

# Chapter Three

---

PART ONE: DECISIONS, RELATIONAL OPERATORS

# Chapter Goals

---

- To implement decisions using the if statement
- To compare integers, floating-point numbers, and Strings
- To write statements using the Boolean data type
- To develop strategies for testing your programs
- To validate user input

***In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.***

# Contents

---

- The **if** Statement
- Relational Operators
- Nested Branches
- Multiple Alternatives
- Problem Solving: Flowcharts
- Problem Solving: Test Cases
- Boolean Variables and Operators
- Analyzing Strings
- Application: Input Validation

# The **if** Statement

---

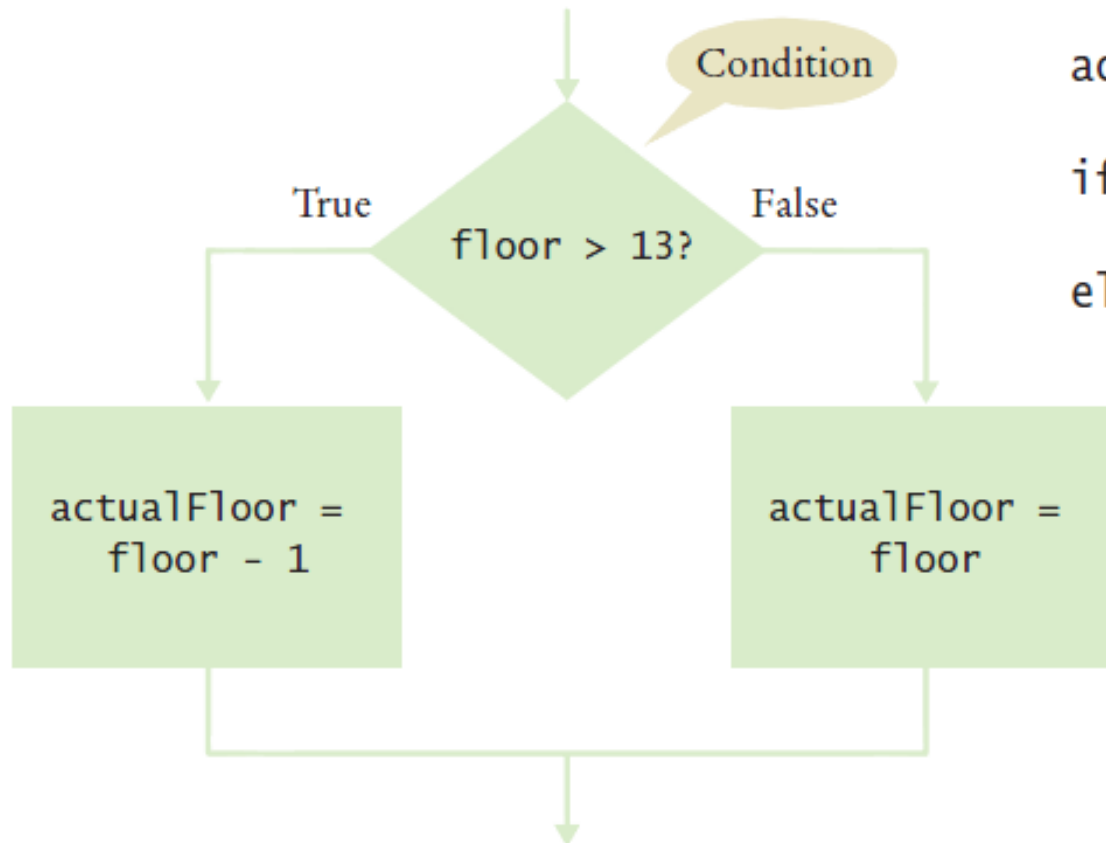
- A computer program often needs to make decisions based on input, or circumstances
- For example, buildings often ‘skip’ the 13<sup>th</sup> floor, and elevators should too
  - The 14<sup>th</sup> floor is really the 13<sup>th</sup> floor
  - So every floor above 12 is really ‘floor - 1’
    - If  $\text{floor} > 12$ ,  $\text{Actual floor} = \text{floor} - 1$
- The two keywords of the if statement are:
  - **if**
  - **else**

The **if** statement allows a program to carry out different actions depending on the nature of the data to be processed.

# Flowchart of the **if** Statement

---

- One of the two branches is executed once
  - True (**if**) branch or False (**else**) branch



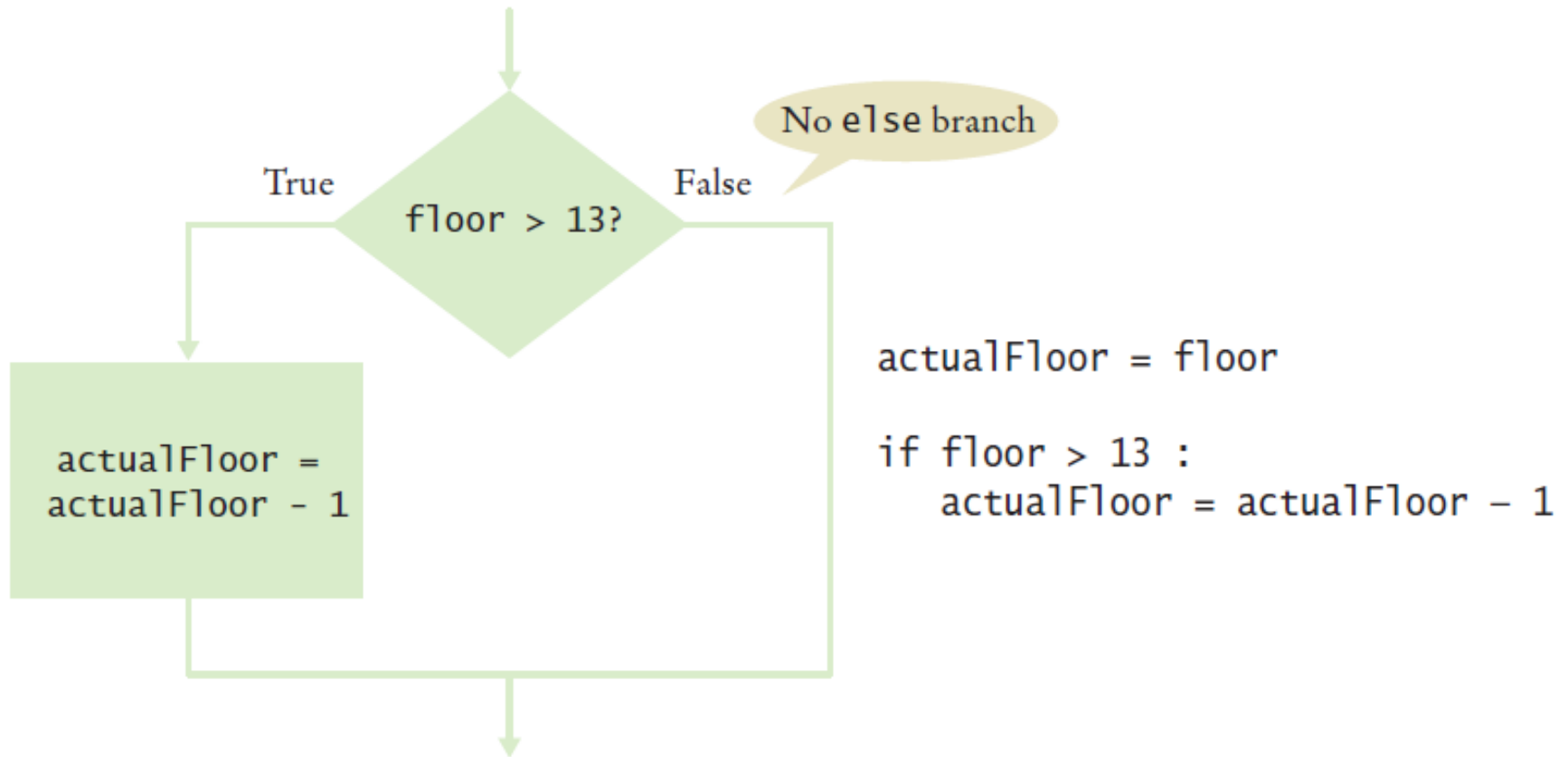
`actualFloor = 0`

```
if floor > 13 :  
    actualFloor = floor - 1  
else :  
    actualFloor = floor
```

# Flowchart with only a True Branch

---

- An **if** statement may not need a 'False' (**else**) branch



# Syntax 3.1: The **if** Statement

Syntax `if condition :`  
`statements`

`if condition :`  
`statements1`  
`else :`  
`statements2`

A condition that is true or false.  
Often uses relational operators:

`==` `!=` `<` `<=` `>` `>=`

(See page 98.)

```
if floor > 13 :  
    actualFloor = floor - 1  
else :  
    actualFloor = floor
```

The colon indicates  
a compound statement.

If the condition is true, the statement(s)  
in this branch are executed in sequence;  
if the condition is false, they are skipped.

If the condition is false, the statement(s)  
in this branch are executed in sequence;  
if the condition is true, they are skipped.

Omit the else branch  
if there is nothing to do.

The if and else  
clauses must  
be aligned.



# Elevatorsim.py

---

```
1  ##
2  # This program simulates an elevator panel that skips the 13th floor.
3  #
4
5  # Obtain the floor number from the user as an integer.
6  floor = int(input("Floor: "))
7
8  # Adjust floor if necessary.
9  if floor > 13 :
10     actualFloor = floor - 1
11 else :
12     actualFloor = floor
13
14 # Print the result.
15 print("The elevator will travel to the actual floor", actualFloor)
```

## Program Run

```
Floor: 20
The elevator will travel to the actual floor 19
```



# Our First Example

---

- Open the file:
  - elevatorsim.py
  - This is a slightly modified program
- Run the program
  - Try a value that is less than 13
    - What is the result?
  - Run the program again with a value that is greater than 13
    - What is the result?
- What happens if you enter 13?

# Our First Example (2)

---

- Revised Problem Statement (1):
  - Check the input entered by the user:
  - If the input is 13, set the value to 14 and print a message
  - Modify the elevatorsim program to test the input

***The relational operator for equal is “==”***

- Modified Problem Statement (2)
  - In some countries the number 14 is considered unlucky.
  - What is the revised algorithm?
  - Modify the elevatorsim program to “skip” both the 13<sup>th</sup> and 14<sup>th</sup> floor

# Compound Statements

---

- Some constructs in Python are **compound statements**.
- **compound statements** span multiple lines and consist of a *header* and a statement block

## **The if statement is an example of a compound statement**

- Compound statements require a colon “:” at the end of the header.
- The statement block is a group of one or more statements, *all indented to the same column*
- The statement block ***starts on the line after the header and ends at the first statement indented less than the first statement in the block***

# Compound Statements

---

- Statement blocks can be nested inside other types of blocks (we will learn about more blocks later)
- Statement blocks signal that one or more statements are part of a given compound statement
- In the case of the if construct the statement block specifies:
  - The instructions that are executed if the condition is true
  - Or skipped if the condition is false

**Statement blocks are visual cues that allow you to follow the logic and flow of a program**

# Tips on Indenting Blocks

---

- Let IDLE do the indenting for you...

```
if totalSales > 100.0 :
↑ discount = totalSales * 0.05
| totalSales = totalSales - discount
| print("You received a discount of $%.2f" % discount)
else :
↑ diff = 100.0 - totalSales
| if diff < 10.0 :
| ↑ print("If you were to purchase our item of the day you can receive a 5% discount.")
| else :
| ↑ print("You need to spend $%.2f more to receive a 5% discount." % diff)
| | ↑
| | |
0 1 2 Indentation level
```

***This is referred to as “block structured” code. Indenting consistently is not only syntactically required in Python, it also makes code much easier to follow.***

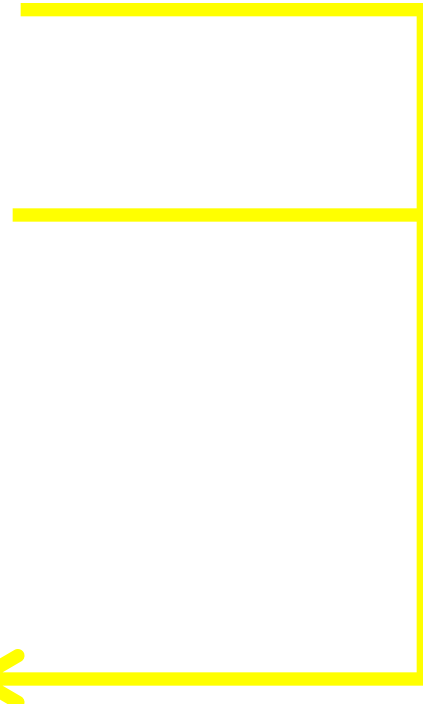
# A Common Error

---

- Avoid duplication in branches
- If the same code is duplicated in each branch then move it out of the **if** statement.

```
if floor > 13 :  
    actualFloor = floor - 1  
    print("Actual floor:", actualFloor)  
else :  
    actualFloor = floor  
    print("Actual floor:", actualFloor)
```

```
if floor > 13 :  
    actualFloor = floor - 1  
else :  
    actualFloor = floor  
print("Actual floor:", actualFloor)
```



# The Conditional Operator

---

- A “shortcut” you may find in existing code
  - It is not used in this book
  - The shortcut notation *can* be used anywhere that a value is expected

True branch                      Condition                      False branch

`actualFloor = floor - 1 if floor > 13 else floor`

`print("Actual floor:", floor - 1 if floor > 13 else floor)`

The diagram shows three yellow brackets above the code. The first bracket spans from the start of the code to the end of the first branch ('- 1'), labeled 'True branch'. The second bracket spans from the start of the condition ('if floor > 13') to the end of the condition, labeled 'Condition'. The third bracket spans from the start of the second branch ('else floor') to the end of the second branch, labeled 'False branch'.

**Complexity is BAD....**

**This “shortcut” is difficult to read and a poor programming practice**

# Relational Operators

---

- Every **if** statement has a condition
  - Usually compares two values with an operator

```
if floor > 13 :  
    ..  
if floor >= 13 :  
    ..  
if floor < 13 :  
    ..  
if floor <= 13 :  
    ..  
if floor == 13 :  
    ..
```

Table 1 Relational Operators

Python	Math Notation	Description
>	>	Greater than
>=	$\geq$	Greater than or equal
<	<	Less than
<=	$\leq$	Less than or equal
==	=	Equal
!=	$\neq$	Not equal



# Assignment vs. Equality Testing

---

- Assignment: *makes* something true.

```
floor = 13
```

- Equality testing: *checks* if something is true.

```
if floor == 13 :
```

# Comparing Strings

---

- Checking if two strings are equal

```
if name1 == name2 :  
    print("The strings are identical")
```

- Checking if two strings are not equal

```
if name1 != name2 :  
    print("The strings are not identical")
```

# Checking for String Equality (1)

---

- For two strings to be equal, they must be of the same length and contain the same sequence of characters:

name1 = J o h n   W a y n e

name2 = J o h n   W a y n e

# Checking for String Equality (2)

---

- If any character is different, the two strings will not be equal:

name1 = J o h n    W a y n e

name1 = J o h n    W a y n e

name2 = J a n e    W a y n e




name2 = J o h n    w a y n e

The sequence “ane”  
does not equal “ohn”

An uppercase “W” is not  
equal to lowercase “w”

# Relational Operator Examples (1)



Table 2 Relational Operator Examples

Expression	Value	Comment
$3 \leq 4$	True	3 is less than 4; $\leq$ tests for “less than or equal”.
 $3 \leq 4$	<b>Error</b>	The “less than or equal” operator is $\leq$ , not $\leq$ . The “less than” symbol comes first.
$3 > 4$	False	$>$ is the opposite of $\leq$ .
$4 < 4$	False	The left-hand side must be strictly smaller than the right-hand side.
$4 \leq 4$	True	Both sides are equal; $\leq$ tests for “less than or equal”.
$3 == 5 - 2$	True	$==$ tests for equality.
$3 != 5 - 1$	True	$!=$ tests for inequality. It is true that 3 is not $5 - 1$ .
 $3 = 6 / 2$	<b>Error</b>	Use $==$ to test for equality.
$1.0 / 3.0 == 0.333333333$	False	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 101.
 $"10" > 5$	<b>Error</b>	You cannot compare a string to a number.

# Relational Operator Examples (2)

---

**Table 2** Relational Operator Examples

 <code>3 = 6 / 2</code>	<b>Error</b>	Use <code>==</code> to test for equality.
<code>1.0 / 3.0 == 0.333333333</code>	False	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 101.
 <code>"10" &gt; 5</code>	<b>Error</b>	You cannot compare a string to a number.

# Another Example

---

- Open the file:
  - `compare.py`
- Run the program
  - What are the results?

# Common Error (Floating Point)

---

- Floating-point numbers have only a limited precision, and calculations can introduce roundoff errors.
- You must take these inevitable roundoffs into account when comparing floating point numbers.



# Common Error (Floating Point, 2)

---

- For example, the following code multiplies the square root of 2 by itself.
- Ideally, we expect to get the answer 2:

```
r = math.sqrt(2.0)
if r * r == 2.0 :
    print("sqrt(2.0) squared is 2.0")
else :
    print("sqrt(2.0) squared is not 2.0 but", r * r)
```

Output:

```
sqrt(2.0) squared is not 2.0 but 2.000000000000000004
```

# The Use of EPSILON

---

- Use a very small value to compare the difference to determine if floating-point values are *'close enough'*
  - The magnitude of their difference should be less than some threshold
  - Mathematically, we would write that  $x$  and  $y$  are close enough if:

$$|x - y| < \epsilon$$

```
EPSILON = 1E-14
r = math.sqrt(2.0)
if abs(r * r - 2.0) < EPSILON :
    print("sqrt(2.0) squared is approximately 2.0")
```

# Lexicographical Order

---

- To compare Strings in 'dictionary' like order:  
    `string1 < string2`
- Notes
  - All UPPERCASE letters come before lowercase
  - 'space' comes before all other printable characters
  - Digits (0-9) come before all letters
  - See Appendix A for the Basic Latin (ASCII) Subset of Unicode

# Operator Precedence

---

- The comparison operators have lower precedence than arithmetic operators
  - ***Calculations are done before the comparison***
  - Normally your calculations are on the 'right side' of the comparison or assignment operator

Calculations

```
actualFloor = floor + 1
```

```
if floor > height + 1 :
```

# Implementing an **if** Statement (1)

---

- 1) Decide on a branching condition

**original price < 128?**

- 2) Write pseudocode for the true branch

**discounted price = 0.92 x original price**

- 3) Write pseudocode for the false branch

**discounted price = 0.84 x original price**

# Implementing an **if** Statement (2)

---

- 4) Double-check relational operators
- Test values below, at, and above the comparison (127, 128, 129)

5) Remove duplication  
**discounted price = \_\_\_\_ x original price**

6) Test both branches  
**discounted price = 0.92 x 100 = 92**  
**discounted price = 0.84 x 200 = 168**

# Implementing an **if** Statement (3)

---

7. Write the code in Python

# A Third Example

---

- The university bookstore has a Kilobyte Day sale every October 24 (10.24), giving an 8 percent discount on all computer accessory purchases if the price is less than \$128, and a 16 percent discount if the price is at least \$128.

```
if originalPrice < 128 :  
    discountRate = 0.92  
else :  
    discountRate = 0.84  
discountedPrice = discountRate * originalPrice
```



# The Sale Example

---

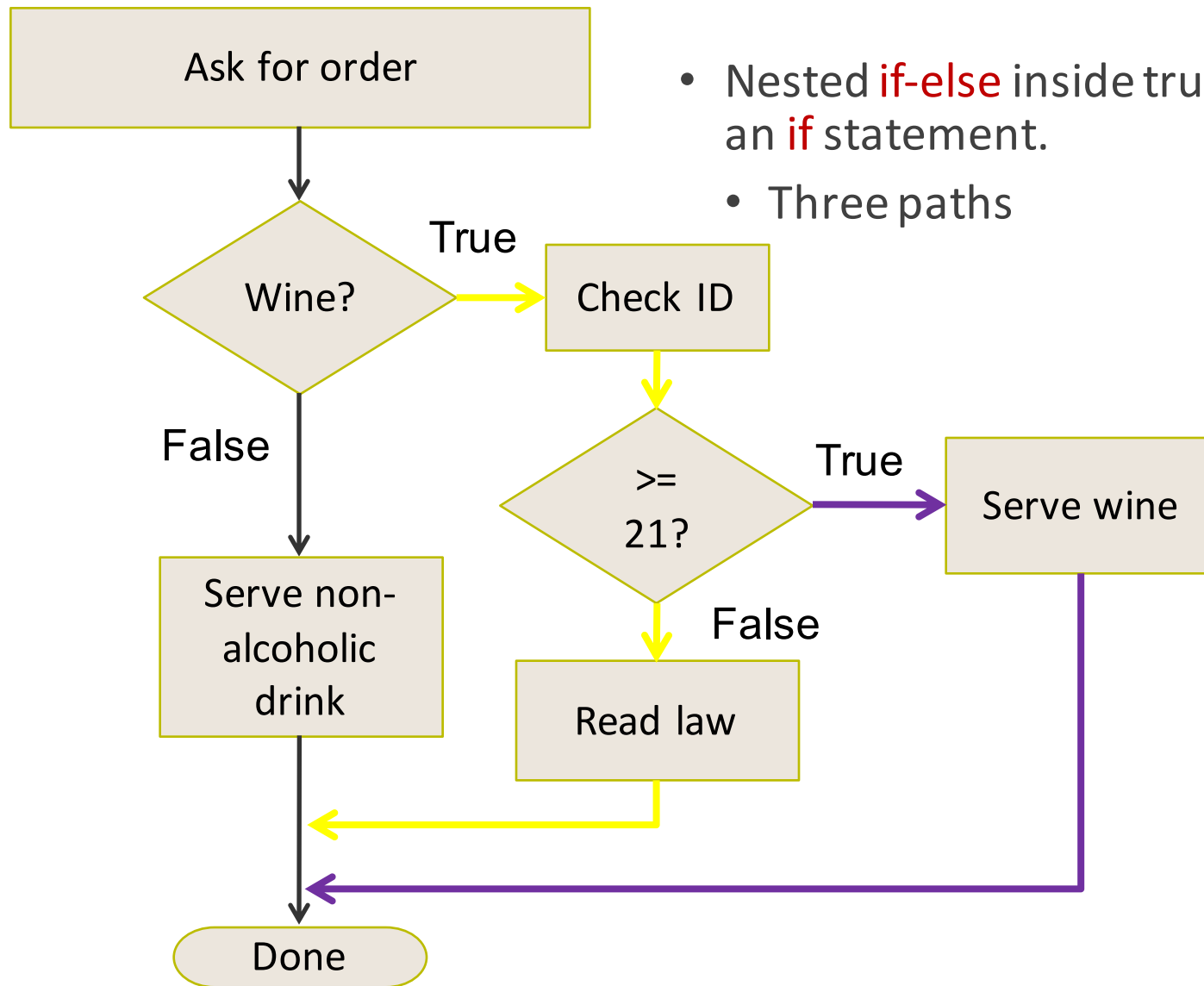
- Open the file:
  - sale.py
- Run the program several time using different values
  - Use values less than 128
  - Use values greater that 128
  - Enter 128
- What results do you get?

# Nested Branches

---

- You can *nest* an **if** inside either branch of an **if** statement.
- Simple example: Ordering drinks
  - Ask the customer for their drink order
  - **if** customer orders wine
    - Ask customer for ID
    - **if** customer's age is 21 or over
      - Serve wine
    - Else
      - Politely explain the law to the customer
  - Else
    - Serve customers a non-alcoholic drink

# Flowchart of a Nested **if**



- Nested **if-else** inside true branch of an **if** statement.
- Three paths

# Tax Example: nested ifs

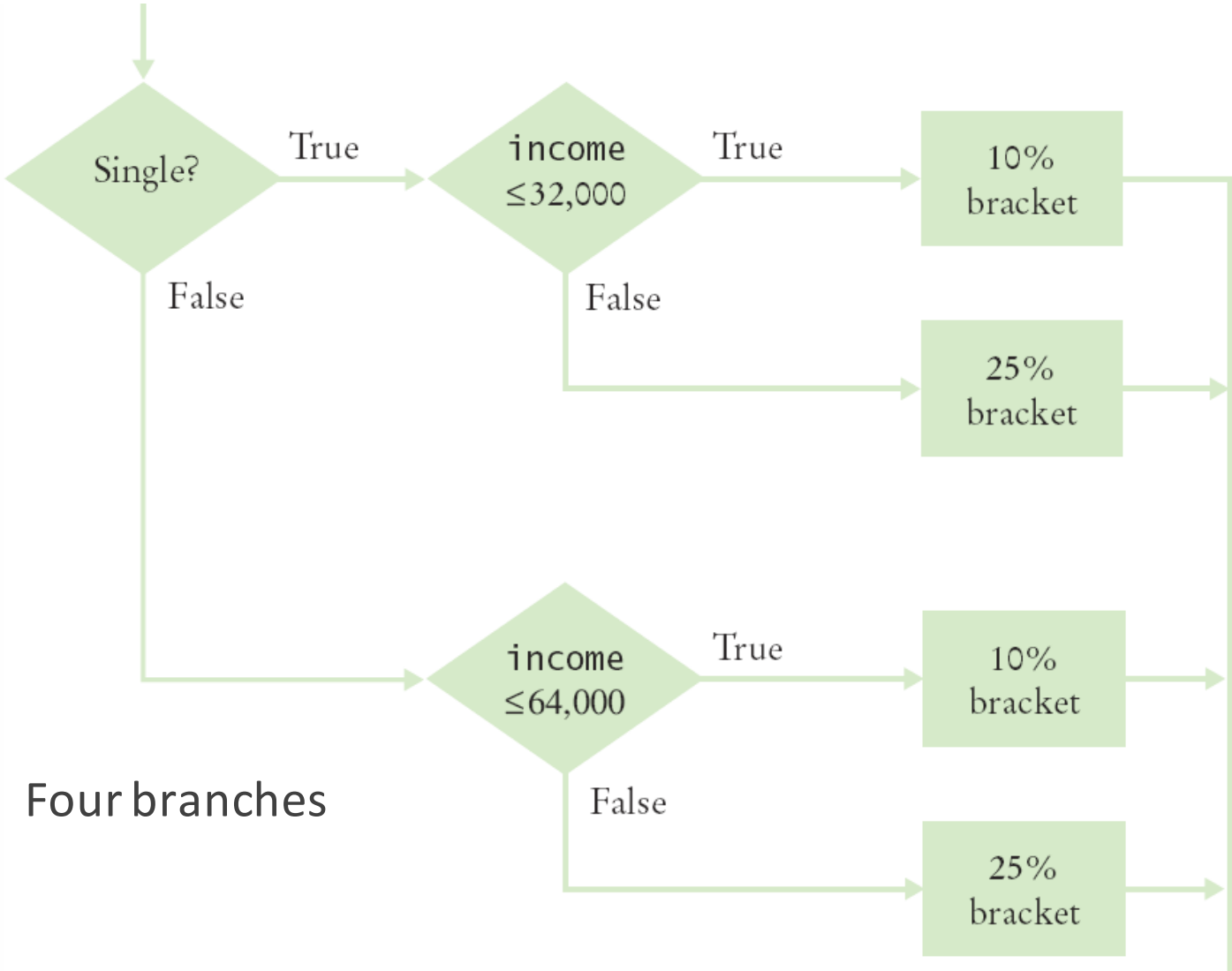
---

- Four outcomes (branches)

- Single
  - $\leq 32000$
  - $> 32000$
- Married
  - $\leq 64000$
  - $> 64000$

If your status is Single and if the taxable income is	the tax is	of the amount over
at most \$32,000	10%	\$0
over \$32,000	\$3,200 + 25%	\$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most \$64,000	10%	\$0
over \$64,000	\$6,400 + 25%	\$64,000

# Flowchart for the Tax Example



- Four branches

# Taxes.py (1)

---

```
1  ##
2  # This program computes income taxes, using a simplified tax schedule.
3  #
4
5  # Initialize constant variables for the tax rates and rate limits.
6  RATE1 = 0.10
7  RATE2 = 0.25
8  RATE1_SINGLE_LIMIT = 32000.0
9  RATE1_MARRIED_LIMIT = 64000.0
10
11 # Read income and marital status.
12 income = float(input("Please enter your income: "))
13 maritalStatus = input("Please enter s for single, m for married: ")
14
15 # Compute taxes due.
16 tax1 = 0.0
17 tax2 = 0.0
18
19 if maritalStatus == "s" :
20     if income <= RATE1_SINGLE_LIMIT :
21         tax1 = RATE1 * income
22     else :
23         tax1 = RATE1 * RATE1_SINGLE_LIMIT
24         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
25 else :
26     if income <= RATE1_MARRIED_LIMIT :
27         tax1 = RATE1 * income
28     else :
29         tax1 = RATE1 * RATE1_MARRIED_LIMIT
30         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
31
32 totalTax = tax1 + tax2
33
```

# Taxes.py (2)

---

- The 'True' branch (Single)
  - Two branches within this branch

```
19 if maritalStatus == "s" :  
20     if income <= RATE1_SINGLE_LIMIT :  
21         tax1 = RATE1 * income  
22     else :  
23         tax1 = RATE1 * RATE1_SINGLE_LIMIT  
24         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
```

# Taxes.py (3)

---

- The 'False' branch (Married)

```
else :  
    if income <= RATE1_MARRIED_LIMIT :  
        tax1 = RATE1 * income  
    else :  
        tax1 = RATE1 * RATE1_MARRIED_LIMIT  
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```



# Running the Tax Example

---

- Open the file:
  - taxes.py
- Run the program several time using different values for income and marital status
  - Use income values less than \$32,000
  - Use income values greater than \$64,000
  - Enter “&” as the marital status
- What results do you get?

# Hand-tracing

---

- Hand-tracing helps you understand whether a program works correctly
- Create a table of key variables
  - Use pencil and paper to track their values
- Works with pseudocode or code
  - Track location with a marker
- Use example input values that:
  - You know what the correct outcome should be
  - Will test each branch of your code

# Hand-tracing the Tax Example

---

tax1	tax2	income	marital status
0	0		

- Setup
  - Table of variables
  - Initial values

```
6 RATE1 = 0.10
7 RATE2 = 0.25
8 RATE1_SINGLE_LIMIT = 32000.0
9 RATE1_MARRIED_LIMIT = 64000.0
```

```
15 # Compute taxes due.
16 tax1 = 0.0
17 tax2 = 0.0
```

# Hand-tracing the Tax Example (2)

---

tax1	tax2	income	marital status
0	0	80000	m

- Input variables
  - From user
  - Update table

```
11 # Read income and marital status.  
12 income = float(input("Please enter your income: "))  
13 maritalStatus = input("Please enter s for single, m for married: ")
```

- Because marital status is not "s" we skip to the else on line 25

```
19 if maritalStatus == "s" :
```

```
25 else :
```

# Hand-tracing the Tax Example (3)

---

- Because income is not  $\leq 64000$ , we move to the else clause on line 28
  - Update variables on lines 29 and 30
  - Use constants

```
26  if income <= RATE1_MARRIED_LIMIT :  
27      tax1 = RATE1 * income  
28  else :  
29      tax1 = RATE1 * RATE1_MARRIED_LIMIT  
30      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```

tax1	tax2	income	marital status
<del>0</del>	<del>0</del>	80000	m
6400	4000		

# Incremental Code and Test

---

- Using the flag problem statement as an example:
  - Compute the data for the first panel
  - Print out the data
    - Color
    - The X and Y coordinates of the top left corner of the panel
    - The width of the panel
    - The height of the panel
  - Check the data
    - If the data is correct:
      - Draw the panel
    - Else
      - Look at your equations
      - Find and fix any errors
      - Check the data again
  - Do the next panel

# Multiple Alternatives

---

## 3.4 Multiple Alternatives

---

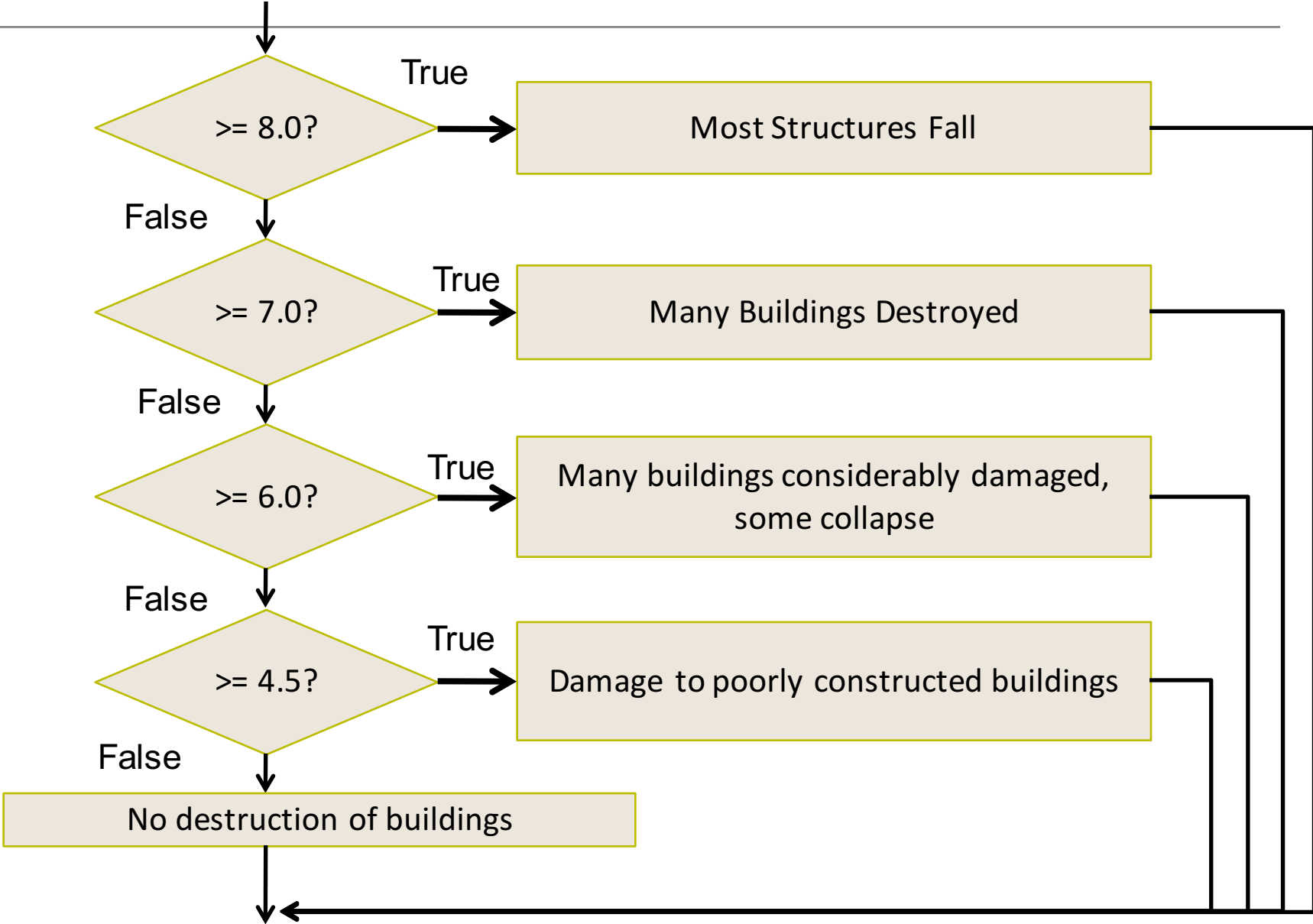
- What if you have more than two branches?
- Count the branches for the following earthquake effect example:
  - 8 (or greater)
  - 7 to 7.99
  - 6 to 6.99
  - 4.5 to 5.99
  - Less than 4.5

When using multiple **if** statements, test the general conditions after the more specific conditions.

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings



# Flowchart of Multiway Branching



# elif Statement

---

- Short for Else, if...
- As soon as one on the test conditions succeeds, the statement block is executed
  - No other tests are attempted
- If none of the test conditions succeed the final else clause is executed

# if, elif Multiway Branching

```
if richter >= 8.0 :    # Handle the 'special case' first
    print("Most structures fall")
elif richter >= 7.0 :
    print("Many buildings destroyed")
elif richter >= 6.0 :
    print("Many buildings damaged, some collapse")
elif richter >= 4.5 :
    print("Damage to poorly constructed buildings")
else :                # so that the 'general case' can be handled last
    print("No destruction of buildings")
```

# What is Wrong With This Code?

---

```
if richter >= 8.0 :  
    print("Most structures fall")  
if richter >= 7.0 :  
    print("Many buildings destroyed")  
if richter >= 6.0 :  
    print("Many buildings damaged, some collapse")  
if richter >= 4.5 :  
    print("Damage to poorly constructed buildings")
```

# earthquake Example

---

- Open the file:
  - earthquake.py
- Run the program with several different inputs

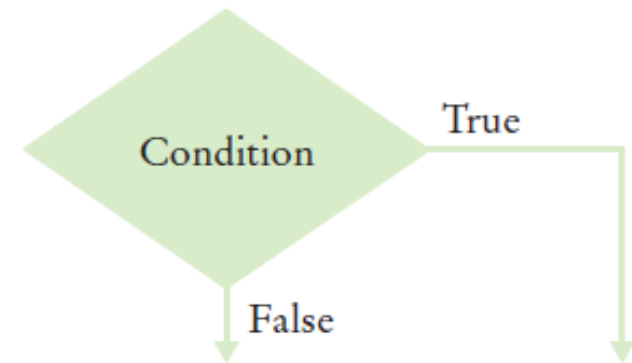
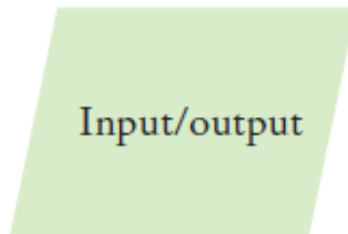
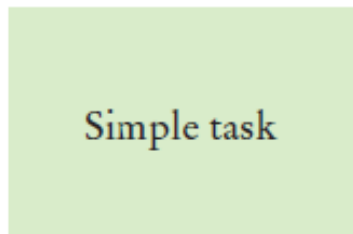
# Using Flowcharts to Develop and Refine Algorithms

---

# 3.5 Problem Solving: Flowcharts

---

- You have seen a few basic flowcharts
- A flowchart shows the structure of decisions and tasks to solve a problem
- Basic flowchart elements:



- Connect them with arrows
  - But never point an arrow inside another branch!

Each branch of a decision can contain tasks and further decisions

# Using Flowcharts

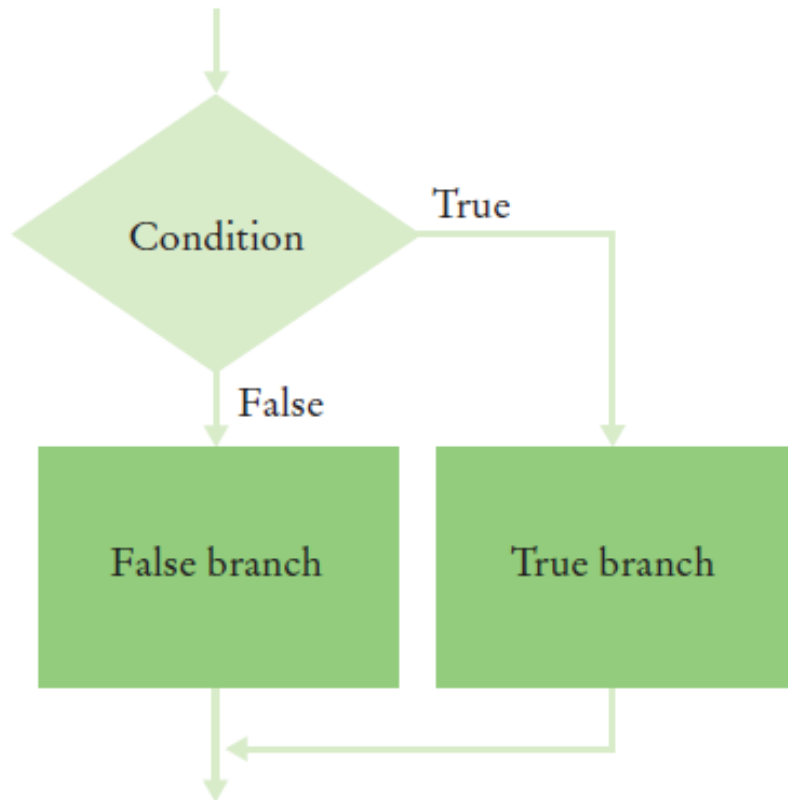
---

- Flowcharts are an excellent tool
- They can help you visualize the flow of your algorithm
- Building the flowchart
  - Link your tasks and input / output boxes in the sequence they need to be executed
  - When you need to make a decision use the diamond (a conditional statement) with two outcomes
  - Never point an arrow inside another branch

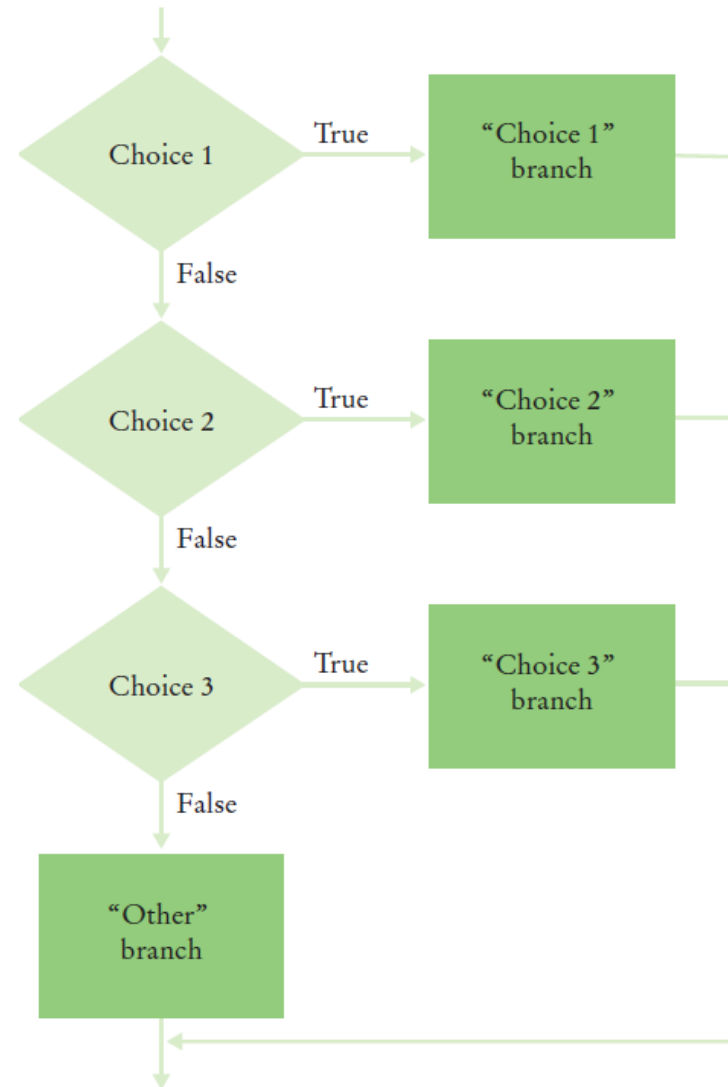


# Conditional Flowcharts

Two Outcomes



Multiple Outcomes

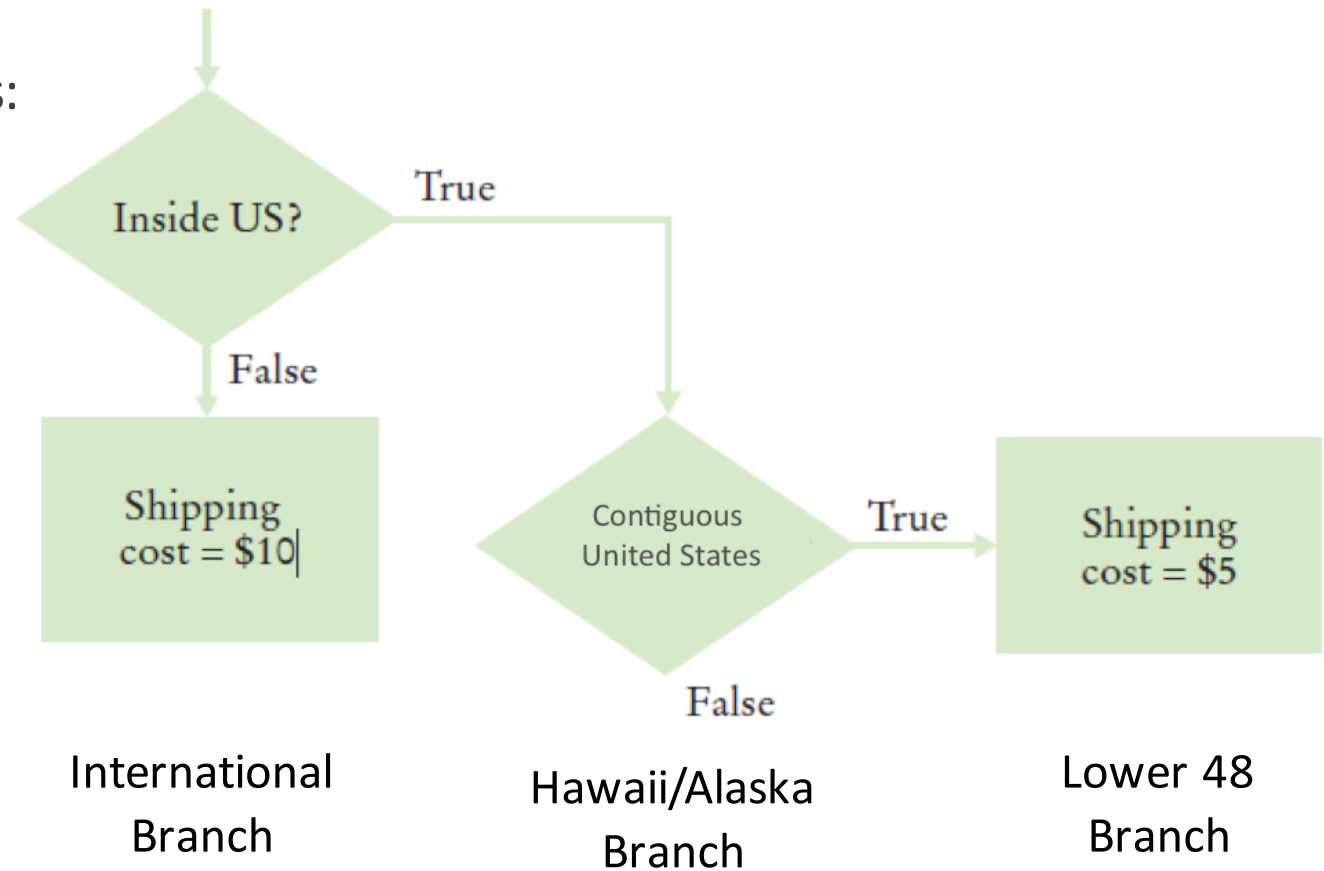


# Shipping Cost flowchart

---

Shipping costs are \$5 inside the contiguous the United States (Lower 48 states), and \$10 to Hawaii and Alaska. International shipping costs are also \$10.

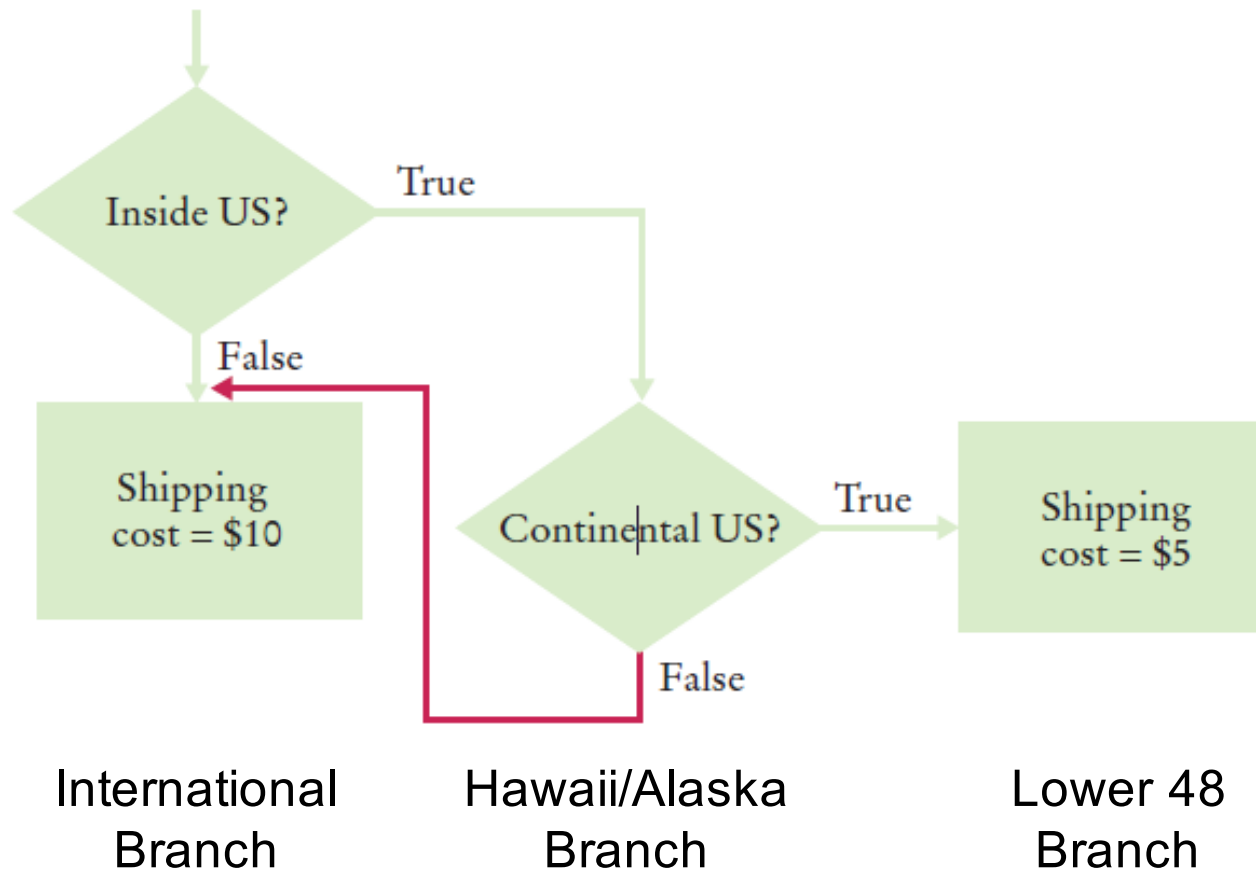
- Three Branches:



# Don't Connect Branches!

Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.

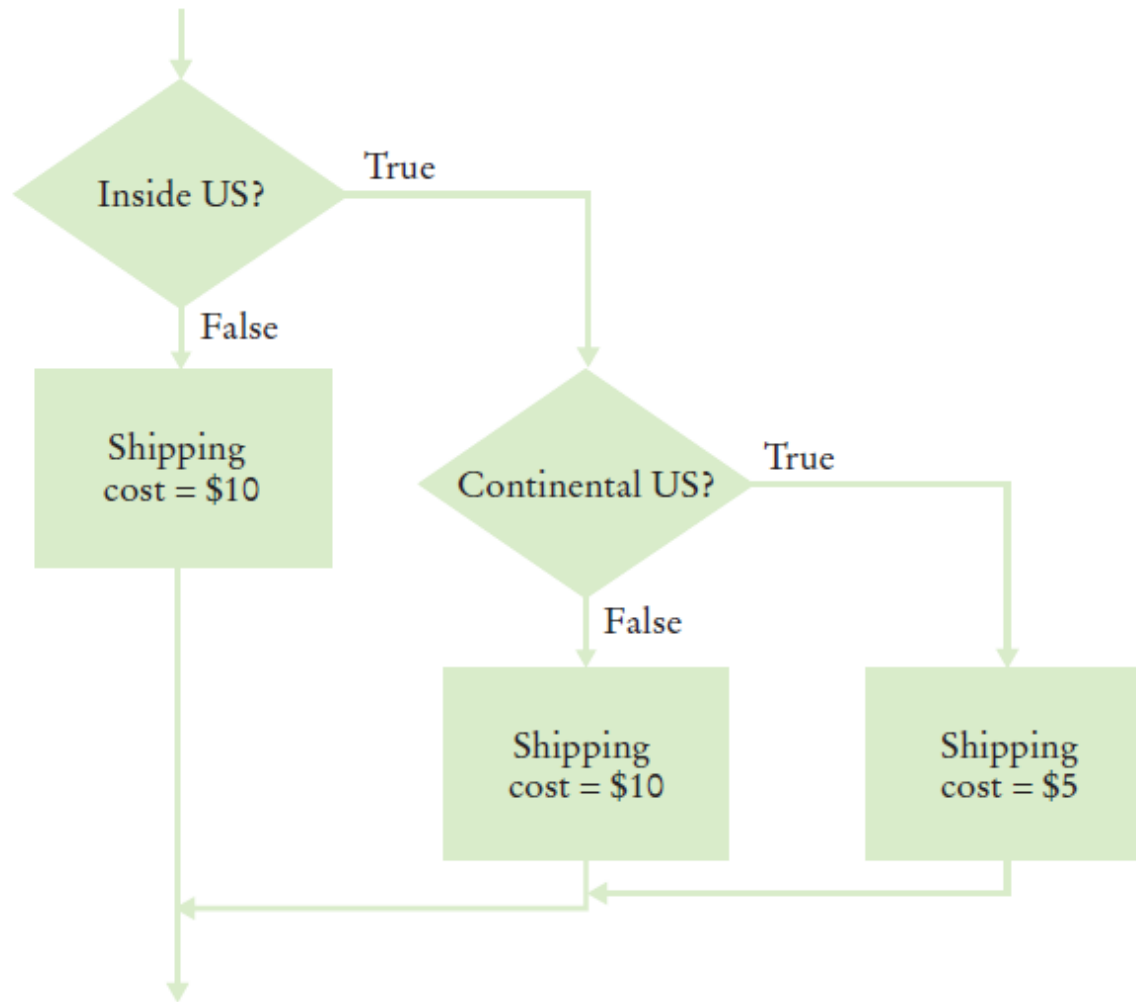
- Don't do this!



# Shipping Cost Flowchart

---

Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.



# Shipping Example

---

- Open the file:
  - Shipping.py
- Run the program with several different inputs?
  - What happens if you enter “usa” as the country?
- We will learn several ways to correct the code later in this chapter

# Complex Decision Making is Hard

---

- Computer systems are used to help sort and route luggage at airports
- The systems:
  - Scan the baggage tags
  - Sorts the items
  - Routes the items to conveyor belts
  - Humans then place the bags on trucks
- In 1993 Denver built a new airport with a “state of the art” luggage system that replaced the human operators with robotic carts
  - The system failed
  - The airport could not open without a luggage system
  - The system was replaced (it took over a year)
  - The cost was almost \$1B.... (yes one billion... 1994 dollars)
  - The company that designed the system went bankrupt

# Building Test Cases

---

# Problem Solving: Test Cases

---

- Aim for complete coverage of all decision points:
  - There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases
  - Test a handful of boundary conditions, such as an income that is at the boundary between two tax brackets, and a zero income
  - If you are responsible for error checking (which is discussed in Section 3.9), also test an invalid input, such as a negative income
- Each branch of your code should be covered with a test case



# Choosing Test Cases

---

- Choose input values that:
  - Test boundary cases and 0 values
  - Test each branch

Test Case	Expected Output	Comment
30,000 s	3,000	10% bracket
72,000 s	13,200	3,200 + 25% of 40,000
50,000 m	5,000	10% bracket
104,000 m	16,400	6,400 + 25% of 40,000
32,000 s	3,200	boundary case
0 s	0	boundary case

# Make a Schedule...

---

- Make a reasonable estimate of the time it will take you to:
  - Design the algorithm
  - Develop test cases
  - Translate the algorithm to code and enter the code
  - Test and debug your program
- Leave some extra time for unanticipated problems

***As you gain more experience your estimates will become more accurate. It is better to have some extra time than to be late***

# Boolean Variables and Operators

---

# Boolean Variables

---

- Boolean Variables
  - A Boolean variable is often called a flag because it can be either up (`true`) or down (`false`)
  - `boolean` is a Python data type
    - `failed = True`
    - Boolean variables can be either `True` or `False`
- There are two Boolean Operators: `and`, `or`
  - They are used to combine multiple conditions

# Combined Conditions: **and**

---

- Combining two conditions is often used in range checking
  - Is a value between two other values?
- Both sides of the **and** must be true for the result to be true

```
if temp > 0 and temp < 100 :  
    print("Liquid")
```

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

# Combined Conditions: **or**

---

- We use **or** if only one of two conditions need to be true
  - Use a compound conditional with an **or**:

```
if temp <= 0 or temp >= 100
:
    print("Not liquid")
```

- If either condition is true
  - The result is true

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

# The *not* operator: **not**

---

- If you need to invert a boolean variable or comparison, precede it with **not**

```
if not attending or grade < 60 :  
    print("Drop?")
```

```
if attending and not(grade < 60) :  
    print("Stay")
```

A	<b>not</b> A
True	False
False	True

- If you are using **not**, try to use simpler logic:

```
if attending and grade >= 60 :  
    print("Stay")
```

# The *not* operator: inequality !

---

- A slightly different operator is used for the **not** when checking for inequality rather than negation.
- Example inequality:
  - The password that the user entered is not equal to the password on file.
  - `if userPassword != filePassword :`

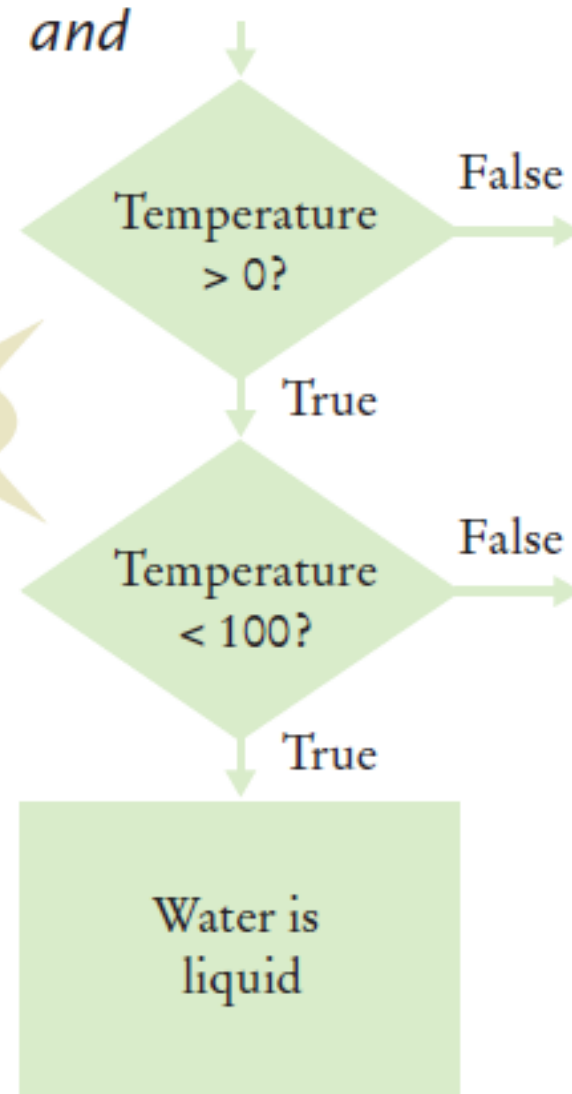


# *and* Flowchart

- This is often called 'range checking'
  - Used to validate that the input is between two values

Both conditions must be true

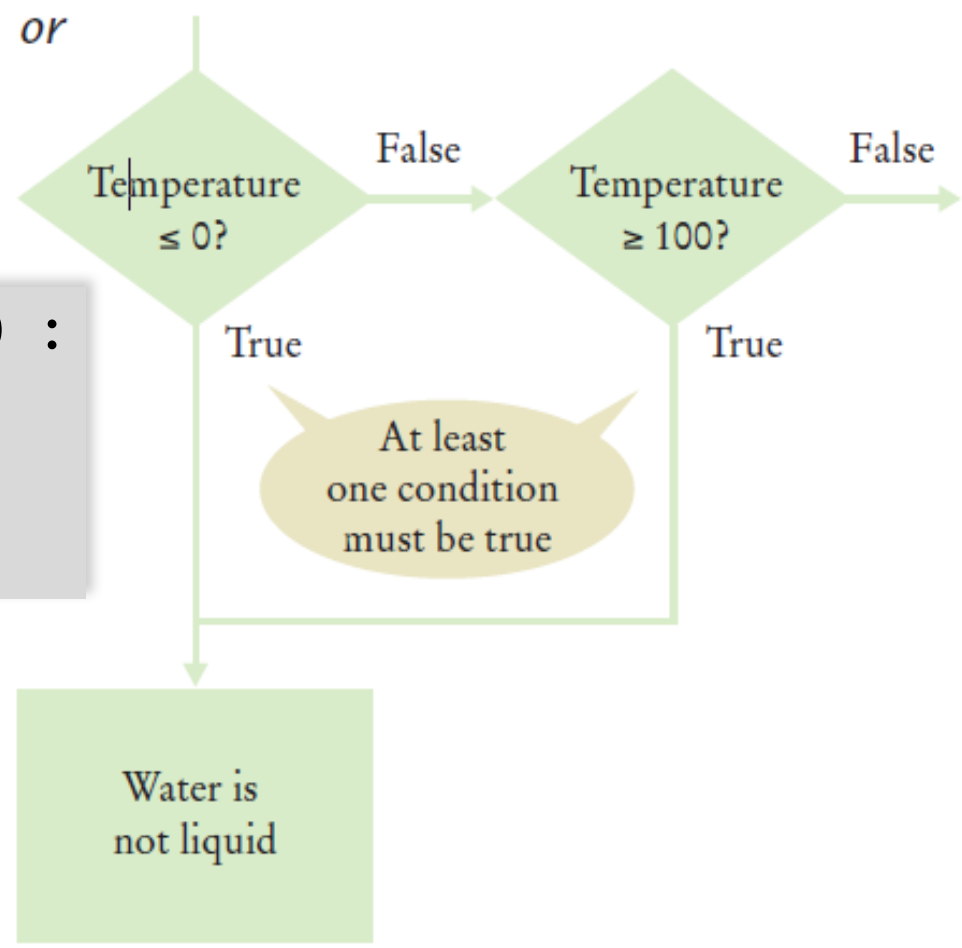
```
if temp > 0 and temp < 100 :  
    print("Liquid")
```



# or flowchart

- Another form of 'range checking'
  - Checks if value is outside a range

```
if temp <= 0 or temp >= 100 :  
    print("Not Liquid")
```



# Comparison Example

---

- Open the file:
  - Compare2.py
- Run the program with several inputs

# Boolean Operator Examples

---

Table 5 Boolean Operator Examples

Expression	Value	Comment
<code>0 &lt; 200 and 200 &lt; 100</code>	False	Only the first condition is true.
<code>0 &lt; 200 or 200 &lt; 100</code>	True	The first condition is true.
<code>0 &lt; 200 or 100 &lt; 200</code>	True	The or is not a test for “either-or”. If both conditions are true, the result is true.
<code>0 &lt; x and x &lt; 100 or x == -1</code>	<code>(0 &lt; x and x &lt; 100) or x == -1</code>	The and operator has a higher precedence than the or operator (see Appendix B).
<code>not (0 &lt; 200)</code>	False	<code>0 &lt; 200</code> is true, therefore its negation is false.
<code>frozen == True</code>	frozen	There is no need to compare a Boolean variable with True.
<code>frozen == False</code>	not frozen	It is clearer to use not than to compare with False.

# Common Errors with Boolean Conditions

---

## Confusing **and** and **or** Conditions

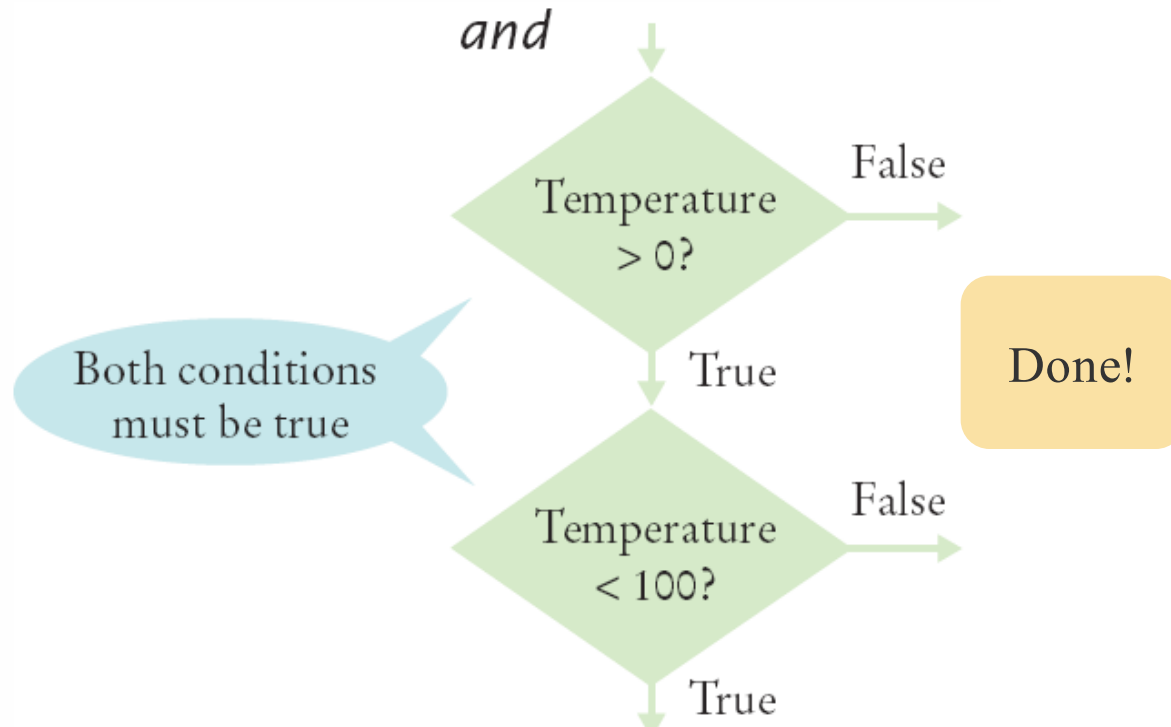
- It is a surprisingly common error to confuse **and** and **or** conditions.
- A value lies between 0 and 100 if it is at least 0 **and** at most 100.
- It lies outside that range if it is less than 0 **or** greater than 100.
- There is no golden rule; you just have to think carefully.

# Short-circuit Evaluation: **and**

---

- Combined conditions are evaluated from left to right
  - If the left half of an **and** condition is false, why look further?

```
if temp > 0 and temp < 100 :  
    print("Liquid")
```

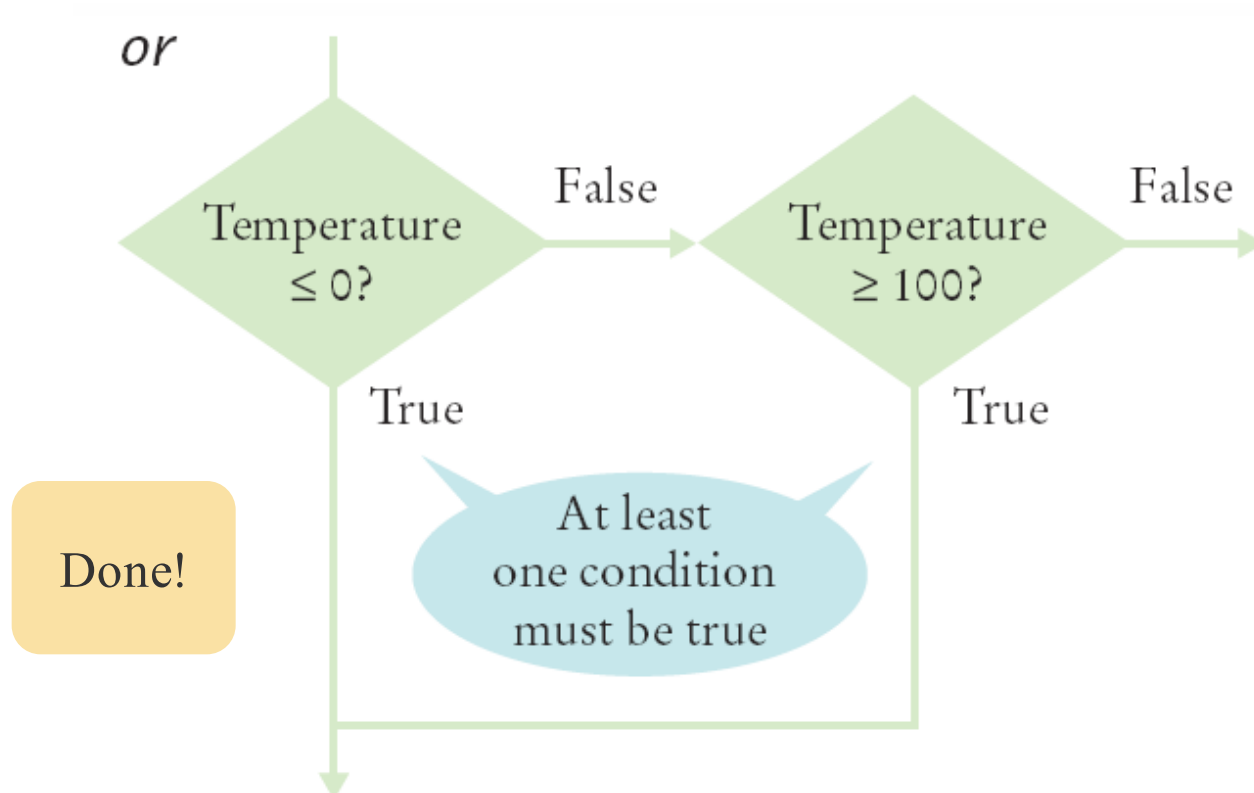


# Short-circuit evaluation: **or**

---

- If the left half of the **or** is true, why look further?

```
if temp <= 0 or temp >= 100 :  
    print("Not Liquid")
```



# De Morgan's law

---

- De Morgan's law tells you how to negate **and** and **or** conditions:
  - $\text{not}(A \text{ and } B)$  is the same as  $\text{not}A \text{ or } \text{not}B$
  - $\text{not}(A \text{ or } B)$  is the same as  $\text{not}A \text{ and } \text{not}B$
- Example: Shipping is higher to AK and HI

```
if (country != "USA"  
    and state != "AK"  
    and state != "HI") :  
    shippingCharge = 20.00
```

```
if not(country=="USA"  
        or state=="AK"  
        or state=="HI") :  
    shippingCharge = 20.00
```

- To simplify conditions with negations of **and** or **or** expressions, it's a good idea to apply De Morgan's law to move the negations to the innermost level.



# Analyzing Strings

---

# Analyzing Strings – The **in** Operator

---

- Sometimes it's necessary to analyze or ask certain questions about a particular string.
  - Sometimes it is necessary to determine if a string contains a given substring. That is, one string contains an exact match of another string.
  - Given this code segment,  
name = "John Wayne"
  - the expression  
"Way" **in** name
  - yields True because the substring "Way" occurs within the string stored in variable name.
  - The **not in** operator is the inverse on the in operator

# Substring: Suffixes

---

- Suppose you are given the name of a file and need to ensure that it has the correct extension

```
if filename.endswith(".html") :  
    print("This is an HTML file.")
```

- The `endswith()` string method is applied to the string stored in `filename` and returns `True` if the string ends with the substring `".html"` and `False` otherwise.

# Operations for Testing Substrings

---

**Table 6** Operations for Testing Substrings

Operation	Description
<code>substring</code> in <code>s</code>	Returns True if the string <code>s</code> contains <code>substring</code> and False otherwise.
<code>s.count(substring)</code>	Returns the number of non-overlapping occurrences of <code>substring</code> in the string <code>s</code> .
<code>s.endswith(substring)</code>	Returns True if the string <code>s</code> ends with the <code>substring</code> and False otherwise.
<code>s.find(substring)</code>	Returns the lowest index in the string <code>s</code> where <code>substring</code> begins, or <code>-1</code> if <code>substring</code> is not found.
<code>s.startswith(substring)</code>	Returns True if the string <code>s</code> begins with <code>substring</code> and False otherwise.

## Methods: Testing String Characteristics (1)

---

**Table 7** Methods for Testing String Characteristics

Method	Description
<code>s.isalnum()</code>	Returns True if string <i>s</i> consists of only letters or digits and it contains at least one character. Otherwise it returns False.
<code>s.isalpha()</code>	Returns True if string <i>s</i> consists of only letters and contains at least one character. Otherwise it returns False.
<code>s.isdigit()</code>	Returns True if string <i>s</i> consists of only digits and contains at least one character. Otherwise, it returns False.

## Methods for Testing String Characteristics (2)

---

**Table 7** Methods for Testing String Characteristics

<code>s.islower()</code>	Returns True if string <i>s</i> contains at least one letter and all letters in the string are lowercase. Otherwise, it returns False.
<code>s.isspace()</code>	Returns True if string <i>s</i> consists of only white space characters (blank, newline, tab) and it contains at least one character. Otherwise, it returns False.
<code>s.isupper()</code>	Returns True if string <i>s</i> contains at least one letter and all letters in the string are uppercase. Otherwise, it returns False.

# Comparing and Analyzing Strings (1)

Table 8 Comparing and Analyzing Strings

Expression	Value	Comment
"John" == "John"	True	== is also used to test the equality of two strings.
"John" == "john"	False	For two strings to be equal, they must be identical. An uppercase "J" does not equal a lowercase "j".
"john" < "John"	False	Based on lexicographical ordering of strings an uppercase "J" comes before a lowercase "j" so the string "john" follows the string "John". See Special Topic 3.2 on page 101.
"john" in "John Johnson"	False	The substring "john" must match exactly.
name = "John Johnson" "ho" not in name	True	The string does not contain the substring "ho".
name.count("oh")	2	All non-overlapping substrings are included in the count.
name.find("oh")	1	Finds the position or string index where the first substring occurs.
name.find("ho")	-1	The string does not contain the substring ho.
name.startswith("john")	False	The string starts with "John" but an uppercase "J" does not match a lowercase "j".
name.isspace()	False	The string contains non-white space characters.
name.isalnum()	False	The string also contains blank spaces.
"1729".isdigit()	True	The string only contains characters that are digits.
"-1729".isdigit()	False	A negative sign is not a digit.

# Comparing and Analyzing Strings (2)

---

**Table 8** Comparing and Analyzing Strings

<code>name.startswith("john")</code>	False	The string starts with "John" but an uppercase "J" does not match a lowercase "j".
<code>name.isspace()</code>	False	The string contains non-white space characters.
<code>name.isalnum()</code>	False	The string also contains blank spaces.
<code>"1729".isdigit()</code>	True	The string only contains characters that are digits.
<code>"-1729".isdigit()</code>	False	A negative sign is not a digit.



# Substring Example

---

- Open the file:
  - Substrings.ph
- Run the program and test several strings and substrings

# Input Validation

---

# Input Validation

---

- Accepting user input is dangerous
  - Consider the Elevator program:
  - Assume that the elevator panel has buttons labeled 1 through 20 (but not 13).

# Input Validation

---

- The following are illegal inputs:
  - The number 13

```
if floor == 13 :  
    print("Error: There is no thirteenth floor.")
```

- Zero or a negative number
- A number larger than 20

```
if floor <= 0 or floor > 20 :  
    print("Error: The floor must be between 1 and 20.")
```

- An input that is not a sequence of digits, such as five:
  - Python's exception mechanism is needed to help verify integer and floating point values (Chapter 7).

# Elevatorsim2.py

---

```
1  ##
2  # This program simulates an elevator panel that skips the 13th floor,
3  # checking for input errors.
4  #
5
6  # Obtain the floor number from the user as an integer.
7  floor = int(input("Floor: "))
8
9  # Make sure the user input is valid.
10 if floor == 13 :
11     print("Error: There is no thirteenth floor.")
12 elif floor <= 0 or floor > 20 :
13     print("Error: The floor must be between 1 and 20.")
14 else :
15     # Now we know that the input is valid.
16     actualFloor = floor
```

# Elevator Simulation

---

- Open the file:
  - `elevatorsim2.py`
- Test the program with a range of inputs including:
  - 12
  - 14
  - 13
  - -1
  - 0
  - 23
  - 19

# Chapter Three Review

---

# Summary: **if** Statement

---

- The **if** statement allows a program to carry out different actions depending on the nature of the data to be processed.
- Relational operators (**<** **<=** **>** **>=** **==** **!=** ) are used to compare numbers and Strings.
- Strings are compared in lexicographic order.
- Multiple **if** statements can be combined to evaluate complex decisions.
- When using multiple **if** statements, test general conditions after more specific conditions.



# Summary: Flowcharts and Testing

---

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.
- Flow charts are made up of elements for tasks, input/output, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.
- Each branch of your program should be covered by a test case.
- It is a good idea to design test cases before implementing a program.

# Summary: Boolean

---

- The type **boolean** has two values, **true** and **false**.
  - Python has two Boolean operators that combine conditions: **and** and **or**.
  - To invert a condition, use the *not* operator.
  - When checking for equality use the **!** operator.
  - The **and** and **or** operators are computed lazily:
    - As soon as the truth value is determined, no further conditions are evaluated.
  - De Morgan's law tells you how to negate **and** and **or** conditions.