http://www.guardian.co.uk/uk/2012/oct/10/prince-sealand-dies

Prince of sovereign principality of Sealand dies, aged 91

Mid-semester feedback

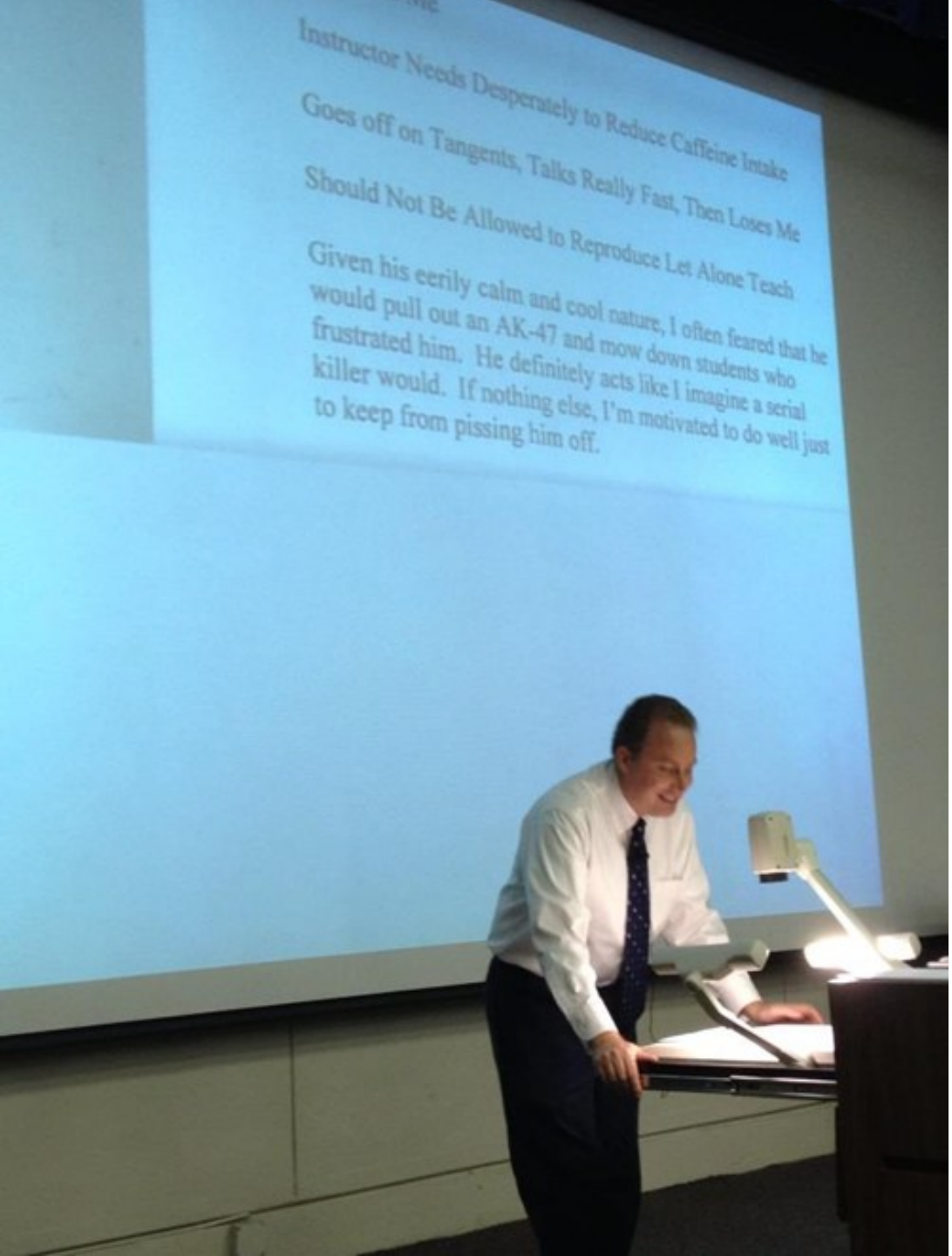
1. What's been most useful to you (and why)?

2. What could be going better / be more useful (and why)?

3. What could students do to improve the class?

4. What could Matt do to improve the class?

5. How many hours per week do you spend on the course *outside* of class/lab?


Instructor Needs Desperately to Reduce Caffeine Intake
Goes off on Tangents, Talks Really Fast, Then Loses Me
Should Not Be Allowed to Reproduce Let Alone Teach
Given his eerily calm and cool nature, I often feared that he would pull out an AK-47 and mow down students who frustrated him. He definitely acts like I imagine a serial killer would. If nothing else, I'm motivated to do well just to keep from pissing him off.

Simple Decisions

- So far, we've viewed programs as sequences of instructions that are followed one after the other.
- What about *if* statements? What about conditional loops?
  - If the user enters an even number, do one thing. If the user enters an odd, do something else.
  - If the user enters 'q', quit the program.

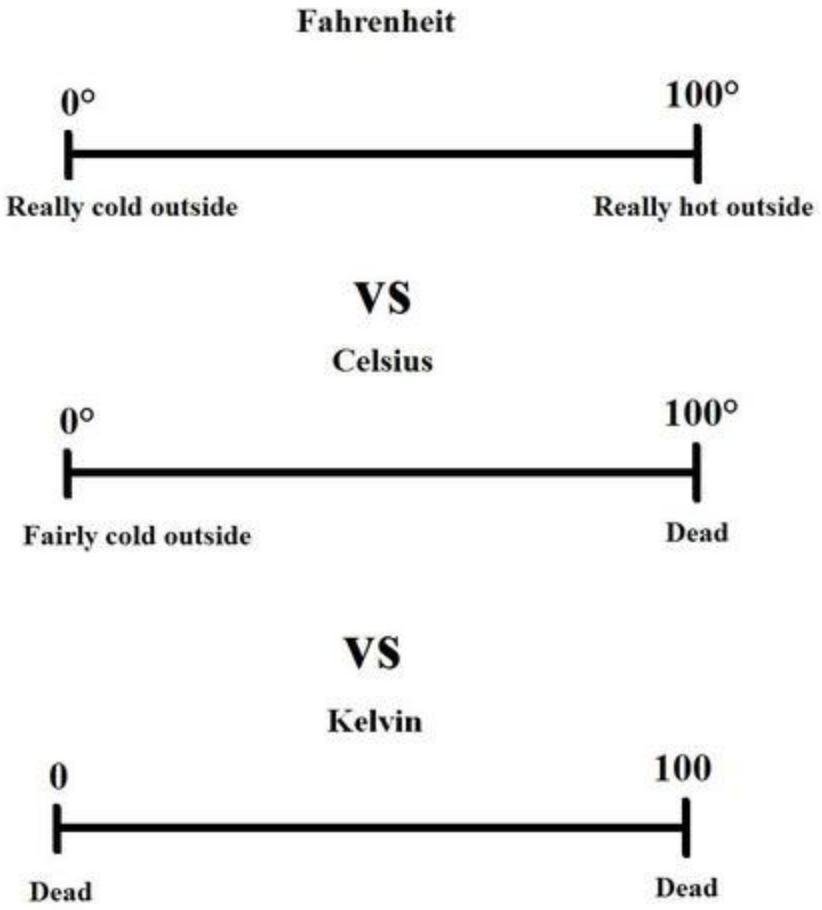
Example: Temperature Warnings

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

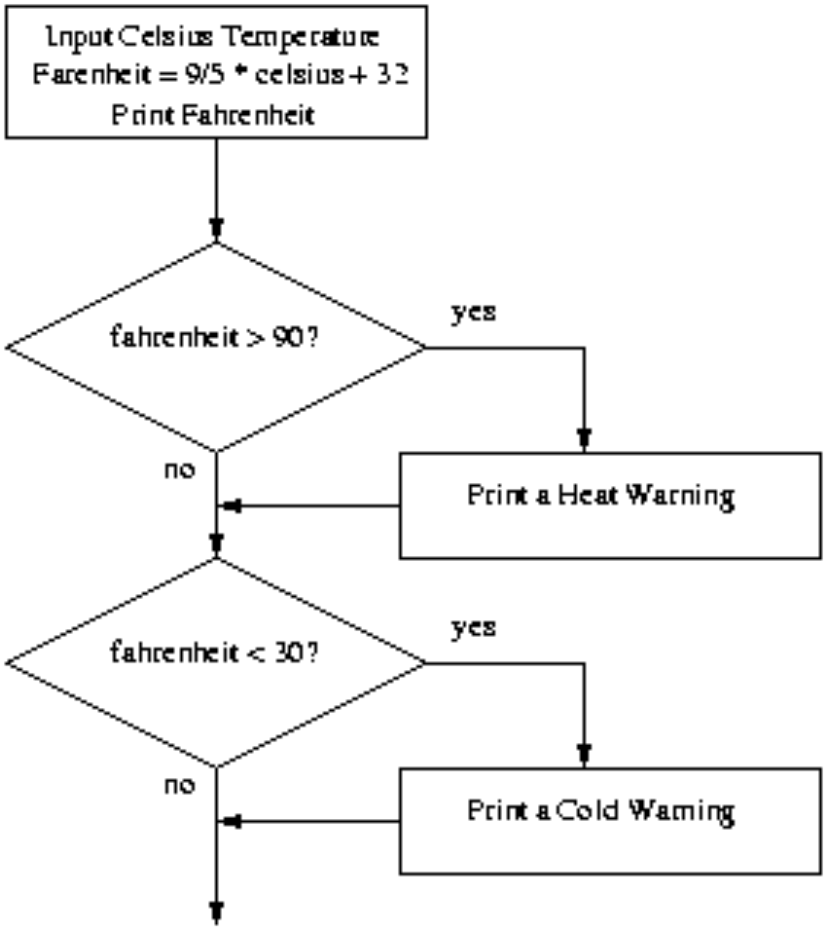
```
# convert.py
#      A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")

main()
```

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



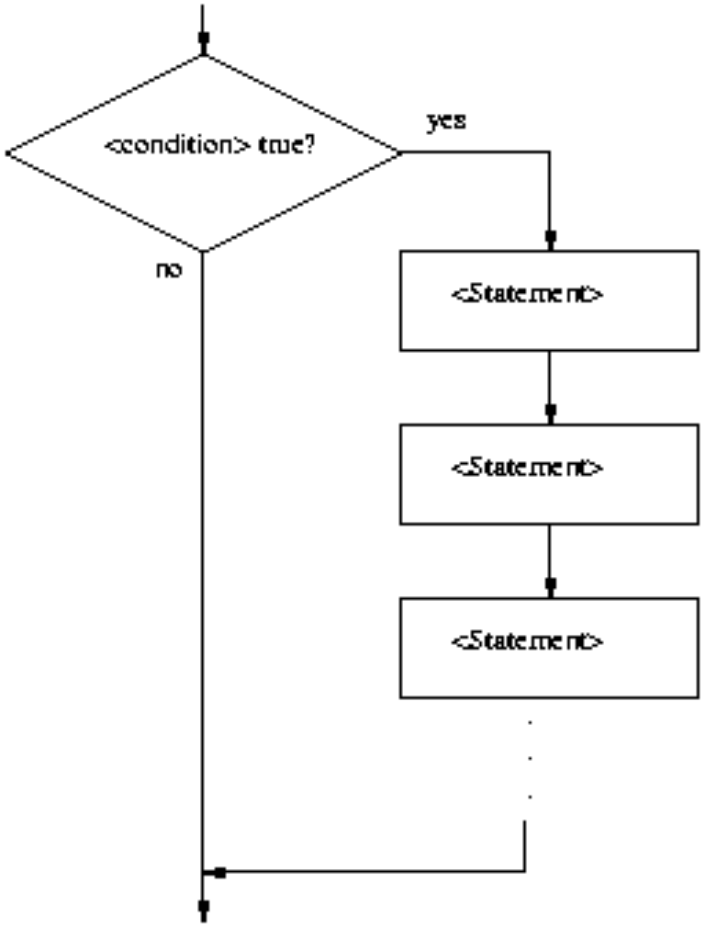
Example: Temperature Warnings

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

def main():
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9 / 5 * 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. Better take a warm cat with you")

main()
```

Example: Temperature Warnings

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

Example: Temperature Warnings



Example: Temperature Warnings

- The body of the `if` either executes or not depending on the condition.
- Either way, control then passes to the next statement after the `if`

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
- Common bug: 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 Unicode. 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 (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
  - Modules are designed to be run directly: programs or scripts
  - Others are made to be imported and used by other programs: libraries
  - Hybrids? 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, include the line `main()` at the bottom
- 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

- When importing a module, Python creates a special variable in the module called `__name__`

- 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

- Module imported ➔ code will see a variable called `__name__` whose value is name of module
- File run directly ➔ code will see the value '`__main__`'
- We can change final lines of programs to:

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

- Most Python modules ends this way

Two-Way Decisions

- Consider the quadratic program as we left it.

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

import math

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

    a, b, c = eval(input("\nPlease 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)

main()
```

Two-Way Decisions

- As per the comment, when $b^2-4ac < 0$, the program 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

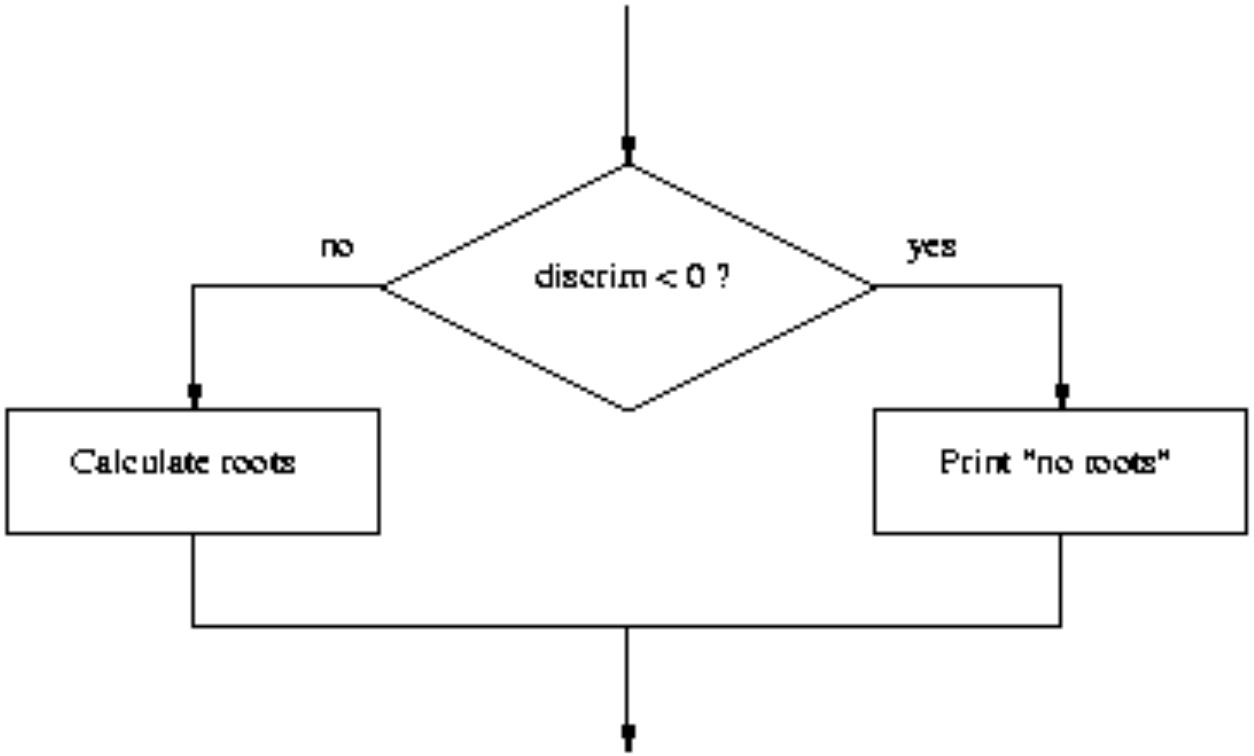
- We can check for this situation. Here's our first attempt.

```
# 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 = eval(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

- First, calculate discriminant (b^2-4ac) and check to make sure it's nonnegative. If it is, program proceeds
- But, there's still a problem. 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
  >>>
  ```
- Almost worse than previous version --- 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. Two decisions, with *mutually exclusive* outcomes (if `discrim >= 0` then `discrim < 0` must be false, & vice versa)

Two-Way Decisions



Two-Way Decisions

- A two-way decision can be implemented by attaching an `else` onto an `if`

- This is called an `if-else` statement:

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

Two-Way Decisions

- First evaluate the condition. If condition is true, the statements under `if` are executed
- If condition is false, the statements under `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 = eval(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

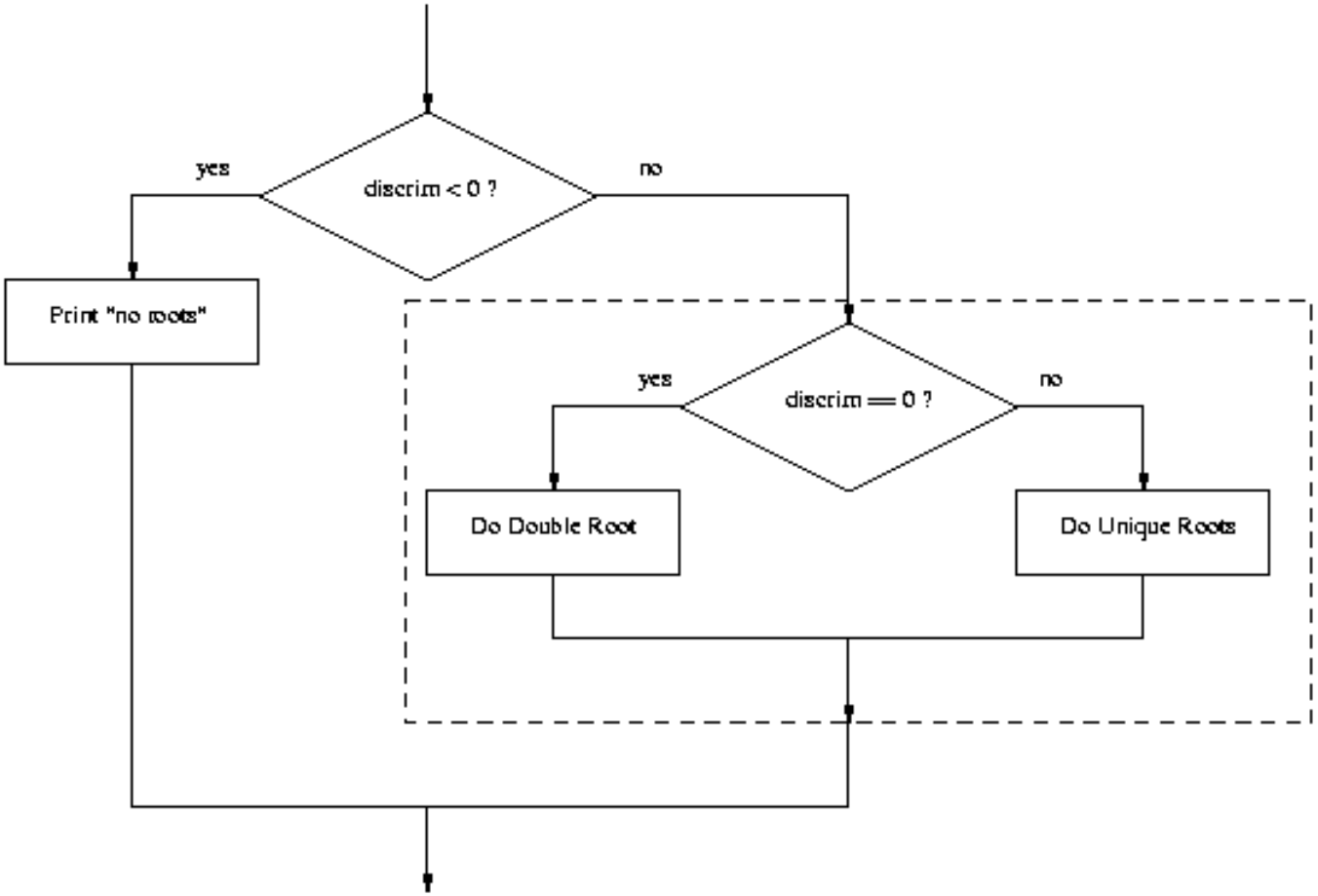
  Please enter the coefficients (a, b, c): 1,2,1

  The solutions are: -1.0 -1.0
```

Multi-Way Decisions

- While correct, this method could be confusing
- Double roots occur when discriminant is exactly 0. 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
  ```
- 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 Decis

- What if we needed to make a five-way decision using nesting? `if-else` statements would be nested four levels deep…
- Can combine 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

- Sets of mutually exclusive code blocks
- Python evaluates each condition in turn looking for the first one that is true. If true condition is found, 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
#    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 = eval(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 quadratic program we used decision structures to avoid taking the square root of a negative number, avoiding a run-time error
- This is true for many programs: decision structures are used to protect against rare errors

Exception Handling

- In quadratic example, we checked data *before* calling `sqrt`. Functions could also check for errors and return a special value to indicate operation was unsuccessful
- E.g., a different square root operation might return a –1 to indicate an error

Exception Handling

- Programmer could write code that catches and deals with errors that arise while the program is running
- “Do these steps, and if any problem crops up, handle it this way.”
- This approach obviates the need to explicitly check at each step in 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")

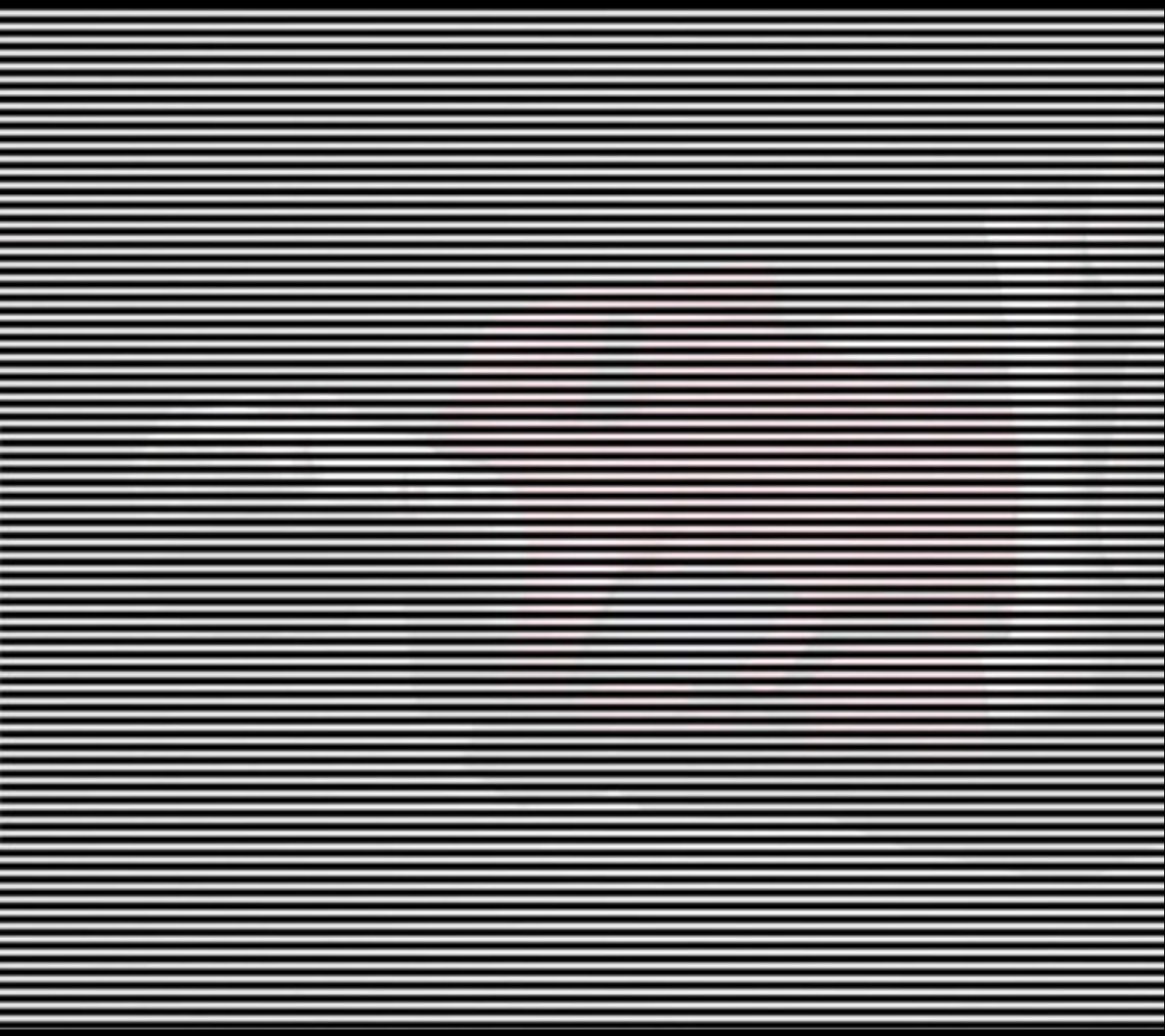
    try:
        a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
    except ValueError:
        print("\nNo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

- Exception handler prints a message indicating 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

- 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

- 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: equivalent to x1 ≥ x2 ≥ x3
  ```

Strategy 1: Compare Each to All

- Whenever you write a decision, there are two crucial questions:
  - When condition is true, is executing the body of 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