Introduction to Python Language

DECISION STRUCTURES

Today

- Comparison operations
- Logic Operations
- Decision Structures

COMPARISON OPERATORS ON `int`, `float`, `string`

- i and j are variable names
- comparisons below evaluate to a Boolean

i > j	
i >= j	
i < j	
i <= j	
i == j	equality test, True if `i` is the same as `j`
i != j	inequality test, True if `i` is not the same as `j`

Notice especially the use of == for *equality*. Since Python uses the = sign to indicate an *assignment* statement, a different symbol is required for the concept of equality. A common mistake in Python programs is using = in conditions, where a == is required.

LOGIC OPERATORS ON bools

- a and b are variable names (with Boolean values)
- comparisons below evaluate to a Boolean

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

`not a` → `True` if `a` is `False`
→ `False` if `a` is `True`

`a and b` → `True` if both are `True`

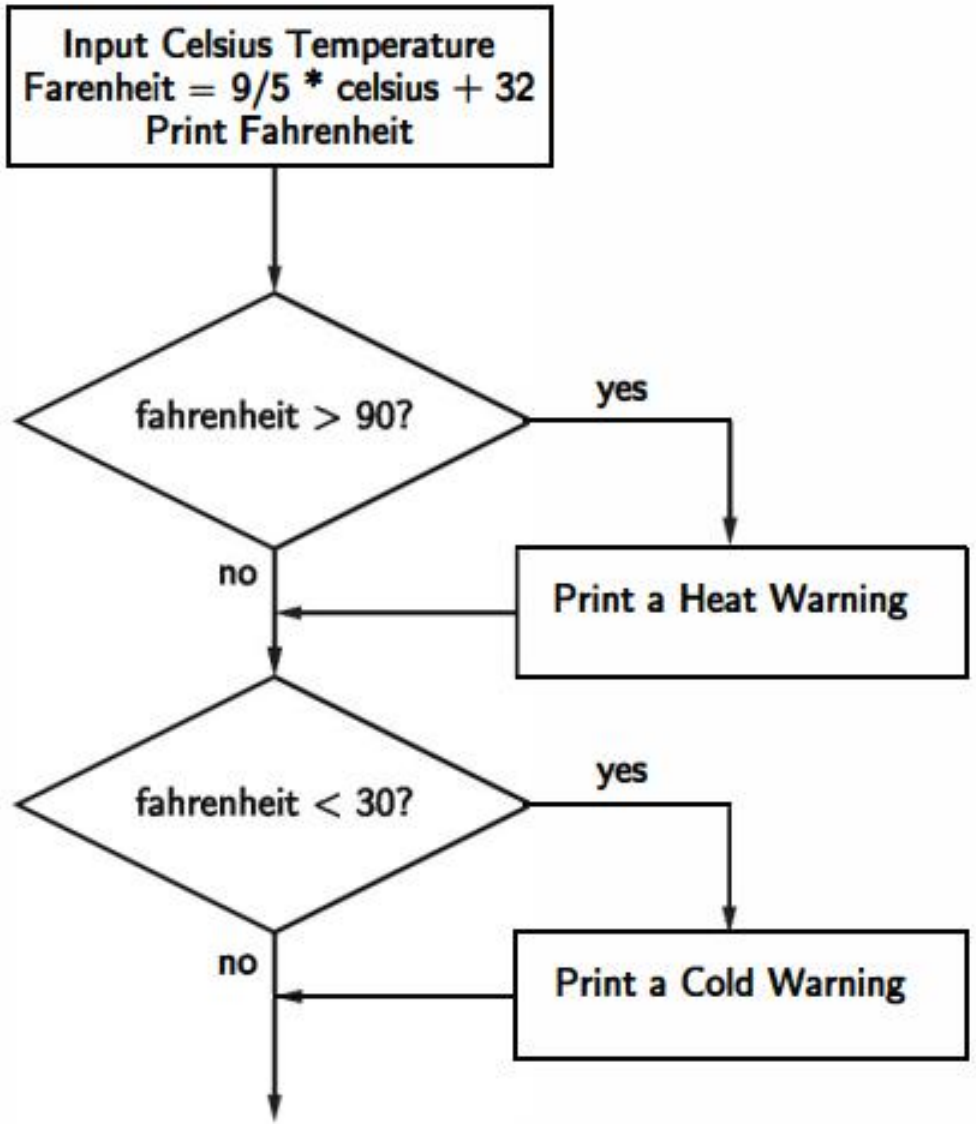
`a or b` → `True` if either or both are `True`

LOGIC OPERATORS ON bools

```
# comparison_examples.py

# Read two numbers from the user
x = int(input("Enter first number: "))
y = int(input("Enter second number: "))

print("\nComparison Results:")

# Basic comparisons
print("x == y :", x == y)     # equal to
print("x != y :", x != y)     # not equal to
print("x > y  :", x > y)      # greater than
print("x < y  :", x < y)      # less than
print("x >= y :", x >= y)     # greater than or equal to
print("x <= y :", x <= y)     # less than or equal to
```

Decision Structures (Branching)

- We have mostly viewed computer programs as sequences of instructions that are followed one after the other. Sequencing is a fundamental concept of programming, but alone it is not sufficient to solve every problem.
- Often it is necessary to alter the sequential flow of a program to suit the needs of a particular situation. This is done with special statements known as *control structures*.
- Control structures are statements that allow a program to execute different sequences of instructions for different cases, effectively allowing the program to "choose" an appropriate course of action.



- Input temperature in Celcius
- Calculate Fahrenheit
- if temperature >90, print «a heat warning»
- if temperature <30, print «a cold warning»

```
if <condition>:
    <body>
```

- One-Way Decision

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

- Two-Way Decision

```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
...
else:
    <default statements>
```

- Multi-Way Decision

Example: Write a program to find the real roots of a quadratic equation:

$$ax^2 + bx + c = 0$$

Such an equation has two real roots and can be calculated by using following formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Let's write a program that can find the solutions to a quadratic equation. The input to the program will be the values of the coefficients *a*, *b*, and *c*. The outputs are the two values given by the quadratic formula.

```
# 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.
print ("This program finds the real solutions to a quadratic")
print ()

a = float (input ("Enter coefficient a: ") )
b - float (input ("Enter coefficient b: ") )
c = float (input ("Enter coefficient 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 )
```

- To compute $\sqrt{x}$, we use `math.sqrt(x)` . This special dot notation tells Python to use the sqrt function that "lives" in the math module.

- This program crashes when it is given coefficients of a quadratic equation that has no real roots. The problem with this code is that when $b^2 - 4ac$ is less than 0, the program attempts to take the square root of a negative number. Since negative numbers do not have real roots, the math library reports an error.
- We can use a decision to check for this situation and make sure that the program can't crash.

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

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3
Traceback (most recent call last):
  File "C:\Users\N.Furkan\AppData\Local\Programs\Python\Python313\Scripts\quadra
tic.py", line 13, in <module>
    discRoot = math.sqrt (b * b - 4 * a * c)
ValueError: math domain error
```

```
print ("This program find s the real solutions to a quadratic\n" )
a = float (input ("Enter coefficient a : " ) )
b = float (input ("Enter coefficient b : " ) )
c = float (input ("Enter coefficient c : " ) )

discrim = b * b - 4 * a * c
if discrim >= 0:

main ()

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

Still, this updated version is not really a complete solution. Do you see what happens when the equation has no real roots? According to the semantics for a simple if, when b*b - 4*a* c is less than zero, the program will simply skip the calculations and go to the next statement. Since there is no next statement, the program just quits.

Still, this updated version is not really a complete solution. Do you see what happens when the equation has no real roots? According to the semantics for a simple if, when b*b - 4*a* c is less than zero, the program will simply skip the calculations and go to the next statement. Since there is no next statement, the program just quits.

```
This program find s the real solutions to a quadratic

Enter coefficient a : 1
Enter coefficient b : 2
Enter coefficient c : 3
>>>
```

This is almost worse than the previous version, because it does not give users any indication of what went wrong; it just leaves them hanging. A better program would print a message telling users that their particular equation has no real solutions.
We could accomplish this by adding another simple decision at the end of the program.

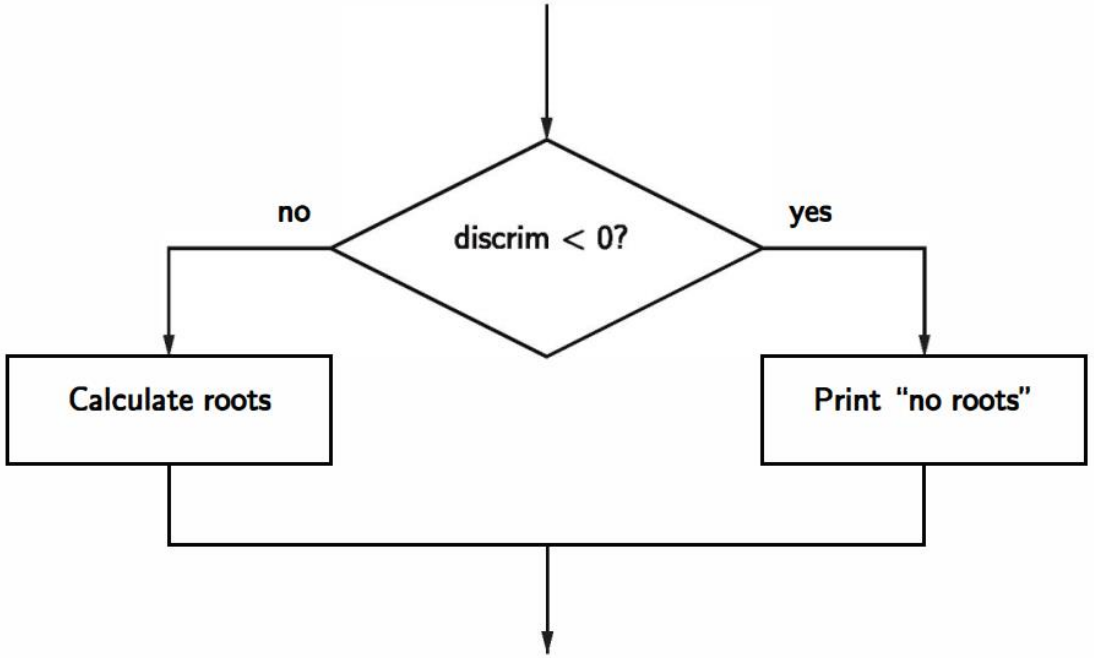
```
if discrim < 0:
print ("The equation has no real roots!" )
```

```
# 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.

print ("This program find s the real solutions to a quadratic\n" )
a = float (input ("Enter coefficient a : " ) )
b = float (input ("Enter coefficient b : " ) )
c = float (input ("Enter coefficient 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 )
if discrim < 0:
    print ("The equation has no real roots!" )
```

We have programmed a sequence of two decisions, but the two outcomes are mutually exclusive. If `discrim >= 0` is *true* then `discrim < 0` must be *false* and vice versa. We have two conditions in the program, but there is really only one decision to make.

In Python, a two-way decision can be implemented by attaching an else clause onto an if clause. The result is called an ***if- else statement***.

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



When the Python interpreter encounters this structure, it will first evaluate 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.

```
import math

print ("This program find s the real solutions to a quadratic\n" )

a = float (input ("Enter coefficient a : " ) )
b = float (input ("Enter coefficient b : " ) )
c = float (input ("Enter coefficient 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)
```

```
This program find s the real solutions to a quadratic

Enter coefficient a : 1
Enter coefficient b : 2
Enter coefficient c : 3

The equation has no real roots!
```

```
This program find s the real solutions to a quadratic

Enter coefficient a : 2
Enter coefficient b : 4
Enter coefficient c : 1

The solutions are : -0.2928932188134524 -1.7071067811865475
```

Multi-way Decisions

The newest version of the quadratic solver is certainly a big improvement, but it still has some quirks (oddity). Here is another example run:

```
This program find s the real solutions to a quadratic

Enter coefficient a : 1
Enter coefficient b : 2
Enter coefficient c : 1

The solutions are : -1.0 -1.0
```

This is technically correct; the given coefficients produce an equation that has a double root at -1. However, the output might be confusing to some users.

The program should be a bit more informative to avoid confusion.

The double-root situation occurs when *discrim* is exactly 0. In this case, *discRoot* is also 0, and both roots have the value $\frac{-b}{2a}$.

If we want to catch this special case, our program actually needs a three-way decision. Here's a quick sketch of the design:

```
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.
```

The program should be a bit more informative to avoid confusion.

The double-root situation occurs when *discrim* is exactly 0. In this case, *discRoot* is also 0, and both roots have the value $\frac{-b}{2a}$.

If we want to catch this special case, our program needs a three-way decision. Here's a quick sketch of the design:

```
 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
```

There is another way to write multi-way decisions in Python that preserves the semantics of the nested structures but gives it a more appealing look.

The idea is to combine an else followed immediately by an *if* into a single clause called an ***elif***.

```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
...
else:
    <default statements>
```

The final version of our program is:

```
import math

print ("This program find s the real solutions to a quadratic\n" )

a = float (input ("Enter coefficient a : " ) )
b = float (input ("Enter coefficient b : " ) )
c = float (input ("Enter coefficient 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 . sq rt (b * b - 4 * a * c)
      root1 = (-b + discRoot) / (2 * a )
      root2 = (-b - discRoot) / (2 * a )
      print ("\nThe solutions are :", root1, root2 )
```

Exception Handling

- Our quadratic program uses decision structures to avoid taking the square root of a negative number and generating an error at runtime.
- This is a common pattern in many programs: using decisions to protect against rare but possible errors.
- In the case of the quadratic solver, we checked the data before the call to the `sqrt` function. Sometimes functions themselves check for possible errors and return a special value to indicate that the operation was unsuccessful.
- For example, a **different square root operation** might return a negative number (say, -1) to indicate an error. Since the square root function should always return the non-negative root, this value could be used to signal that an error has occurred.
- The program would check the result of the operation with a decision:

```
discRt = otherSqrt(b*b - 4*a*c)
if discRt < 0:
    print("No real roots.")
else:
    ...
```

- Programs can become so cluttered with special case checks that the main algorithm for standard cases gets lost.
- Programming language designers have created exception-handling mechanisms to tackle common errors. These allow programmers to write code that catches and manages errors during execution. Instead of checking each step for success, a program can simply state, "Execute these steps, and handle any issues that arise in this way."

Here is a version of the quadratic program that uses Python's exception mechanism to catch potential errors in the math . sqrt function:

```
import math
print ("This program find s the real solutions to a quadratic\n" )

try :

a = float (input ("Enter coefficient a : " ) )
b = float (input ("Enter coefficient b : " ) )
c = float (input ("Enter coefficient 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" )
```

Notice that this is basically the very first version of the quadratic program with the addition of a try . . . except around the heart of the program.

A try statement has the general form:

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

- When Python encounters a try statement, it attempts to execute the statements inside the body. If these statements execute without error, control then passes to the next statement after the try . . . except.
- If an error occurs somewhere in the body, Python looks for an except clause with a matching error type.
- If a suitable except is found, the handler code is executed.

The original program without the exception handling produced the following error:

```
Traceback (most recent call last):
  File "quadratic.py", line 23, in <module>
    main()
  File "quadratic.py", line 16, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

- The last line of this error message indicates the type of error that was generated, namely a ValueError.

The updated version of the program provides an except clause to catch the ValueError. Here's how it looks in action:

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

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

No real roots
```

- Instead of crashing, the exception handler catches the error and prints a message indicating that the equation does not have real roots.

Interestingly, our new program also catches errors caused by the user typing invalid input values. Let's run the program again, and this time type "x" as the first input.

Interestingly, our new program also catches errors caused by the user typing invalid input values. Let's run the program again, and this time type "x" as the first input.

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

Enter coefficient a: x

No real roots
```

- Python raised a ValueError executing float ("x") because "x" is not convertible to a float. This caused the program to exit the try and jump to the except clause for that error.

- Ofcourse, the last message looks a bit of strange.

Final version of our program is:

```
import math
print ("This program find s the real solutions to a quadratic\n" )
try :
      a = float (input ("Enter coefficient a : " ) )
      b = float (input ("Enter coefficient b : " ) )
      c = float (input ("Enter coefficient 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 ("Invalid coefficient given" )
except :
      print ("\nSomething went wrong, sorry!" )
```

- If the exception is not a ValueError, this program just prints a general apology. As a challenge, you might see whether you can find an erroneous input that produces the apology.

- excObj: variable named as «exception object» is assigned to an error and kept for later usage or logging purposes