Applicazione del pensiero computazionale

1 Dall’algoritmo al programma

Il concetto di pensiero computazionale che è correlato al concetto di **algoritmo**. Il termine algoritmo deriva dalla trascrizione latina del nome del matematico persiano al-Khwarizmi, vissuto nel IX secolo D.C., che è considerato uno dei primi ad aver introdotto il concetto di algoritmo scrivendo il libro *Regole di ripristino e riduzione* [20].

Per poter realizzare un algoritmo è necessario individuare il problema di cui l’algoritmo rappresenta la soluzione. Per raggiungere un qualunque obiettivo, come ad esempio la risoluzione di un problema di matematica o l’individuazione del percorso più efficiente per arrivare in palestra, è necessario raccogliere e analizzare le informazioni che servono per arrivare alla soluzione. In questa prima fase bisogna riconoscere quali dati, tra quelli disponibili, servono e quali sono ridondanti, valutare se sono sufficienti a raggiungere lo scopo e se non sono ambigui.

«*Un algoritmo è un procedimento che risolve un determinato problema attraverso un numero finito di passi elementari in un tempo ragionevole*» [21].
«*Termine che indicò nel medioevo i procedimenti di calcolo numerico fondati sopra l’uso delle cifre arabiche. Nell’uso odierno, anche con riferimento all’uso dei calcolatori, qualunque schema o procedimento matematico di calcolo; più precisamente, un procedimento di calcolo esplicito e descrivibile con un numero finito di regole che conduce al risultato dopo un numero finito di operazioni, cioè di applicazioni delle regole*»[22].
« *In informatica, insieme di istruzioni che deve essere applicato per eseguire un’elaborazione o risolvere un problema*»[23].

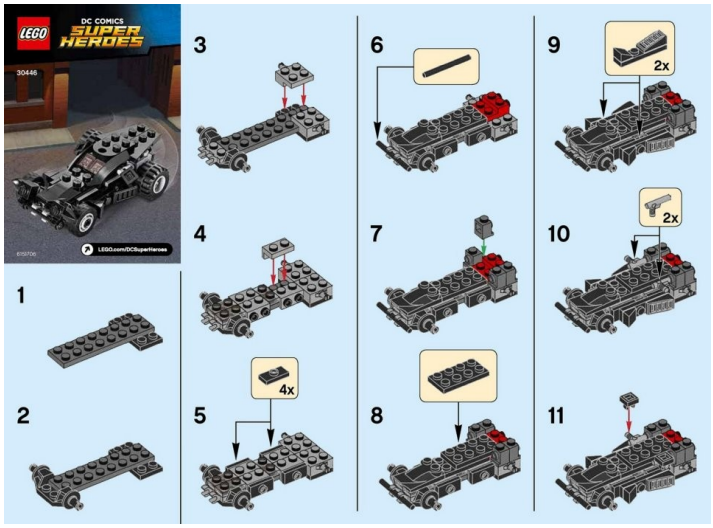
La parola algoritmo può far pensare a chissà quali procedimenti complessi, in realtà eseguiamo un algoritmo molto più spesso di quanto pensiamo, ad esempio quando seguiamo i passi di una ricetta di cucina, le istruzioni di montaggio per i mattoncini Lego e risolviamo un’espressione algebrica. Solitamente una ricetta di cucina è descritta a parole, le istruzioni di montaggio sono presentate come schema e l’espressione è rappresentata mediante la notazione matematica. Dagli esempi appena citati si evidenzia che per poter parlare di algoritmo devono essere presenti le seguenti proprietà:

A	*Istruzioni semplici, non ambigue, ed elaborate in sequenza*
B	*Raggiungere un obiettivo conosciuto*
C	*Esecuzione in un tempo finito*

Fase A: l'algoritmo è composto da istruzioni essenziali, comprensibili da chi deve eseguire il procedimento, ed inoltre devono essere realizzate in sequenza. Osserviamo gli esempi di seguito.

Ricetta per il tiramisù:
1. separare i tuorli dagli albumi
2. con le fruste elettriche montare i tuorli e metà dose di zucchero
3. aggiungere il mascarpone
4. pulire le fruste
5. montare a neve gli albumi versando il restante zucchero
…..
10. *livellate la superficie e lasciate rassodare per un paio d'ore in frigorifero*
11. *prima di servire il tiramisù spolverizzate la superficie con del cacao amaro*

Le istruzioni Lego:



L’espressione algebrica:

$5+10+(32-9)*5$

Dalle sequenze delle tre situazioni presentate viene messa in evidenza la necessità di elaborare i passaggi secondo un procedimento ordinato.

Fase **B.** Le istruzioni sono eseguite per raggiungere un determinato scopo, nel nostro caso fare il tiramisù, assemblare i mattoncini Lego e calcolare l’espressione.

Fase **C.** L’obiettivo deve essere raggiunto in un tempo limitato e conveniente: se ho degli amici a cena, il tiramisù deve essere pronto, non avrebbe senso terminarlo il giorno seguente.

Come tutte le macchine il computer non ha nessuna capacità decisionale o discrezionale ma si limita a compiere determinate azioni secondo procedure prestabilite, i programmi. Si può affermare che il computer è in grado di compiere un’unica azione: eseguire istruzioni. In ambito computazionale gli algoritmi rappresentano le sequenze di operazioni da eseguire per svolgere un determinato compito, tuttavia affinché posano essere eseguiti dal computer necessitano di essere tradotti in un linguaggio comprensibile per la macchina. Per fare ciò vengono utilizzati i **linguaggi di programmazione**.

Il linguaggio naturale rappresenta lo strumento grazie al quale gli individui riescono a comunicare tra loro. Il linguaggio può essere verbale (scritto o parlato), gestuale, visivo o musicale, in particolare, il linguaggio verbale si sviluppa seguendo delle regole precise: la sintassi e la semantica. La **sintassi** studia come le parole si combinano per formare la frase, la **semantica** si occupa del significato da attribuire alle parole e alle frasi. La sintassi e la semantica non sono uguali per tutti i linguaggi verbali.

Il linguaggio naturale è ambiguo, ovvero un’espressione può avere diversi significati, cioè può essere soggetta a più interpretazioni, pertanto non è adatto per comunicare con il computer. Un linguaggio di programmazione, invece, è una lingua formale utilizzata per descrivere in modo univoco l’insieme di azioni consecutive che un computer deve eseguire, come il linguaggio naturale, ha una sintassi e una semantica. Un **linguaggio formale** utilizza informazioni chiare e non presenta ambiguità, cioè ad un’informazione non si possono attribuire più significati. Questo tipo di linguaggio utilizza i connettivi logici grazie ai quali è possibile costruire frasi complesse formate dall’unione di frasi più semplici. Per esempio nella frase:

Ho visto Dario con il binocolo

L’espressione algebrica:

$5+10+(32-9)*5$

Dalle sequenze delle tre situazioni presentate viene messa in evidenza la necessità di elaborare i passaggi secondo un procedimento ordinato.

Fase **B.** Le istruzioni sono eseguite per raggiungere un determinato scopo, nel nostro caso fare il tiramisù, assemblare i mattoncini Lego e calcolare l’espressione.

Fase **C.** L’obiettivo deve essere raggiunto in un tempo limitato e conveniente: se ho degli amici a cena, il tiramisù deve essere pronto, non avrebbe senso terminarlo il giorno seguente.

Come tutte le macchine il computer non ha nessuna capacità decisionale o discrezionale ma si limita a compiere determinate azioni secondo procedure prestabilite, i programmi. Si può affermare che il computer è in grado di compiere un’unica azione: eseguire istruzioni. In ambito computazionale gli algoritmi rappresentano le sequenze di operazioni da eseguire per svolgere un determinato compito, tuttavia affinché posano essere eseguiti dal computer necessitano di essere tradotti in un linguaggio comprensibile per la macchina. Per fare ciò vengono utilizzati i **linguaggi di programmazione**.

Il linguaggio naturale rappresenta lo strumento grazie al quale gli individui riescono a comunicare tra loro. Il linguaggio può essere verbale (scritto o parlato), gestuale, visivo o musicale, in particolare, il linguaggio verbale si sviluppa seguendo delle regole precise: la sintassi e la semantica. La **sintassi** studia come le parole si combinano per formare la frase, la **semantica** si occupa del significato da attribuire alle parole e alle frasi. La sintassi e la semantica non sono uguali per tutti i linguaggi verbali.

Il linguaggio naturale è ambiguo, ovvero un’espressione può avere diversi significati, cioè può essere soggetta a più interpretazioni, pertanto non è adatto per comunicare con il computer. Un linguaggio di programmazione, invece, è una lingua formale utilizzata per descrivere in modo univoco l’insieme di azioni consecutive che un computer deve eseguire, come il linguaggio naturale, ha una sintassi e una semantica. Un **linguaggio formale** utilizza informazioni chiare e non presenta ambiguità, cioè ad un’informazione non si possono attribuire più significati. Questo tipo di linguaggio utilizza i connettivi logici grazie ai quali è possibile costruire frasi complesse formate dall’unione di frasi più semplici. Per esempio nella frase:

Ho visto Dario con il binocolo

non è chiaro se Dario è stato visto attraverso il binocolo, o se era Dario ad avere il binocolo. Se fossimo stati presenti all’evento non avremmo avuto alcun dubbio.

Esistono diversi linguaggi di programmazione (C, C++, Java, Python, ecc.) ognuno con delle sue peculiarità e la scelta dell'uno o dell'altro dipende da diversi fattori come efficienza, efficacia, adattabilità, portabilità e altro ancora.

Python è il linguaggio formale che più di altri si presta per lo scopo di questa trattazione. Python è stato sviluppato nel 1991 dall'informatico olandese Guido van Rossum, e il nome deriva dalla passione del suo ideatore per il gruppo di comici Monty Python. È un linguaggio potente, di facile apprendimento e lettura perché semplice e intuitivo. Viene impiegato per la realizzazione di desktop, siti e applicazioni web,calcolo scientifico e numerico, interfacce grafiche, giochi, database, ed altro. La NASA, Google e YouTube usano Python. Nel periodo in cui è stato scritto questo documento le versioni disponibili sono la 2.7.14 e la 3.6.4.

Per comprendere il significato di sintassi e semantica di un linguaggio analizziamo i seguenti esempi:

1) *Matteo mangiano la mela.*

"Matteo mangiano" è sintatticamente sbagliata perché, secondo la grammatica italiana, il verbo deve concordare con il soggetto nella persona e nel numero; la frase sintatticamente corretta è "Matteo mangia...". Dal punto di vista semantico, tuttavia, è una frase comprensibile.

2) *La mela mangia Matteo.*

È una frase sintatticamente corretta, ma non semanticamente perché nella realtà è improbabile che si verifichi una tale situazione.

Per comprendere meglio la differenza tra sintassi e semantica utilizziamo il comando *if-else* di Python, il cui significato sintattico è: *se condizione vera fai questo altrimenti fai un altra cosa*. Esempio:

a) *se* Maria supera l'esame (*condizione da verificare*)
   può andare in vacanza (*condizione verificata*)
   altrimenti
   rimane a casa (*condizione non verificata*)

utilizzando l'apposito costrutto Python la frase di sopra si traduce come segue:

```
if Maria supera l'esame:
    può andare in vacanza      <-- Istruzione indentata di if
else:
    rimane a casa              <-- Istruzione indentata di else
```

La regola sintattica del costrutto usato nell'esempio è:
if condizione
    istruzione
else
    istruzione

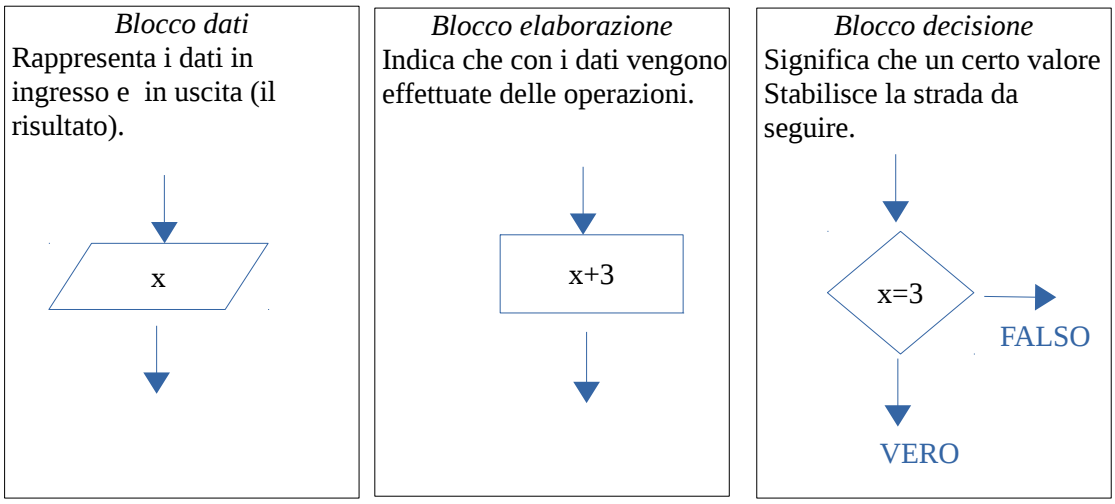
Come si può notare sono state seguite delle regole di posizionamento che in un linguaggio di programmazione sono chiamate regole di indentazione. L'**indentazione** è un rientro rispetto al margine sinistro, una convenzione tipica dei linguaggi di programmazione per facilitare la lettura dei programmi. In Python l'indentazione ha un preciso ruolo sintattico, indica il blocco di istruzioni che formano il comando.

c) *if* Maria supera l'esame:
  _può andare in vacanza
  else:
  rimane a casa ← Istruzione non indentata

Nell'esempio di sopra la sintassi del costrutto *if-else* non è corretta perché il comando "rimane a casa" non segue le regole di indentazione.

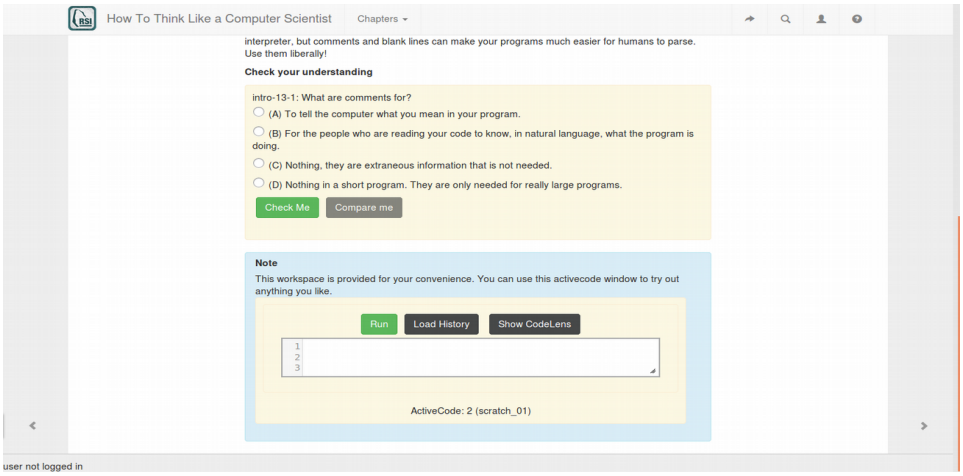
Regola di ogni linguaggio di programmazione è quella di assegnare un nome al programma che realizza l'algoritmo. Supponiamo di dover calcolare il saldo del conto corrente, possiamo chiamare *Calcola_saldo* il programma che lo implementa. Un file è un contenitore di dati che ha un nome e un'estensione, cioè un suffisso posto dopo il nome del file e preceduto dal punto, tale suffisso varia da linguaggio a linguaggio. Il contenitore del nostro programma sarà un file a cui daremo un nome, *Calcola_saldo.py.*, dove ".py" è l'estensione di Python. Se utilizziamo, per esempio, il linguaggio editore Word per scrivere una ricerca sul sistema solare, il file è *Sistema_solare.docx*, se usiamo LibreOffice abbiamo invece *Sistema_solare.odt*.
Il file che contiene le foto delle vacanze al mare può essere *Mare_2017.jpg*.
I file audio hanno attualmente l'estensione *.mp3*. Quindi, il programma visto sopra diventa *Calcola_saldo.py*, Sarebbe *Calcola_saldo.java* se il linguaggio usato fosse Java.

Nei capitoli seguenti impareremo le principali strutture di Python e analizzeremo esempi di piccoli programmi con l'aiuto di schemi, tabelle e diagrammi di flusso. Questi ultimi sono rappresentazioni grafiche utilizzate per rappresentare i passaggi informali che portano alla stesura di un programma. I principali elementi che compongono i diagrammi di flusso sono:



Per poter eseguire una eventuale verifica del programma, è possibile utilizzare un ambiente di lavoro (*workspace*) disponibile nel libro interattivo *How to Think like a Computer Scientist – Interactive Python*[1]. Una volta aperto il libro interattivo, digitare il capitolo *1.13 Comments* e andare in fondo alla pagina:

Figura 3.1.1



1 www.interactivepython.org/runestone/static/thinkcspy/index.html
È un prodotto open source del Runestone Interactive Project (*runestoneinteractive.org*, *runestone.academy*) guidato da Brad Miller e David Ranum.

La maschera colorata di azzurro (Figura 3.1.3) mostra l'ambiente a disposizione per eseguire le verifiche. È possibile digitare i comandi nello spazio bianco, poi il tasto verde *Run* per visualizzare il risultato. Per vedere i passaggi del programma si digita il tasto *Show in Codelens*, per chiudere la finestra il tasto *Hide Codelens*.

Figura 3.1.2

2 Strutture dati: variabili e liste

Le **variabili** possono essere considerate come contenitori nel quali memorizziamo informazioni utili, che verranno riutilizzate o modificate. Il termine per identificarle, successivamente viene scelto dal programmatore. Per esempio, abbiamo un salvadanaio che contiene 20 euro:

Questa annotazione indica che alla variabile salvadanaio è stato assegnato il valore 20.È necessario ricordare una regola fondamentale

VARIABILE = VALORE

a sinistra del simbolo "=" (uguale) abbiamo sempre il nome della variabile, mentre a destra c'è sempre il valore da assegnarle.

Le variabili sono identificate attraverso dei vocaboli scelti dal programmatore. Possiamo quindi avere "pluto", "supercar", ecc., ma è preferibile che abbiano dei nomi significativi, adeguati al contesto, per permettere una comprensione più immediata. Nell'esempio precedente, al posto di salvadanaio avremmo anche potuto utilizzare un altro termine:

paperino=20

ma si nota come non sia spontaneo pensare ad un salvadanaio contenente dei soldi.

Nei linguaggi di programmazione esistono le **keywords**, parole chiave, chiamate così perché eseguono precise funzioni e per questo non possono essere usate come nomi di variabili. In Python, e in tanti altri linguaggi di programmazione, ci sono, per esempio, le parole chiave **and**, **else**, **for**, **if**, **in**, **not**, **or**, **print, while**, **True**, **False**, che incontreremo più avanti. Queste sono le più comuni, ma ce ne sono altre di cui parleremo via via.

Le variabili non hanno una lunghezza prestabilita, possono essere composte da più parole, oppure da parole e cifre:

```
salvadanaiomio=15
salvadanaio2=25
```

Python prevede precise regole per assegnare i nomi delle variabili:

- il nome di una variabile non deve contenere spazi

```
salvadanaio mio=15
```
ERRORE!

```
salvadanaiomio=5
```

- possono contenere il trattino basso "_", per rendere più leggibili i nomi composti

```
salvadanaio_mio=25
```

- non possono iniziare con un numero

```
23lattine=8
```
ERRORE!

```
lattine23=8
```

-non possono contenere caratteri speciali come &, $, /

```
&lattine=15
```
ERRORE!

```
sconto$=10
```
ERRORE!

Python fa distinzione tra caratteri minuscoli e maiuscoli, per cui la parola "cane" è diversa da "Cane", tuttavia, per semplicità ed evitare errori, i caratteri maiuscoli si usano con attenzione.

Le variabili che hanno come valore un numero sono di tipo numerico, e possono essere modificate attraverso operazioni matematiche.

Supponiamo di avere un salvadanaio elettronico che registra i movimenti e mostra il totale aggiornato sul suo display. La nonna ci regala 10€ che li mettiamo nel salvadanaio. La variabile, a cui prima abbiamo assegnato 20, viene ora modificata con un'operazione di somma:

```
salvadanaio = salvadanaio + 10
```

È come scrivere `salvadanaio = 20+10`, ma l'operazione di addizione viene eseguita direttamente con la variabile `salvadanaio` perché ci possono essere state molte operazioni che la coinvolgono e che cambiano continuamente il suo valore. È necessario memorizzare tutte le variazioni in modo da poter effettuare i calcoli sui valori aggiornati. Se non si tiene traccia di tutte le modifiche si rischiano errori che possono avere conseguenze importanti.

Consideriamo ora lo scambio di valore tra due variabili. Supponiamo di avere due salvadanai (s1 e s2), e di voler scambiare i rispettivi contenuti:

- `s1=500.000`
- `s2=20`

Il risultato dello scambio deve essere:

- `s1=20`
- `s2=500.000`

Se scriviamo `s1=s2`, che significa "s1 prende il valore di s2": `s1=20 e s2=20`

Se scriviamo `s2=s1` , che significa "s1 prende il valore di s2": `s1=500 e s2=500`.

Nel primo caso abbiamo perso il valore di s1, nel secondo il valore di s2.

Quando si fa l'assegnamento di un valore a una variabile si perde il dato inserito in precedenza, pertanto è necessaria una variabile di appoggio che ci permetta di memorizzare momentaneamente il valore di un conto per poi trasferirlo all'altro:

- `appoggio=salvadanio1`
  la variabile appoggio prende il valore di salvadanaio1: `appoggio=500.000`

- `salvadanaio1=salvadanaio2`
  salvadanaio1 prende il valore di salvadanio2: `salvadanaio1=20`

- `salvadanaio2=appoggio`
  salvadanaio2 prende il valore di salvadanaio1: `salvadanaio2=500.000`

In Python esiste un comando per identificare e controllare il tipo numerico: **int**() che indica un intero, e **float**() che indica un numero con la virgola:

`a=int(12)` corrisponde a 12
`a=float(12)` inserisce la virgola: 12,0
`b=int(23,5)` elimina i valori che seguono la virgola: 23
`b=float(23,5)` corrisponde a 23,5

In Python gli operatori numerici sono:

Simboli	Operatori	Espressioni
+	addizione	7+4=11
-	sottrazione	7-4=3
*	moltiplicazione	7*4=28
/	divisione	7 / 4=1,75
//	divisione intera	7 // 4 = 1
%	modulo	3

L'operazione di modulo rappresenta il resto di una divisione intera.

Per esempio, “nove modulo tre” ha resto zero:

9 % 3=0 infatti 9/3=3 con resto zero

Invece

7 % 4=3 infatti 7 // 4=1

Il valore di una variabile può essere anche il risultato di un’espressione, ossia di una serie finita di operazioni aritmetiche:

```
espressione = 5+6*2
stampa (espressione)
```

è uguale a 5+(6*2)

```
17
```

Il risultato è 17 e non 22 perché le operazioni di moltiplicazione e divisione hanno la precedenza su addizione e sottrazione, nell’ordine in cui sono scritte da sinistra a destra. Vengono rispettate le regole matematiche di precedenza tra gli operatori. Per esempio, risolviamo l’espressione 5+6/2*4:

Da sinistra verso destra la prima operazione da eseguire è 6/2=3
L’espressione diventa 5+3*4
Da sinistra verso destra la seconda operazione da eseguire è 3*4
L’espressione diventa 5+12=17

Per sicurezza possiamo utilizzare le parentesi tonde che ci aiutano a mettere in evidenza le sezioni che hanno la precedenza sulle altre: 5+((6/2)*4).

Utilizziamo Python e scriviamo il programma `Area_triangolo.py` che calcola l’area di un triangolo con base 4 e altezza 7. Intuiamo subito quali siano le variabili da usare ed i valori da assegnare:

```
#calcola l’area di un triangolo

base=4
altezza=7

#notare che nell’espressione sono utilizzate le variabili

#base e altezza
area_triangolo=int(base*altezza/2)

#il comando di stampa per visualizzare risultato
print(area_triangolo)
```

Le parti in verde sono i commenti i quali hanno funzione esplicativa e sono ignorati dal calcolatore ai fini della programmazione. Servono per facilitare la comprensione del programma. Tutti i linguaggi di programmazione prevedono la possibilità di inserire commenti, ciascuno con la propria modalità, secondo Python queste note devono essere precedute dal segno di cancelletto "#", senza questo simbolo viene segnalato un errore di sintassi e il programma non viene eseguito.

In python il comando **print** rappresenta l'istruzione di stampa (che normalmente avviene sul display).

```
12
```

Le variabili ricoprono un ruolo decisivo poiché il loro valore può variare in modo dinamico, per chiarire meglio il concetto esaminiamo l'esempio che segue:
a) area_triangolo=4*7/ 2
b) area_triangolo = base*altezza/ 2
Nel caso a) il valore della variabile area triangolo resta immutato a meno che non si modifichino direttamente i valori 4*7/2, nel caso b) ogni volta che cambia il valore della variabile base e/o della variabile altezza cambia anche il valore della variabile area_triangolo. Pertanto nel caso a) area_triangolo ha un valore costante durante l'esecuzione del programma, mentre nel caso b) il suo valore è dinamico e dipende da altre variabili.

Le variabili possono non avere valore numerico ed essere costituite anche da una sequenza di caratteri, in questo caso si dice che sono di tipo stringa.

Un carattere può essere una lettera, un numero o un segno di punteggiatura.

Una stringa non è semplicemente una collezione, ma una sequenza di caratteri, per cui gli elementi che la compongono seguono un ordine da sinistra a destra.

Solitamente le stringhe sono riconoscibili perché vengono inserite tra le virgolette, quando è possibile, con l'istruzione **str**() è possibile convertire un valore nel tipo stringa. Di seguito sono descritti alcuni esempi per mostrare le caratteristiche delle variabili di tipo striga:
- stringa1 = "Ciao!"
I caratteri sono in ordine, il primo è la "c" e l'ultimo il punto esclamativo.
print(stringa1)
- stringa2 = "!oaiC"
La stringa non ha senso, ma anche in questo caso i caratteri sono in ordine.
print(stringa2)
- stringa3 = " "
È una stringa vuota.

print(stringa3)
- stringa4 = “La somma di 3+3...”
La somma tra virgolette non viene considerata un’operazione matematica.
print(stringa4)
- numero = 3+3
Questa non è una stringa, viene eseguita l’addizione.
print(numero)
- num=4
nuovo=num+num
print(nuovo)
cambia_in_stringa=str(nuovo)
r=s+s
print(r)
In questo caso n è un numero, mentre s è una stringa perché n è stato convertito in stringa.
Visualizziamo i comandi di stampa:

```
Ciao!
!oaic
      <-------  questa è la stringa vuota
La somma di 3+3...
6
8
88
```

In Python le stringhe possono essere inserite sia tra doppi apici sia tra singoli apici, ciò risulta vantaggioso nel caso in cui la stringa contenga uno dei due segni:

parola_straniera = ‘ “Shopping” è un termine inglese entrato nel vocabolario italiano ’

la stringa è delimitata dagli apici singoli e contiene una parola evidenziata tra gli apici doppi.

Le stringhe possono essere concatenate, ossia combinate con altre stringhe oppure con variabili di altro tipo per dar luogo a risultati più complessi. I simboli “+” e “*” sono utilizzati per effettuare la concatenazione di stringhe, la virgola “,” si usa con altri tipi di dato , vediamo alcuni esempi:

```
print ('Mi piace "Star Wars"...')

nome = "Mario "#notare lo spazio dopo il nome e prima degli apici
cognome = "Rossi"
print (nome + cognome)#sono state concatenate le variabili
print (nome + "Rossi")#sono state concatenate variabile e stringa
print ("1"+"2") #non sono stati inseriti spazi

print (nome * 3)#è una concatenazione particolare
print ((nome + cognome) * 3)#viene ripetuta la stringa
print ("ciao" * 5)
numero = 5
#per concatenare tipi diversi si usa il simbolo ","
print ("Ho letto",numero, "pagine")
#non si possono concatenare tipi diversi con il simbolo "+"
print ("Ho letto"+numero+ "pagine")#errore!
```

Visualizzazione stampa:

```
Mi piace "Star Wars"...
Mario Rossi
Mario Rossi
12
Mario Mario Mario
Mario RossiMario RossiMario Rossi
ciaociaociaociaociao
Ho letto 5 pagine
ERRORE!
```

Le variabili, infine, possono assumere il valore vero o falso, e in questo caso sono di **tipo booleano**, dal nome del matematico George Boole che nel 1847 considerato uno dei fondatori della logica matematica. Definì un'algebra (l'algebra booleana) basata su due variabili a due valori, vero o falso. Vero o falso sono detti valori di verità, in quanto affermano, oppure negano, le cose, o i fatti, che ci circondano; se un valore è vero non può essere anche falso, uno esclude l'altro.
Consideriamo, per esempio, i seguenti enunciati che esprimono un valore di verità:

Premessa	Conclusione
2*3+2=8	vero
I Minions sono viola	falso, infatti i Minions sono gial
10>5 5>8	vero falso

Gli esempi mostrano che gli enunciati possono essere costituiti da espressioni matematiche o proposizioni, la cui soluzione è ricavata attraverso il ragionamento logico: ci sono delle premesse per le quali si giunge ad una conclusione.
Con gli enunciati è possibile sia eseguire operazioni di confronto, mediante i relativi operatori, sia comporre espressioni booleane, o logiche, attraverso gli operatori booleani. In entrambi i casi il risultato è sempre un valore di verità.

Gli operatori di confronto sono:

Simboli	Operatori	Espressioni vere
==	uguaglianza	2 == 2
!=	disuguaglianza	2 != 5
<	minore	2<5
<=	minore o uguale	x <= 5
>	maggiore	7>5
>=	maggiore o uguale	x >= 5

In questo modo le variabili vengono confrontate e il risultato sarà dato dalla conferma (vero) o dalla negazione (falso) del confronto:

2==2	vero
2 >5	falso
x >= 5	vero o falso dipende dal valore di x
pandistelle != macine	vero, infatti le due stringhe sono diverse
pandistelle == macine	falso, infatti non sono uguali

Occorre fare attenzione a non confondere l'operatore di uguaglianza (==) con l'operazione di assegnamento (=), infatti

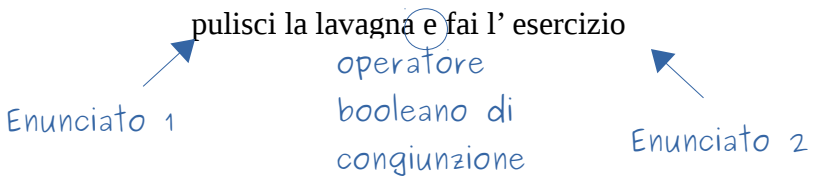
2==2	è un'operazione di confronto
numero = 2	alla variabile numero assegno il valore 2

Gli operatori booleani (o logici) sono:

e	congiunzione	x1 e x2
o	disgiunzione	x1 o x2
non	negazione	non x1

x1 e x2 rappresentano un qualsiasi enunciato:
- con la *congiunzione* "*e*" il risultato è vero solo *se entrambi* gli enunciati sono veri;
- con la *disgiunzione* "*o*" il risultato è vero *se almeno uno* degli enunciati è vero;
- con la *negazione "non"* il risultato è dato dalla negazione dell'enunciato.

Supponiamo che la professoressa chieda a Dario: "pulisci la lavagna e fai l'esercizio". L'espressione "pulisci la lavagna e fai l'esercizio" rappresenta un'espressione booleana:



In questo caso l'insegnante vuole che vengano eseguite entrambe le richieste, per cui l'espressione è vera solo se risultano vere le due condizioni, è falsa in tutti gli altri casi. Per comprendere meglio la situazione, utilizziamo la tabella di verità (o

tabella logica) che mostra le soluzioni dell'espressione booleana partendo dai possibili valori degli enunciati.

I valori di verità sono due (vero o falso) e ciascun valore di verità esclude l'altro, quindi, avendo due enunciati, le combinazioni attendibili sono quattro.
Per semplificare l'elaborato, gli enunciati vengono sostituiti da variabili.

	Enunciati		Espressione booleana
numero	lavagna_pulita	esercizio_svolto	lavagna_pulita e esercizio_svolto
1	V	V	V
2	V	F	F
3	F	V	F
4	F	F	F

1- "lavagna_pulita" e "esercizio_svolto" sono entrambe vere:
Dario ha eseguito gli incarichi a lui assegnati. L'espressione booleana è vera.

2- "lavagna_pulita" è vero e "esercizio_svolto" è falso:
Dario ha solo pulito la lavagna. L'espressione falsa.

3- "lavagna_pulita" è falso e "esercizio_svolto" è vero:
Dario ha solo eseguito l'esercizio. L'espressione è falsa.

4- "lavagna_pulita" è falso e "esercizio_svolto" è falso:
Dario non ha eseguito quanto gli è stato chiesto. L'espressione è falsa.

Con l'operatore di congiunzione il risultato è sempre falso, tranne quando sono veri entrambi gli enunciati.

In Python gli operatori booleani sono **and**, **or** e **not** e i valori di verità sono **True** e **False**. Osserviamo l'algoritmo che rappresenta la terza riga della tabella di verità:

```
lavagna_pulita=False
esercizio_svolto=True

espressione=(lavagna_pulita and esercizio_eseguito)
print(espressione)
```

Visualizziamo il risultato:

```
False
```

Se usiamo l'operatore di disgiunzione, la frase diventa:

pulisci la lavagna o fai l'esercizio

In questo caso per la professoressa è sufficiente che venga eseguita una richiesta, tuttavia possono anche essere realizzate entrambe. Vediamo la tabella di verità:

	Enunciati		Espressione booleana
numero	lavagna_pulita	esercizio_svolto	lavagna_pulita o esercizio_svolto
1	V	V	V
2	V	F	V
3	F	V	V
4	F	F	F

1- "lavagna_pulita" e "esercizio_svolto" sono vere:
Dario ha eseguito entrambi gli incarichi nonostante alla professoressa ne bastasse uno. L'espressione booleana è vera.

2- "lavagna_pulita" è vero e "esercizio_svolto" è falso: Dario ha pulito la lavagna. L'espressione è vera perché per l'insegnante era sufficiente eseguire un solo compito.

3- "lavagna_pulita" è falso e "esercizio_svolto" è vero. L'espressione è vera.

4- “lavagna_pulita” è falso e “esercizio_svolto” è falso: Dario ha fatto finta di non sentire la professoressa e non ha eseguito nemmeno uno degli incarichi richiesti. L’espressione è falsa.
Con l’operatore di disgiunzione il risultato è sempre vero, tranne quando sono falsi entrambi gli enunciati.
Utilizziamo Python e osserviamo l’algoritmo che rappresenta la seconda riga della tabella di verità:

```
lavagna_pulita=True
esercizio_svolto=False

espressione=(lavagna_pulita or esercizio_eseguito)
print(espressione)
```

Risultato:

```
True
```

Con l’operazione di negazione si contraddice quanto affermato:
la frase “Ho fame” diventa semplicemente “Non ho fame”.
In questo caso abbiamo un solo enunciato per cui le combinazioni attendibili sono due, vediamo la tabella di verità:

Numero	Ho fame	Non ho fame
1	V	F
2	F	V

1- “Ho fame” è vero: sento appetito, quindi l’espressione “non ho fame” è falsa.
2- “Ho fame” è falso: non sento appetito, “non ho fame” è vera.
Con l’operazione di negazione viene invertito il valore di verità.

Vediamo l’algoritmo della tabella:

```
ho_fame=True

espressione=not(ho_fame)
print(espressione)
```

Risultato:

```
False
```

Un' espressione è definita espressione booleana solo se contiene operatori logici:
x1 and x2 è un'espressione booleana,
x1 <= x2 non è un'espressione booleana, è un'operazione di confronto
((x1 <= x2) or (x3 + 2)) è un'espressione booleana.

3.2.1 Liste

Una lista può essere considerata come una variabile che può contenere un numero di elementi variabile ma fissato a priori, viene creata da un'istruzione in cui i dati sono racchiusi tra le parentesi quadre:

```
lista=[elemento1, elemento2,elemento3,…]
```

```
prova_lista=["Pippo", 7, [5]]
```

Si noti che i valori della lista possono essere di tipo diverso.
La lista dell'esempio, `prova_lista`, è composta da tre elementi, ciascuno dei quali occupa una posizione individuata da un indice che inizia da zero:

Lista	"Pippo"	7	[5]
Indice	0	1	2

La stringa "Pippo" è in posizione 0, il numero 7 è in posizione 1, il dato tipo lista [5] è in posizione due. Se vogliamo indicare un particolare dato della lista, possiamo selezionarlo nel modo seguente:

```
selezione=lista[indice]
```

Se vogliamo selezionare il numero 7, che corrisponde al secondo elemento di `prova_lista`:

```
selezione=prova_lista[1]
print(selezione)
```

Per conoscere la lunghezza della lista si usa l'istruzione **len**():

```
lunghezza_prova_lista=len(prova_lista)
print(lunghezza_prova_lista)
```

Il comando di stampa può anche essere eseguito direttamente sul dato selezionato:

```
print("L'elemento in posizione 1 è",prova_lista[1])
print("La lista contiene",len(prova_lista,"elementi")
```

La stampa degli ultimi tre comandi è:

```
L’elemento in posizione 1 è 7
La lista contiene 3 elementi
```

3 Input e output

Sono chiamate input le informazioni iniziali riguardanti i dati d’ingresso del programma. Riprendiamo l’esempio della ricetta del tiramisù già preso in esame all’inizio del capitolo, prima di iniziare la preparazione dobbiamo assicurarci di avere tutti gli ingredienti che rappresentano i dati di input del procedimento.
Output invece è il risultato finale del procedimento che, a seconda dei casi, può essere una visualizzazione a video o una stampa su carta vengono.
Riprendendo l’esempio della ricetta l’assemblaggio e la lavorazione degli ingredienti dà origine al tiramisù che rappresenta il prodotto finale della procedura ed è pertanto l’output.

Tutti i programmi per produrre un output hanno bisogno di dati di input i quali, a volte, possono essere richiesti direttamente all’utente che li digita sulla tastiera.
Per esempio, consideriamo il calcolo dinamico dell’area di un triangolo, in cui il programma richiede di inserire la base e l’altezza da tastiera per poi procedere con il calcolo dell’area, così facendo al variare dei dati di input varia il valore dell’output.
Per introdurre i dati da tastiera Python mette a disposizione il comando **input**(). È necessario ricordare che i dati inseriti da tastiera sono di tipo stringa, pertanto occorre convertirli in numero utilizzando le istruzioni **int**() oppure **float**().

```
#calcola l’area di un triangolo

 stringa1=input(“Inserisci la base: “)
 stringa2=input(“Inserisci l’altezza: “)
 #è necessario convertire il tipo stringa in numero
 base=float(stringa1)
 altezza=float(stringa2)

 area_triangolo=float(base*altezza/2)

 #per mostrare il valore dell’area si usa il comando di
 #stampa
 print(“L’area del triangolo è “,area_triangolo)
```

Quando viene avviato il programma, il primo output è il seguente:

```
Inserisci la base:
```

L’utente inserisce il valore e preme invio:

```
Inserisci la base: 6
```

Stessi passaggi per l’inserimento dell’altezza, a seguire l’utente preme nuovamente invio ed ottiene il risultato:

```
Inserisci la base: 6
Inserisci l’altezza: 7
L’area del triangolo è 42
```

3.4 Funzioni

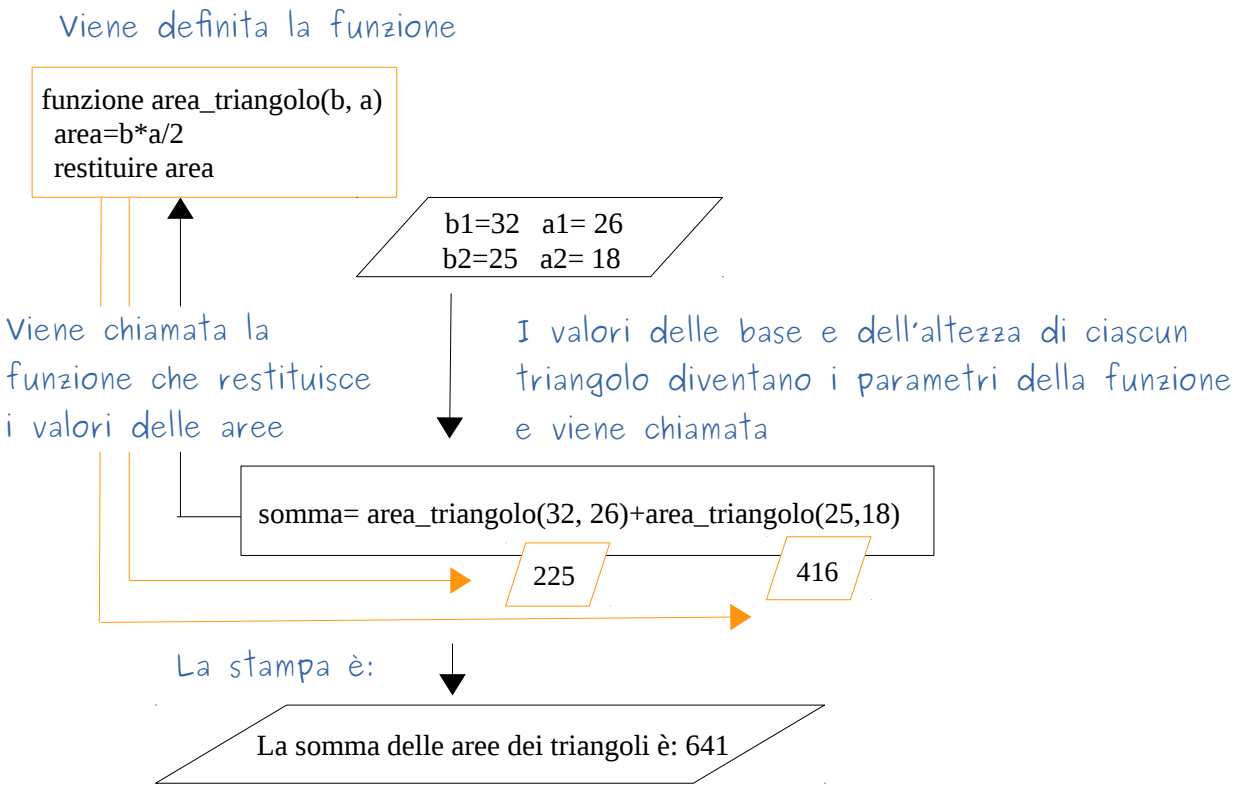
Una **funzione** è composta da un insieme di istruzioni, e restituisce un determinato output, e spesso si ripetono in alcune parti

I programmi solitamente sono lunghi e complessi, pertanto è consuetudine dividerli in vari sotto-programmi ciascuno dei quali fornisce una soluzione che contribuisce al raggiungimento del risultato finale. Una funzione rappresenta un programma che prende come argomento dei dati (detti parametri) e produce dei risultati. I programmi usano le funzioni come sotto programmi**.** Per poter essere utilizzate le funzioni devono essere definite, ossia:

- avere un nome,
- indicare i **parametri**, ossia i dati su cui la funzione esegue le operazioni,
- indicare le istruzioni che portano al risultato.

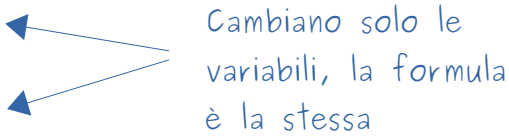
Quando la funzione è utilizzata all'interno del programma si usa l'espressione **chiamata di funzione**.

Consideriamo, per esempio, di dover calcolare la somma delle aree di due triangoli i cui dati sono:
triangolo1 base1=32 e altezza1=26
triangolo2 base2=25 e altezza2=18
Per semplificare saranno usate le variabili b1,a1 e b2,a2.
Descriviamo le fasi del programma per il calcolo dell'area del triangolo mediante un diagramma di flusso:



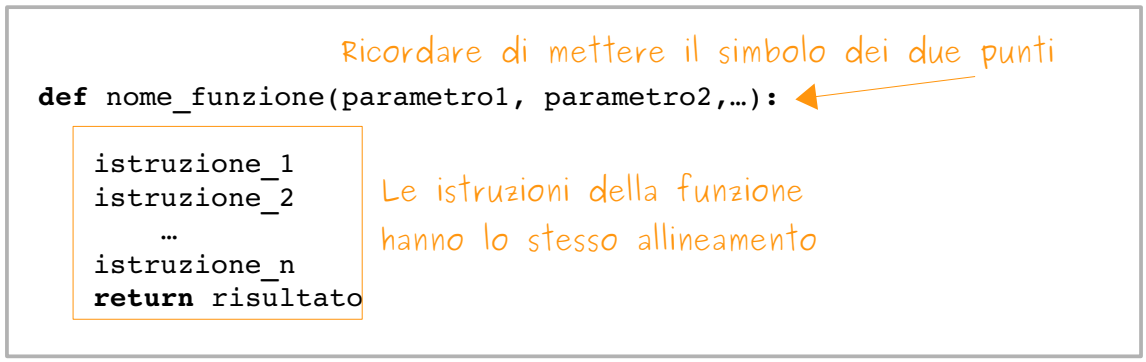
Senza la funzione area_triangolo calcoli che differiscono soltanto per i valori numerici utilizzati dovrebbero essere ripetuti più di una volta, come mostrato di seguito:

step 1: area1 =base1*altezza1/2
restituire area triangolo1
step 2: area2 =base2*altezza2/2
restituire area triangolo2
step 3: somma=area1+area2
stampa(La somma delle aree dei triangoli è: somma)



La suddivisione del programma in funzioni permette di organizzare il lavoro, ne facilita la lettura e favorisce il controllo e la gestione degli errori logici che inevitabilmente si commettono nella stesura di programmi di grandi dimensioni.
Per tutti i linguaggi di programmazione sono disponibili delle funzioni predefinite, ovvero già pronte all'uso, che costituiscono quelle che si chiamano funzioni di libreria. Abbiamo già visto alcuni esempi di Python **print**(), **int**(), **float**(), **str**(), **boolean**(), **input**().

Vediamo la sintassi di Python per definire le funzioni:



Per creare una funzione è necessario utilizzare la parola chiave **def** seguita dal nome che riteniamo adeguato, e dai parametri necessari. Le istruzioni che compongono il corpo della funzione devono seguire una comune indentazione. Se la funzione deve restituire un risultato, viene chiusa dal comando **return**, altrimenti possiamo ometterlo e utilizzare **print**().

Calcola_somma _aree_triangoli.py

```
#viene definita la funzione
  def area_triangolo(b, a):
        area=b*a/2
        return area

  b1=32
  a1=26
  b2=25
  a2=18

  #per fare la somma si ha la chiamata di funzione
  somma= area_triangolo(b1, a1)+area_triangolo(b2, a2)
  print("La somma delle aree dei triangoli è: ",somma)
```

La stampa del risultato:

```
La somma delle aree dei triangoli è: 481
```

5 Cicli iterativi. Ciclo while

Nella stesura di un programma si presenta molto spesso la necessità di ripetere le stesse operazioni, i metodi con cui è possibile fare ciò si chiamano cicli.

Cerchiamo di capire il concetto attraverso un esempio: supponiamo di avere una contenitore con 15 palline e di volerne spostare 5 qualsiasi in una cesta vuota.

Quali sono le azioni da compiere? Supponiamo che debbano essere eseguite da un bambino o a un robot:

1- prendi una prima pallina e spostala nella cesta

2- prendi un'altra e sposta anche questa nella cesta

Ripeti queste azioni fino a quando le cinque palline non sono spostate nella cesta.

Quindi un programma completo che chiamiamo *Pallina_nella_cesta* potrebbe essere il seguente:

```
1 prendere la pallina
   mettere nella cesta
2 prendere la pallina
   mettere nella cesta
3 prendere la pallina
   mettere nella cesta
4 prendere la pallina
   mettere nella cesta
5 prendere la pallina
   mettere nella cesta
 "Bene! La cesta è piena!"
```

Nell'esempio si nota che le istruzioni vengono eseguite 5 volte, ma se le palline da spostare invece di 5 fossero state 100, quante istruzioni avrebbe avuto il nostro programma?

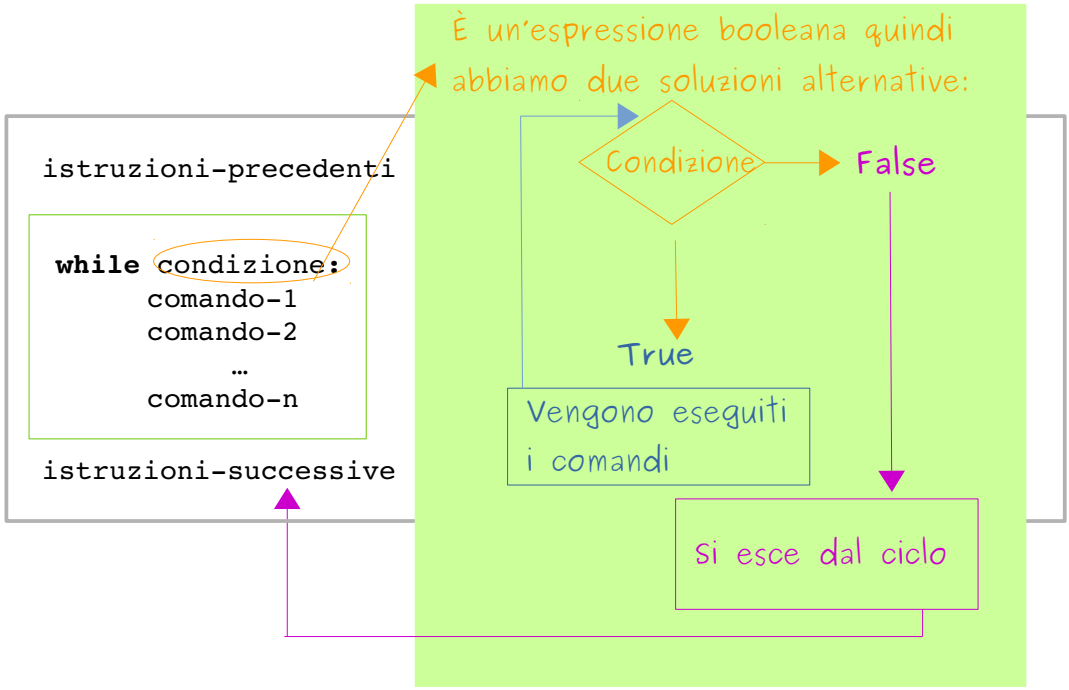
Per facilitarci il compito sono stati introdotti i cicli , con i quali si evita di ripetere la scrittura di istruzioni uguali. Per rendere più leggibile il programma bisogna eliminare le istruzioni ridondanti, e questo si realizza utilizzando i cicli.

Pallina-nella-cesta2

```
1 ripetere per 5 volte
      prendere la pallina
      mettere nella cesta
2 print("Bene! La cesta è piena!")
```

Il programma Pallina-nella-cesta2 è decisamente più semplice, inoltre per cambiare il numero di palline da 5 a 100, basta sostituire 100 a 5 nella prima istruzione "ripeti...". La ripetizione di istruzioni uguali si chiama iterazione.

In Python ci sono due modalità iterative, il comando **while** e il comando **for** che vedremo più avanti. La figura mostra la sintassi del ciclo while.



Viene valutata la condizione, se è vera il ciclo esegue il blocco di comandi (da ricordare che è possibile avere anche un solo comando) che sarà ripetuto finché la condizione risulta vera, altrimenti (se falsa) si esce dal ciclo e vengono eseguite le istruzioni successive.

Per programmare i cicli è fondamentale individuare il numero di volte che il ciclo deve essere eseguito (5 o 100 nel caso delle palline) e le azioni che devono essere iterate, queste ultime possono essere istruzioni qualsiasi come ad esempio assegnamenti, operazioni matematiche, di confronto, comandi **if-else** (nel nostro esempio "prendere la pallina, mettere nella cesta").

Per visualizzare la struttura del ciclo in un programma, ovvero i comandi che devono essere eseguiti nel ciclo iterativo, devono essere indentati rispetto all'istruzione **while**.

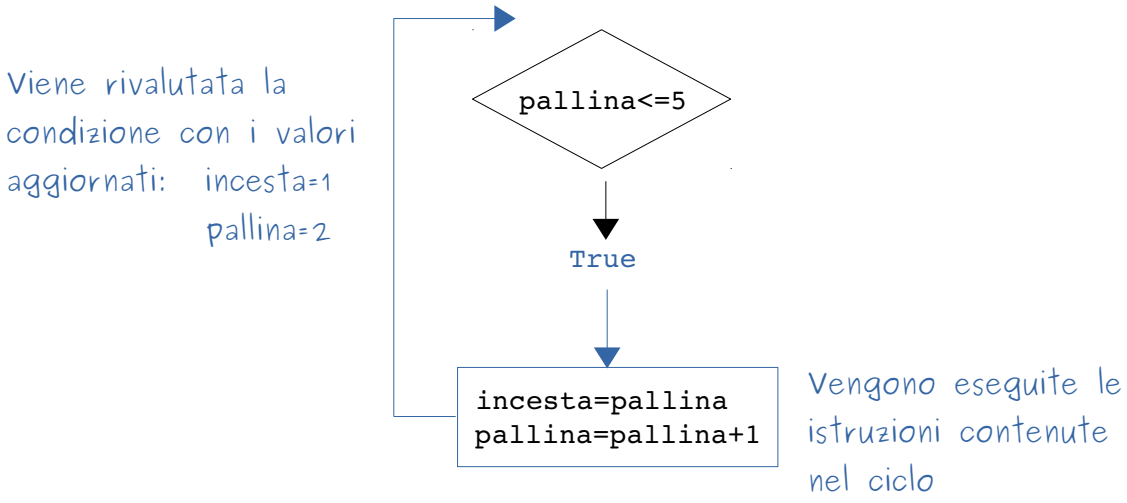
Scriviamo un frammento di programma Python per mettere le cinque palline nella cesta e lo chiamiamo `Pallina-nella-cesta.py`

```
incesta=0
pallina=1

while pallina<=5:#ricordare i due punti ":" dopo la condizione
    #i comandi interni hanno lo stesso allineamento
      incesta=pallina
      pallina=pallina+1#le istruzioni ripetute sono assegnamenti
#i comandi esterni al ciclo seguono un allineamento diverso
print("Bene! La cesta è piena!")
```

Analisi dei passaggi del ciclo.

Iterazione 1

I dati all'inizio del ciclo sono :
`incesta=0`, all'inizio la cesta è vuota,
`pallina=1`, perché prendo la prima pallina; pallina è minore o uguale a 1? Vero, vengono eseguite le istruzioni nel ciclo.

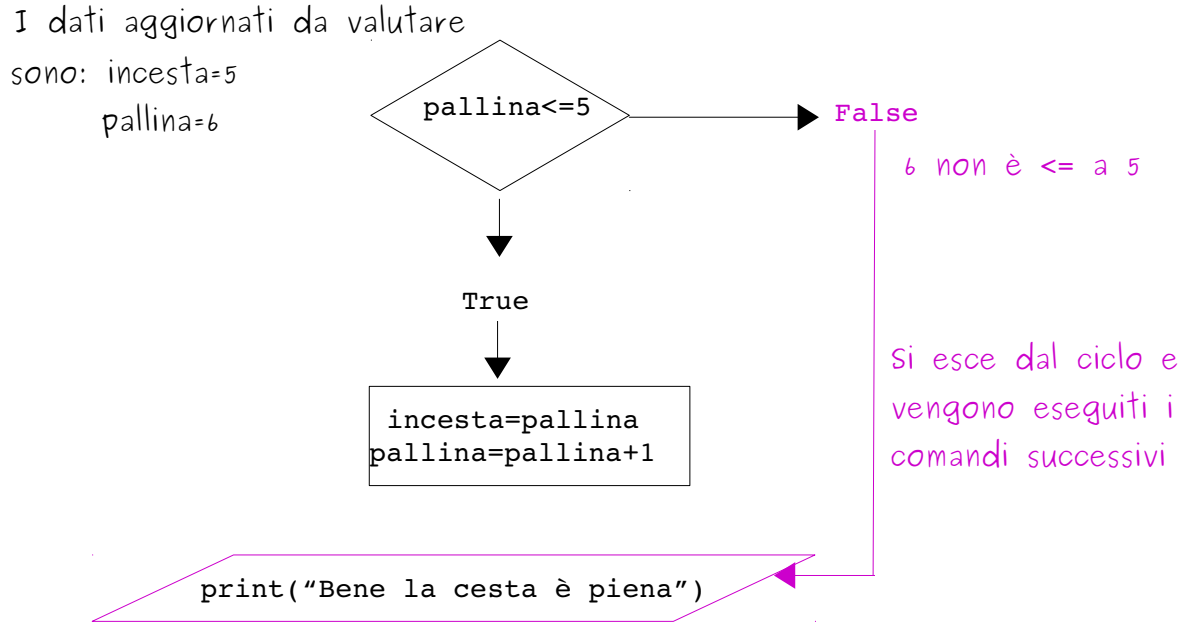


La tabella seguente illustra come variano le variabili ad ogni iterazione del ciclo:

Iterazione	Dati aggiornati	Condizione pallina<=5	Risultato condizione	Ripeti iterazione
1	incesta=0 pallina=1	1<=5	True	si
2	incesta=2 pallina=3	3<=5	True	Si
3	incesta=3 pallina=2	4<=5	True	Si
4	incesta=4 pallina=5	5<=5	True	Si
5	incesta=5 pallina=6	6<=5	False	No

Come si nota nella tabella, al passaggio 5 la condizione è falsa, pertanto le istruzioni che fanno parte del ciclo vengono ignorate e si prosegue con i comandi successivi utilizzando gli ultimi dati aggiornati `incesta=5` e `pallina=6`.

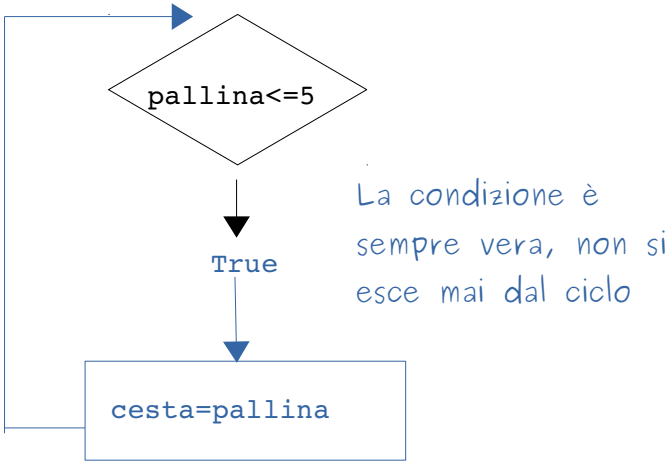
Iterazione 5



Al termine della sua esecuzione il programma `Pallina-nella-cesta.py` produce i seguenti effetti:
- la cesta contiene le cinque palline perché il robot ha eseguito i comandi
- sul display del robot appare la stampa

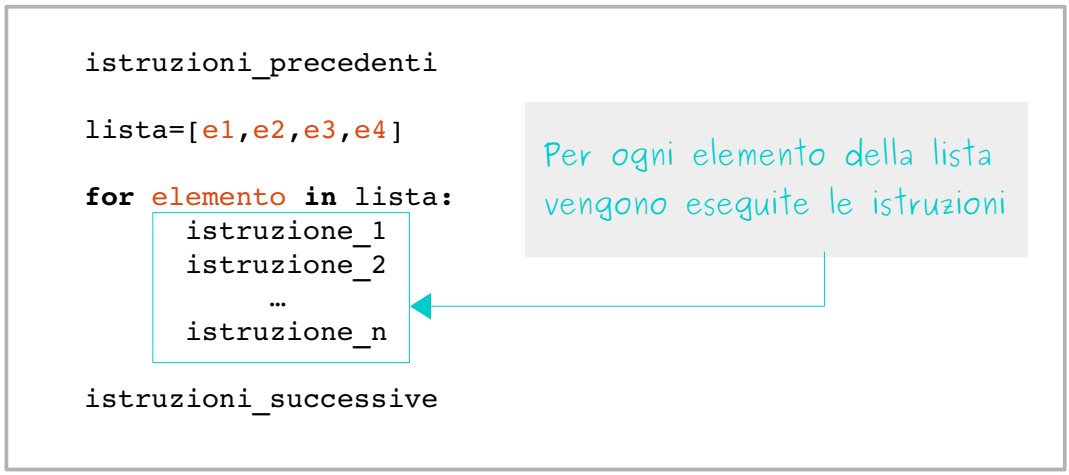
Particolare attenzione deve essere prestata alla definizione della condizione che deve essere controllata ad ogni iterazione e al corretto aggiornamento delle variabili usate in essa. In caso di errore il ciclo potrebbe non terminare mai. Questo errore è tipico dei programmatori alle prime armi.
Nel nostro programma `Pallina-nella-cesta.py`
- la variabile `incesta` ci ha permesso di immagazzinare i dati necessari per conseguire l'obiettivo,
- con pallina abbiamo contato gli elementi da spostare e controllato il ciclo. La variabile pallina determina il risultato della condizione, quindi, a ogni passaggio del ciclo questo valore deve essere incrementato, altrimenti il ciclo sarebbe andato avanti all'infinito. La variabile funziona da contatore.

Nel programma Pallina-nella-cesta.py il contatore è la variabile pallina, proviamo ad omettere l'istruzione che lo aggiorna, ossia pallina=pallina+1.
Il programma elabora i dati iniziali cesta=0 e pallina=1
- esegue l'operazione di confronto,
- il risultato è vero quindi viene eseguito l'unico comando del ciclo,
- il ciclo continua all'infinito perché il contatore non viene aggiornato e la condizione risulta sempre vera!



5.1 Ciclo for

Anche il comando **for** , come il while, permette di iterare un gruppo di istruzioni, in questo caso il ciclo si avvale dei valori di una lista. La sintassi del ciclo for è mostrata di seguito:

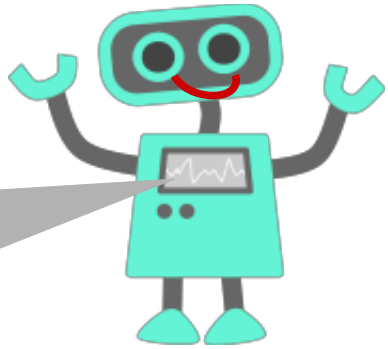


Il significato del costrutto è “per ogni elemento nella lista esegui le seguenti istruzioni”. Nel ciclo le istruzioni devono avere una uguale indentazione, rientrata rispetto all’istruzione for. Quelle che non fanno parte del ciclo, seguono un diverso allineamento.
Supponiamo di aver invitato a casa degli amici e il nostro robot li saluta con un messaggio sul display, vediamo il programma Saluta.py nella Figura...
Notare che la lista si chiama amici e ciascun elemento che la compone è rappresentato dal termine nome.

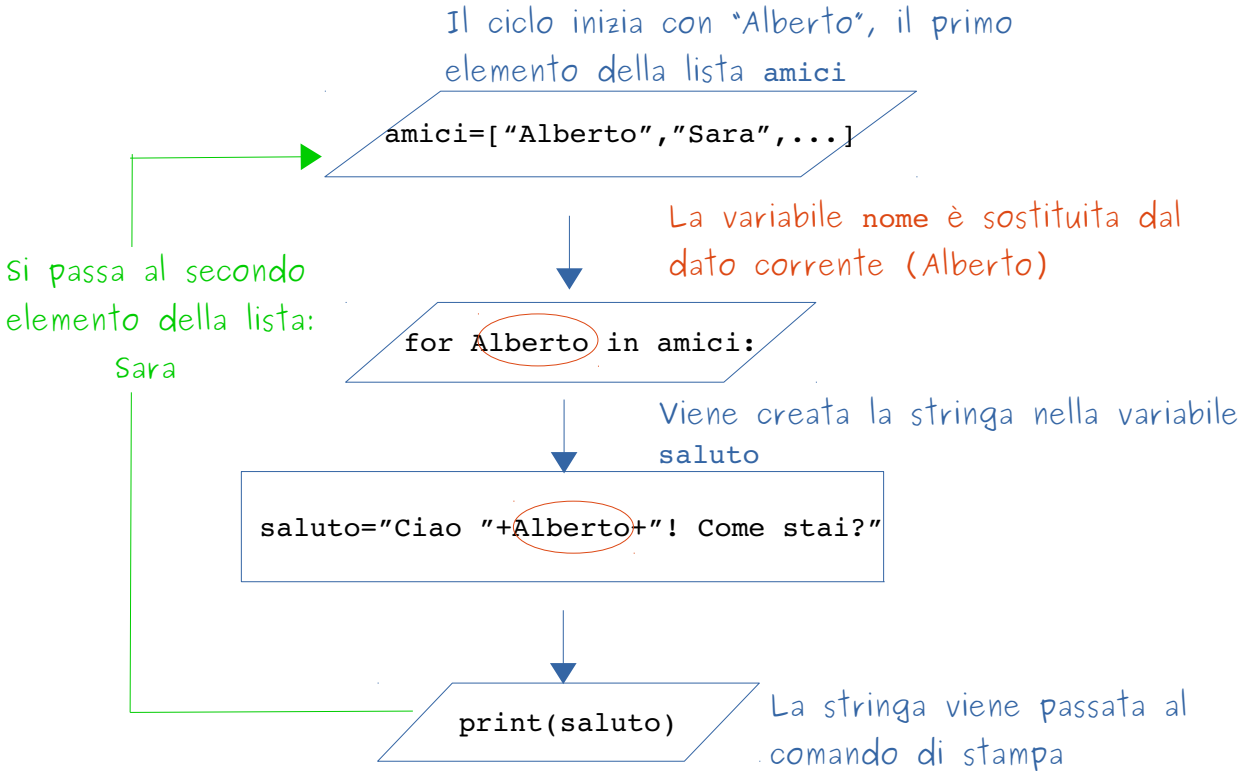
```
amici=[“Alberto”,”Sara”,”Mattia”,”Emma”,”Margherita”]

for nome in amici:
      saluto=”Ciao ”+nome+”! Come stai?”
      print(saluto)

print(“Ho salutato tutti?”)
```

Sul display del robot appare:

```
Ciao Alberto! Come stai?
Ciao Sara! Come stai?
Ciao Mattia! Come stai?
Ciao Emma! Come stai?
Ciao Margherita! Come stai?
Ho salutato tutti?
```



Nello schema seguente viene mostrato il funzionamento del ciclo.



Questi passaggi sono eseguiti per ciascun dato della lista, arrivati all'ultimo, nel nostro caso Margherita, si esce dal ciclo e si passa al comando successivo: `print("Ho salutato tutti?")`.

Mettendo a confronto il ciclo while con il ciclo for, si può notare che il ciclo for esegue un numero di iterazioni predeterminate, mentre el ciclo while le iterazioni si ripetono finché la condizione è vera.

6 Istruzione condizionale if-else

Come nella quotidianità, durante la composizione di un algoritmo, ci troviamo a dovere compiere delle scelte in base a particolari situazioni. Per esempio, a Francesca piacerebbe andare a fare un giro in bicicletta, però deve controllare il tempo, infatti:
-se piove, è meglio rimanere in casa,
-altrimenti (cioè se non piove) può uscire in bicicletta.
Francesca quindi deve scegliere in base al tempo: guarda dalla finestra, vede che sta piovendo e decide di restare a casa. Osserviamo l'algoritmo Tempo_e_scelta:

```
stampa(Francesca deve decidere cosa fare…)
se piove
    stampa(Rimane in casa.)
altrimenti
    stampa(Esce.)

stampa(...è in compagnia del suo cagnolino.)
```

È la condizione in base alla quale viene fatta la scelta

Nel caso in cui piovesse

Alternativa nel caso in cui non piovesse

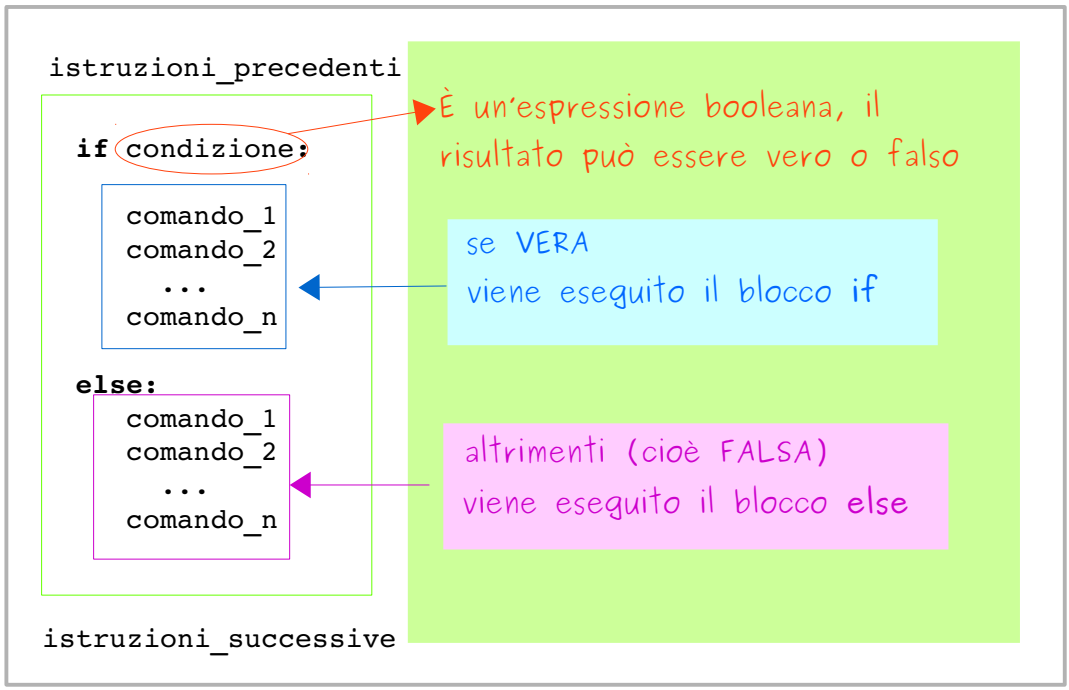
Se la condizione "piove" è vera, l'algoritmo stampa:

```
Francesca deve decidere cosa fare...
Rimane in casa.
...è in compagnia del suo cagnolino.
```

Se invece la condizione "piove" è falsa, si passa direttamente all'istruzione prevista per questo caso, e la stampa è la seguente:

```
Francesca deve decidere cosa fare...
Esce.
...è in compagnia del suo cagnolino
```

In Python abbiamo il comando **if-else**, il cui significato è analogo all'espressione "*se...altrimenti*", e si presenta nel seguente modo:



La condizione è rappresentata da un'espressione booleana:
- se è vera, vengono eseguite le istruzioni del blocco **if**,
- altrimenti, se è falsa, si passa direttamente ai comandi relativi a **else**.

Adesso vediamo `Tempo-e-scelta.py`, l'algoritmo in python.

`piove=False` è il dato di partenza, il valore della variabile `piove` sarà elaborato nella condizione.

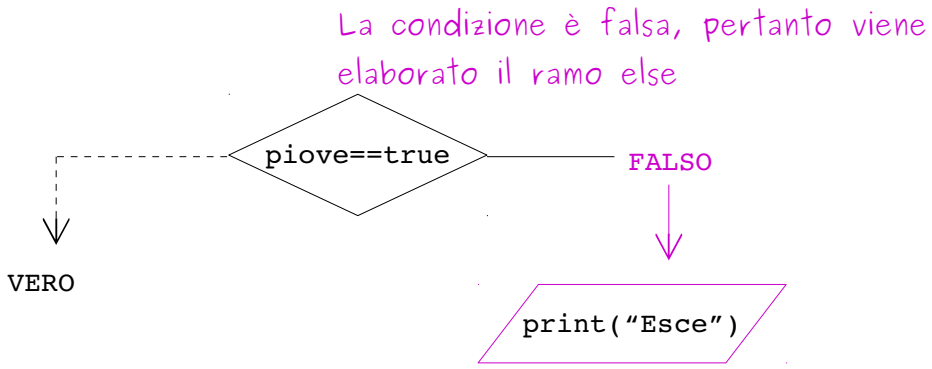
```
#notare l'indentazione differente per distinguere i comandi
 #questa istruzione precedente viene comunque eseguita
  print("Francesca deve decidere cosa fare...")
  piove=False        #non piove

  if piove==True:  #ricordare ":" dopo la condizione
      print("Rimane in casa.")#allineamento uguale a
  else: #ricordare ":"
      print("Esce.")  #allineamento uguale a blocco if

  #questa istruzione successiva viene comunque eseguita
  print("...è in compagnia del suo cagnolino.)
```

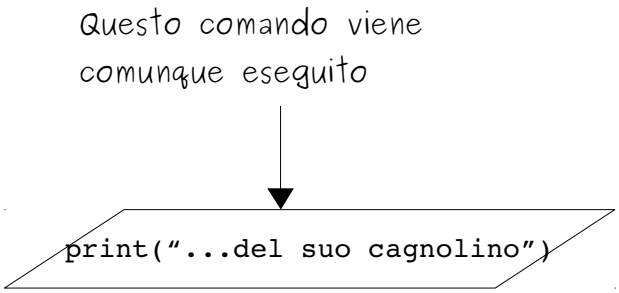
if è sempre seguito dalla condizione e dal simbolo dei due punti, **else** solo dai due punti.
Le istruzioni del blocco if e del blocco else, si distinguono perché seguono un comune allineamento dopo la rispettiva parola chiave, infatti per uscire dal comando condizionale è sufficiente riportare l'indentazione a inizio riga come istruzioni_successive.
La stampa del programma:

```
Francesca deve decidere cosa fare...
Esce.
... è in compagnia del suo cagnolino.
```

Analizziamo il programma col diagramma di flusso.

Il dato di partenza è piove=false, subito dopo viene valutata la condizione in base alla quale viene deciso il passaggio successivo.





Il comando condizionale si può presentare anche in modi differenti:
1. ramo if senza quello else,
2.più comandi condizionali annidati uno dentro l'altro,
3.scelte multiple con il comando elif.

1. Ramo if senza else

Abbiamo soltanto le istruzioni che dipendono da **if**.
Manca il ramo else che rappresenta un'alternativa nel caso in cui la condizione sia falsa.

```
#notare l'indentazione differente per distinguere i comandi
 print("Francesca deve decidere cosa fare...")

 piove=False        #non è una giornata piovosa

 if piove==True:
      print("Rimane in casa.")
 #è stato rimosso il blocco else

 print("...è in compagnia del suo cagnolino.")
```

Stampa:

```
Francesca deve decidere cosa fare...

...è in compagnia del suo cagnolino.
```

2. if-else annidati

Ci sono situazioni in cui le scelte sono subordinate ad altre, e sono annidate una dentro l'altra. In questo caso i costrutti principali **if-else** a loro volta contengono, al loro interno, altri **if-else**.

Mattia deve decidere dove pranzare, il seguente algoritmo prevede delle scelte. Mattia sceglie di mangiare a casa:

Come si vede dall'esempio sopra, abbiamo tre gruppi di **se** ed **altrimenti**, i più esterni contengono gli altri due.

Le scelte, in ogni caso, sono compiute in base alla valutazione delle rispettive condizioni:

- A, mangia a casa,
- B, ha mangiato anche la frutta,
- C, c'è il suo amico Filippo.

La condizione A viene analizzata per prima, è vera, pertanto vengono selezionati i comandi che seguono **se**, però fino alla condizione B, infatti a questo punto viene compiuta un'altra scelta.

Anche B è vera quindi è eseguita l'istruzione vincolata a **se**. L'algoritmo stampa:

```
Mattia mangia a casa o a scuola?
A casa, pasta al pesto e frutta.
A pranzo Mattia non ha voluto la mela!
La mangia a merenda.
```

Scriviamo l'algoritmo in modo formale con Python, `Casa-o-scuola.py`.

```
#notare l'indentazione differente per distinguere i comandi
print("Mattia mangia a casa o a scuola?")

a_casa=True   #Mattia decide di mangiare a casa

if a_casa==True:  #la condizione è valutata ed è vera
    print("A casa, pasta al pesto e frutta.")
    frutta=False
    if frutta==True:#ricordare ":"
        print("A merenda il gelato")
    else:
       print("Però a pranzo Mattia non ha voluto la mela!")
       print("La mangia a merenda.")

else:#ricordare ":"
   print("A scuola...")
   filippo=False #Filippo è assente
   if filippo==True:
       print("E si siede accanto a Filippo.")
   else:
       print("E pranza con i suoi compagni di classe.")
```

La corretta indentazione mette in evidenza i diversi gruppi **if-else** annidati.
È opportuno fare attenzione agli **if-else** annidati in quanto potrebbero diventare troppo complessi e quindi poco leggibili. In presenza di molti **if-else** annidati è possibile che si verifichino errori nel legame tra un **if** e il suo rispettivo **else**. Pertanto è opportuno evitare di avere molti annidamenti multipli.

3- Comando elif

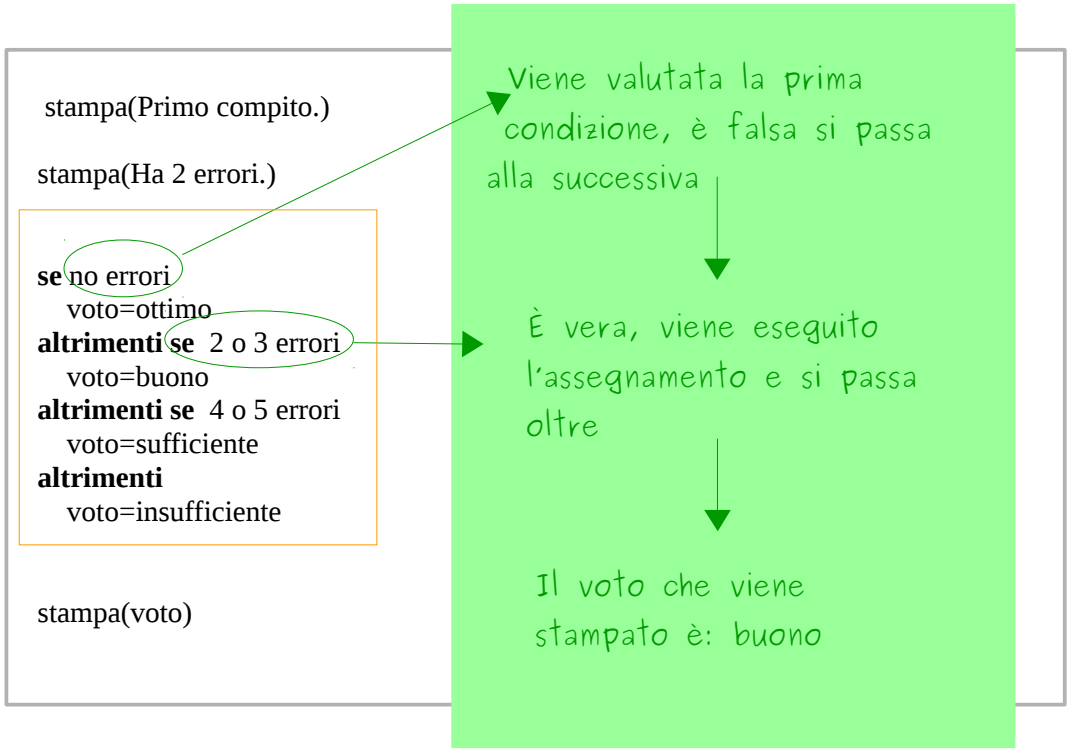
È l'abbreviazione di **else-if** ("*altrimenti...se*") e può essere utilizzata quando ci sono scelte multiple.

Le condizioni sono controllate in sequenza, se la prima è falsa si passa alla successiva, e così di seguito. Trovata quella vera vengono eseguite le relative istruzioni e si prosegue con i comandi esterni al costrutto **elif**. Vediamo un esempio.

La professoressa di matematica ha corretto i compiti e deve assegnare i voti:

- se non ci sono errori il voto è ottimo,
- altrimenti se ci sono 2-3 errori il voto è buono,
- altrimenti se ci sono 4-5 errori il voto è sufficiente,
- altrimenti, con più di 5 errori, è insufficiente.

Supponiamo che il primo compito corretto abbia 2 errori, vediamo l'algoritmo Voto_compito:



La quantità di **elif** che possono essere inseriti non è prestabilita, invece l'else finale deve essere unico.

Il programma `Voto_compito.py` è il seguente:

```
#notare l'indentazione differente per distinguere i comandi
print("Primo compito")
errore=2
print("Errori:",errore)#stampa la stringa seguita dal
                       #numero 2 che
if errore==0:
     voto="ottimo"
elif ((errore==2) or (errore==3)):#ricordare ":"
     voto="buono"
elif ((errore==4) or (errore==5)):
     voto="sufficiente"
else:  #cioè errore>5
     voto="insufficiente"

print("Voto: "+voto)#stampa stringa + variabile
```

La stampa del risultato è:

```
Primo compito
Errori: 2
Voto: insufficiente
```

7 Esempi di applicazione del pensiero computazionale

Programma “Crescente_decrescente_nonOrdinato.py”

Stabilisci se i gruppi seguenti di numeri , composti da quattro numeri ciascuno, sono scritti in ordine crescente oppure in ordine decrescente[24], oppure non sono ordinati.
a) 198, 189, 99, 88
b) 209, 211, 1111, 2111
…..
g) 987, 879, 789, 778
h) 1212, 1220, 2102, 2201

I numeri devono essere confrontati fra loro e sono:
- crescenti quando $a<=b<=c<=d$
- decrescenti quando $a>=b>=c>=d$

Se nessuna delle due condizioni crescente o decrescente è verificata, allora i numeri sono ordinati.

`Crescente_decrescente_nonOrdinato.py` è il programma di tipo funzione che può essere applicato a gruppi di quattro numeri.

```
#Crescente_decrescente_nonOrdinato

 def cresce_decresce_nonordinato(a,b,c,d):
     if a<=b and b<=c and c<=d:

        print("I numeri sono in ordine crescente.")


     else:
        if a>=b and b>=c and c>=d:
             print(“I numeri sono in ordine decrescente.”)
        else:
           print(“I numeri non sono ordinati.”)

 #la funzione viene applicata ai numeri dell’esercizio

 cresce_decresce(198,189,99,88)
 cresce_decresce(34,46,25,78)#i numeri non sono in ordine
 cresce_decresce(34,46,46,78)#ci sono due numeri uguali
```

Descrizione del funzionamento del programma.
In esso possiamo distinguere due sezioni di controllo alternative:

1. Se la condizione `a<=b and b<=c and c<=d` è vera si prosegue con il ramo `if` e l'istruzione di stampa.
2. Se è falsa si passa al ramo `else` che contiene un ulteriore ramo `if-else`:

a>=b and b>=c and c>=d	
True	False
print("...decrescente")	else print(...non ordinati)

Consideriamo per esempio il gruppo di numeri 34 46 25 78.
Si esegue il primo controllo:
- `34<=46 and 46<=25 and 25<=78` è falso pertanto si passa al ramo `else`;
- `34>=46 and 46>=25 and 25>=78` è falso si passa al ramo `else` con l'esecuzione della stampa `I numeri non sono ordinati.`

La figura mostra la stampa dei tre gruppi di numeri a cui è stato applicato il programma `Crescente_decrescente_nonOrdinato.py`

```
I numeri sono in ordine decrescente.
I numeri non sono ordinati.
I numeri sono in ordine crescente.
```

Esercizio.
Sul quaderno prova ad eseguire i passaggi degli altri due gruppi di numeri (198, 189, 99, 88 e 34 46 46 78).

Programma “Proprietà_associativa.py” [25]

In base alla proprietà associativa dell’addizione in un’addizione di più addendi il risultato non cambia se a due o più addendi si sostituisce la loro somma:
$a+(b+c) = (a+b)+c$

Considerata la definizione, per verificare la proprietà associativa deve essere eseguito un confronto fra i due tipi di addizione:
- una nella forma $a+(b+c)$
- l’altra nella forma $(a+b)+c$
La verifica deve essere eseguita su gruppi di addendi uguali.

Il programma `Proprietà_associativa.py` crea tre funzioni, due per definire ciascuna delle forme di addizione ($a+(b+c)$ e $(a+b)+c$) e l’altra per l’esecuzione del confronto.

```
#Proprietà_associativa

 def associativa1(a,b,c):
     somma1=b+c
     somma2=a+somma1
     print(“La somma di”,a,”+(”,b,”+”,c,”) “è,somma2,”.”)

     return somma2#la funzione restituisce un risultato

 def associativa2(a,b,c):
     add1=a+b
     add2=add1+c
     print(“La somma di (”,a,”+”,b,”) +”,c,è”,add2,”.”)
          return add2#la funzione restituisce un risultato

  def confronta (s1,s2) :
      if somma2==add2:
           print(“Il risultato dell’addizione non cambia.”)
      else:
           print(“Sono stati inseriti addendi diversi!”)

  #chiamate di funzione
  #il risultato di associativa1 viene assegnato a s1
  s1=associativa1(6,2,3)
  #il risultato di associativa2 viene assegnato a s2
  s2=associativa2(6,2,3)
  #la funzione confronta viene applicata ai risultati
  confronta(s1,s2)
```

Descrizione del funzionamento del programma.
Per osservare il funzionamento del programma sono utilizzati gli addendi 6, 2, 3.
La funzione `associativa1` elabora e visualizza il calcolo di *a+(b+c)*, e restituisce il risultato nella variabile `somma`:
- alla variabile `somma1` viene assegnato il valore di `b+c`
- alla variabile `somma2` viene assegnato il valore di `a+somma1`
- la funzione restituisce il valore di `somma2`

b+c	somma1	a+somma1	somma2	return somma2
2+3	5	5+6	11	11

La funzione `associativa2` elabora e visualizza il calcolo di *(a+b)+c*:
- alla variabile `add1` viene assegnato il valore di `a+b`
- alla variabile add2 viene assegnato il valore di `add1+c`
- la funzione restituisce il valore di `add2`

a+b	add1	add1+c	add2	return add2
6+2	8	8+3	11	11

L'ultima funzione (`confronta`) fa il confronto tra i risultati delle due precedenti per mostrare che con la proprietà associativa il risultato non cambia:
in questo esempio la condizione `somma==add2` è vera, pertanto viene eseguita l'istruzione del comando **if**.

if somma1==add2	Risultato condizione
11==11	
print("Il risultato...non cambia")	True

La stampa del programma `Proprietà_associativa.py` è la seguente:

```
La somma di 6+(2+3) è 11.
La somma di (6+2)+3 è 11.
Il risultato dell'addizione non cambia.
```

Esercizio.
Sul quaderno prova ad eseguire i passaggi delle tre funzioni utilizzando parametri diversi: associativa1(5,7,9) e associativa2(27,3,11).

I numeri di Fibonacci [26]

I numeri di Fibonacci sono associati a un modello matematico ideale legato alla crescita di una popolazione di conigli:
- una coppia di conigli produce una coppia di coniglietti ogni mese
- per il primo mese i coniglietti non si riproducono
- non muore nessun coniglio.

Ogni numero di Fibonacci è composto dalla somma dei due precedenti come mostra la Tabella 5.
- F1=1
- F2=1
- Fi = F(i-1)+F(i-2) con i>2

Tabella 1

Numero di Fibonacci	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Valore	1	1	2	3	5	8	13	21	34	55

Programma "Primi_10_numeri_di_Fibonacci.py"

Un programma che crea e stampa i primi dieci numeri di Fibonacci può eseguire la somma e la stampa per ciascuno:

```
f1=1          print(f1)
f2=1          print(f2)
f3=f1+f2      print(f3)
 ....
f10=f9+f8     print(f10)
```

Questo procedimento tuttavia può essere migliorato calcolando i valori all'interno di un ciclo come nel programma `Primi_10_Numeri di Fibonacci.py`

```
#Primi_10_Numeri di Fibonacci

print("I primi dieci numeri di Fibonacci sono:")
numeri=[1,2,3,4,5,6,7,,8,9,10]
a=1
b=1
print(a)
print(b)
for i in numeri:
    c=a+b
    print(c)
    a=b #scambia a con b
    b=c #scambia b con c
```

Descrizione del funzionamento del programma.
I due primi numeri F1 e F2 sono i dati di partenza con valore 1, pertanto il ciclo viene ripetuto 8 volte.
All'interno del ciclo viene eseguita la somma e lo scambio dei valori per aggiornare i dati da sommare ad ogni passaggio.

Iterazione	c=a+b	print(c)	scambia a con b	scambia b con c
1	1+1	2	1	2
2	1+2	3	2	3
3	2+3	5	3	5
4	3+5	8	5	8
5	5+8	13	8	13
6	8+13	21	13	21
7	13+21	34	21	34
8	21+34	55	34	55

La visualizzazione del programma `Primi_10_Numeri di Fibonacci.py` appare nel modo seguente:

```
I primi dieci numeri di Fibonacci sono:
1
1
2
3
5
8
13
21
34
55
```

Programma “Numero_di_Fibonacci.py”

Utilizzando i numeri di Fibonacci può essere analizzato un ulteriore esempio, il programma `Numero_di_Fibonacci.py` , in cui viene restituito l’i-esimo numero di Fibonacci dove i è il parametro di input.

Le premesse sono le medesime dell’esempio precedente, i dati di partenza sono dati dalle variabili `a=1` e `b=1` che corrispondono a F1 e F2.

```
#Numero_di_Fibonacci

def fibonacci(n):
   if n<=2:
      #i primi due numeri di Fibonacci sono uguali a 1
      return 1

   print(“Il numero Di Fibonacci corrispondente a”,n,”è:”)
   a=1
   b=1
   i=3#il calcolo parte dal numero 3
   while i<=n:#n è il parametro della funzione
       c=a+b
       a=b
       b=c
       i=i+1
   return b
  #chiamate di funzione direttamente nel comando print
  print(fibonacci(12))
  print(fibonacci(36))
  print(fibonacci(55))
```

Descrizione del funzionamento del programma.

I primi due numeri hanno valore 1, pertanto viene effettuato un primo controllo con il comando `if`.

Il ciclo quindi parte da un valore uguale a 3 (vedi Tabella 1) e si ferma quando supera il valore del parametro inserito nella chiamata di funzione:

fibonacci(n)	if n<=2	while i<=n	numero di cicli
fibonacci(12)	False	3<=12	13
fibonacci(36)	False	3<=36	37
fibonacci(55)	False	3<=55	56

Il funzionamento del ciclo è identico a quello mostrato nella Tabella 6, con la differenza che in questo caso viene restituito il valore di `b` (**`return`** `b`).

La chiamata di funzione avviene direttamente con il comando di stampa **`print`**`(fibonacci(12))`.
Il programma `Numero_di_Fibonacci.py` stampa:

```
Il numero Di Fibonacci corrispondente a 12 è:
144
Il numero Di Fibonacci corrispondente a 36 è:
14930352
Il numero Di Fibonacci corrispondente a 55 è:
139583862445
```

I risultati della stampa mostarno come i numeri di Fibonacci crescono in modo esponenziale, cioè molto velocemente, quindi il programma `Numero_di_Fibonacci.py` non darà più risultato in tempi utile quando *i* diventa molto grande.

Esercizio.
Sul quaderno prova ad eseguire il funzionamento dei programmi `Primi_10_Numeri di Fibonacci.py` e `Numero_di_Fibonacci.py`.

Programma “Moltiplicazione.py”

In questo esempio viene mostrato come si può stampare il risultato di una moltiplicazione di due fattori utilizzando l’operazione di addizione.
La moltiplicazione viene definita in modo seguente: dati due interi positivi *n* e *m*, *n* viene addizionato per *m* volte, quindi *n*m = n+n+n+n+n.*

Per evitare di scrivere inutilmente l’addizione tante volte, è preferibile utilizzare un ciclo nel quale viene eseguita l’addizione. Devono essere considerati i due fattori (*n,m*) e un’altra variabile nella quale vengono raccolti i valori dell’addizione:
- il primo fattore , *n*, deve essere addizionato
- la variabile *prodotto* contiene la somma di n
- il secondo, *m*, deve contare il numero di addizioni da eseguire per questo compie un conteggio alla rovescia da *m* a *0*.

```
#Moltiplicazione
 def moltiplica(n,m):
       prodotto=0
       while m!=0:#il ciclo conta da m a zero
                 prodotto=prodotto+n
                 m=m-1#il ciclo conta da m a zero
       print(“Il risultato è:”,prodotto)

  #per inserire i valori da tastiera
  n=input("Inserisci un fattore")
  m=input("Inserisci un altro fattore")
  #converte input da tastiera in intero
  n_intero=int(n)
  m_intero=int(m)
  moltiplica(n_intero,m_intero)#chiamata di funzione
```

Descrizione del funzionamento del programma `Moltiplicazione.py`.
I dati di partenza (`n,m`) sono inseriti da tastiera tramite la funzione `input()`. È necessario notare che questi valori sono di tipo stringa, pertanto devono essere convertiti in intero per poter eseguire il calcolo. La variabile `prodotto` parte da zero. Come esempio di dati in input utilizziamo 3 e 5.
Il ciclo si arresta quando `m` è uguale a zero.
Fuori dal ciclo viene eseguita la stampa del risultato raccolto in `prodotto`.

passaggio	n	m	prodotto=prodotto+n	m=m-1
1	5	3	0=0+5	3=3-1
2	5	2	5=5+5	2=2-1
3	5	1	10=10+5	1=1-1
4	5	0		

Il programma prima di mostrare il risultato chiede l'inserimento dei dati, l'utente inserisce il primo valore e preme invio:

```
Inserisci un fattore
3
```

Dopo il secondo input (vedere il paragrafo *Input Output*), 5 in questo caso, il programma `Moltiplicazione.py` stampa:

```
Il risultato è: 15
```

Programma “Tabellina_di_n.py”
In questo esempio è mostrato un programma che stampa la Tavola Pitagorica (tabellina) di *n*.

La tabellina riporta i prodotti della moltiplicazione di due numeri interi (*n*,m) da 1 a 10, in cui *n* rimane costante. Consideriamo, per esempio, la tabellina del 3:
3*1=3 3*2= 6 3*3=9 3*4=12 ……. 3*9= 27 3*10=30

Per arrivare alla soluzione si possono eseguire tutte le moltiplicazioni, ma è preferibile utilizzare un ciclo in cui:
- *n* , di cui vogliamo stampare la tabellina per cui rimane costante,
- *m*, il moltiplicatore che deve aggiornare il valore da 1 a 10,
- *prodotto*, contiene il calcolo del prodotto tra *n* e *m*

```
#Tabellina_di_n
 def tabellina(n):
   m=1
   prodotto=0
   print(“La tabellina del”,n,”è:”)
   while m<=10:#parte da 1 e arriva a 10
       prodotto=n*m
       m=m+1
       print(prodotto)

 #per inserire i valori da tastiera
 n=input("Inserisci un fattore")
 #converte input da tastiera in intero
 m=int(n)
 tabellina(m) #chiamata di funzione
```

Descrizione del funzionamento del programma.
Il dato di partenza `n` è inserito da tastiera tramite la funzione **`input()`**.
Il ciclo parte con il moltiplicatore `m=1`, ad ogni passaggio viene aggiornato per eseguire la moltiplicazione e arrivare a 10.

passaggio	n	m	prodotto=n*m	m=m+1
1	3	1	3=3*1	1=1+1
2	3	2	6=3*2	2=2+1
3	3	3	9=3*3	3=3+1
...	...	...	...	...
10	3	10	30=3*10	10=10+1

Il programma prima di mostrare il risultato chiede l'inserimento dei dati, l'utente passa il valore e preme invio:

```
Inserisci un fattore
3
```

Successivamente il programma stampa:

```
3
6
9
12
15
18
21
24
27
30
```

Programma “Moltiplicazione_egizia.py”

È un modo di moltiplicare due numeri trovato nel papiro di Ahmes, dal nome dello scriba che lo trascrisse verso il 1650 a.C. da un originale risalente al Regno Medio, e composto fra il 2000 ed il 1800 a.C.. Si tratta del documento egizio a carattere matematico più esteso: è largo circa 30 cm e lungo 5,46 m. Pare che questo documento fosse un testo di carattere didattico, orientato alle applicazioni pratiche. Secondo alcuni si tratta di un libro scolastico, secondo altri del taccuino di un allievo.

Prevede il dimezzamento progressivo di uno dei due fattori (ignorando eventuali resti) e, parallelamente, la duplicazione dell’altro fattore. Probabilmente queste operazioni si eseguivano in maniera semplice sull’abaco, strumento di calcolo dello scriba egiziano.

Consideriamo per esempio 25*17=425, il calcolo viene eseguito con un ciclo che prevede a ogni passo:
- divisione per 2 del moltiplicatore;
- moltiplicazione per 2 del moltiplicando;
finché il moltiplicando non si azzera. Il prodotto viene calcolato partendo da 0 e sommando a ogni passo del ciclo il valore parziale del moltiplicatore se il moltiplicando è dispari.

Il calcolo può essere seguito con l’ausilio della Tabella 2:
- nella colonna di sinistra si mostrano i valori successivi del moltiplicando
- nella colonna centrale i valori del moltiplicatore
- nella colonna di destra i valori del prodotto.

```
#Moltiplicazione_egizia
def m_egizia(n,m):
    prodotto=0
    while n!=0:
          if n%2!=0:#se n è dispari viene eseguita la somma
              prodotto=prodotto+m
          n=n//2#viene eseguita la divisione intera per due
          m=m*2#m viene sempre moltiplicato per due
    return prodotto

 risultato=m_egizia(25,17)#chiamata di funzione
 print(“Il risultato è:”,risultato)
```

La Tabella 2 mostra l’esecuzione del programma.

Tabella 2

n//2	m*2	prodotto=prodotto+m
25	17	17=0+17
12	34	17
6	68	17
3	136	153=17+136
1	272	425=153+272
0		

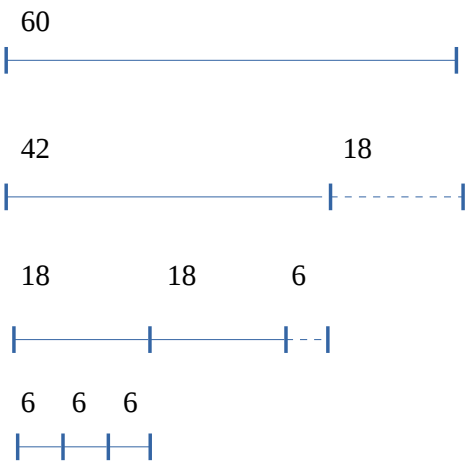
Stampa del risultato:

```
Il risultato è: 425
```

Programma “MCD_Euclide.py”

Per determinare il massimo comun divisore tra due interi qualsiasi, MCD(a,b), i numeri devono essere scomposti in fattori primi e il risultato è dato dal prodotto di tutti i fattori primi comuni ai due.
Il MCD(60,42) è 6, infatti:
60=2*2*3*5 e 42=2*3*7 , i fattori primi in comune sono 2*3.
Per risolvere questo problema si utilizza l’algoritmo di Euclide, un algoritmo antichissimo.

Per calcolare il MCD(60,42) si costruiscono due segmenti di lunghezza 60 e 42:
- si toglie il minore dal maggiore per quante volte è possibile, in questo caso una.
- se vi è un segmento residuo(il 18, che rappresenta il resto della divisione tra 60 e 42) si toglie il residuo dal più piccolo dei segmenti precedenti (il 42) per quante volte è possibile (due volte)
- si ripete il procedimento finché non vi è più residuo.
- l’ultimo segmento costruito (il 6) dà il massimo comun divisore tra i segmenti originali.[27]



Osservando la figura, si può osservare che la divisione intera (modulo) viene ripetuta finché c’è il resto. L’ultimo costituisce il MCD. (Tabella 3)
È necessario quindi :
- fare un ciclo per ripetere la divisione
- considerare due variabili *a* e *b* che scambiano i valori all’interno del ciclo per poter eseguire le divisioni
- e una terza *r* che mantiene aggiornato il valore del resto

Tabella 3

a	b	a%b=r
60	42	60%42=18
42	18	42%18=6
18	42	18%6=0

```
#MCD_Euclide
def mcd_euclide(a,b):
  print("MCD tra",a,"e",b,)
  while(b != 0):
      r=a%b
      a=b #scambia a con b
      b=r
  return a

print("Il MCD è",mcd_euclide(60,42))
```

Descrizione del funzionamento del programma `MCD_Euclide`.
Le variabili `x` e `y` conservano i valori di `a` e `b` (60,42) per essere utilizzati nella stampa del risultato.
All'interno del ciclo viene reiterata l'operazione di modulo mediante gli scambi dei valori tra le variabili.
Il ciclo termina quando la divisione dà resto zero. La funzione restituisce `a`, la variabile messa in evidenza col colore rosso.

a	b	r=a%b	Scambia a con b	Scambia b con r
60	42	18	42	18
42	18	6	18	6
18	6	0	6	0

Il risultato del programma è:

```
Il MCD tra 60 e 42 è: 6
```

Programma “Inverti_lista.py”

Data una certa lista, vogliamo stampare gli elementi che la compongono in ordine inverso, e mostrarli come tipo stringa:
gusti_gelato=[Cioccolato, Fragola,Nocciola, Fiordilatte]
gusti_gelato_invertiti=Fiordilatte, Nocciola, Fragola, Cioccolata

Per fare questa inversione occorre utilizzare un ciclo in cui:
- si parte dall’ultimo elemento della lista gusti_gelato per arrivare al primo,
- gli elementi selezionati in ciascun passaggio vengono assegnati ad una nuova variabile.

```
#Inverti_lista

gusti=["Cioccolato","Fragola",”Nocciola”,”Fiordilatte”]
i=len(gusti)-1
gusti_invertiti=””#variabile tipo stringa
while i >=0:
      elemento=gusti[i]
      gusti_invertiti=gusti_invertiti+””+elemento+”, ”
      i=i-1
print(gusti_invertiti)
```

Descrizione del funzionamento del programma.
Nelle liste gli elementi sono numerati partendo dallo zero, come mostrato nella Tabella 4.
La funzione **len**() restituisce la lunghezza della lista, quindi, per ottenere l’ultimo elemento della lista, occorre fare `len(gusti)-1`. In questo modo il ciclo parte con l’indice `i` uguale a 3.

Tabella 4

Elemento 0	Elemento 1	Elemento 2	Elemento 3
Cioccolato	Fragola	Nocciola	Fiordilatte

`gusti[i]` sono gli elementi della lista che vengono assegnati alla variabile `elemento` (Tabella 15) ad ogni passaggio del ciclo.
Nella variabile `gusti_invertiti` (Tabella 5) vengono concatenati gli elementi in senso inverso rispetto a `gusti`.

Tabella 5

i	elemento	gusti_invertiti
3	Fiordilatte	Fiordilatte
2	Nocciola	Fiordilatte, Nocciola
1	Fragola	Fiordilatte, Nocciola, Fragola
0	Cioccolato	Fiordilatte, Nocciola, Fragola,Cioccolato

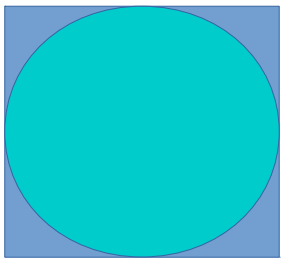
La stampa di `gusti_invertiti` mostra gli elementi tipo stringa.
Il risultato del programma è:

```
Fiordilatte, Nocciola, Fragola, Cioccolata
```

Programma “Circonferenza.py”

Come mostrato nella figura di lato, un quadrato contiene un cerchio.
Il lato del quadrato misura 30 cm di lato, qual è la circonferenza del cerchio? [28]

30



Conoscendo la misura del lato del quadrato conosciamo anche il diametro del cerchio dal quale possiamo ricavare la circonferenza:
*diametro=raggio*2*
circonferenza = raggio x 2 x Π

```
#Circonferenza

lato_quadrato=30
diametro=lato_quadrato
pi_greco=3.14
circonferenza=diametro*pi_greco
print(“La circonferenza del cerchio è: ”,circonferenza)
```

Descrizione del funzionamento del programma Cerchio.py.
Il dato di input è il valore del lato del quadrato, pertanto alla variabile `diametro` viene assegnato il valore di `lato_quadrato`. Avendo a disposizione i dati necessari si procede al calcolo della circonferenza. Il programma stampa:

```
La circonferenza del cerchio è: 94,2
```