MASTER MEIM 2022-2023

# Python Programming Course Lesson 3

Branching Programs: IF statement

Lesson given by Prof. Mariacarla Staffa

Department of Science and Technology of the University of Naples Parthenope

# Overview

- 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 part of the lesson, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.*
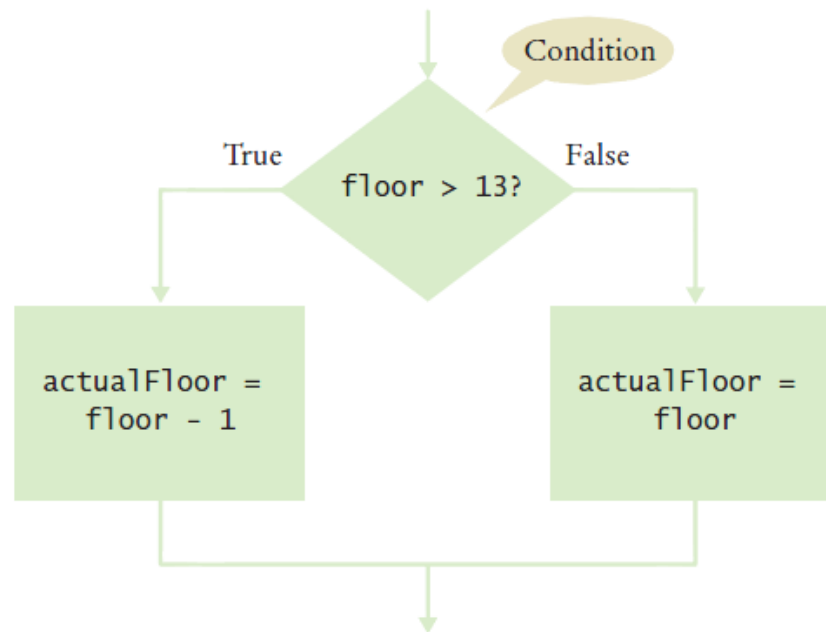
# The `if` Statement

- 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'
    - If floor > 12, Actual floor = floor - 1

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

- The two keywords of the if statement are:
  - `if`
  - `else`

# Flowchart of the `if` Statement

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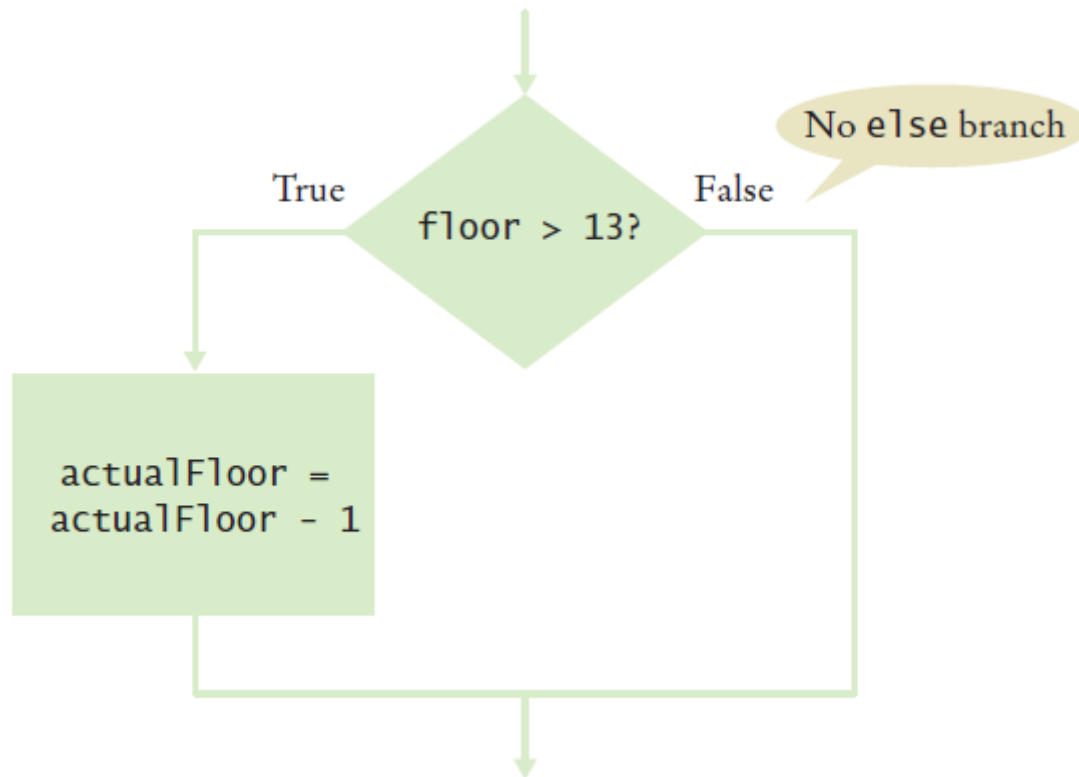


```
actualFloor = 0

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

# Exercise: elevatorsim.py

- Write a program that simulates an elevator panel that skips the 13th floor
- Given a floor in input
- Adjust the floor if necessary
- print ''The elevator will travel the the actual floor ....''

**Program Run**

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

# Compound Statements

- Some constructs in Python are **compound statements including the IF.**

- **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.

- The statement block is a group of one or more statements, <u>*all indented to the same column*</u>

- 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 login and flow of a program

# Tips on Indenting Blocks

- Let Wing 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
```
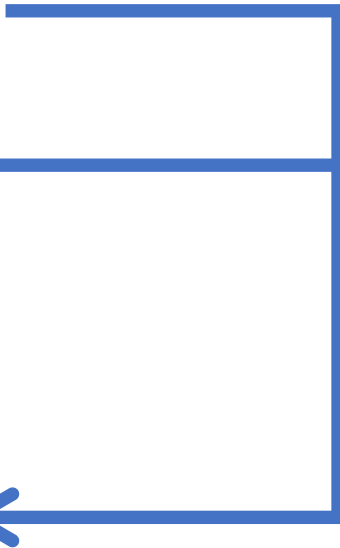
*This is referred to as "block structured" code. Indenting consistently is not only syntactically required in Python, it also makes code much easier to follow.*

# 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)
```

# 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 :
```

# Exercise: compare.py
# 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")
```

# 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
    - The order check refer to the Basic Latin (ASCII) Subset of Unicode

# 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. |

# 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:

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

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

# **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 :
```

# Exercise: sale.py

A shop gives 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.

Write a program that:

- *Ask the original price*

- Determine *the discount rate*

- *Compute and print the discount*

# 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?

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

# Nested Branches wine.py

- You can *nest* an <span style="color:red">if</span> inside either branch of an <span style="color:red">if</span> statement.
- Simple example:  Ordering drinks
  - Ask the customer for their drink order
  - <span style="color:red">if</span> customer orders wine
    - Ask customer for his/her age
    - <span style="color:red">if</span> customer's age is 21 or over
      - Serve wine
    - Else
      - Politely explain the law to the customer
  - Else
    - Serve customers a non-alcoholic drink

MASTER IN ENTREPRENEURSHIP
INNOVATION MANAGEMENT
IN COLLABORATION WITH **MIT SLOAN**

IN COLLABORATION WITH

MIT MANAGEMENT
SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

# Alternative multiple

# 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
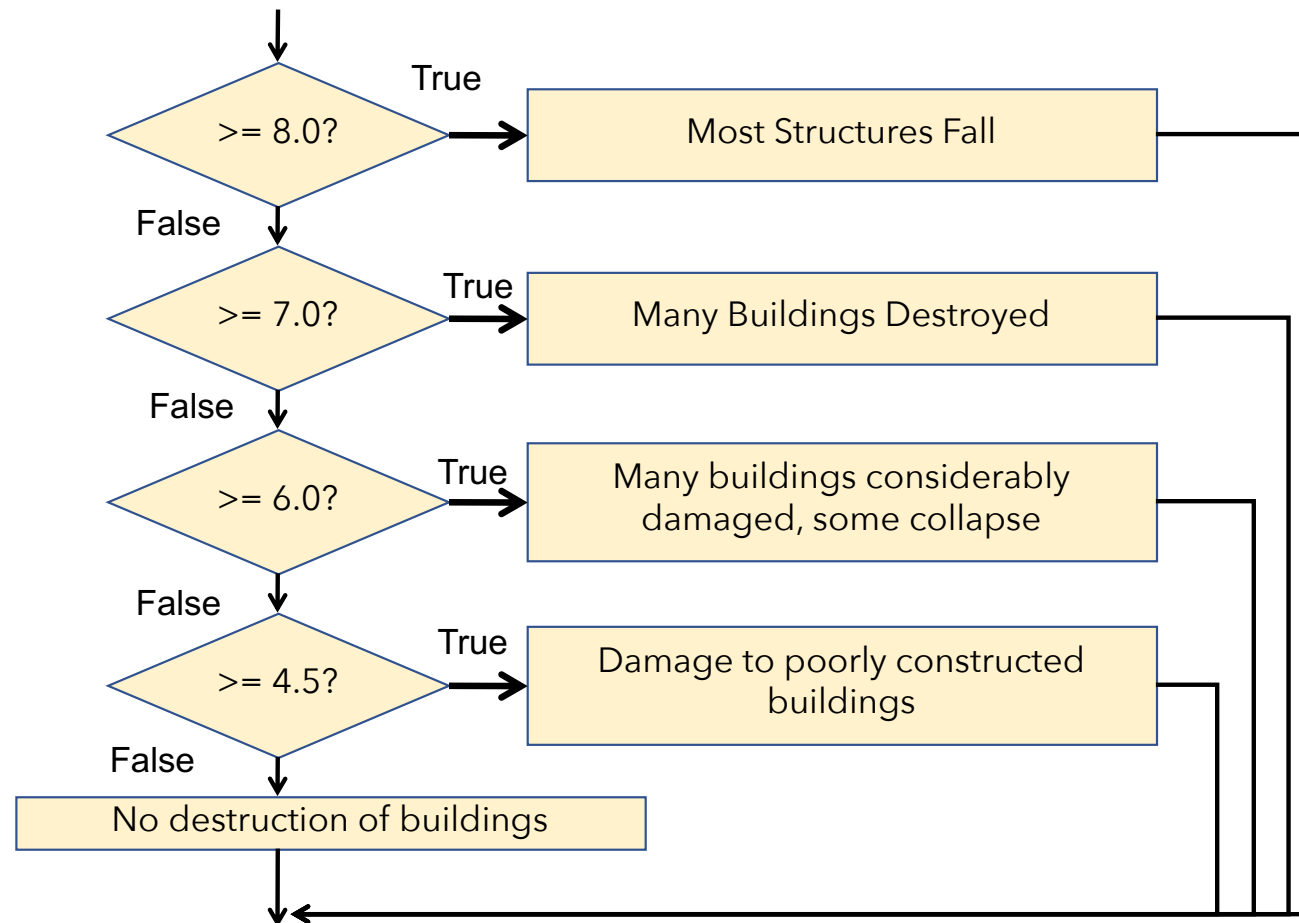
# `elif` Statement

- Short for Else, if…
- As soon as one 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

```python
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?

```python
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")
```

# Exercise: earthquake.py

Write a program that:

Enter a magnitude on the Richter scale and Print:

- ''Most structures fall'' if the magnitude is >=8.0

- ''Many buildings destroyed'' if the magnitude is >=7.0

- ''Many buildings considerably damaged'', some collapse» if the magnitude is >=6.0

- ''Damage to poorly constructed buildings'' if the magnitude is >=4.5

- ''No destruction of buildings'' in the other cases

# 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

```python
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")
```

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

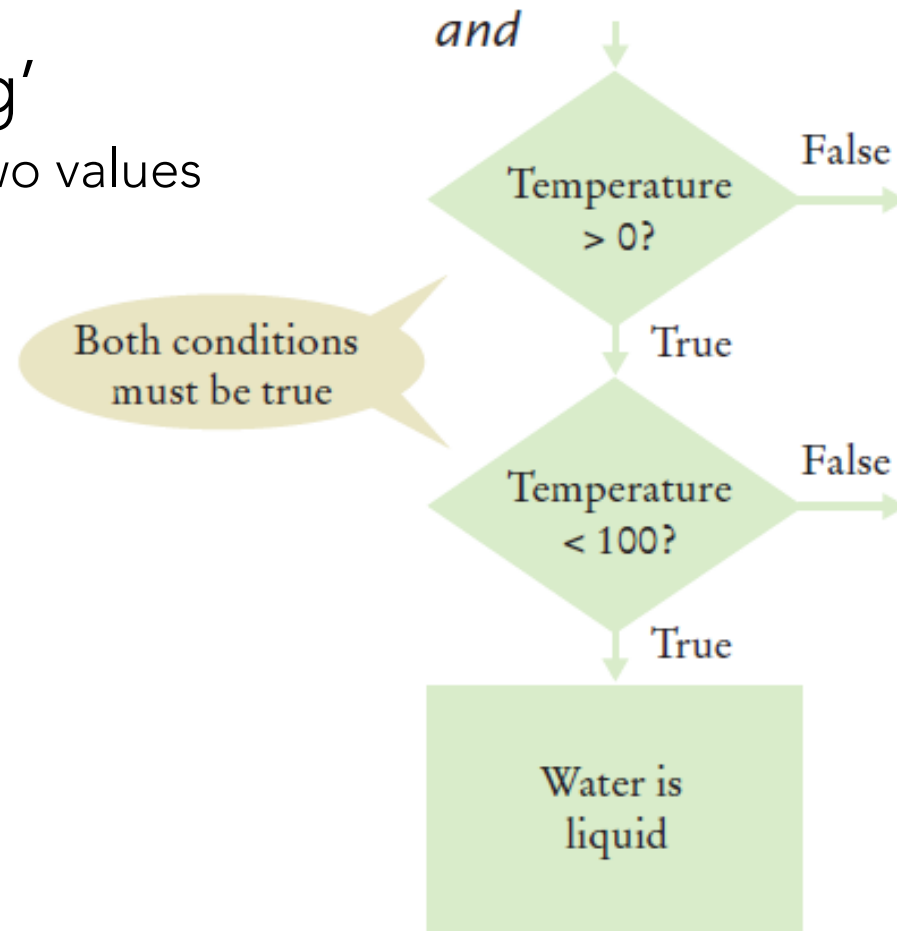| A | not A |
|-------|-------|
| True | False |
| False | True |

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

# 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

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

*and*

Temperature > 0? — False

Both conditions must be true

True

Temperature < 100? — False

True

Water is liquid

# *or* flowchart

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

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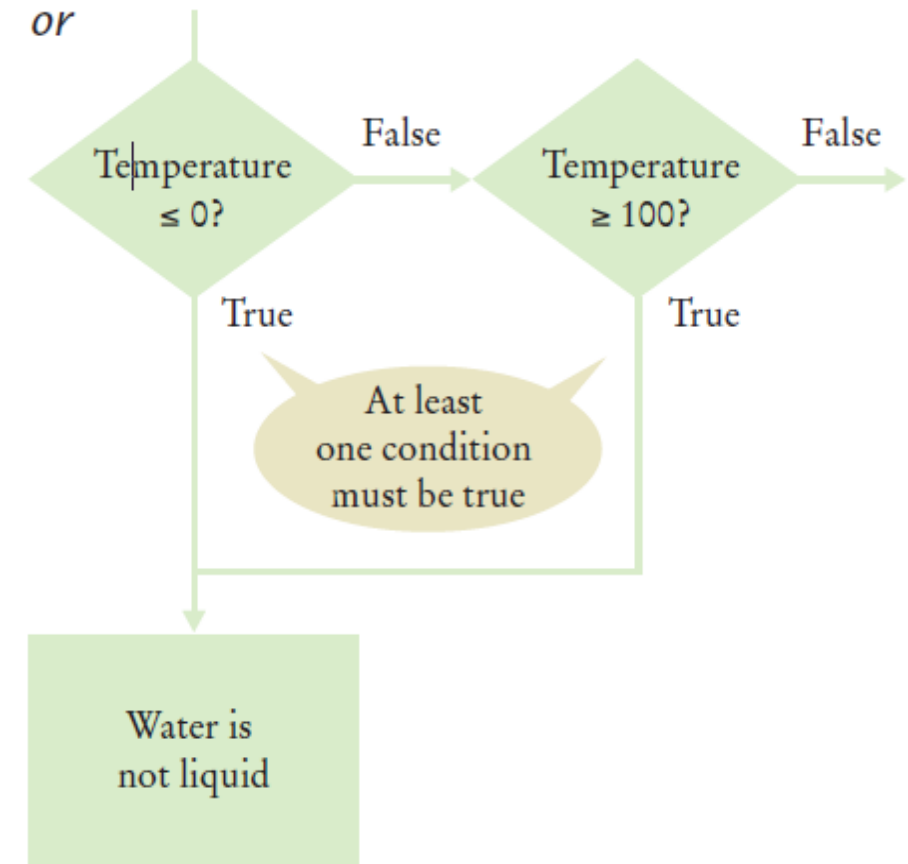
# Comparison Example

- Open the file:
  - Compare2.py

- Run the program with several inputs
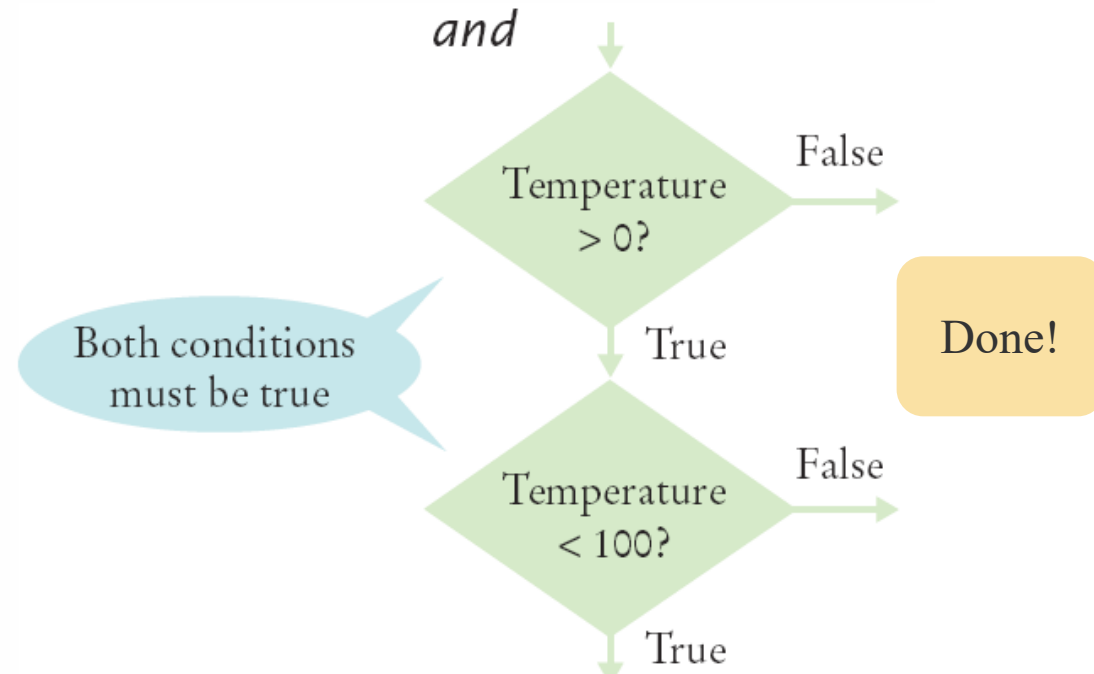
# Boolean Operator Examples

| | Table 5 Boolean Operator Examples | |
|---|---|---|
| Expression | Value | Comment |
| 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. |

# 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")
```

*and*

Temperature > 0? → False

Both conditions must be true

True

Temperature < 100? → False

Done!

True

# 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")
```



*or*

Temperature ≤ 0? — False → Temperature ≥ 100? — False →

True

Done!

True

At least one condition must be true
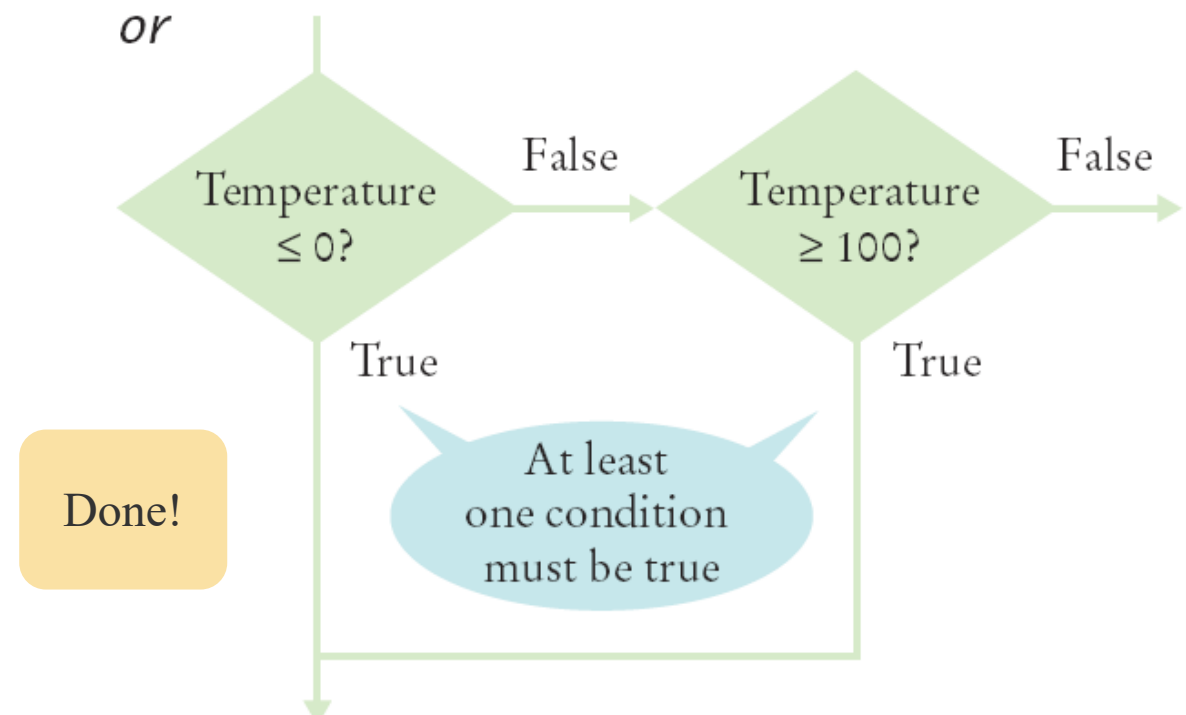
# De Morgan's law

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

MASTER IN ENTREPRENEURSHIP
INNOVATION MANAGEMENT
IN COLLABORATION WITH MIT SLOAN

IN COLLABORATION WITH
MIT MANAGEMENT
SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

# String Analysis

# 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** |
| *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. |

# Exercise: substring.py

Write a program that:

- Ask for a string and for a substring

- Check if the substring is contained in the string.

- If yes:
  - Count the occurrences of substring in string
  - Indicate whether the string start or finish with the  substring

- If no:
  - Print "the *substring* is not contained in *string*"

# 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 (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. |

# Exercise: id_student.py

- Write a program that:

- Ask for a id_student. By taking in mind that a student id must be composed of a first uppercase Letter, the number "85" and a sequence of other 6 random numbers, verify the correctness of the id.

- Try with these ids: N87001324, N85003456, n85006734, N8500NN89, N8500

# 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 `String`s.

- `String`s 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.