



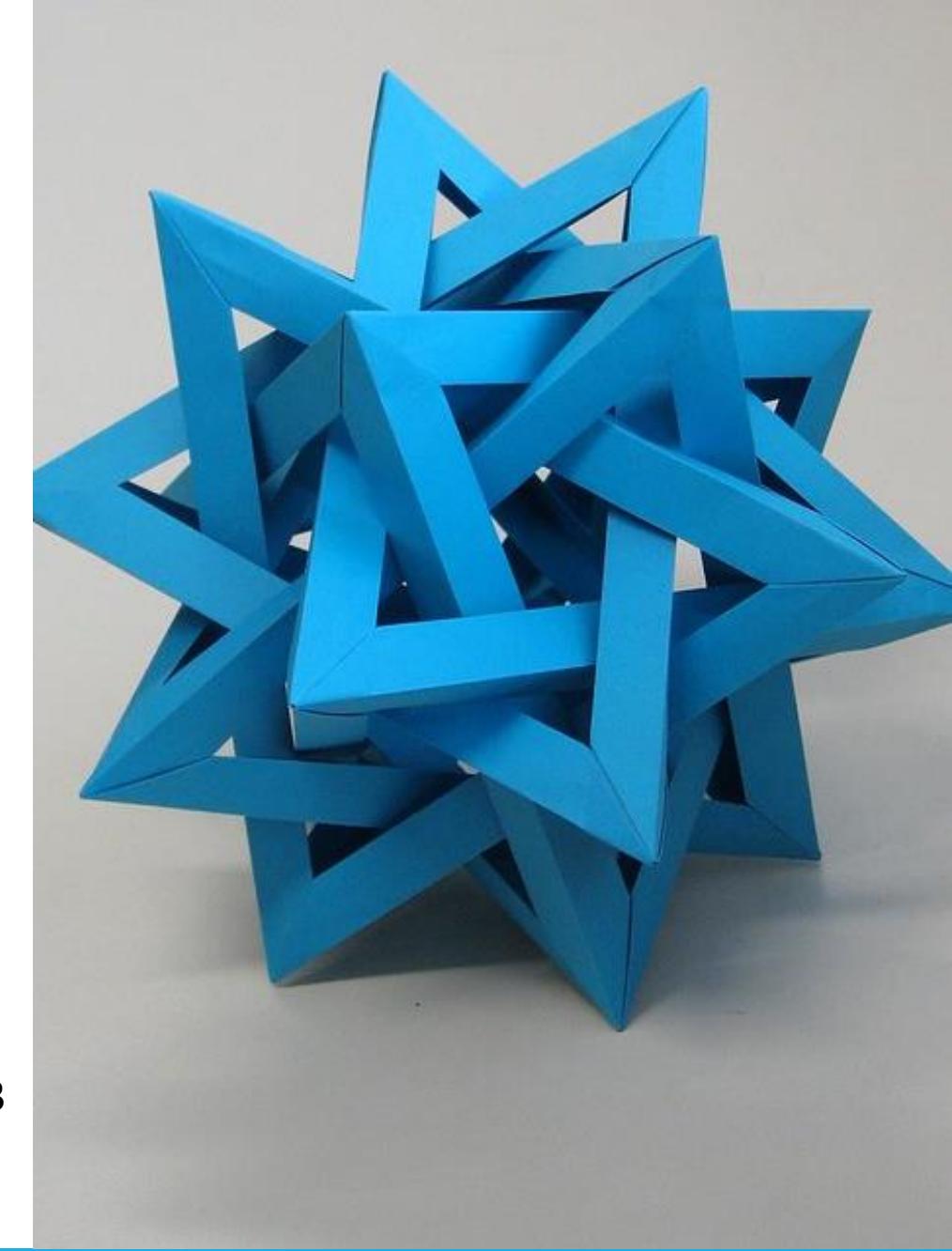
# Unità P3: Decisioni

---

DECISIONI, CONDIZIONI BOOLEANE, ANALISI  
DI STRINGHE, VALIDAZIONE DEGLI INPUT



Capitoli 2 e 3



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

# Obiettivi dell’Unità

- Input di dati numerici e testuali (stringhe)
- Formattazione dell’output
- Implementare decisioni con l’istruzione if
- Confrontare numeri (interi e floating-point) e stringhe
- Scrivere istruzioni usando dati di tipo booleano
- Validare l’input

In questa Unità imparerete a codificare in Python decisioni semplici e complesse. Imparerete inoltre ad applicare quanto appreso al controllo dei dati forniti in input e alla validazione dei risultati.

# Contenuti

- Input di dati e formattazione dell'output
- L'istruzione **if**
- Operatori relazionali
- Cicli annidati
- Condizioni con scelta multipla
- Variabili e operatori booleani
- Analisi di stringhe
- Applicazione: validazione degli input

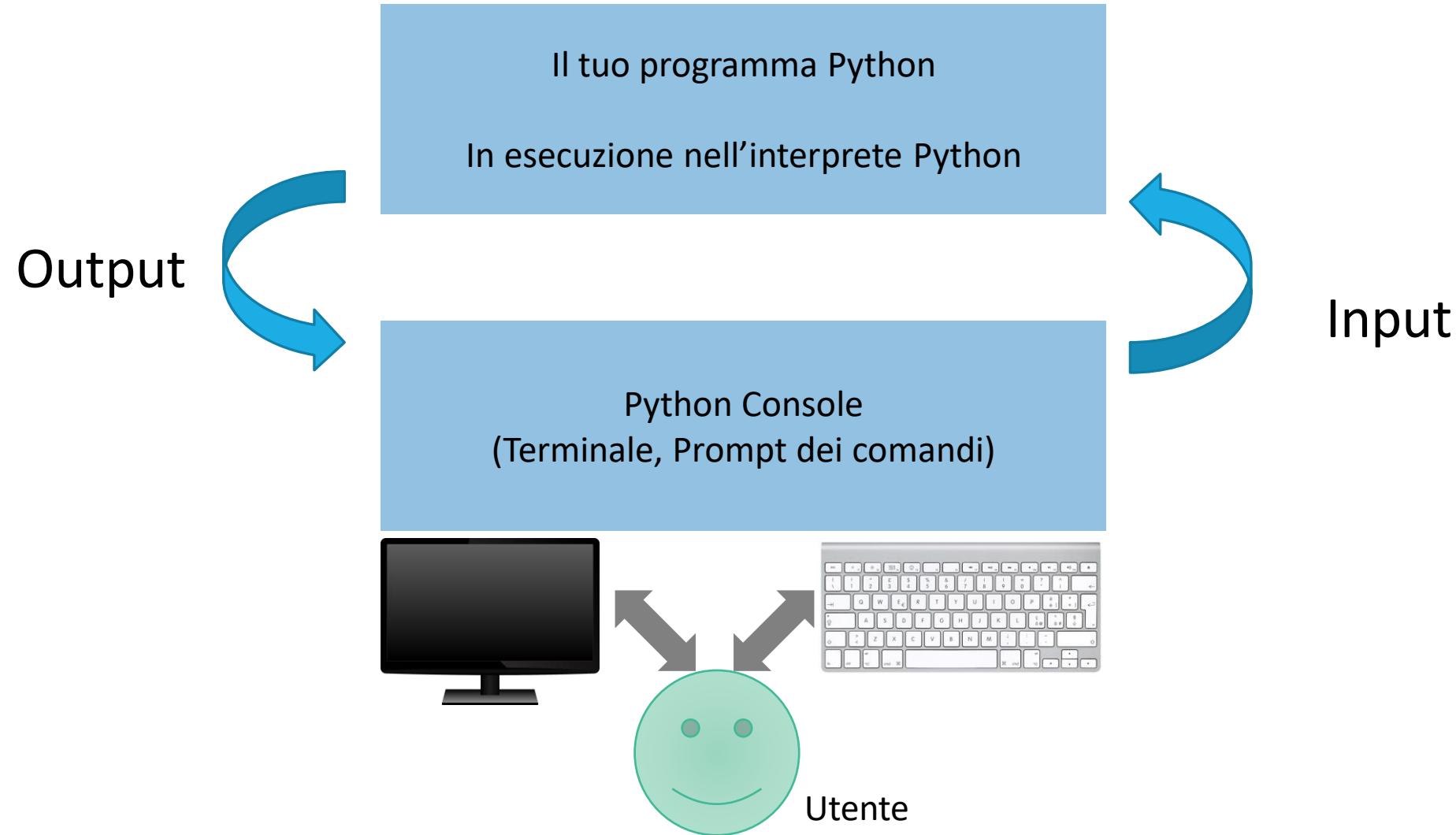
# Input

---



2.5

# Input e Output



# Input e Output

- Puoi leggere una **Stringa** dalla console con la funzione **input()**:

- `name = input("Inserisci il tuo nome")`

- Se serve un input **numerico** (invece che una stringa), si deve convertire la Stringa in un **numero** (intero o virgola mobile)

- `age_string = input("Inserisci la tua età: ") # String`  
`age = int(age_string) # Conversione a int`

- ...o in una singola istruzione:

- `age = int(input("Inserisci la tua età: "))`  
`price = float(input("Inserisci il prezzo: "))`

# Output Formattato

---



2.5

# Output formattato

- Talvolta è opportuno inserire dei valori numerici in stringhe di testo in modo da ottenere un output più chiaro e ordinato da visualizzare
- Python offre diverse soluzioni
  - Concatenazione di stringhe
  - **f-String**
  - L'operatore di formattazione `%`
  - Il metodo `.format()`



Non verranno trattati in dettaglio – utili per chi programmerà in C

Vedere  
<https://pyformat.info/>  
per informazioni su `%` e `format`

# Esempio (confronto dei metodi)



```
pi = 3.14
```

```
r = 2.0
```

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

```
print('The area of a circle of radius '+str(r)+ ' is  
'+str(area))
```

```
print(f'The area of a circle of radius {r} is {area}')
```

```
print('The area of a circle of radius %f is %f' % (r, area))
```

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

# f-String (stringhe formattate)

- Una **stringa formattata** o **f-String** è una stringa preceduta dal carattere '**f**' or '**F**'.
- Queste stringhe possono contenere dei **campi sostituibili**, nella forma di espressioni racchiuse tra parentesi graffe **{ }.**
- Mentre le stringhe classiche hanno normalmente un valore costante, le f-String sono praticamente espressioni valutate al momento dell'esecuzione.

NOTA: Le f-String non sono trattate nel libro di testo.

Vedi:

[https://docs.python.org/3/reference/lexical\\_analysis.html#f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)

<https://realpython.com/python-f-strings/>

# f-String: Esempi

```
f"the result is {result}"
```

```
the result is 5
```

```
f"the sum is {a+b}"
```

```
the sum is 15
```

```
f"the result is {result=}"
```

```
the result is result=5
```

- **Variabili** definite nelle linee di codice precedenti
- **Espressioni aritmetiche** che coinvolgono una o più variabili o valori definiti nelle linee di codice precedenti
- Il valore corrente viene convertito in stringa e «sostituito» al posto delle {...}
- Aggiungendo il simbolo **=**, viene inserito anche il *nome della variabile* (utile per debug)

# Formattazione nelle f-String

- È possibile modificare il formato in cui vengono stampati i valori, utilizzando degli *specificatori di formato*
- Gli specificatori di formato vengono inclusi tra le graffe, separati dal simbolo :
- `f"La distanza è {dist:8.2} metri"`

# Specificatori di formato nelle f-String

- Carattere ' : '
- Allineamento<sup>§</sup> (opzionale), es. <
- Segno<sup>+</sup> (+ o - o spazio, opzionale)
- # (opzionale): formato 'alternativo'
- Ampiezza (opzionale), es. 20
- Carattere di raggruppamento delle migliaia: , oppure \_ (opzionale)
- ' .' e "precisione" (opzionale), es. .2
- Tipo di conversione<sup>‡</sup>, es. f

<https://docs.python.org/3/library/string.html#format-spec>

<http://cis.bentley.edu/sandbox/wp-content/uploads/Documentation-on-f-strings.pdf>

# § Opzioni di allineamento

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.

- ❖ L'allineamento, se presente, può essere preceduto da un carattere di riempimento (opzionale), es. \_

<https://docs.python.org/3/library/string.html#format-specification-mini-language>

# † Opzioni di segno

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).
' '' (one 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#format-spec>

# ‡ Tipo di conversione

Value	Type	Meaning
str	's'	String format. This is the default type for strings and may be omitted.
	'b'	Binary format. Outputs the number in base 2.
	'c'	Character. Converts the integer to the corresponding unicode character before printing.
int	'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.
	'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.
float	'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.

# Stampare numeri reali

- Quando occorre stampare un numero reale con un determinato numero di cifre decimali, l'ideale è utilizzare il formato:

```
{     valore :     ampiezza .     precisione f }
```

Valore (float)  
da stampare

Numero (minimo) di caratteri che verranno stampati (compresa la parte intera, in segno, il punto decimale, la parte frazionaria). Lo spazio in eccesso viene “riempito” con spazi, ed il risultato viene allineato a destra

Numero di cifre che verranno stampate dopo il punto decimale

Usa il formato  
“float”  
**(non dimenticare!)**

# Esempi

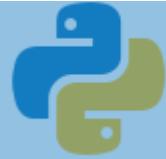
- `f'{pi:10.3f}'`
  - `'.....3.142'`
- `f'{pi:8.6f}'`
  - `'3.141593'`
- `f'{pi:10.6f}'`
  - `'...3.141593'`
- `f'{7:10.6f}'`
  - `'...7.000000'`

{ *valore* : *ampiezza* . *precisione* f }

# Arrotondamento

- Notare la **differenza** tra:
- **print(round(pi, 3))**
  - Arrotonda il valore float alla terza cifra, e stampa il risultato
  - Non posso controllare l'ampiezza complessiva
    - Non inserisce spazi all'inizio, non allinea a destra il valore
  - Se le cifre decimali arrotondate fossero degli 0, non verrebbero stampati
    - print(round(7, 3)) -> '7' e non '7.000'
- **print(f'{pi:8.3f}')**
  - Definisce il numero di caratteri da stampare
  - Più ordinato e flessibile, se utilizzato per stampare
- **Conclusione:** meglio round() per arrotondare in formule matematiche, meglio f'{ :f}' per stampare

# Esempio



```
##  
# This program prints the price per ounce for a six-pack of cans.  
  
# Define constant for pack size.  
CANS_PER_PACK = 6  
  
# Obtain price per pack and can volume.  
user_input = input("Please enter the price for a six-pack: ")  
pack_price = float(user_input)  
  
user_input = input("Please enter the volume for each can (in ounces): ")  
can_volume = float(user_input)  
  
# Compute pack volume.  
pack_volume = can_volume * CANS_PER_PACK  
  
# Compute and print price per ounce.  
price_per_ounce = pack_price / pack_volume  
print(f'Price per ounce: {price_per_ounce:.2f}') # using f-Strings
```

volume2.py

# Operatore di formattazione %

- La stampa di numeri floating point può sembrare strana:
  - Price per liter: 1.21997
- Per controllare il modo in cui il numero viene rappresentato si usa l'operatore %  
"string with *format specifiers*" % ( value, value, ... )
- Ex: "Price per liter: %.2f" % (price)
  - Ogni indicatore di formato viene sostituito, nell'ordine, da un valore calcolato o presente in una variabile
  - La formattazione è controllabile dall'utente.

# Principali specificatori di formato

- **%s** – Stringa (o altri oggetti rappresentabili come stringa, ad esempio numeri)
- **%d** – Numero intero
- **%f** – Numero in Floating point
- **%.<number of digits>f** – Numero Floating point con un numero fisso di cifre dopo il punto decimale
- **%x/%X** – Numero intero rappresentato in base 16 (hex) (minuscolo/maiuscolo)

# Output formattato

- Esempi

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

Price per liter: 1.22

```
print("Price per liter %10.2f" %(price))
```

Price per liter: 1.22



# Syntassi: operatore di formattazione

## Sintassi

*stringaConInformazioniDiFormato* % (*valore<sub>1</sub>*, *valore<sub>2</sub>*, ..., *valore<sub>n</sub>*)

La stringa con informazioni di formato può contenere uno o più indicatori di formato, oltre a caratteri letterali.

Per imporre un formato a un singolo valore non servono le parentesi.

```
print("Quantity: %d Total: %10.2f" % (quantity, total))
```

Spesso l'argomento di print è una stringa con informazioni di formato.

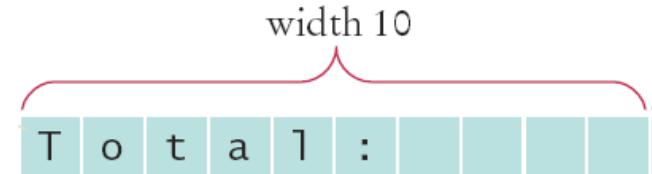
Indicatori di formato.

I valori a cui imporre il formato: nella stringa risultante, ciascun valore sostituisce uno degli indicatori.

# Indicatori di formato: Esempi

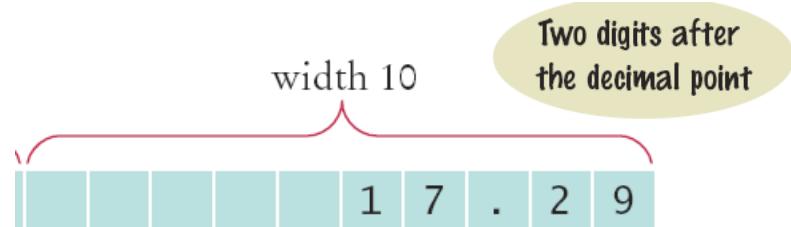
- Giustifica una stringa a sinistra:

```
print("%-10s" %("Total:"))
```



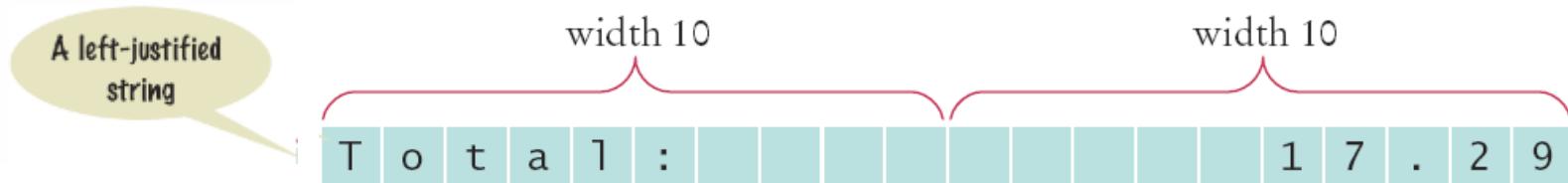
- Giustifica a destra un numero con due cifre decimali

```
print("%10.2f" %(price))
```



- È possibile usare indicatori multipli per stampare più dati:

```
print("%-10s%10.2f" %("Total: ", price))
```



# Indicatori di formato: Esempi

Table 9 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	2 4	Use d with an integer.
"%5d"	2 4	Spaces are added so that the field width is 5.
"%05d"	0 0 0 2 4	If you add 0 before the field width, zeroes are added instead of spaces.
"Quantity:%5d"	Q u a n t i t y :      2 4	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1 . 2 1 9 9 7	Use f with a floating-point number.
".2f"	1 . 2 2	Prints two digits after the decimal point.
"%.2f"	1 . 2 2	Spaces are added so that the field width is 7.
"%s"	H e l l o	Use s with a string.
"%d %.2f"	2 4 1 . 2 2	You can format multiple values at once.
"%9s"	H e l l o	Strings are right-justified by default.
"%-9s"	H e l l o	Use a negative field width to left-justify.
"%d%%"	2 4 %	To add a percent sign to the output, use %%.

# Indicatori di formato: regola generale

- Carattere '%'
- Flag di conversione (opzionale) – es. -
- Lunghezza minima del campo (opzionale) – es. 20
- ' .' e "precisione" (opzionale) – es. .2
- Un carattere che indica il tipo di conversione – es. f

<https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>

# Indicatori di formato: carattere di conversione

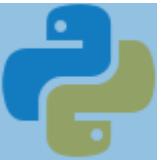
Carattere	Significato
'd' / 'i'	Signed integer decimal.
'o'	Signed octal value.
'x' / 'X'	Signed hexadecimal (lowercase/uppercase).
'e' / 'E'	Floating point exponential format (lowercase/uppercase).
'f' / 'F'	Floating point decimal format.
'g' / 'G'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'c'	Single character (accepts integer or single character string).
'r'	String (converts any Python object using repr()).
's'	String (converts any Python object using str()).
'a'	String (converts any Python object using ascii()).
'%'	No argument is converted, results in a '%' character in the result.

<https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>

# Indicatori di formato: flag di conversione

Flag	Meaning
'#'	The value conversion will use the “alternate form”.
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

<https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>



# Esempio

```
##  
# This program prints the price per ounce for a six-pack of cans.  
  
# Define constant for pack size.  
CANS_PER_PACK = 6  
  
# Obtain price per pack and can volume.  
user_input = input("Please enter the price for a six-pack: ")  
pack_price = float(user_input)  
  
user_input = input("Please enter the volume for each can (in ounces): ")  
can_volume = float(user_input)  
  
# Compute pack volume.  
pack_volume = can_volume * CANS_PER_PACK  
  
# Compute and print price per ounce.  
price_per_ounce = pack_price / pack_volume  
print("Price per ounce: %8.2f" % price_per_ounce) # using %-formatting
```

volume2.py

# Confronto tra metodi di formattazione

## OPERATORE: %

- "your age is %d" % ( age )
- Stringa di formato
- % placeholder
  - Specificano il tipo di dato e le opzioni di formattazione
- Il valore vero e proprio viene inserito prendendolo dai valori in %(val, val,...) rispettando l'ordine

## F-STRING

- f"your age is {age}"
- Stringa preceduta dalla lettera "f"
- {...} placeholder
  - Specificano quale variabile andrà a sostituire
  - Possono anche contenere un'espressione
- {age} è una variabile esistente nel codice python in esame

# L'istruzione if

---



3.1

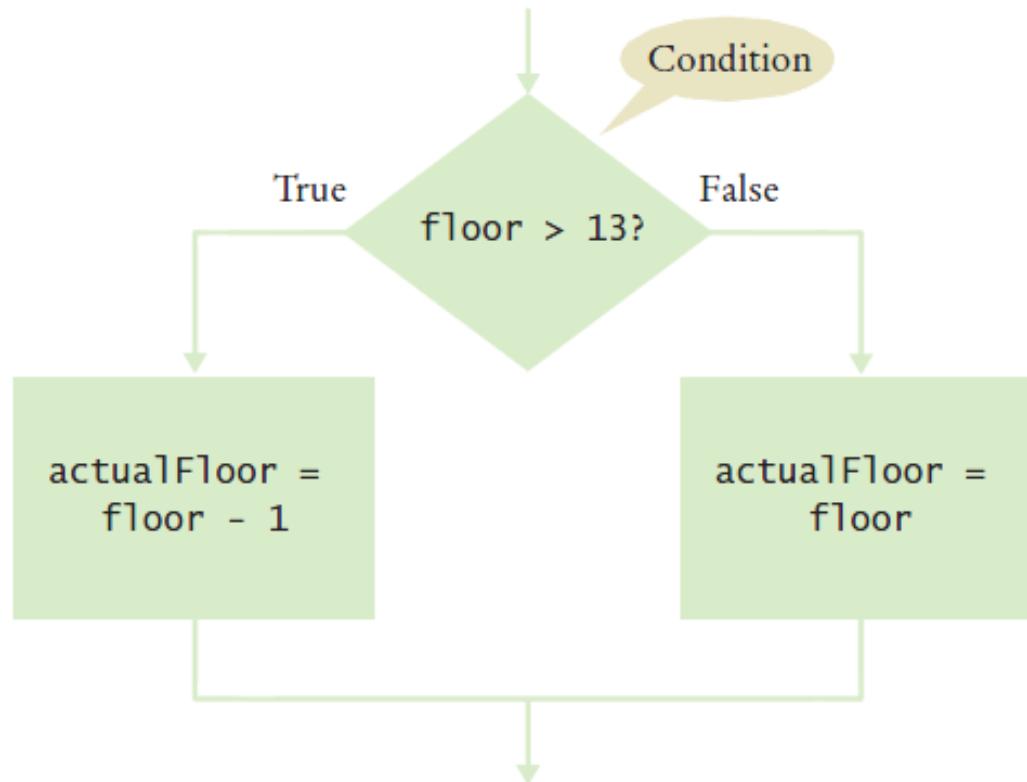
# L'istruzione if

- In un programma è spesso necessario prendere delle decisioni in base al valore di ingresso o di variabili interne
- Per esempio, nella costruzione dei palazzi (in America) si “salta” spesso il 13° piano, e così devono fare gli ascensori
  - Il 14° piano è in realtà il 13° piano
  - Quindi il numero di ogni piano sopra al 12°, è in realtà ‘piano – 1’
    - `if floor > 12, actual_floor = floor - 1`
- Le due keyword che permettono di realizzare un costrutto condizionale sono:
  - `if`
  - `else`

L'istruzione if permette a un programma di fare cose diverse in base al valore dei dati che sta elaborando.

# Flowchart dell'istruzione if

- Uno solo dei due rami è eseguito
  - Diramazione VERA (**if**) o Diramazione FALSA (**else**)

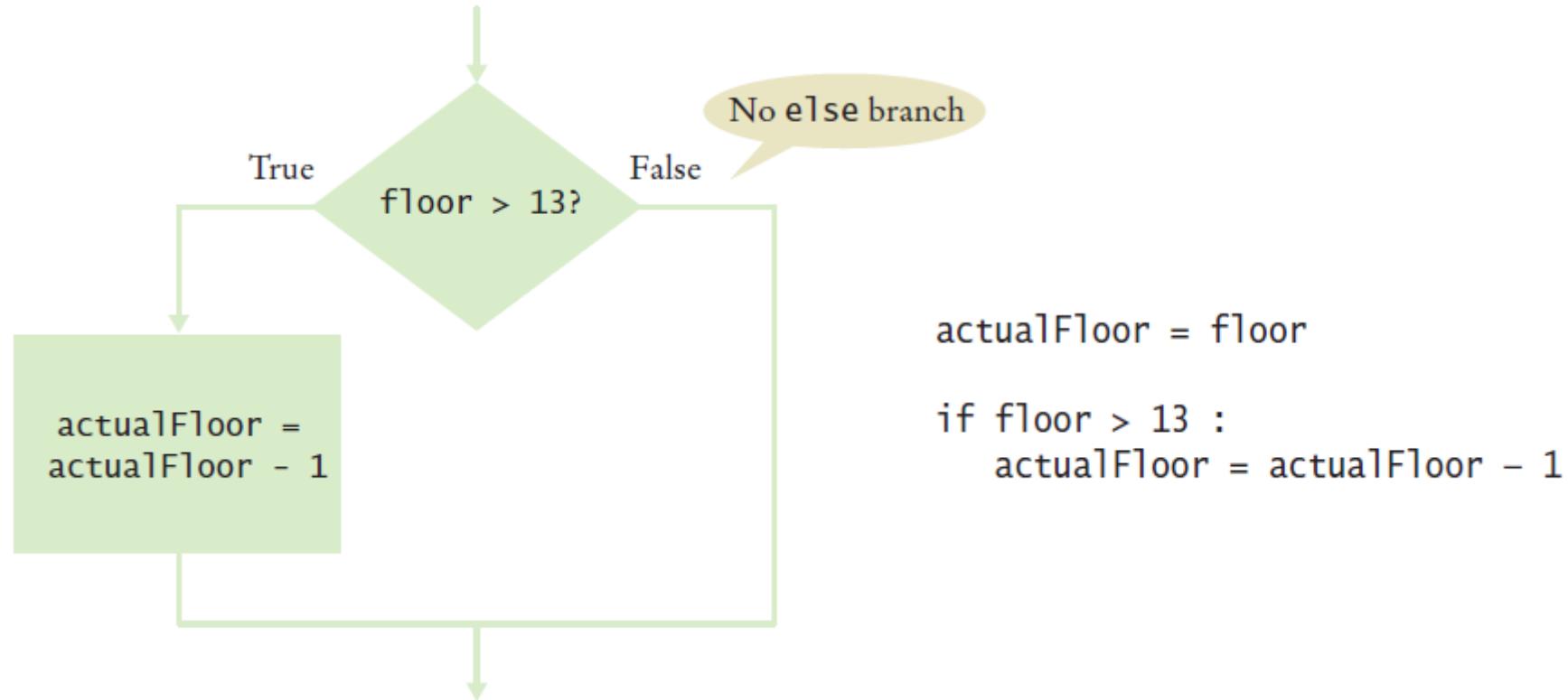


```
actualFloor = 0  
  
if floor > 13 :  
    actualFloor = floor - 1  
else :  
    actualFloor = floor
```

**Indentazione:**  
Le istruzioni da eseguire nella  
diramazione if o else devono  
essere indennate di alcuni spazi  
(2 o 4)

# Flowchart con il solo ramo VERO

- Un'istruzione if può comparire da sola (senza il ramo 'Falso')



# Sintassi 3.1: L'istruzione if

## Sintassi

```
if condizione :  
    enunciati
```

```
if condizione :  
    enunciati1,  
else :  
    enunciati2
```

## Esempio

Una *condizione* che è vera o falsa. Spesso contiene operatori relazionali ( $==$   $!=$   $<$   $<=$   $>$   $>=$ ).

Le clausole **if** e **else** devono essere incolonnate correttamente.

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

Se il ramo **else** non contiene azioni, la clausola va omessa.

Il carattere "due punti" identifica un enunciato composto.

Se la *condizione* è vera, gli enunciati presenti in questo ramo vengono eseguiti in sequenza; se la *condizione* è falsa, vengono ignorati.

Se la *condizione* è falsa, gli enunciati presenti in questo ramo vengono eseguiti in sequenza; se la *condizione* è vera, vengono ignorati.

# Esempio

```
##  
# 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:  
    actual_floor = floor - 1  
else:  
    actual_floor = floor  
  
# Print the result.  
print("The elevator will travel to the actual floor", actual_floor)
```



## Program Run

Floor: 20

The elevator will travel to the actual floor 19

 elevatorsim.py

# Esempio 1

- Aprire il file: elevatorsim.py
- Eseguire il programma
  - Prova con valori minore di 13
    - Che risultato viene fornito?
  - Esegui il programma con un valore maggiore di 13
    - Che risultato viene fornito?
- Cosa succede se viene inserito il numero 13?
  - Risultato errato



elevatorsim.py

# Esempio 1 - correzione

- Correggere il codice precedente in modo che:
  - Il programma controlli la validità dell'input dell'utente
  - Se l'input è **13** (valore non accettabile), stampi un **messaggio di errore**
  - **Sostuisca** il valore 13 con il valore 14
- L'operatore di uguaglianza è “**==**”

Avvertenza:

**Non confondere = con ==**

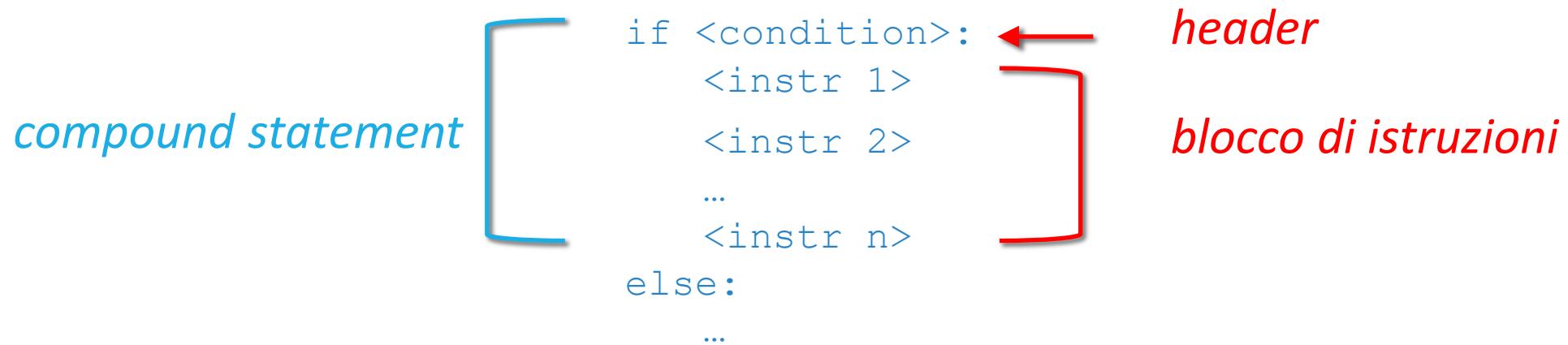
= dichiara una variabile  
= assegna un valore  
== confronta due valori

# Esempio 1 – modifica facoltativa

- Modificare il programma in modo che:
  - Consideri anche il numero 17 come numero sfortunato (e salti quindi sia il 13 che il 17 nella «numerazione» dei piani).

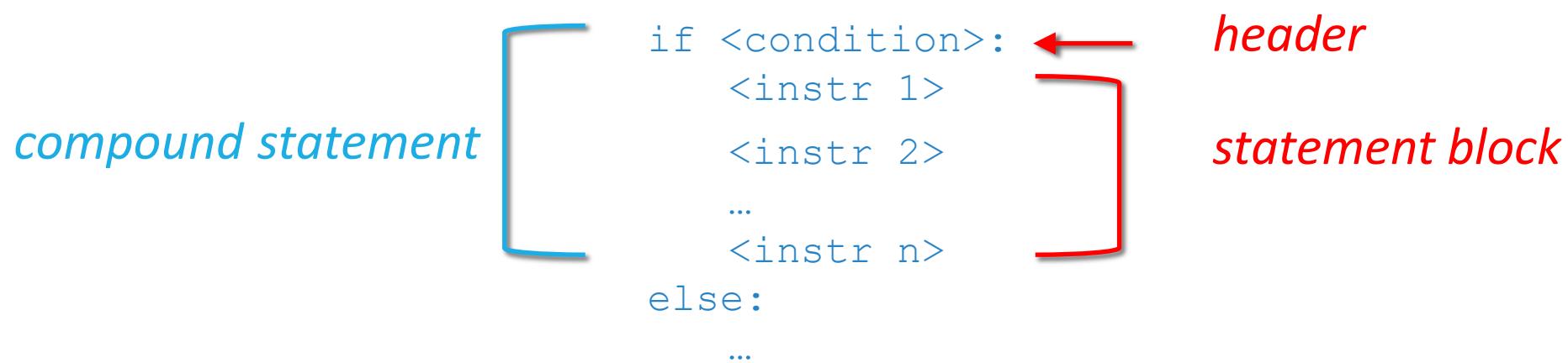
# I Compound Statement

- Alcuni costrutti Python sono **compound statement (istruzioni composte)**.
  - L'istruzione `if` è un esempio di compound statement
- I compound statement si sviluppano su **più di una riga** di codice e consistono di un **header** (intestazione) e di **blocco di istruzioni**
- I compound statement richiedono un “`:`” alla fine dell' header:



# I Compound Statement

- Il blocco di istruzioni è un insieme di una o più istruzioni, tutte con la stessa indentazione
- Il blocco di istruzioni
  - Inizia sulla riga successiva all'header
  - Contiene tutte le istruzioni indentate rispetto all'header
  - Si conclude appena si trova un'istruzione con un livello di indentazione inferiore
- Quasi tutti gli IDE indentano automaticamente i blocchi.



# I Compound Statement

- I blocchi di istruzioni possono essere **annidati all'interno di altri blocchi di istruzioni**
- Nel caso dell'istruzione **if** il blocco di istruzioni specifica:
  - Le istruzioni che devono essere eseguite se la condizione è vera
  - ... o saltate se la condizione è falsa
- I blocchi di istruzioni sono indentati anche per fornire all'utente un aiuto visivo nel capire la logica e il flusso del programma

# Consigli sull'indentazione dei blocchi

- Lascia che sia pyCharm ad indentare ... (*menu: Code – Auto-Indent lines*)

```
if totalSales > 100.0 :  
    discount = totalSales * 0.05  
    totalSales = totalSales - discount  
    print("You received a discount of $%.2f" % discount)  
else :  
    diff = 100.0 - totalSales  
    if diff < 10.0 :  
        print("If you were to purchase our item of the day you can receive a 5% discount.")  
    else :  
        print("You need to spend $%.2f more to receive a 5% discount." % diff)  
        ↑  
        ↑  
        ↑  
0 1 2 Indentation level
```

- Questo codice è chiamato “strutturato a blocchi”. Indentare correttamente non solo è obbligatorio in Python, ma permette anche di rendere il codice più leggibile.

# Errore comune

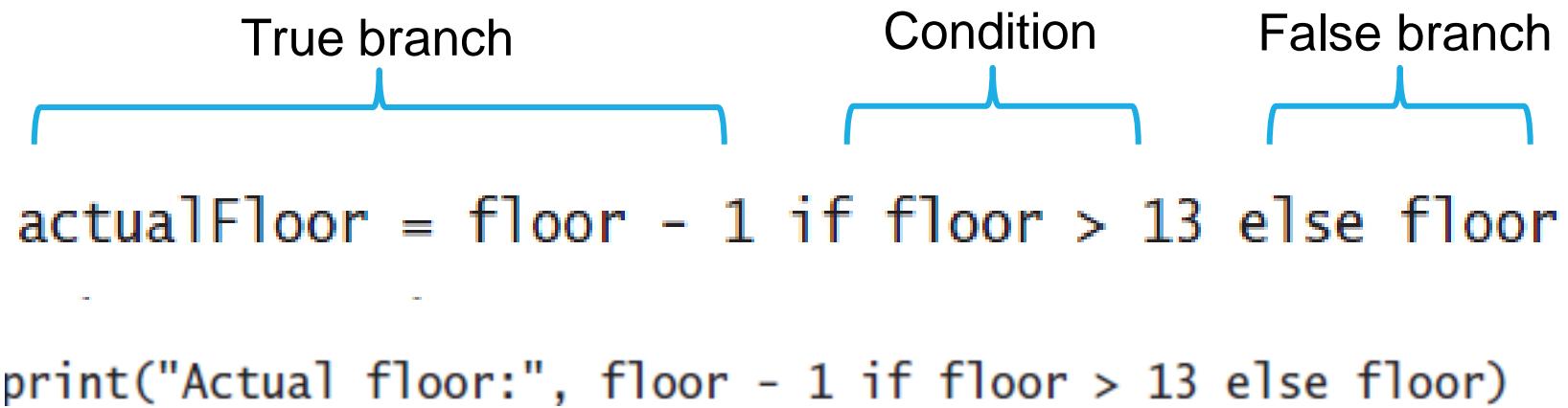
- Bisogna evitare di duplicare il codice nelle diramazioni
- Se lo stesso codice è eseguito in entrambe le diramazioni, allora si può muovere fuori dall' **if**.

```
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)
```

# L'operatore condizionale

- Una “scorciatoia” che potreste trovare in Python
  - Non è usata in questo corso
  - Può essere utilizzata ovunque serva inserire un valore



**La complessità non è utile quando si programma....  
Questa “scorciatoia” è difficile da leggere e può creare confusione**

# Operatori relazionali

---



3.2

# Operatori relazionali

- Ogni istruzione **if** è associata a una **condizione** che generalmente confronta due valori con un **operatore**

```
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

# Assegnazione vs. Test di uguaglianza

- Assegnazione:

```
floor = 13
```

- Test di uguaglianza: *controlla che qualcosa sia vero*

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

Mai confondere

=

con

==

# Confronto di stringhe

- Verificare se due stringhe sono uguali

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

- Verificare se due stringhe sono diverse

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

# Verifica di uguaglianza fra stringhe

- Se uno qualunque dei caratteri è diverso, le stringhe non sono uguali:

name1 = 

J	o	h	n		W	a	y	n	e
---	---	---	---	--	---	---	---	---	---

name2 = 

J	a	n	e		W	a	y	n	e
---	---	---	---	--	---	---	---	---	---

  
La sequenza "ane"  
non è uguale a "ohn"

name1 = 

J	o	h	n		W	a	y	n	e
---	---	---	---	--	---	---	---	---	---

name2 = 

J	o	h	n		w	a	y	n	e
---	---	---	---	--	---	---	---	---	---

  
La lettera maiuscola "W" non è  
uguale alla lettera minuscola "w"

# Operatori relazionali: Esempi (1)

Espressione	Valore	Commento
$3 \leq 4$	Vero	$3 \leq$ è minore di 4; $\leq$ verifica se “è minore di o uguale a”.
 $3 \leqslant 4$	Errore	L’operatore “minore di o uguale a” è $\leq$ , non $\leqslant$ . Il simbolo “minore di” figura per primo.
$3 > 4$	Falso	$>$ è il contrario di $\leq$ .
$4 < 4$	Falso	La parte sinistra deve essere strettamente minore della parte destra perché il risultato sia vero.
$4 \leq 4$	Vero	Le due espressioni confrontate sono uguali; $\leq$ verifica se “è minore di o uguale a”.
$3 == 5 - 2$	Vero	$==$ verifica l’uguaglianza.
$3 != 5 - 1$	Vero	$!=$ verifica l’ineguaglianza, ed è vero che 3 è diverso da $5 - 1$ .
 $3 = 6 / 2$	Errore	Per le verifiche di uguaglianza bisogna usare $==$ .
$1.0 / 3.0 == 0.3333333333$	Falso	Anche se i valori sono molto simili, non sono esattamente uguali (Errori comuni 3.2).
 $"10" > 5$	Errore	Non si può confrontare una stringa con un numero.

# Esempio

- Aprire il file: compare.py
- Eseguire il programma
  - Quali risultati fornisce?



compare.py

# Errore comune (Floating Point)

- I numeri in floating-point hanno una precisione limitata (dal numero di bit), e i calcoli su di essi possono introdurre errori di approssimazione.
- Bisogna tenere conto di questi errori quando si confrontano numeri in floating-point.

# Errore comune (Floating Point, 2)

- Per esempio, questo codice moltiplica la radice quadrata di 2 per se stessa.
- Ci aspetteremmo 2 come risultato:

```
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 2.000000000000004

# L'uso di una costante EPSILON

- Si può utilizzare una valore molto piccolo per verificare se la differenza tra due numeri floating-point è ‘*abbastanza piccola*’
  - Considero i due numeri uguali se la loro differenza è minore di un certo limite EPSILON
  - Cioè x e y sono «uguali» se:

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

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

# L'uso di `math.isclose()`

- Il confronto a meno di un errore può essere ottenuto anche con la funzione `isclose()` del modulo `math`

```
from math import isclose
if isclose( r*r, 2.0 ):
```
- Si può definire l'errore *relativo* (specificando `rel_tol`) o l'errore *assoluto* (`abs_tol`)

## Syntax

```
math.isclose(a, b, rel_tol, abs_tol)
```

$$|a - b| \leq \max\left(\text{rel}_{tot} \cdot \max(|a|, |b|), \text{abs}_{tot}\right)$$

## Parameter Values

Parameter	Description
<code>a</code>	Required. The first value to check for closeness
<code>b</code>	Required. The second value to check for closeness
<code>rel_tol = value</code>	Optional. The relative tolerance. It is the maximum allowed difference between value <code>a</code> and <code>b</code> . Default value is <code>1e-09</code>
<code>abs_tol = value</code>	Optional. The minimum absolute tolerance. It is used to compare values near 0. The <code>value</code> must be at least 0

# L'ordine alfabetico

- Per confrontare due stringhe alfabeticamente si usa un operatore relazionale:
  - `string1 < string2`
- Note
  - Le lettere MAIUSCOLE vengono prima delle minuscole
    - ‘A’ viene prima di ‘a’, ma anche ‘Z’ viene prima di ‘a’
  - Lo ‘spazio’ viene prima di tutti i caratteri stampabili
  - I numeri (0-9) vengono prima delle lettere
  - L’ordine delle lettere è definito dal codice Basic Latin (ASCII) Subset of Unicode
    - L’ordine delle lettere accentate non è sempre logico né corretto....

# Precedenze degli operatori

- Gli operatori aritmetici hanno la precedenza sugli operatori relazionali.
  - Il calcolo è fatto **PRIMA** del confronto
  - Nella maggior parte dei casi i calcoli sono a destra dell'operatore di confronto o assegnazione.

Calculations

```
actual_floor = floor + 1
```

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

# Esempio

---

# Esempio - Vendite

- Un negozio di computer annuncia una nuova offerta chiamata «Kilobyte Day sale». L'offerta garantisce uno sconto dell'8% su tutti gli accessori per computer il cui prezzo è inferiore a \$128, e del 16% se il prezzo è almeno \$128.
- Scrivere un programma che dato il prezzo di un accessorio, ne fornisce il prezzo scontato.

# Esempio – Vendite - Suggerimenti

- 1) Definire la condizione di diramazione dei rami dell'esecuzione
  - Original price < 128 ?
- 2) Scrivere lo pseudo-codice per il ramo «vero»
  - Discounted price =  $0.92 * \text{original price}$
- 3) Scrivere lo pseudo-codice per il ramo «falso»
  - Discounted price =  $0.84 * \text{original price}$

# Esempio – Vendite - Suggerimenti

- 4) Controllare più volte gli operatori di confronto
  - Test con valori uguali, minori e superiori al confronto (127, 128, 129)
- 5) Rimuovere duplicazioni di codice
  - Discounted price = \_\_\_\_\_ \* original price
- 6) Pensare ai test per controllare entrambi i rami dell'esecuzione
  - Discounted price =  $0.92 * 100 = 92$
  - Discounted price =  $0.84 * 200 = 168$
- 7) Scrivere il codice Python

# Esempio – Vendite - Soluzione

- Apri il file: sale.py
- Esegui il programma diverse volte provando:
  - Prezzi minori di \$128
  - Prezzi maggiori di \$128
  - Prezzo uguale a \$128
  - Prezzi non validi (ad esempio negativi)
- Che risultati ottieni?



sale.py

```
if original_price < 128 :  
    discount_rate = 0.92  
else:  
    discount_rate = 0.84  
discounted_price = discount_rate * original_price
```

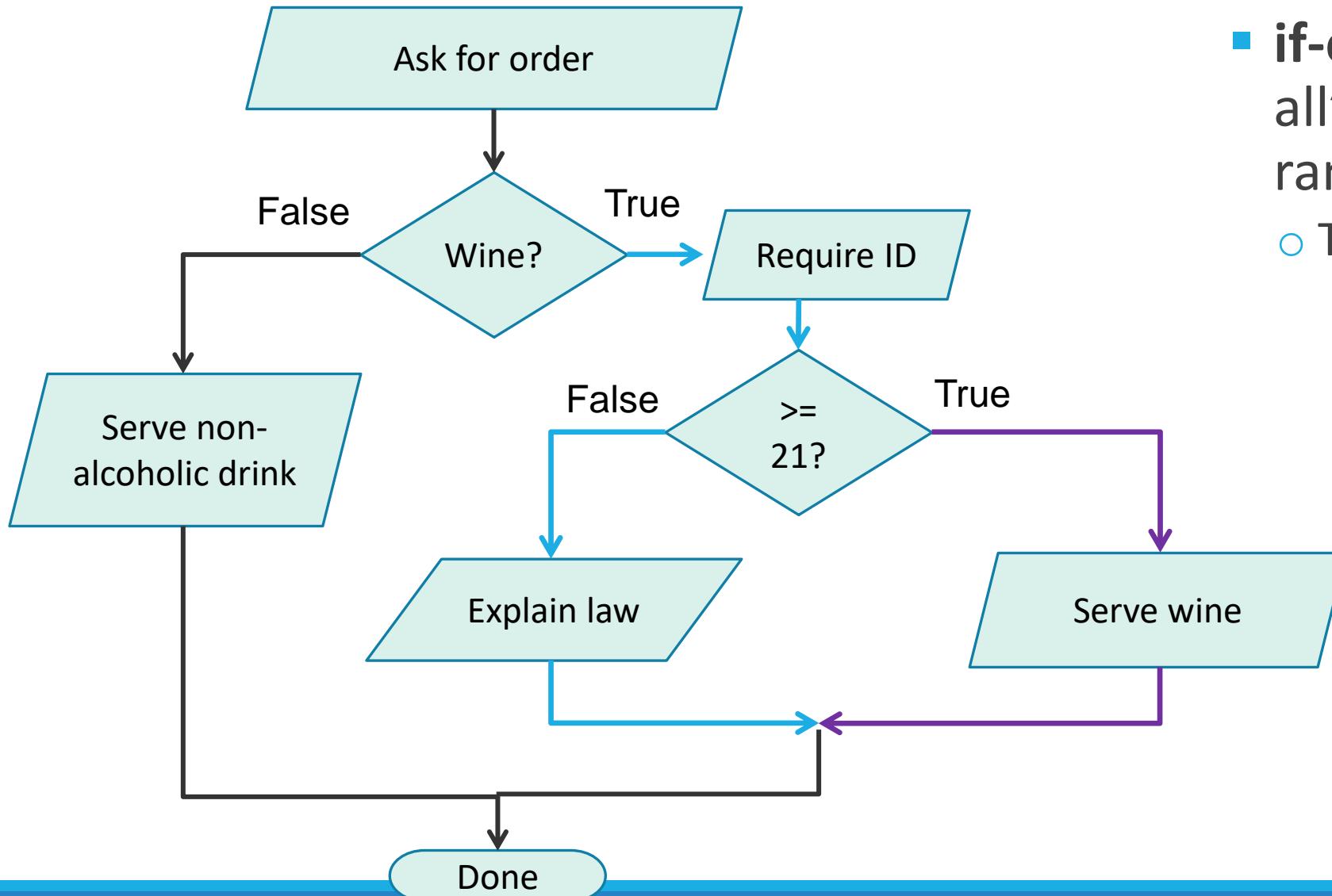
# Decisioni annidate

---



3.3

# Flowchart di un if annidato



- **if-else** annidato all'interno di un ramo **true**.
  - Tre cammini diversi



# Diramazioni annidate

- È possibile annidare un **if** all'interno di una delle due diramazioni di un altro **if**.
- Semplice esempio: Ordinare una bevanda (pseudo codice)

*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*



# Esempio Tasse: if annidati

- Quattro possibilità (rami)

---

<b>Per stato civile “non coniugato” e reddito imponibile</b>	<b>le tasse sono</b>	<b>della somma superiore a</b>
al massimo \$ 32 000	10%	\$ 0
oltre \$ 32 000	\$ 3200 + 25%	\$ 32 000

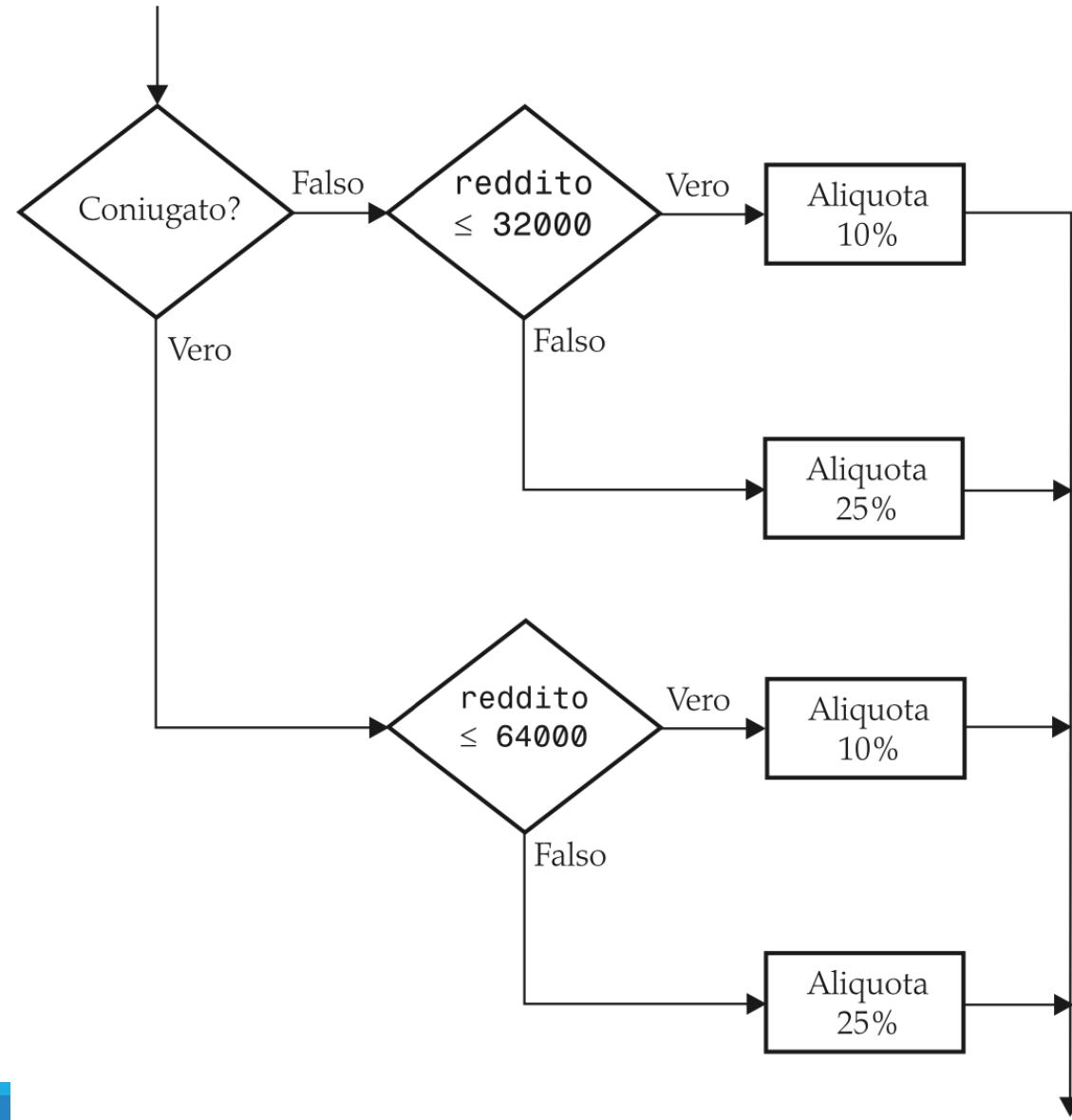
---

<b>Per stato civile “coniugato” e reddito imponibile</b>	<b>le tasse sono</b>	<b>della somma superiore a</b>
al massimo \$ 64 000	10%	\$ 0
oltre \$ 64 000	\$ 6400 + 25%	\$ 64 000

---

# Esempio Tasse - Flowchart

- Quattro rami



# Soluzione

```
1 ##  
2 # This program computes income taxes, using a simplified tax schedule.  
3 #  
4  
5 # Initialize constant variables for the tax rates and rate limits.  
6 RATE1 = 0.10  
7 RATE2 = 0.25  
8 RATE1_SINGLE_LIMIT = 32000.0  
9 RATE1_MARRIED_LIMIT = 64000.0  
10  
11 # Read income and marital status.  
12 income = float(input("Please enter your income: "))  
13 maritalStatus = input("Please enter s for single, m for married: ")  
14  
15 # Compute taxes due.  
16 tax1 = 0.0  
17 tax2 = 0.0  
18  
19 if maritalStatus == "s" :  
20     if income <= RATE1_SINGLE_LIMIT :  
21         tax1 = RATE1 * income  
22     else :  
23         tax1 = RATE1 * RATE1_SINGLE_LIMIT  
24         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)  
25 else :  
26     if income <= RATE1_MARRIED_LIMIT :  
27         tax1 = RATE1 * income  
28     else :  
29         tax1 = RATE1 * RATE1_MARRIED_LIMIT  
30         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)  
31  
32 totalTax = tax1 + tax2  
33
```

 taxes.py

# Soluzione: Single

- Il ramo ‘Vero’ (singolo)
  - Due ulteriori rami all’interno del ramo ‘vero’

```
19 if maritalStatus == "s" :  
20     if income <= RATE1_SINGLE_LIMIT :  
21         tax1 = RATE1 * income  
22     else :  
23         tax1 = RATE1 * RATE1_SINGLE_LIMIT  
24         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
```

# Soluzione: Sposato

- Il ramo ‘Falso’ (sposato)

```
else :  
    if income <= RATE1_MARRIED_LIMIT :  
        tax1 = RATE1 * income  
    else :  
        tax1 = RATE1 * RATE1_MARRIED_LIMIT  
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```

# Esempio Tasse - esecuzione

- Aprire il file: taxes.py
- Eseguire il programma più di una volta utilizzando valori diversi per reddito e stato civile:
  - Si usi un reddito minore di \$32,000
  - Si usi un reddito superiore a \$64,000
  - Si metta “&” come stato civile
- Che risultati si ottengono? Perché?



taxes.py

# Hand-tracing (tracciare a mano)

- L'**Hand-tracing** è un metodo che aiuta a capire se un programma funziona correttamente. Consiste nel tracciare a mano su un foglio l'evoluzione delle variabili.
- Crea una tabella della **variabili** più importanti
  - Tieni traccia dei loro valori (con carta e penna)
- Utilizza il codice ( o lo pseudocodice) per tracciare l'esecuzione del programma
- Utilizza dati di input di esempio che:
  - Generino un risultato prevedibile e noto
  - Testino tutti i rami del codice

# Hand-tracing - Esempio Tasse (1)

tax1	tax2	income	marital status
0	0		

- Setup
  - Tabella delle varabili
  - Valori iniziali

```
6 RATE1 = 0.10
7 RATE2 = 0.25
8 RATE1_SINGLE_LIMIT = 32000.0
9 RATE1_MARRIED_LIMIT = 64000.0
```

```
15 # Compute taxes due.
16 tax1 = 0.0
17 tax2 = 0.0
```

# Hand-tracing - Esempio Tasse (2)

tax1	tax2	income	marital status
0	0	80000	m

- Variabili di Input
  - Letti dall'utente
  - Aggiornare tabella

```
11 # Read income and marital status.  
12 income = float(input("Please enter your income: "))  
13 maritalStatus = input("Please enter s for single, m for married: ")
```

- Because marital status is not "s" we skip to the else on line 25

```
19 if maritalStatus == "s" :
```

```
25 else :
```

# Hand-tracing - Esempio Tasse (3)

- Dato che il reddito non è  $\leq 64000$ , il codice va alla linea 28
  - Aggiornare le variabili delle righe 29 e 30
  - Usare le costanti definite nel codice

```
26 if income <= RATE1_MARRIED_LIMIT :  
27     tax1 = RATE1 * income  
28 else :  
29     tax1 = RATE1 * RATE1_MARRIED_LIMIT  
30     tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```

tax1	tax2	income	marital status
0	0	80000	m
6400	4000		

# Scelte multiple

---



3.4

# Scelte multiple

- Come comportarsi se ci sono più di due alternative (vero/falso)?
- Esempio: determinare gli effetti di un terremoto in base al valore della sua intensità in base alla scala Richter

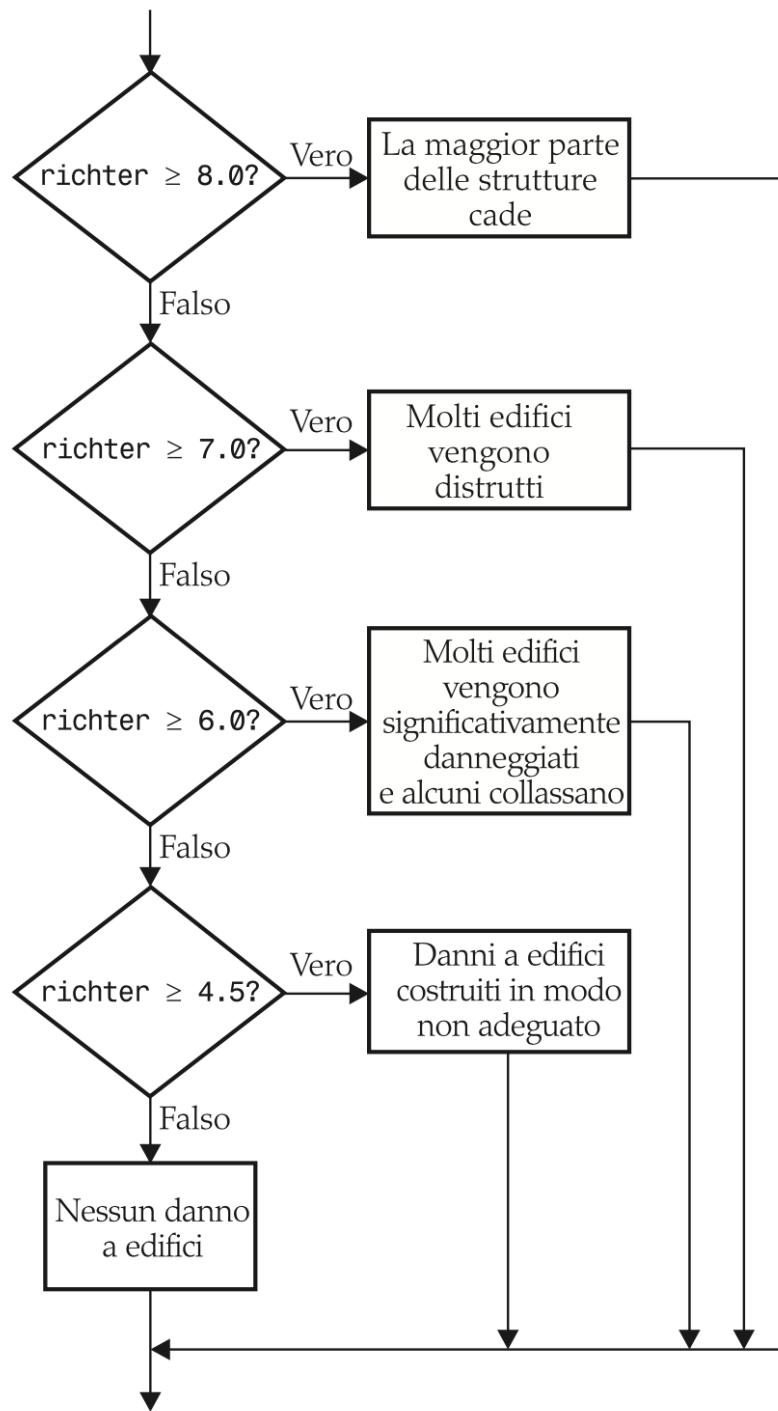
- 8 (o superiore)
- Da 7 a 7.99
- Da 6 a 6.99
- Da 4.5 a 5.99
- Inferiore a 4.5

**Tabella 4**  
Scala Richter

	<b>Valore</b>	<b>Effetto</b>
	8	La maggior parte delle strutture cade
	7	Molti edifici vengono distrutti
	6	Molti edifici vengono significativamente danneggiati e alcuni collassano
	4,5	Danni a edifici costruiti in modo non adeguato

Quando si usano **if** multipli,  
bisogna testare le condizioni più  
generali **dopo** le condizioni più  
specifiche.

# Flow-chart di scelte multiple



# Istruzione `elif`

- Abbreviazione di: `else, if...`
- Appena una delle condizioni di test è vera, il relativo blocco di istruzioni viene eseguito
  - I rimanenti test non vengono valutati
- Se nessun test ha successo, viene eseguito l' `else` finale

# Scelte multiple con `if`, `elif`

```
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")
```

# Che cosa non va con questo codice?

```
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")
```

# Esempio - Terremoto

- Aprire il file: `earthquake.py`
- Eseguire il programma con diversi input



`earthquake.py`

# Istruzione **pass**

- Nello sviluppo del codice, può risultare conveniente scrivere prima lo «scheletro» della logica di funzionamento, per poi «riempire» le istruzioni in un secondo momento
  - Dal punto di vista sintattico, però, non è possibile avere un blocco if:, else:, elif: senza alcuna istruzione
- **pass** è un'istruzione ‘speciale’, che non fa nulla, ma permette la scrittura sintatticamente corretta (in fase di sviluppo), e prima o poi sarà sostituita da codice «vero»

```
if num > 0 :  
    # caso positivo  
    pass  
elif num < 0 :  
    # caso negativo  
    pass  
else :  
    # caso nullo  
    pass
```

# Variabili e Operatori Booleani

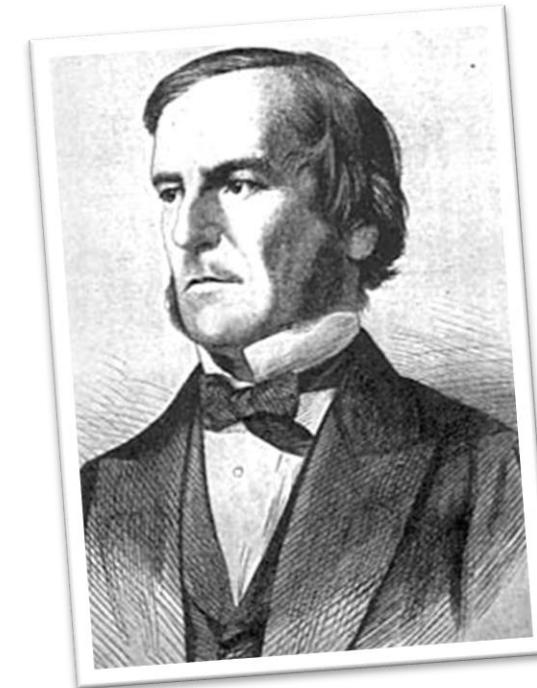
---



3.7

# La logica Booleana degli elaboratori elettronici

- Nel 1847 George Boole introdusse un nuovo tipo di formalismo logico, basato esclusivamente su enunciati che potevano essere veri o falsi
- Gli elaboratori utilizzano la logica booleana



# Variabili Booleane

- Variabili Booleane
  - Una variabile Booleana può solo essere **Vera** o **Falsa**
    - failed = True
  - **bool** è un tipo di dato in Python (può avere i valori **True** e **False**)
  - Una **variabile booleana** è spesso utilizzata come **flag**  (indicatore) proprio perché può essere solo vera o falsa
  - La **condizione** di un'istruzione **if** è un valore Booleano
- Ci sono tre diversi operatori Booleani principali: **and**, **or**, **not**
  - Vengono utilizzati per combinare diverse condizioni Booleane.

# Condizioni multiple: and

- Combinare condizioni Booleane è utile, ad esempio, nel controllare se un valore cade in un certo intervallo
  - ENTRAMBE le condizioni in un **and** devono essere VERE affinché il risultato sia VERO

```
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

# Ricordarsi una condizione

- Le variabili Booleane possono essere usate per «ricordarsi» una certa condizione, e testarla più avanti.

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

```
isLiquid = temp > 0 and temp < 100  
# Boolean value True/False  
  
if isLiquid :  
    print("Liquid")
```

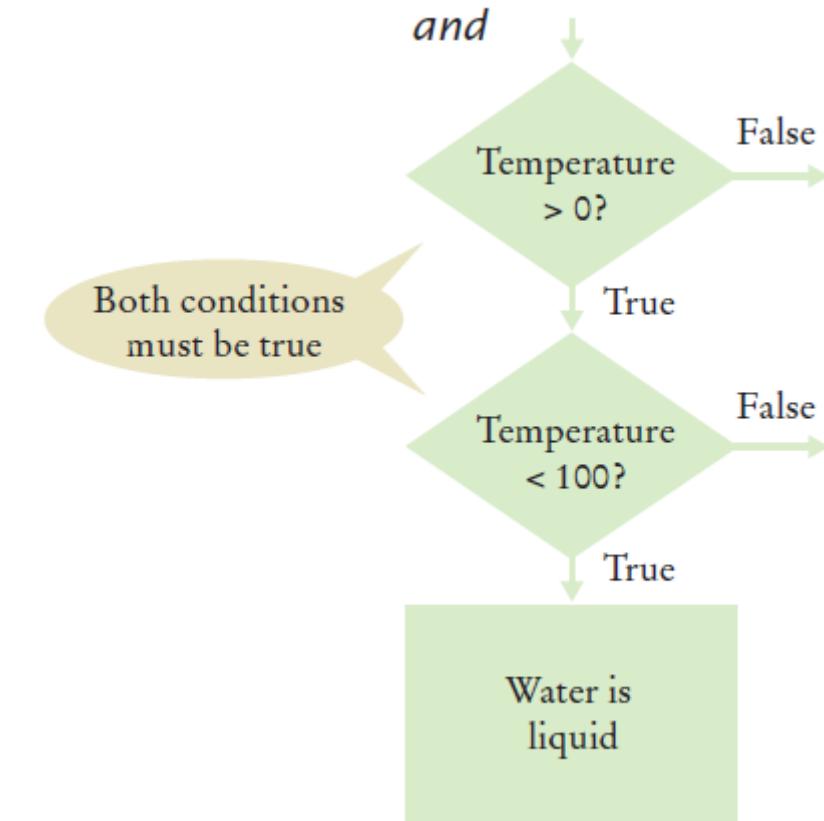
# Operatori di confronto concatenati

- Linguaggio corrente: “Se la *temperatura* è tra 0 e 100 ...”
- Math:  $0 \leqslant \text{temp} \leqslant 100$
- Python: `0 <= temp and temp <= 100`
- Si può anche scrivere: `0 <= temp <= 100`
  - Python permette l’uso di *operatori di confronto concatenati*
  - La maggior parte degli altri linguaggi di programmazione non lo permette

# Flowchart dell'operatore «and»

- Si chiama generalmente ‘range checking’
  - Utilizzato per verificare se un valore cade in un certo intervallo

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



# Condizioni multiple: or

- Si utilizza l'**or** se **una sola delle condizioni** deve essere Vera affinché la condizione multipla sia Vera:

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

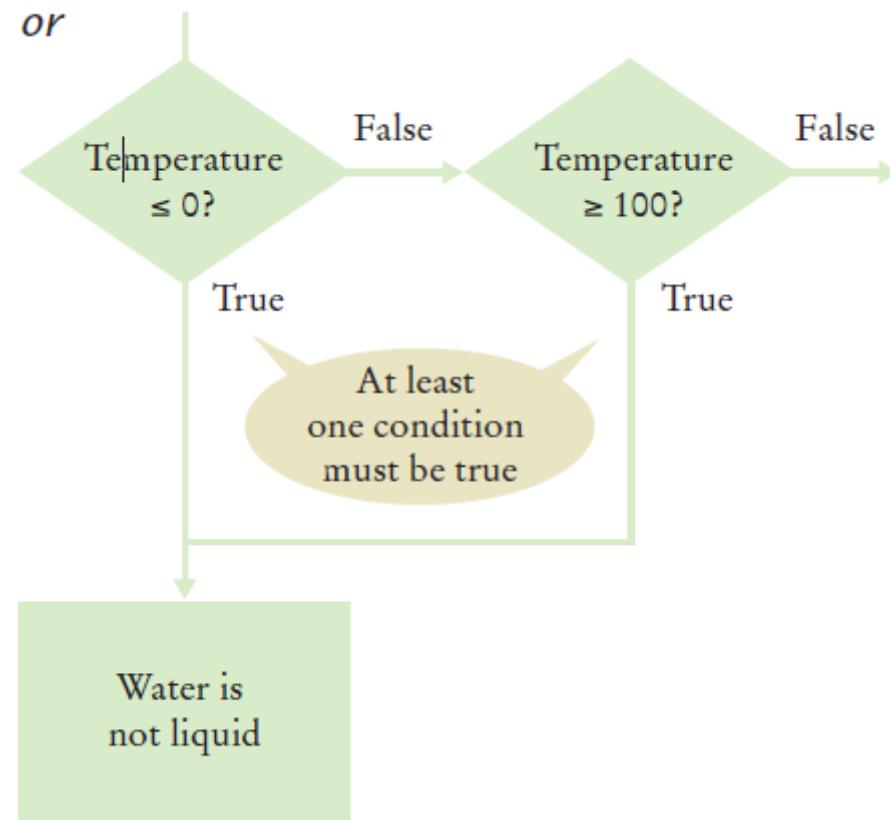
- Se una qualunque delle condizioni è vera
  - Il risultato è Vero

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

# Flowchart dell'or

- Un'altra forma di 'range checking'
  - Controlla se il valore è FUORI dall'intervallo

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



# L'operatore **not**

- Se c'è bisogno di **invertire** una variabile booleana o il risultato di un confronto, si deve utilizzare il **not**

```
if not attending or grade < 60 :  
    print("Drop?")
```

```
if attending and not(grade < 60) :  
    print("Stay")
```

A	not A
True	False
False	True

- Per chiarezza, provare a rimpiazzare il **not** con altri operatori

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

# Nota

```
if not ( a == b ):  
# è equivalente a  
if a != b :
```



# Esempio - Confronti

- Aprire il file: compare2.py
- Eseguire il programma con diversi input



compare2.py

# Esempio di Operatori Booleani

Espressione	Valore	Commento
<code>0 &lt; 200 and 200 &lt; 100</code>	False	Soltanto la prima condizione è vera.
<code>0 &lt; 200 or 200 &lt; 100</code>	True	La prima condizione è vera.
<code>0 &lt; 200 or 100 &lt; 200</code>	True	L'or non è una verifica di tipo “o o”: se entrambe le condizioni sono vere, il risultato è vero.
<code>0 &lt; x and x &lt; 100 or x == -1</code>	<code>(0 &lt; x and x &lt; 100) or x == -1</code>	L'and ha una precedenza più elevata dell'or (Appendice A).
<code>not (0 &lt; 200)</code>	False	<code>0 &lt; 200</code> è True, quindi la sua negazione è False.
<code>frozen == True</code>	frozen	Non c'è bisogno di confrontare una variabile booleana con il valore True.
<code>frozen == False</code>	not frozen	È più chiaro usare not che confrontare con False.

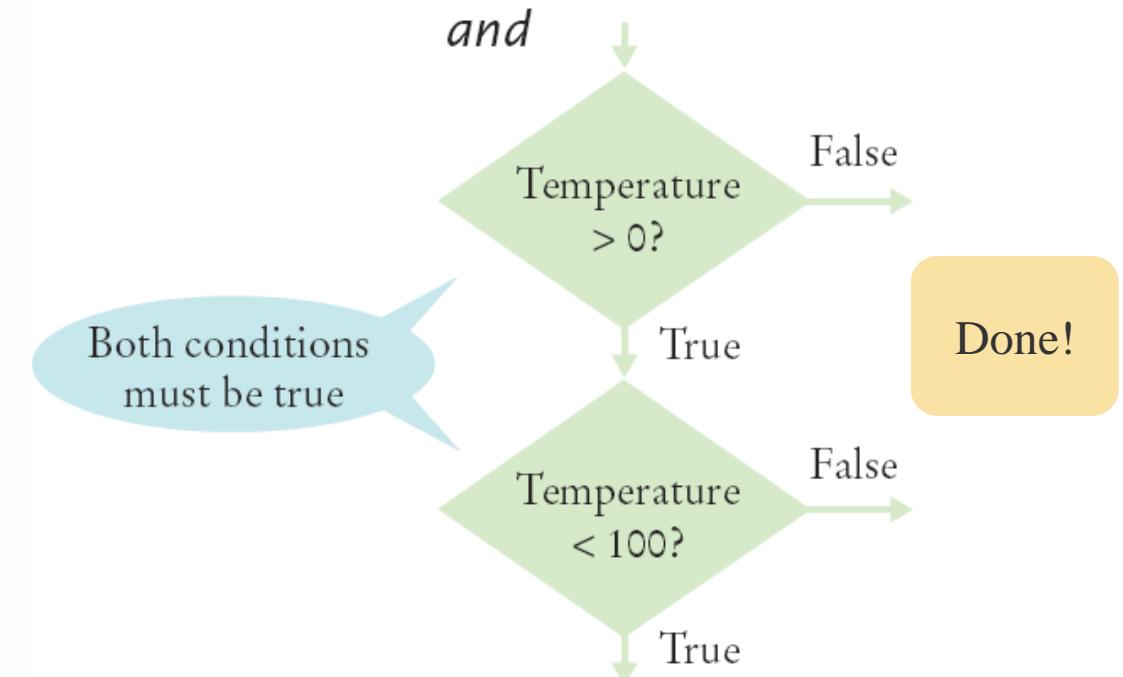
# Errori comuni con le Condizioni Booleane

- Confondere **and** con **or**
  - È un errore sorprendentemente comune confondere **and** e **or**.
  - Un valore cade tra 0 e 100 se vale almeno 0 **and** al più 100.
  - È fuori dall'intervallo se è minore di 0 **or** maggiore di 100.
- Non c'è una regola fissa. Bisogna **pensarci con attenzione**.

# Valutazione delle condizioni: and

- Condizioni multiple sono valutate da sinistra a destra
  - Se la parte sinistra di una condizione **and** è falsa, a cosa serve valutare la parte destra?

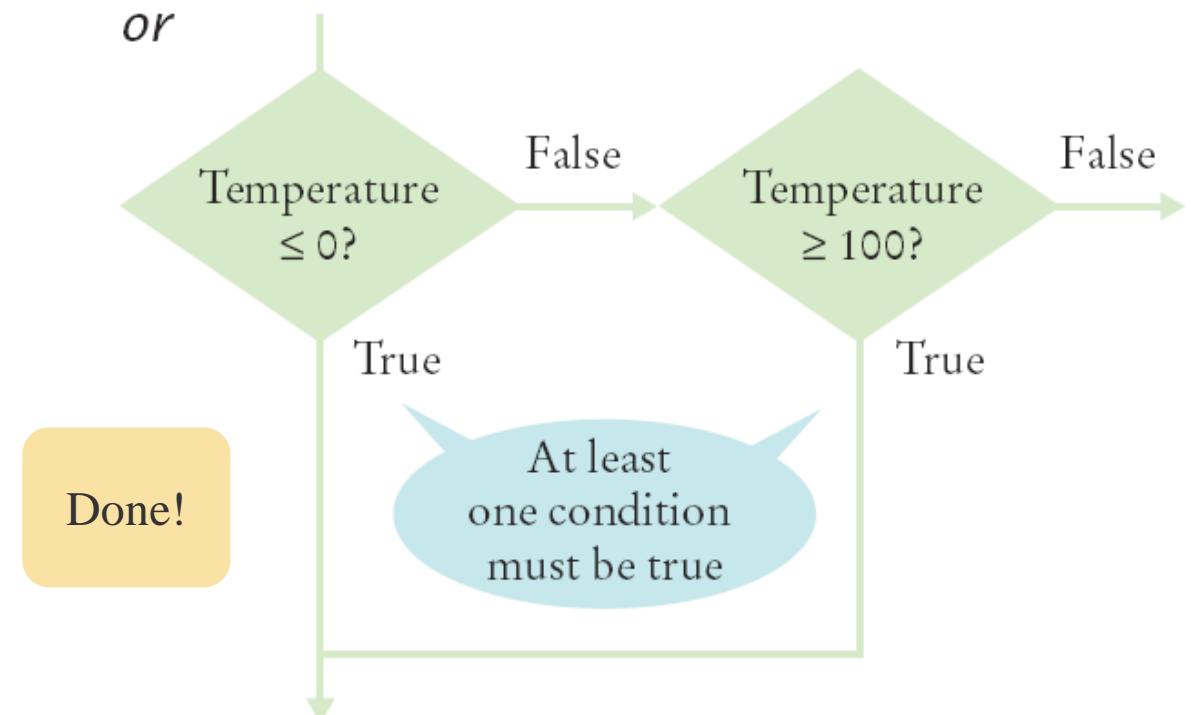
```
if temp > 0 and temp < 100 :  
    print("Liquid")
```



# Valutazione delle condizioni: or

- Se la parte sinistra dell'**or** è vera, a cosa serve valutare la parte destra?

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



# Qualche proprietà

- Commutativa:
  - $A \text{ and } B = B \text{ and } A$
  - $A \text{ or } B = B \text{ or } A$
- Associativa:
  - $A \text{ and } B \text{ and } C = (A \text{ and } B) \text{ and } C = A \text{ and } (B \text{ and } C)$
  - $A \text{ or } B \text{ or } C = (A \text{ or } B) \text{ or } C = A \text{ or } (B \text{ or } C)$
- Distributiva:
  - $A \text{ and } (B \text{ or } C) = A \text{ and } B \text{ or } A \text{ and } C$
  - $A \text{ or } (B \text{ and } C) = (A \text{ or } B) \text{ and } (A \text{ or } C)$

# La legge di De Morgan

- La legge di De Morgan spiega come negare una combinazione di **and** e **or**:
  - `not(A and B)` è la stessa cosa di `not(A) or not(B)`
  - `not(A or B)` è la stessa cosa di `not(A) and not(B)`
- Esempio: Una spedizione per AK (Alaska) e HI (Hawaii) è più costosa

```
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
```

# Analisi di Stringhe

---



3.8

# Analisi di Stringhe – L'operatore `in`

- È spesso necessario analizzare una stringa
- Esempio: sapere se una stringa **contiene al suo interno una certa sottostringa**.
  - Dato il seguente codice:  
`name = "John Wayne"`
  - L'espressione  
`"Way" in name`
  - Ritorna **Vero** perché la sottostringa "Way" appare all'interno della variabile Stringa `name`
- L'operatore `not in` è l'inverso dell'operatore `in`

# Sottostringhe: Suffissi

- Supponiamo di avere un nome di un file in una stringa, e di dover verificare che abbia la corretta estensione

```
if filename.endswith(".html") :  
    print("This is an HTML file.")
```

- Il metodo **endswith()** applicato a una stringa, ritorna «Vero» se la stringa termina con la sottostringa fornita come parametro. E «Falso» altrimenti.

# Operazioni per analizzare sottostringhe

**Tabella 6**

Operazioni di verifica  
per sottostringhe

<b>Operazione</b>	<b>Descrizione</b>
<code>sottostringa in s</code>	Restituisce True se la stringa <i>s</i> contiene <i>sottostringa</i> , altrimenti False.
<code>s.count(ssottostringa)</code>	Restituisce il numero di ricorrenze non sovrapposte di <i>sottostringa</i> nella stringa <i>s</i> .
<code>s.endswith(sottostringa)</code>	Restituisce True se la stringa <i>s</i> termina con <i>sottostringa</i> , altrimenti False.
<code>s.find(sottostringa)</code>	Restituisce il più basso valore dell'indice nella stringa <i>s</i> dove inizia una occorrenza di <i>sottostringa</i> , oppure -1 se <i>sottostringa</i> non ricorre in <i>s</i> .
<code>s.startswith(sottostringa)</code>	Restituisce True se la stringa <i>s</i> inizia con <i>sottostringa</i> , altrimenti False.

# Metodi: Testare le caratteristiche di una stringa

**Tabella 7**

Metodi per verificare alcune caratteristiche di una stringa

<b>Metodo</b>	<b>Descrizione</b>
<code>s.isalnum()</code>	Restituisce <code>True</code> se la stringa <i>s</i> è costituita da sole lettere o cifre e contiene almeno un carattere, altrimenti <code>False</code> .
<code>s.isalpha()</code>	Restituisce <code>True</code> se la stringa <i>s</i> è costituita da sole lettere e contiene almeno un carattere, altrimenti <code>False</code> .
<code>s.isdigit()</code>	Restituisce <code>True</code> se la stringa <i>s</i> è costituita da sole cifre e contiene almeno un carattere, altrimenti <code>False</code> .
<code>s.islower()</code>	Restituisce <code>True</code> se la stringa <i>s</i> contiene almeno una lettera e tutte le lettere della stringa sono minuscole, altrimenti <code>False</code> .
<code>s.isspace()</code>	Restituisce <code>True</code> se la stringa <i>s</i> è costituita da soli caratteri di spaziatura ( <i>whitespace</i> , cioè spazi, caratteri <i>tab</i> e caratteri <i>newline</i> ) e contiene almeno un carattere, altrimenti <code>False</code> .
<code>s.isupper()</code>	Restituisce <code>True</code> se la stringa <i>s</i> contiene almeno una lettera e tutte le lettere della stringa sono maiuscole, altrimenti <code>False</code> .

# Confronto e Analisi di stringhe

Espressione	Valore	Commento
"John" == "John"	True	L'operatore == viene usato anche per verificare l'uguaglianza tra due stringhe.
"John" == "john"	False	Perché due stringhe siano uguali devono essere identiche: una lettera "J" maiuscola è diversa da una lettera "j" minuscola.
"john" < "John"	False	Sulla base dell'ordinamento lessicografico delle stringhe, una "J" maiuscola precede una "j" minuscola, per cui la stringa "john" segue la stringa "John" (si veda la sezione Argomenti avanzati 3.2).
"john" in "John Johnson"	False	La sottostringa "john" deve ricorrere in modo esatto nella stringa.
name = "John Johnson" "ho" not in name	True	La stringa non contiene la sottostringa "ho".
name.count("oh")	2	Nel conteggio sono comprese tutte le sottostringhe non sovrapposte.
name.find("oh")	1	Nella stringa, trova la posizione (o indice) in cui ricorre per la prima volta la sottostringa "oh".
name.find("ho")	-1	La stringa non contiene la sottostringa "ho".
name.startswith("john")	False	La stringa inizia con "John" ma una "J" maiuscola non è uguale a una "j" minuscola.
name.isspace()	False	La stringa contiene caratteri che non sono di spaziatura.
name.isalnum()	False	La stringa contiene anche spazi.
"1729".isdigit()	True	La stringa contiene soltanto cifre.
"-1729".isdigit()	False	Un segno negativo non è una cifra.

# Esempio - Sottostringhe

- Aprire il file: substrings.py
- Eseguire il programma e testarlo con diverse stringhe e sottostringhe



substrings.py

# Validare una Stringa

- Negli Stati Uniti, i numeri di telefono sono fatti di 3 parti: area code exchange, and line number, solitamente scritti nella forma:

(###)###-####

# Validare una Stringa (codice)

- È possibile esaminare una stringa per verificare che contenga un numero di telefono formattato correttamente (es.: (703)321-6753)

```
valid = ( len(string) == 13  
          and string[0] == "("  
          and string[4] == ")"  
          and string[8] == "-"  
          and string[1:4].isdigit()  
          and string[5:8].isdigit()  
          and string[9:13].isdigit() )
```

# Validazione degli Input

---



3.9

# Validazione degli Input

- Accettare un input senza controllarlo è in generale pericoloso
  - Si consideri il programma dell'ascensore:
    - Si assuma che il pannello dell'ascensore abbia solo i bottoni da 1 a 20 (ma senza il 13).

# Validazione degli Input

- I seguenti input non sono validi:

- Il numero 13

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

- Zero o un valore negativo
  - Un numero maggiore di 20

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

- Un input che non sia una sequenza di cifre numeriche:
    - Per verificare che i valori siano interi o floating point si usa il meccanismo delle **eccezioni** che verrà trattato nel Capitolo 7.

# Soluzione

```
1  ##
2  # This program simulates an elevator panel that skips the 13th floor,
3  # checking for input errors.
4  #
5
6  # Obtain the floor number from the user as an integer.
7  floor = int(input("Floor: "))
8
9  # Make sure the user input is valid.
10 if floor == 13 :
11     print("Error: There is no thirteenth floor.")
12 elif floor <= 0 or floor > 20 :
13     print("Error: The floor must be between 1 and 20.")
14 else :
15     # Now we know that the input is valid.
16     actualFloor = floor
```



elevatorsim2.py

# Simulazione dell'Ascensore

- Aprire il file: elevatorsim2.py  elevatorsim2.py
- Testare il programma con vari input, tra cui:
  - 12
  - 14
  - 13
  - 1
  - 0
  - 23
  - 19

# Regola generale

- Non fidarsi MAI dell'input dell'utente (non assumere mai che sia corretto né sensato)
- Quando si legge un input dall'utente, **controllare sempre che contenga valori accettabili**, prima di continuare l'esecuzione
- Se il valore letto non è accettabile, stampare un messaggio e :
  - Richiedere di inserire nuovamente il valore (vedi i Cicli, Capitolo 4)
  - Uscire del programma:

```
from sys import exit  
exit("Valore non accettabile")
```

È impossibile rendere tutto il  
programma a prova di stupido perchè  
gli stupidi sono molto ingegnosi....  
*(Unattributed variant to Murphy's Law)*

# Sommario

---

# Sommario: istruzione **if**

- L'istruzione **if** permette a un programma di eseguire azioni diverse a seconda dei dati che sta processando.
- Gli operatori relazionali (`<` `<=` `>` `>=` `==` `!=`) vengono utilizzati per confrontare numeri e stringhe.
- Le stringhe sono confrontate in ordine alfabetico.
- **if** multipli possono essere combinati per valutare decisioni complesse.
- Quando si utilizzano **if** multipli, si devono prima testare le condizioni più generali e poi quelle più specifiche.

# Sommario: booleani

- Il tipo di dato **booleano** ha solo due possibili valori, **Vero** e **Falso**.
  - Python ha due operatori booleani che permettono di combianre condizioni booleane: **and** e **or**.
  - Per invertire una condizione, si usa l'operatore **not**.
  - Gli operatori **and** e **or** valutano le condizioni da sinistra a destra:
    - Appena il valore Vero viene determinato, le condizioni rimanenti non sono valutate.
  - La legge di De Morgan spiega come negare condizioni multiple con **and** e **or**.

# Sommario: input/output

- Si utilizzi la funzione `input()` per leggere dati dal tastiera.
- Se l'input non è una stringa, si usino le funzioni `int()` o `float()` per convertire il testo inserito in numero
- Si utilizzino le `f`-String per una formattazione dell'output più agevole
- Si utilizzino le definizioni di formato per specificare come i valori dovrebbero essere formattati (quando si stampano).