Python Programming Essentials

Unit 2: Structured Python

CHAPTER 7: DECISION STRUCTURES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

LECTURE 1

Objectives

- To understand the programming pattern simple decision and its implementation using a Python **if** statement.
- To understand the programming pattern two-way decision and its implementation using a Python **if-else** statement.

Objectives

- To understand the programming pattern multi-way decision 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

LECTURE 2

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.

Simple Decisions

- 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 = float(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")
```

Example:

Temperature Warnings

- Let’s say we want to modify the 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:

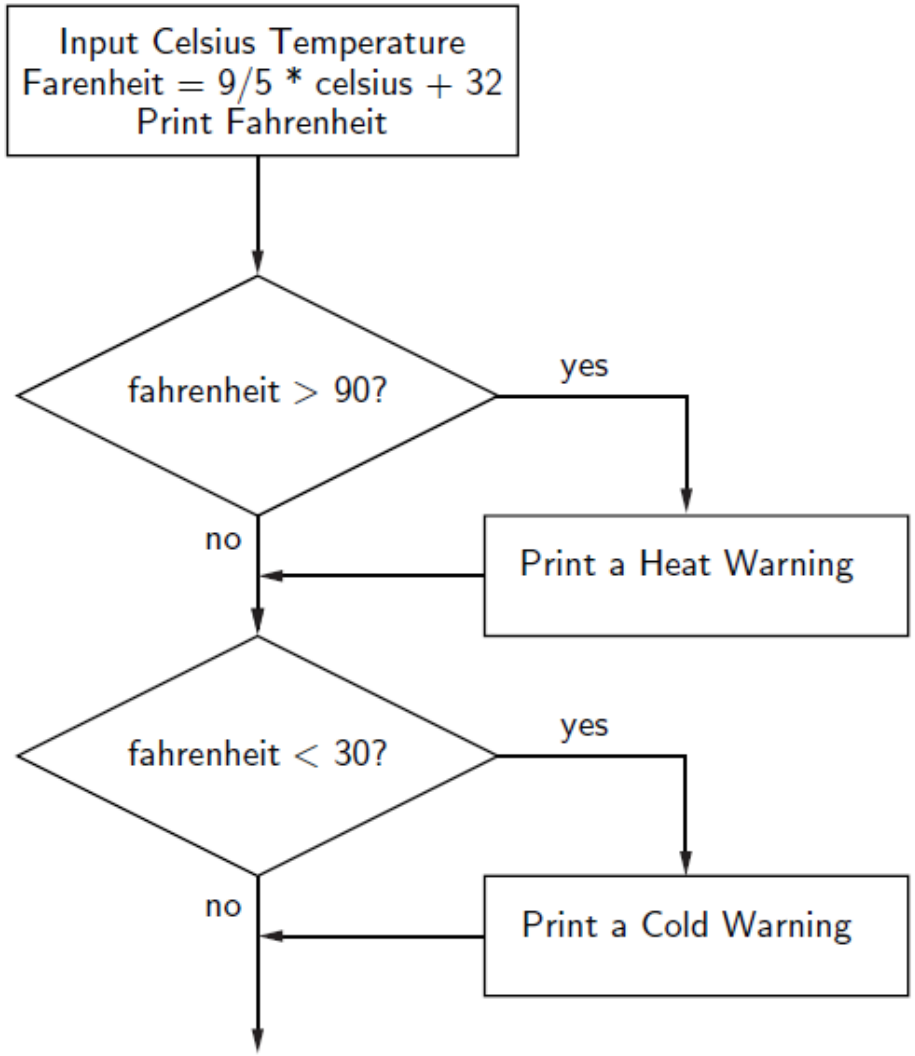
Temperature Warnings

```
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.



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 = float(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. 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.

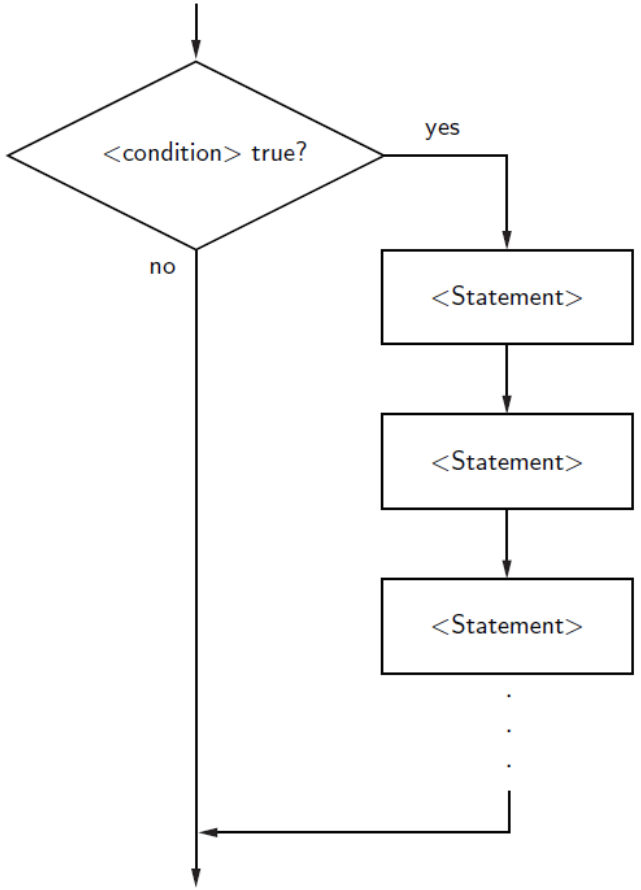
Example:

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.

Example:

Temperature Warnings



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

LECTURE 3

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 Unicode. Because of this, all upper-case Latin 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
```

Boolean Logic

LECTURE 4

Boolean Variable

A Boolean variable is a variable which has only two possible value: True or False.

The logic decision operation return a Boolean value (True or False).

True or False can also be represented as 1 or 0 in other languages.

Truth table is the result of a logic function.

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Logic Condition Example

Compound Logic Condition: (X >= 0 and X <=3) or (Y <= 4 and Y>=-1)

Boolean variable:

p = (X >= 0); q = (X <= 3); r = (Y <= 4); s = (Y >=-1);

t = (p and q); u = (r and s);

v = (t or u);

There are totally 7 logic operations in this compound logic condition.

== is a logical operation in Python.

= is an assignment in Python.

Python logical operator only support and, or, not. (This set is logic complete)

Boolean Identity Theorems

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Conditional Program Execution

LECTURE 5

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

LECTURE 6

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\n")

    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)
```

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 = 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)
```

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?

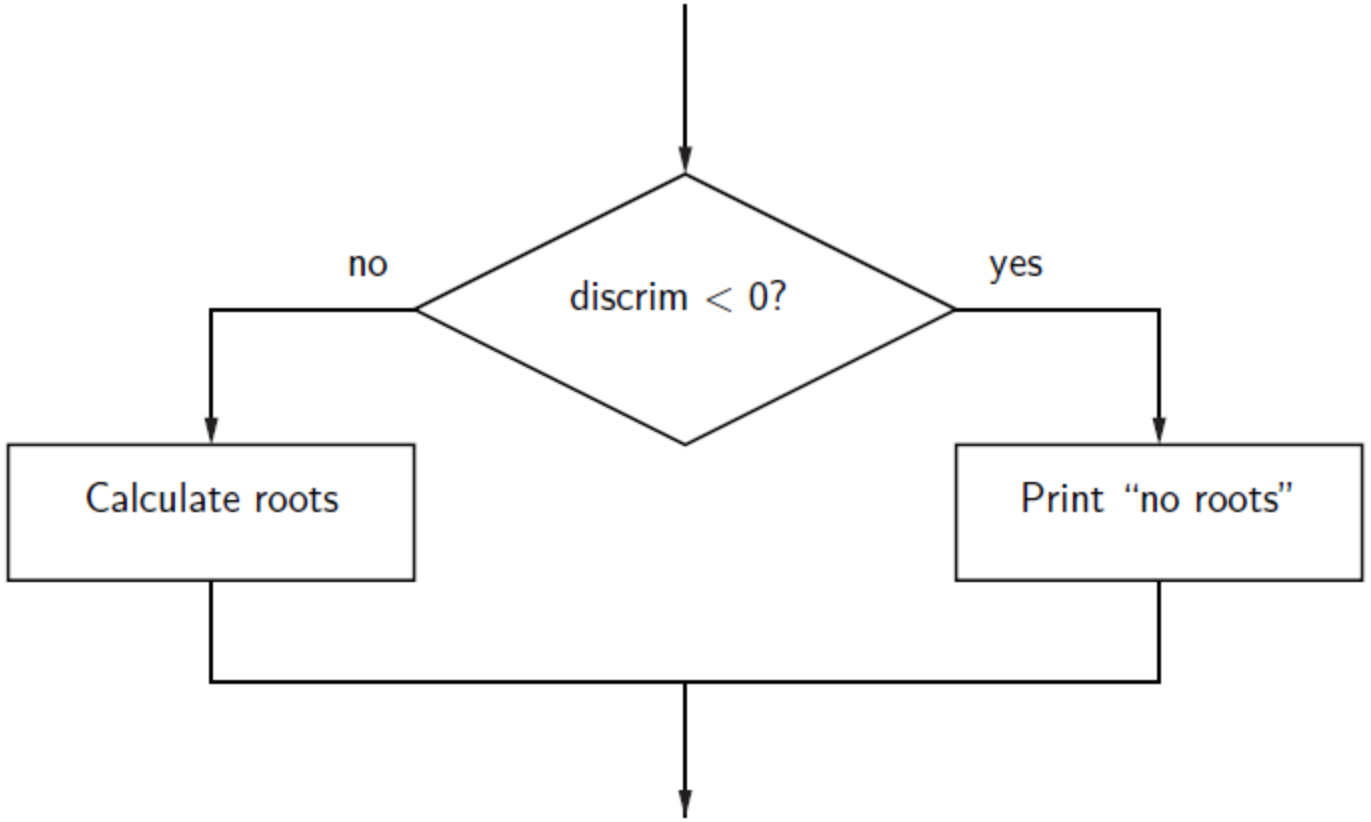
Two-Way Decisions

- 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 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 = 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 )
```

Two-Way Decisions

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

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

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

Enter coefficient a: 2
Enter coefficient b: 5
Enter coefficient c: 2

The solutions are: -0.5 -2.0
```

Multi-Way Decisions

LECTURE 7

Multi-Way Decisions

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

```
This program finds 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
```

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!

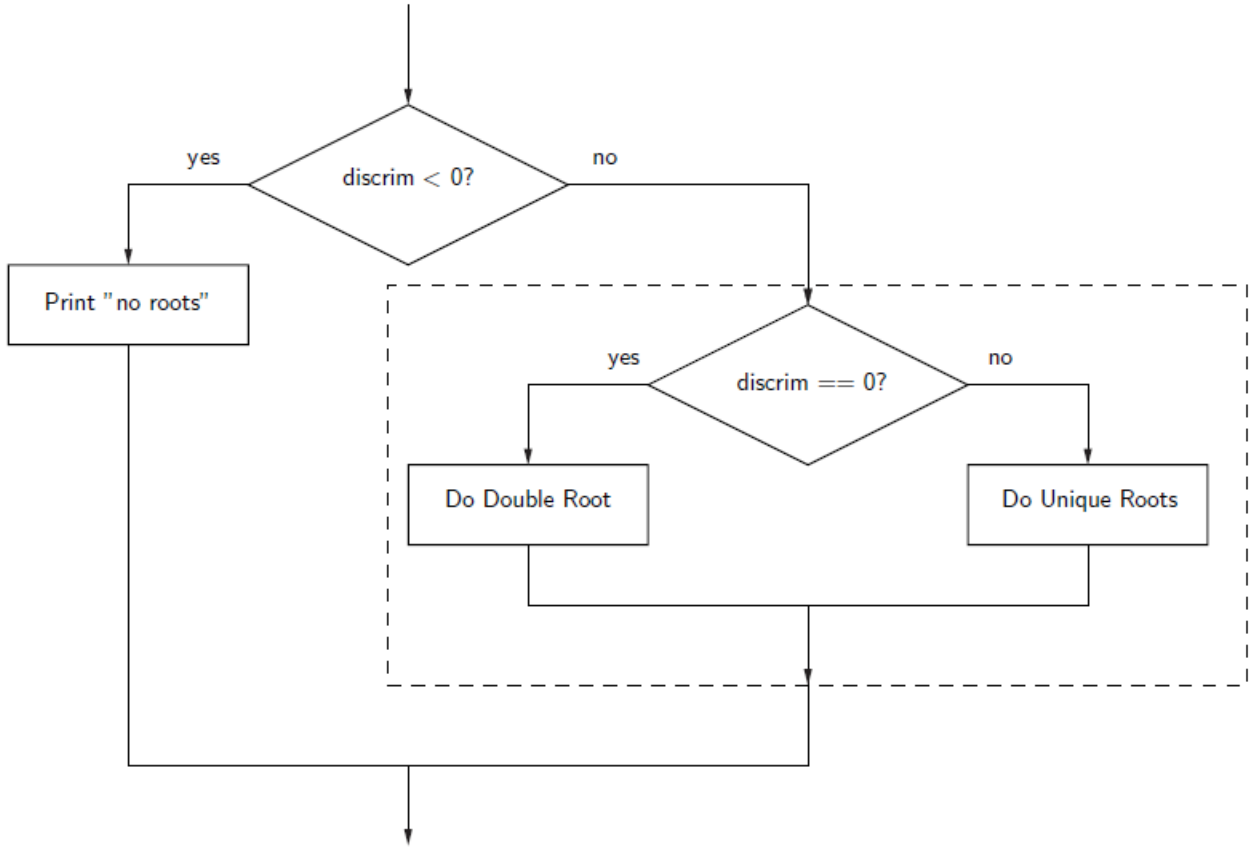
Multi-Way Decisions

- 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 off 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
import math

def main():
    print("This program finds 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.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```

Eexception Handling

LECTURE 8

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
import math

def main():
    print ("This program finds 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")
```

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

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

No real roots
```

Exception Handling

- Instead of crashing, the exception handler prints a message indicating that there are no real roots.
- The `try…except` can be used to catch any kind of error and provide for a graceful exit.
- 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 = 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!")
```

Exception Handling

- The multiple excepts act like `elif`s. 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 to that identifier the actual exception object.

Case Study : Max of Three

LECTURE 9

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", maxval)
```

Strategy 1:

Compare Each to All

This looks like a three-way decision, where we need to execute one of the following:

```
maxval = x1
maxval = x2
maxval = 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:
    maxval = 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 **maxval** 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:
    maxval = x1
elif x2 >= x1 and x2 >= x3:
    maxval = x2
else:
    maxval = x3
```

- We're comparing each possible value against all the others to determine which one is largest.

Strategy 1:
Compare Each to All

- NowWhat 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.

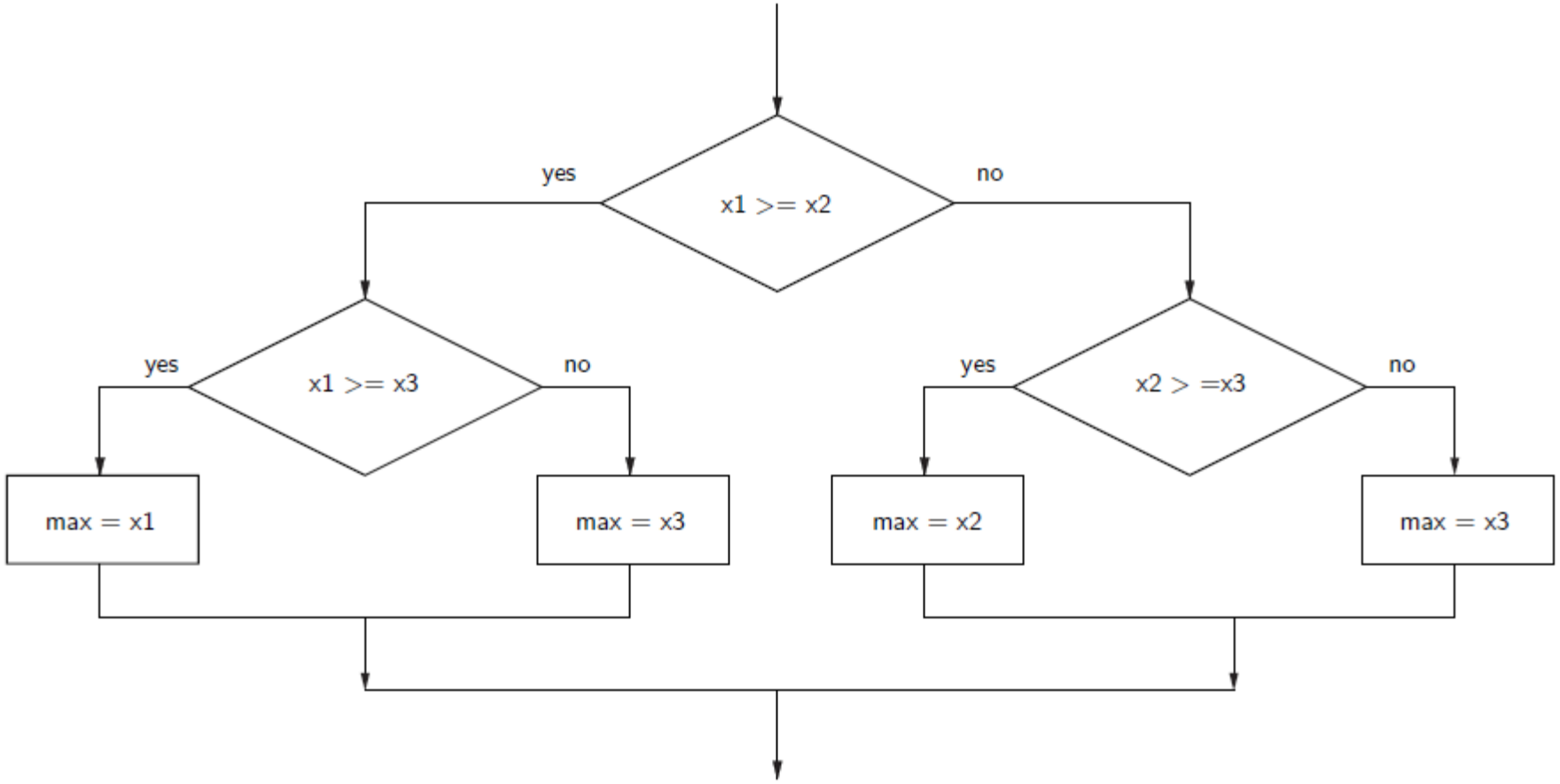
Strategy 2:

Decision Tree

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

Strategy 2:

Decision Tree



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-elses` nested three levels deep with eight assignment statements!

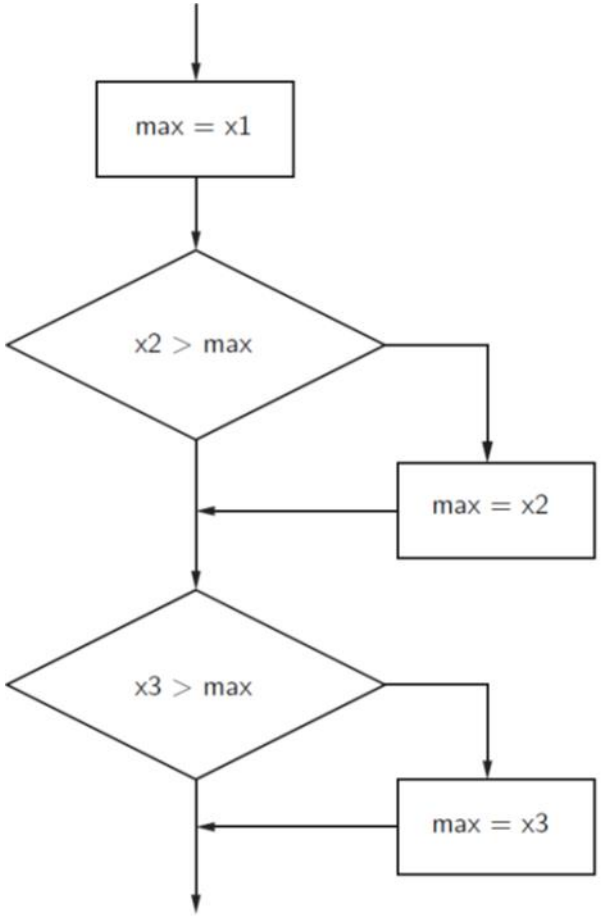
Strategy 3:

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.

```
maxval = x1
if x2 > maxval:
    maxval = x2
if x3 > maxval:
    maxval = x3
```

Strategy 3:

Sequential Processing

- 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 Processing

```
# program: maxn.py
#   Finds the maximum of a series of numbers

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

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

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

    print("The largest value is", max)
```

Strategy 4:

Use built-in API

- 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

LECTURE 10

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 some 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.

Homework

LECTURE 10

Homework 7

Read Chapter Summary

Work on True/False Questions, Multiple Choice Questions, Discussion.

Work on Programming Exercises: 5, 8, and 11

(note: Programming Exercises 2, 3 is the same as chapter 5)

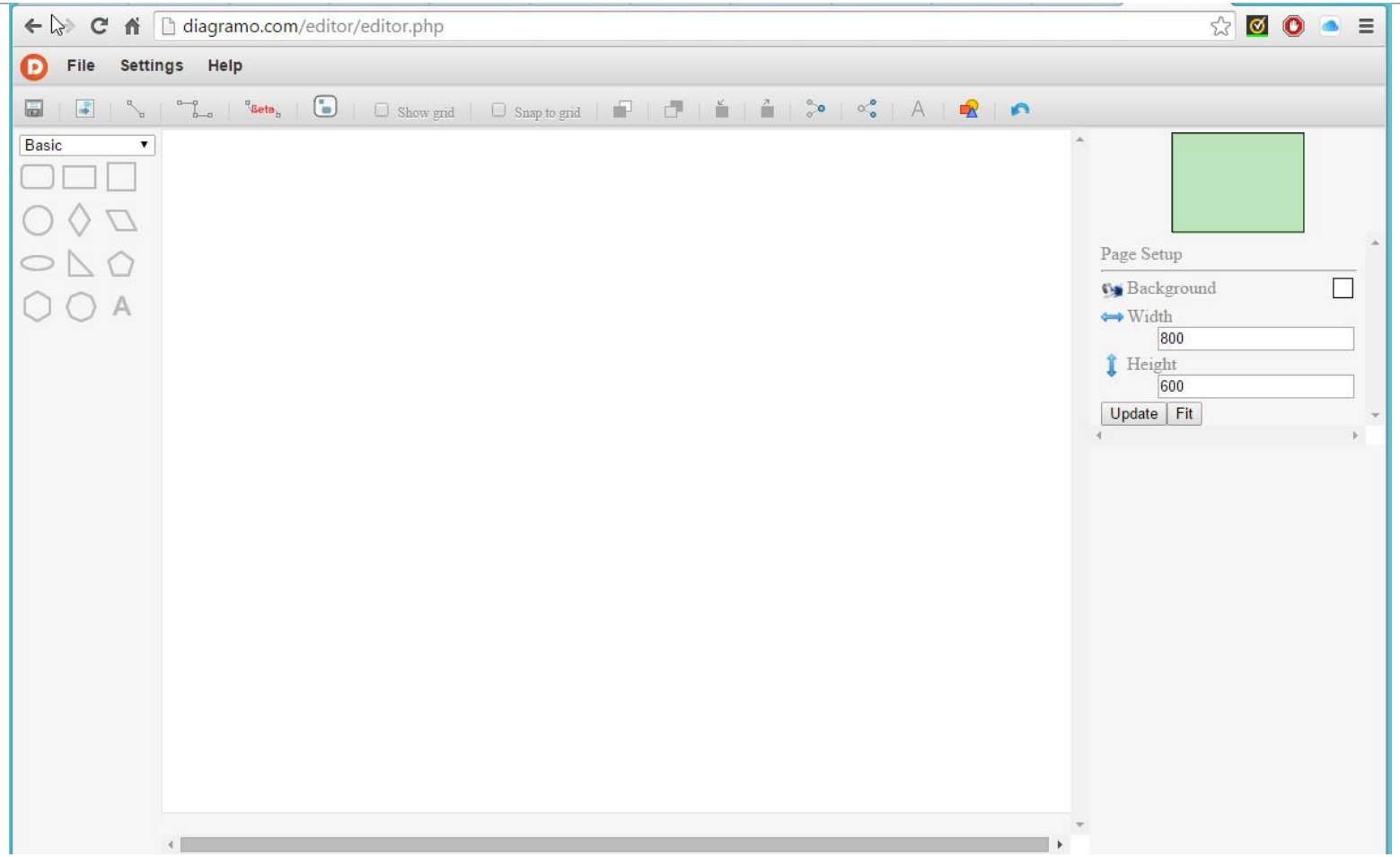
Appendix

LECTURE 12

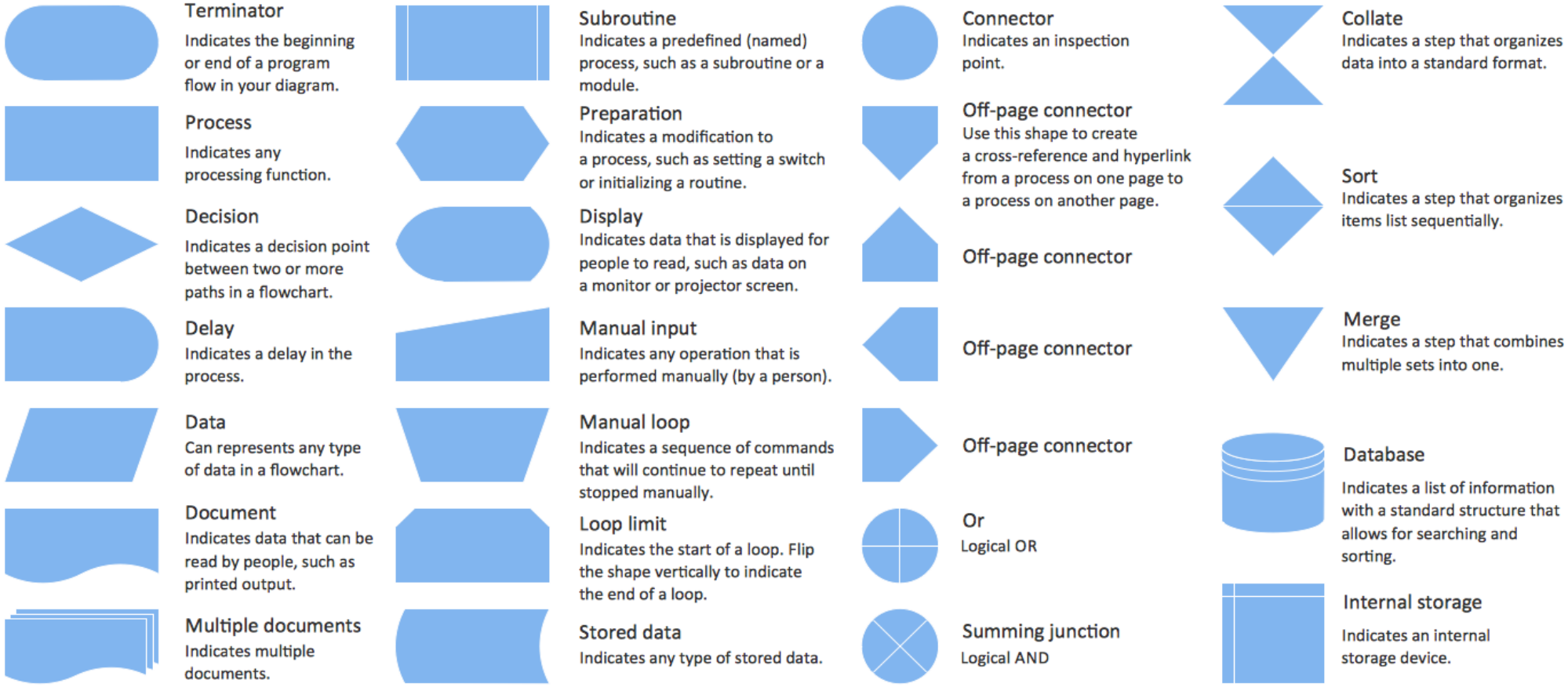
Flowchart Tool:

Design Tool for Structural Programming

http://diagramo.com/ (HTML5 Version) or Dia Diagram Editor (Windows Version)



Object Oriented Programming is Higher Level Design (Module and Package)
Structural Programming is for Module Contents



Terminator
Indicates the beginning or end of a program flow in your diagram.

Process
Indicates any processing function.

Decision
Indicates a decision point between two or more paths in a flowchart.

Delay
Indicates a delay in the process.

Data
Can represents any type of data in a flowchart.

Document
Indicates data that can be read by people, such as printed output.

Multiple documents
Indicates multiple documents.

Subroutine
Indicates a predefined (named) process, such as a subroutine or a module.

Preparation
Indicates a modification to a process, such as setting a switch or initializing a routine.

Display
Indicates data that is displayed for people to read, such as data on a monitor or projector screen.

Manual input
Indicates any operation that is performed manually (by a person).

Manual loop
Indicates a sequence of commands that will continue to repeat until stopped manually.

Loop limit
Indicates the start of a loop. Flip the shape vertically to indicate the end of a loop.

Stored data
Indicates any type of stored data.

Connector
Indicates an inspection point.

Off-page connector
Use this shape to create a cross-reference and hyperlink from a process on one page to a process on another page.

Off-page connector

Off-page connector

Off-page connector

Or
Logical OR

Summing junction
Logical AND

Collate
Indicates a step that organizes data into a standard format.

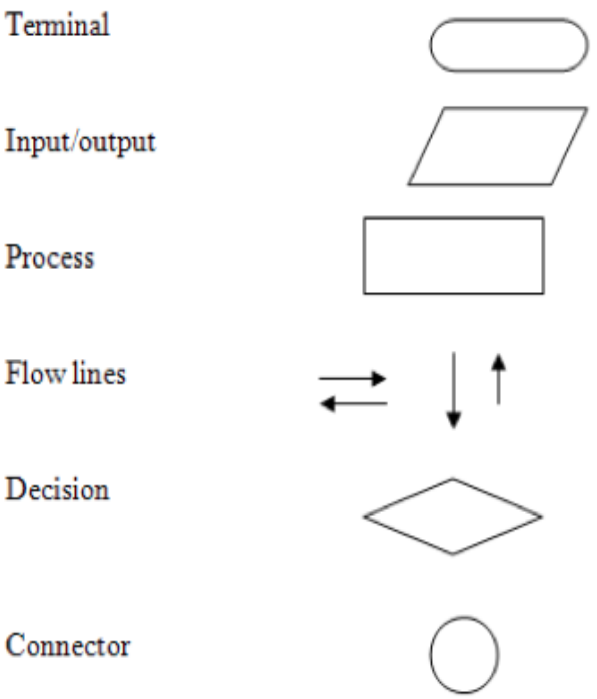
Sort
Indicates a step that organizes items list sequentially.

Merge
Indicates a step that combines multiple sets into one.

Database
Indicates a list of information with a standard structure that allows for searching and sorting.

Internal storage
Indicates an internal storage device.

Flowchart Examples



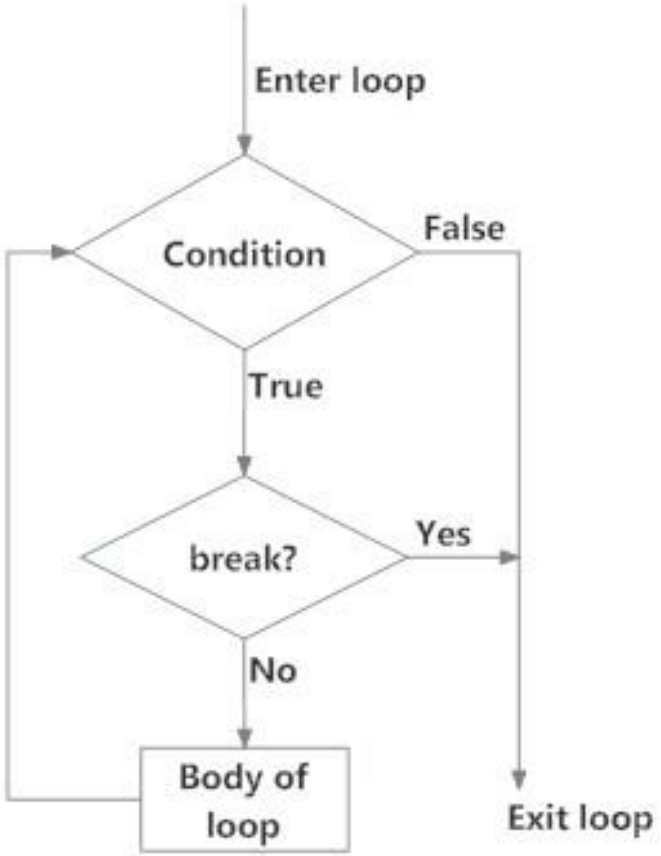
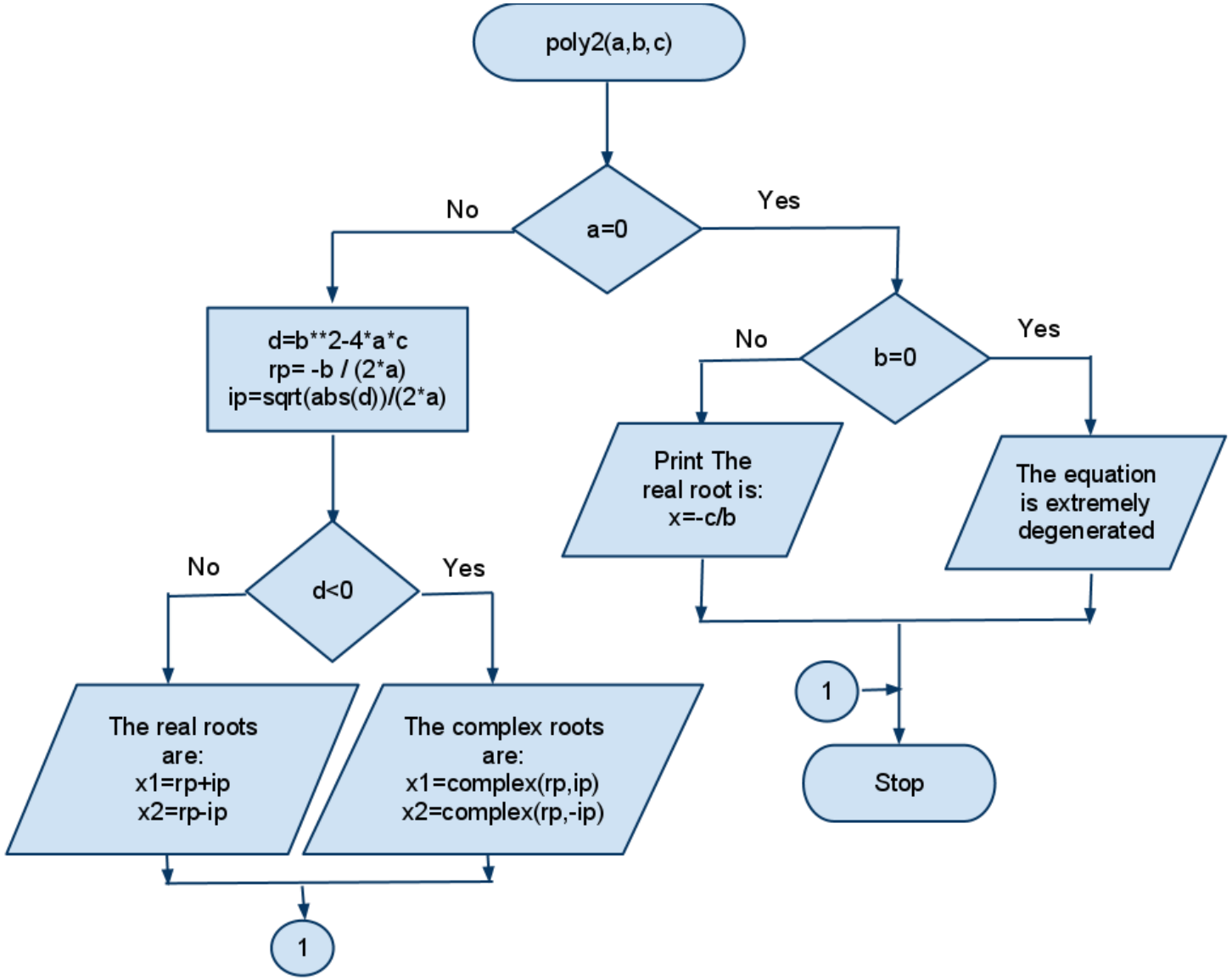


Fig: flowchart of break


poly2(a,b,c)
a=0
No
Yes
d=b**2-4*a*c
rp= -b / (2*a)
ip=sqrt(abs(d))/(2*a)
b=0
No
Yes
Print The
real root is:
x=-c/b
The equation
is extremely
degenerated
d<0
No
Yes
The real roots
are:
x1=rp+ip
x2=rp-ip
The complex roots
are:
x1=complex(rp,ip)
x2=complex(rp,-ip)
1
1
Stop

Appendix B

LECTURE 13

Match-Case

switch statement

ACTIVITY

Match Case

- With the introduction of **Python 3.10**, several new features were introduced, and one of them was – python match case. The Python match case is similar to the switch case statement, which is recognized as structural pattern matching in python.

Switch Case Structure

- The switch control block has similar functionality as that of an if-else ladder. However, writing multiple if statements are not the most effective way of doing so. Instead, by using the switch case statement, we can simply combine them into a single structure.
- The switch case statement in a C/C++ program would look something like this:

C++ Switch Statement

```
01  switch (variable to be evaluated):
02  {
03        case value1 : //statement 1
04                      break;
05
06        case value2 : //statement 2
07                      break;
08
09        case value_n : //statement n
10                       break;
11
12        default:  //default statement
13  }
```

Match Case Statement

- To implement switch-case like characteristics and if-else functionalities, we use a match case in python. A match statement will compare a given variable's value to different shapes, also referred to as the pattern. The main idea is to keep on comparing the variable with all the present patterns until it fits into one.

Syntax of Match Case in Python

The match case consists of three main entities :

- The match keyword
- One or more case clauses
- Expression for each case

The case clause consists of a pattern to be matched to the variable, a condition to be evaluated if the pattern matches, and a set of statements to be executed if the pattern matches.

Syntax of Match Case in Python

```
1   match variable_name:
2         case ‘pattern1’ : //statement1
3         case ‘pattern2’ : //statement2
4            …
5         case ‘pattern n’ : //statement n
```

Example of a Match Case Python Statement

```
1   quit_flag = False
2   match quit_flag:
3       case True:
4           print("Quitting")
5           exit()
6       case False:
7           print("System is on")
```

Example of a Match Case Python Statement

- Here, we have a variable named quit_flag which we have assigned a Boolean value of False by default. Thus, we have a match case that will compare the patterns with the ‘quit_flag’ variable.
- We have two cases for two possible values of quit_flag – True and False. For the first case, if the variable is True, then it will print ‘Quittting’ and execute the exit() function to end the program. In case if it is false, it will just print a statement saying that ‘System is on.’

Match Case Python for Function Overloading

- We can also use structural pattern matching for function overloading in python. With function overloading, we have multiple functions with the same name but with a different signature.
- Depending on the case condition, we can access the same function name with different signature values. With function overloading, we can improve the readability of the code.

Match Case Python for Function Overloading

- As of now, python doesn’t support function overloading. The following example can be used to verify it –
- Here, we shall be creating a user-defined function name calc() which accepts one argument. If the argument type is an integer, then we will return the square of the number. Else if the argument type is float, then we will return the cube of a number.

Match Case Python for Function Overloading

```
01   def calc(n:int):
02       return (n**2)
03
04   def calc(n:float):
05       return (n**3)
06   n = 9.5
07   is_int = isinstance(n, int)
08
09   match is_int:
10       case True : print("Square is :",calc(n))
11       case False : print("Cube is:", calc(n))
```

Pattern Values

- The match case in python allows a number, string, ‘None’ value, ‘True’ value, and ‘False’ value as the pattern. Thus, different types of patterns are possible, and we will be looking into some of the possible patterns.

Wildcard Pattern

- We can have a wildcard pattern too. A wildcard pattern is executed when none of the other pattern values are matched.

Following shows an example of a wildcard pattern.

- We will take the same example of quit. But here, instead of passing a Boolean value to the variable, we will pass an integer value to the ***'quit_flag'*** variable.

Wildcard Pattern

```
01  quit_flag = 4
02
03  match quit_flag:
04      case True:
05          print("Quitting")
06          exit()
07      case False:
08          print("System is on")
09      case _:
10          print("Boolean Value was not passed")
```

Wildcard Pattern

- Here the underscore has been used as a wildcard pattern. It will not bind the ‘quit_flag’ variable’s value to anything but match the variable’s value to the statement.

Multiple pattern values using OR operator

- We can also have optional values for a given case. Using the OR operator, we can execute a single expression for multiple possibilities of pattern for the given variable.
- In the below code, we have a variable name ‘sample’. If the variable’s value is of Boolean type, then it will execute a common statement for both the conditions. Else, for any other value, it will print ‘Not a Boolean value’.

Multiple pattern values using OR operator

```
1   sample = True
2
3   match sample:
4       case (True|False):
5           print("It is a boolean value")
6       case _ :
7           print("Not a boolean value")
```

Checking for a collection of values

The pattern can also be a collection of values. It will match the pattern against the entire collection. Let us take a list ‘list1’ as an example. We will take the entire list collection as the pattern.

```
1   list1 = ['a', 'b', 'c', 'd']
2
3   match list1:
4       case ['e','f'] : print("e,f present")
5       case ['a','b','c','d'] : print("a,b,c,d present")
```

Named constants as patterns

- We can have a named constant as a pattern for match case statements. The constant should be a qualified name addressed by a dot operator. It works like a literal, but it never minds.

```
01  class switch:
02      on = 1
03      off = 0
04
05  status = 0
06
07  match status:
08      case switch.on :
09          print('Switch is on')
10      case switch.off :
11          print('Switch is off')
```

Inline if statement in match case format

- We can also add an if statement to a pattern for a match case in python. That if the condition is also known as ‘guard’. The expression for a given pattern would be evaluated only if the guard is True. If the guard is False, it does not execute that pattern’s statement.
- Let us take an example for the same. We have a variable ‘n’ will is assigned a numerical value. We have three cases here – n is negative, n is zero, and n is positive. The match case will check the guard and accordingly print the statement.

Inline if statement in match case format

```
1   n = 0
2   match n:
3       case n if n < 0:
4           print("Number is negative")
5       case n if n == 0:
6           print("Number is zero")
7       case n if n > 0:
8           print("Number is positive")
```