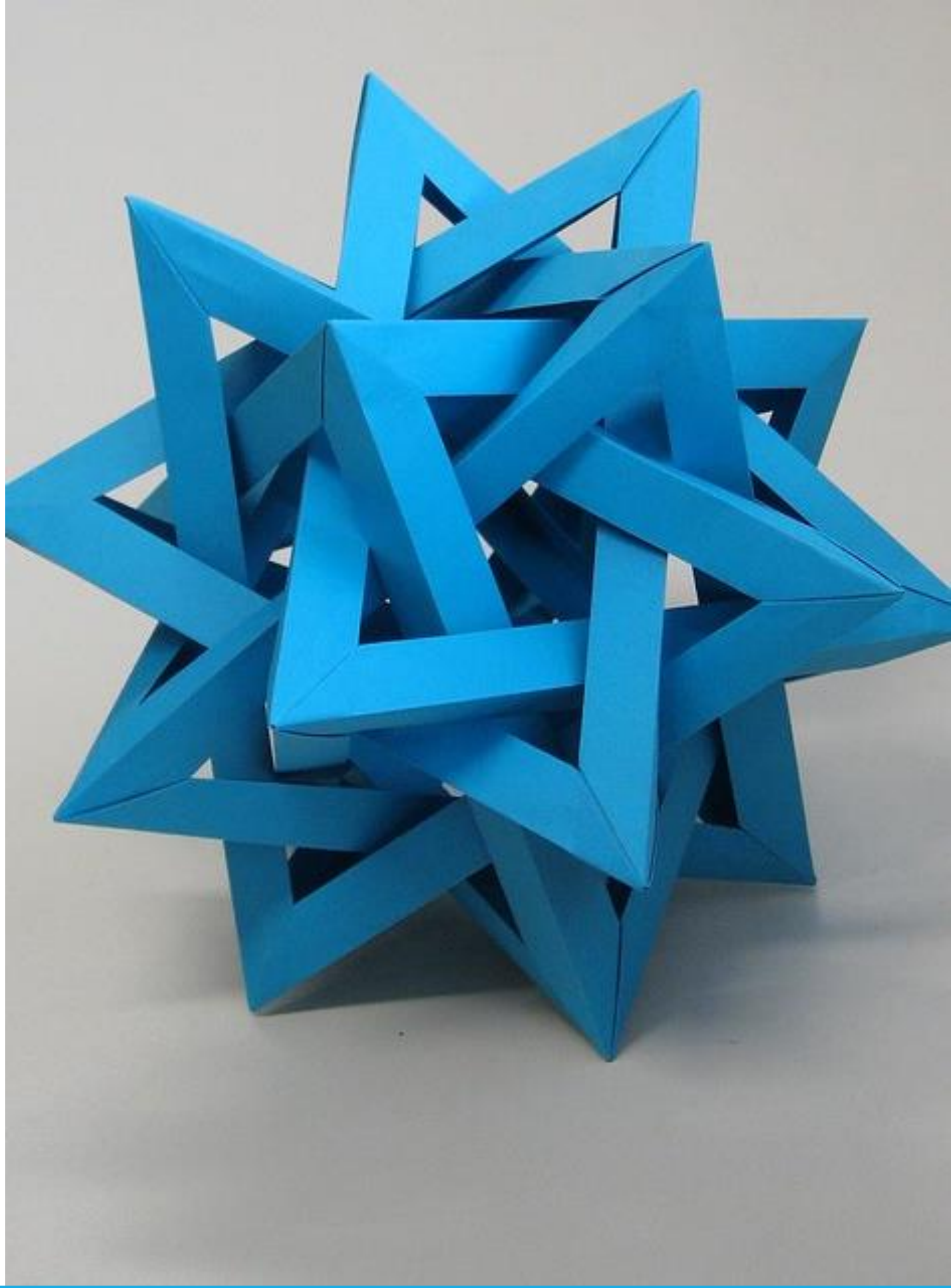
POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica

Unit P3: Decisions

DECISIONS, BOOLEAN CONDITIONS, STRING ANALYSIS, AND INPUT VALIDATION

📘 Chapter 3



This Photo by Unknown Author is licensed under CC BY-SA

Unit Goals

- Implement decisions using the **if statement**
- Compare Numbers (integer and floating-point) and strings
- Boolean data
- Validating user input
- Formatting the output

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

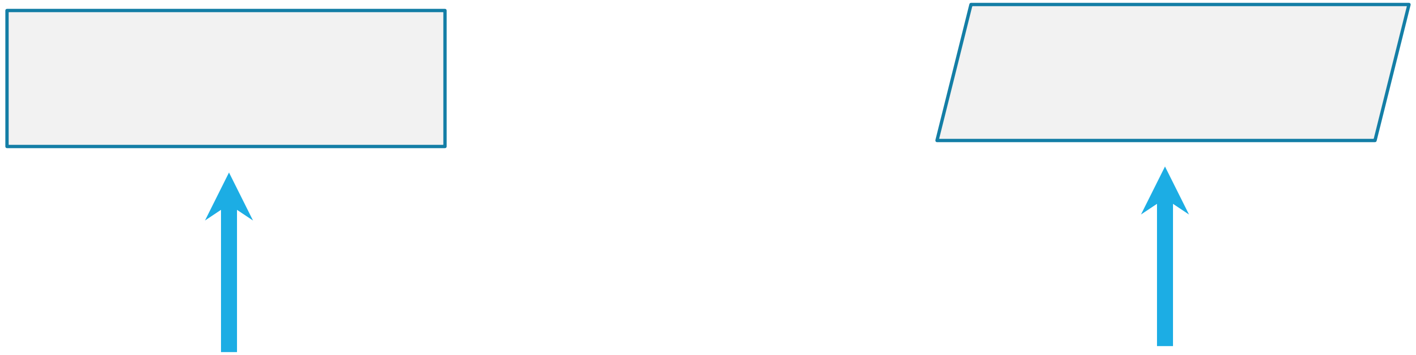
- The `if` Statement
- Relational Operators
- Nested Branches
- Multiple Alternatives
- Boolean Variables and Operators
- Analyzing Strings
- Data Input and Formatted Output
- Application of decisions: Input Validation

The `if` statement

📘 3.1

Summary…

- **Up to know, we’re able to write Python programs that are equivalent to “linear” flowcharts.**
  - Only including these two blocks:



Assignments, arithmetic, etc.

```
x = 2
str = ‘ciao’
```

Etc.

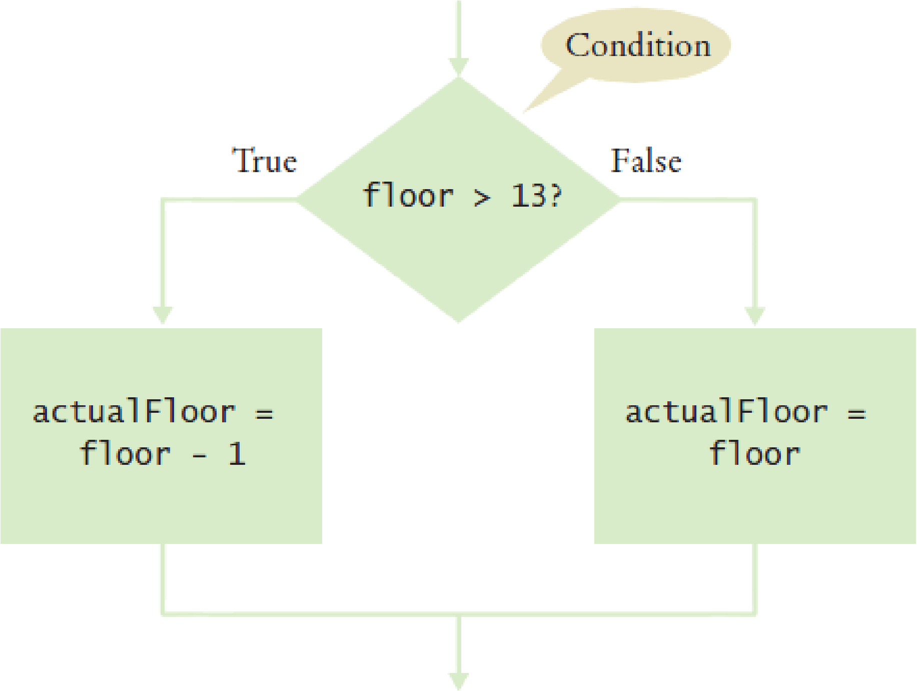
Input/Output

```
print(x)
x = int(input(‘valore?’))
```

Etc.

The `if` Statement

- Implements the following flowchart structure (that we've already seen many times):
  - ○ **True** (`if`) branch or **False** (`else`) branch

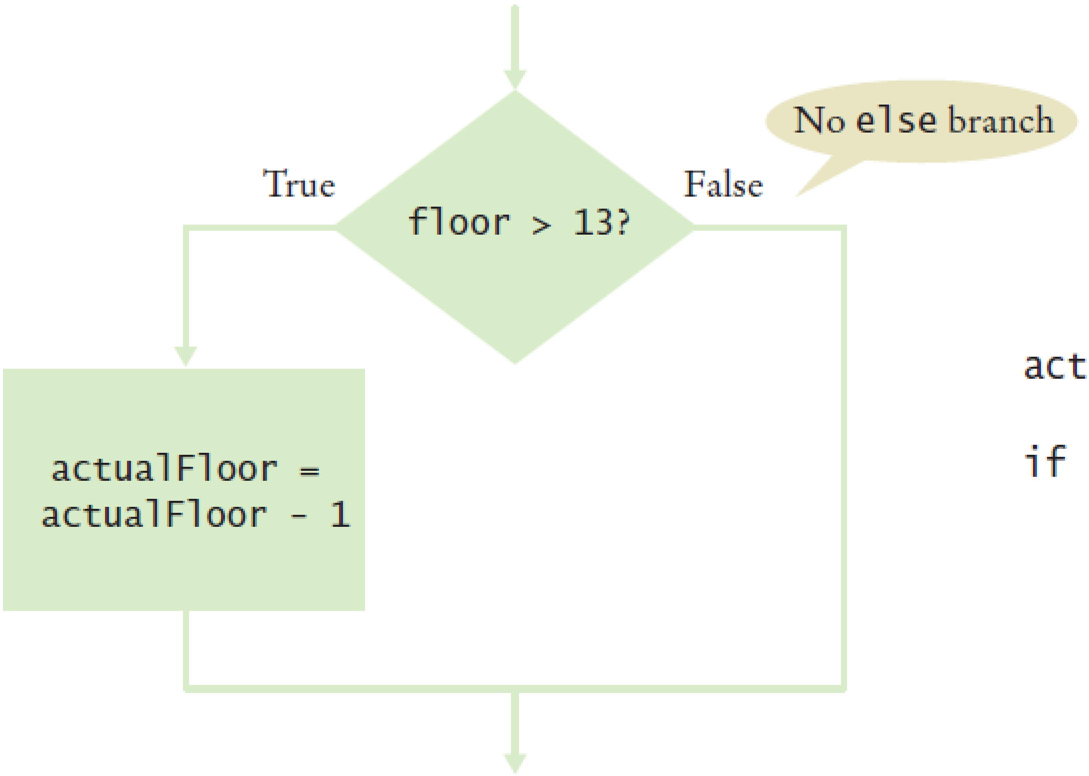


```
actualFloor = 0

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)*

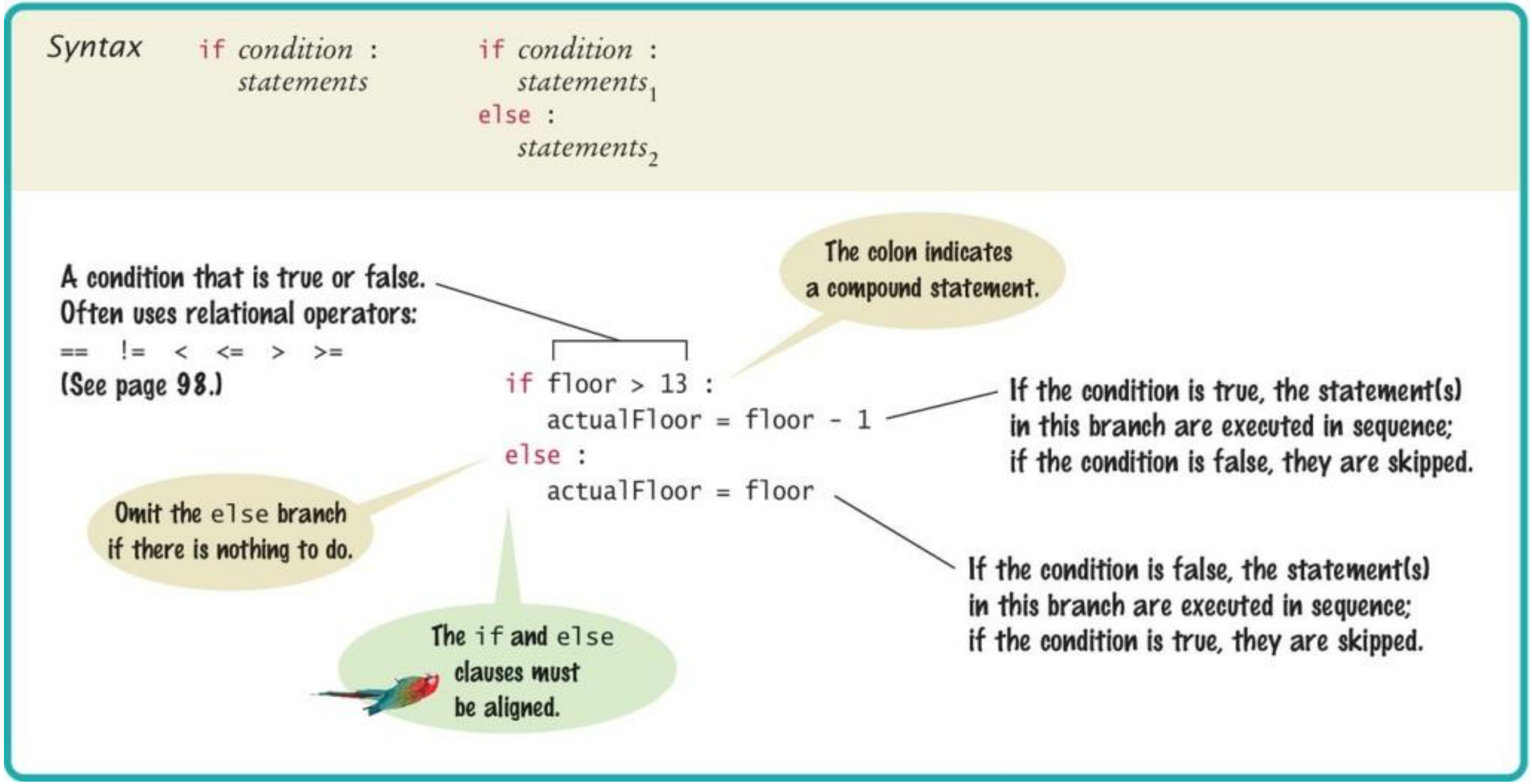
Flowchart with only a True Branch

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



```
actualFloor = floor

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

Syntax 3.1: 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.

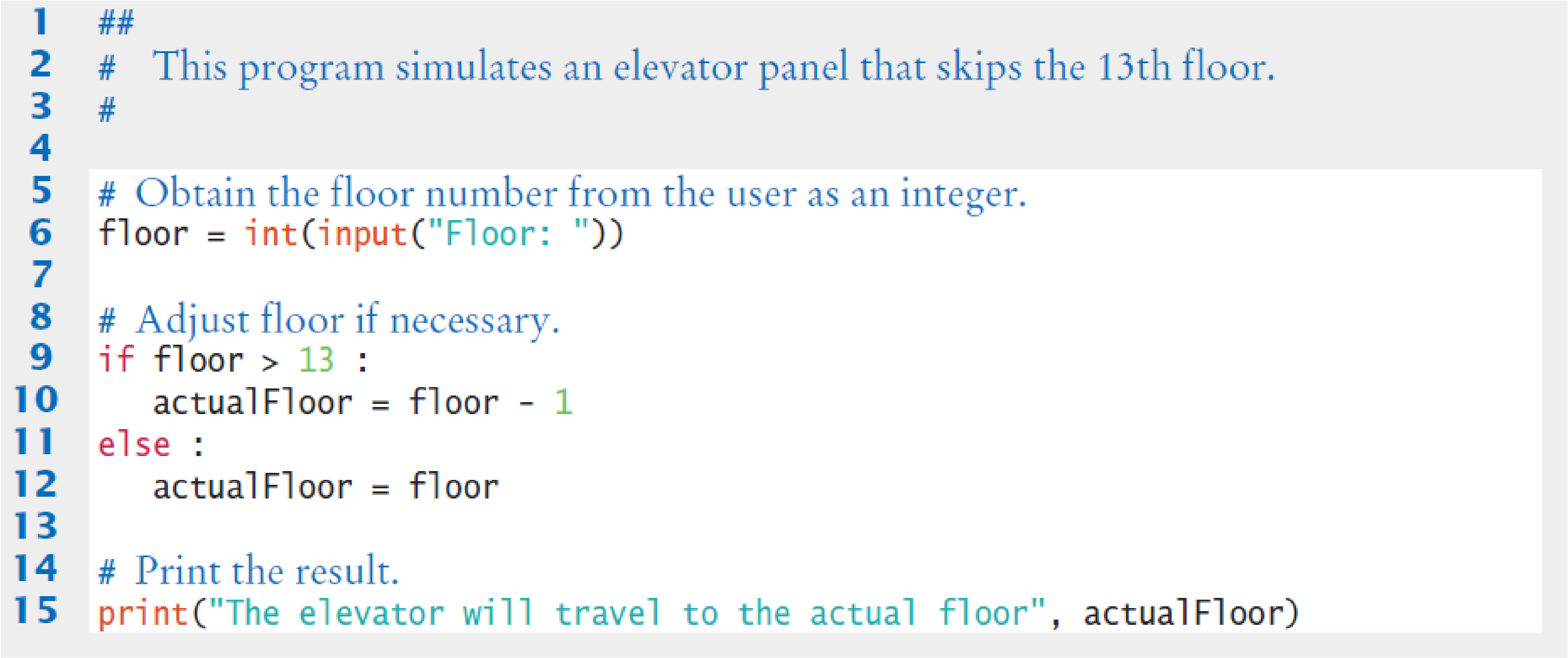
If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

Omit the else branch if there is nothing to do.

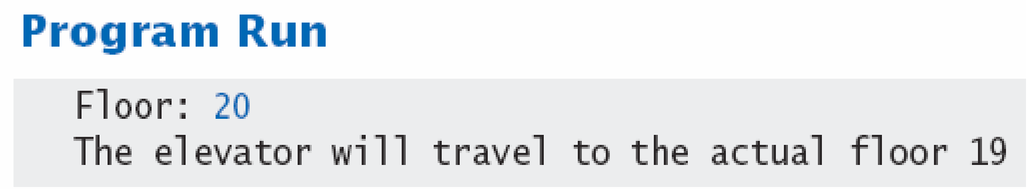
If the condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

The if and else clauses must be aligned.

Elevatorsim.py



```
##
#   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 result.
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 than 13
    - What is the result?
  - Run the program again with a value that is greater than 13
    - What is the result?
- What happens if you enter 13?

Example 1 - corrected

- Revised Problem Statement (1):
  - Check the input entered by the user:
  - If the input is 13, print an error message “There’s no floor 13”
- 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 17 is also considered unlucky.
  - What is the revised algorithm?
  - Modify the elevatorsim program to “skip” both the 13th and 17th floor

Compound Statements

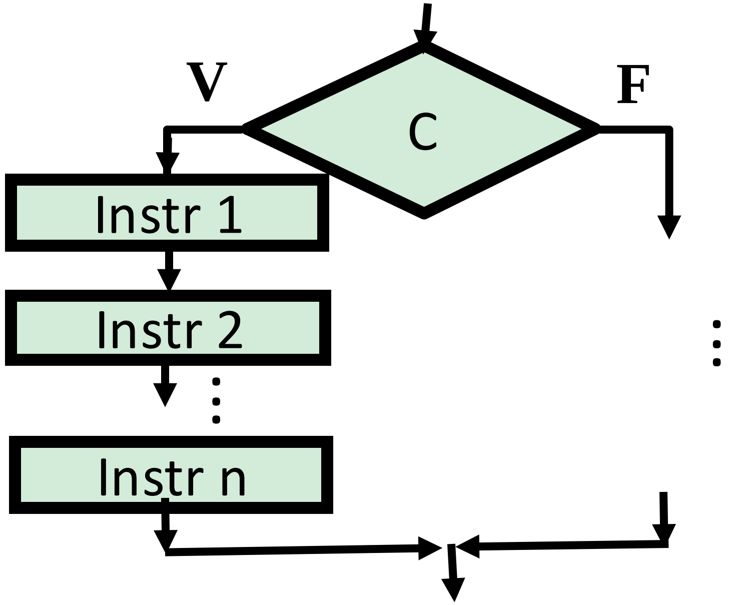
- The if statement is an example of **compound statement**.
- Compound statements span multiple lines and consist of a header and a statement block
- Compound statements require a colon ":" at the end of the header.

compound statement

```
if <condition>:
    <instr 1>
    <instr 2>
    …
    <instr n>
else:
    …
```

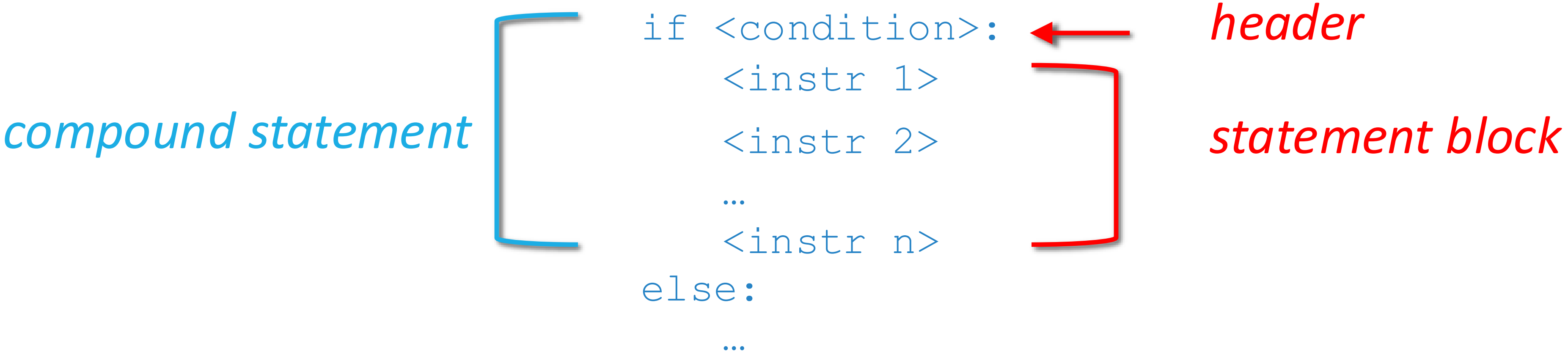
header

Statement block



Compound Statements

- The statement block is a group of one or more statements, all with the same indentation
- The statement block
  - starts on the line after the header
  - ends at the first statement that is less indented
- Most IDEs automatically indent the statement block.



```
if <condition>:
    <instr 1>
    <instr 2>
    …
    <instr n>
else:
    …
```

Compound Statements

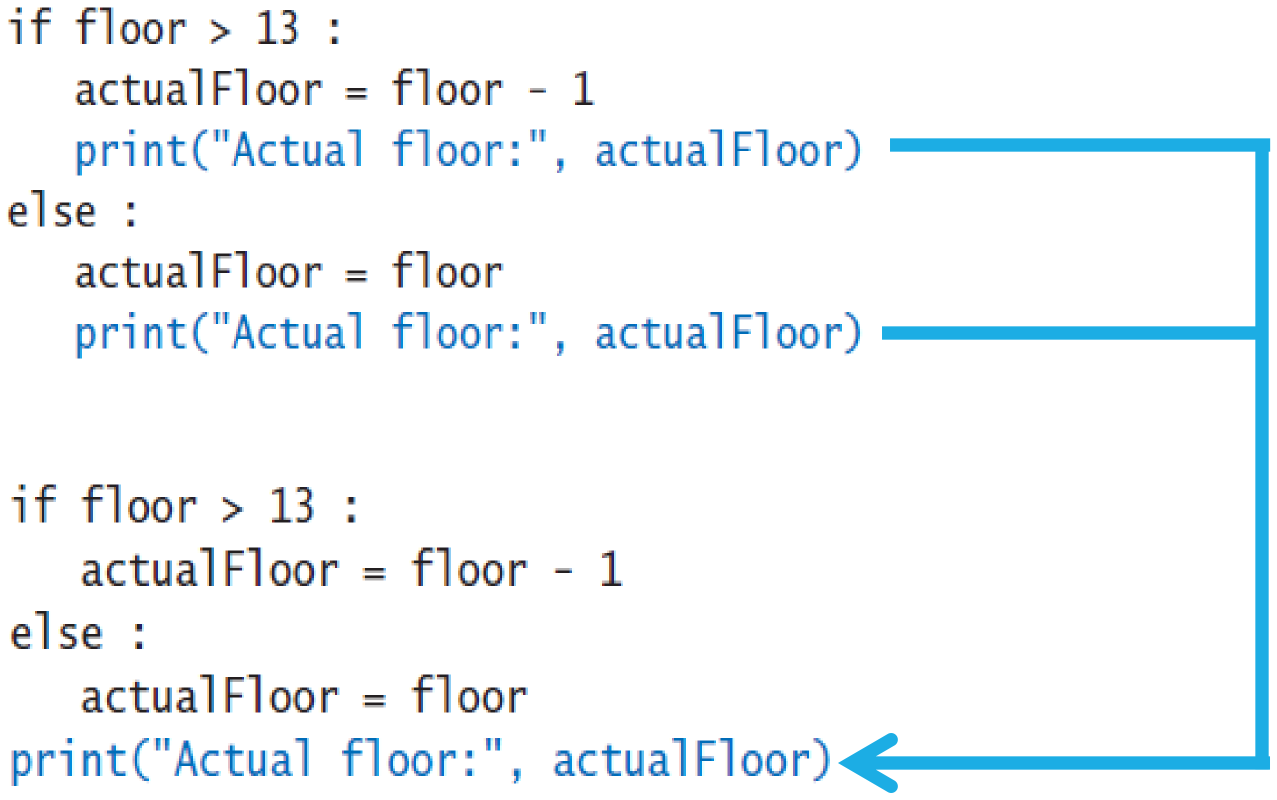
- Statement blocks can be **nested** inside the blocks of other compound statements (of the same or other block type)

- 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 intended also to provide programmers a visual cue 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.



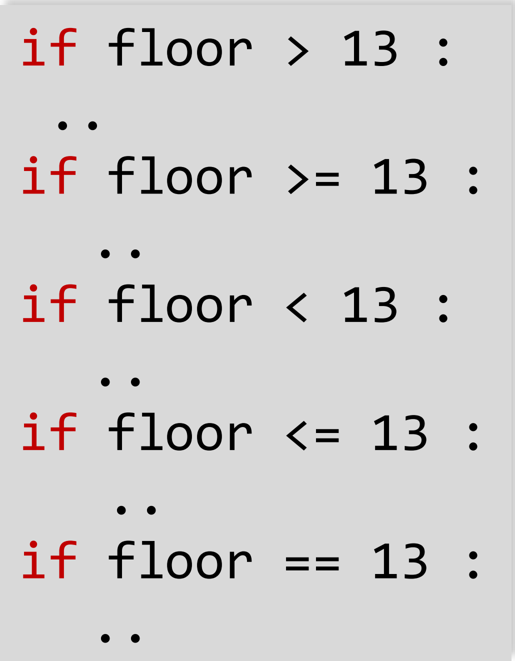
```
if floor > 13 :
   actualFloor = floor - 1
   print("Actual floor:", actualFloor)
else :
   actualFloor = floor
   print("Actual floor:", actualFloor)
```

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

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 :
    ..
```

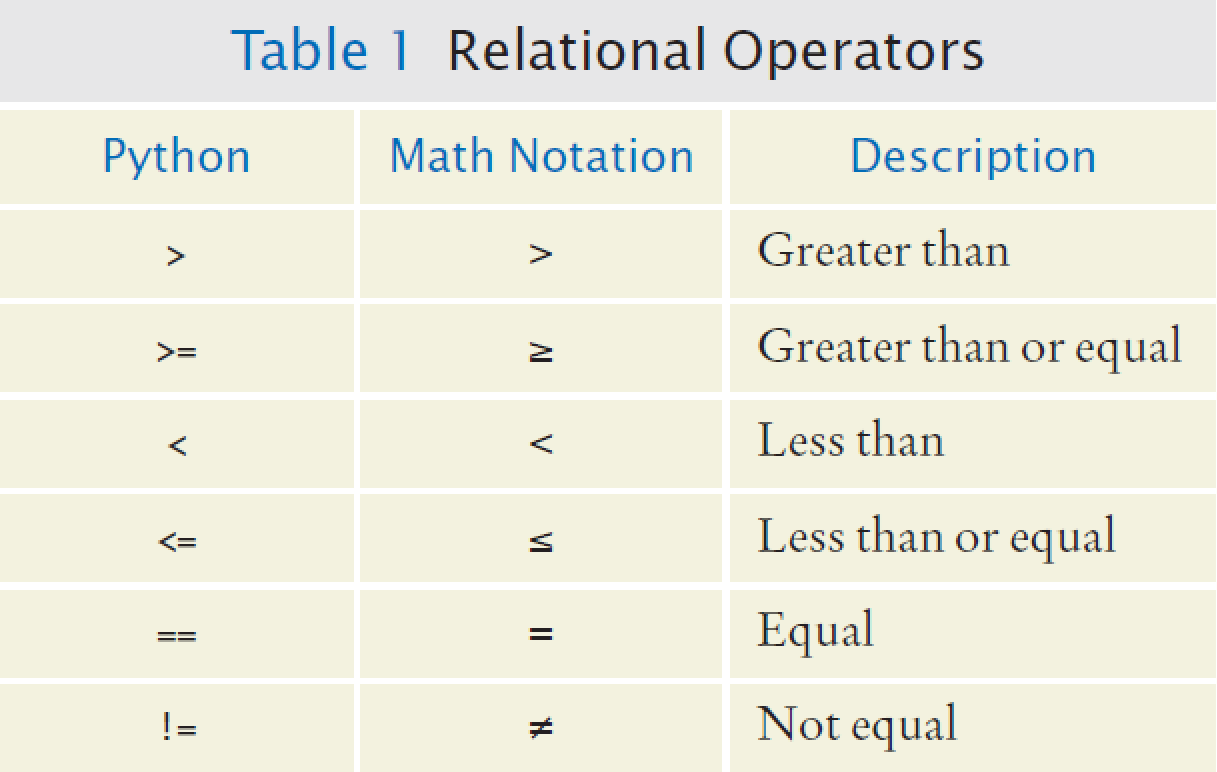


Table 1 Relational Operators

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

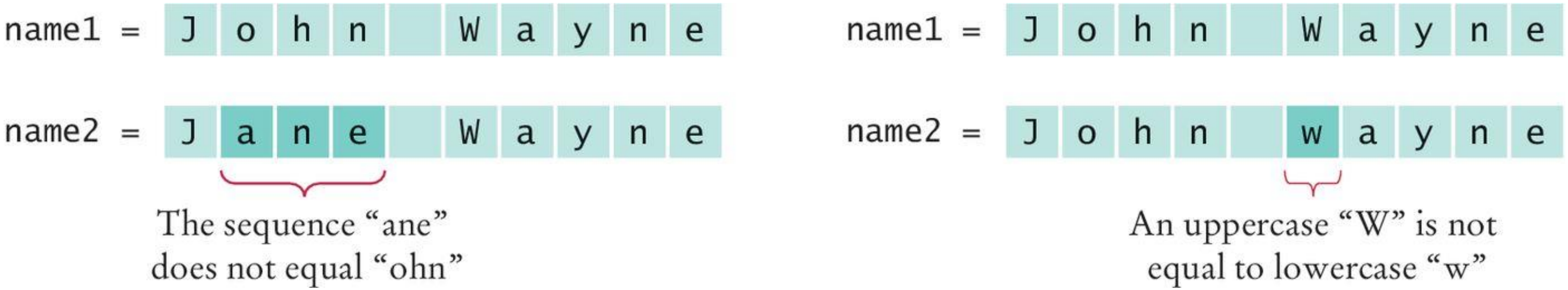
```
if name1 == name2 :
   print("The strings are identical")
```

- Checking if two strings are not equal

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

Checking for String Equality

- Two strings are equal if they contain the same characters, in the same order
- If any 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
name1 = J o h n W a y n e
name2 = J o h n w a y n e

The sequence “ane” does not equal “ohn”

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

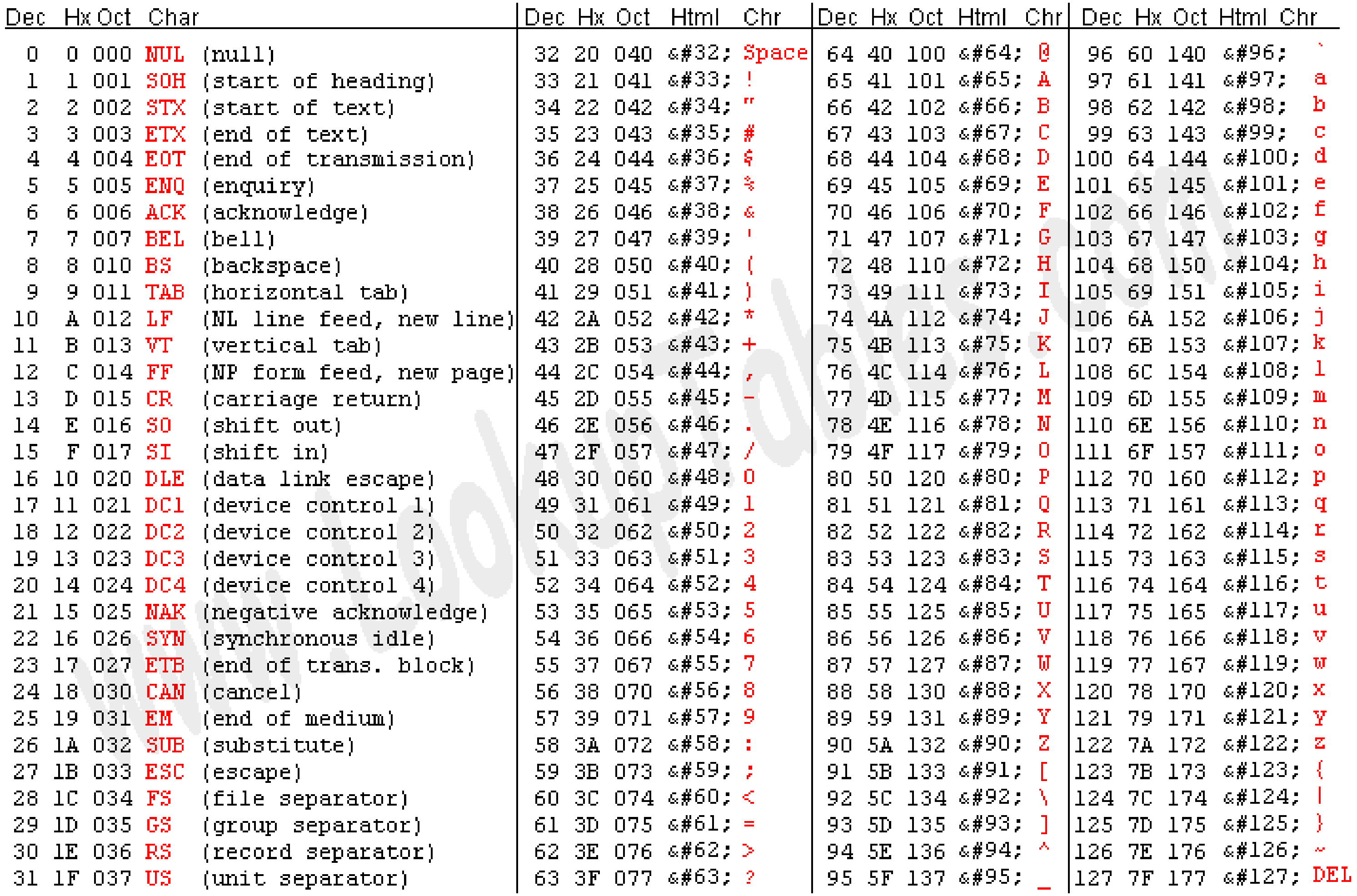
Lexicographical Order

- We can compare Strings in **‘dictionary’ like** order:
  - `string1 < string2`
  - `True if string1 comes before string2 in the dictionary`

- 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

- **Why this order?**
  - The order is ruled by the Basic Latin (ASCII) Subset of Unicode
    - Accented characters are not always logical

Link with ASCII Codes



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

For Unicode characters see: https://unicode-table.com/

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")
```

Relational Operator Examples and Errors

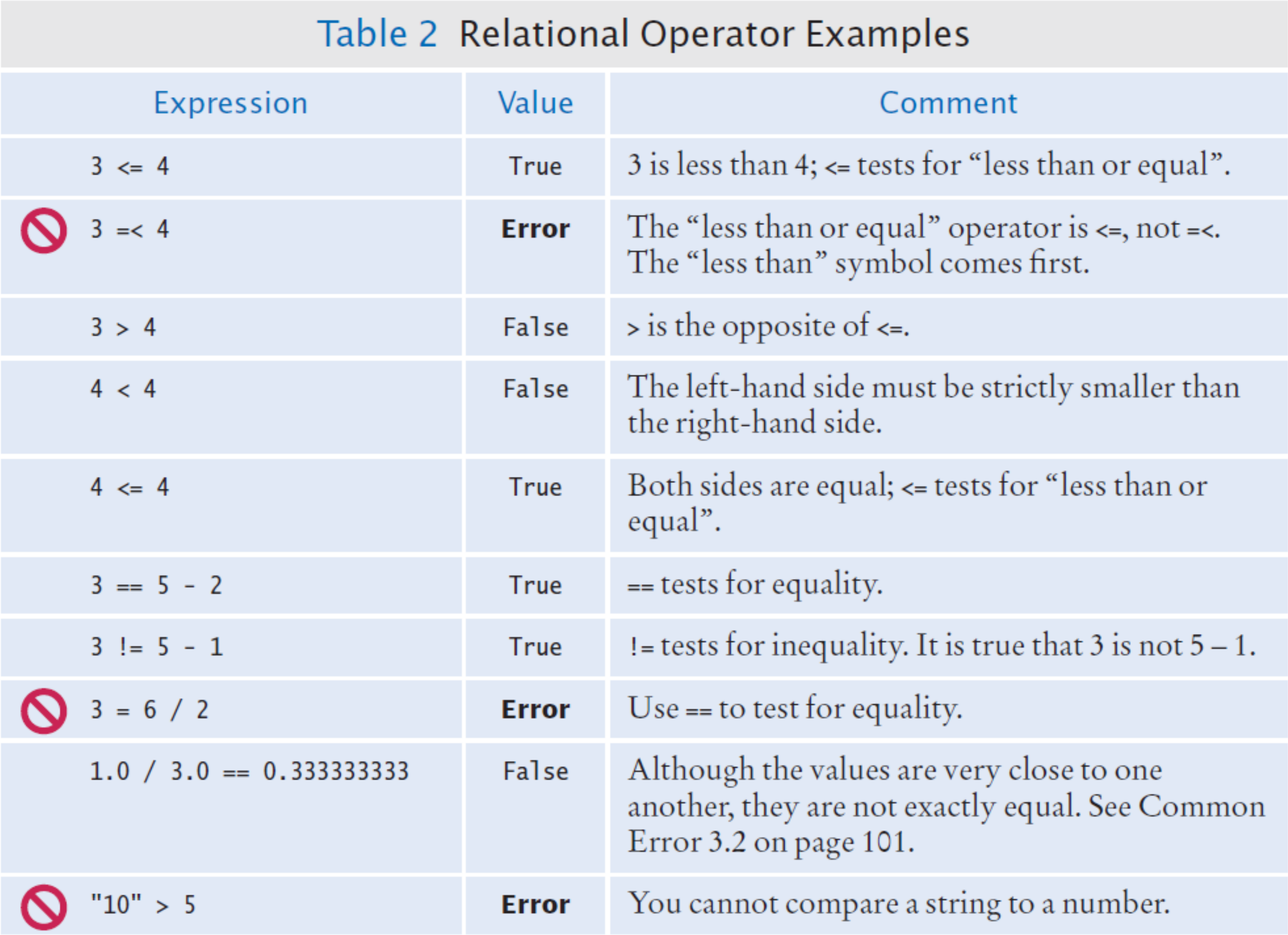


Table 2 Relational Operator Examples

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.

Operator Precedence

- The comparison operators have lower precedence than arithmetic operators
  - **Calculations are done before the comparison**

Calculations

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

Example

The Sale Example

- The university bookstore has a sale, 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)

- 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

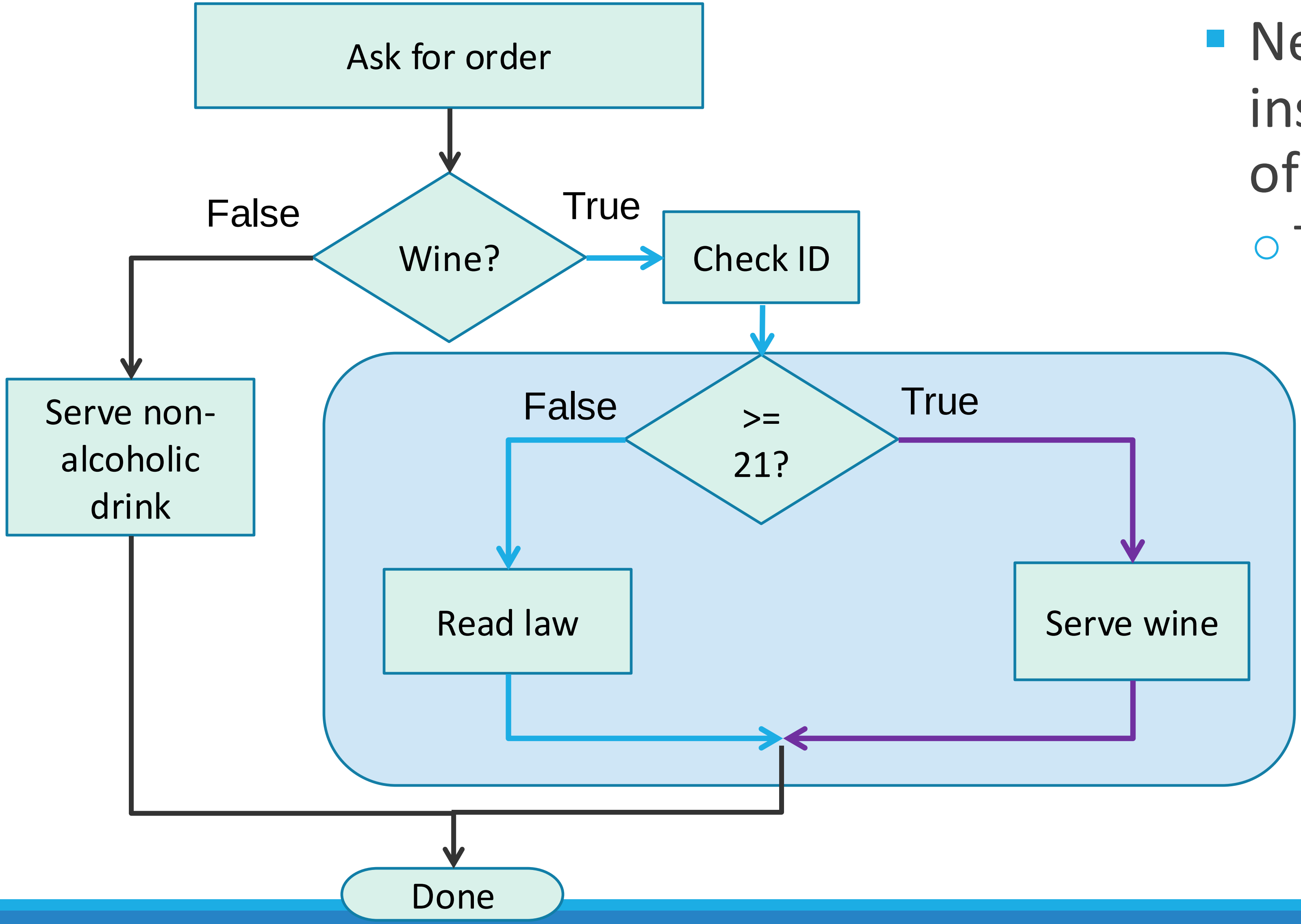
Nested Branches

- You can nest an **if** inside either branch of another **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
```

nested IF

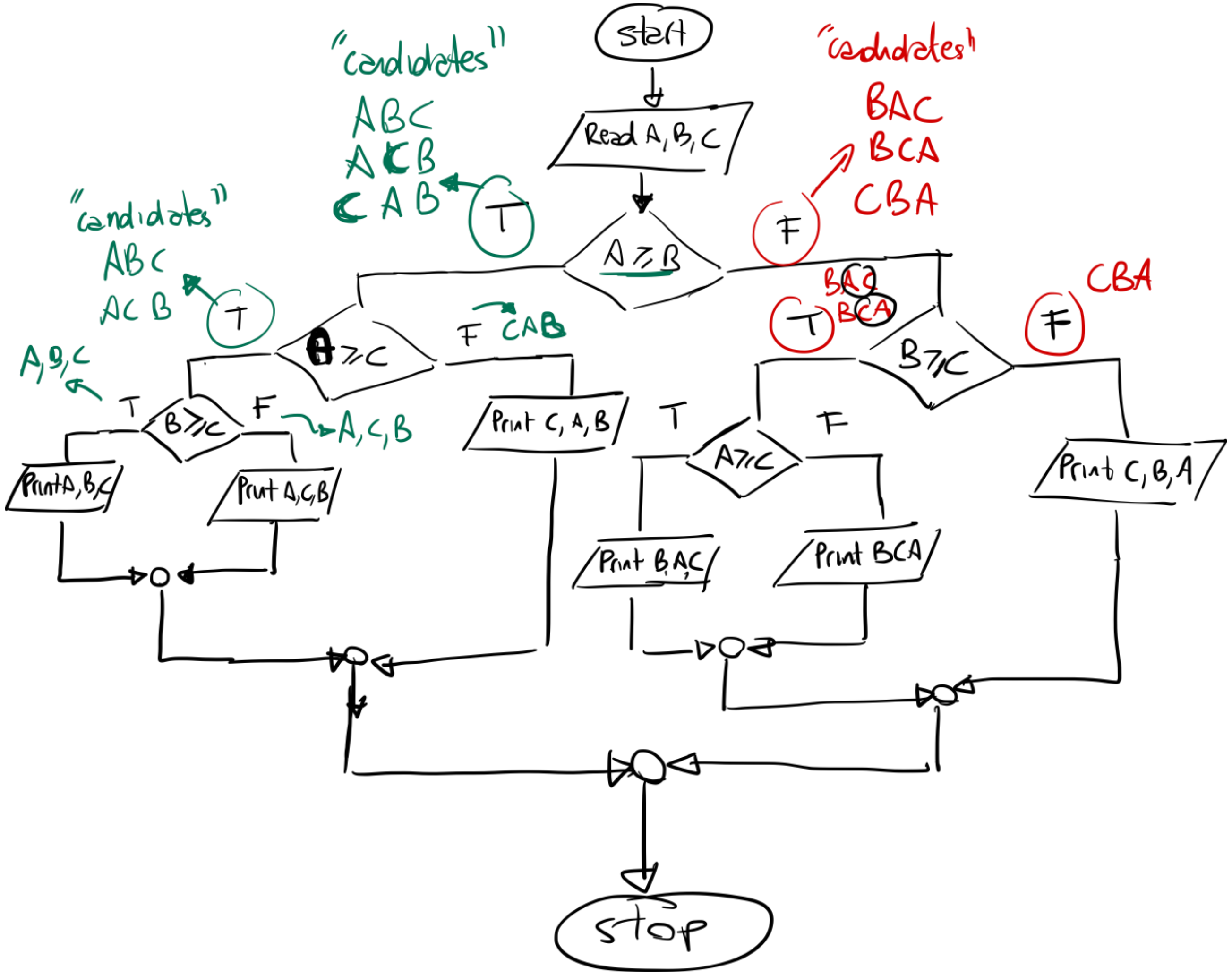
Flowchart of a Nested if



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

Example: from Flowchart Exercises

- Order 3 values A, B and C.



Tax Example: nested ifs

- Four outcomes (branches)

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

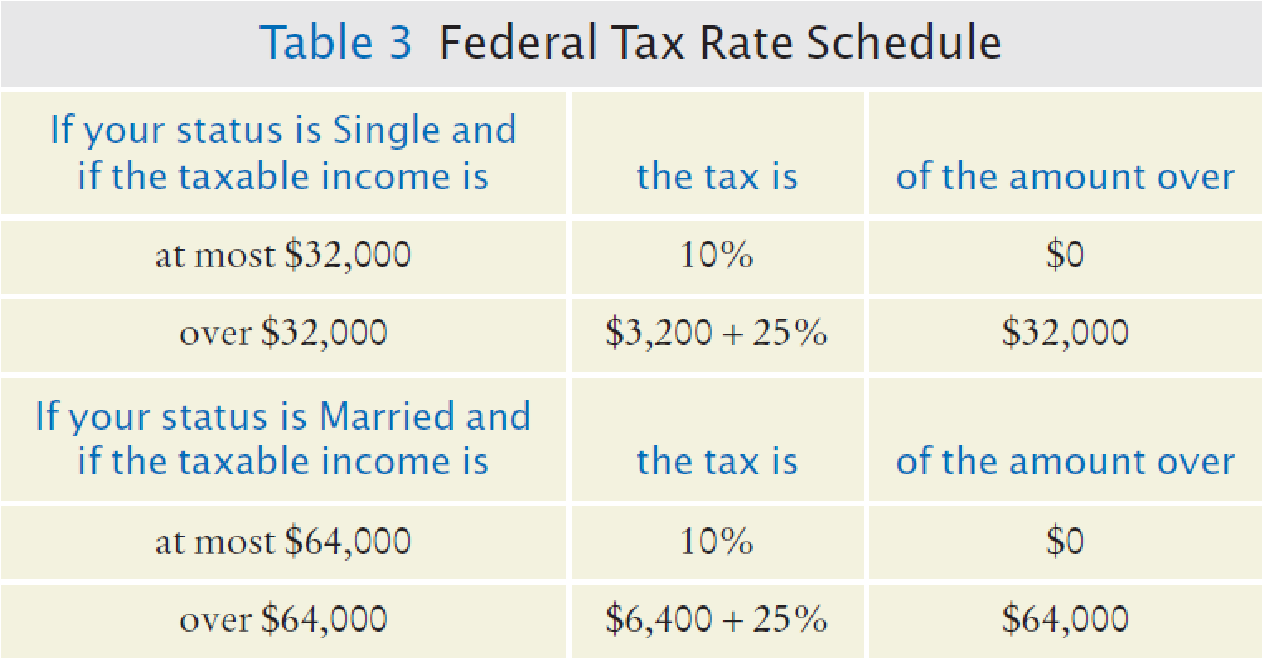
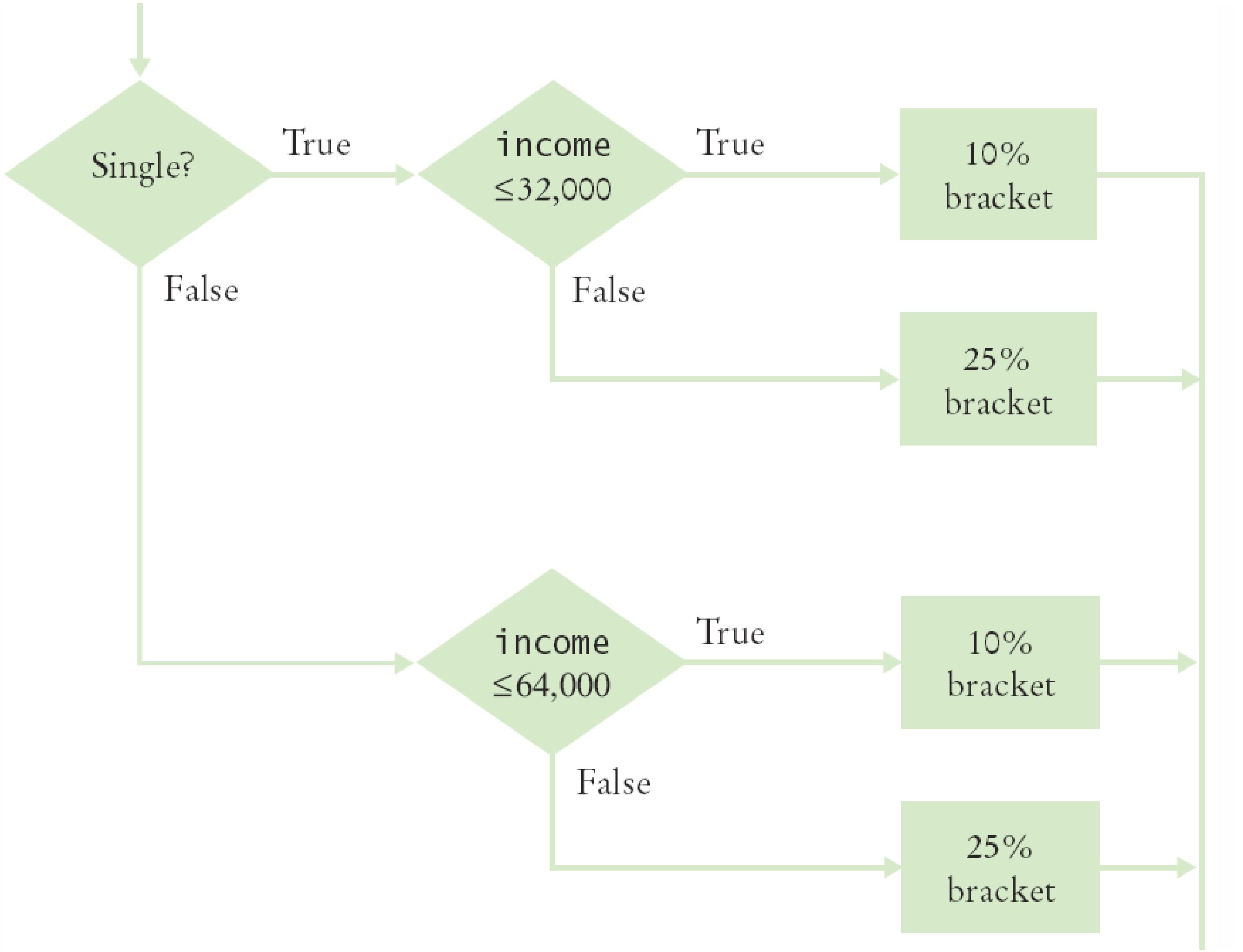


Table 3 Federal Tax Rate Schedule

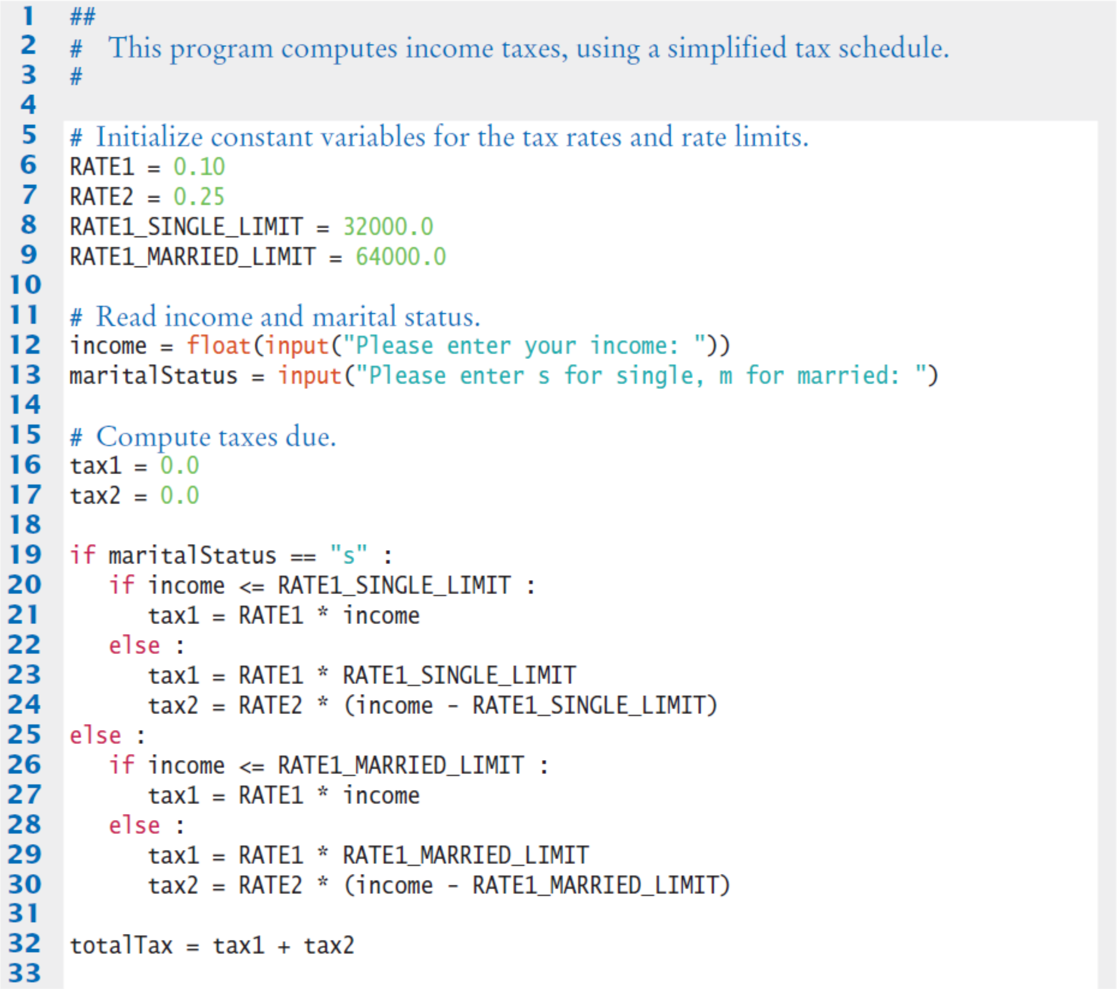
If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

Flowchart for the Tax Example

- Four branches



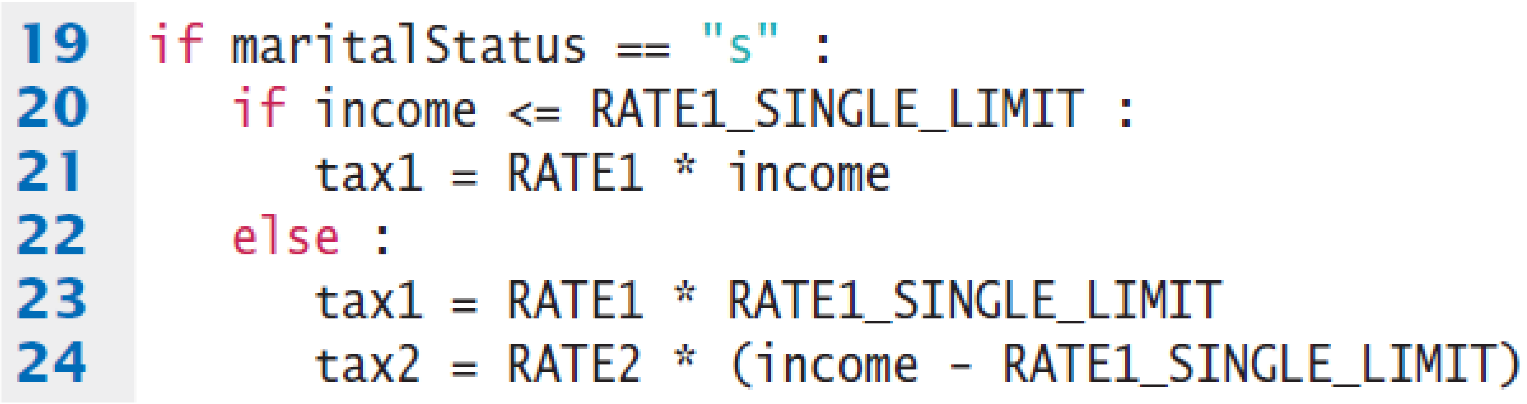
Taxes.py (1)



```
##
#  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 = 0.0
tax2 = 0.0

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
```

Taxes.py (2)

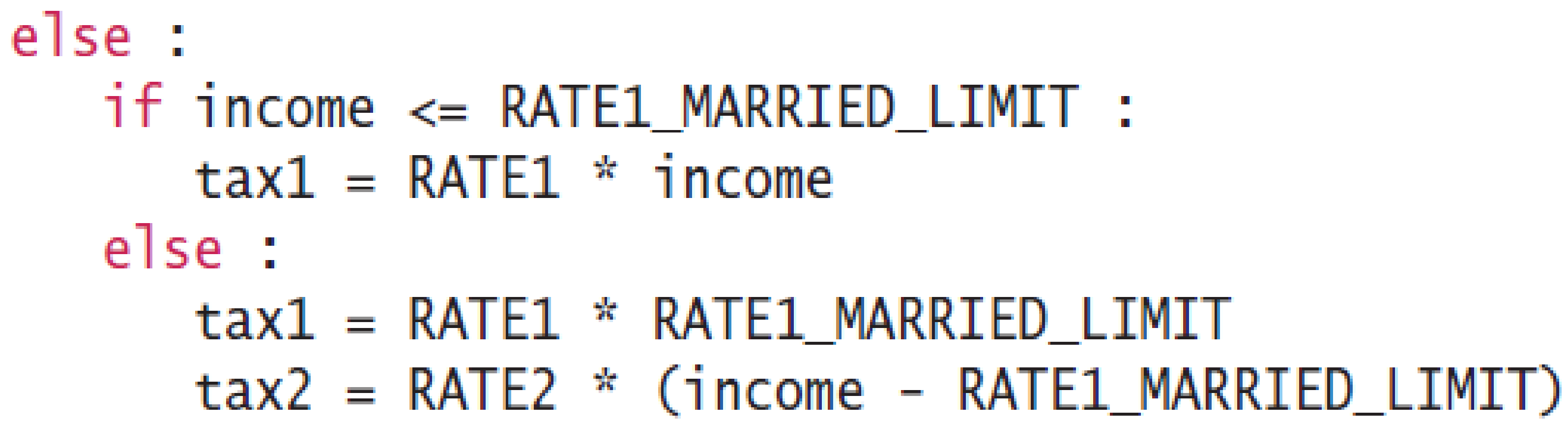
- The ‘True’ branch (Single)
  - Two branches within this branch



```
if maritalStatus == "s" :
    if income <= RATE1_SINGLE_LIMIT :
        tax1 = RATE1 * income
    else :
        tax1 = RATE1 * RATE1_SINGLE_LIMIT
        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

- 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?

Multiple Alternatives

📘 3.4

Multiple Alternatives

- What if you have more than two branches?
- Example: determine the effects of an earthquake based on its Richter Scale intensity:
  - 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.

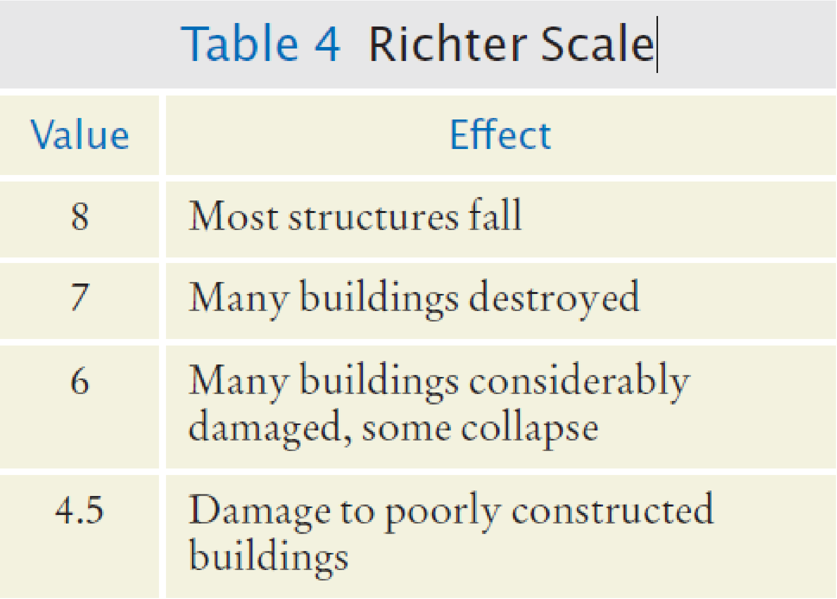
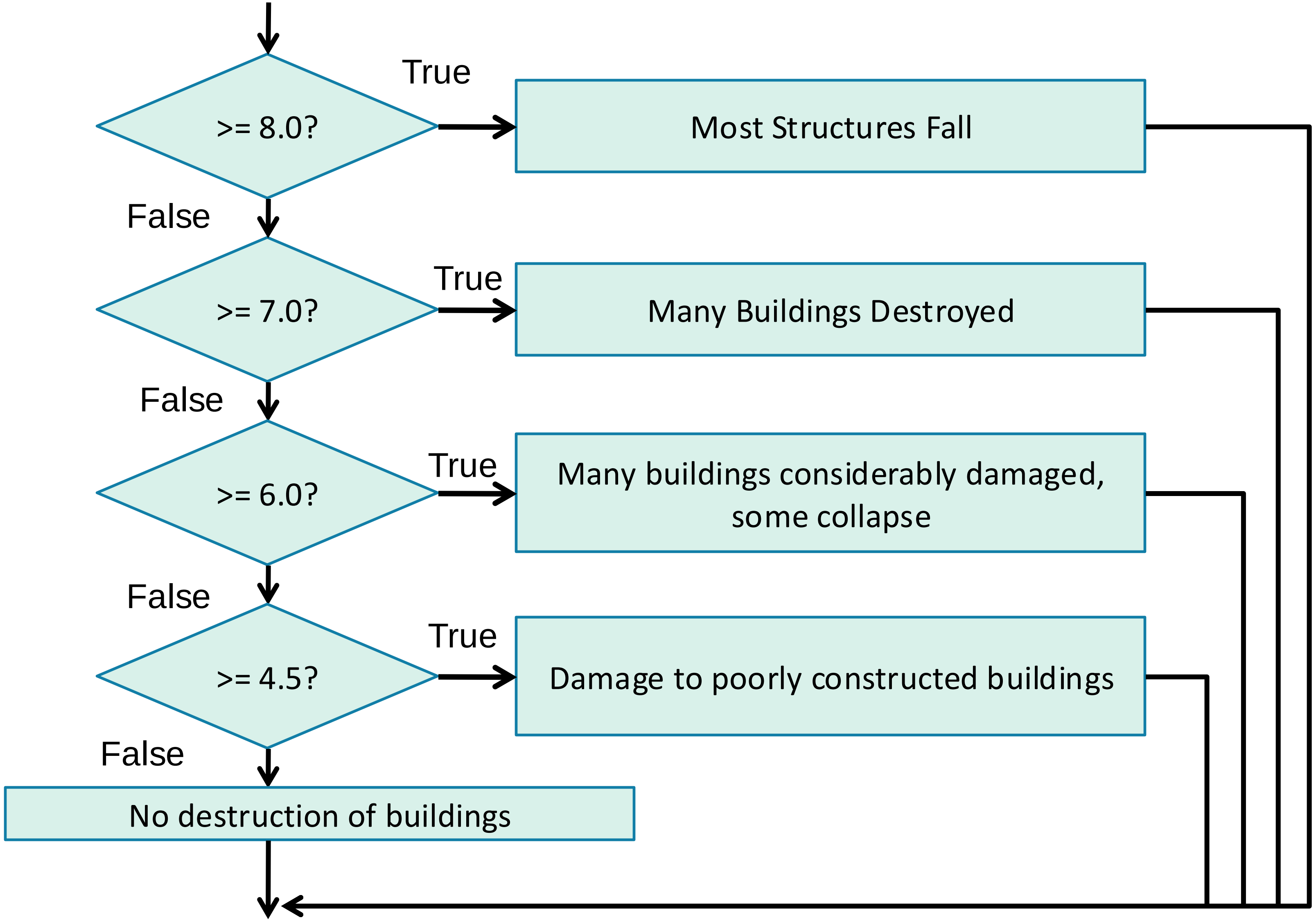


Table 4 Richter Scale

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


>= 8.0?
True
Most Structures Fall
False
>= 7.0?
True
Many Buildings Destroyed
False
>= 6.0?
True
Many buildings considerably damaged, some collapse
False
>= 4.5?
True
Damage to poorly constructed buildings
False
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")
```

Some values will enter multiple branches (all values >= 6 in this case). It is not what we want **in this case!**

`elif` Statement

- Simplifies the writing of multiple-choice decisions…
- Short for: *Else, if*…
- As soon as one of the tested conditions is true, the statement block is executed
  - **No other tests are attempted**
- If none of the tested conditions is true the final `else` block is executed

```
if condition1:
    # block of instructions executed
    # if condition1 is true
elif condition2:
    # block of instructions executed
    # if condition1 is false,
    # and condition2 is true
…
elif conditionN:
    # block of instructions executed
    # if all previous conditions are false,
    # and conditionN is true
else:
    # block of instructions executed
    # if all previous conditions are false
```

`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")
```

THE ORDER IS RELEVANT!!!!!

Multiple choices with/without elif

With

```
if richter >= 8.0 :
  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 :
  print("No destruction of buildings")
```

Without

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

Unreadable «diagonal» code

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 to verify their truth (true or false)
- Computers adopt Boolean logic

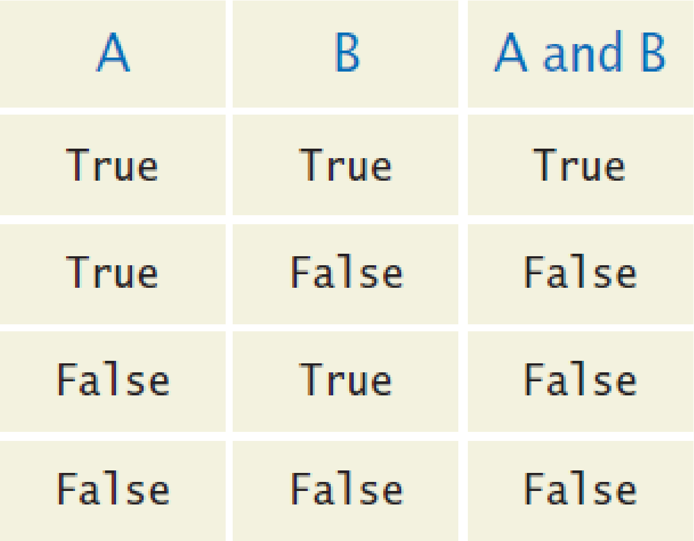
Boolean Variables

- 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")
```

Shorthand form for:

```
if isLiquid == True :
     print(“Liquid”)
```

Chained Comparison Operators

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

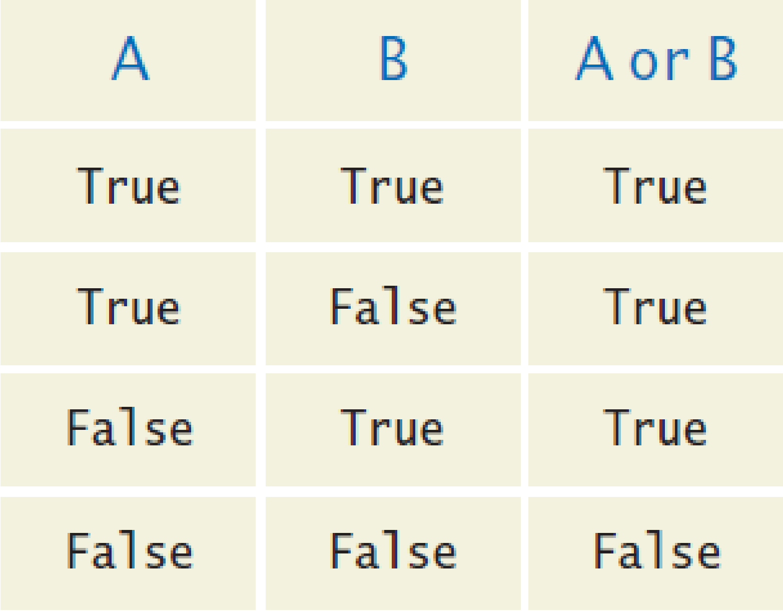
- You may also write: `0 <= temp <= 100`
  - Python allows *chained comparison operators*
  - *Most other programming languages do not allow this*
  - Tip: avoid this shortcut, use an explicit **and**

Combined Conditions: or

- We use **or** if **it’s enough that one of the two conditions is true** for the compound condition to be true:

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

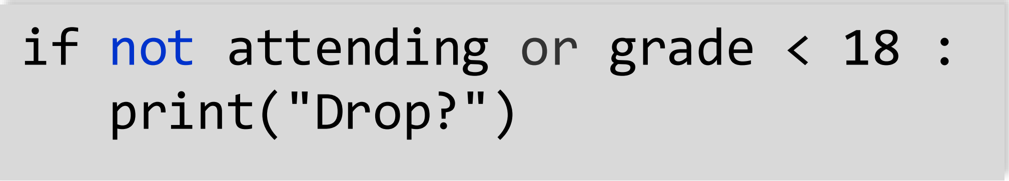
- If *either* condition is true
  - The result is true



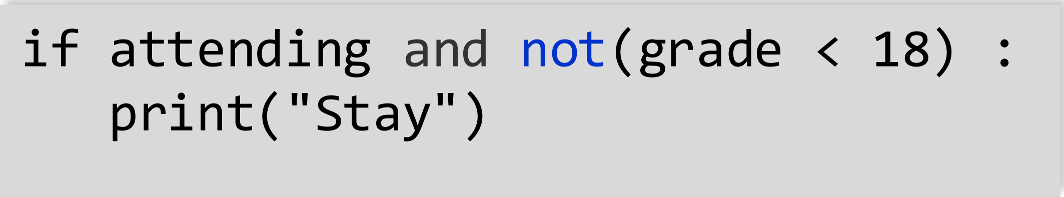
A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

The not operator: not

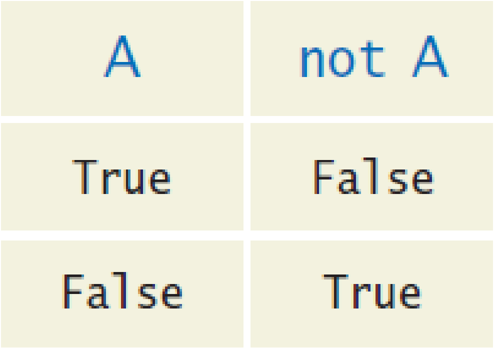
- If you need to invert a boolean variable or comparison, precede it with not



```
if not attending or grade < 18 :
    print("Drop?")
```



```
if attending and not(grade < 18) :
    print("Stay")
```



A	not A
True	False
False	True

Boolean Operator Examples

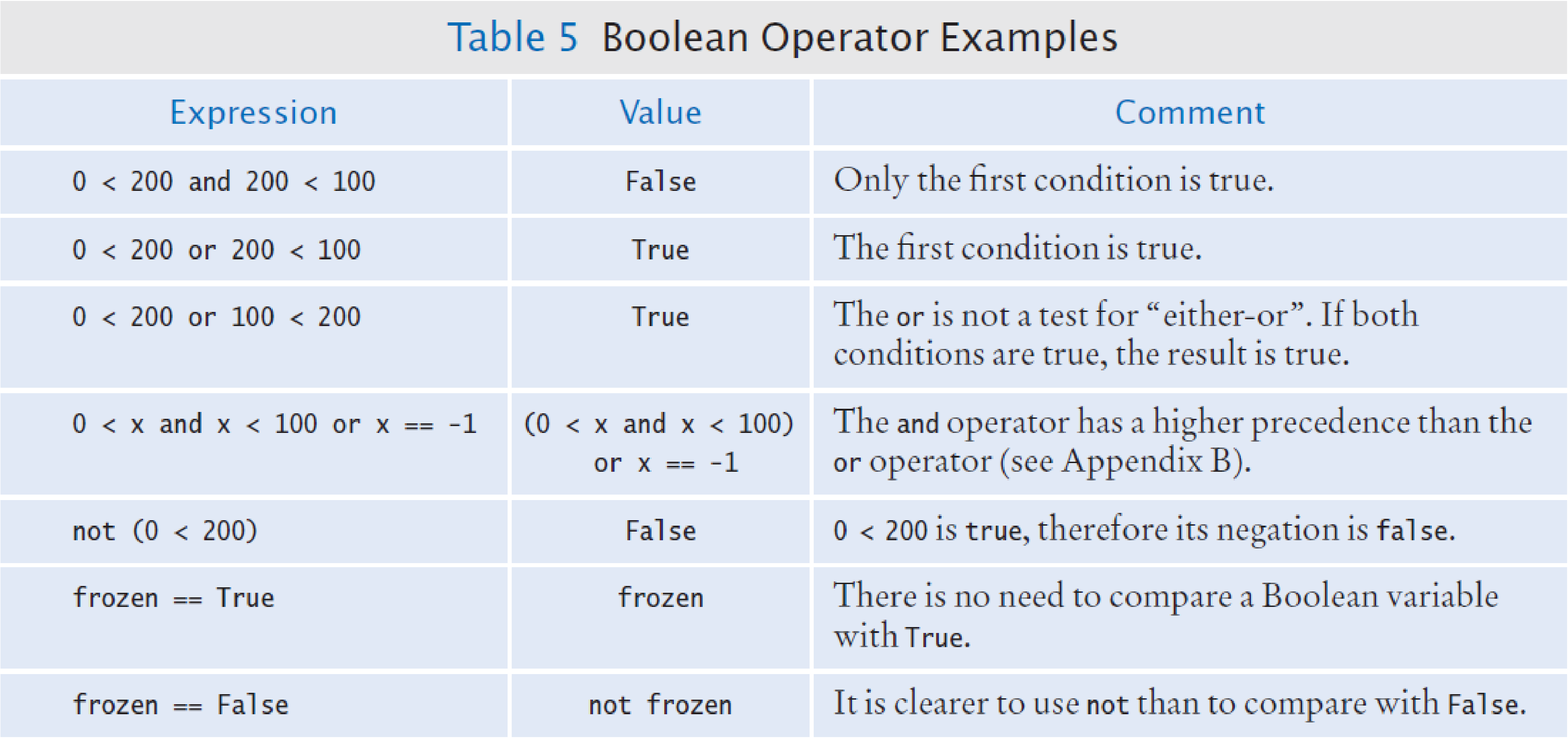


Table 5 Boolean Operator Examples

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:
  - It is a **surprisingly common error.**

- Examples:
  - 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.

Some Boolean Properties

- **Commutative:**
  - A and B = B and A
  - A or B = B or A

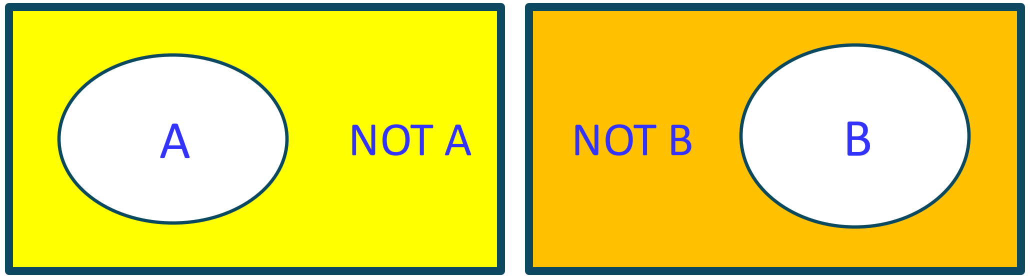
- **Associative:**
  - A and B and C = (A and B) and C = A and (B and C)
  - A or B or C = (A or B) or C = A or (B or C)

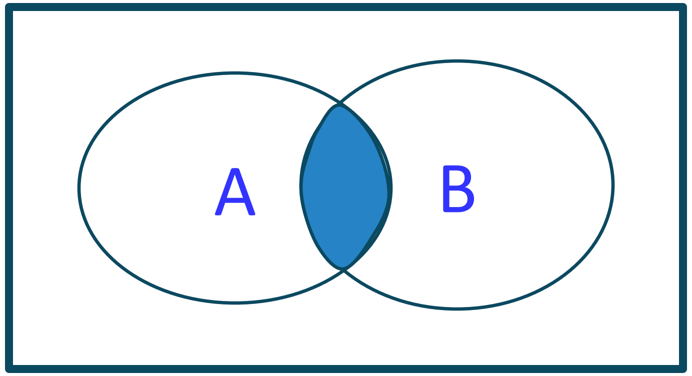
- **Distributive:**
  - A and (B or C) = A and B or A and C
  - A or (B and C) = (A or B) and (A or C)

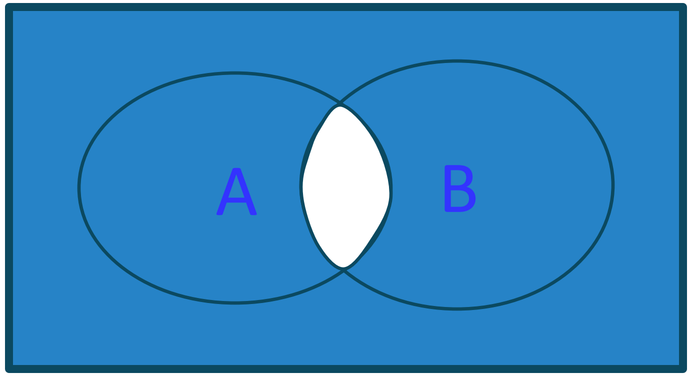
De Morgan’s law

- 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: `not(A and B)`







String analysis

📘 3.8

String Analysis

- We will now see some operators that involve "checks" inside strings:
  - As such, usable within conditions

- In particular:
  - Functions for Analyzing Substrings
  - Functions for checking string characteristics

Analyzing Strings – The `in` Operator

- Used to determine if a string contains a given substring.

  - Given this code segment:

```
name = "John Wayne”
```

  - the expression

```
"Way" in name
```

  - is `True` because the substring `"Way"` occurs within the string stored in the 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(".docx") :
   print("This is a Microsoft Word file.")
```

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

Operations for Testing Substrings

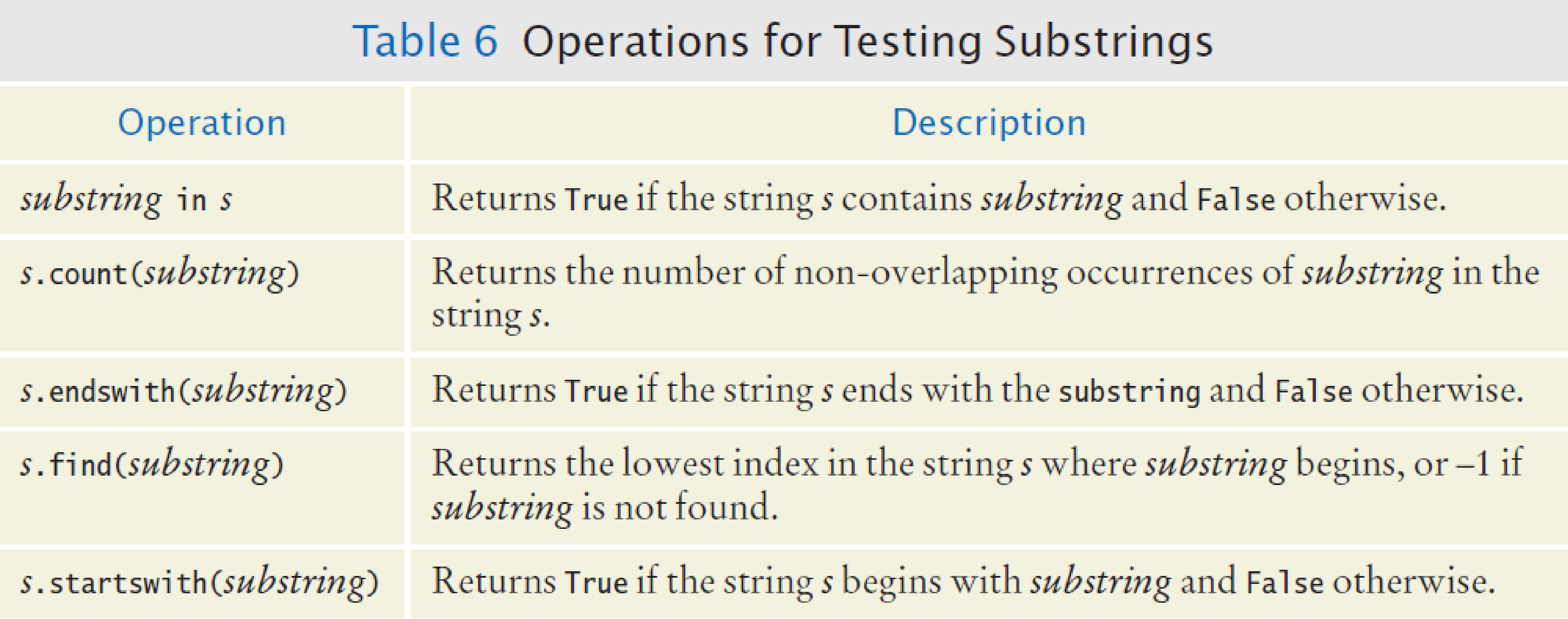


Table 6 Operations for Testing Substrings

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)

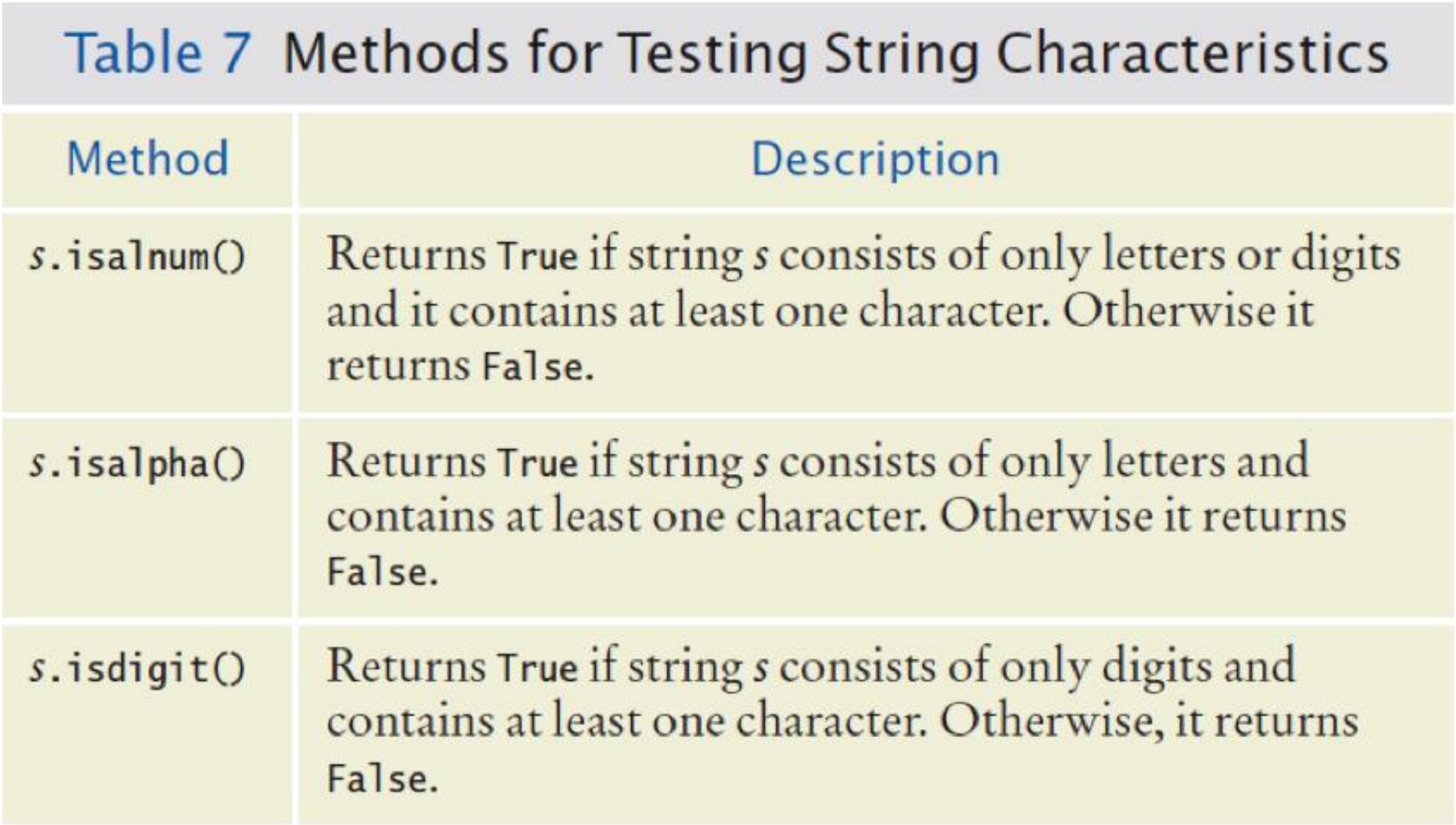


Table 7 Methods for Testing String Characteristics

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)

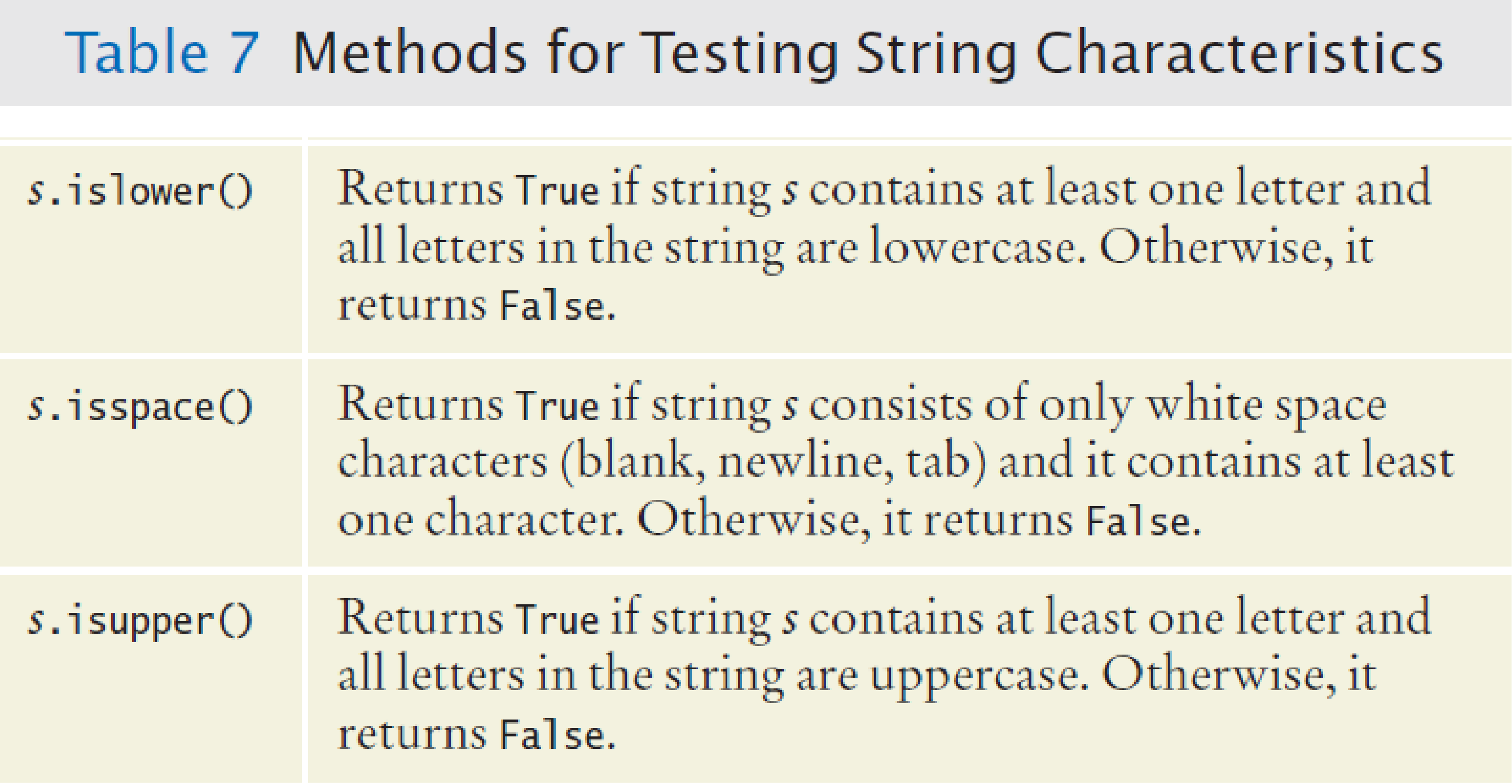


Table 7 Methods for Testing String Characteristics

`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

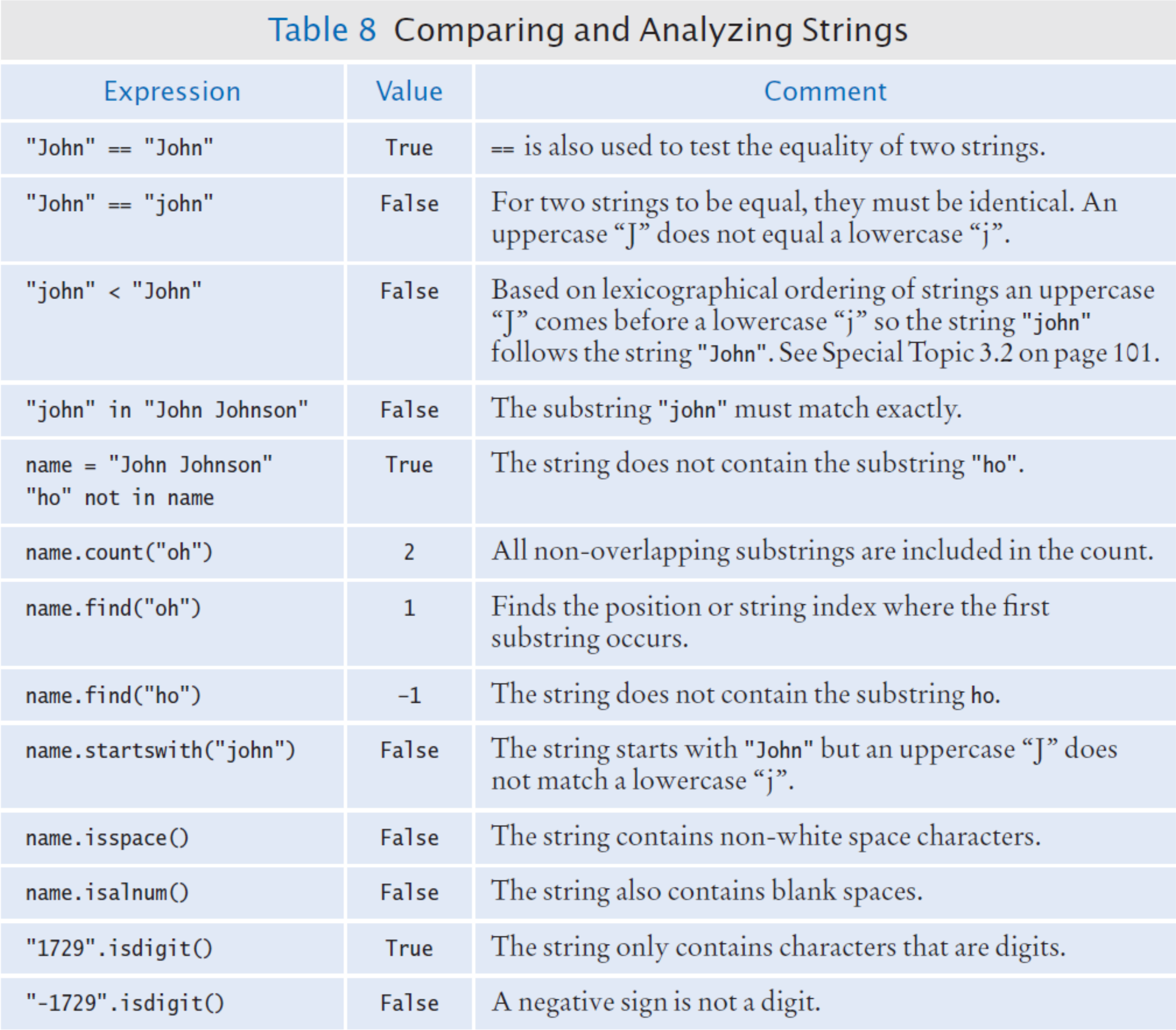


Table 8 Comparing and Analyzing Strings

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.

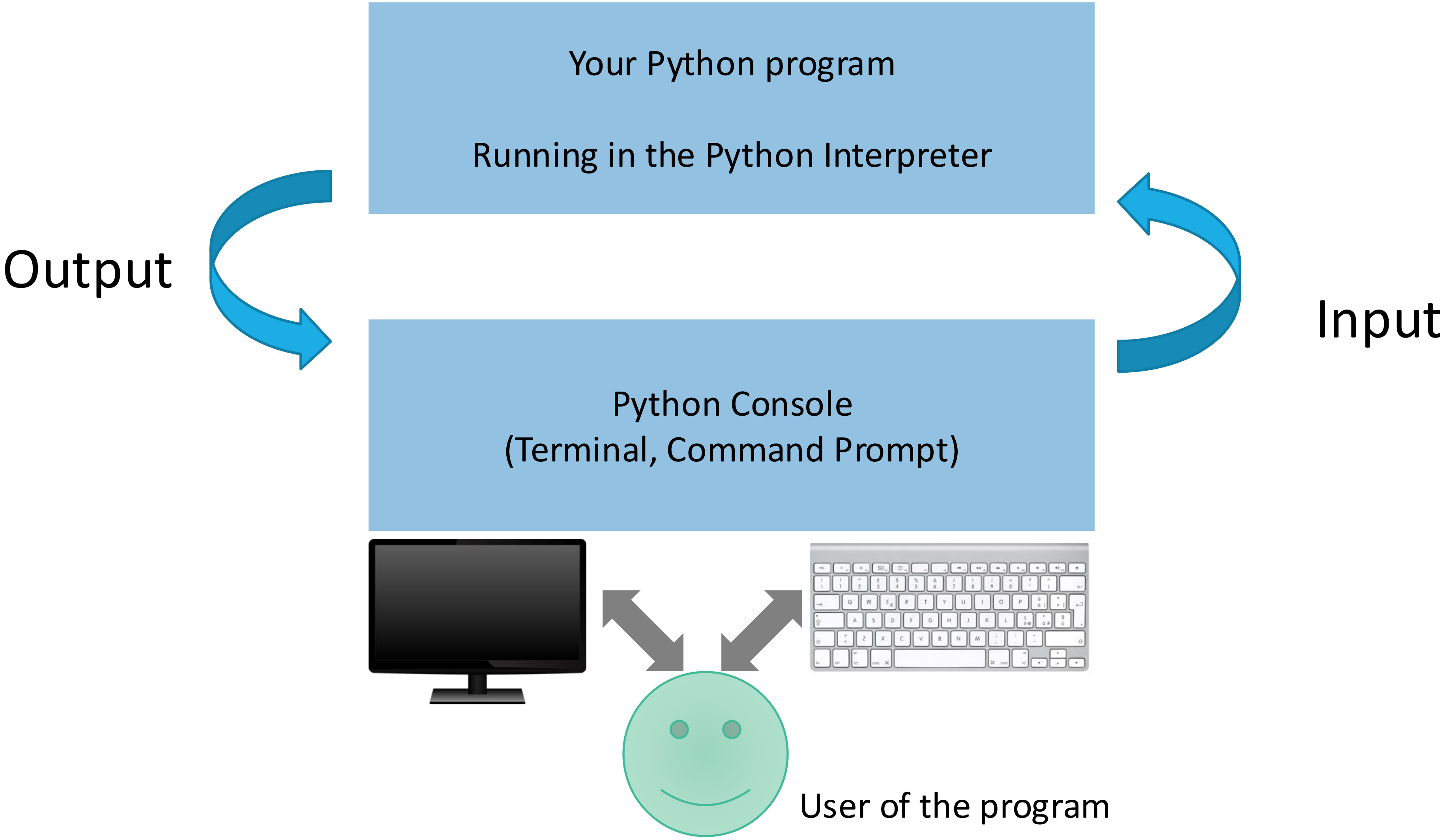
`name = "John Johnson"`
`"ho" not in name`
	`True`	The string does not contain the substring `"ho"`.

`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.

Input (Reprise)

📘 2.5

Input and Output



Your Python program

Running in the Python Interpreter

Output

Input

Python Console
(Terminal, Command Prompt)

User of the program

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

- Sometimes, we need to print output values with a certain **format** (to obtain a more ordered and easy-to-read display)
  - Example: print a real number with 3 fractional digits
  - Example: align text left or right, etc…
- Several methods are available in Python
  - **String concatenation (not very convenient)**
  - **f-Strings**
  - Formatting operator %
  - `.format()` method

We won't see the last two methods…
f-Strings are enough for all our purposes.
Additional information can be found in:
https://pyformat.info/

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 {}.
- The most “modern” way to format output in Python
- 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

```
result = 5
print(f"the result is {result}")
```

```
the result is 5
```

```
a = 7
b = 8
print(f"the result is {a+b}")
```

```
the result is 15
```

```
username = “Pedro”
print(f'my name is {username=}')
```

```
my name is username=Pedro
```

- The value within {…} is converted to string and printed
- It can also be the result of an expression
- Adding the **=**, the name of the variable will be included in the string.

Formatting in f-Strings

- We can modify the **output format** by adding **format specifiers** within the {…}, separated with a : symbol
- Example: Integer values with a fixed number of characters

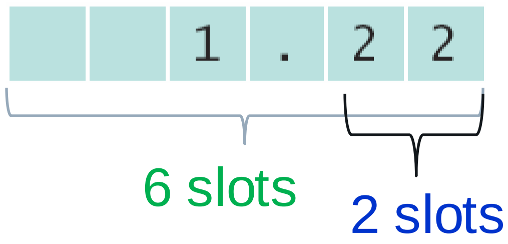
```
dist = 5
print(f"The distance is {dist:3} meters“)  # Adds spaces
# Output: The distance is   5 meters

print(f"The distance is {dist:03} meters“)  # Adds zeros
# Output: The distance is 005 meters
```

Formatting in f-Strings

- Example: Real numbers with a fixed number of characters

```
dist = 5.235
print(f"The distance is {dist:.2f} meters“)  # Rounds to .XX
# Output: The distance is 5.24 meters

print(f"The distance is {dist:6.1f} meters“)  # Rounds+pads
# Output: The distance is    5.2 meters

print(f"The distance is {dist:06.1f} meters“)  # Rounds+pads
# Output: The distance is 0005.2 meters
```

6 total chars
including dot



Formatting in f-Strings

- Example: String on a fixed number of characters

```
name="Bob"
surname="Marley"

print(f“|{name:8}|{surname:>8}|“)

# Output: |Bob     |  Marley|
```

8 chars
left-aligned

8 chars
right-aligned

Format specifiers in the f-String (Simplified)

```
{VarName : [[fill]align][sign][0][width][.precision][type]}
```

- Symbol ‘ : ’
  - `Separates the variable name from the required format`
- **align:** Alignment options for example: <
  - **fill:** only when align is used, specifies the character to fill the remaining slots
- **sing:** (+ or -), add sign to numbers (only valid for int/float)
- **0:** prepend zeros for numbers.
- **type:** conversion type
- See links for more details…

https://docs.python.org/3/library/string.html#formatspec
http://cis.bentley.edu/sandbox/wp-content/uploads/Documentation-on-f-strings.pdf

Alignment options

Option	Meaning
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types.
'^'	Forces the field to be centered within the available space.

https://docs.python.org/3/library/string.html#formatspec

Sign options

Argomento avanzato

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

https://docs.python.org/3/library/string.html#formatspec

Conversion types

Value	Type	Meaning
str	's'	String format. This is the default type for strings and may be omitted.
int	'b'	Binary format. Outputs the number in base 2.
	'c'	Character. Converts the integer to the corresponding unicode character before printing.
	'd'	Decimal Integer. Outputs the number in base 10.
	'o'	Octal format. Outputs the number in base 8.
	'x' / 'X'	Hex format. Outputs the number in base 16, using lower/upper-case letters
	'n'	Number. Same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
float	'e' / 'E'	Exponent notation. Prints in scientific notation using the letter ‘e’ or ‘E’ to indicate the exponent. Default precision is 6.
	'f'	Fixed-point notation. Displays as a fixed-point number. Default precision is 6.
	'F'	Fixed-point notation. Same as 'f', but converts nan to NAN and inf to INF.
	'g'	General format. For a given precision p, rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. Default precision is 6.
	'G'	General format. Same as 'g' except switches to 'E' if the number gets too large.
	'n'	Number. Same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.
	'%'	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.
	None	Similar to 'g', except that fixed-point notation, when used, has at least one digit past the decimal point. The default precision is as high as needed to represent the particular value.

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

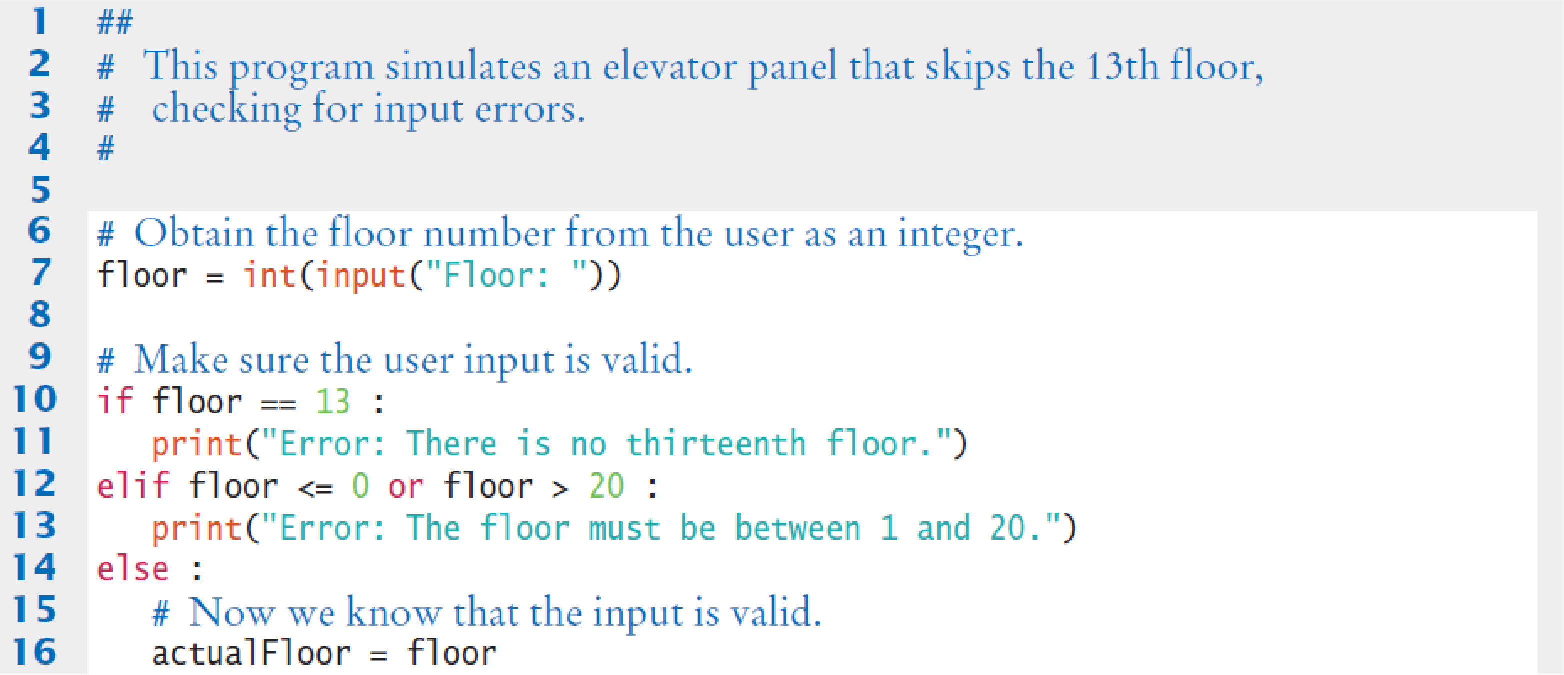
```
if floor == 13 :
    print("Error: There is no thirteenth floor.")
```

  - 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



```
##
#  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 = int(input("Floor: "))

# Make sure the user input is valid.
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
```

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)

Another Example

- US phone numbers have three parts: area code, exchange, and line number, usually written in the form: **(###)###-####**
- Use a boolean variable to validate if a string input by a user contains a correctly formatted US phone number:

```
phone = input(“Give me a phone number:”)
valid = ( len(phone) == 13
          and phone[0] == "("
          and phone[4] == ")"
          and phone[8] == "-"
          and phone[1:4].isdigit()
          and phone[5:8].isdigit()
          and phone[9:13].isdigit() )
```

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 `f`-strings with format specifiers to specify how values should be formatted.