Laboratorio di programmazione

Python

A.A. 2020-2021

Lezione 2

Turtle program

Utilizziamo **turtle**, il **modulo** Python per disegnare su schermo

- Python è composto da moduli
- Ogni modulo contiene delle istruzioni già scritte
- I moduli si *importano* nei programmi con lo *statement* **import**
- le istruzioni del modulo saranno disponibili nel **namespace** del modulo (il suo nome)

Il primo programma

Create un file `turtle_program.py` e scrivete queste linee

```
import turtle                # Importo modulo turtle
window = turtle.Screen()     # Creo una finestra dove lavorare
raffaello = turtle.Turtle() # Creo una tartaruga e la assegno alla variabile "raffaello"

raffaello.forward(50)        # Dico a "raffaello" di andare avanti di 50 passi
raffaello.left(90)           # Dico a "raffaello" di girare a sinistra di 90 gradi
raffaello.forward(30)        # Dico a "raffaello" di andare avanti di 30 passi

window.mainloop()            # Attende che l'utente chiuda la finestra di gioco
```

Salvate il file ed eseguite il codice dalla *tab* **SHELL**

Capiamo il programma

```
import turtle
```

Abbiamo chiesto a Python di importare il modulo chiamato `turtle`

- il modulo contiene 2 nuovi tipi di dato: `Turtle` e `Screen`
- i nuovi tipi di dato sono nel **namespace** `turtle`

Capiamo il programma

```
window = turtle.Screen()
raffaello = turtle.Turtle()
```

Abbiamo creato un **oggetto** di tipo (cioè della **classe**) `Screen()`

- creando questo oggetto si e aperta una **window** che contiene una **canvas** per disegnare
- il tipo esiste solo nel suo namespace, quindi va chiamato per esteso: `turtle.Screen()`

Abbiamo creato un **oggetto** di tipo (cioè della **classe**) `.Turtle()`

- l'oggetto `Turtle` è assegnato alla variabile `raffaello`: nel programma ci riferiremo a "raffaello" come specifica tartaruga con cui giocare
- il tipo esiste solo nel suo namespace, quindi va chiamato per esteso: `turtle.Turtle()`

Capiamo il programma

```
window = turtle.Screen()
raffaello = turtle.Turtle()
```

Abbiamo creato un **oggetto** di tipo (cioè della **classe**) `Screen()`

- creando questo oggetto si e aperta una **window** che contiene una **canvas** per disegnare
- il tipo esiste solo nel suo namespace, quindi va chiamato per esteso: `turtle.Screen()`

Abbiamo creato un **oggetto** di tipo (cioè della **classe**) `.Turtle()`

- l'oggetto `Turtle` è assegnato alla variabile `raffaello`: nel programma ci riferiremo a "raffaello" come specifica tartaruga con cui giocare
- il tipo esiste solo nel suo namespace, quindi va chiamato per esteso: `turtle.Turtle()`

Capiamo il programma

```
raffaello.forward(50)
raffaello.left(90)
raffaello.forward(30)
```

Diciamo alla tartaruga *raffaello* di muoversi: avanti, ruotare a sinistra, avanti

- i movimenti sono fatti **chiamando dei metodi** di **raffaello**: `forward` e `left`
- qualunque oggetto `Turtle` conosce questi **metodi**, ma noi li applichiamo a *raffaello*

Capiamo il programma

```
window.mainloop()
```

Diciamo nell'ultima linea all'oggetto `Screen` che si chiama *window* di attendere un evento esterno (click mouse, tastiera, ...)

- la richiesta è fatta con un **metodo** di `window`
- tutti gli oggetti `Screen` conoscono questo metodo, ma noi vogliamo che la nostra *window* aspetti

---

Il programma "finisce" quando l'utente chiude la finestra (o con un `KeybordInterrupt`)

Esercizi

1. Fate disegnare a raffaello un quadrato utilizzando `forward` e `left`

Metodi

In un programma, un **oggetto** può avere

- **metodi**, cioè azioni che può fare
- **attributi**, cioè proprietà

Ad esempio i tipi `Turtle` possono spostarsi in avanti con il **metodo** `forward`:

```
raffaello.forward(10)
```

oppure possono cambiare il proprio **attributo** colore con:

```
raffaello.color('red')
```

dopo questo comando la `Turtle` *raffaello* scriverà in rosso.
Per l'oggetto `Turtle` il colore, la sua posizione iniziale, la dimensione della "penna", ecc...sono tutti **attributi che ne definiscono lo stato**

La descrizione di come è fatto qualunque oggetto in Python si può avere con la funzione *built-in* `help()`

```
help(raffaello)
help(window)
```

Esercizi

1. cambiate il colore della tartaruga e dello sfondo della finestra (con il metodo `bgcolor()`)
2. chiedete all'utente il colore della tartaruga e dello sfondo usando la funzione `input()`
3. fate disegnare un triangolo alla tartaruga

Istanze

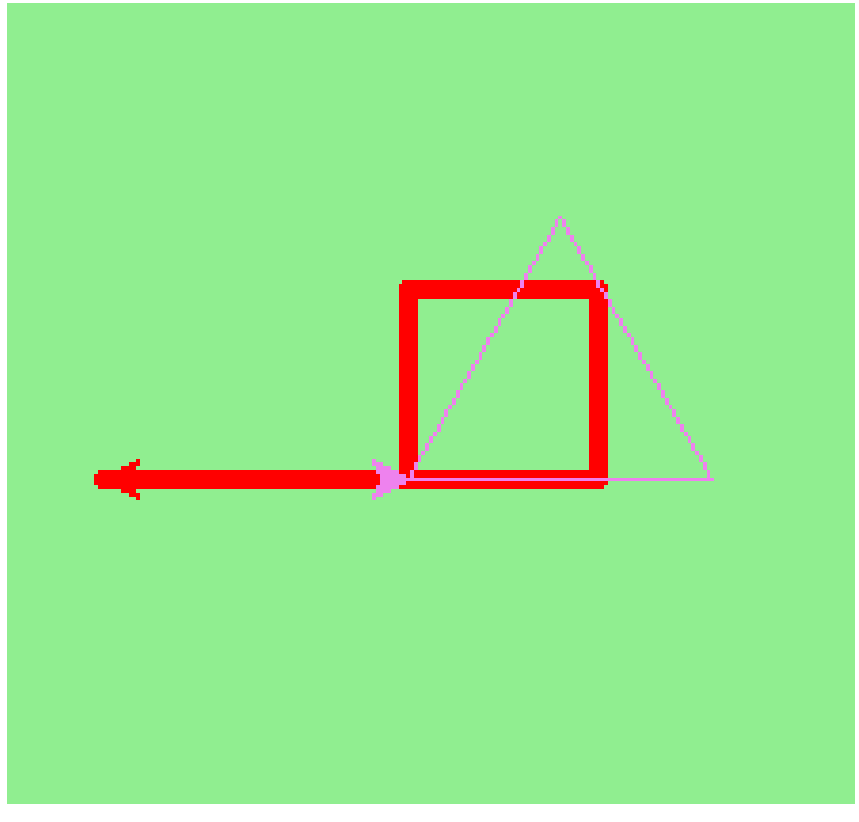
Nel `turtle_program` possiamo avere diversi oggetti di tipo `Turtle` ognuno con un suo *stato* e che si occupa di fare un disegno diverso.
Ogni **oggetto** che creiamo è detto **istanza** della classe `Turtle`.

Ad esempio potremmo avere 2 tartarughe *raffaello* e *donatello* che partendo dallo stesso punto possono muoversi in direzioni diverse con colori e tratti di dimensioni diverse:

Istanze

Nel `turtle_program` possiamo avere diversi oggetti di tipo `Turtle` ognuno con un suo *stato* e che si occupa di fare un disegno diverso.
Ogni **oggetto** che creiamo è detto **istanza** della classe `Turtle`.

Ad esempio potremmo avere 2 tartarughe *raffaello* e *donatello* che partendo dallo stesso punto possono muoversi in direzioni diverse con colori e tratti di dimensioni diverse:



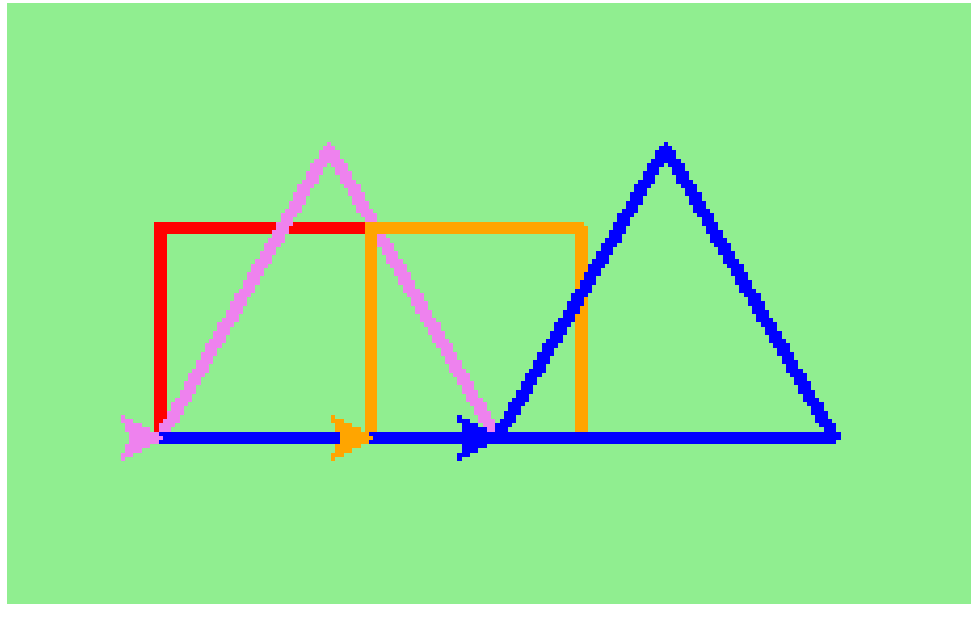
```python
import turtle

window = turtle.Screen()                # istanza "window" di Screen
window.bgcolor("lightgreen")            # imposto lo stato della "window"
window.title("Raffaello & Donatello")

raffaello = turtle.Turtle()             # Istanza di Turtle chiamata raffaello
raffaello.color("red")                  # attributi per lo stato di raffaello
raffaello.pensize(5)

donatello = turtle.Turtle()             # Istanza di Turtle chiamata donatello
donatello.color("violet")               # attributi per lo stato di donatello

raffaello.forward(50)                   # movimenti per disegnare quadrato
raffaello.left(90)                      # con raffaello
raffaello.forward(50)
raffaello.left(90)
raffaello.forward(50)
raffaello.left(90)
raffaello.forward(50)
raffaello.left(90)

donatello.forward(80)                   # movimenti per disegnare un triangolo
donatello.left(120)                     # con donatello
donatello.forward(80)
donatello.left(120)
donatello.forward(80)
donatello.left(120)

raffaello.right(180)                    # giro e sposto dall'origine raffaello
raffaello.forward(80)                   #

window.mainloop()                       # aspetto che l'utente chiuda "window"
```

Considerazioni

1. Disegnare queste figure ci ha fatto scrivere **comandi ripetitivi**
2. ...e se avessimo dovuto disegnare un cerchio?
3. Le ultime rotazioni non erano "necessarie" ma hanno riportato le tartarughe nella posizione iniziale
    - è più semplice se poi ho bisogno di "comporre più movimenti"
4. Spostare raffaello dall'origine è una operazione che abbiamo "staccato" dal blocco "disegna"
    - abbiamo diviso il programma in 2 blocchi logici: "disegna" e "spostati"
5. Le linee bianche non sono "a caso", ma rappresentano come mentalmente abbiamo diviso il programma
6. ...ancora più utile sarebbe descrivere con commenti ogni blocco, per seguire meglio il ragionamento
7. Ogni istanza `Turtle()` conosce tutti i comandi, ma ha un suo comportamento indipendente

Esercizi

1. Modificate il programma per avere 4 tartarughe che disegnano uno accanto all'altro 2 quadrati e 2 triangoli:



Loop Control

Disegnare il quadrato ha richiesto molte istruzioni ripetitive. Farlo 2 volte...
..Ma poteva andare peggio: se fosse stato un ottagono?

Nei programmi, se individuiamo un gruppo di istruzioni che può essere ripetuto, possiamo usare delle **strutture di controllo di loop** per semplificarci la vita. In Python sono:

- `for` loop, quando il numero di ripetizioni è noto in partenza
- `while` loop, quando il numero di ripetizioni non è noto ma c'è una **condizione** di termine

---

Quando si progetta un programma è fondamentale costruzione blocchi di istruzioni che possano essere *riutilizzati* più volte perchè:

1. ci risparmia fatica
2. come sopra
3. come sopra

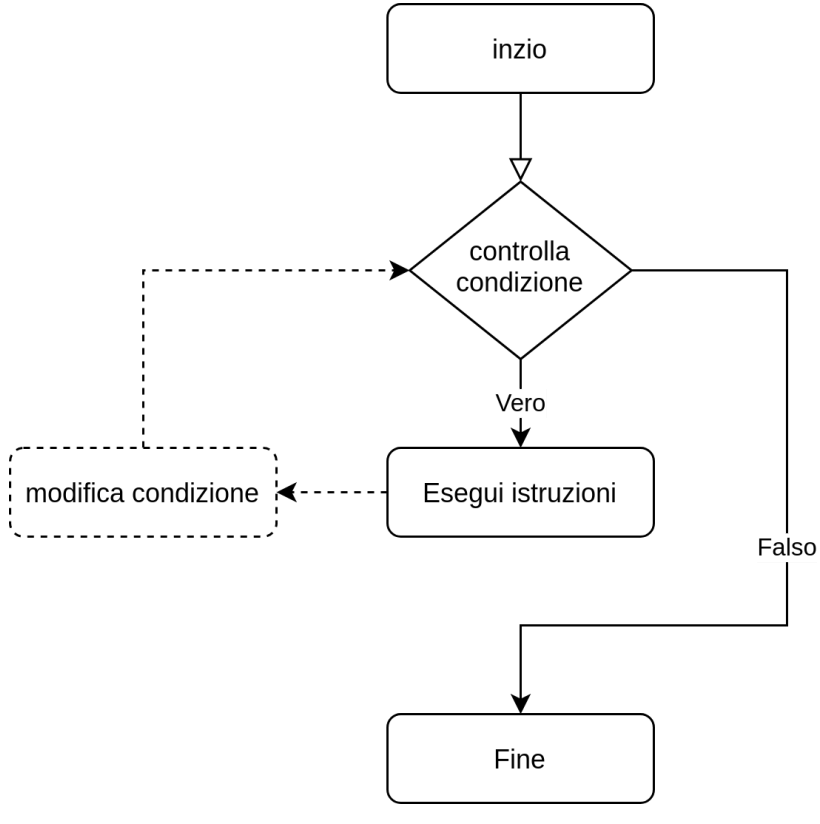
... scrivere programmi *è un lavoro per pigri*

For loop

Mentre il programma è in esecuzione, l'interprete tiene traccia di quale è la prossima istruzione deve eseguire (quella sotto...).
Con un **loop** `for` si può altrare il flusso dall'alto in basso e **tornare indietro per ripetere le operazioni**

For loop

Mentre il programma è in esecuzione, l'interprete tiene traccia di quale è la prossima istruzione deve eseguire (quella sotto...).
Con un **loop** `for` si può altrare il flusso dall'alto in basso e **tornare indietro per ripetere le operazioni**



In Python i **loop for** si costruiscono con
la keyword `for`, una **variabile**, la keyword `in`, una **sequenza da iterare** e token `:`

```
for X in SEQUENZA :
    # istruzioni da ripetere
    ISTRUZIONI

# fine del blocco ripetuto
```

- la SEQUENZA da iterare è un qualunque tipo di Python usato per raccolte di valori (**collections**, li vedremo)
- il **blocco** di istruzioni da ripetere è indentato
- il blocco di istruzioni è ripetuto **solo se la condizione è soddisfatta**.
- il blocco di istruzioni da ripetere (**body**) termina rientrando dall'indentazione

Semplifichiamo il *turtle program*

Per disegnare un quadrato dobbiamo ripetere lo stesso **blocco** di istruzioni 4 volte:

```
# loop for inizia con keyword for, blocco istruzioni con :
for i in [0, 1, 2, 3] :
    # le istruzioni dentro al 'blocco' da ripetere sono indentate
    raffaello.forward(50)
    raffaello.left(90)

# qui il blocco da ripetere finisce perchè rientra l'indentazione
```

- Le istruzioni da `forward` e `left` da ripetere sono **indentate** rispetto al resto del programma
- La raccolta di valori `[0, 1, 2, 3]` serve a ripetere 4 volte il **blocco**
- La variabile `i` viene aggiornata con i valori della raccolta ad ogni **iterazione**

L'istruzione

```
for i in [0, 1, 2, 3] :
```

si traduce:

```
per i che assume in ordine i valori della raccolta [0, 1, 2, 3] fai:
```

> ***In Python si inizia a contare da `0`***

Range

Nel `turtle_program`, `i` non è utilizzata e potrebbe assumere qualunque **tipo** di valore. Ma per capire cosa stiamo facendo è comodo usare gli interi:

```
for i in [0, 1, 2, 3] :
```

In Python si usa così spesso il **loop su numeri interi** che esiste una funzione *built-in* `range()` che li calcola:

Range

Nel `turtle_program`, `i` non è utilizzata e potrebbe assumere qualunque **tipo** di valore. Ma per capire cosa stiamo facendo è comodo usare gli interi:

```
for i in [0, 1, 2, 3] :
```

In Python si usa così spesso il **loop su numeri interi** che esiste una funzione *built-in* `range()` che li calcola:

In [1]:
```
range(4)
```

Out[1]:
```
range(0, 4)
```

Range

Nel `turtle_program`, `i` non è utilizzata e potrebbe assumere qualunque **tipo** di valore. Ma per capire cosa stiamo facendo è comodo usare gli interi:

```
for i in [0, 1, 2, 3] :
```

In Python si usa così spesso il **loop su numeri interi** che esiste una funzione *built-in* `range()` che li calcola:

In [1]:
```
range(4)
```

Out[1]:
```
range(0, 4)
```

> *Utilizzo:* `range(start, stop_escluso, step)`. *Lo start ha valore default 0 che si può omettere*

quindi possiamo scrivere

```
# inizio loop or con keyword for, inizio il blocco con :
for i in range(4):
    raffaello.forward(50)
    raffaello.left(90)
```

per ripetere **4** volte le istruzioni.

> *In Python si inizia a contare da* `0`*, quindi* `i` *non avrà mai il* ***valore*** *4 nel loop*

---

quindi possiamo scrivere

```
# inizio loop or con keyword for, inizio il blocco con :
for i in range(4):
    raffaello.forward(50)
    raffaello.left(90)
```

per ripetere **4** volte le istruzioni.

> *In Python si inizia a contare da `0`, quindi `i` non avrà mai il **valore** 4 nel loop*

Ma anche

```
for c in ['red', 'purple', 'yellow', 'blue']:
```

sarebbe stato valido: `c` avrebbe assunto 4 valori `string`, uno ad ogni **iterazione**

For loop su raccolte

Questo loop for è valido e si ripete 4 volte:

```
for c in ['red', 'purple', 'yellow', 'blue']:
    raffaello.forward(50)
    raffaello.left(90)
```

allora cosa succede se scriviamo:

```
for c in ['red', 'purple', 'yellow', 'blue']:
    raffaello.color(c)     # usiamo il valore di c assegnato ad ogni iterazione
    raffaello.forward(50)
    raffaello.left(90)
```

Esercizi

1. Scrivete un programma che visualizza la scritta "turtles" 100 volte.

2. Scrivere un programma come il punto 1. ma i numeri vengono chiesti 10 volte all'utente con `input()`

3. Scrivete un programma che per ogni elemento di questo gruppo `xs = [12, 10, 32, 3, 66, 17, 42, 99, 20]`:

    - scrive su schermo "numeri" e poi ognuno dei numeri andando a capo ad ogni numero (usate loop for)
    - scrive "quadrati" e poi per ognuno scrive il quadrato del numero andando a capo ad ogni numero (usate loop for)
    - scrive "totale" e la somma di tutti i numeri nella raccolta
    - scrive prodotto ed il risultato del prodottodi tutti i numeri nella raccolta

Esercizi

1. Scrivete un programma che disegna un quadrato, un esagono, un ottagono, un decagono. Tutti partendo dallo stesso punto, ma con colori diversi

2. Semplificate il `turtle_program` per disegnare 2 quadrati e 2 trinagoli con 4 tartarughe..utilzzando i loop `for`

3. utilizzando i metodi `penup()` e `pendown()` di `Turtle` modificate il programma per disegnare i 2 quadrati e i due triangoli *senza le linee di spostamento* delle tartarughe.

Note:

- Potete nascondere e far riapparire ogni tartaruga con metodi `hideturtle()` e `showturtle()`
- Potete accelerare o rallentare le tartarughe con il metodo `speed()`
- Potete cambiare la dimensione del tratto della matitta con il metodo `pensize()`

Boolean expressions

Un valore booleano è di tipo **bool** e può essere **vero** (`True`) o **falso** (`False`)

Boolean expressions

Un valore booleano è di tipo **bool** e può essere **vero** (`True`) o **falso** (`False`)

In [1]:
```
type(True)
```

Out[1]:
```
bool
```

Una espressione booleana è una espressione il cui risultato è un valore booleano

Una espressione booleana è una espressione il cui risultato è un valore booleano

In [2]:
```
5 == 5  # valuto se due numeri sono uguali
```

Out[2]:
```
True
```

Una espressione booleana è una espressione il cui risultato è un valore booleano

In [2]:
```
5 == 5  # valuto se due numeri sono uguali
```

Out[2]:
```
True
```

In [1]:
```
5 == 7  # valuto se due numeri sono uguali
```

Out[1]:
```
False
```

Una espressione booleana è una espressione il cui risultato è un valore booleano

In [2]:
```
5 == 5  # valuto se due numeri sono uguali
```

Out[2]:
```
True
```

In [1]:
```
5 == 7  # valuto se due numeri sono uguali
```

Out[1]:
```
False
```

In [4]:
```
'ciao' == 'casa'  # valuto se due stringhe sono uguali
```

Out[4]:
```
False
```

Una espressione booleana è una espressione il cui risultato è un valore booleano

In [2]:
```
5 == 5  # valuto se due numeri sono uguali
```

Out[2]: True

In [1]:
```
5 == 7  # valuto se due numeri sono uguali
```

Out[1]: False

In [4]:
```
'ciao' == 'casa'  # valuto se due stringhe sono uguali
```

Out[4]: False

In [5]:
```
a = 5    # assegno valore a variabile
a == 9   # valuto se la veriabile è uguale ad un valore
```

Out[5]: False

Operatori di confronto

L'operatore `==` è uno dei 6 operatori di confronto più comuni

```
x == y   # .. True se x uguale a y
x != y   # .. True se x diverso da y
x >  y   # .. True se x è maggiore di y
x <  y   # .. True se x è minore di y
x >= y   # .. True se x è maggiore OPPURE uguale a y
x <= y   # .. True se x è minore OPPURE ugnuale a y
```

> *L'operatore di confronto `==` è diverso dell'operatore assegnazione `=`*
>
> *Gli operatori `>=` e `<=` non possono essere invertiti: =< e => non esistono*

Le variabili booleane possono essere visualizzate con `print()` come tutte le variabili

Operatori di confronto

L'operatore `==` è uno dei 6 operatori di confronto più comuni

```
x == y   # .. True se x uguale a y
x != y   # .. True se x diverso da y
x >  y   # .. True se x è maggiore di y
x <  y   # .. True se x è minore di y
x >= y   # .. True se x è maggiore OPPURE uguale a y
x <= y   # .. True se x è minore OPPURE ugnuale a y
```

> *L'operatore di confronto `==` è diverso dell'operatore assegnazione `=`*
>
> *Gli operatori `>=` e `<=` non possono essere invertiti: =< e => non esistono*

Le variabili booleane possono essere visualizzate con `print()` come tutte le variabili

In [6]:
```
a = True
print(a)
```

```
True
```

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

In [7]:
```
3 > 5 or 3 < 5
```

Out[7]:
```
True
```

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

In [7]:
```
3 > 5 or 3 < 5
```

Out[7]:
```
True
```

- `and`, "e" che ritorna True **se entrambe** le condizione sono vere

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

In [7]:
```
3 > 5 or 3 < 5
```

Out[7]:
```
True
```

- `and`, "e" che ritorna True **se entrambe** le condizione sono vere

In [8]:
```
3 > 1 and 3 > 2
```

Out[8]:
```
True
```

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

In [7]:
```
3 > 5 or 3 < 5
```

Out[7]:
```
True
```

- `and`, "e" che ritorna True **se entrambe** le condizione sono vere

In [8]:
```
3 > 1 and 3 > 2
```

Out[8]:
```
True
```

- `not`, che "nega" la condizione **invertendo** il risultato

Operatori Logici

Per costruire espressioni booleane più complesse si possono utilizzare i 3 operatori logici

- `or`, "oppure" che ritorna True **se almeno una** delle condizione è vera

In [7]:
```
3 > 5 or 3 < 5
```

Out[7]:
```
True
```

- `and`, "e" che ritorna True **se entrambe** le condizione sono vere

In [8]:
```
3 > 1 and 3 > 2
```

Out[8]:
```
True
```

- `not`, che "nega" la condizione **invertendo** il risultato

In [10]:
```
not 3 > 5
```

Out[10]:
```
True
```

Operatori Opposti

Ognuno dei 6 operatori di confronto ha il suo opposto e conoscerlo ci aiuta a usare pochi `not` che sono a difficili da leggere in un programma

operatore	opposto	ci salva da...
`a == b`	`a != b`	`not ( a == b)`
`a != b`	`a == b`	`not ( a != b)`
`a <  b`	`a >= b`	`not ( a <  b)`
`a >  b`	`a <= b`	`not ( a >  b)`
`a >= b`	`a <  b`	`not ( a >= b)`
`a <= b`	`a >  b`	`not ( a <= b)`

inoltre

leggi De Morgan
`not (a and b) == (not a) or  (not b)`
`not (a or  b) == (not a) and (not b)`

> *Con la pratica troverete modi diversi di risolvere un problema:* ***i buoni programmi sono progettati*** *tramite scelte che enfatizzano semplicità, chiarezza ed eleganza*

Tabelle di verità degli operatori

Se abbiamo 2 variabili booleane `a` e `b`

Operatore `and`

a	b	a `and` b
True	True	True
True	False	False
False	True	False
False	False	False

Operatore `or`

a	b	a `or` b
True	True	True
True	False	True
False	True	True
False	False	False

Considerazioni

Python, come altri linguaggi sfrutta la *short-circuit evaluation* per evitare conti non necessari ed essere più veloce

---

```
CONDIZIONE_1 or CONDIZIONE_2 or CONDIZIONE_3 or CONDIZIONE_4
```

Nell'operatore `or` la condizione a sinistra è la prima ad essere controllata.
Se è `True` sappiamo già che il risultato finale sarà `True` quindi nessun'altra condizione sarà controllata

---

Considerazioni

Python, come altri linguaggi sfrutta la *short-circuit evaluation* per evitare conti non necessari ed essere più veloce

---

```
CONDIZIONE_1 or CONDIZIONE_2 or CONDIZIONE_3 or CONDIZIONE_4
```

Nell'operatore `or` la condizione a sinistra è la prima ad essere controllata.
Se è `True` sappiamo già che il risultato finale sarà `True` quindi nessun'altra condizione sarà controllata

---

```
CONDIZIONE_1 and CONDIZIONE_2 and CONDIZIONE_3 and CONDIZIONE_4
```

Nell'operatore `and` la condizione a sinistra è la prima ad essere controllata.
Se è `False` sappiamo già che il risultato finale sarà `False` quindi nessun'altra condizione sarà controllata

---

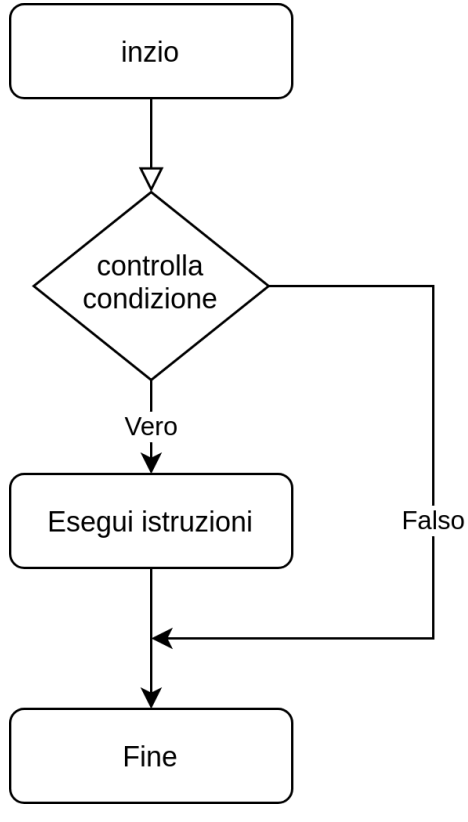
> ***Progettate bene l'ordine delle condizioni quando scrivete un programma: l'efficenza ne risentirà parecchio***

Condizioni

In molti casi avremo bisogno di **cambiare il comportamento del programma** a seconda che si verifichi o meno una **condizione**.

Condizioni

In molti casi avremo bisogno di **cambiare il comportamento del programma** a seconda che si verifichi o meno una **condizione**.



Gli **statement condizionali** si costruiscono con
la keyword `if`, una **Boolean expression** e il token `:`

```
if CONDIZIONE :
    # istruzioni condizionali
    ISTRUZIONI
# fine del blocco istruzioni
```

- la CONDIZIONE è una **Boolean expression**: assume valore **True** o **False**.
- il **blocco** di istruzioni è indentato ed è eseguito **solo se la condizione è True**.
- il blocco di istruzioni (**body**) termina rientrando dall'indentazione

Gli **statement condizionali** si costruiscono con
la keyword `if`, una **Boolean expression** e il token `:`

```
if CONDIZIONE :
    # istruzioni condizionali
    ISTRUZIONI
# fine del blocco istruzioni
```

- la CONDIZIONE è una **Boolean expression**: assume valore **True** o **False**.
- il **blocco** di istruzioni è indentato ed è eseguito **solo se la condizione è True**.
- il blocco di istruzioni (**body**) termina rientrando dall'indentazione

In [11]:
```
if True :
    print('ciao')
```

```
ciao
```

Gli **statement condizionali** si costruiscono con
la keyword `if`, una **Boolean expression** e il token `:`

```
if CONDIZIONE :
    # istruzioni condizionali
    ISTRUZIONI
# fine del blocco istruzioni
```

- la CONDIZIONE è una **Boolean expression**: assume valore **True** o **False**.
- il **blocco** di istruzioni è indentato ed è eseguito **solo se la condizione è True**.
- il blocco di istruzioni (**body**) termina rientrando dall'indentazione

In [11]:
```
if True :
    print('ciao')
```

```
ciao
```

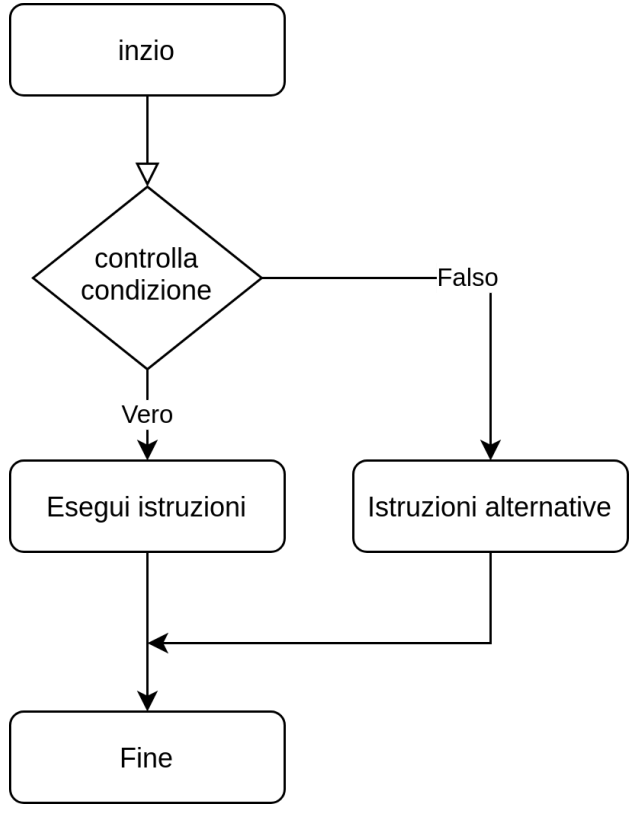
In [12]:
```
if False :
    print('ciao')
```

Condizioni con alternativa

Negli **statement condizionali** è possibile specificare una alternativa al blocco operazioni

Condizioni con alternativa

Negli **statement condizionali** è possibile specificare una alternativa al blocco operazioni



Gli **statement condizionali** con **alternativa** si costruiscono con l'aggiunta del blocco che inizia con keyword `else` e il token `:`

```
if CONDIZIONE :
    # istruzioni se True
    ISTRUZIONI
else :
    # istruzioni se False
    ISTRUZIONI_ALTERNATIVE

# fine del blocco istruzioni
```

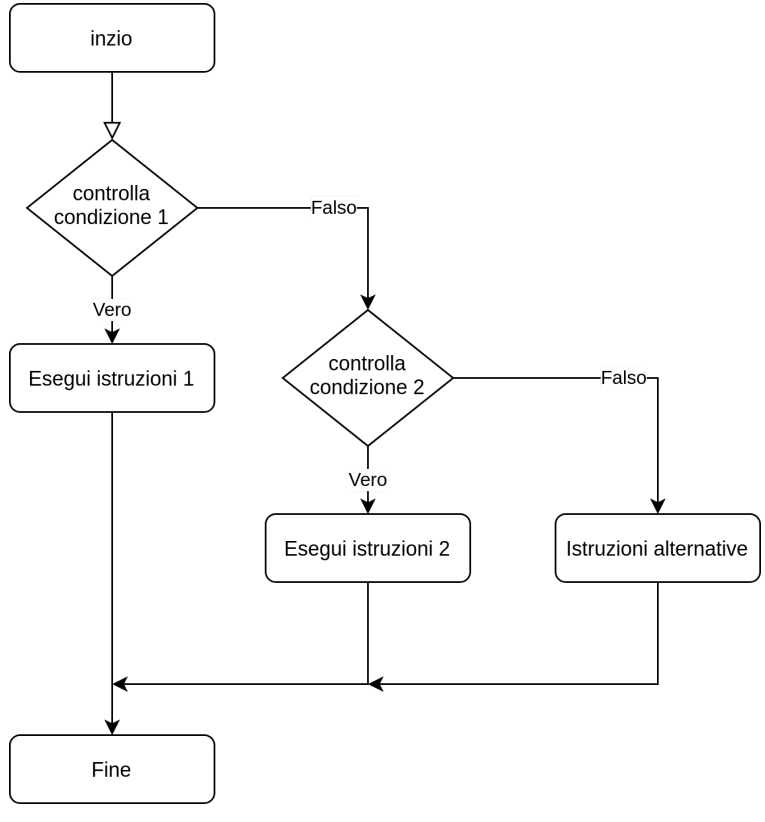
- il blocco *ISTRUZIONI* è eseguito **solo se la condizione è** `True`.
- la keyword `else` termina il blocco *ISTRUZIONI* rientrando dall'indentazione.
- il blocco *ISTRUZIONI_ALTERNATIVE* è eseguito **se la condizione è** `False`.
- la struttura condizionale termina termina rientrando dall'indentazione nel blocco `else`
- Non c'è limite al numero di istruzioni che possono esserci in ognuno dei blocchi `if` o `else`

Condizioni concatenate

Spesso ci sono più di 2 alternative e servono più di due blocchi di istruzioni

Condizioni concatenate

Spesso ci sono più di 2 alternative e servono più di due blocchi di istruzioni



Le condizioni concatenate si ottengono con
l'aggiunta di blocchi che iniziano con keyword `elif`, **boolean expression** e `:`

```
if CONDIZIONE_1 :
    # istruzioni se CONDIZIONE_1 True
    ISTRUZIONI_1

elif CONDIZIONE_2 :
    # istruzioni se CONDIZIONE_2 True
    ISTRUZIONI_2

else :
    # istruzioni se False
    ISTRUZIONI_ALTERNATIVE

# fine del blocco istruzioni
```

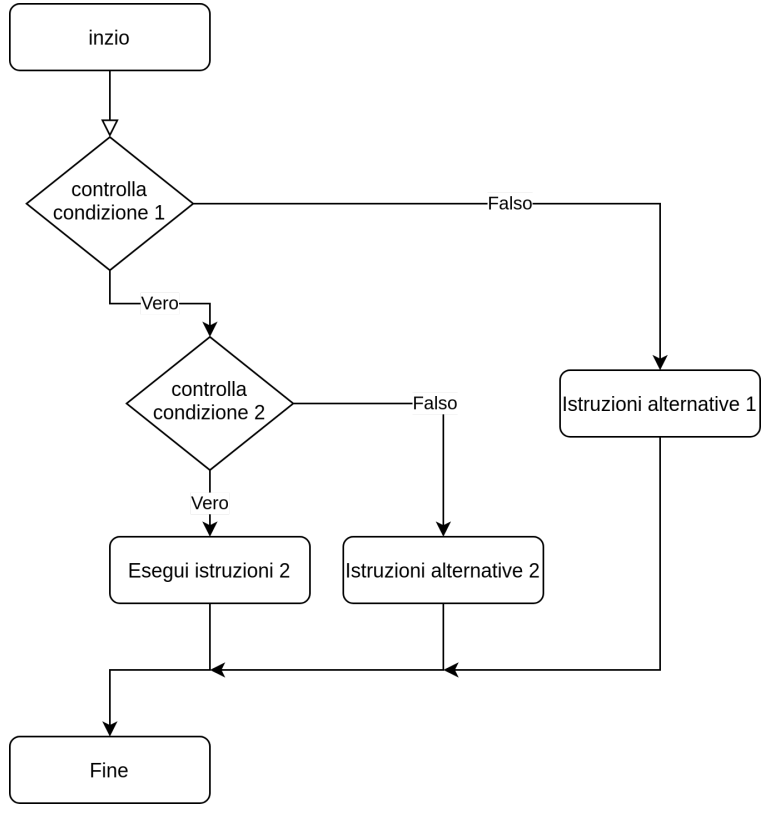
- ogni blocco *ISTRUZIONI* è eseguito **solo se la SUA condizione è `True`**.
- la keyword `elif` termina il blocco *ISTRUZIONI_1* rientrando dall'indentazione.
- *ISTRUZIONI_ALTERNATIVE* è eseguito **se l'ultima condizione(2) è `False`**.
- la struttura condizionale termina rientrando dall'indentazione nel blocco `else`
- Non c'è limite al numero di istruzioni che possono esserci in ognuno dei blocchi `if` o `elif` o `else`
- i blocchi sono esclusivi: se ne eseguo uno **non** eseguo gli altri

Condizioni annidate

I blocchi condizionali possono anche contenere altri blocchi condizionali *annidati*

Condizioni annidate

I blocchi condizionali possono anche contenere altri blocchi condizionali *annidati*



Le condizioni annidate, anche se visualmente rendono l'idea delle diverse alternative, diventano molto difficili da seguire logicamente in poco tempo

```
if CONDIZIONE_1 :
    # nuova condizione annidata come blocco ISTRUZIONI_1
    if CONDIZIONE_2 :
        ISTRUZIONI_2
    else:
        ISTRUZIONI_ALTERNATIVE_2
else :
    ISTRUZIONI_ALTERNATIVE_1
```

Le condizioni annidate, anche se visualmente rendono l'idea delle diverse alternative, diventano molto difficili da seguire logicamente in poco tempo

```
if CONDIZIONE_1 :
    # nuova condizione annidata come blocco ISTRUZIONI_1
    if CONDIZIONE_2 :
        ISTRUZIONI_2
    else:
        ISTRUZIONI_ALTERNATIVE_2
else :
    ISTRUZIONI_ALTERNATIVE_1
```

> *In generale, è meglio limitare il più possibile le condizioni annidate*

Le condizioni annidate, anche se visualmente rendono l'idea delle diverse alternative, diventano molto difficili da seguire logicamente in poco tempo

```
if CONDIZIONE_1 :
    # nuova condizione annidata come blocco ISTRUZIONI_1
    if CONDIZIONE_2 :
        ISTRUZIONI_2
    else:
        ISTRUZIONI_ALTERNATIVE_2
else :
    ISTRUZIONI_ALTERNATIVE_1
```

> *In generale, è meglio limitare il più possibile le condizioni annidate*

Gli **operatori logici** aiutano a ridurre le condizioni annidate

```
if CONDIZIONE_1 :
    if CONDIZIONE_2 :
        ISTRUZIONI
```

si può infatti scrivere

```
if CONDIZIONE_1 and CONDIZIONE_2 :
    ISTRUZIONI
```

Esercizi

1. Scrivere un codice che chieda in input all'utente un range di numeri (inizio e fine) e poi stampi a video solo i numeri pari
2. Modificare il codice al punto 1. per scrivere `pari:` e il numero se quest'ultimo è pari, altrimenti scrivere `dispari:` e il numero
3. Modificare il codice al punto 2. per scrivere solo i numeri pari multipli di 3 e i numeri dispari multipli di 5

Esercizi

1. Modificare il `turtle_program` nel modo seguente:
    - Chiedere all'utente la lunghezza `l` del lato poligono
    - Se `l` è un numero pari, disegnare un quadrato del colore della tartaruga, altrimenti disegnate un triangolo nero