PGR107
Python Programming

Lecture 3 – Decisions

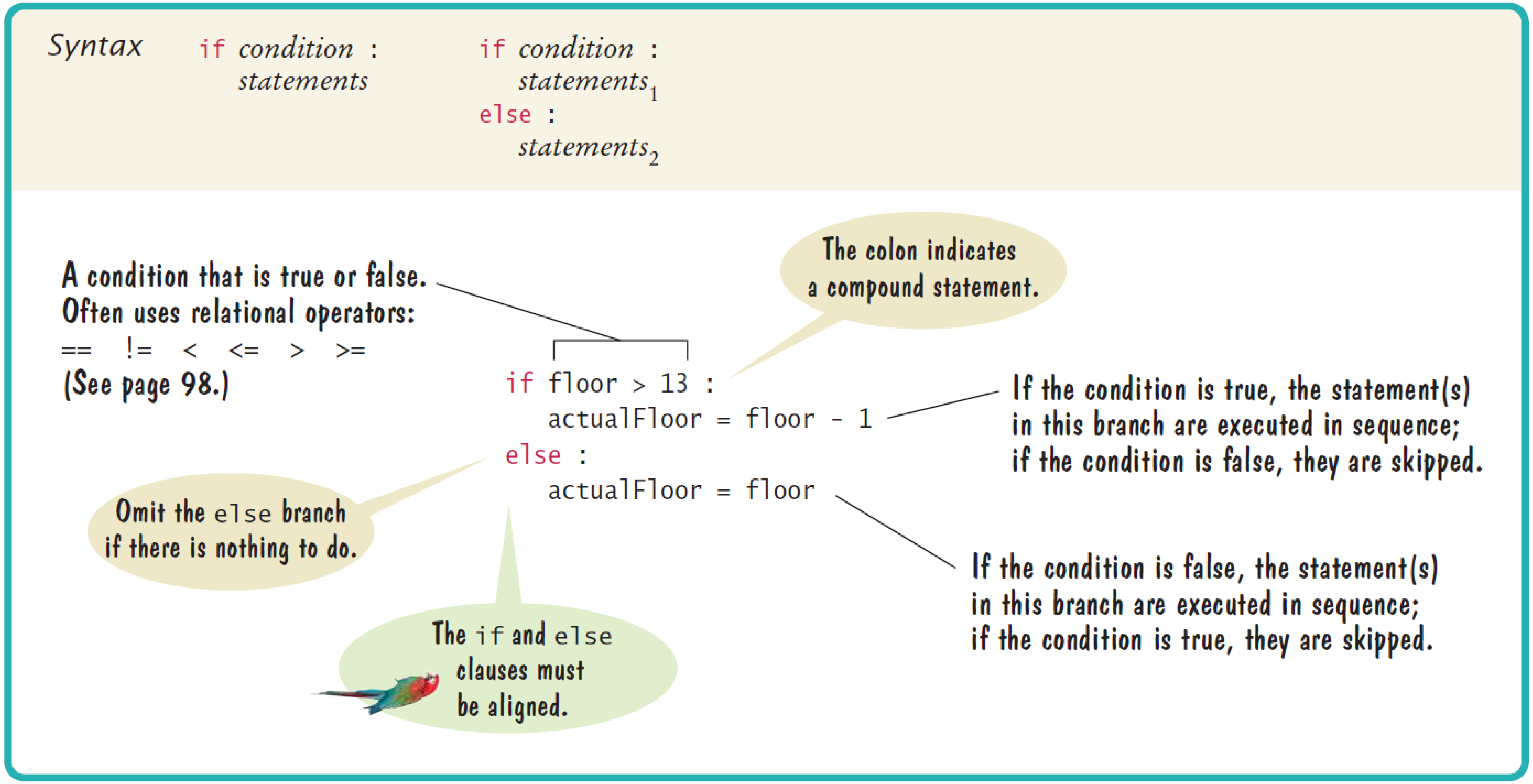
Kristiania University College

Chapter 3 - Decisions

Chapter Goals

- To implement decisions using if statements
- To compare integers, floating-point numbers, and strings
- To write statements using Boolean expressions
- To develop strategies for testing your programs
- To validate user input

The if Statement



Syntax

```
if condition :
    statements
```

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

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

A condition that is true or false. Often uses relational operators: == != < <= > >= (See page 98.)

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.

Omit the else branch if there is nothing to do.

The if and else clauses must be aligned.

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

The if Statement

- The statement block in “if statement” can be a group of one or more statements.

```
if totalSales > 100.0 :    # The header ends in a colon.
   discount = totalSales * 0.05    # Lines in the block are indented to the same level
   totalSales = totalSales - discount
   print("You received a discount of", discount)
```

Statements in a statement block must be indented to the same level.

Conditional Expressions (Optional)

- Python has a conditional operator of the form

  value1 if condition else *value2*

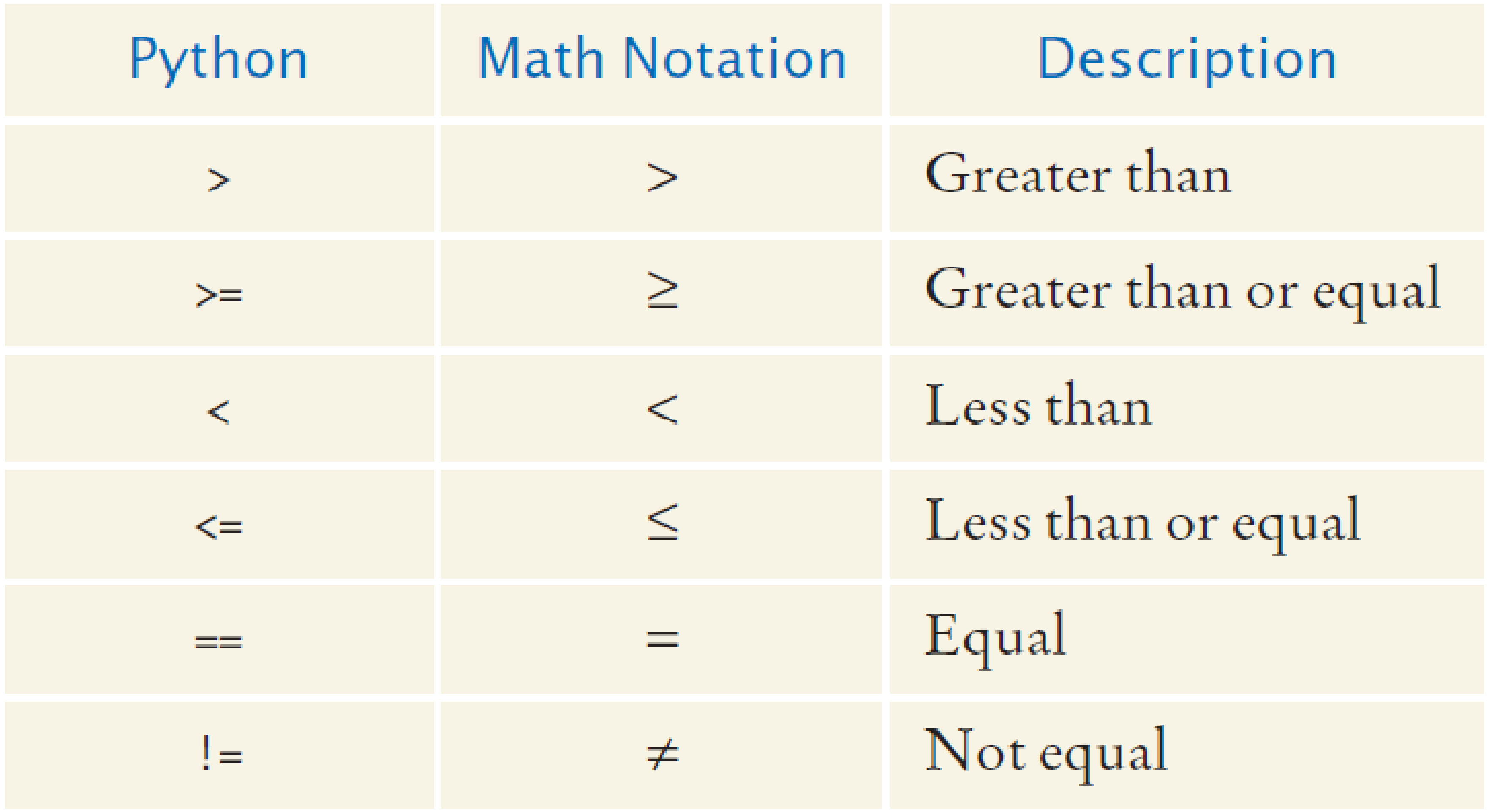
- The value of that expression is either value1 if the condition is true or value2 if it is false.

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

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

Relational Operators

- In Python, you use a relational operator to compare numbers and strings.



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

Relational Operators

- **Example 1:**

```
floor = 13     # Assign 13 to floor
if floor == 13 :     # Test whether floor equals 13
```

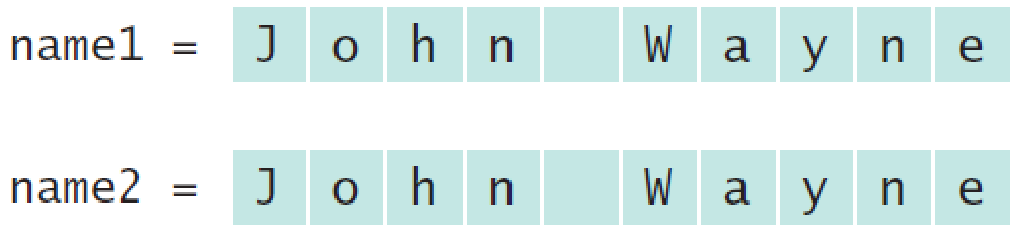
- **Example 2:**

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

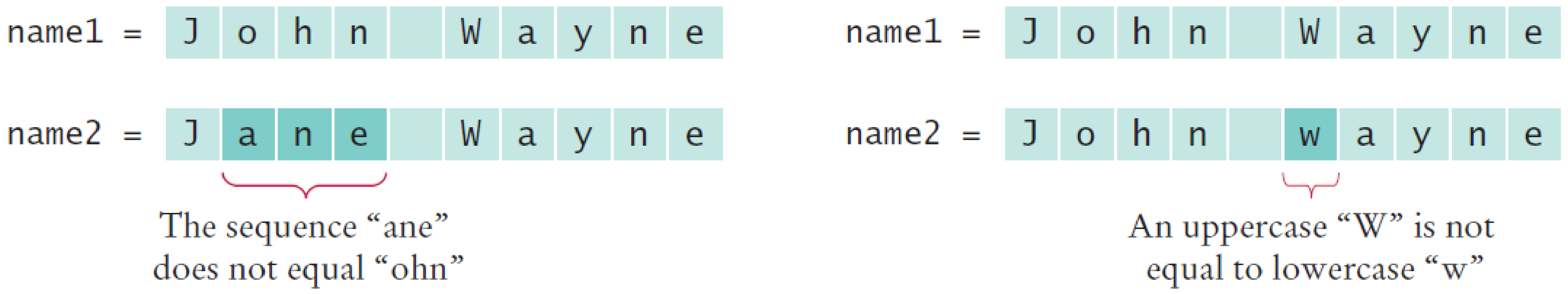
Relational Operators

- **Example 3:**

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

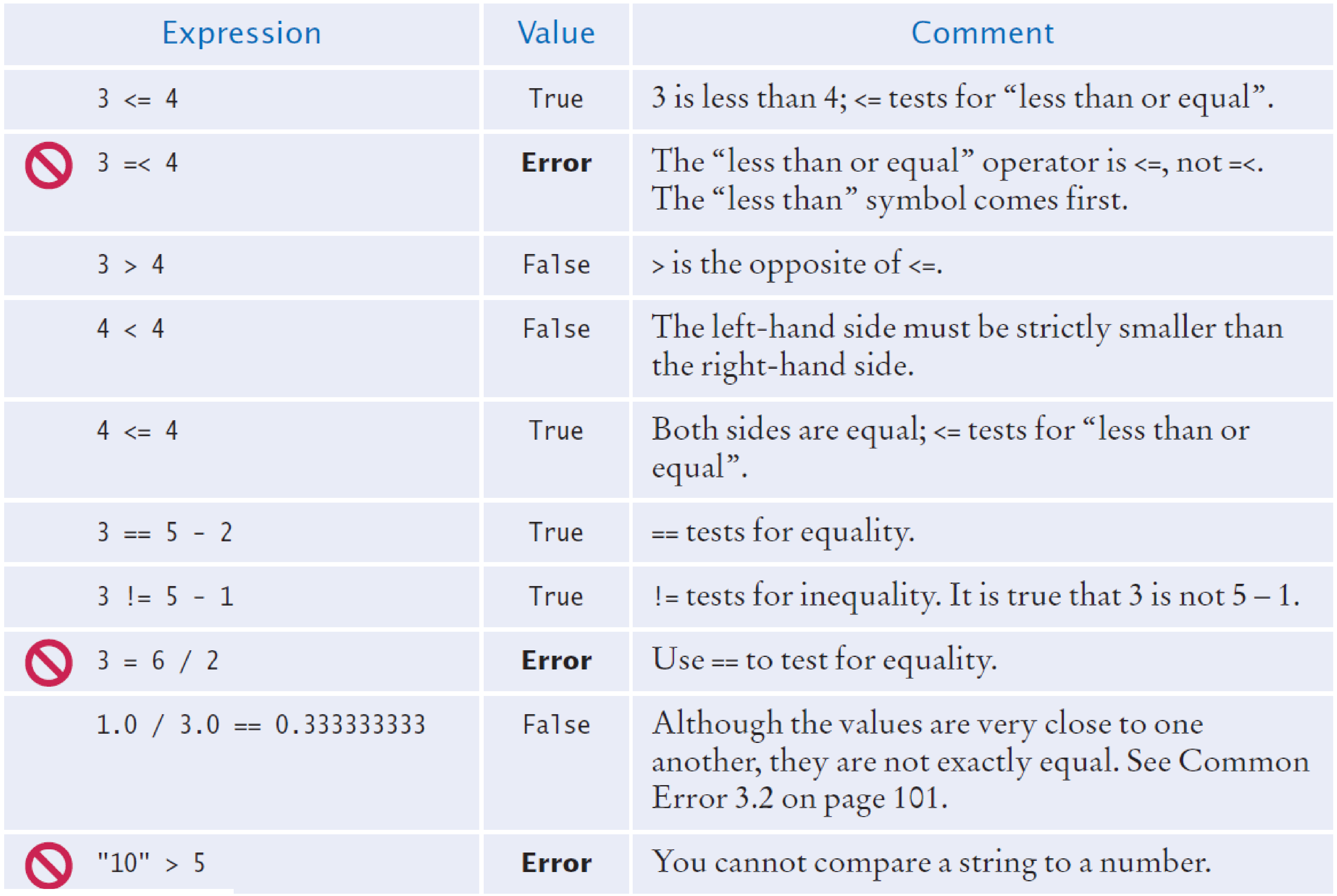


If even one character is different, the two strings will not be equal:



name1 = J o h n W a y n e
name2 = J a n e W a y n e
The sequence "ane" does not equal "ohn"
name1 = J o h n W a y n e
name2 = J o h n w a y n e
An uppercase "W" is not equal to lowercase "w"

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.

Lexicographic Ordering of Strings

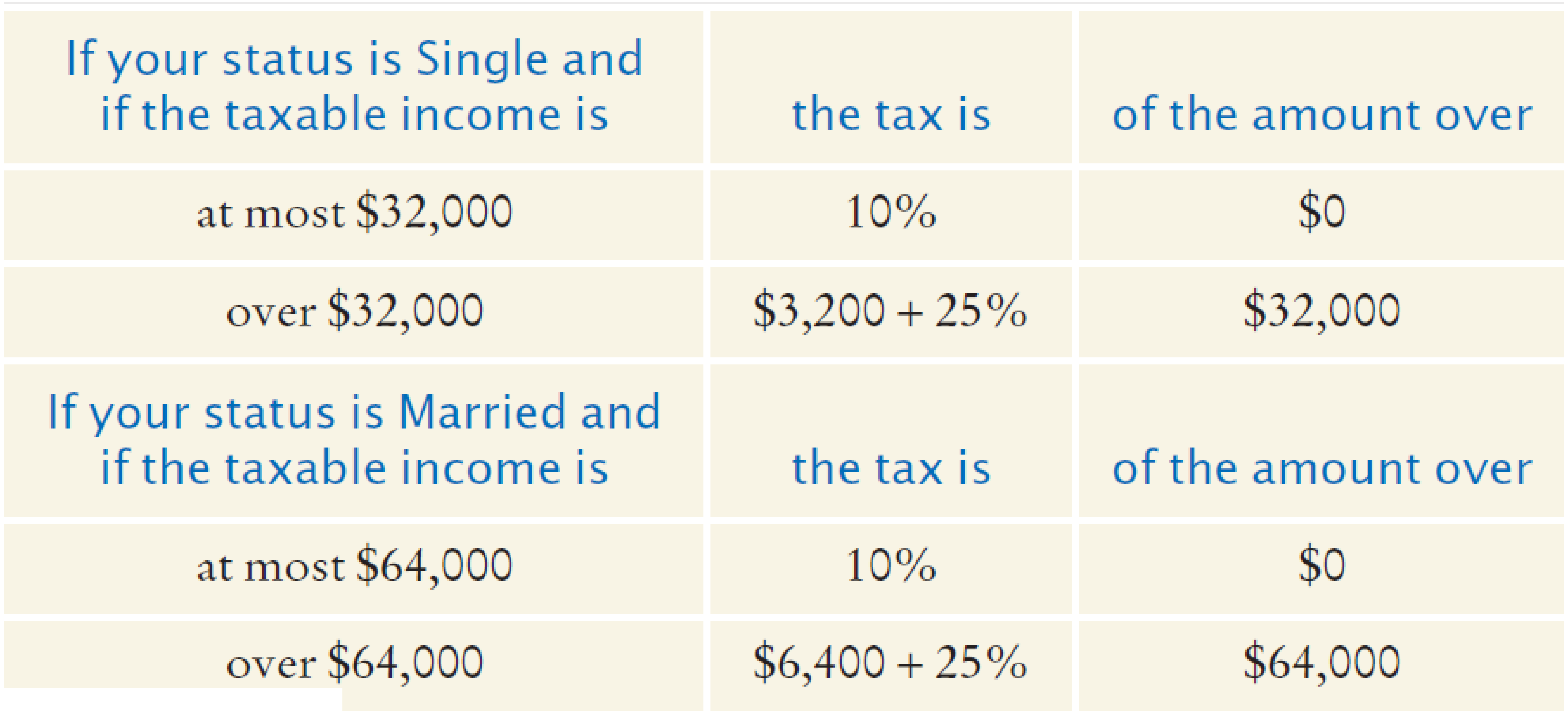
- In Python:
  - ✓ All uppercase letters come before the lowercase letters. For example, "Z" comes before "a".
  - ✓ The space character comes before all printable characters.
  - ✓ Numbers come before letters.

- If one of the strings ends, the longer string is considered the “larger” one. For example, compare **"car"** with **"cart"**. The first three letters match, and we reach the end of the first string. Therefore "car" comes before "cart" in the lexicographic ordering.

- When you reach a mismatch, the string containing the “larger” character is considered “larger”. For example, let’s compare **"cat“** with **"cart"**. The first two letters match. Because <u>**t**</u> comes after <u>**r**</u>, the string "cat" comes after "cart" in the lexicographic ordering.

Nested Branches

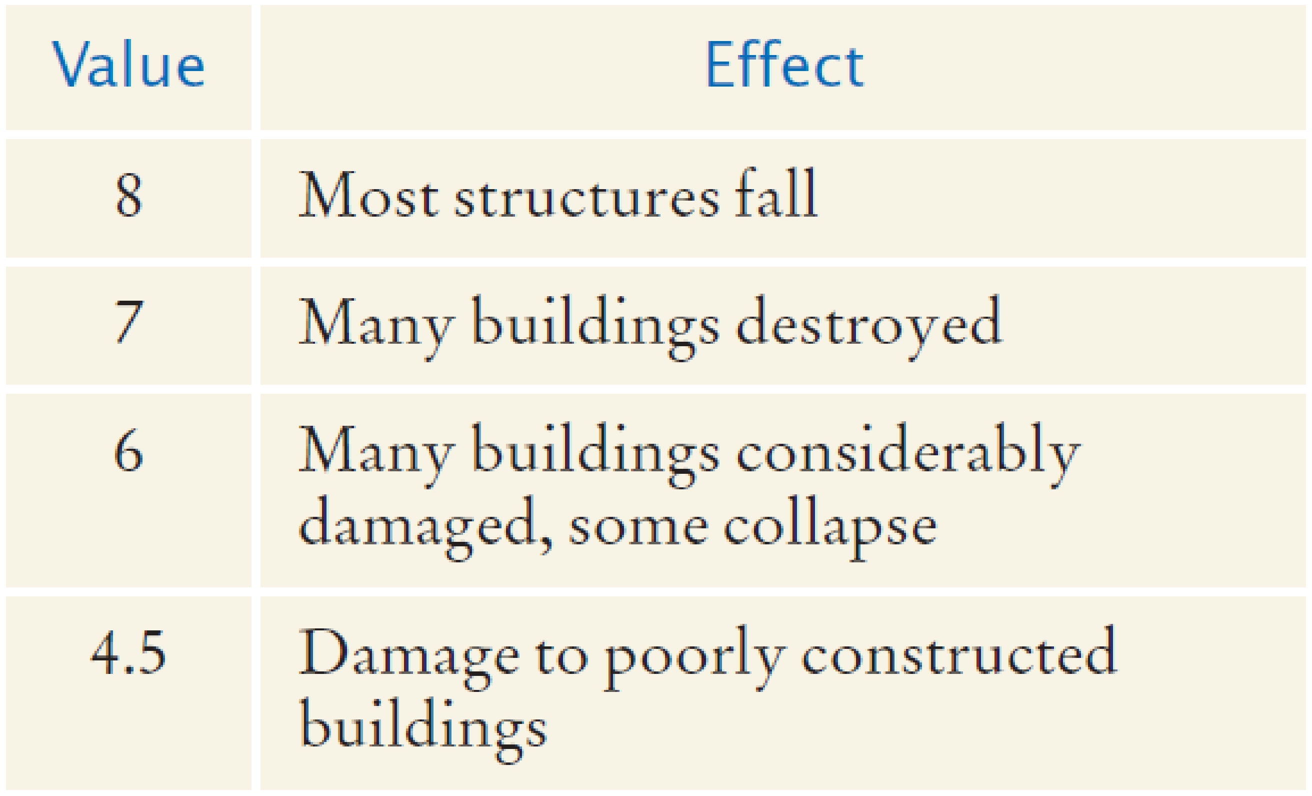
- It is often necessary to include an if statement inside another. Such an arrangement is called a **nested** set of statements.
- Example: Federal Tax Rate Schedule in the US.



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

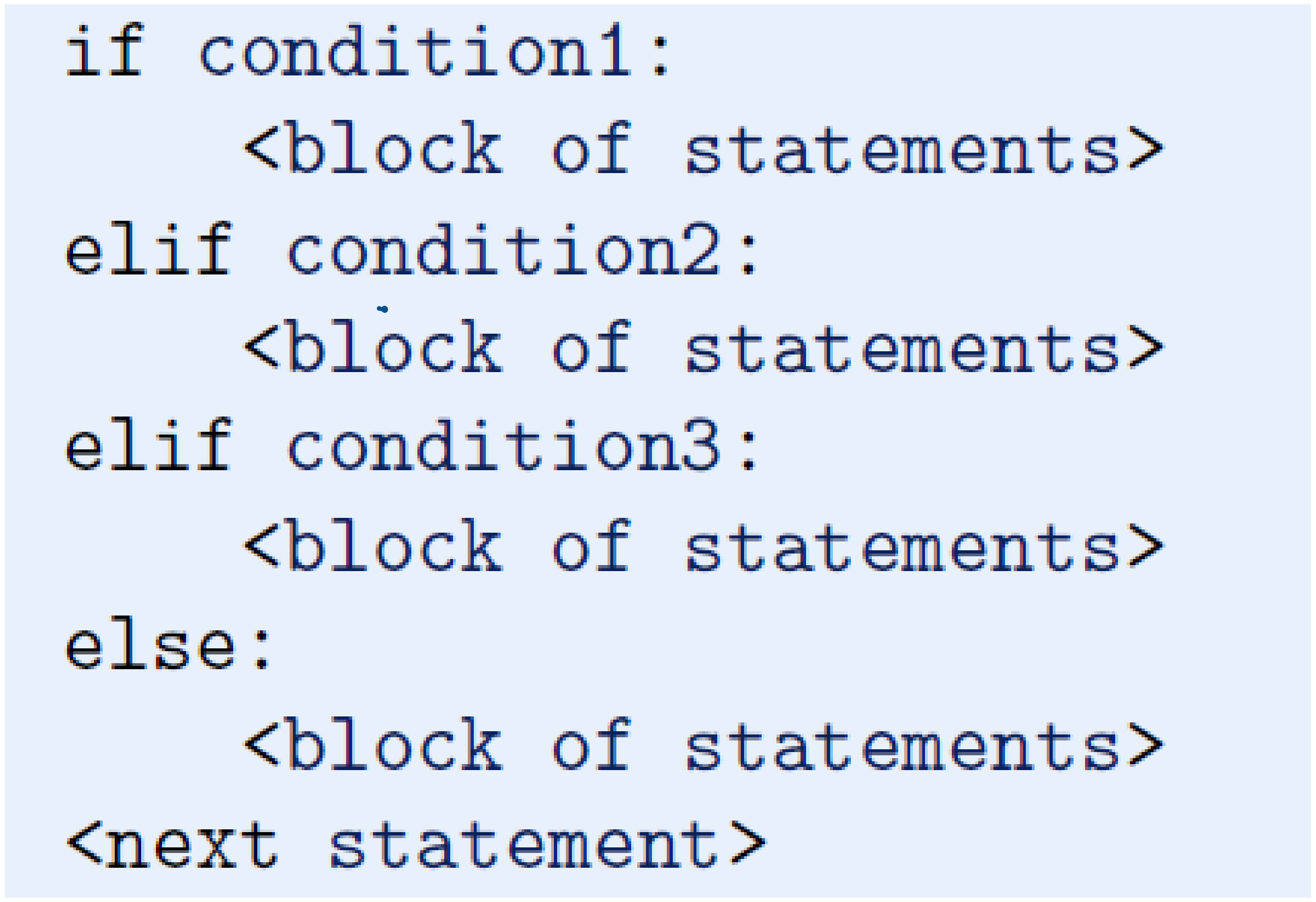
Multiple Alternatives

- Let’s consider a program that displays the effect of an earthquake, as measured by the 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



```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

Boolean Variables and Operators

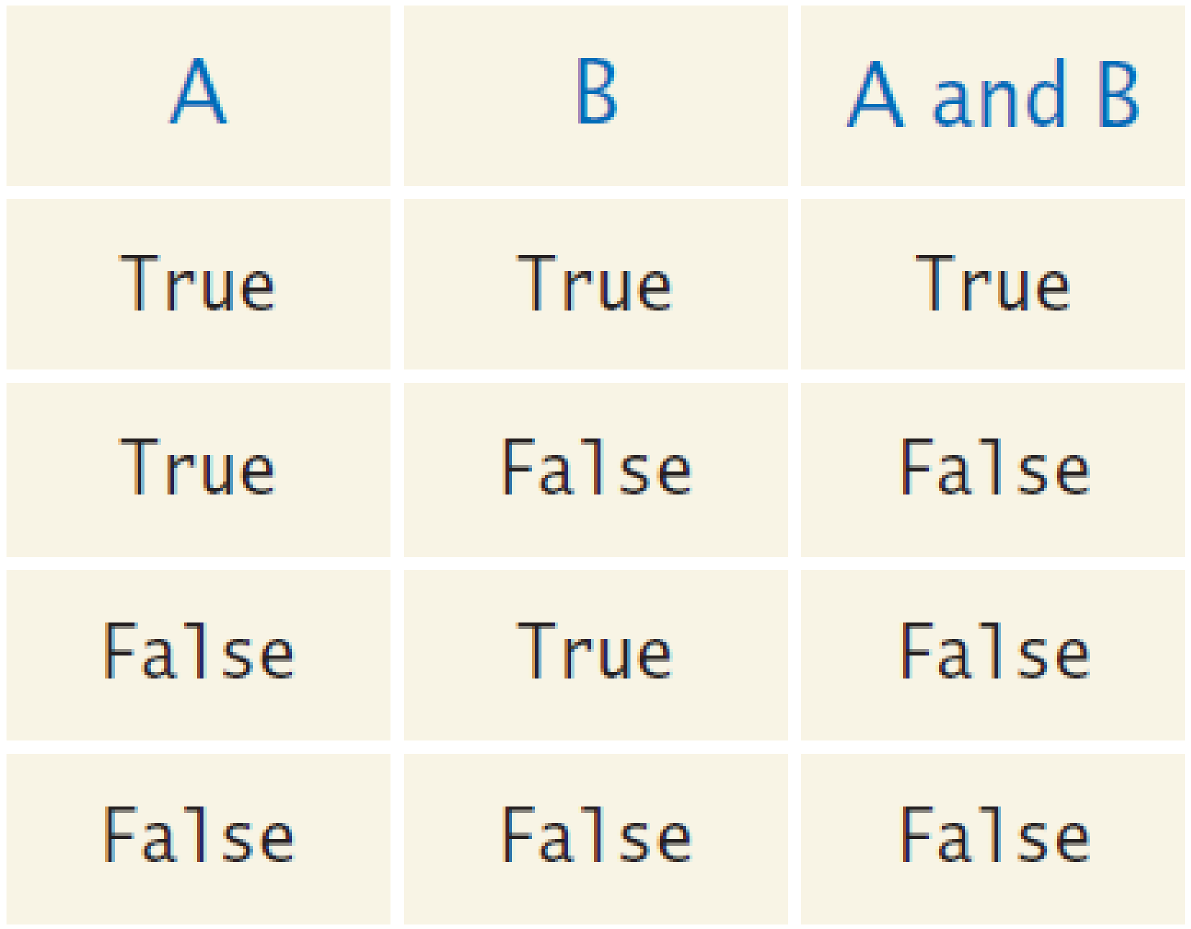
- To store a condition that can be true or false, you use a ***Boolean variable***.

- 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.

The Boolean type bool has two values, False and True.

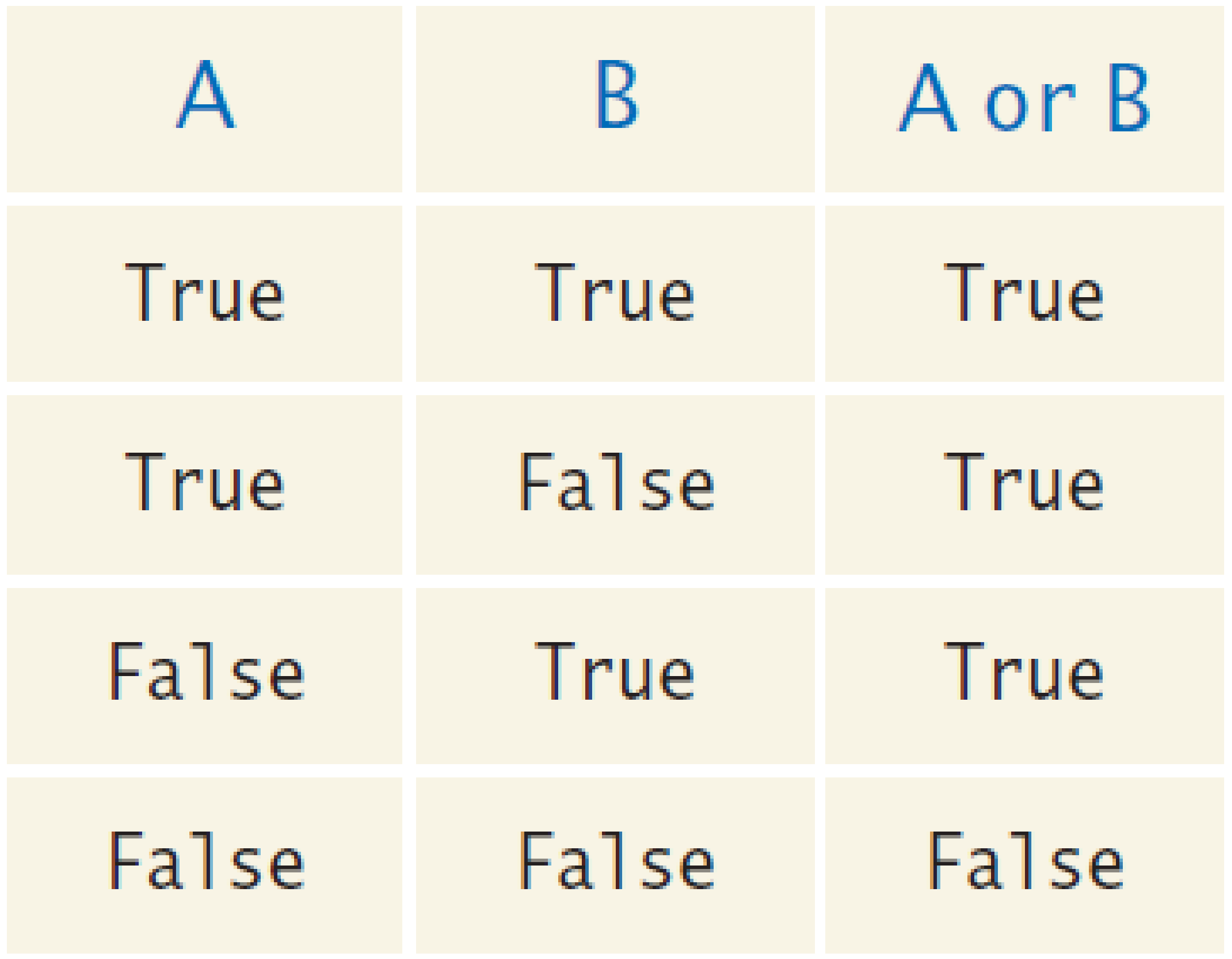
A Boolean variable is also called a flag because it can be either up (true) or down (false).

«and» Operator



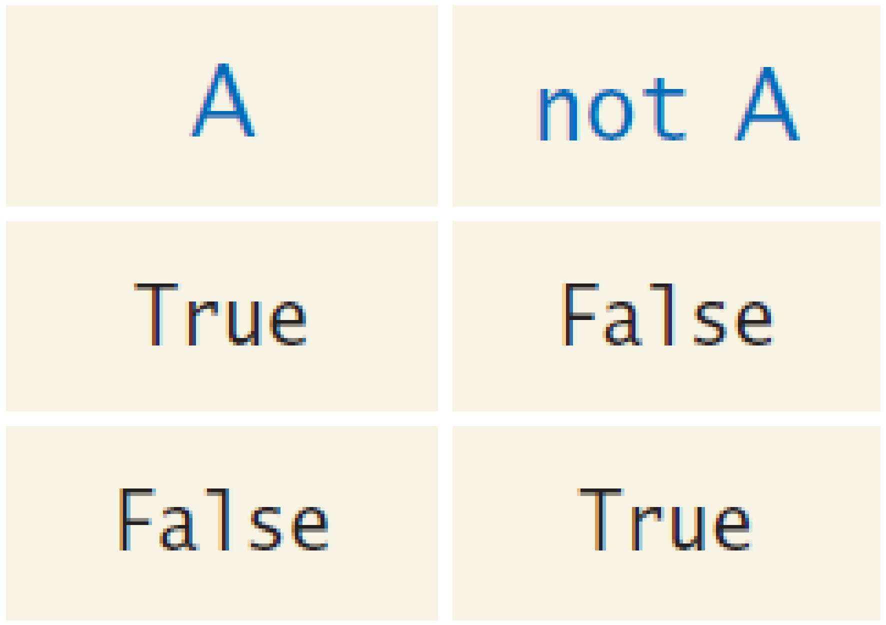
A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

«or» Operator



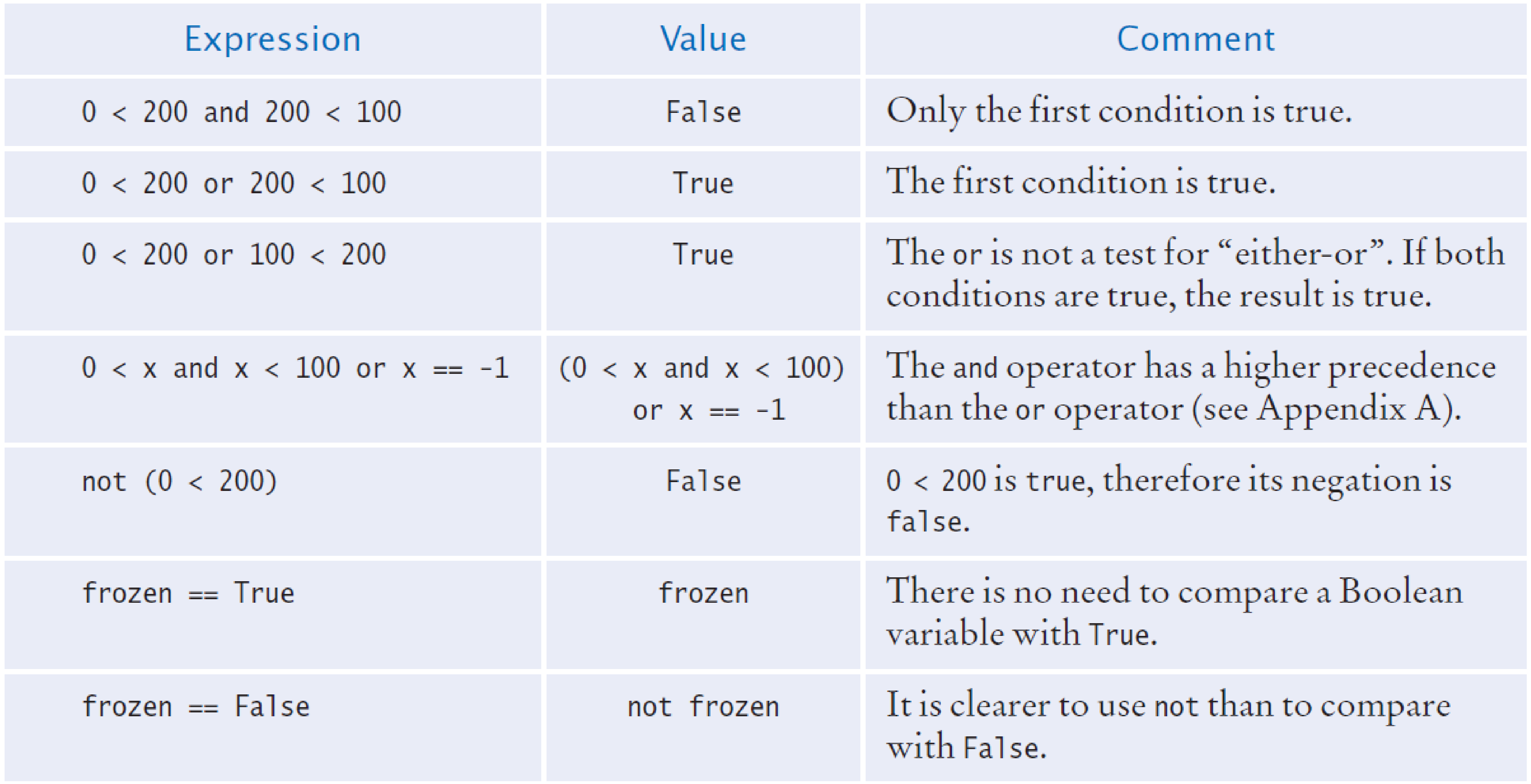
A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

«not» Operator



A	not A
True	False
False	True

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 A).
`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`.

De Morgan’s Law

- De Morgan’s law tells you how to negate **“and”** and **“or”** conditions.

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

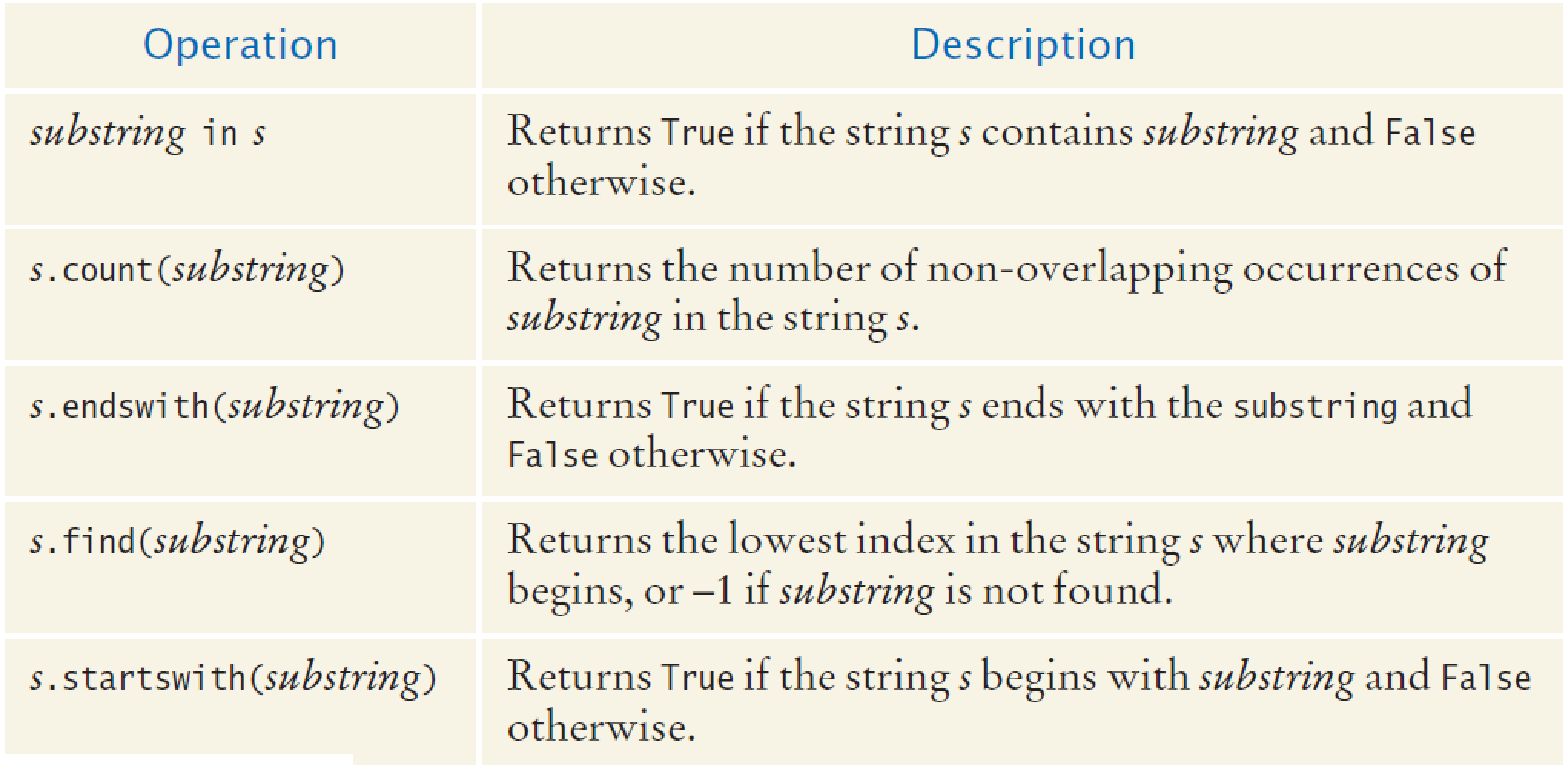
`not (A and B)`	is the same as	`not A or not B`
`not (A or B)`	is the same as	`not A and not B`

```
not (country == "USA") or not (state != "AK") or not (state != "HI")
```

```
country != "USA" or state == "AK" or state == "HI"
```

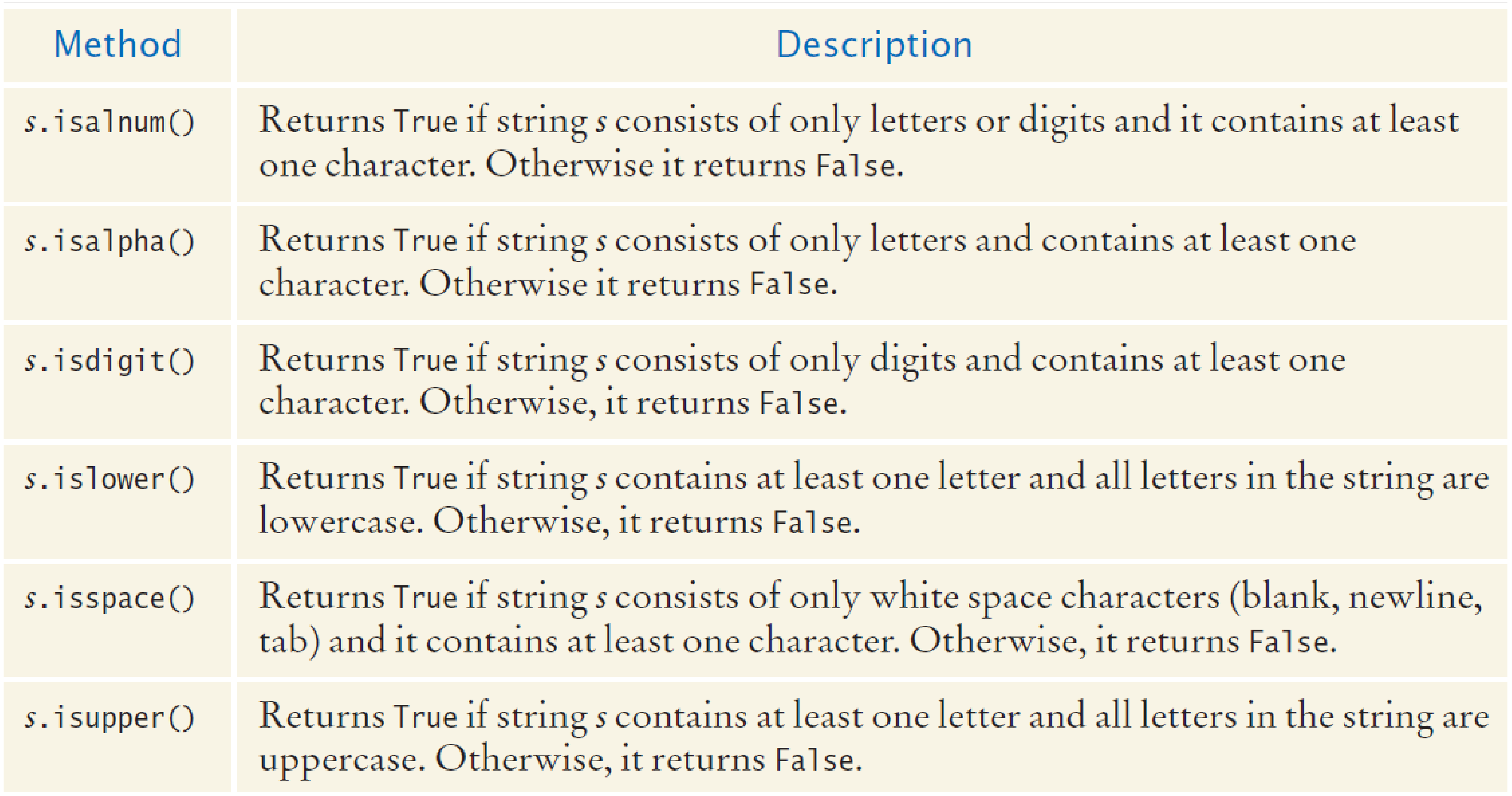
Analyzing Strings

- Sometimes it is necessary to determine if a string contains a given substring. That is, one string contains an exact match of another string.



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 For Testing Strings 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.

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.

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.

Like a quality control worker, you want to make sure that user input is correct before processing it.

End of Chapter 3