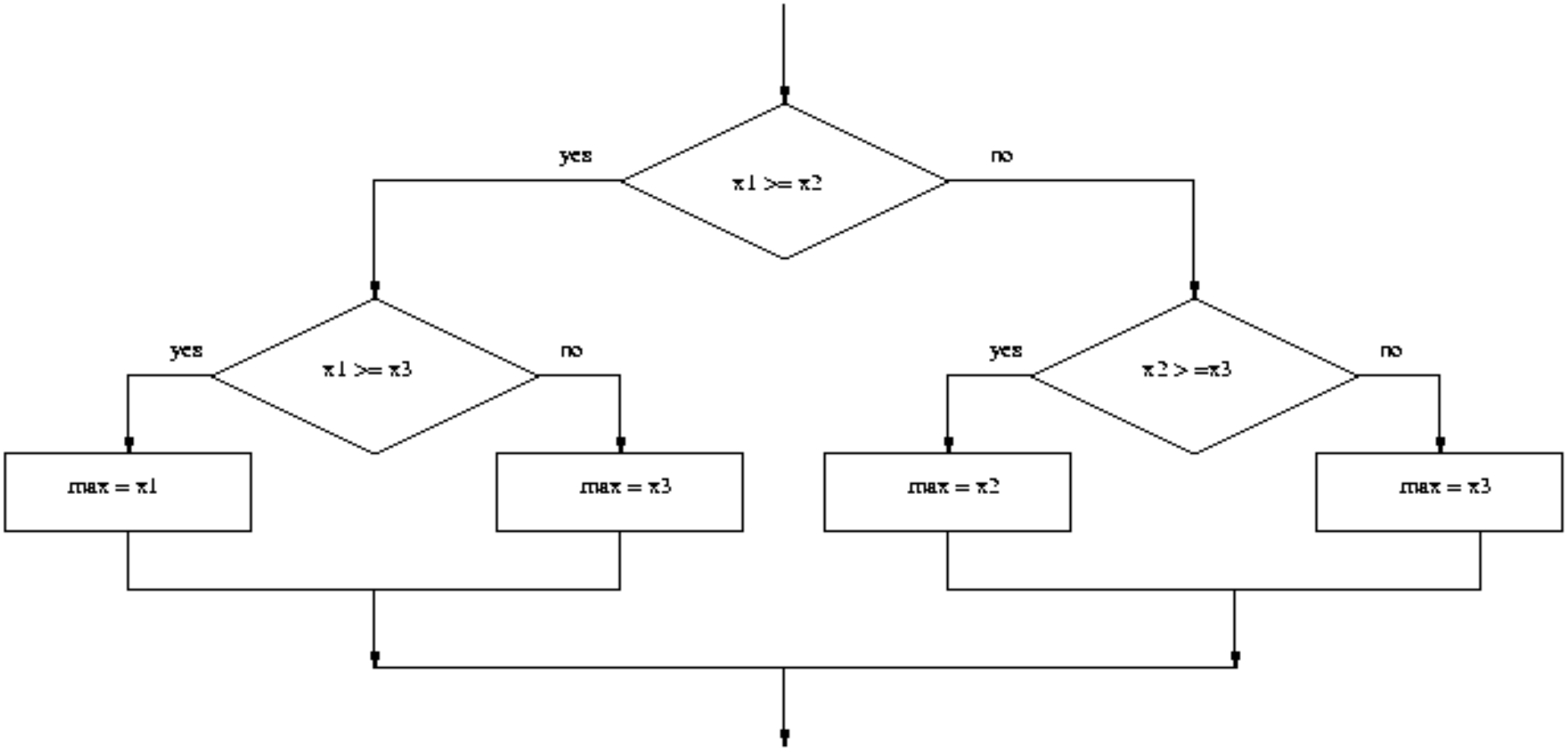
- We would need four Boolean expressions, each consisting of four conditions *and*ed together.

- **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.

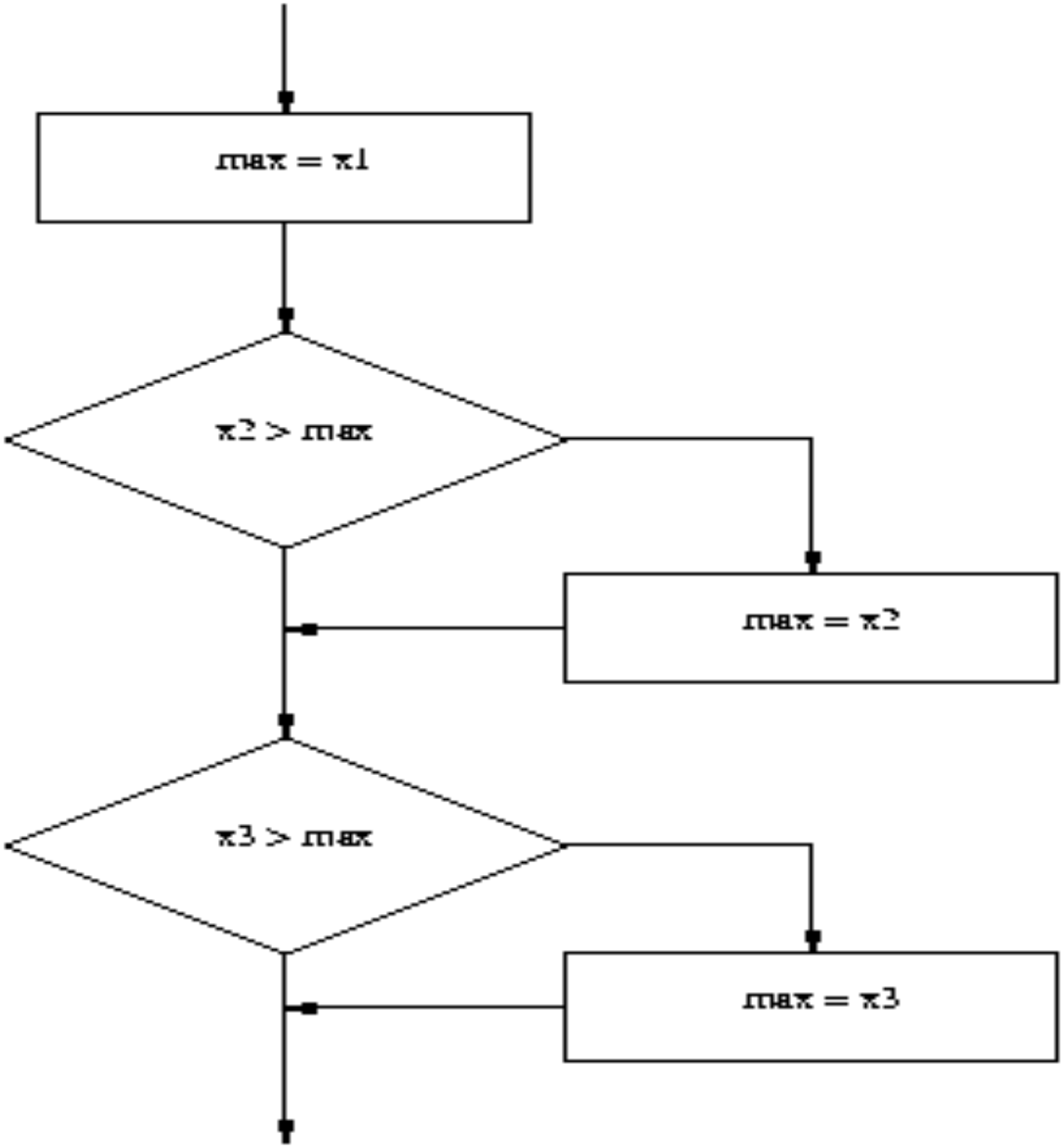
Strategy 3:
Sequential Processing

- 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 P



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 = eval(input("How many numbers are there? "))

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

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = eval(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 = eval(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

- **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.