Decision Structures

Reading Assignment

- Chapter 3 Sections 1, 2, 3, 4, 7, 8 and 9.

Chapter Learning Outcomes

At the end of this chapter, you will be able to

- To implement decisions using if statements
- To compare integers, floating-point numbers, and strings
- To write statements using Boolean expressions
- To validate user input

The if Statement

- A computer program often needs to make decisions based on input, or circumstances.
- The **if statement** allows a program to carry out different actions depending on the nature of the data to be processed.

- The **if statement** is used to implement a decision.
- The two keywords of the **if statement** are:
  - *if*
  - *else*

- When a condition is fulfilled, one set of statements is executed.
- Otherwise, another set of statements is executed.

The if Statement - Example

- In some countries, the number 13 is considered unlucky.
  - To avoid offending tenants, buildings owners sometimes skip the 13th floor; floor 12 is immediately followed by floor 14.
  - Of course, floor 13 is not left empty, it is simply called floor 14.

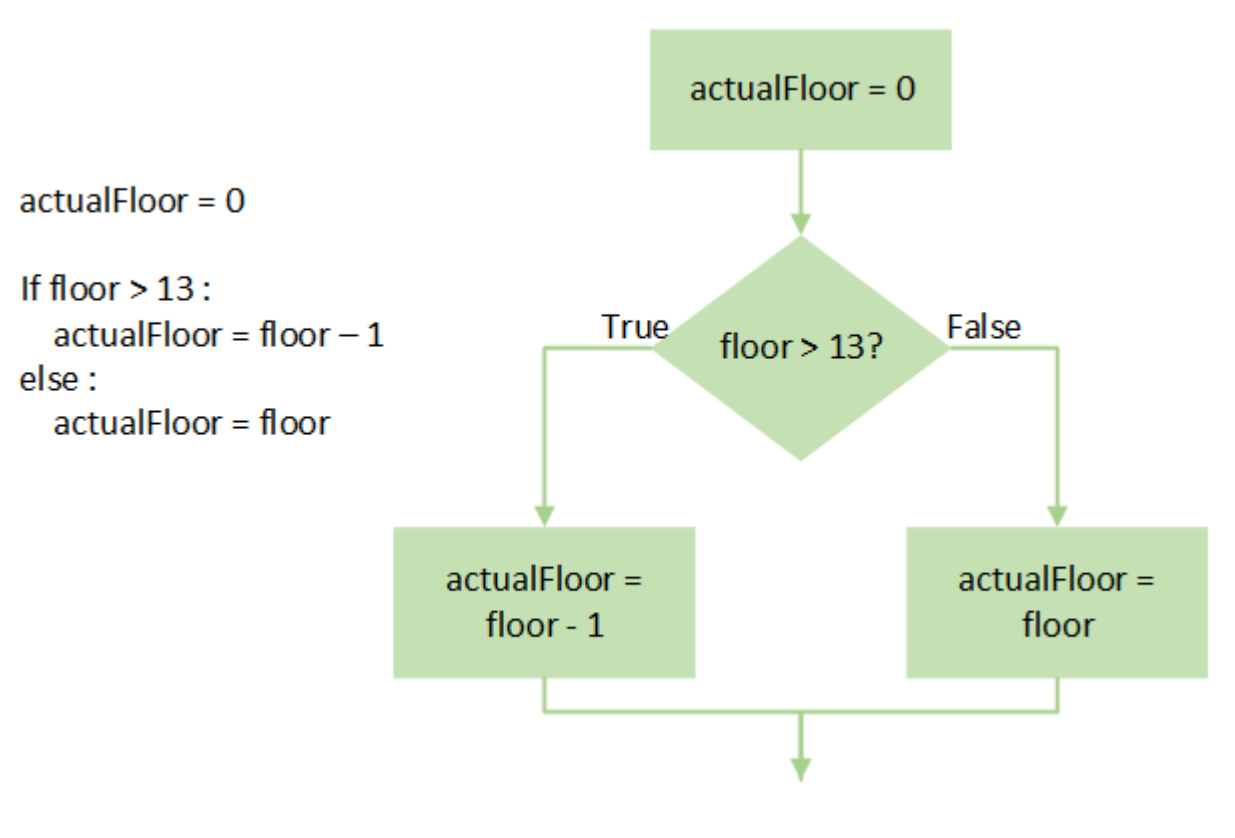
- The computer that controls the building elevators needs to compensate this foible and adjust all floor numbers above 13.

The if Statement - Example

- Let's simulate this process in Python.
  - We will ask the user to type in the desired floor number and then compute the actual floor.
  - When the input is above 13, then we need to decrement the input to obtain the actual floor.

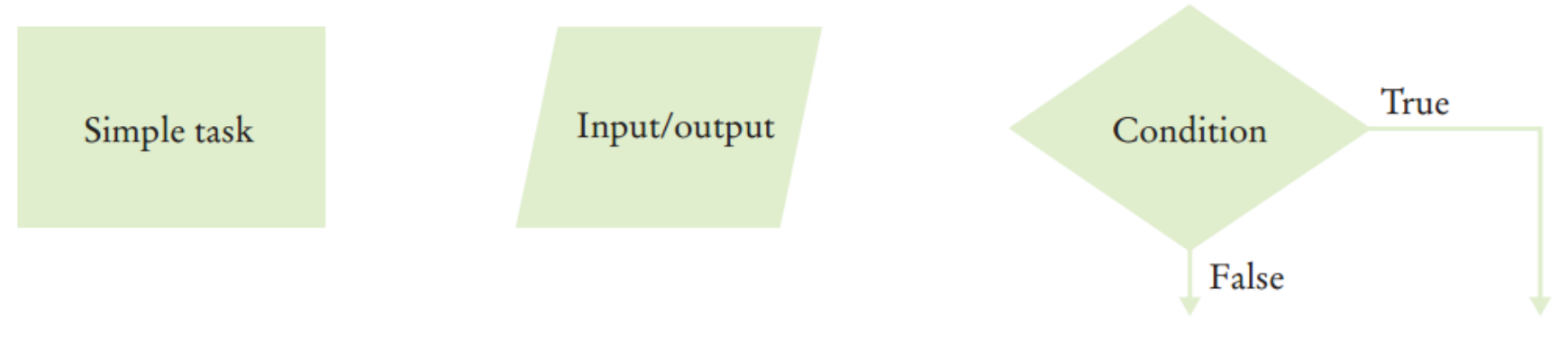
The if Statement - Example

- For example, if the user provides an input of 20, the program determines the actual floor as 19.
- Otherwise, we simply use the supplied floor number.



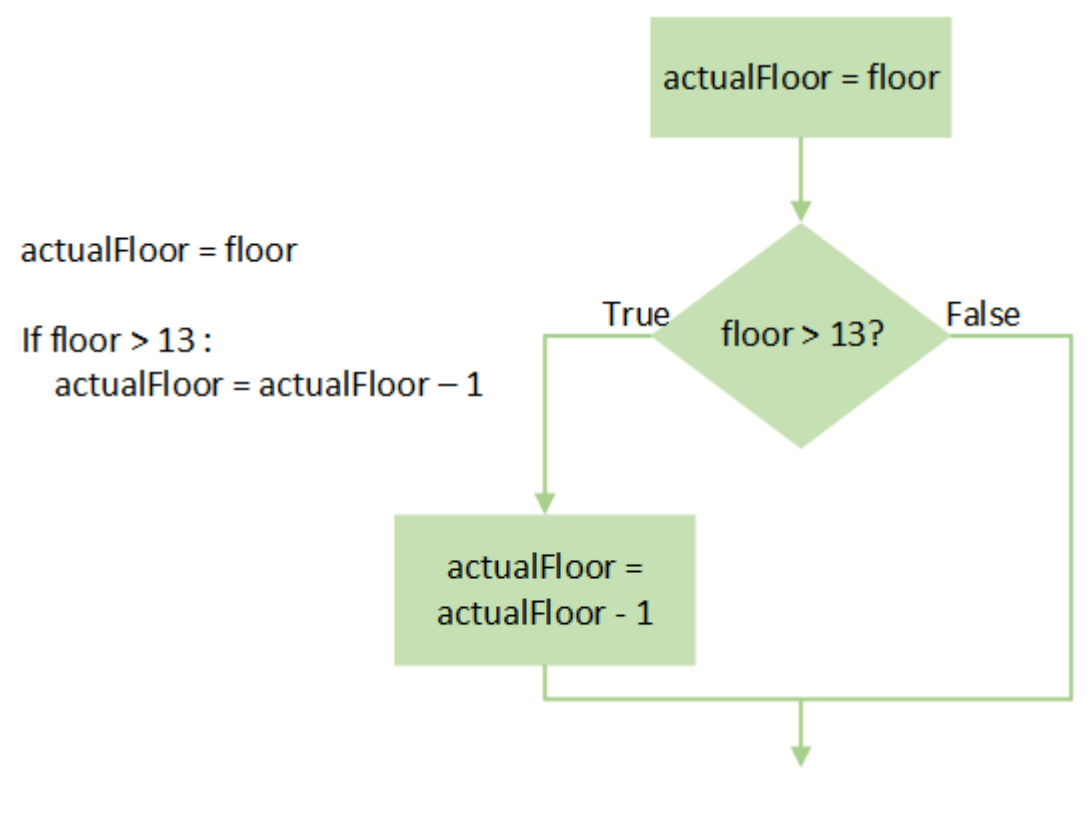
- The figure is called a **flowchart**.
- A **flowchart** shows the structure of decisions and tasks that are required to solve a certain problem (usually a complex one).
- Flowcharts help the programmer to visualize the flow of control.

Elements of a Flowchart



The if Statement - Example

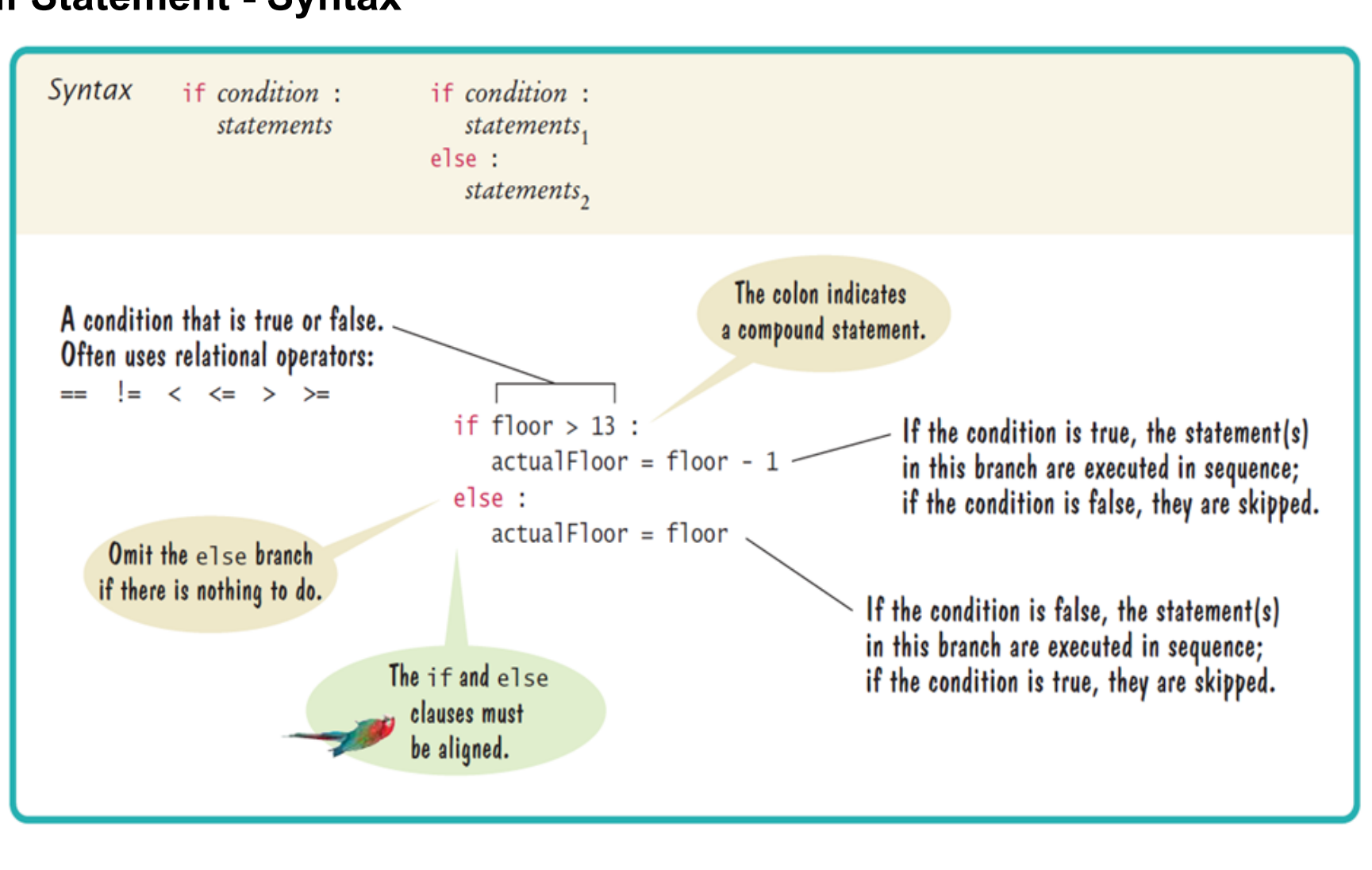
- Sometimes, it happens that there is nothing to do in the else branch of the statement.
  - In that case, you can omit it entirely, such as in this example:



The Compound if Statement

- Some constructs in Python are **compound** statements, which span **multiple lines** and consist of a ***header*** and a ***statement block***.
- In **Compound** statements,
  - the header requires a **colon (:)** at the end of it, and
  - the statement block consisting of a group of one or more statements, is such that each statement is indented to the **same indentation level**.
- A **statement block** begins on the line following the header and ends at the first statement indented less than the first statement in the block.
- Any number of spaces can be used to indent statements within a block, but all statements within the block must have the same indentation level.
- Note that **comments** are not statements and thus can be indented to any level.

if Statement - Syntax




```python
##
#  This program simulates an elevator panel that skips the 13th floor.
#

# Obtain the floor number from the user as an integer.
floor = int(input("Floor: "))

# Adjust floor if necessary.
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor

# Print the results
print("The elevator will travel to the actual floor", actualFloor)
```

Student Activity

- In some Asian countries, the number 14 is considered unlucky. Some building owners play it safe and skip both the thirteenth and the fourteenth floor. How would you modify the sample program to handle such a building?

```python
## Enter your code here.
```

Relational Operators

- Every if statement contains a condition.
- In many cases, the condition involves comparing two values.
  - For example, in the previous examples we tested floor > 13.
- The comparison > is called a **relational operator**.

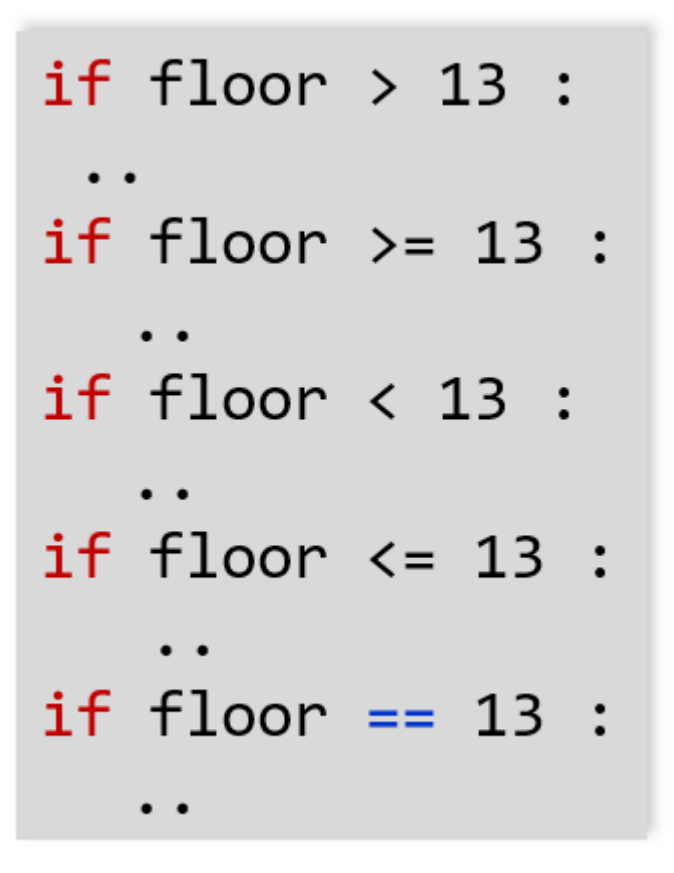


Table 1 Relational Operators

Python	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Relational Operators

Assignment vs. Equality Testing

- In Python, **=** already has a meaning, namely **assignment**.
- The **==** operator denotes **equality testing**:
  - floor = 13 (**# Assign 13 to floor**)
  - if floor == 13 : (**# Test whether floor equals 13**)

Comparing Strings

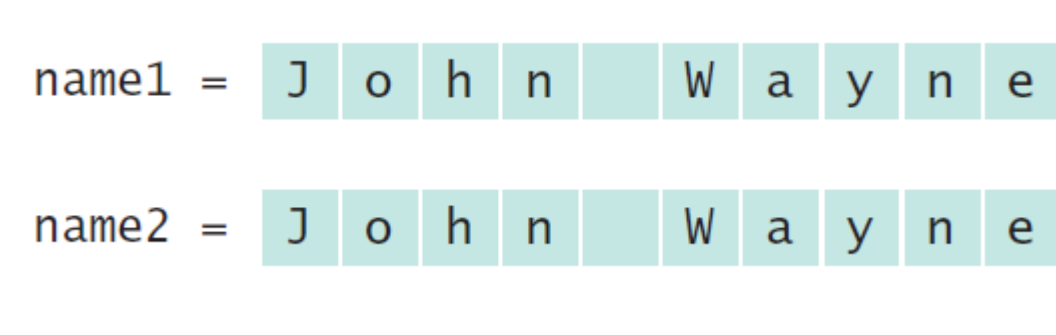
- Strings can also be compared using Python's relational operators.
  - For example, to test whether two strings are equal, use the **==** operator.
  - or to test if they are not equal, use the **!=** operator.

```python
name1 = input("Enter the first name ")
name2 = input("Enter the second name ")
if name1 == name2:
    print("The strings '"+ name1 + "' and '" + name2 + "' are identical.")
```

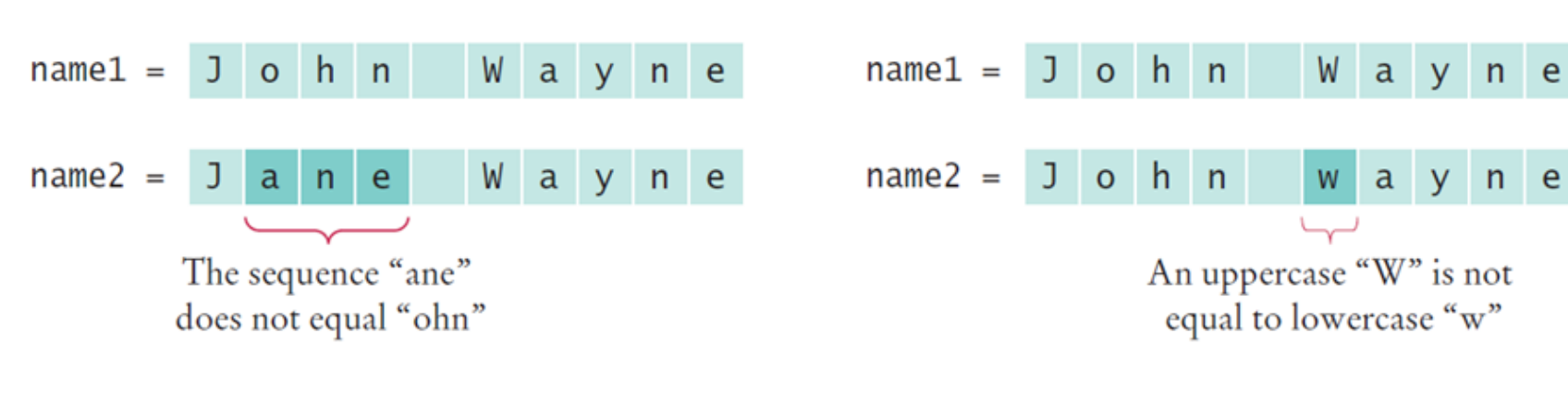
```python
name1 = input("Enter the first name ")
name2 = input("Enter the second name ")
if name1 != name2:
    print("The strings '"+ name1 + "' and '" + name2 + "' are not identical.")
```

When are two strings equal?

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



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



Student Activity

```python
print(3 <= 4)
```

```python
print(3 =< 4)
```

```python
print(3 > 4)
```

```python
print(4 < 4)
```

```python
print(4 <= 4)
```

```python
print(3!=5-3)
```

```python
print(3==6/2)
```

```python
print(1.0 / 3.0 == 0.333333333)
```

```python
print(3<=4<5)
```

```python
print("AB" < "AC")
```

```python
print('AB' == chr(65)+chr(66))
```

```python
print("10" > 5)
```

```python
s1 = "This is a long string."
s2 = "This is a l0ng string."
if s1==s2:
    comparison = "identical"
else:
    comparison = "not identical"
print ("The string s1 and s2 are",comparison)
```

Nested Branches

- It is often necessary to include an **if statement** inside another. Such an arrangement is called a **nested set of statements**.
  - Nested decisions are required for problems that have multiple levels of decision making.

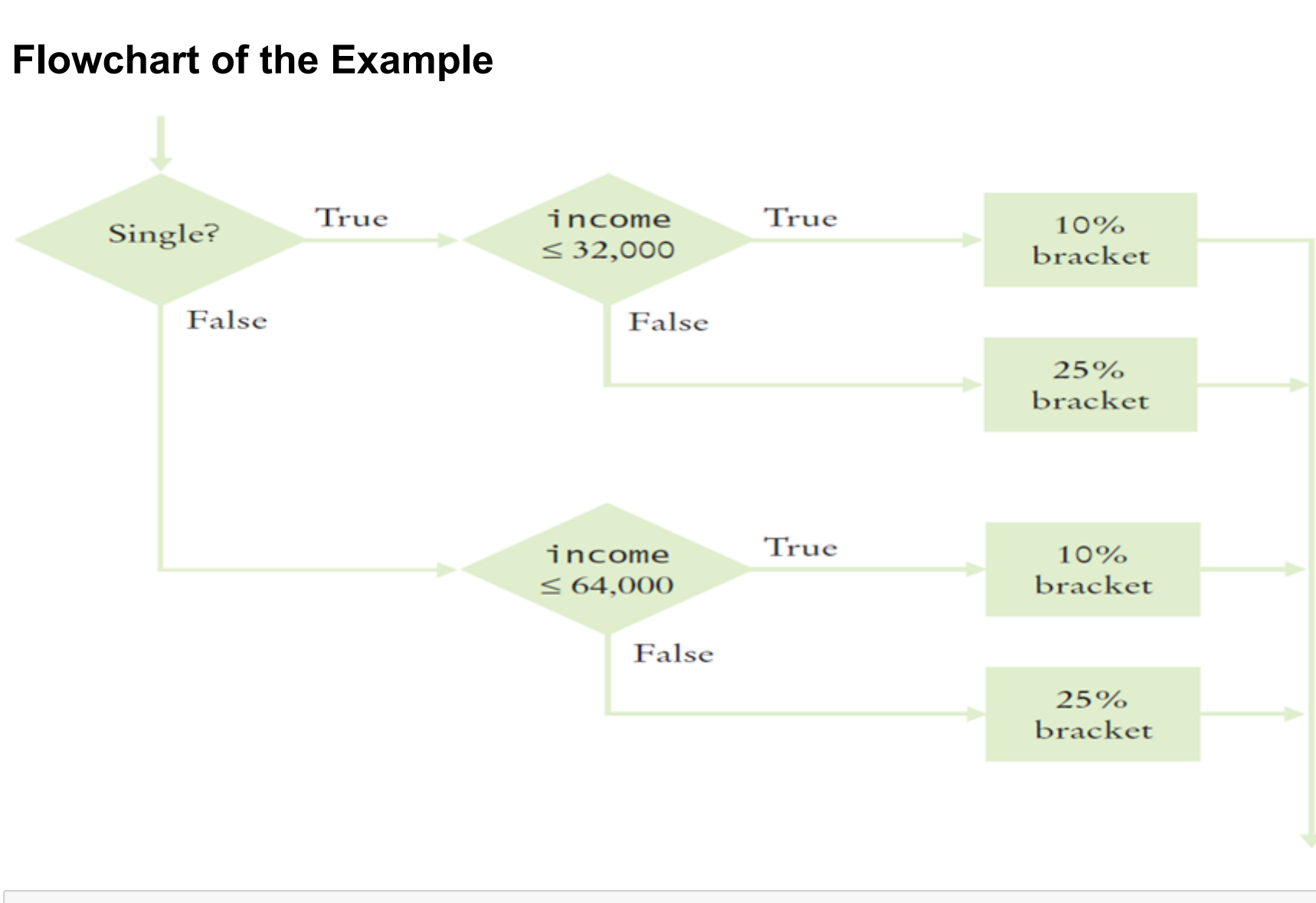
Nested Branches (Example)

- In the United States, different tax rates are used depending on the taxpayer's marital status. There are different tax schedules for single and for married taxpayers. Married taxpayers add their income together and pay taxes on the total.

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 of the Example




```python
#  This program computes income taxes, using a simplified tax schedule.
# Initialize constant variables for the tax rates and rate limits.
RATE1 = 0.10
RATE2 = 0.25
RATE1_SINGLE_LIMIT = 32000.0
RATE1_MARRIED_LIMIT = 64000.0
# Read income and marital status
income = float(input("Please enter your income: "))
maritalStatus = input("Please enter s for single, m for married: ")
# Compute taxes due.
tax1 = 0
tax2 = 0
if maritalStatus == "s" :
   if income <= RATE1_SINGLE_LIMIT :
      tax1 = RATE1 * income
   else :
      tax1 = RATE1 * RATE1_SINGLE_LIMIT
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
else :
   if income <= RATE1_MARRIED_LIMIT :
      tax1 = RATE1 * income
   else :
      tax1 = RATE1 * RATE1_MARRIED_LIMIT
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)

totalTax = tax1 + tax2
# Print the results.
print("The tax is $%.2f" % totalTax)
```

Nested Branches - Student Activity

- Write a program that reads an integer and prints whether it is `negative`, `zero`, or `positive`.

Multiple Alternatives

- Multiple if statements can be combined to evaluate complex decisions.
- For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale

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

Multiple Alternatives

- You could use multiple **if statements** to implement multiple alternatives, like this:

```python
richter = 8.0
if richter >= 8.0:
    print("Most structures fall")
else:
    if richter >= 7.0:
        print("Many building destroyed")
    else:
        if richter >= 6.0:
            print("Many buildings considerably damaged, some collapse")
        else:
            if richter >= 4.5:
                print("Damage to poorly constructed buildings")
            else:
                print("No destruction of buildings")
```

- but this becomes difficult to read and, as the number of branches increases, the code begins to shift further and further to the right due to the required indentation.

Multiple Alternatives

- Python provides the special construct **elif** for creating **if** statements containing multiple branches.
- Using the **elif** statement, the code segment can be rewritten as:

```python
richter = 8.0
if richter >= 8.0:
    print("Most structures fail")
elif richter >= 7.0:
    print("Many buildings destroyed")
elif richter >= 6.0:
    print("Many buildings considerably damaged, some collapse")
elif richter >= 4.5:
    print("Damage to poorly constructed buildings")
else:
    print("No destruction of buildings")
```

- As soon as one of the four tests succeeds, the effect is displayed, and no further tests are attempted.
- If none of the four cases applies, the **final else clause** applies, and a default message is printed.

Multiple Alternatives

- Here you must sort the conditions and test against the largest cutoff first.
- Suppose we reverse the order of tests:

```python
richter = 7.1
if richter >= 4.5: #Tests in worng order
    print("Damage to poorly constrcuted buildings")
elif richter >= 6.0:
    print("Many buildings considerably damaged, some collapse")
elif richter >= 7.0:
    print ("Many buildings destroyed")
elif richter >= 8.0:
    print ("Most structures fail")
```

- The remedy is to test the more specific conditions first.
  - Here, the condition **richter >= 8.0** is more specific than the condition **richter >= 7.0**,
  - and the condition **richter >= 4.5** is more general (that is, fulfilled by more values) than either of the first two.

Multiple Alternatives

- In this example, it is also important that we use an **if/elif** sequence, not just multiple independent if statements.
- Consider this sequence of independent tests.

```python
richter = 7.1
if richter >= 8.0: #Didn't use else
    print("Most structures faill")
if richter >= 7.0:
    print("Many buildings destroyed")
if richter >= 6.0:
    print("Many buildings considerably damaged, some collapse")
if richter >= 4.5:
    print("Damge to poorly constrcuted buildings")
```

- Now the alternatives are no longer exclusive. If **richter** is 7.1, then the last three tests all match, and three messages are printed.

Multiple Alternatives

```python
#  This program prints a description of an earthquake, given the Richter scale
#  magnitude.
# Obtain the user input.
richter = float(input("Enter a magnitude on the Richter scale: "))
# Print the description
if richter >= 8.0 :
   print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else :
   print("No destruction of buildings")
```

Boolean Variables and Operators

- Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere.
- To store a condition that can be true or false, you use a **Boolean variable**.
  - A boolean variable is also called a flag because it can be either up (**True**) or down (**False**).

- In Python, the **bool** data type has exactly two values, denoted `False` and `True`.
- These values are not strings or integers; they are special values, just for **Boolean variables**.

```python
flag = True
print("The variable flag is of type", type(flag))
if flag :
    print ("A flag is raised")
else :
    print("No flag is raised")
```

Logical `and` and `or` Operators

- Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water.
  - (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.)
  - Water is liquid if the temperature is greater than zero and less than 100:

```python
temp = 10
if temp > 0 and temp < 100:
    print("Liquid")
```

- The condition of the test has two parts, joined by the **and** operator.
- Each part is a Boolean value that can be `True` or `False`.
- The combined expression is `True` if both individual expressions are `True`.
- If either one of the expressions is `False`, then the result is also `False`.

- Similarly, we have the **or** operator.
- A combined expression with the **or** operator is `True` if at least one of them is `True`.
- If both expressions are `False`, then the result is also `False`.

Truth Tables for the Logical Operators

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

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

A	not A
True	False
False	True

- Python utilizes **short circuit evaluation** when evaluating expressions involving the logical operators **and** and **or**
  - If the first expression of the **and** evaluates to `False`, the second expression is not evaluated.
  - Similarly for the expressions of the **or** operator. Can you determine how?

- Consider the following example:

```python
value1 = 6
value2 = 10
value3 = 0
if value1 > value2 and value2/value3 > 1 :
    print("This evaluates to True")
else :
    print("This evaluates to False, but at least, it does not crash")
```

Conditions "Values"

- A condition in an `if` statement can be `True` or `False`, for example

```python
print(4 < 5)
print(4 >= 5)
```

- A condition in an `if` statement that has non-zero value is always considered to be true, although its value is not a Boolean type. As a result, a *zero* of *any type* is considered false.

```python
if 3:
    print('test1')
if 0:
    print('test2')
if 0.0101:
    print('test3')
if 0.000000 :
    print('test4')
if "" :
    print('test5')
if not "" :
    print('test6')
```

Boolean Variables and Operators - Student Activity

- Let us test whether water is not liquid at a given temperature.
- That is the case when the temperature is at most 0 or at least 100.

```python

```

Student Activity

In [ ]:
```
#  This program demonstrates comparisons of numbers, using Boolean expressions.
x = float(input("Enter a number (such as 3.5 or 4.5): "))
y = float(input("Enter a second number: "))

if x == y :
   print("They are the same.")
else :
   if x > y :
      print("The first number is larger")
   else :
      print("The first number is smaller")

   if -0.01 < x - y and x - y < 0.01 :
      print("The numbers are close together")

   if x > 0 and y > 0 or x < 0 and y < 0 :
      print("The numbers have the same sign")
   else :
      print("The numbers have different signs")
```

Analyzing Strings

- Sometimes it is necessary to determine if a string contains a given **substring**.
  - i.e., one string contains an exact match of another string.
- for example, given the code segment,

In [ ]:
```
name = "John Wayne"
print("Way" in name)
```

- Python also provides the inverse of the `in` operator, **`not in`**

Operators 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.

In [ ]:
```
name = "John Johnson"
print("john" in "John Johnson")
```

In [ ]:
```
print("ho" not in name)
```

In [ ]:
```
print(name.count("oh"))
```

In [ ]:
```
print(name.find("oh"))
```

In [ ]:
```
print(name.find("ho"))
```

In [ ]:
```
print(name.startswith("john"))
```

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`.
`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`.

In [ ]:
```
name = "John Johnson"
name.isspace()
```

In [ ]:
```
name.isalnum()
```

In [ ]:
```
"1729".isdigit()
```

In [ ]:
```
"-1729".isdigit()
```

Input Validation

- An important application for the if statement is input validation.
- Whenever your program accepts user input, you need to make sure that the user-supplied values are valid before you use them in your computations

- Consider our elevator simulation program.
- Assume that the elevator panel has buttons labeled 1 through 20 (but not 13). - The following are illegal inputs:
  - The number 13
  - Zero or a negative number
  - A number larger than 20
  - An input that is not a sequence of digits, such as five.

Input Validation

- In each of these cases, we will want to give an error message and exit the program.
- It is simple to guard against an input of 13:

In [ ]:
```
floor = 13
if floor == 13:
    print("Error: There is no thirteen floor.")
```

- Here is how you ensure that the user does not enter a number outside the valid range:

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

Input Validation

In [ ]:
```
#  This program simulates an elevator panel that skips the 13th floor,
#  checking for input errors.
# Obtain the floor number from the user as an integer.
floor=input("Floor: ")

# Make sure the user input is valid.
if floor.isdigit() :
    floor = int(floor)

    if floor == 13 :
        print("Error: There is no thirteenth floor.")
    elif floor <= 0 or floor > 20 :
        print("Error: The floor must be between 1 and 20.")
    else :
        # Now we know that the input is valid
        actualFloor = floor
        if floor > 13 :
            actualFloor = floor - 1
        print("The elevator will travel to the actual floor", actualFloor)

else : # the user did not input digits
    print("Error: You should enter only digits.")
```

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.
- 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 `bool` 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.