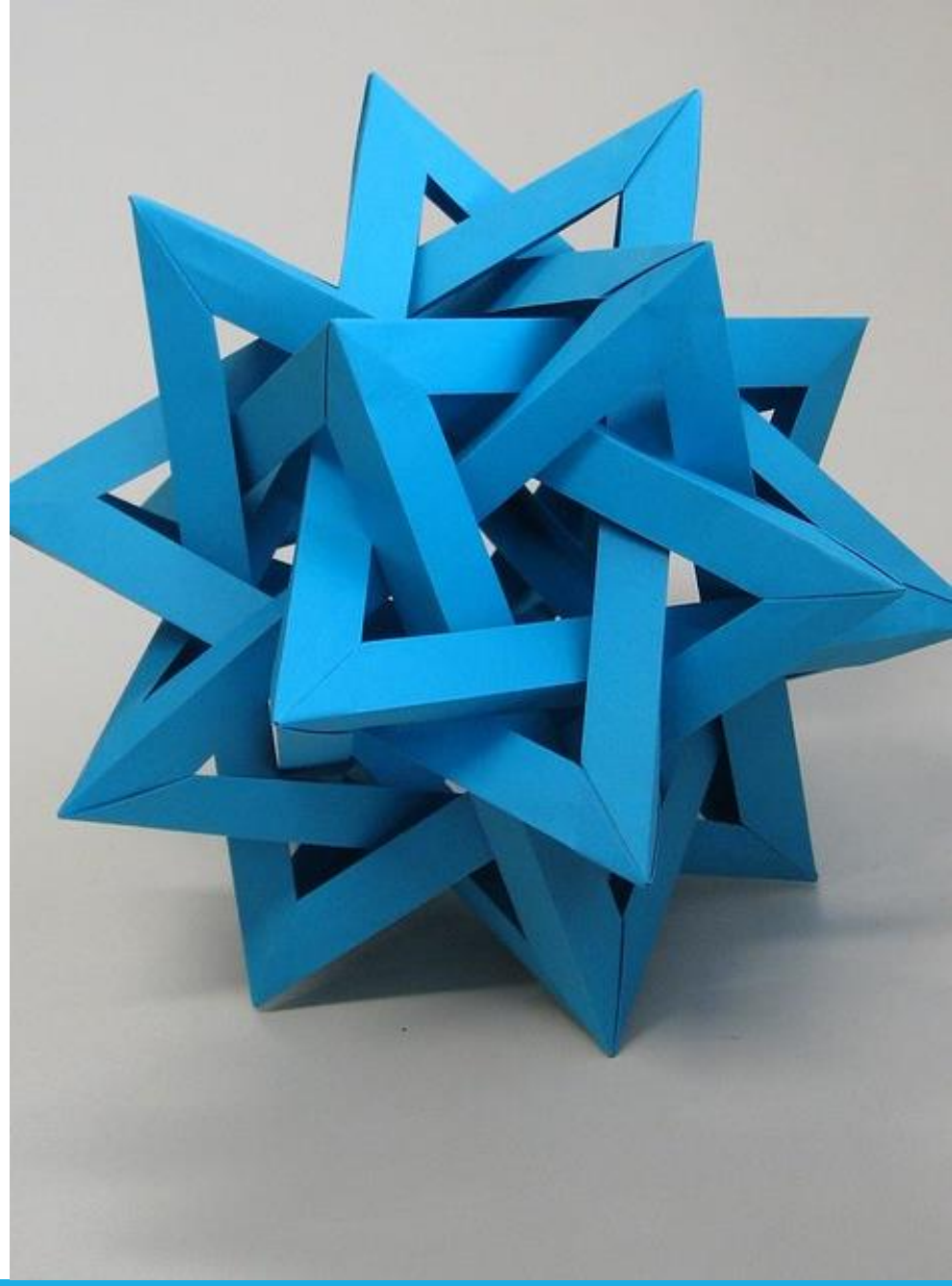
POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica

Unit P3: Decisions

DECISIONS, BOOLEAN CONDITIONS, STRING ANALYSIS, AND INPUT VALIDATION

Chapter 3



Unit Goals

- Input of Numeric and String data
- Formatting the output
- Implement decisions using the if statement
- Compare Numbers (integer and floating-point) and Strings
- Write statements using the Boolean data type
- Validate user input

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

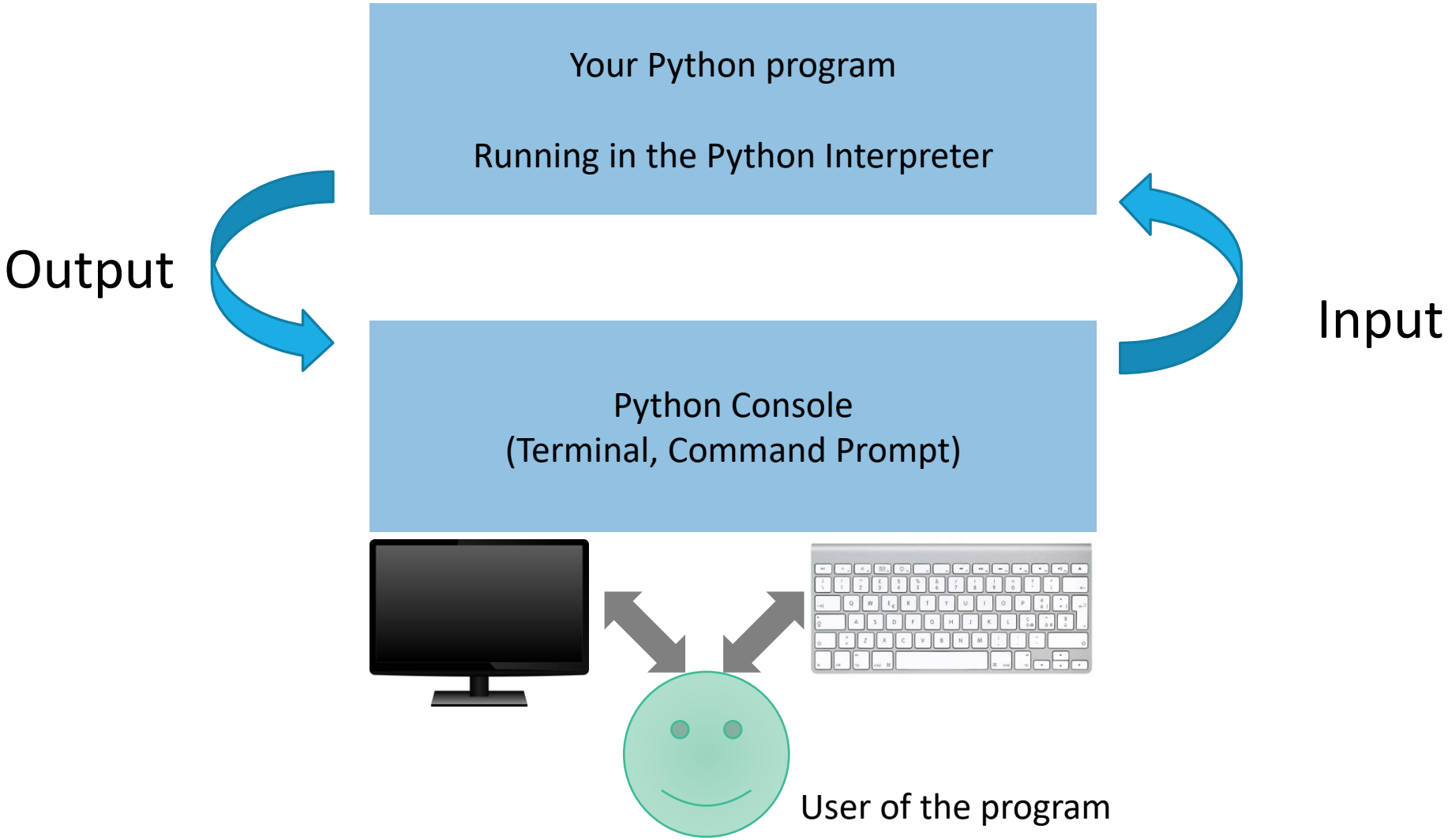
Contents

- Data Input and Formatted Output
- The `if` Statement
- Relational Operators
- Nested Branches
- Multiple Alternatives
- Boolean Variables and Operators
- Analyzing Strings
- Application: Input Validation

Input

2.5

Input and Output



Input and Output

- You can read a **String** from the console with the `input()` function:
  - `name = input("Please enter your name")`
- If **numeric** (rather than string) input is needed, you must convert the String value to a number

```
ageString = input("Please enter age: ") # String input
age = int(ageString) # Converted to int
```

- …or in a single step:

```
age = int(input("Please enter age: "))
price = float(input("Please enter the price: "))
```

Formatted output

2.5

Formatted output

- Inserting values inside strings, mainly for the purposes of an ordered and easy-to-read display
- Several methods are available in Python
  - String concatenation
  - Formatting operator `%`
  - `.format()` method
  - `f`-Strings

Example

```
pi = 3.14
r = 2
area = (r**2) * pi

print('The area of a circle of radius '+str(r)+' is '+str(area))
print('The area of a circle of radius %f is %f' % (r, area))
print('The area of a circle of radius {r} is {a}'.format(r=r, a=area))
print(f'The area of a circle of radius {r} is {area}')
```

Format operator %

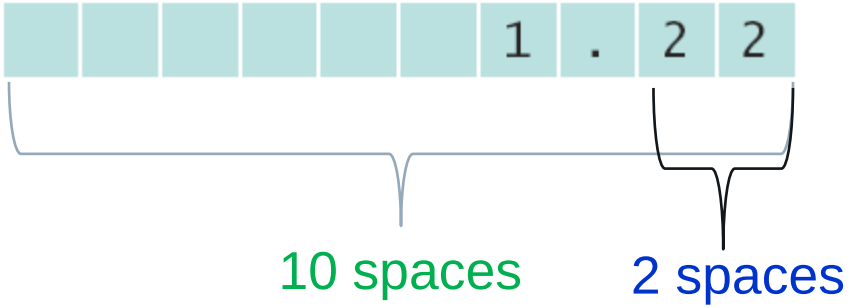
- Outputting floating point values can look strange:
  - Price per liter: 1.21997
- To control the output appearance of numeric variables, use the format operator **%**

  ```
  "string with format specifiers" % ( value, value, … )
  ```
- Ex: `"Price per liter: %.2f" % (price)`
  - Each format specifiers is replaced by a computed value
  - You may control the details of the formatting

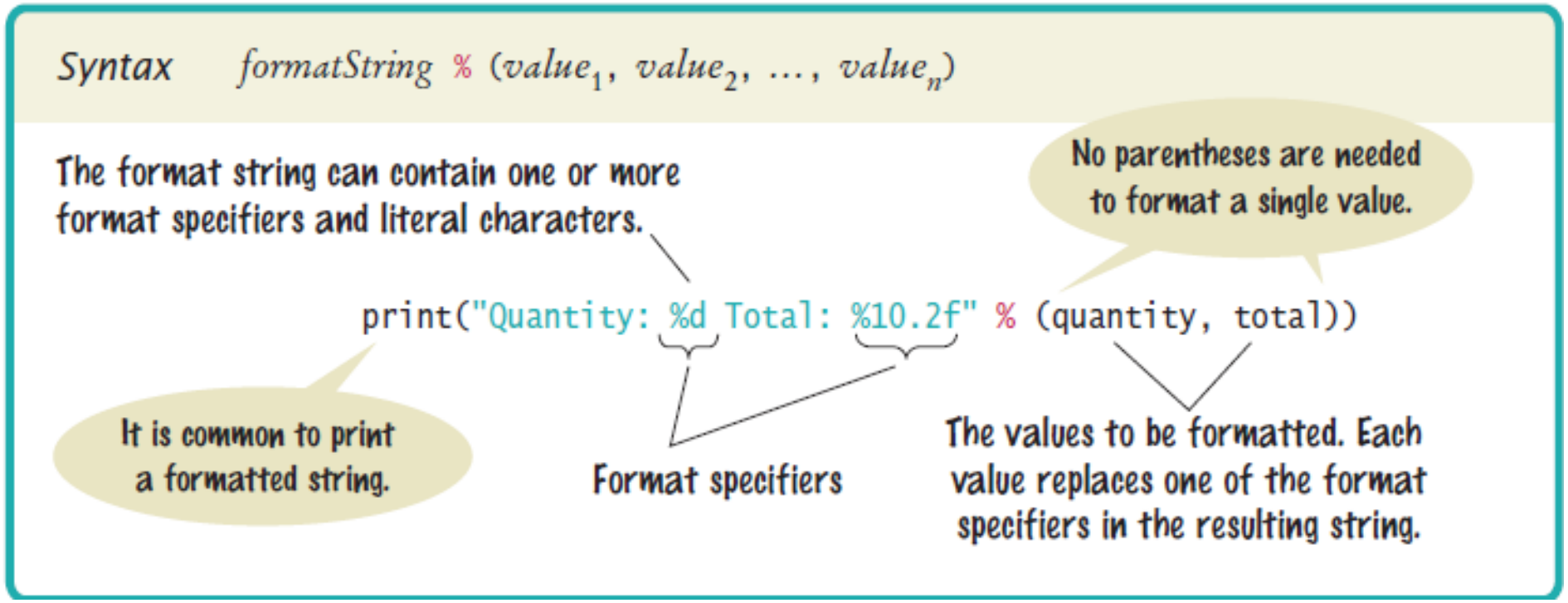
Formatted output

- Examples

```
print("Price per liter %.2f"  %(price))
Price per liter: 1.22
print("Price per liter %10.2f"  %(price))
Price per liter:       1.22
```

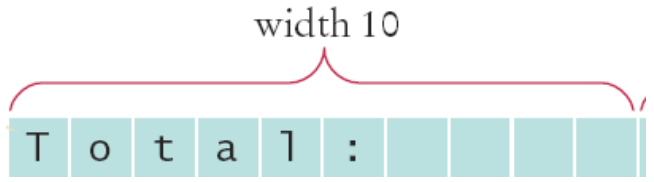


Syntax: format operator

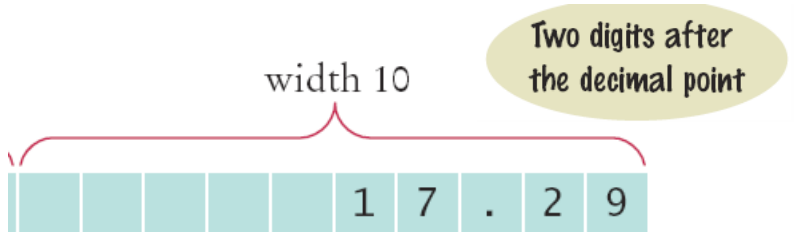


Format flag examples

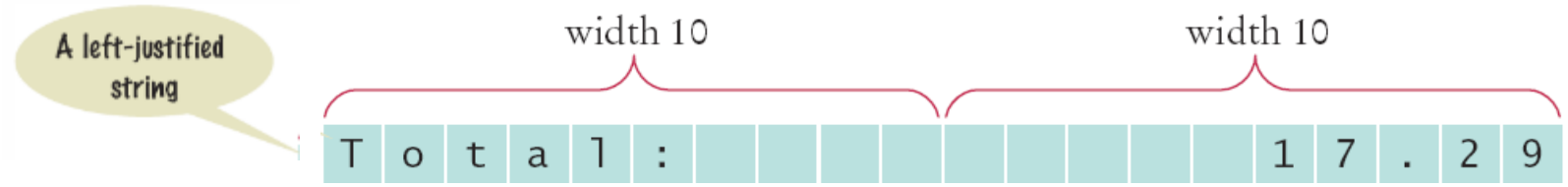
- Left Justify a String:
  ```
  print("%-10s" %("Total:"))
  ```



- Right justify a number with two decimal places
  ```
  print("%10.2f" %(price))
  ```



- And you can print multiple values:
  ```
  print("%-10s%10.2f" %("Total: ", price))
  ```



Volume2.py

ch02/volume2.py

```
1  ##
2  #   This program prints the price per ounce for a six-pack of cans.
3  #
4
5  # Define constant for pack size.
6  CANS_PER_PACK = 6
7
8  # Obtain price per pack and can volume.
9  userInput = input("Please enter the price for a six-pack: ")
10 packPrice = float(userInput)
11
12 userInput = input("Please enter the volume for each can (in ounces): ")
13 canVolume = float(userInput)
14
15 # Compute pack volume.
16 packVolume = canVolume * CANS_PER_PACK
17
18 # Compute and print price per ounce.
19 pricePerOunce = packPrice / packVolume
20 print("Price per ounce: %8.2f" % pricePerOunce)
```

Format Specifier Examples

Table 9 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	24	Use d with an integer.
"%5d"	␣␣␣24	Spaces are added so that the field width is 5.
"%05d"	00024	If you add 0 before the field width, zeroes are added instead of spaces.
"Quantity:%5d"	Quantity:␣␣␣24	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1.21997	Use f with a floating-point number.
"%.2f"	1.22	Prints two digits after the decimal point.
"%7.2f"	␣␣␣1.22	Spaces are added so that the field width is 7.
"%s"	Hello	Use s with a string.
"%d %.2f"	24 1.22	You can format multiple values at once.
"%9s"	␣␣␣␣Hello	Strings are right-justified by default.
"%-9s"	Hello␣␣␣␣	Use a negative field width to left-justify.
"%d%%"	24%	To add a percent sign to the output, use %%.

f-Strings (Formatted String Literals)

- A formatted string literal or f-string is a string literal that is prefixed with 'f' or 'F'.
- These strings may contain replacement fields, which are expressions delimited by curly braces {}.
- While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

F-Strings are not in the book. See:
https://docs.python.org/3/reference/lexical_analysis.html#f-strings

f-String Examples

```
f"the result is {result}"
f"the result is {a+b}"
f'my name is {username}'
```

Formatting in f-Strings

- Format specifiers may be added inside the `{}`, separated with a `:` symbol
- `f"The distance is {dist:8.2} meters"`
- The syntax and meaning of the format specifiers is the same as the % operator

Formatting methods comparison

FORMATTING OPERATOR: %

- `"your age is %d" % ( age )`
- Format string
- % placeholders
  - Specify the data type and formatting options
- Actual values are inserted
  - Specify which variable will be used to replace the placeholders
  - Taken from the values in `%(val, val, … )` second argument

F-STRINGS

- `f"your age is {age}"`
- String prefixed by an “`f`” letter
- `{…}` placeholders
  - Specify which variable will be used to replace the placeholders
  - May also be an expression
- `{age}` is an actual variable in the surrounding python code

The `if` statement

3.1

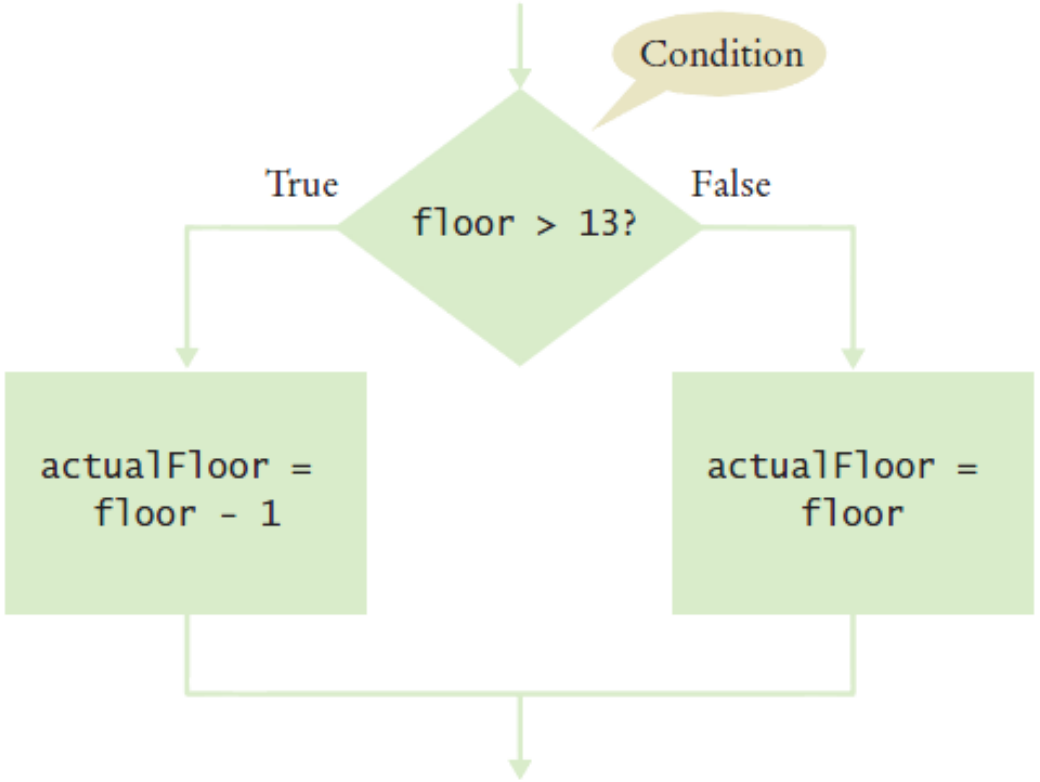
The `if` Statement

- A computer program often needs to make decisions based on input, or circumstances
- For example, buildings often 'skip' the 13th floor, and elevators should too
  - The 14th floor is really the 13th floor
  - So every floor above 12 is really 'floor – 1'
    - if floor > 12, actual floor = floor - 1
- The two keywords composing 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

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



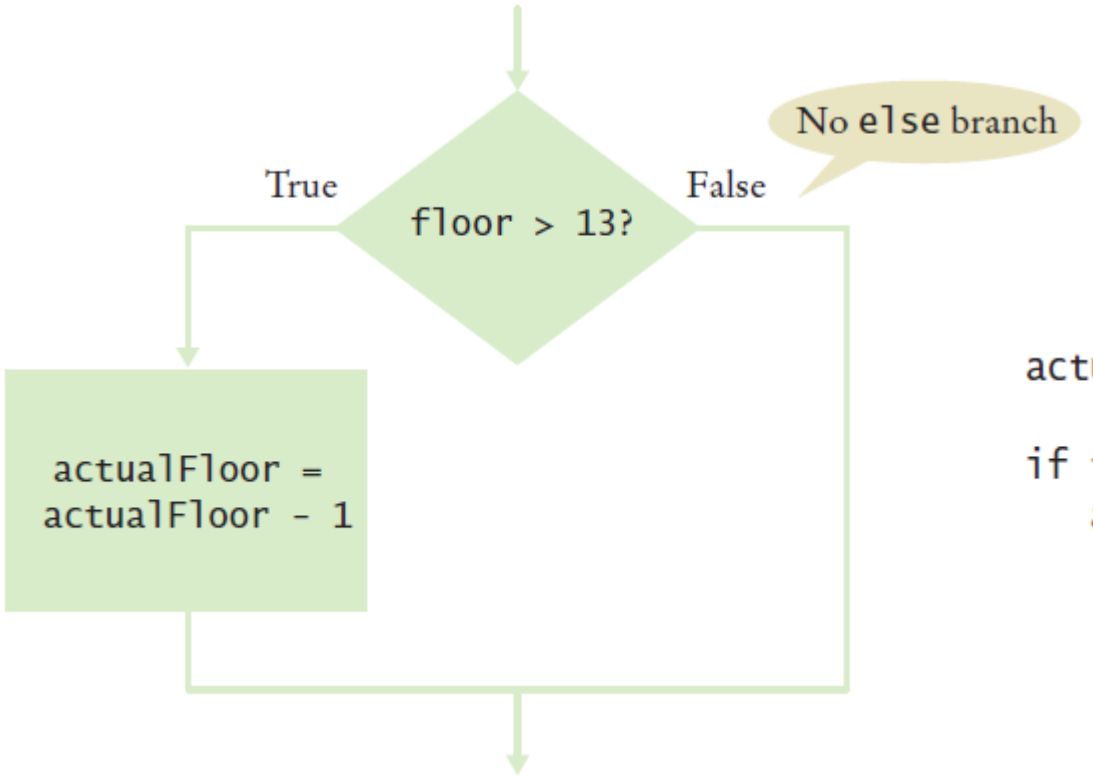

```
actualFloor = 0

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

Indentation:
The content of the `if` and `else` branches must be *indented by some spaces (usually 2 or 4)*

Flowchart with only a True Branch

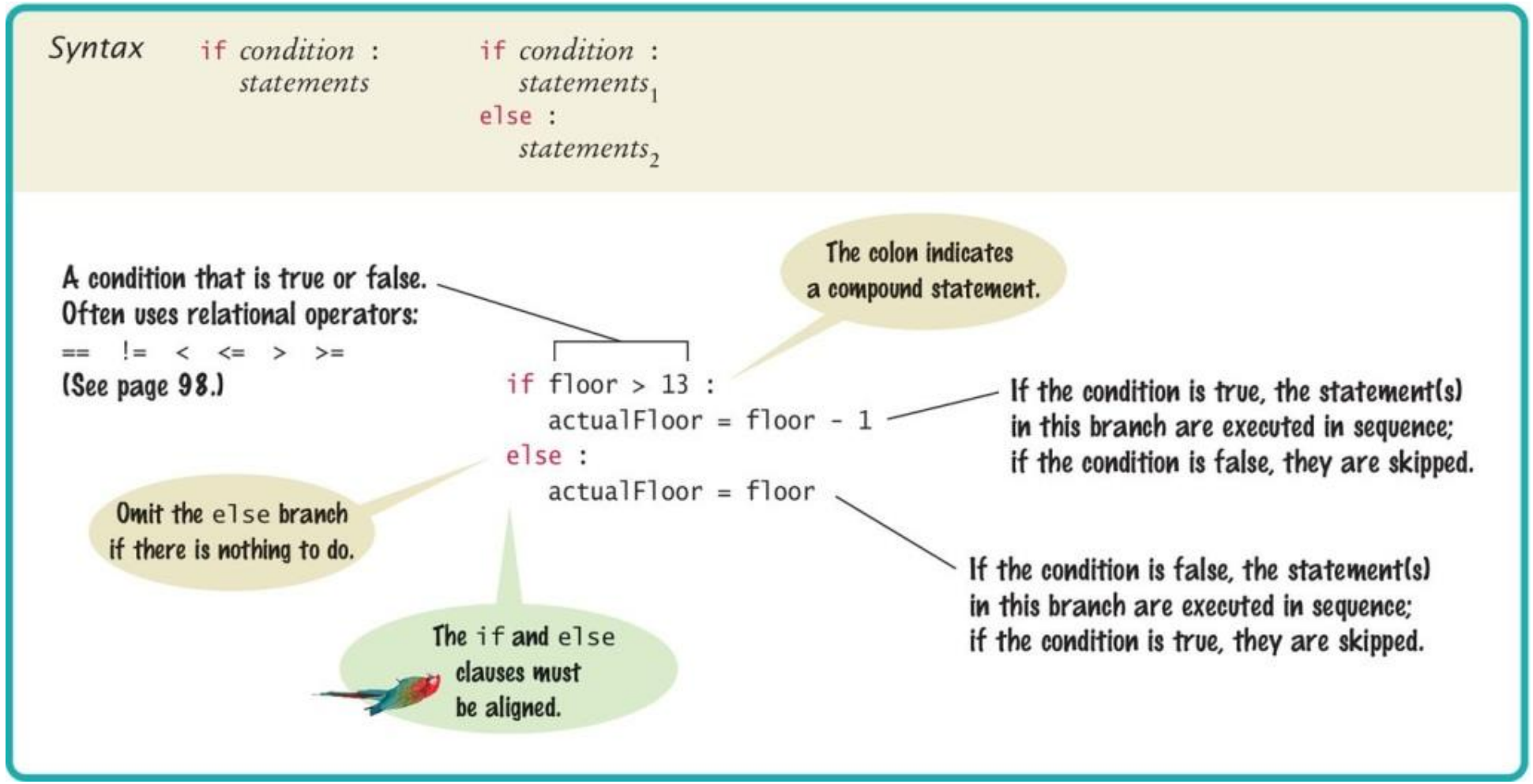
- An `if` statement may not need a 'False' (`else`) branch




```
actualFloor = floor

if floor > 13 :
   actualFloor = actualFloor – 1
```

Syntax 3.1: The `if` Statement



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
```

Example 1

- Open the file: elevatorsim.py
- Run the program
  - Try a value that is less that 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?

Example 1 - corrected

- 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 "=="

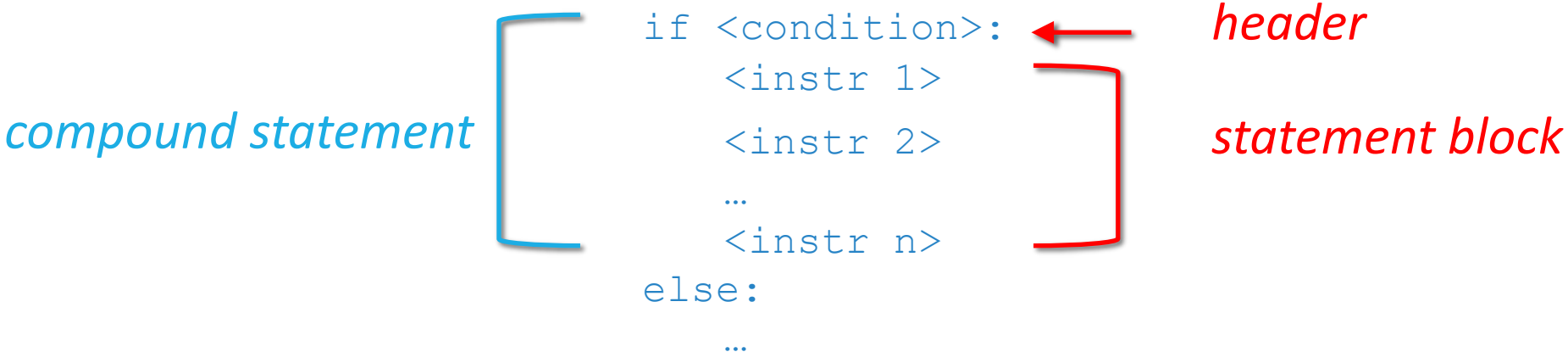
Important Warning:
Do not confuse = with ==
= declares a variable
= assigns a value
== makes an equality comparison

Example 1 – proposed addendum

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

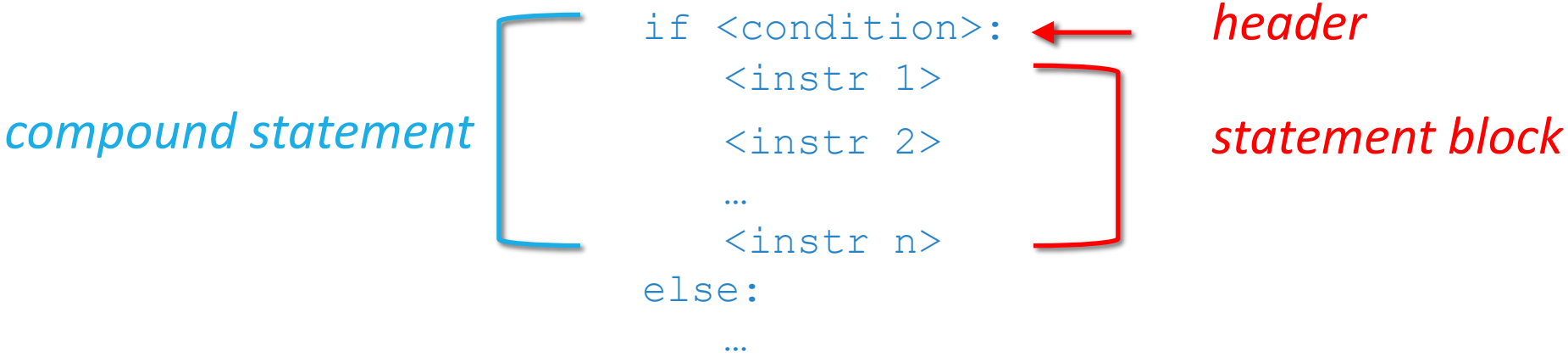
Compound Statements

- Some constructs in Python are **compound statements**.
  - The if statement is an example of a compound statement
- Compound statements span multiple lines and consist of a header and a statement block
- Compound statements require a colon "**:**" at the end of the header.



Compound Statements

- 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
  - ends at the first statement that is less indented
- Most IDEs properly indent the statement block.



Compound Statements

- Statement blocks can be **nested** inside the blocks of other compound statements (of the same or other block type)
- 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

A Common Error

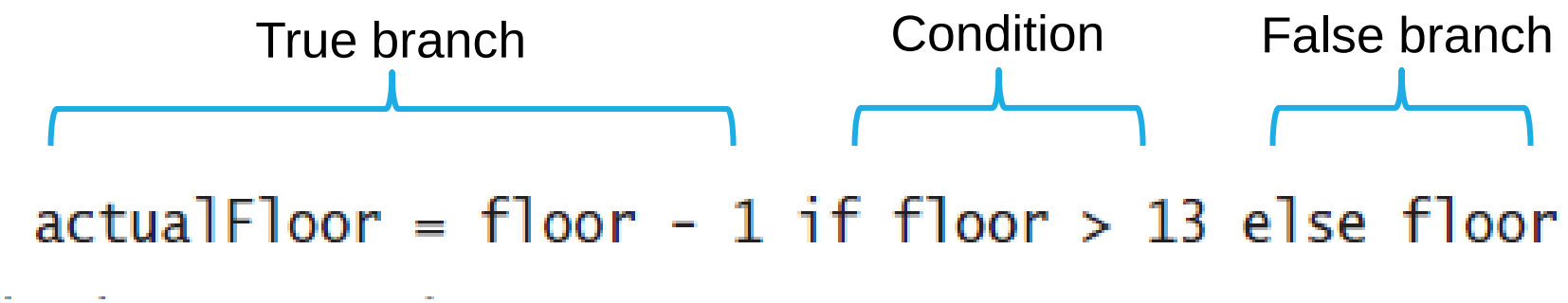
- Avoid duplication in branches
- If the same code is duplicated in each branch then move it out of the if statement.

```
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 course
  - The shortcut notation can be used anywhere a value is expected




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

Complexity is BAD….
This “shortcut” is difficult to read and a poor programming practice

Relational operators 📘 3.2

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
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Assignment vs. Equality Testing

- Assignment: makes something true.

```
floor = 13
```

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

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

Never confuse
=
with
==

Comparing Strings

- Checking if two strings are equal

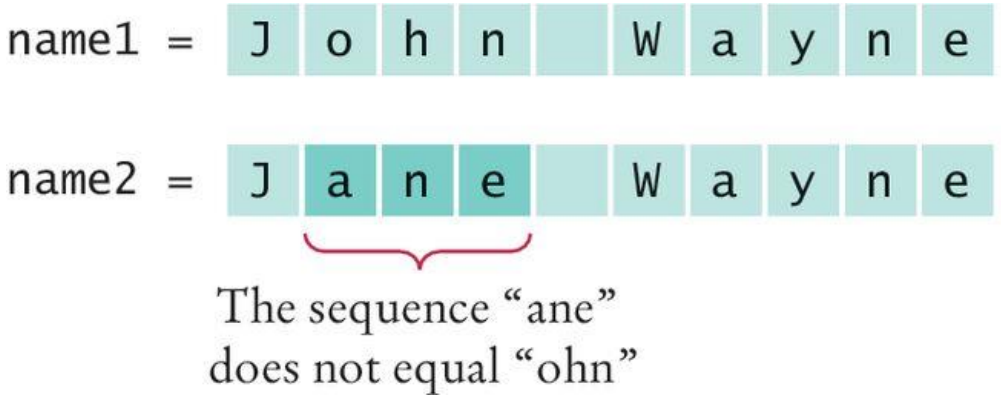
```
if name1 == name2 :
   print("The strings are identical")
```

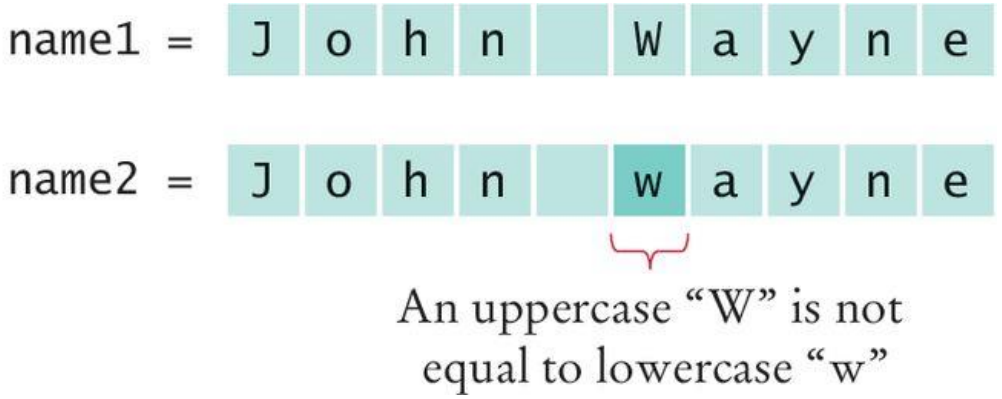
- Checking if two strings are not equal

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

Checking for String Equality

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





Relational Operator Examples (1)

Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`True`	3 is less than 4; <= tests for "less than or equal".
`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`False`	> is the opposite of <=.
`4 < 4`	`False`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`True`	Both sides are equal; <= 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`	**Error**	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`	**Error**	You cannot compare a string to a number.

Relational Operator Examples (2)

Table 2 Relational Operator Examples

⊘	`3 = 6 / 2`	**Error**	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`	**Error**	You cannot compare a string to a number.

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 2.0000000000000004
```

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:

$$|x - y| < \varepsilon$$

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

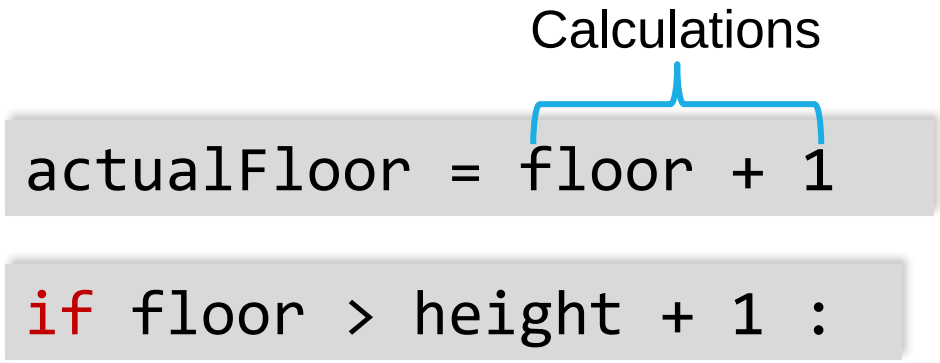
Lexicographical Order

- To compare Strings in ‘dictionary’ like order:
  - `string1 < string2`

- Notes
  - All UPPERCASE letters come before lowercase
    - ‘A’ comes before ‘a’, but also ‘Z’ comes before ‘a’
  - ‘space’ comes before all other printable characters
  - Digits (0-9) come before all letters
  - The order is ruled by the Basic Latin (ASCII) Subset of Unicode
    - Accented characters are not always logical

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



Example

The Sale 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.

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 * original price
- 3) Write pseudocode for the false branch
  - Discounted price = 0.84 * 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 = _____ * original price
- 6) Test both branches
  - Discounted price = 0.92 * 100 = 92
  - Discounted price = 0.84 * 200 = 168
- 7) Write the code in Python

The Sale Example (solution)

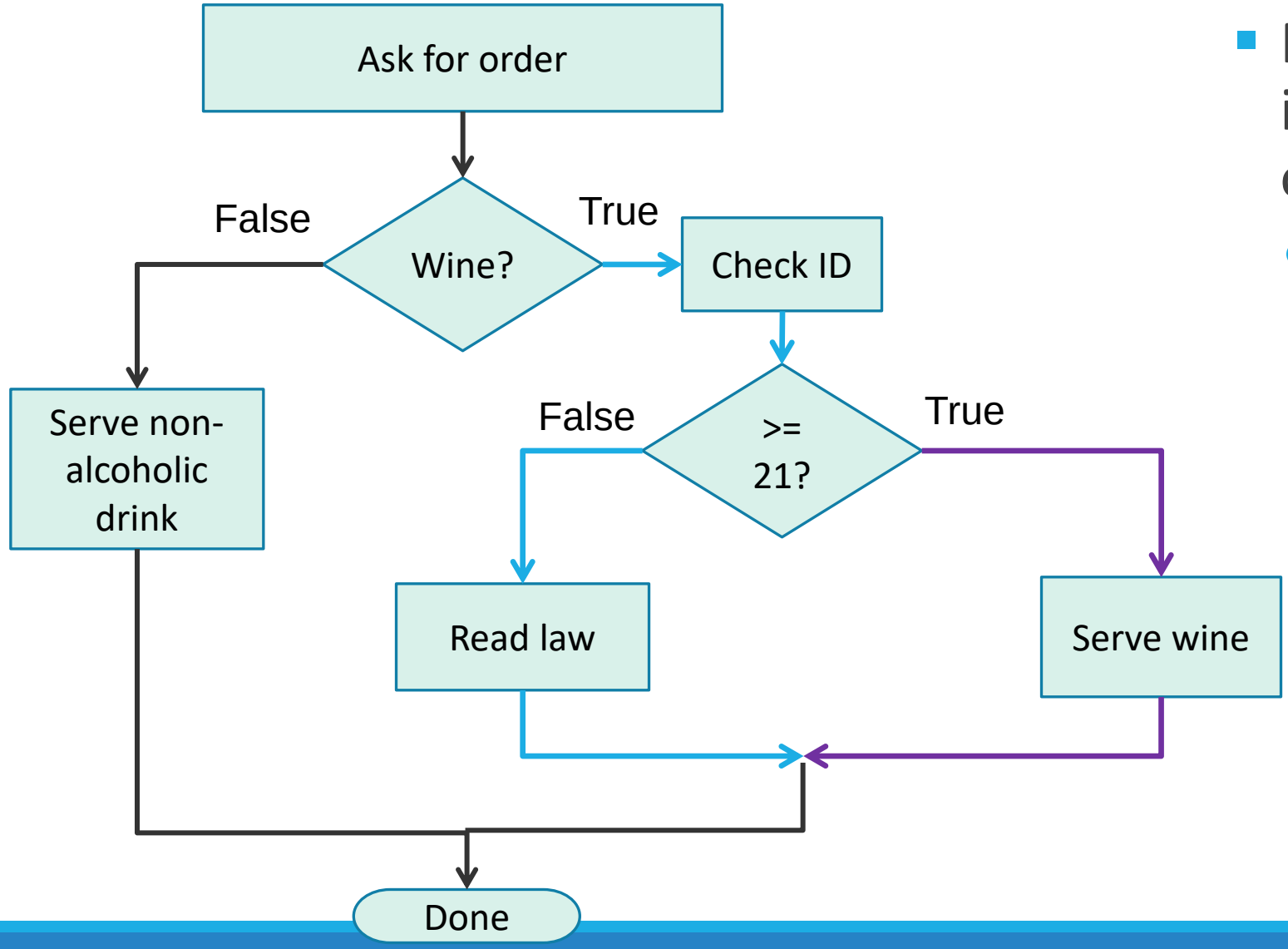
- Open the file:
  - sale.py
- Run the program several time using different values
  - Use values less than 128
  - Use values greater that 128
  - Enter 128
  - Enter invalid inputs
- What results do you get?

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

Nested Branches

3.3

Flowchart of a Nested if

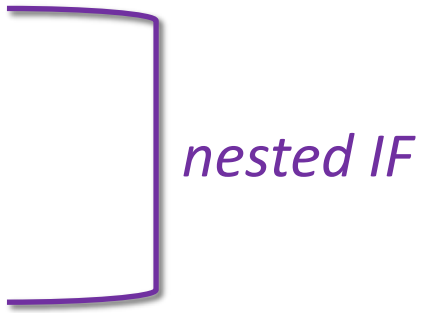


- Nested if-else inside true branch of an if statement.
  - Three paths

Nested Branches

- You can nest an **if** inside either branch of an **if** statement.
- Simple example: Ordering drinks (pseudo code)

```
Ask the customer for his/her 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 customer a non-alcoholic drink
```



Tips on Indenting Blocks

- Let pyCharm do the indenting for you… *(menu: Code – Auto-Indent lines)*

```
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 syntactically required in Python, but also makes code much easier to follow.

Tax Example: nested ifs

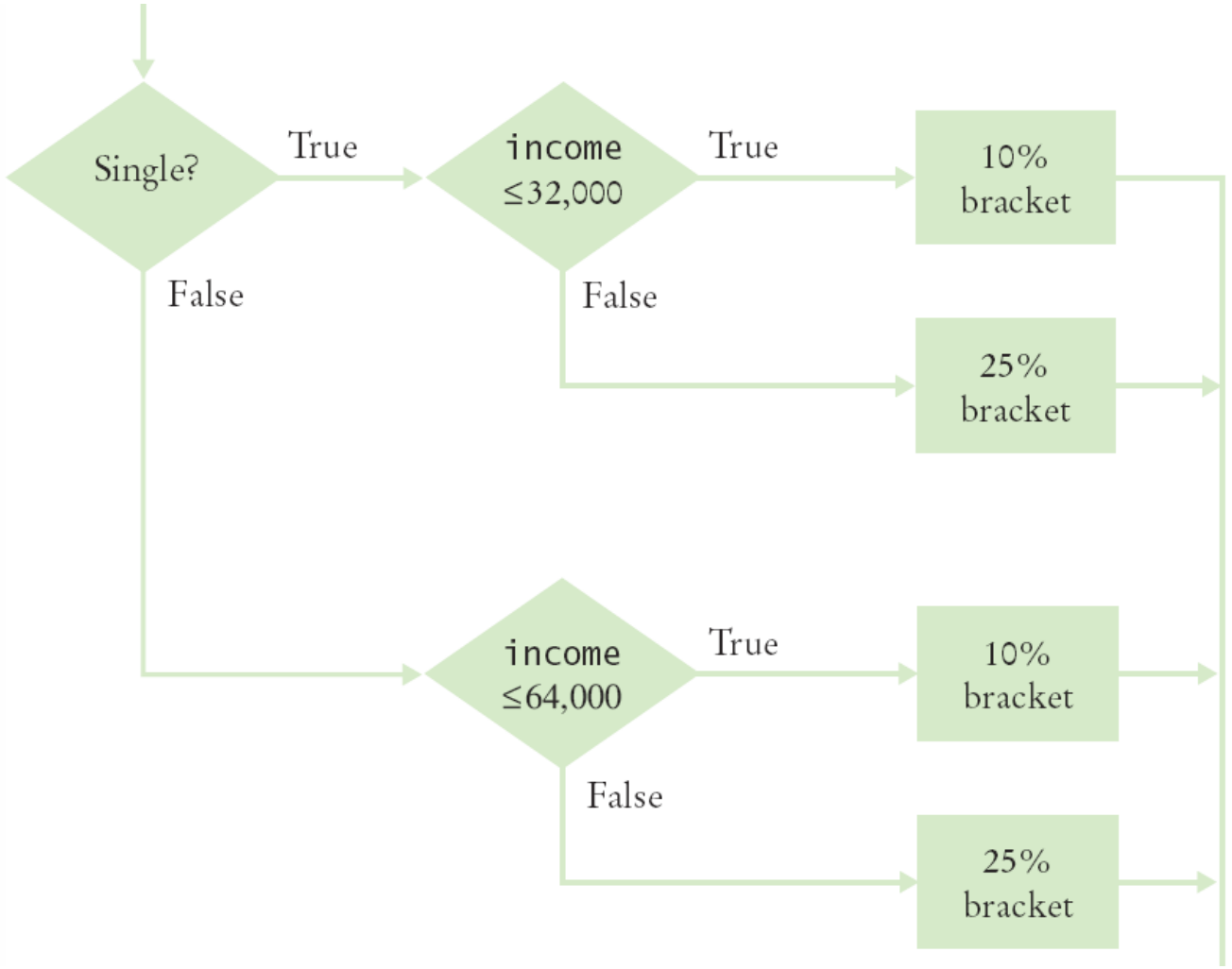
- Four outcomes (branches)

- Single
  - <= 32000
  - > 32000
- Married
  - <= 64000
  - > 64000

Table 3 Federal Tax Rate Schedule		
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)
```

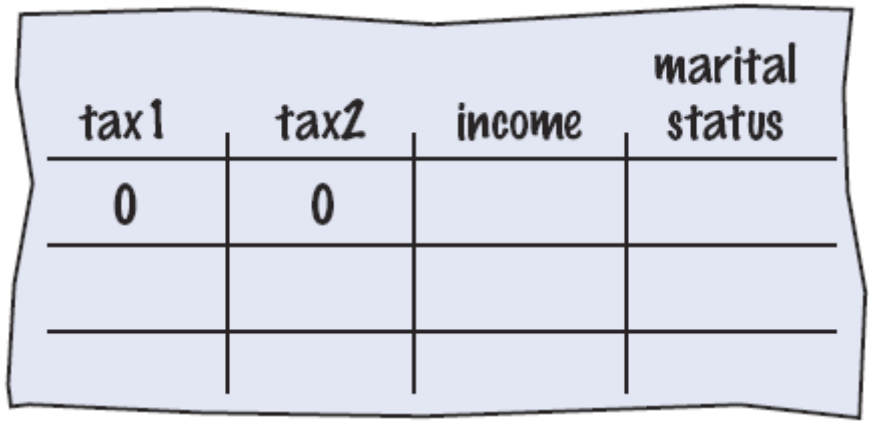
Running the Tax Example

- 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		

```
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
```

- Setup
  - Table of variables
  - Initial values

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 <= 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
~~0~~	~~0~~	80000	m
6400	4000		

Multiple Alternatives

3.4

Multiple Alternatives

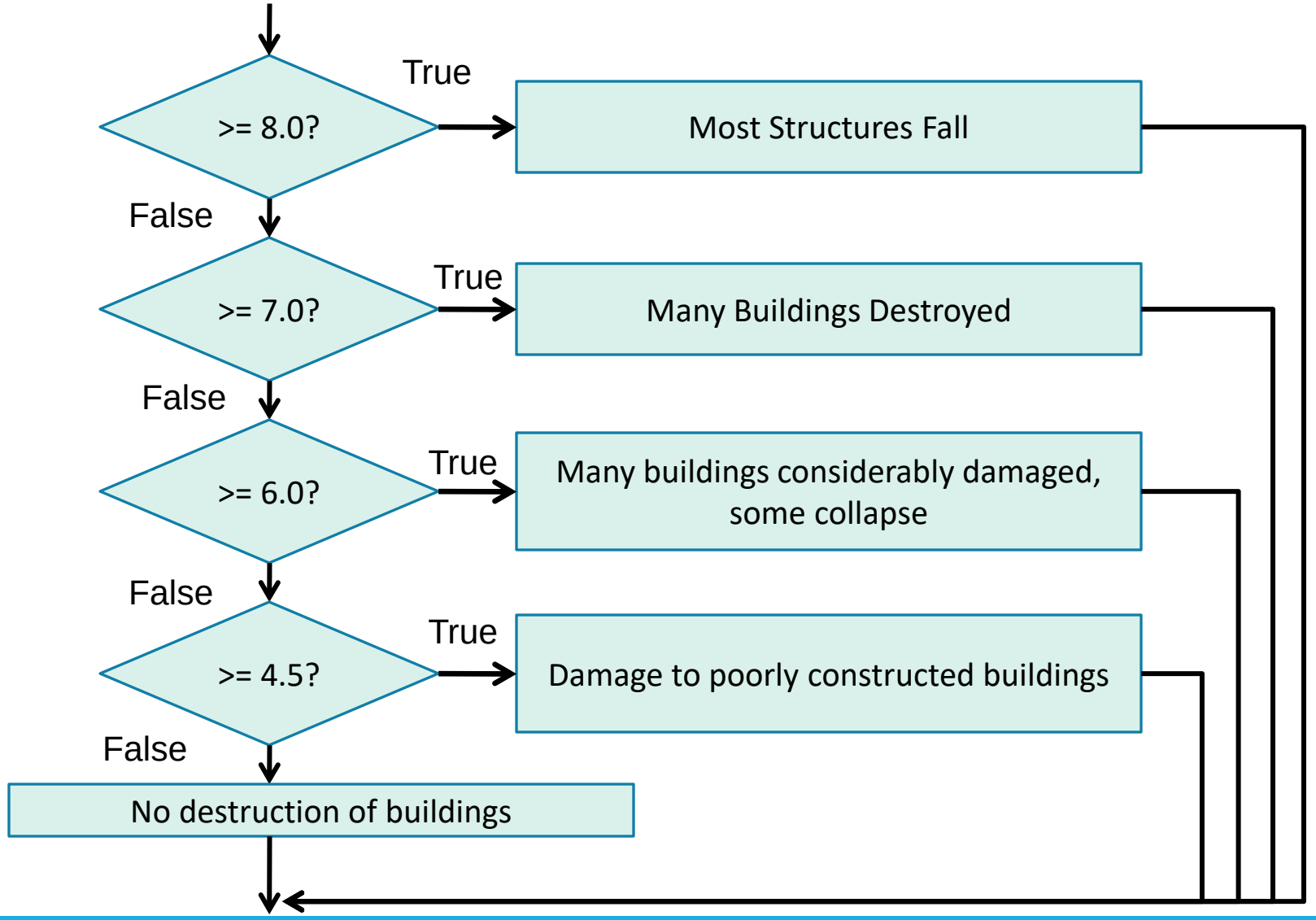
- What if you have more than two branches?
- Count the branches for the following earthquake effect example:
  - 8 (or greater)
  - 7 to 7.99
  - 6 to 6.99
  - 4.5 to 5.99
  - Less than 4.5

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

Table 4 Richter Scale

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 of 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

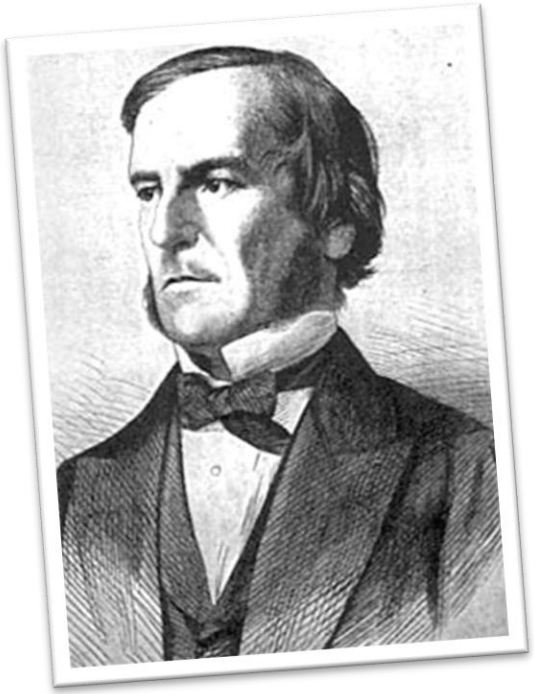
- Open the file:
  - earthquake.py
- Run the program with several different inputs

Boolean Variables and Operators

3.7

The Boolean logic of electronic computers

- In 1847 George Boole introduced a new type of formal logic, based exclusively on statements for which it was possible do verify their truth (true or false) in an algebraic way
- Computers adopt Boolean logic



Boolean Variables

- Boolean Variables
  - Boolean variables can be either `True` or `False`
    - `failed = True`
  - `bool` is a Python data type
  - A Boolean variable is often called a flag 🏁 because it can be either up (true) or down (false)
  - The condition of the `if` statement is, in fact, a Boolean value
- There are three Boolean Operators: `and`, `or`, `not`
  - They are used to combine multiple Boolean 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

Remembering a condition

- Boolean variables may be used to “remember” a condition, and test it later.

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

```
isLiquid = temp > 0 and temp < 100
# Boolean value True/False

if isLiquid :
   print("Liquid")
```

Chained Comparison Operators

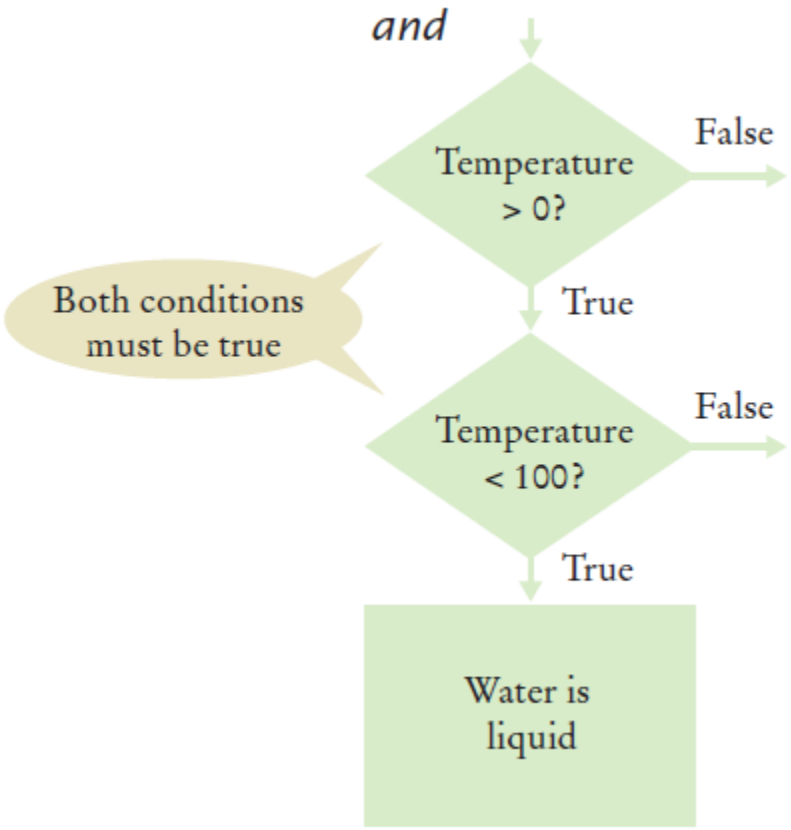
- Natural language: “If *temperature* is within the range from 0 to 100”
- Maths: $0 \leqslant temp \leqslant 100$
- Python: `0 <= temp and temp <= 100`

- You may also write: `0 <= temp <= 100`
  - Python allows *chained comparison operators*
  - *Most other programming languages do not allow this*
  - Tip: avoid this shortcut, use an explicit and

and Flowchart

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

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



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")
```

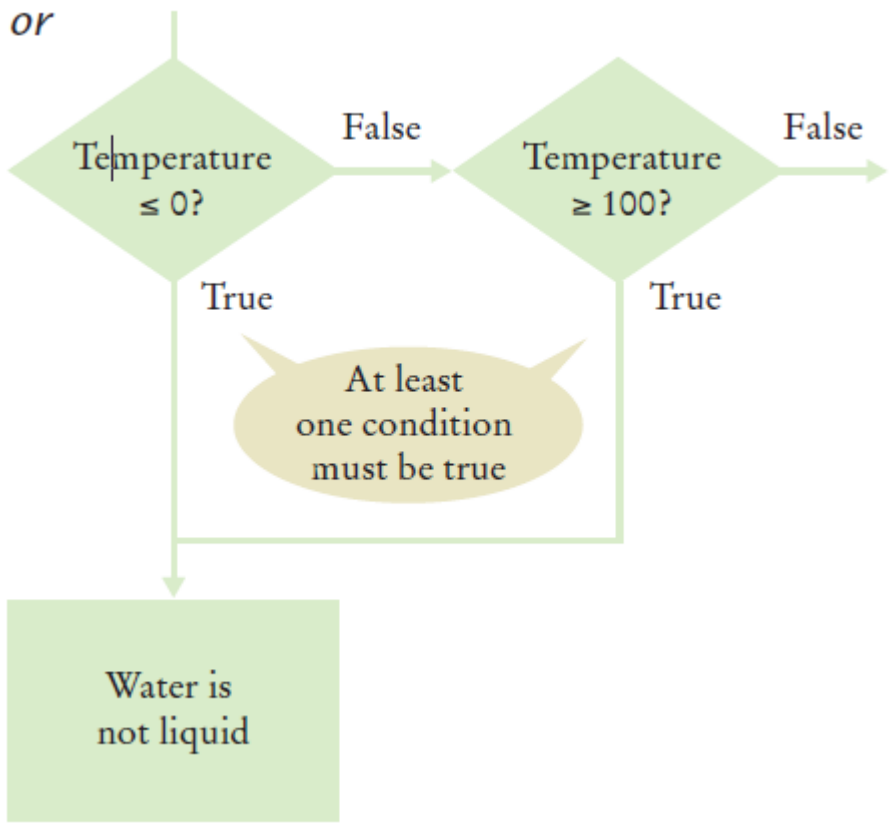
- If *either* condition is true
  - The result is true

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

or flowchart

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

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



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	not A
True	False
False	True

- For clarity, try to replace not with simpler logic

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

Note

```
if not ( a == b ):
# Is equivalent to
if a != b :
```

Comparison Example

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

Boolean Operator Examples

Table 5 Boolean Operator Examples

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

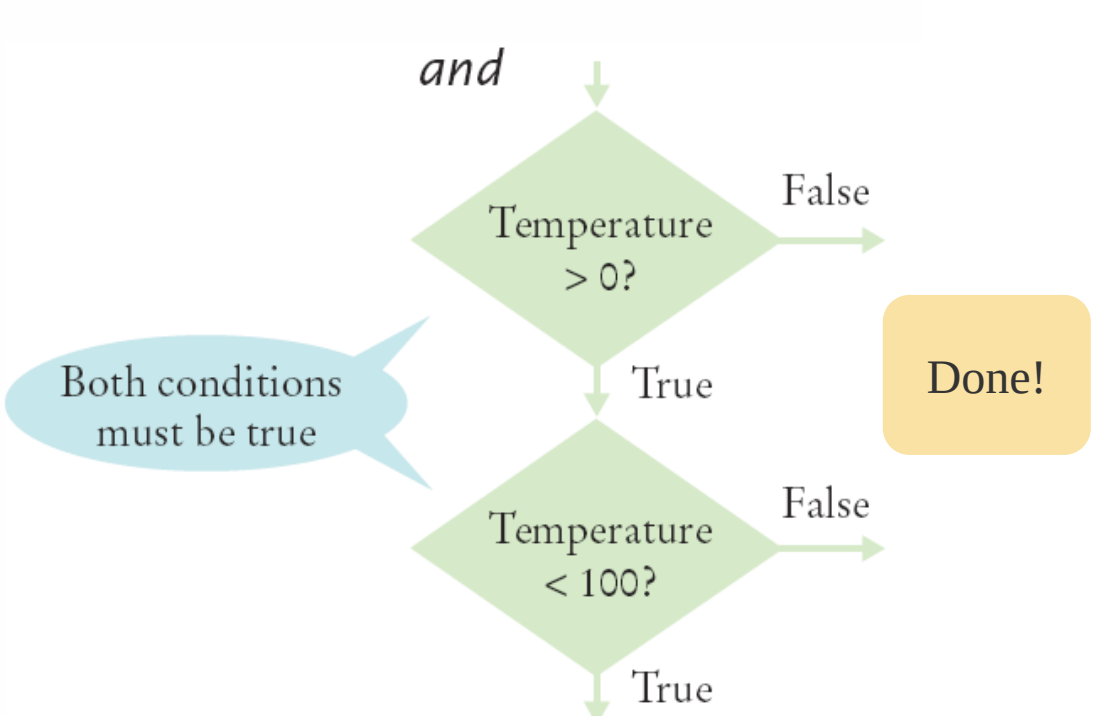
Common Errors with Boolean Conditions

- Confusing and with 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?

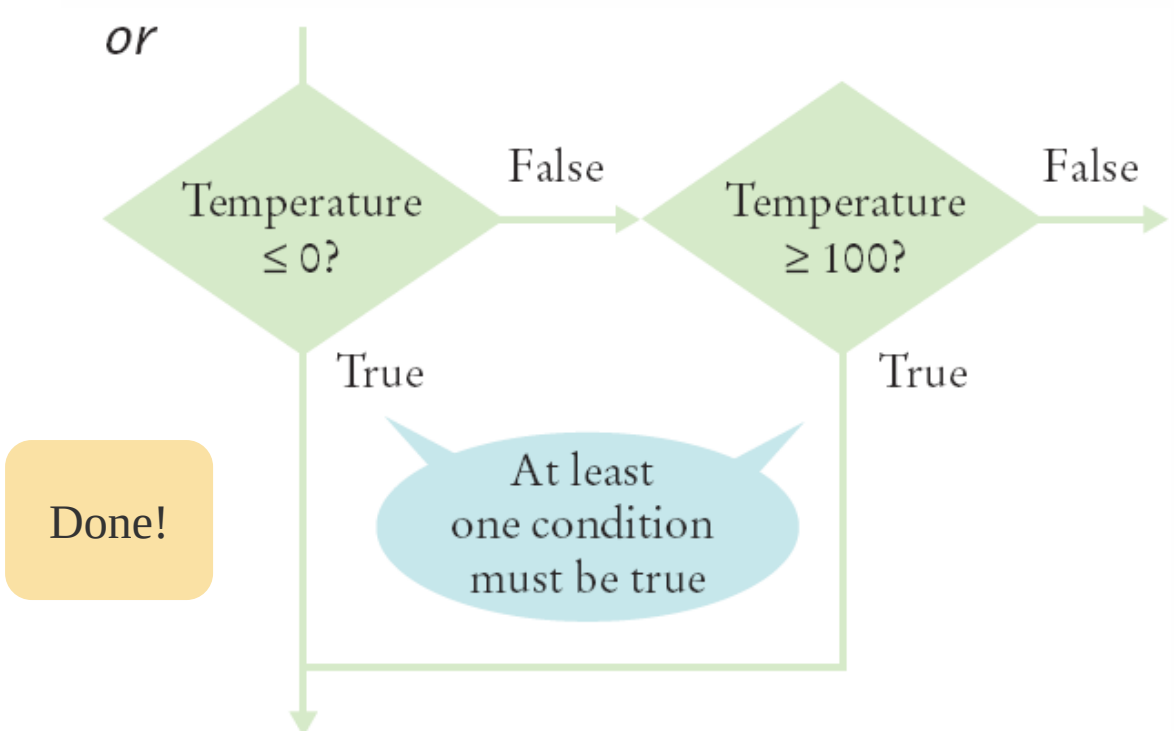
```
if temp > 0 and temp < 100 :
   print("Liquid")
```



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")
```



Some Boolean Properties

- Commutative:
  - A and B = B and A
  - A or B = B or A
- Associative:
  - A and B and C = (A and B) and C = A and (B and C)
  - A or B or C = (A or B) or C = A or (B or C)
- Distributive:
  - A and (B or C) = A and B or A and C
  - A or (B and C) = (A or B) and (A or C)

De Morgan's law

- De Morgan's law tells you how to negate and and or conditions:
  - `not(A and B)` is the same as `not(A) or not(B)`
  - `not(A or B)` is the same as `not(A) and 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.

String analysis

3.8

Analyzing Strings – The `in` Operator

- Sometimes it's necessary to analyze or ask certain questions about a particular string
- Example: it is necessary to determine if a string contains a given substring. That is, if 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 of 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
substring `in` *s*	Returns `True` if the string *s* contains *substring* and `False` otherwise.
s.`count`(*substring*)	Returns the number of non-overlapping occurrences of *substring* in the string *s*.
s.`endswith`(*substring*)	Returns `True` if the string *s* ends with the `substring` and `False` otherwise.
s.`find`(*substring*)	Returns the lowest index in the string *s* where *substring* begins, or –1 if *substring* is not found.
s.`startswith`(*substring*)	Returns `True` if the string *s* begins with *substring* and `False` otherwise.

Methods: Testing String Characteristics (1)

Table 7 Methods for Testing String Characteristics

Method	Description
`s.isalnum()`	Returns `True` if string *s* consists of only letters or digits and it contains at least one character. Otherwise it returns `False`.
`s.isalpha()`	Returns `True` if string *s* consists of only letters and contains at least one character. Otherwise it returns `False`.
`s.isdigit()`	Returns `True` if string *s* 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

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

Comparing and Analyzing Strings

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.

Substring Example

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

Input Validation

3.9

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

General rule

- Never trust user input
- When you read information from the user, **always** check that it contains acceptable values, before continuing with the program
- If values are not acceptable, print a message, and:
  - Ask again for a correct value (see Loops, Chapter 4)
  - Exit from the program:

```
from sys import exit
exit("Value not acceptable")
```

It is impossible to make anything foolproof because fools are so ingenious…
(Unattributed variant to Murphy's Law)

Summary

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: Boolean

- The type boolean has two values, `True` and `False`.
  - Python has two Boolean operators that combine conditions: `and` , `or`.
  - To invert a condition, use the `not` operator.
  - The 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 & or conditions.

Summary: python overview

- Use the `input()` function to read keyboard input in a console window.
- If the input is not a string, use `int()` or `float()` to convert it to a number
- Use the format specifiers to specify how values should be formatted.
- Use `f`-strings for easier formatting