Fondamenti di Informatica

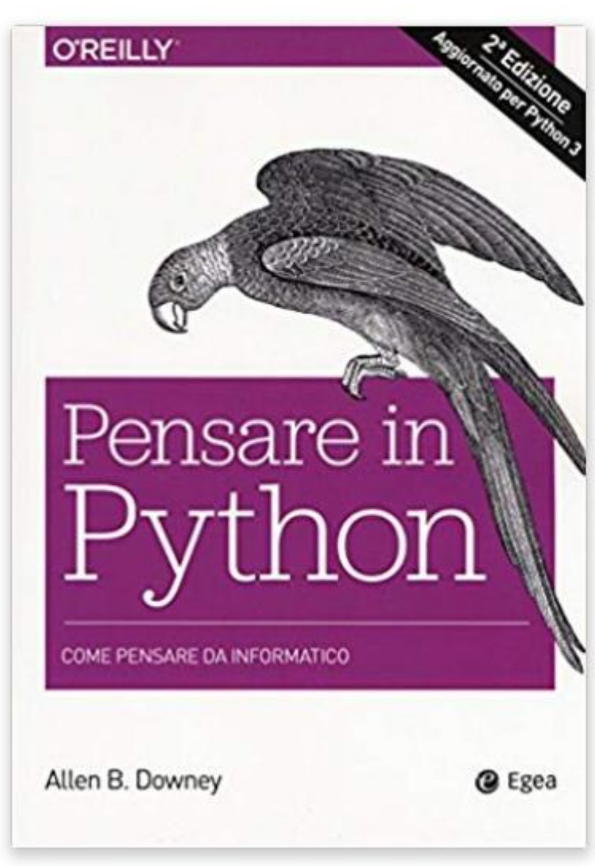
AA 2019/2020

Eng. Ph.D. Michela Paolucci

DISIT Lab http://www.disit.dinfo.unifi.it
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Libro di Testo



- **Titolo del Libro: Pensare in Python**
- **Autore : Allen Downey**
- **Editore: EGEA**
- **Data di Pubblicazione: 2018**
- **Genere: libro. elaborazione dati**
- **Argomento : Python, linguaggio**
- **ISBN-10: 8823822645**
- **ISBN-13: 9788823822641**

Bibliografia –

- Concetti di informatica e fondamenti di Python, Cay Horstmann, Rance D. Necaise, Apogeo
- Pensare in Python - Come pensare da Informatico, Allen Downey, Green Tea Press
- Pensare da informatico - Imparare con Python, Allen Downey Jeffrey Elkner Chris Meyers
- …

Programma del Corso

Anno Accademico 2019-20

Laurea Triennale (DM 270/04) - INGEGNERIA GESTIONALE

Programma del corso - Cognomi A-H

▪ Introduzione al linguaggio python
o Tipi, variabili e costanti
o Operatori ed espressioni
o Istruzioni
▪ Rappresentazione dei dati
o Numeri
o Interi
o Caratteri e stringhe
Elementi di sintassi di un linguaggio

Esecuzione di programmi e ambienti: notebooks, IDE, console
▪ Il linguaggio python
o tipi mutabili e immutabili
o operatori ed espressioni
o istruzioni
o funzioni
o cicli while e for
o esecuzione condizionale
▪ Strutture dati e algoritmi elementari: Liste, Dizionari, Insiemi, iterazioni su strutture dati
Costo di esecuzione e complessità
Il modello di costo
Cenni sulla complessità di un algoritmo:
▪ Algoritmi di ordinamento su vettori
o Sequential-sort
▪ Cenni sugli alberi
o Alberi
o Alberi binari di ricerca: i) Visita in forma ricorsiva; ii) Ricerca; iii) Inserimento ordinato
Cenni su analisi dei dati, lettura e scrittura di file in forma tabulare, grafici.

Pagina del Corso

http://www.disit.org/drupal/?q=node/7020

Qui trovate:

- AVVISI
- Slide del corso
- Approfondimenti

D T

Distributed Systems and Internet Technologies Lab
Distributed Data Intelligence and Technologies Lab
Department of Information Engineering (DINFO)
University of Florence

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

http://www.disit.dinfo.unifi.it

HOME ABOUT RESEARCH INNOVATION EDUCATION AND COURSES HOWTO EVENTS

CORSO DI FONDAMENTI DI INFORMATICA, TRIENNALE, GESTIONALE E MECCANICA A-L, AA 2018/2019

AVVISI

ATTENZIONE: l'esame orale relativo all'appello del 23 Gennaio si terrà in data 30 Gennaio:
- aula 007, viale morgagni ore 9:00

LIBRO DI TESTO AA 2019/2020

- Allen Downey. Pensare in Python. EGEA

ORARIO DEL CORSO AA 2019/2020

L'orario è consultabile a questo link

SLIDE DEL CORSO AA 2019/2020

Si ricorda che le slide del corso NON sono in alcun modo sostitutive del libro di testo.

Modalità d’esame – alcune linee guida -

L'esame si compone di una prova scritta e una orale.

La prova scritta è svolta su carta A4.

Si accede alla prova orale solo se la parte di programmazione è corretta e funzionante

La prova orale può essere sostenuta a partire dalla settimana seguente alla prova scritta, non oltre la prova scritta successiva.

La prova orale inizia con la discussione dell'elaborato, e prosegue con l'approfondimento di tutti i contenuti del programma del corso.

Orario del Corso e Ricevimento

Docente: PAOLUCCI MICHELA

Ingegneria FIRENZE - A.A. 2019/2020 - 2° periodo

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
08:15						
09:15						
10:15						
11:15	FonDiInf(A-L) (Auditorium B - C.D.M.) FonDiInf(A-L) (001 - C.D.M.)			FonDiInf(A-L) (Auditorium A - C.D.M.) FonDiInf(A-L) (001 - C.D.M.)		
12:15	FonDiInf(A-L) (Auditorium B - C.D.M.) FonDiInf(A-L) (001 - C.D.M.)			FonDiInf(A-L) (Auditorium A - C.D.M.) FonDiInf(A-L) (001 - C.D.M.)		
14:00				FonDiInf(A-L) (Auditorium A - C.D.M.) FonDiInf(A-L) (001 - C.D.M.)		
15:00						
16:00						
17:00						
18:00						

Legenda: C.D.M.: Centro Didattico Morgagni

- Il ricevimento si svolge su appuntamento contattando la docente via e-mail:
  - michela.paolucci@unifi.it

Python : Concetto di Funzione

Esecuzione di programmi e ambienti: notebooks, IDE, console
▪ Il linguaggio python
o tipi mutabili e immutabili
o operatori ed espressioni
o istruzioni
o funzioni
o cicli while e for
o esecuzione condizionale
▪ Strutture dati e algoritmi elementari: Liste, Dizionari, Insiemi, iterazioni su strutture dati
Costo di esecuzione e complessità
Il modello di costo
Cenni sulla complessità di un algoritmo:
▪ Algoritmi di ordinamento su vettori
o Sequential-sort
▪ Cenni sugli alberi
o Alberi
o Alberi binari di ricerca: i) Visita in forma ricorsiva; ii) Ricerca; iii) Inserimento ordinato
Cenni su analisi dei dati, lettura e scrittura di file in forma tabulare, grafici.

Funzioni (1)

- Una funzione è una sequenza di istruzioni che esegue un calcolo, alla quale viene assegnato un nome
- Per definire una funzione, dovete specificarne il nome e scrivere la sequenza di istruzioni
- In un secondo tempo, potete "chiamare" la funzione mediante il nome che le avete assegnato
- Abbiamo già visto un esempio di una chiamata di funzione:

```
>>> type(32)
<type 'int'>
```

- Il nome di questa funzione è type
- L'espressione tra parentesi è chiamata argomento della funzione (INPUT), e il risultato che produce è il tipo di valore dell'argomento che abbiamo inserito
- Si usa dire che una funzione "prende" o "riceve" un argomento e, una volta eseguita l'elaborazione, "ritorna" o "restituisce" un risultato
- Il risultato è detto valore di ritorno (OUTPUT)

Funzioni (2)

- Altri esempi di funzioni che abbiamo già visto sono quelle usate per interagire con la console
  - print() e input()

- Esempio1:

```
messaggio = "Hello World!"
print(messaggio)
```

- print prende in ingresso la variabile messaggio di tipo stringa e scrive tale stringa su console

- Esempio2:

```
res = input("Come ti chiami?")

print(res)
```

#input prende in ingresso la stringa digitata dall'utente su console e come output rende tale stringa.

- Il programmatore in questo caso, si preoccupa poi di usare l'operatore di assegnazione per assegnare alla variabile res l'output della funzione input (ovvero la stringa digitata in console), per poi darla in input alla funzione print e stamparla nuovamente in console

Funzioni di conversioni di tipo (1)

- Python fornisce una raccolta di funzioni predefinite o built-in, che convertono i valori da un tipo all'altro
- La funzione int prende un dato valore e lo converte, se possibile, in intero (CAST)
- Se la conversione è impossibile compare un messaggio d'errore:

```
>>> int('32') #come input si passa una stringa che Python riesce a convertire in un intero
32
>>> int('Ciao') #come input si passa una stringa che Python NON riesce a convertire in un intero
ValueError: invalid literal for int() with base 10: 'Ciao'
```

- int può anche convertire valori in virgola mobile in interi, ma non arrotonda bensì tronca la parte decimale

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Funzioni di conversioni di tipo (2)

- La funzione float converte interi e stringhe in numeri a virgola mobile:

```
>>> float(32) #come input si passa un intero
32.0
>>> float('3.14159’) #come input si passa una stringa
3.14159
```

- Infine, str converte l’argomento in una stringa:

```
>>> str(32)
'32’
>>> str(3.14159)
'3.14159'
```

Funzioni come 'Scatole Nere' (1)

- Una funzione è una sequenza di istruzioni che esegue un calcolo, alla quale viene assegnato un nome
- Una funzione viene invocata o chiamata al fine di eseguire la serie di istruzioni di sua competenza
- Supponiamo che la seguente riga di codice faccia parte di un programma:

```
>>>price = round(2.546773, 3) #assegna il valore in virgola mobile 2.546 alla variabile price
```

- Scrivendo tale riga di codice, il programma INVOCA (o CHIAMA) la funzione
- round() restituisce il risultato della propria elaborazione al punto del programma in cui la funzione è stata invocata. In questo caso quindi rende un valore che viene ASSEGNATO alla variabile price
- Dopodiché il programma riprende il proprio flusso di esecuzione

Funzioni come ‘Scatole Nere’ (2)

Input1: 2.546773

Input2: 3

round(var1, var2)

Output: 2.54

INPUT

OUTPUT

Ci ricorda…

DATI → ELABORATORE / ALGORITMO → RISULTATI

Funzioni come ‘Scatole Nere’: diagramma di flusso

Flusso di esecuzione del programma principale

#Cosa si scrive nel programma

#Python

…

price = round(2.546773,3)

…

Flusso di esecuzione della invocazione di una funzione

Il programma chiama la funzione round e passa come input **2.546773 e 3** ‘round(2.546773,3)’

round

Arrotonda **2.546773 a 3** cifre decimali

Restituisce **2.546** all’invocante della funzione

Assegna il valore **2.546** alla variabile price ‘price = …’

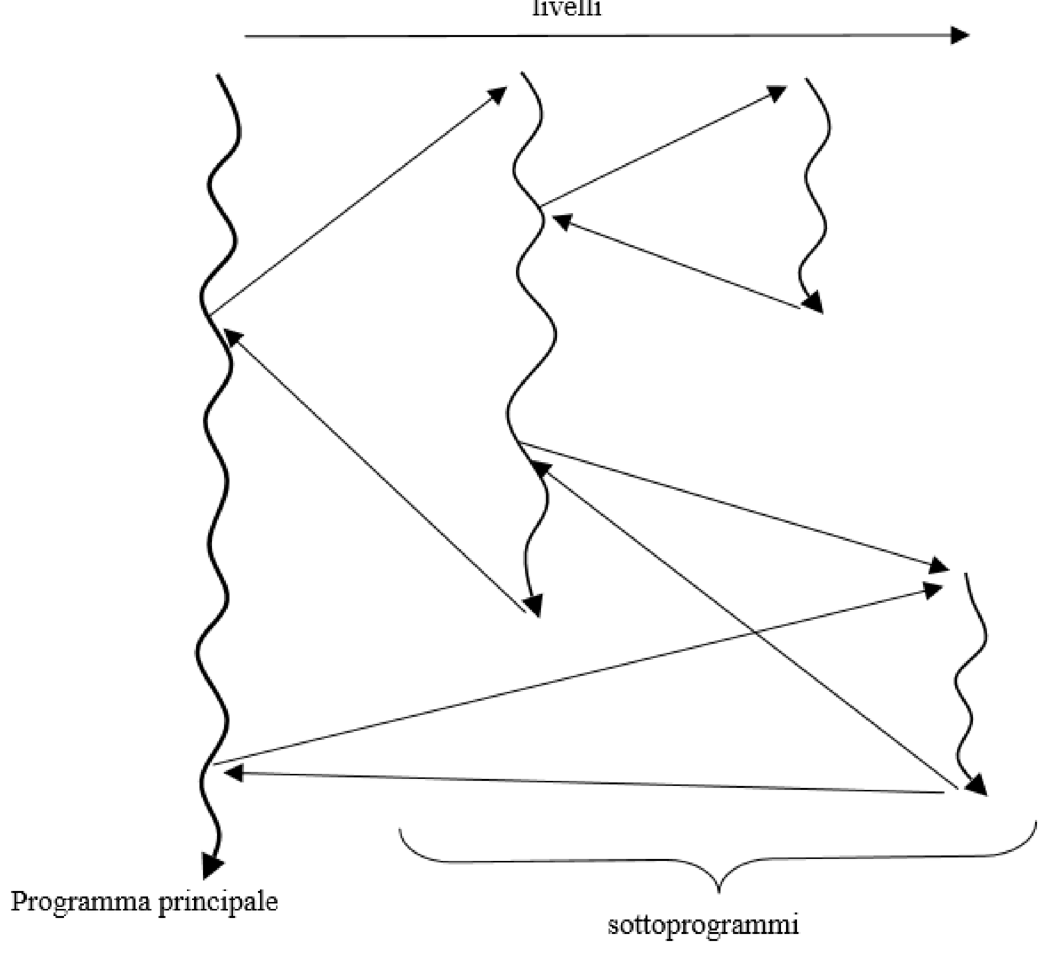
Flusso di esecuzione della funzione round

Flusso di esecuzione (1)

- L'esecuzione inizia sempre dalla prima riga del programma e le istruzioni sono eseguite una alla volta dall'alto verso il basso
- Le definizioni di funzione non alterano il flusso di esecuzione del programma ma va ricordato che le istruzioni all'interno delle funzioni non vengono eseguite fino a quando la funzione non viene chiamata
- Una chiamata di funzione è una sorta di deviazione nel flusso di esecuzione:
  - Invece di proseguire con l'istruzione successiva, il flusso salta alla prima riga della funzione chiamata ed esegue tutte le sue istruzioni
  - alla fine della funzione il flusso riprende dal punto dov'era stato deviato
- Sinora è tutto abbastanza semplice, ma dovete tenere conto che una funzione può chiamarne un'altra al suo interno. Nel bel mezzo di una funzione, il programma può dover eseguire le istruzioni situate in un'altra funzione. Ma mentre esegue la nuova funzione, il programma può doverne eseguire un'altra ancora!

Flusso di esecuzione (2)

- Fortunatamente, Python sa tener bene traccia di dove si trova, e ogni volta che una funzione viene completata il programma ritorna al punto che aveva lasciato
- Giunto all'ultima istruzione, dopo averla eseguita, il programma termina
- QUINDI:
- Quando leggete un programma non limitatevi sempre a farlo dall'alto in basso. Spesso ha più senso cercare di seguire il flusso di esecuzione



Funzioni come ‘Scatole Nere’: diagramma di flusso

Flusso di esecuzione del programma principale

Flusso di esecuzione della invocazione di una funzione

Il programma chiama la funzione round e passa come input **2.546773 e 3** ‘round(2.546773,3)’

round

Arrotonda **2.546773 a 3** cifre decimali

Restituisce **2.546** all’invocante della funzione

Assegna il valore **2.546** alla variabile price ‘price = …’

Flusso di esecuzione della funzione round

```
#Cosa si scrive nel programma
#Python
…
price = round(2.546773,3)
…
```

Funzioni come ‘Scatole Nere’ (3)

- Domanda: Come fa la funzione round() a portare a termine il proprio compito?
- In realtà non ci deve interessare, questo perché stiamo chiamando una funzione predefinita di Python, significa che un programmatore prima di noi si è occupato di scrivere le righe di codice Python necessarie per permetterci di ottenere il risultato voluto
- Per chi chiama la funzione è necessario conoscere le specifiche della funzione stessa: «Se si forniscono gli argomenti x, n la funzione restituisce il valore x arrotondato a n cifre decimali»
- Ci si riferisce spesso ad un dispositivo di cui si conoscono le specifiche ma di cui NON si conoscono i dettagli realizzativi (nel nostro caso le righe di codice), chiamando ‘Black Box’ o Scatola Nera
- Quindi anche la funzione round(…) può essere immaginata come una Scatola Nera
- Quando si progetteranno funzioni, queste dovranno essere chiamate da altri programmi o da altri programmatori, che allo stesso modo potranno usarle senza sapere cosa succede al suo interno, ma solo conoscendo: nome della funzione, INPUT e OUTPUT (e dove si trova la funzione….)

Funzioni come 'Scatole Nere' (4)

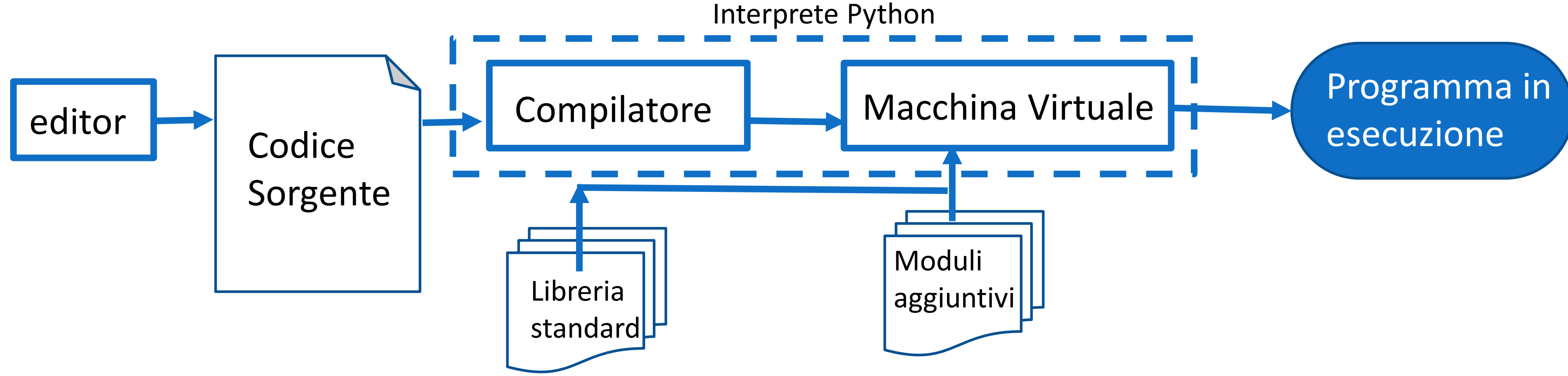
- Le funzioni possono ricevere più argomenti in ingresso (INPUT) MA restituiscono un solo valore (OUTPUT)
- Inoltre possono esistere funzioni che NON hanno parametro di INPUT
- Un esempio di funzione che NON necessita di parametro di input, è la funzione help() che, se chiamata fornisce una serie di indicazioni per lavorare in Python

Prompt dei comandi - python

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 3.8's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.8/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

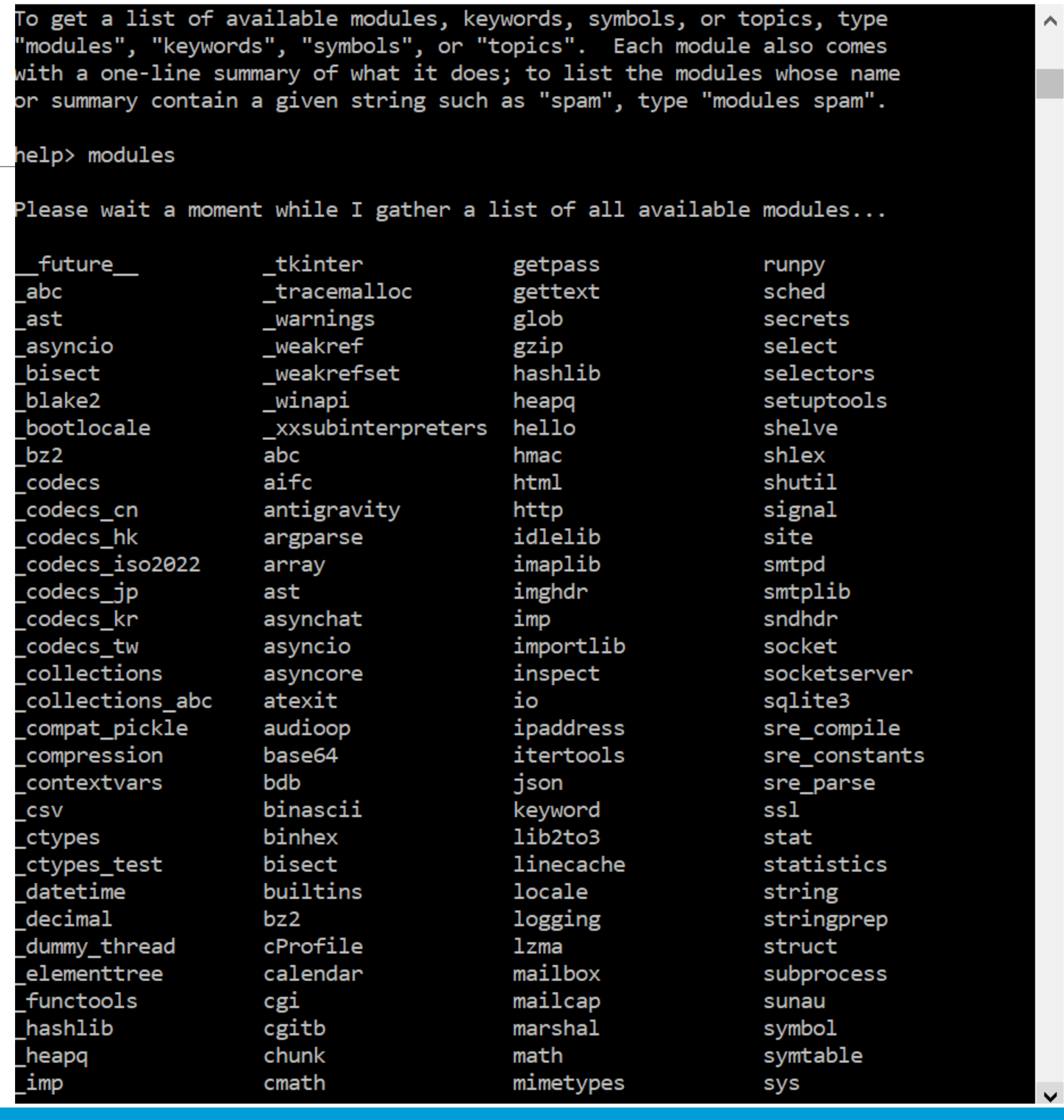
Python: Libreria Standard e Moduli

- Python contiene una **Libreria standard** che consente di generare programmi complessi
- Una **libreria** è una raccolta di codice che è stato scritto da altri programmatori ed è pronto per essere utilizzato
- Una libreria standard è una libreria che viene considerata parte del linguaggio e deve essere presenta in un qualsiasi ambiente di sviluppo Python
- La libreria standard di Python è organizzata in **moduli**: funzioni e tipi di dati correlati fanno parte di uno stesso modulo
- Le funzioni definite in un modulo devono essere caricate esplicitamente in un programma PRIMA che questo le possa utilizzare



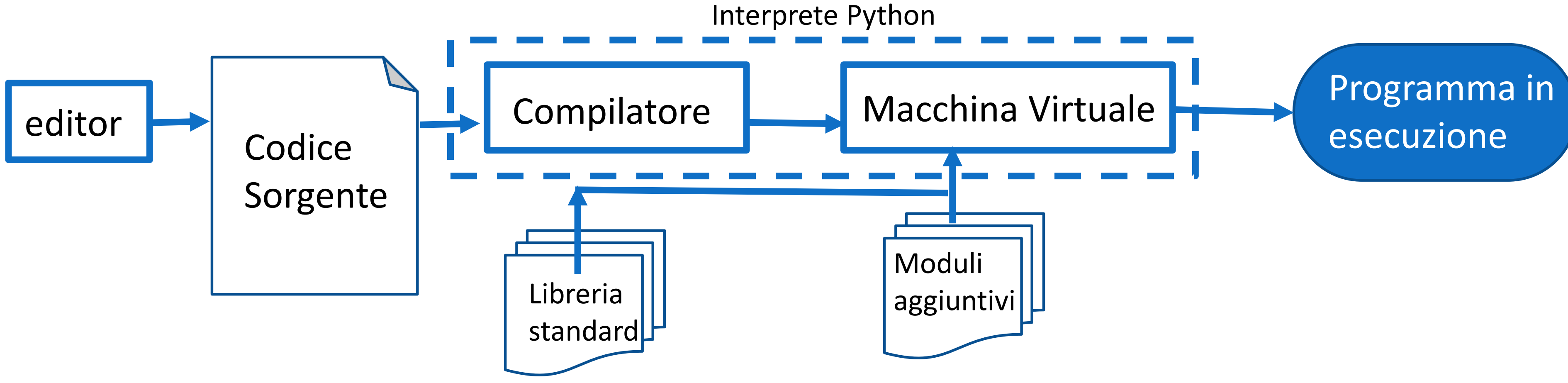
Quali sono i Moduli per Python 3.8?

- Riprendiamo la funzione help()
- Questa funzione ci permette di avere dettagli su: << … available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics“>> e ci dice anche quali comandi scrivere su prompt per avere tali dettagli
- Supponiamo di voler vedere la lista dei moduli, si digita allora “modules”



Modulo math: Funzioni matematiche (1)

- Python è provvisto di un **modulo** matematico che contiene le più comuni operazioni matematiche (math, uno dei moduli visti come disponibili nella schermata precedente)
- Un modulo è un file che contiene una raccolta di funzioni correlate
- Prima di poter usare le funzioni contenute in un modulo, dobbiamo dire all'interprete di caricare il modulo in memoria



Modulo math: Quali sono le funzioni disponibili? (1)

- Riprendiamo la funzione help()
- …
- Supponiamo di voler vedere la lista dei moduli, si digita allora “modules”

  Se vogliamo sapere la lista di funzioni disponibili per un certo modulo, si digita il nome del modulo stesso. Ad esempio ‘math’

Prompt dei comandi - python

```
help> math
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.
```

Modulo math: Quali sono le funzioni disponibili? (1)

- Supponiamo di voler vedere la lista dei moduli, si digita allora “modules”
- Se vogliamo sapere la lista di funzioni disponibili per un certo modulo, si digita il nome del modulo stesso. Ad esempio ‘math’
- In questo modo si visualizza la specifica di ogni funzione
- Ad esempio:

  sin(x, /)

  Return the sine of x (measured in radians).

Prompt dei comandi - python

```
    Raises ValueError if either of the arguments are negative.

pow(x, y, /)
    Return x**y (x to the power of y).

prod(iterable, /, *, start=1)
    Calculate the product of all the elements in the input iterable.

    The default start value for the product is 1.

    When the iterable is empty, return the start value.  This function is
    intended specifically for use with numeric values and may reject
    non-numeric types.

radians(x, /)
    Convert angle x from degrees to radians.

remainder(x, y, /)
    Difference between x and the closest integer multiple of y.

    Return x - n*y where n*y is the closest integer multiple of y.
    In the case where x is exactly halfway between two multiples of
    y, the nearest even value of n is used. The result is always exact.

sin(x, /)
    Return the sine of x (measured in radians).

sinh(x, /)
    Return the hyperbolic sine of x.

sqrt(x, /)
    Return the square root of x.

tan(x, /)
    Return the tangent of x (measured in radians).

tanh(x, /)
    Return the hyperbolic tangent of x.
```

Funzioni matematiche (2)

- Prima di poter usare le funzioni contenute in un modulo, dobbiamo dire all'interprete di caricare il modulo in memoria
- Questa operazione viene detta "importazione":

```
>>> import math
>>> print(math)
<module 'math' (built-in)>
```

- L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso
- Per chiamare una funzione inclusa in un modulo, dobbiamo specificare, nell'ordine:
  - **il nome del modulo che la contiene e il nome della funzione, separati da un punto**
  - **Questo formato è chiamato notazione a punto o dot notation**

Funzioni matematiche (3)

- Python è provvisto di un **modulo** matematico che contiene le più comuni operazioni matematiche
- Un modulo è un file che contiene una raccolta di funzioni correlate
- Prima di poter usare le funzioni contenute in un modulo, dobbiamo dire all'interprete di caricare il modulo in memoria
- Questa operazione viene detta "importazione":

```
>>> import math
>>> print(math)
<module 'math' (built-in)>
```

- L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso
- Per chiamare una funzione inclusa in un modulo, dobbiamo specificare, nell'ordine:
  - **il nome del modulo che la contiene e il nome della funzione, separati da un punto**
  - **Questo formato è chiamato notazione a punto o dot notation**

Funzioni matematiche (4)

- Per chiamare una funzione inclusa in un modulo, dobbiamo specificare, nell'ordine:
  - **il nome del modulo che la contiene e il nome della funzione, separati da un punto**
  - **Questo formato è chiamato notazione a punto o dot notation**
- Esempio:
- Log10 è una funzione definita nel modulo math:

math.log10(INPUT)

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> potenza_segnale = 1.58
>>> potenza_rumore = 0.000000423
>>> rapporto = potenza_segnale/potenza_rumore
>>> decibel = 10*math.log10(rapporto)
>>> print(decibel)
65.72316719579379
>>>
```

Funzioni matematiche (5)

- Esempio:
- sin è una funzione definita nel modulo math:

  math.sin(INPUT)

- calcola il seno della variabile radianti
- Il nome della variabile spiega già che sin e le altre funzioni trigonometriche (cos, tan, ecc.) accettano argomenti espressi in radianti

Seleziona Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> radianti=0.7
>>> altezza = math.sin(radianti)
>>> print(altezza)
0.644217687237691
>>>
```

Funzioni matematiche (6)

- Esempio:
- calcola il seno della variabile radianti
- Il nome della variabile spiega già che sin e le altre funzioni trigonometriche (cos, tan, ecc.) accettano argomenti espressi in radianti
- Per convertire da gradi in radianti occorre dividere per 360 e moltiplicare per 2π:

```
>>> gradi = 45
>>> radianti = gradi / 360.0 * 2 * math.pi
>>> math.sin(radianti)
0.707106781187
```

Prompt dei comandi - python

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64
 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> radianti=0.7
>>> altezza = math.sin(radianti)
>>> print(altezza)
0.644217687237691
>>> gradi = 45
>>> radianti = gradi/360.0*2*math.pi
>>> math.sin(radianti)
0.7071067811865476
>>>
```

NOTA:
L'espressione **math.pi** ricava la variabile pi dal modulo matematico. Il suo valore è un'approssimazione di π, accurata a circa 15 cifre

Errori di arrotondamento (1)

- Gli errori di arrotondamento sono inevitabili quando si effettuano calcoli con i numeri in virgola mobile
- Nell'hardware del processore, i numeri sono rappresentati con il sistema di numerazione binario
- Come avviene con i numeri decimali, si hanno errori di arrotondamento
- Visto l'esempio, la variabile total dovrebbe avere valore 435

  MA nel sistema binario NON esiste una rappresentazione esatta

  del numero 4.35 (100.0101100, con periodo…), così come nel

  sistema decimale non esiste la rappresentazione esatta per 1/3

Prompt dei coma...

```
>>> price = 4.35
>>> quantity = 100
>>> total = price * quantity
>>> print(total)
434.99999999999994
>>>
```

- La rappresentazione usata dal PC è di poco inferiore a 4.35, quindi moltiplicando per 100, viene la cifra che si vede in console
- Per gestire gli errori di arrotondamento è bene ad esempio, arrotondare alla cifra intera più vicina

Composizione (1)

- Finora, abbiamo considerato gli elementi di un programma - variabili, espressioni e istruzioni - separatamente, senza parlare di come utilizzarli insieme
- Una delle caratteristiche più utili dei linguaggi di programmazione è la loro capacità di prendere dei piccoli pezzi e comporli tra loro
- Per esempio, l'argomento di una funzione può essere un qualunque tipo di espressione, incluse operazioni aritmetiche

```
x = math.sin(gradi / 360.0 * 2 * math.pi)
```

- E anche chiamate di funzione:

```
x = math.exp(math.log(x+1))
```

- Potete mettere quasi ovunque un valore o un'espressione a piacere, con una eccezione**: il lato sinistro di una istruzione di assegnazione deve essere un nome di una variabile**
- Ogni altra espressione darebbe un errore di sintassi (vedremo più avanti le eccezioni a questa regola)

Composizione (2)

Prompt dei comandi - python

```
>>>
>>>
>>>
>>>
>>> ore = 22
>>> minuti = ore *60
>>> minuti
1320
>>> ore * 60 = minuti
  File "<stdin>", line 1
SyntaxError: cannot assign to operator
>>>
```

CORRETTO

ERRORE!!

Errori di parentesi

- Supponiamo di avere la seguente espressione:

  ((a+b) + x /2+ (b+c)

- Che cosa non torna?
- Contando le parentesi si nota che nella espressione, le parentesi aperte sono tre mentre le chiuse sono solo due!
- Python darà quindi errore di sintassi

Definire nuove funzioni

Sintassi:

```
def nomeDiFunzione(nomeParametro1, nomeParametro2, …, nomeParametroN):
    enunciati
```

Esempio:

```
def area(base, altezza):
    area=base*altezza
    return area
```

header o intestazione della funzione

Corpo della funzione

Parametri in ingresso alla funzione (input)
Sono i valori che devo passare alla funzione al momento della chiamata

main.py saved

```
1  def area(base, altezza):
2      area=base*altezza
3      return area
4
5  #flusso principale del programma
6  b = 2
7  a = 4
8  A = area(b, a)#chiamata della funzione
9  print("Area: %d" % A)
```

```
Area: 8
```

Enunciato **return**: termina la funzione e restituisce il risultato, ovvero l'output (non è obbligatoria la sua presenza, dipende da che funzione si crea)

Definire nuove funzioni (1)

- Finora abbiamo soltanto usato funzioni che sono parte integrante di Python, ma è anche possibile crearne di nuove
- Una definizione di funzione specifica il **nome** di una nuova funzione e la **sequenza di istruzioni** che viene eseguita quando la funzione viene chiamata
- Ecco un esempio:

NOTA: iniziamo a definire funzioni che NON presentano al loro interno l'enunciato **return**

```
def stampa_brani():
    print ("Terror di tutta la foresta egli è")
    print ("Con l'ascia in mano si sente un re.")
```

- La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite:

```
stampa_brani()
```

Collaudo di una funzione (1)

- Si è visto come definire una funzione, MA se si esegue un programma che contiene SOLO una definizione, non succede nulla!
- Questo perché la funzione, per produrre i risultati voluti, deve essere invocata (o chiamata)
- Per poter collaudare una funzione, il programma deve contenere:
- La definizione della funzione
- Una serie di enunciati che invocano la funzione e che permettano di visualizzarne il valore restituito o il risultato voluto (ad esempio la scrittura sul console)
- Ecco l'esempio di un programma completo:

main.py saved

```python
def stampa_brani():
    print ("Terror di tutta la foresta egli è")
    print ("Con l'ascia in mano si sente un re.")

stampa_brani()
```

```
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
>
```

Collaudo di una funzione (2)

main.py saved

```python
def stampa_brani():
    print ("Terror di tutta la foresta egli è")
    print ("Con l'ascia in mano si sente un re.")

stampa_brani()
```

```
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
>
```

- Quando si scrive un programma, è necessario fare attenzione anche all'ordine delle definizioni delle funzioni e delle istruzioni presenti nel programma
- Mentre l'interprete Python analizza il codice sorgente, legge tutte le definizioni di funzioni (header) e tutti gli enunciati (istruzioni, etc.) presenti nel flusso principale del programma
- Gli enunciati presenti nella definizione delle funzioni NON vengono eseguiti finché la funzione non viene CHIAMATA
- Gli enunciati che NON sono nel corpo delle definizioni delle funzioni (sono quindi nel flusso principale), vengono eseguiti nell'ordine in cui figurano nel codice
- Quindi è importante che ciascuna funzione venga definita PRIMA di essere invocata!!

Collaudo di una funzione (3)

- E' importante che ciascuna funzione venga definita PRIMA di essere invocata!!

main.py   saving...

```
1   stampa_brani()
2   def stampa_brani():
3       print ("Terror di tutta la foresta egli è")
4       print ("Con l'ascia in mano si sente un re.")
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    stampa_brani()
NameError: name 'stampa_brani' is not defined
```

- Se si inverte l'ordine e si chiama una funzione PRIMA di averla definita, allora Python darà errore!

- Questo perché il compilatore legge dalla prima riga in poi, e appena legge NON sa che la funzione stampa_brani() verrà definita più avanti nel codice

- Per convenzione, esiste una funzione (main()) che individua il flusso principale del programma, ovvero il punto di partenza del programma:

def main():

    ……

Ovviamente tale funzione principale dovrà poi essere chiamata a sua volta

Collaudo di una funzione (4)

main.py

```
1  def main():
2      stampa_brani()
3  def stampa_brani():
4      print ("Terror di tutta la foresta egli è")
5      print ("Con l'ascia in mano si sente un re.")
6
7  main()
```

definizione della funzione principale main()

Chiamata della funzione principale main() e INIZIO della esecuzione del programma

```
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
```

main.py

```
1  def main():
2      stampa_brani()
3  def stampa_brani():
4      print ("Terror di tutta la foresta egli è")
5      print ("Con l'ascia in mano si sente un re.")
```

Questo programma contiene SOLO definizioni Quindi non produce risultati: nessuna funzione viene chiamata!

Definire nuove funzioni (2)

- Ecco un esempio:

```
def stampa_brani():
    print ("Terror di tutta la
            foresta egli è")
    print ("Con l'ascia in mano
            si sente un re.")
```

- La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite:

```
stampa_brani()
```

Prompt dei comandi - python

```
>>> def stampa_brani():
...     print("Terror di tutta la foresta egli è")
...     print("Con l'ascia in mano si sente un re.")
...
>>> stampa_brani()
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
>>>
```

Definire nuove funzioni (3)

- Una volta definita una funzione, si può utilizzarla all'interno di un'altra funzione
- Per esempio, per ripetere due volte il brano precedente possiamo scrivere una funzione ripeti_brani:

```
def ripeti_brani():
    stampa_brani()
    stampa_brani()
```

- E quindi se si richiama la funzione ripeti_brani, la frase verrà stampata due volte:

Terror di tutta la foresta egli è

Con l'ascia in mano si sente un re.

Terror di tutta la foresta egli è

Con l'ascia in mano si sente un re.

Definire nuove funzioni (4)

Prompt dei comandi - python

```
>>> def stampa_brani():
...     print("Terror di tutta la foresta egli è")
...     print("Con l'ascia in mano si sente un re.")
...
>>> stampa_brani()
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
>>> def ripeti_brani():
...     stampa_brani()
...     stampa_brani()
...
>>> ripeti_brani()
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
>>>
```

Definire nuove funzioni (5)

main.py saved

```
1   def stampa_brani():
2       print ("Terror di tutta la foresta egli è")
3       print ("Con l'ascia in mano si sente un re.")
4
5   def ripeti_brani():
6       stampa_brani()
7       stampa_brani()
8       stampa_brani()
9
10  def main():
11      stampa_brani()
12      ripeti_brani()
13
14  if __name__ == '__main__':
15      main()
```

```
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è
Con l'ascia in mano si sente un re.
```

- L'interprete di **Python**, prima di eseguire il codice di qualsiasi file con estensione .py, assegna delle variabili "speciali", tra queste si trova __name__
- Nel caso in cui il modulo viene eseguito come programma principale(come main), l'interprete assegnerà alla variabile __name__ la stringa "__main__"; altrimenti gli verrà assegnato come valore il nome del file .py.

Definire nuove funzioni (6)

- Come collegare definizioni scritte in più file con il programma principale?
- Supponiamo di aver scritto alcune definizioni per stampare una serie di stringhe in un file che chiamiamo stampa_base.py e di voler utilizzare queste funzioni dal file principale….
- ?????

Definire nuove funzioni (6)

- Come collegare definizioni scritte in più file con il programma principale?
- Supponiamo di aver scritto alcune definizioni per stampare una serie di stringhe in un file che chiamiamo base.py e di voler utilizzare queste funzioni dal file principale….

base.py

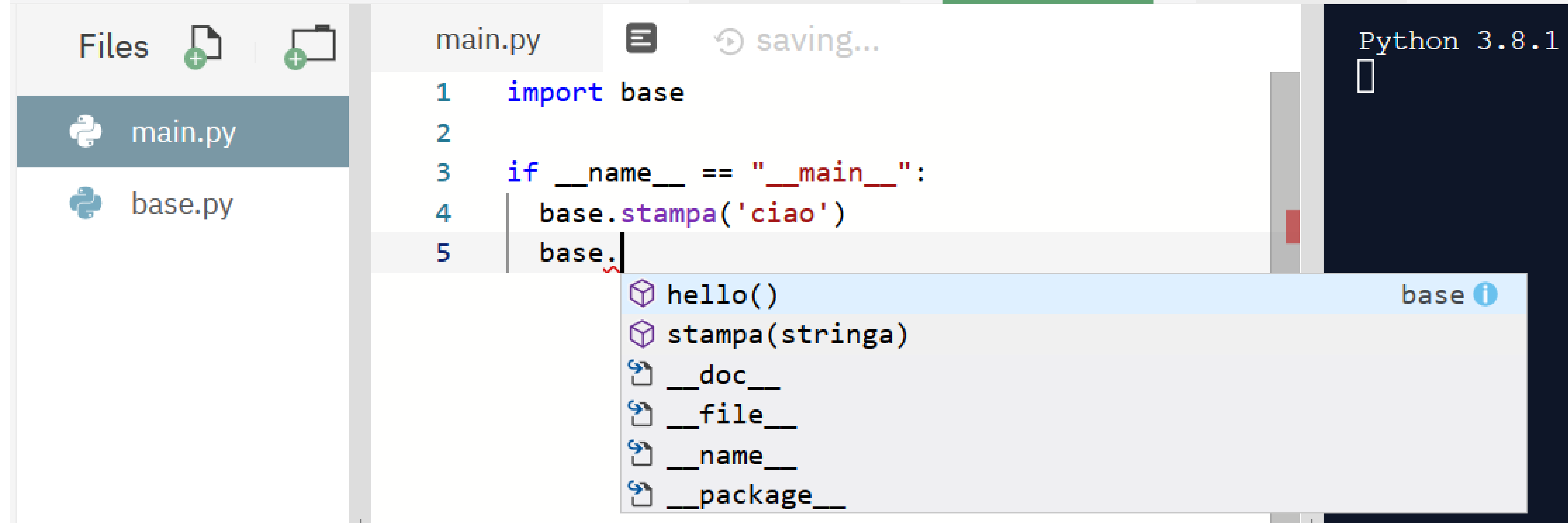
```
def stampa(stringa):
    print("Stampo: ", stringa)

def hello():
    print("hello World!")
```

main.py

```
import base

if __name__ == "__main__":
    base.stampa('ciao’)
    base.hello()
```

Definire nuove funzioni (6)

- Come collegare definizioni scritte in più file con il programma principale?
- Supponiamo di aver scritto alcune definizioni per stampare una serie di stringhe in un file che chiamiamo stampa_base.py e di voler utilizzare queste funzioni dal file principale….



Definizioni e loro utilizzo

- Raggruppando l linee di codice scritte, ovvero le due definizioni:

```
def stampa_brani():
        print ("Terror di tutta la foresta egli è")
        print ("Con l'ascia in mano si sente un re.")

def ripeti_brani():
    stampa_brani()
    stampa_brani()
    stampa_brani()

def main():
    stampa_brani()
    ripeti_brani()
```

- Si può notare che:
  - Le definizioni di funzione sono eseguite come le altre istruzioni, ma il loro effetto è solo quello di creare una nuova funzione
  - Le istruzioni all'interno di una definizione non vengono eseguite fino a quando la funzione non viene chiamata, e la definizione di per sé non genera alcun risultato
- INOLTRE: una funzione deve essere definita prima di poterla usare:

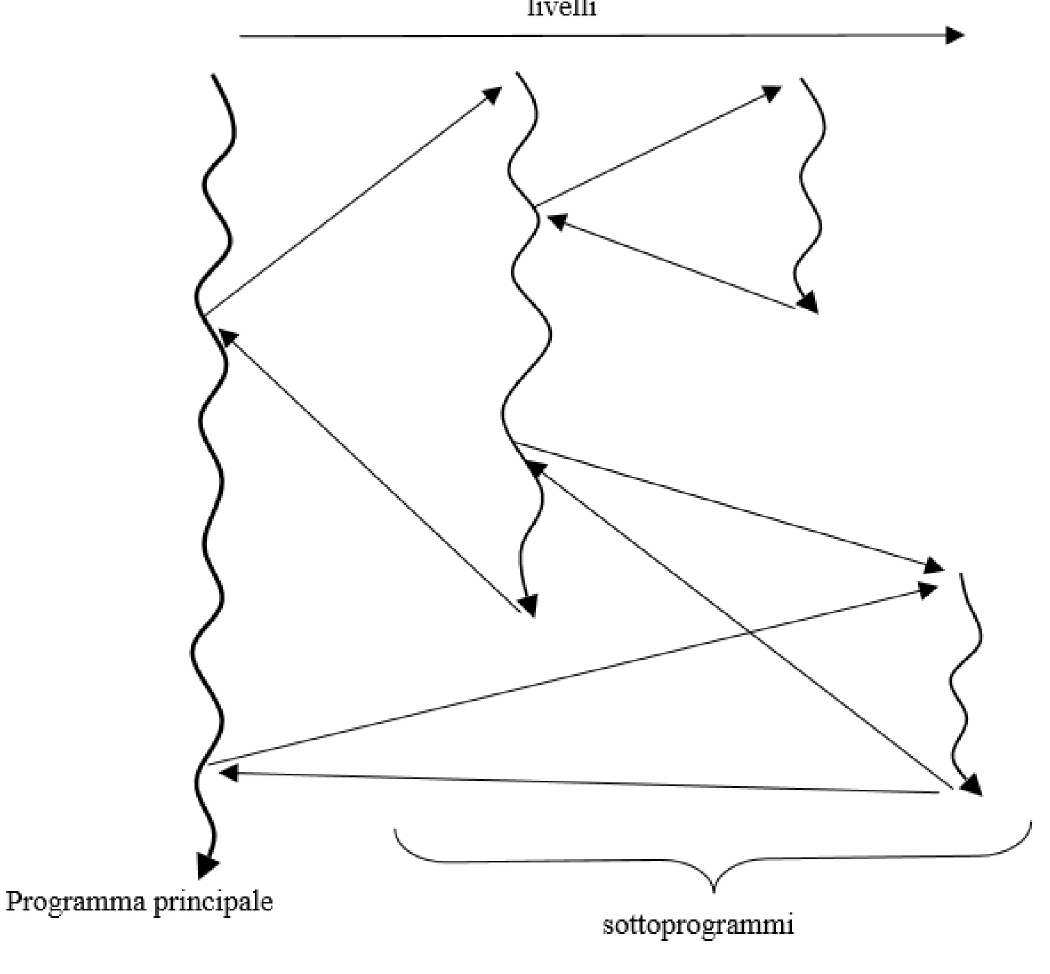
la definizione della funzione deve sempre precedere la sua chiamata

Flusso di esecuzione (1)

- Per assicurarvi che una funzione sia definita prima del suo uso, dovete conoscere l'ordine in cui le istruzioni vengono eseguite, cioè il flusso di esecuzione del programma
- L'esecuzione inizia sempre dalla prima riga del programma e le istruzioni sono eseguite una alla volta dall'alto verso il basso
- Le definizioni di funzione non alterano il flusso di esecuzione del programma ma va ricordato che le istruzioni all'interno delle funzioni non vengono eseguite fino a quando la funzione non viene chiamata
- Una chiamata di funzione è una sorta di deviazione nel flusso di esecuzione:
  - Invece di proseguire con l'istruzione successiva, il flusso salta alla prima riga della funzione chiamata ed esegue tutte le sue istruzioni
  - alla fine della funzione il flusso riprende dal punto dov'era stato deviato
- Sinora è tutto abbastanza semplice, ma dovete tenere conto che una funzione può chiamarne un'altra al suo interno. Nel bel mezzo di una funzione, il programma può dover eseguire le istruzioni situate in un'altra funzione. Ma mentre esegue la nuova funzione, il programma può doverne eseguire un'altra ancora!

Flusso di esecuzione (2)

- Fortunatamente, Python sa tener bene traccia di dove si trova, e ogni volta che una funzione viene completata il programma ritorna al punto che aveva lasciato
- Giunto all'ultima istruzione, dopo averla eseguita, il programma termina
- QUINDI:
- Quando leggete un programma non limitatevi sempre a farlo dall'alto in basso. Spesso ha più senso cercare di seguire il flusso di esecuzione



Il concetto di sottoprogramma (1)

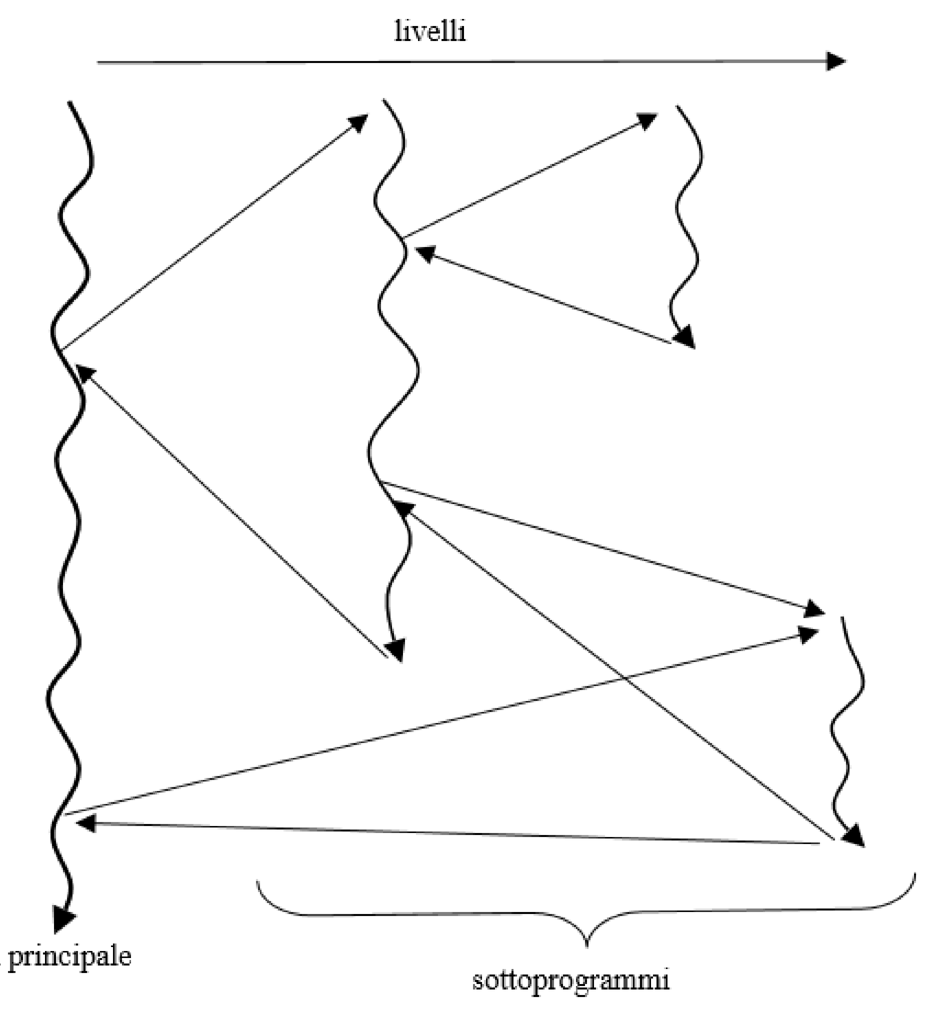
- Comunemente la realizzazione di un algoritmo prevede l'uso di algoritmi più semplici
- Gli algoritmi più semplici spesso costituiscono porzioni che possono essere riutilizzate più volte nello stesso o in altri contesti
- Questo meccanismo può essere usato per suddividere l'algoritmo iniziale in un insieme di algoritmi più semplici che, una volta risolti, possono portare alla soluzione del problema completo iniziale
- E' quindi necessario un metodo per la definizione e l'aggregazione delle istruzioni: blocchi o sequenze, procedure e funzioni
- Questo meccanismo si realizza con la formalizzazione di sottoprogrammi

Il concetto di sottoprogramma (2)

- La forma elementare per la suddivisione/raggruppamento di istruzioni è il costrutto sequenza per la realizzazione di un blocco di istruzioni
- Il blocco delimita un gruppo di istruzioni, ma non ha un identificativo che lo contraddistingue
- In Python i delimitatori di blocco di istruzioni sono gli spazi (concetto di compound)
- A livello intuitivo quindi un programma è un blocco/sequenza di istruzioni identificato da un nome, che viene evocato al momento della chiamata
- A tal punto il programma chiamante trasferisce al sottoprogramma i dati necessari all'elaborazione secondo una convenzione posizionale

Il concetto di sottoprogramma (3)

- I sottoprogrammi si possono classificare in procedure e funzioni
- Sono particolari strutture di controllo che, in seguito alla loro invocazione, **alterano il flusso del programma**
- Il programma principale può usare un sottoprogramma per effettuare determinate operazioni
- Il sottoprogramma a sua volta può richiamare altri sottoprogrammi
- I sottoprogrammi possono essere usati in più punti di un programma
- E' importante che i singoli sottoprogrammi lavorino su variabili proprie o su copie, comunque solo in modo controllato



Il dominio di validità delle variabili, lo scope (1)

- Ogni sottoprogramma può avere delle variabili e queste variabili hanno validità SOLO nel blocco in cui sono definite e nei blocchi sottostanti SE in questi non sono definite variabili omonime
- Il campo di validità di una variabile è detto scope
- Le variabili sono dichiarate in un certo contesto che è determinato dal costrutto sequenza (ovvero dalla coppia di parentesi che troviamo sia nei sottoprogrammi che nei blocchi annidati di istruzioni…if, while, for, etc.)
- La variabile a è visibile per tutto il campo di validità (**variabile globale**)
- Le altre variabili hanno uno scope limitato (**variabili locali**)

Il dominio di validità delle variabili, lo scope (1)

- Esempio:

```
a = 19 #variabile GLOBALE è definita FUORI dalle funzioni

def stampa_intero():
    b =3+a #sempre valida perché è definita FUORI dalle funzioni
    print(b) #b è variabile locale
def main():
    c = 10
    print(a) #sempre valida
    stampa_intero()  #print(b) ERRORE!! b NON ha validità qui!! b ha
                  #validità solo nella funzione stampa_intero()
    print(c)
```

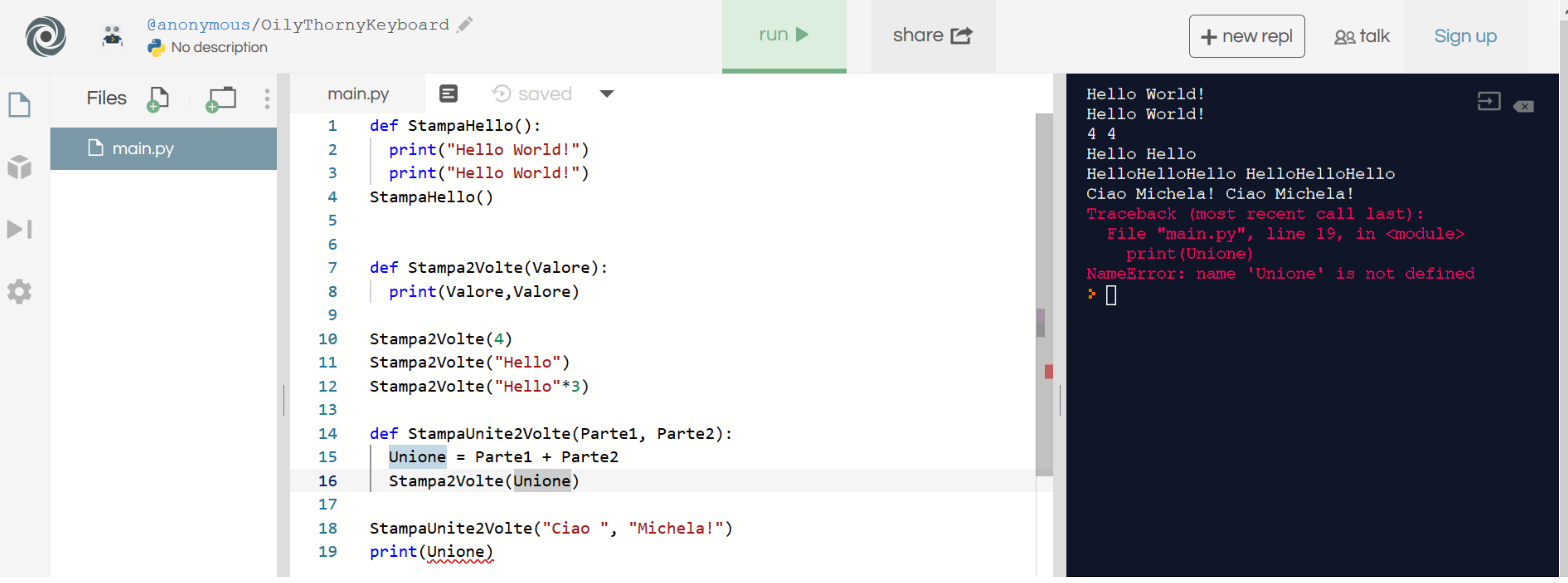
Il dominio di validità delle variabili, lo scope (2)

```
a = 19 #a variabile GLOBALE

def stampa_intero():
    b =3+a #b locale
    print(b)

def main():
  c = 10 # c locale
  print(a) #sempre valida
  stampa_intero()
  print(c) #sempre valida
```

a b c

- b è una variabile GLOBALE quindi ha validità per tutto il programma:
  - Può essere utilizzata dalle funzioni
- b è una variabile LOCALE della funzione stampa_intero():
  - Se si prova ad usare b ad esmpio dentro il main(), Python darà ERRORE!!
- c è una variabile LOCALE del main()

https://repl.it/languages/python3



Parametri e argomenti (1)

- Alcune delle funzioni che abbiamo visto richiedono degli argomenti (o parametri in ingresso, parametri in INPUT)
- Per esempio, se si vuole trovare il seno di un numero chiamando la funzione math.sin, si deve passare quel numero come argomento (INPUT)
- Alcune funzioni ricevono più di un argomento:
  - a math.pow ne servono due, ovvero la base e l'esponente dell'operazione di elevamento a potenza
  - All'interno della funzione, gli argomenti che le vengono passati sono assegnati ad altrettante variabili chiamate parametri
- Ecco un esempio di definizione di una funzione che riceve un argomento:

```
def stampa2volte(bruce):
    print(bruce)
    print(bruce)
```

Parametri e argomenti (2)

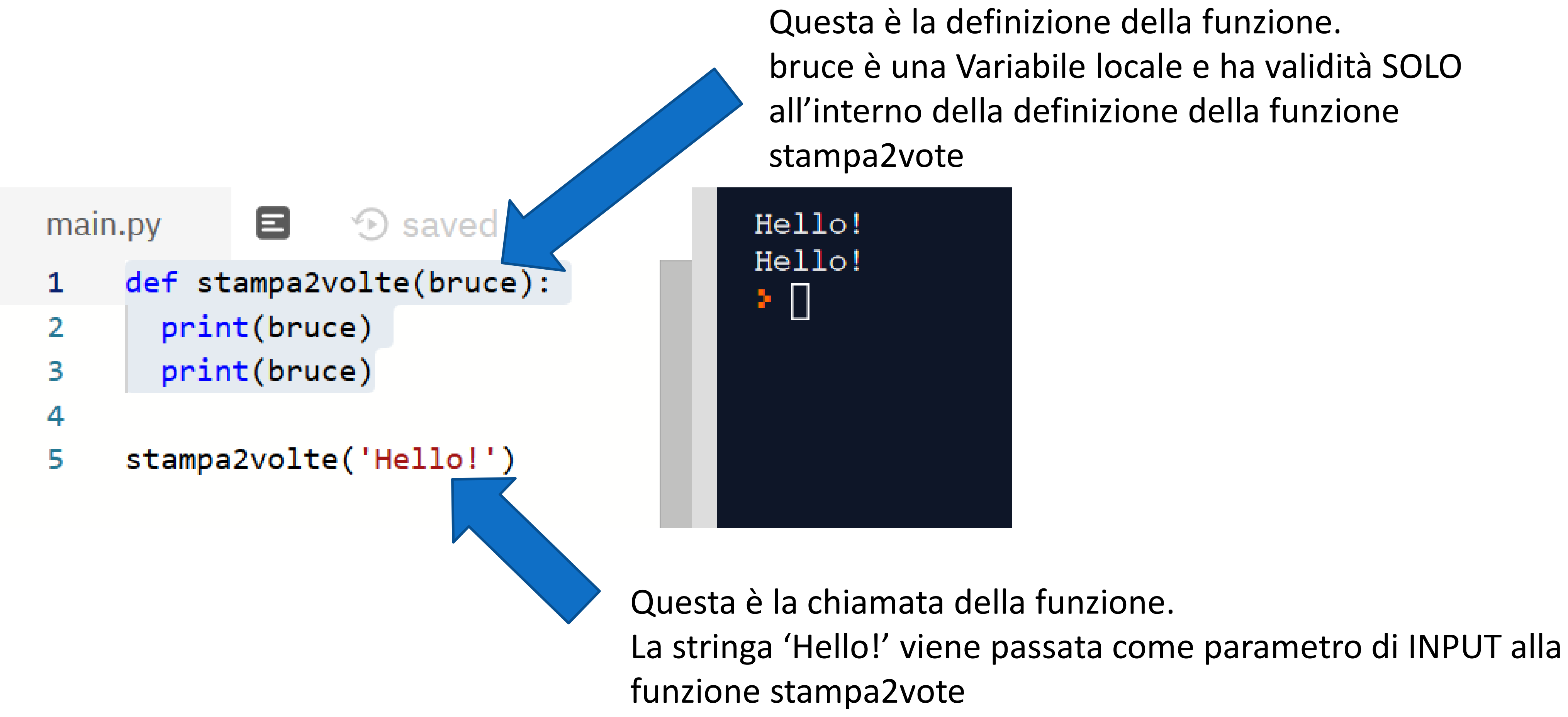
- Ecco un esempio di definizione di una funzione che riceve un argomento:

```
def stampa2volte(bruce):
    print(bruce)
    print(bruce)
```

- Questa funzione assegna l'argomento ricevuto ad un parametro chiamato bruce
- Quando la funzione viene chiamata, stampa il valore del parametro (qualunque esso sia) due volte
- Questa funzione lavora con qualunque valore che possa essere stampato.

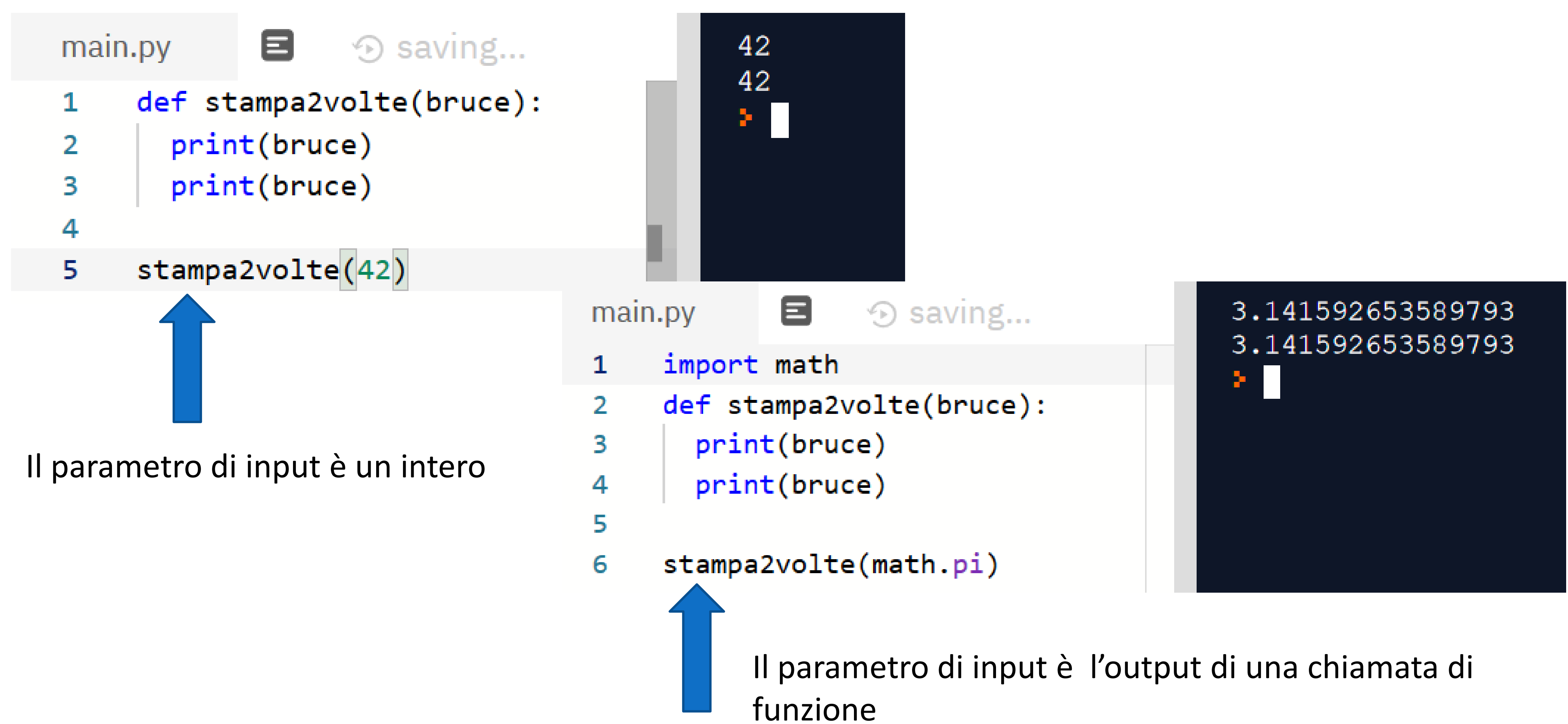
Parametri e argomenti (3)

Questa è la definizione della funzione.
bruce è una Variabile locale e ha validità SOLO all'interno della definizione della funzione stampa2vote



Questa è la chiamata della funzione.
La stringa 'Hello!' viene passata come parametro di INPUT alla funzione stampa2vote

Parametri e argomenti (4)



Parametri e argomenti (4)

- Le stesse regole di composizione che valgono per le funzioni predefinite si applicano anche alle funzioni definite dall'utente, pertanto possiamo usare come argomento per stampa2volte qualsiasi espressione:

main.py

```
1  def stampa2volte(bruce):
2      print(bruce)
3      print(bruce)
4
5  stampa2volte('Spam '*4)
```

```
Spam Spam Spam Spam
Spam Spam Spam Spam
```

Il parametro di input è una espressione

- L'argomento viene valutato prima della chiamata alla funzione
- Le espressioni 'Spam '*4 e math.cos(math.pi) vengono valutate **una volta sola**

main.py

```
1  import math
2  def stampa2volte(bruce):
3      print(bruce)
4      print(bruce)
5
6  stampa2volte(math.cos(math.pi))
```

```
-1.0
-1.0
```

Parametri e argomenti (5)

- Si puo’ anche usare una variabile come parametro di una funzione

```
def stampa2volte(bruce):
    print(bruce)
    print(bruce)

michael = 'My name is Michael’
stampa2volte(michael)
```

main.py   saving...

```
1  def stampa2volte(bruce):
2    print(bruce)
3    print(bruce)
4
5  michael = 'My name is Michael'
6  stampa2volte(michael)
```

```
My name is Michael
My name is Michael
›
```

- Il nome della variabile che passiamo come argomento (michael) non ha niente a che fare con il nome del parametro nella definizione della funzione (bruce)

Variabili e parametri sono locali (1)

- Quando si usa una variabile nella definizione di una funzione, essa è locale, cioè esiste solo all'interno della funzione (ambito di validità / SCOPE)
- Per esempio:

```
def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)
```

- Questa funzione prende due argomenti, li concatena e poi ne stampa il risultato due volte richiamando la funzione `stampa2volte`
- Nel momento in cui questa funzione verrà chiamata, sarà quindi necessario passare due parametri in ingresso

Variabili e parametri sono locali (2)

main.py

```python
def stampa2volte(bruce):
    print(bruce)
    print(bruce)

def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)

michael = 'My name is Michael'
stampa2volte(michael)
cat2volte('Hello, ','Michael!')
print(cat)
```

```
My name is Michael
My name is Michael
Hello, Michael!
Hello, Michael!
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(cat)
NameError: name 'cat' is not defined
```

- Quando cat2volte termina, la variabile cat viene 'distrutta'
- Se si prova a stamparla, infatti si ottiene un messaggio d'errore
- NOTA: stessa cosa vale per la variabile bruce

Diagrammi di stack (1)

- Per tenere traccia di quali variabili possono essere usate e dove, è talvolta utile disegnare un diagramma di stack
- Come i diagrammi di stato, i diagrammi di stack mostrano il valore di ciascuna variabile, ma in più indicano a quale funzione, tale variabile appartiene
- Ogni funzione è rappresentata da un frame, un riquadro con il nome della funzione a fianco e la lista dei suoi parametri e delle sue variabili all'interno
- Il diagramma di stack nel caso dell'esempio precedente, è illustrato in Figura
- I frame sono disposti in una pila che indica quale funzione ne ha chiamata un'altra e così via. Nell'esempio, stampa2volte è stata chiamata da cat2volte, e cat2volte è stata a sua volta chiamata da __main__, che è un nome speciale per il frame principale. Quando si crea una variabile che è esterna ad ogni funzione, essa appartiene a __main__.

```
1   def stampa2volte(bruce):
2       print(bruce)
3       print(bruce)
4
5   def cat2volte(parte1, parte2):
6       cat = parte1 + parte2
7       stampa2volte(cat)
8
9   riga1 = 'Hello, '
10  riga2 = 'Michael!'
11  cat2volte(riga1,riga2)
```

<modulo>	riga1 → 'Hello, ' riga2 → 'Micheal!'
cat2volte	parte1 → 'Hello, ' parte2 → 'Micheal!' cat → 'Hello, Micheal!'
stampa2volte	bruce → 'Hello, Micheal! '

Diagrammi di stack (2)

- Per tenere traccia di quali variabili possono essere usate e dove, è talvolta utile disegnare un diagramma di stack
- Come i diagrammi di stato, i diagrammi di stack mostrano il valore di ciascuna variabile, ma in più indicano a quale funzione, tale variabile appartiene
- Ogni funzione è rappresentata da un frame, un riquadro con il nome della funzione a fianco e la lista dei suoi parametri e delle sue variabili all'interno
- Il diagramma di stack nel caso dell'esempio precedente, è illustrato in Figura

```
1   def stampa2volte(bruce):
2       print(bruce)
3       print(bruce)
4
5   def cat2volte(parte1, parte2):
6       cat = parte1 + parte2
7       stampa2volte(cat)
8
9   riga1 = 'Hello, '
10  riga2 = 'Michael!'
11  cat2volte(riga1,riga2)
```

<modulo>
riga1 → 'Hello, '
riga2 → 'Micheal!'

cat2volte
parte1 → 'Hello, '
parte2 → 'Micheal!'
cat → 'Hello, Micheal!'

stampa2volte
bruce → 'Hello, Micheal! '

Diagrammi di stack (3)

- I frame sono disposti in una pila che indica quale funzione ne ha chiamata un’altra e così via
- Nell’esempio, stampa2volte è stata chiamata da cat2volte, e cat2volte è stata a sua volta chiamata da __main__, che è un nome speciale per il frame principale
- Quando si crea una variabile che è esterna ad ogni funzione, essa appartiene a __main__

```
1   def stampa2volte(bruce):
2       print(bruce)
3       print(bruce)
4
5   def cat2volte(parte1, parte2):
6       cat = parte1 + parte2
7       stampa2volte(cat)
8
9   riga1 = 'Hello, '
10  riga2 = 'Michael!'
11  cat2volte(riga1,riga2)
```

Frame	Variabile	Valore
<modulo>	riga1 →	‘Hello, ’
	riga2 →	‘Micheal!’
cat2volte	parte1 →	‘Hello, ’
	parte2 →	‘Micheal!’
	cat →	‘Hello, Micheal!’
stampa2volte	bruce →	‘Hello, Micheal! ’

Diagrammi di stack (4)

- Quando si crea una variabile che è esterna ad ogni funzione, essa appartiene a __main__
- I due programmi sottostanti sono quindi equivalenti

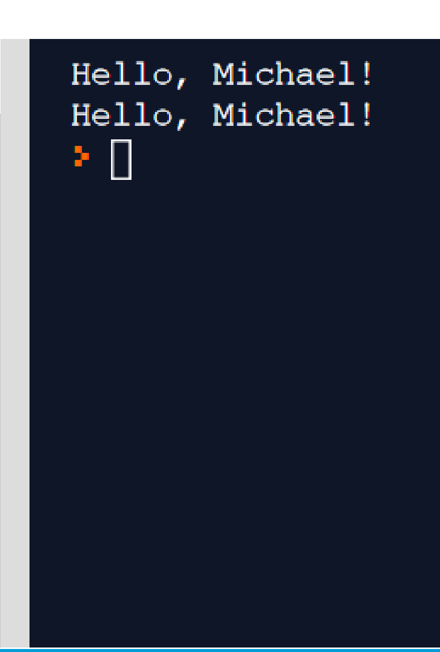
main.py

```python
def stampa2volte(bruce):
    print(bruce)
    print(bruce)

def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)

def main():
    riga1 = 'Hello, '
    riga2 = 'Michael!'
    cat2volte(riga1,riga2)

if __name__ == '__main__':
    main()

```

```
Hello, Michael!
Hello, Michael!
```

main.py

```python
def stampa2volte(bruce):
    print(bruce)
    print(bruce)

def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)

riga1 = 'Hello, '
riga2 = 'Michael!'
cat2volte(riga1,riga2)
```



Diagrammi di stack (5)

- Ogni parametro fa riferimento allo stesso valore del suo argomento corrispondente
  - parte1 ha lo stesso valore di riga1
  - parte2 ha lo stesso valore di riga2
  - bruce ha lo stesso valore di cat
- Se si verifica un errore durante la chiamata di una funzione, Python mostra:
  - il nome della funzione
  - il nome della funzione che l'ha chiamata
  - il nome della funzione che a sua volta ha chiamato quest'ultima e così via, fino a raggiungere il primo livello che è sempre __main__

```
1   def stampa2volte(bruce):
2       print(bruce)
3       print(bruce)
4
5   def cat2volte(parte1, parte2):
6       cat = parte1 + parte2
7       stampa2volte(cat)
8
9   riga1 = 'Hello, '
10  riga2 = 'Michael!'
11  cat2volte(riga1,riga2)
```

<modulo>	riga1 →	'Hello, '
	riga2 →	'Micheal!'
cat2volte	parte1 →	'Hello, '
	parte2 →	'Micheal!'
	cat →	'Hello, Micheal!'
stampa2volte	bruce →	'Hello, Micheal! '

Diagrammi di stack (6)

- Se si verifica un errore durante la chiamata di una funzione, Python mostra:
  - il nome della funzione
  - il nome della funzione che l'ha chiamata
  - il nome della funzione che a sua volta ha chiamato quest'ultima e così via, fino a raggiungere il primo livello che è sempre __main__

<modulo>	riga1 →	'Hello, '
	riga2 →	'Micheal!'
cat2volte	parte1 →	'Hello, '
	parte2 →	'Micheal!'
	cat →	'Hello, Micheal!'
stampa2volte	bruce →	'Hello, Micheal! '

- Se si cerca di accedere a cat dall'interno di stampa2volte, si ottiene un errore di tipo NameError:

main.py   saving...

```
1   def stampa2volte(bruce):
2     print(bruce)
3     print(bruce)
4     print(cat)
5
6   def cat2volte(parte1, parte2):
7     cat = parte1 + parte2
8     stampa2volte(cat)
9
10  riga1 = 'Hello, '
11  riga2 = 'Michael!'
12  cat2volte(riga1,riga2)
```

```
Hello, Michael!
Hello, Michael!
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    cat2volte(riga1,riga2)
  File "main.py", line 8, in cat2volte
    stampa2volte(cat)
  File "main.py", line 4, in stampa2volte
    print(cat)
NameError: name 'cat' is not defined
```

Diagrammi di stack (7)

- Questo elenco di funzioni è detto traceback
- Il traceback vi dice in quale file è avvenuto l’errore, e in quale riga, e quale funzione era in esecuzione in quel momento
- Mostra anche la riga di codice che ha causato l’errore. L’ordine delle funzioni nel traceback è lo stesso di quello dei frame nel diagramma di stack
- La funzione attualmente in esecuzione si trova in fondo all’elenco.

<modulo>	riga1 →	‘Hello, ’
	riga2 →	‘Micheal!’
cat2volte	parte1 →	‘Hello, ’
	parte2 →	‘Micheal!’
	cat →	‘Hello, Micheal!’
stampa2volte	bruce →	‘Hello, Micheal! ’

main.py saving...

```
1   def stampa2volte(bruce):
2       print(bruce)
3       print(bruce)
4       print(cat)
5
6   def cat2volte(parte1, parte2):
7       cat = parte1 + parte2
8       stampa2volte(cat)
9
10  riga1 = 'Hello, '
11  riga2 = 'Michael!'
12  cat2volte(riga1,riga2)
```

```
Hello, Michael!
Hello, Michael!
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    cat2volte(riga1,riga2)
  File "main.py", line 8, in cat2volte
    stampa2volte(cat)
  File "main.py", line 4, in stampa2volte
    print(cat)
NameError: name 'cat' is not defined
```

Funzioni produttive e funzioni vuote (1)

- Alcune delle funzioni che abbiamo usato, tipo le funzioni matematiche, restituiscono dei risultati
- in mancanza di definizioni migliori, personalmente le chiamo funzioni produttive
- Altre funzioni, come stampa2volte, eseguono un'azione ma non restituiscono alcun valore
- Le chiameremo funzioni vuote
- Quando si chiama una funzione produttiva, quasi sempre è per fare qualcosa di utile con il suo risultato, tipo assegnarlo a una variabile o usarlo come parte di un'espressione

main.py   saving...

```python
import math

radianti = math.pi
x = math.cos(radianti)
print(x)
aureo = (math.sqrt(5) + 1) / 2
print(aureo)
```

```
-1.0
1.618033988749895
```

Funzioni produttive e funzioni vuote (2)

- Se si chiama una funzione in modalità interattiva (da riga di comando), Python ne mostra il risultato:

Prompt dei comandi - python

```
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> math.sqrt(5)
2.23606797749979
>>>
```

- Ma, se si chiama una funzione produttiva da riga di comando così come è, il valore di ritorno è perso! Non si è memorizzato il valore in nessuna variabile
- Questo script in effetti calcola la radice quadrata di 5, ma non conserva né visualizza il risultato, per cui non è di grande utilità
- Le funzioni vuote possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non restituiscono un valore

Importare con from (1)

- Python fornisce due metodi per importare i moduli. Ne abbiamo già incontrato uno:

```
import math
```

- Quando si importa math, si ottiene un oggetto modulo di nome math che contiene delle costanti come pi (il pi-greco) e funzioni come sin e exp
- Ma se si tenta di accedere a pi direttamente, si ottiene un errore
- Un'alternativa è di importare un oggetto da un modulo in questa maniera:

```
>>> from math import pi
```

- Dopo questa riga di codice, è possibile accedere a pi direttamente, senza usare la notazione a punto (math.pi)

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019,
19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license"
 for more information.
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> print(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> from math import pi
>>> print(pi)
3.141592653589793
>>>
```

Importare con from (2)

- Un'alternativa ulteriore è quella di importare *tutto* da un modulo in questa maniera:

>>> from math import *

- Dopo questa riga di codice, è possibile accedere a pi direttamente, senza usare la notazione a punto (math.pi)
- Il vantaggio di importare tutto dal modulo matematico è che il codice diventa più conciso
- Per contro, potrebbero insorgere delle omonimie tra nomi definiti in moduli diversi, o tra un nome del modulo importato e una delle variabili definite nel programma da cui si effettua l'import

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019,
19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license"
 for more information.
>>> import math
>>> print(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> from math import *
>>> print(pi)
3.141592653589793
```

Funzioni produttive e funzioni vuote (3)

- Se si prova ad assegnare il risultato ad una variabile, si ottiene un valore speciale chiamato None (nulla)

main.py

```
1  def stampa2volte(bruce):
2    print(bruce)
3    print(bruce)
4
5  risultato = stampa2volte('Bing')
6  print(risultato)
```

```
Bing
Bing
None
```

- Il valore None non è la stessa cosa della stringa 'None’

- È un valore speciale che appartiene ad un tipo tutto suo:

  - Le funzioni che abbiamo **scritto** finora, sono tutte vuote
  - Cominceremo a scriverne di produttive a breve

main.py

```
1  def stampa2volte(bruce):
2    print(bruce)
3    print(bruce)
4
5  risultato = stampa2volte('Bing')
6  print(risultato)
7  print(type(risultato))
```

```
Bing
Bing
None
<class 'NoneType'>
```

Esempio di Funzione Produttiva (1)

- Le funzioni Produttive restituiscono un valore (producono un valore in output), per poterlo fare dal punto di vista sintattico, presentano al loro interno l'enunciato '**return**'
- Vediamo allora l'esempio inziale:

Esempio:

```
def area(base, altezza):
    area=base*altezza
    return area
```

header o intestazione della funzione

Corpo della funzione

Parametri in ingresso alla funzione (input)
Sono i valori che devo passare alla funzione al momento della chiamata

main.py saved

```
1  def area(base, altezza):
2      area=base*altezza
3      return area
4
5  #flusso principale del programma
6  b = 2
7  a = 4
8  A = area(b, a)#chiamata della funzione
9  print("Area: %d" % A)
```

```
Area: 8
```

Enunciato **return**: termina la funzione e restituisce il risultato, ovvero l'output (non è obbligatoria la sua presenza, dipende da che funzione si crea)

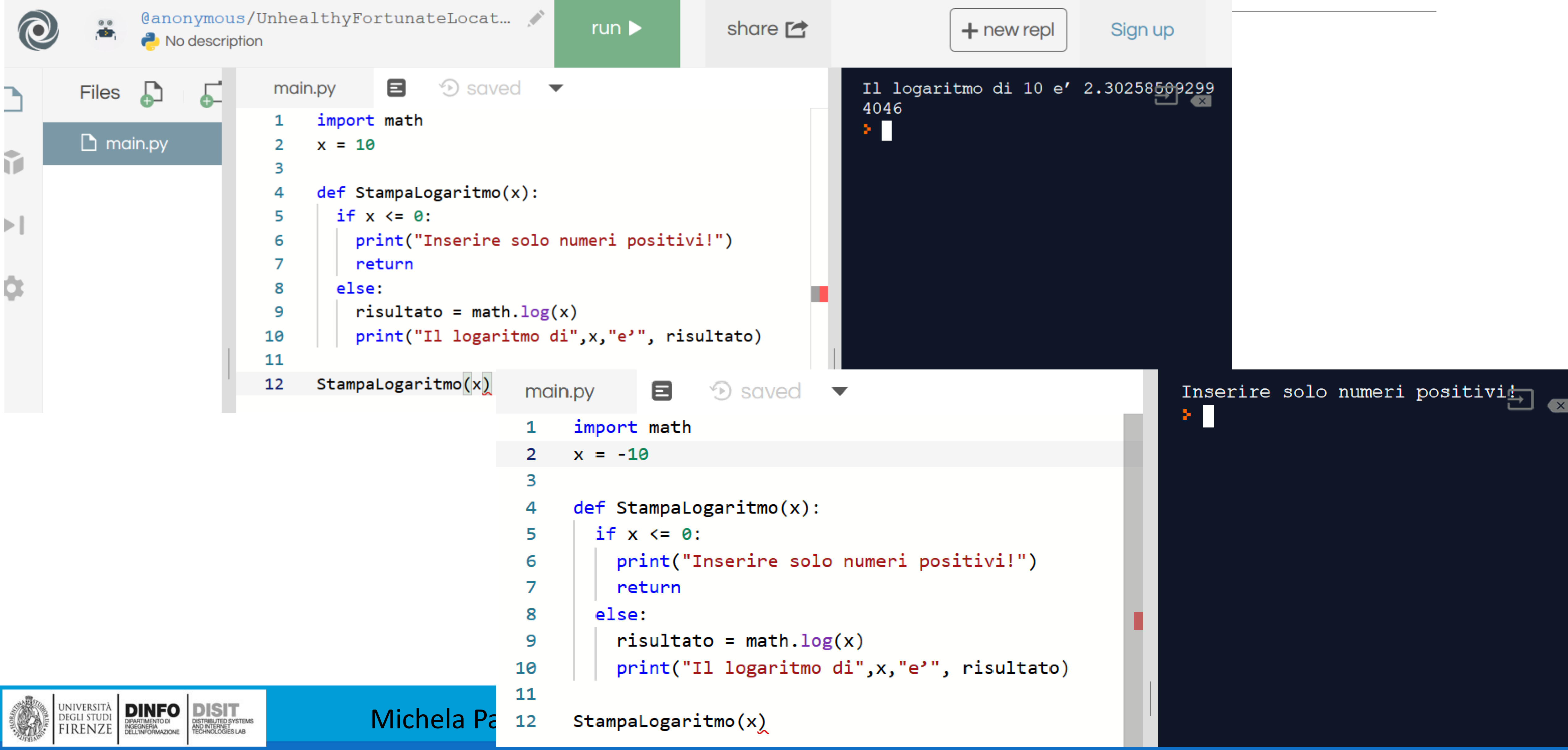
Istruzione return

L'istruzione return permette di terminare l'esecuzione di una funzione prima di raggiungerne la fine.

Questo può servire ad esempio quando viene riconosciuta una condizione d'errore

```
import math
x = -10

def StampaLogaritmo(x):
    if x <= 0:
        print("Inserire solo numeri positivi!")
        return
    else:
        risultato = math.log(x)
        print("Il logaritmo di",x,"e’", risultato)

StampaLogaritmo(x)
```

Istruzione return - 2



Istruzione return - 3

```
def myfunction():
        return 3+3
        print("Hello, World!")

print(myfunction())
```

NOTA: gli statement che si trovano dopo il return NON vengono eseguiti, infatti… se si Lancia il programma contenente la sola funzione descritta sopra, come risultato si avrà solo la stampa della somma

```
6
>
```

Istruzione return - 4

```
def myfunction():
	i=0
	while i<3:
		print('hello')
		return
		i=i+1

myfunction()
```

```
def myfunction():
	i=0
	while i<3:
		print('hello')
		i=i+1

myfunction()
```

- Che differenza c' fra queste due funzioni? Cosa ci si aspetta come risultato se si lanciano?

Sintassi: Funzione Produttiva e Vuota

- Da quanto visto appare chiaro che:
- Le funzioni Vuote NON presentano, nella loro definizione, l'enunciato 'return'. Le funzioni vuote possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non restituiscono un valore
- Al contrario le Funzioni Produttive, ovvero le funzioni che restituiscono un valore (producono un valore in output), presentano al loro interno l'enunciato 'return'

```
def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)
```

FUNZIONE VUOTA

```
def area(base, altezza):
    area=base*altezza
    return area
```

FUNZIONE PRODUTTIVA

Perché usare le funzioni? (1)

- Potrebbe non essere ancora ben chiaro perché valga la pena di suddividere il programma in funzioni
- Ecco alcuni motivi:
  - Creare una nuova funzione dà modo di dare un nome a un gruppo di istruzioni, rendendo il programma più facile da leggere e da correggere
  - Le funzioni possono rendere un programma più breve eliminando il codice ripetitivo
  - Se in un secondo tempo si deve fare una modifica, basterà farla in un posto solo
  - Dividere un programma lungo in funzioni, permette di correggere le parti una per una, per poi assemblarle in un complesso funzionante
  - Funzioni ben fatte sono spesso utili per più programmi
  - Quando se ne è scritta e corretta una, si riusare tale e quale in più programmi diversi

Debug

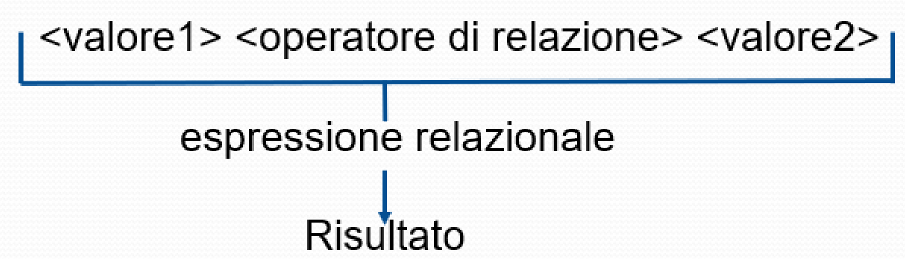
- Se si usa un editor di testo per scrivere gli script, si potrebbero incontrare dei problemi nell'indentare il codice con spazi e tabulazioni
- Il modo migliore per evitare problemi di questo tipo è usare esclusivamente gli spazi, non le tabulazioni. La maggior parte degli editor studiati per Python lo fanno in modo predefinito, ma alcuni no
- Tabulazioni e spazi di solito sono invisibili, rendendo difficoltoso il debug: cercate quindi di usare un editor che gestisca l'indentazione automaticamente o una IDE
- Non dimenticate di salvare il programma prima di eseguirlo: alcuni ambienti di sviluppo lo fanno automaticamente, ma altri no. In quest'ultimo caso, il programma che eseguite potrebbe non essere lo stesso che state guardando nell'editor
- Il debug può richiedere molto tempo se continuate a eseguire tutte le volte lo stesso programma non corretto!
- Siate certi che il codice che state guardando sia lo stesso che eseguite. Se avete qualche dubbio, mettete qualcosa come print('ciao') all'inizio del programma ed eseguitelo di nuovo. Se non vi compare 'ciao', allora non state eseguendo il programma giusto!

Istruzioni condizionali (1)

- Riprendiamo quanto visto per l'istruzione condizionale if, in particolare tenendo presente gli Operatori Booleani (o Relazionali) e gli Operatori Logici
- Ricordiamo che una espressione booleana è una espressione che può essere vera o falsa

Operatori Relazionali (*)

- Gli operatori relazionali e di uguaglianza (o disuguaglianza) confrontano il primo operando con il secondo per testare la validità della relazione in esame
- Gli operatori relazioni sono operatori **binari**:



- Il Risultato di una espressione relazionale è:
  - False se è falsa
  - True se la relazione testata è vera
- NOTA: ATTENZIONE A NON CONFONDERE L'OPERATORE RELAZIONALE == CON L'OPERATORE DI ASSEGNAZIONE =

Relazionali		
	Minore	<
	Maggiore	>
	Maggiore o uguale	>=
	Minore o uguale	<=
	Uguaglianza	==
	Diverso	!=

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Istruzioni condizionali (2)

- Riprendiamo quanto visto per l'istruzione condizionale if, in particolare tenendo presente gli Operatori Booleani (o Relazionali) e gli Operatori Logici
- Ricordiamo che una espressione booleana è una espressione che può essere vera o falsa
- Ci sono tre operatori Logici: and, or, not. Il significato di questi operatori è simile a quello comune (e, o, non)

Operatori Logici (*)

Logici	Congiunzione Negazione Disgiunzione	and not or

A	**NOT** *A*
F	*V*
V	*F*

A	*B*	*A* **AND** *B*
F	*F*	*F*
F	*V*	*F*
V	*F*	*F*
V	*V*	*V*

A	*B*	*A* **OR** *B*
F	*F*	*F*
F	*V*	*V*
V	*F*	*V*
V	*V*	*V*

Istruzioni condizionali (3)

```
a = 29

b = 12

c = 9

if(a>b and a!=6):

            print ("Entrambe le condizioni verificate!\n\n")

if(c>b or a!=6):

            print("Almeno una delle due condizioni è vera!\n\n")

if((c>b or a!=6) and (c>b or c!=6)):

            print("Esempio di composizione...")

#può servire che il corpo sia privo di istruzioni, in questo #caso si usa l'istruzione pass

a=1

if(a<10):

  pass

else:

  print("Hello")
```

```
Entrambe le condizioni verificate!


Almeno una delle due condizioni è vera!


Esempio di composizione...
>
```

Esecuzione Alternativa (1)

- Una seconda forma di istruzione if è l'esecuzione alternativa, dove esistono due possibili azioni (corpo1 e corpo2)
- Il valore della condizione determina quale delle due azione verrà eseguita

```
a = 29

b = 12


if(a>b and a!=29):

  print ("Entrambe le condizioni verificate!\n\n")

else:

  print ("NON sono verificate entrambe le condizioni!\nInfatti a vale:"+str(a)+"\n")
```

```
NON sono verificate entrambe le condizioni!
Infatti a vale: 29

>
```

Condizioni in serie (1)

- A volte è necessario considerare più di due possibili sviluppi
- Nel programma ci saranno quindi più di due ramificazioni (es. corpo1,.. , corpoN)
- **elif** è l'abbreviazione di else if
- Ovviamente viene eseguito solo uno dei tre rami (corpo3)
- Non c'è alcun limite al numero di istruzioni elif
- Se esiste una clausola else (che quindi è facoltativa) , deve essere messa per ultima come nel seguente esempio:

```
a = 29
b = 29

if(a>b):
   print ("a è maggiore di b!") #corpo1
elif(a<b):
   print ("a è minore di b!") #corpo2
else:
   print ("a è uguale a b!") #corpo3
```

```
a è uguale a b!
>
```

Condizioni nidificate (1)

- E' possibile anche inserire una istruzione condizionale, nel corpo di un'altra istruzione condizionale:

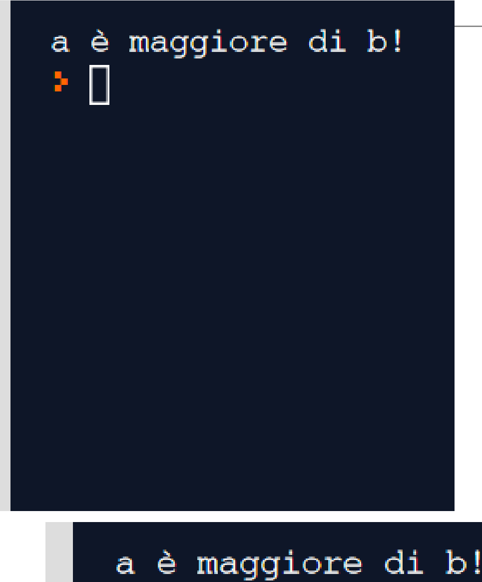
```
a = 29
b = 12

if(a>b):
    print ("a è maggiore di b!")
else:
    if(a<b):
        print ("a è minore di b!")
    else:
        print ("a è uguale a b!")
```

- Si noti che le condizioni ramificate risultano di difficile lettura, meglio usarle con moderazione!

Condizioni nidificate (2)

main.py saved

```python
a = 29
b = 12

if(a>b):
    print ("a è maggiore di b!")
elif(a<b):
    print ("a è minore di b!")
else:
    print ("a è uguale a b!")
```



main.py saving...

```python
a = 29
b = 12

if(a>b):
    print ("a è maggiore di b!")
else:
    if(a<b):
        print ("a è minore di b!")
    else:
        print ("a è uguale a b!")
```

a è maggiore di b!

- Si noti che:
- Questi due programmi producono lo stesso risultato
- le condizioni ramificate risultano di difficile lettura, meglio usarle con moderazione!
- La prima soluzione è più leggibile

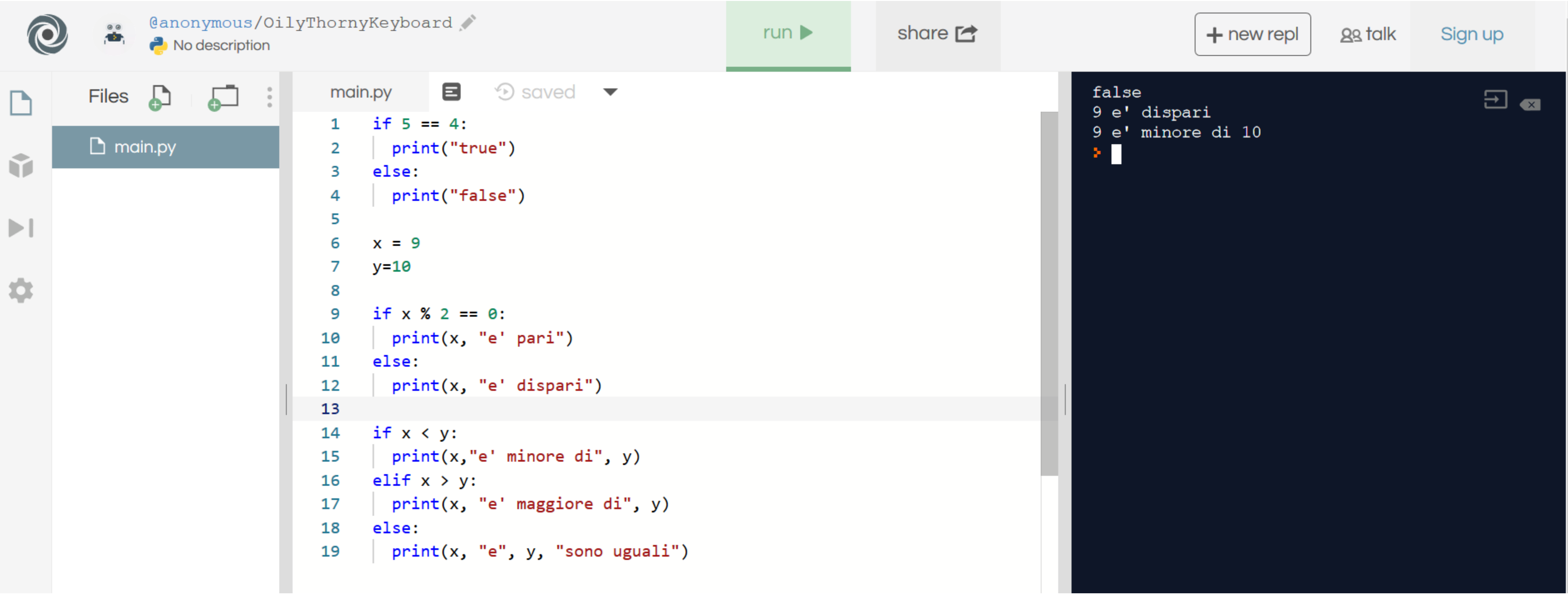
Condizioni nidificate (3)

main.py saved

```
x=9
#operazioni annidate
if x>0:
    if x < 10:
        print("x e’ un numero positivo minore di 10")
print() #stampo linea vuota
#il seguente codice è equivalente:
if 0<x and x<10:
    print("x e’ un numero positivo minore di 10.")
print() #stampo linea vuota
#ancora codice equivalente
if 0<x<10:
    print("x e’ un numero positivo minore di 10.")
```

```
x e’ un numero positivo minore di 10

x e’ un numero positivo minore di 10.

x e’ un numero positivo minore di 10.
```

Condizioni nidificate (4)



Concetto di Ricorsione (1)

- Si è visto il concetto di definizione di funzione e di chiamata di una funzione
- Quando si realizza un programma risulta quindi del tutto normale che una funzione ne chiami un'altra
- E' anche possibile che una funzione richiami se stessa
- Per funzione ricorsiva o algoritmo ricorsivo, si intende un procedimento/algoritmo definito in termini di se stesso
- Per esempio anche il prodotto di due numeri può essere definito in termini ricorsivi:
  - Definizione della funzione Prodotto in modo NON ricorsivo:

    P(q,r) = q*r
  - Definizione della funzione Prodotto scritta in modo ricorsivo:

    Se (q>0), allora: P(q,r) = r + P(q-1, r)

    Altrimenti P(q,r) = 0
  - Il criterio di arresto permette di contenere il processo ricorsivo che altrimenti in teoria, andrebbe avanti all'infinito

Esempio di Ricorsione: il fattoriale (1)

- Un numero fattoriale viene rappresentato dal numero stesso con il punto esclamativo
- La definizione naturale è basata sul principio di induzione:

  Se n<1, allora n! = 1 #passo base

  Altrimenti, n! = n *(n-1)! #passo di induzione
- La possibilità di usare la ricorsione nei linguaggi di programmazione permette la traduzione diretta della definizione di un programma
- La funzione fattoriale traduce in modo diretto la definizione data, separando passo base e passo di induzione

```
def fattoriale(n):
    if(n<1):
        return 1;
    else:
        return n*fattoriale(n-1)
```

Esempio di Ricorsione: il fattoriale (2)

```
def fattoriale(n):
    if(n<1):
        return 1;
    else:
        return n*fattoriale(n-1)

def main():
    N = 10
    f = fattoriale(N)
    print('Calcolo di '+str(N)+' (funzione ricorsiva)!\n’)
    print(f)

main()
```

```
Calcolo di 10fattoriale (funzione ricorsiva)!

3628800
>
```

Esempio di Ricorsione: il fattoriale (3)

main.py saving...

```
1   def fattoriale(n):
2       if(n<1):
3           return 1;
4       else:
5           return n*fattoriale(n-1)
6
7
8   def main():
9       N = 10
10      f = fattoriale(N)
11      print('Calcolo di '+str(N)+ 'fattoriale (funzione ricorsiva)!\n')
12      print(f)
13
14  main()
```

```
Calcolo di 10fattoriale (funzione ricorsiva)!

3628800
>
```

Esempio di Ricorsione: righe vuote (1)

Deriva dal concetto di chiamata di una funzione

Una funzione può chiamare un'altra funzione

Una funzione può anche chiamare se stessa: questo è il concetto di Ricorsione

```
def NRigheVuote(n):
    if n > 0:
        print()
        NRigheVuote(n-1)

N=5
print("---Nrighe--")
NRigheVuote(N)
print("ho stampato "+str(N)+" righe vuote")
```

Esempio di Ricorsione: righe vuote (2)

main.py saved

```
def UnaRigaVuota():
    print()
def TreRigheVuote():
    UnaRigaVuota()
    UnaRigaVuota()
    UnaRigaVuota()

print("--3righe vuote---")
TreRigheVuote()
print("ho stampato tre righe vuote")

def NRigheVuote(n):
    if n > 0:
        print()
        NRigheVuote(n-1)
N=5
print("---Nrighe--")
NRigheVuote(N)
print("ho stampato "+str(N)+" righe vuote")
```

```
--3righe vuote---



ho stampato tre righe vuote
---Nrighe--





ho stampato 5 righe vuote
```

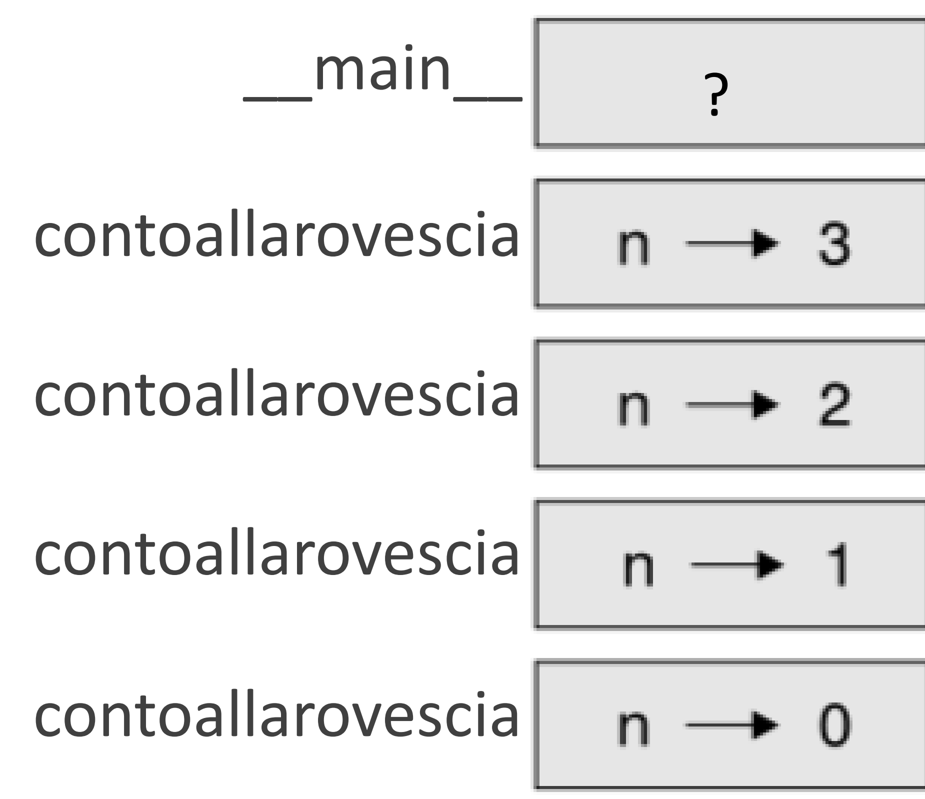
Esempio di Ricorsione: conto alla rovescia (1)

```python
def contoallarovescia(n):
    if(n<0):
        print("Non conto, immettere numero positivo")
    elif(n==0):
        print("Fine!")
    else:
        print(n)
        contoallarovescia(n-1)

def main():
    contoallarovescia(-9)
    contoallarovescia(10)


main()
```

```
Non conto, immettere numero positivo
10
9
8
7
6
5
4
3
2
1
Fine!
```

Concetto di ricorsione: stack delle chiamate

- Ogni volta che una funzione viene chiamata, Python crea un nuovo frame della funzione, contenente le variabili locali definite all'interno della funzione ed i suoi parametri
- Nel caso di una funzione ricorsiva possono esserci più frame riguardanti una stessa funzione allo stesso tempo
- Il livello superiore dello stack, è il frame per il main
- Se nel main NON si creano variabili locali e si passano parametri (chiamando le funzioni), il frame del main è vuoto. Altrimenti cosa succede?



Ricorsione infinita (1)

```
def Ricorsione():
    Ricorsione()

Ricorsione()
```

- Se una ricorsione non raggiunge mai il suo stato di base, la chiamata alla funzione viene eseguita all'infinito ed in teoria il programma non giunge mai alla fine
- Questo è il caso di ricorsione **infinita** che **NON** deve verificarsi
- Nella maggior parte degli ambienti un programma con una ricorsione infinita non viene eseguito senza fine, dato che ogni chiamata ad una funzione impegna un po' di memoria del computer e questa memoria prima o poi finisce.
- Python stampa un messaggio d'errore quando è stato raggiunto il massimo livello di ricorsione possibile

Ricorsione infinita (2)

Python stampa un messaggio d'errore quando è stato raggiunto il massimo livello di ricorsione possibile:

main.py saved

```
1  def Ricorsione():
2      Ricorsione()
3
4  Ricorsione()
```

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    Ricorsione()
  File "main.py", line 2, in Ricorsione
    Ricorsione()
  File "main.py", line 2, in Ricorsione
    Ricorsione()
  File "main.py", line 2, in Ricorsione
    Ricorsione()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Funzioni Produttive

Ripartendo da… Esempio di Funzione Produttiva (1)

- Le funzioni Produttive restituiscono un valore (producono un valore in output), per poterlo fare dal punto di vista sintattico, presentano al loro interno l'enunciato '**return**'
- Vediamo allora l'esempio inziale:

Esempio:

```
def area(base, altezza):
    area=base*altezza
    return area
```

header o intestazione della funzione

Corpo della funzione

Parametri in ingresso alla funzione (input)
Sono i valori che devo passare alla funzione al momento della chiamata

main.py saved

```
1  def area(base, altezza):
2      area=base*altezza
3      return area
4
5  #flusso principale del programma
6  b = 2
7  a = 4
8  A = area(b, a)#chiamata della funzione
9  print("Area: %d" % A)
```

```
Area: 8
```

Enunciato **return**: termina la funzione e restituisce il risultato, ovvero l'output (non è obbligatoria la sua presenza, dipende da che funzione si crea)

Confronto tra Funzione produttiva e non P. (1)

```
#Funzione che non restituisce niente (si ha solo la stampa nel prompt) con nessun input
def Stampa_valore():
  print("Testo deciso dalla funzione!")
#Definizione di funzione che non restituisce niente con un valore in input
def Stampa_valore_input(x):
  print("Stampo il valore passato dal prompt: "+x)
#Definizione di una funzione che ha come input due valori (x e y) e restituisce un valore in output
(z)
def Stampa_Somma(x,y):
  z= x+y
  return(z)

#----------------chiamata delle funzioni--------------------
Stampa_valore()
x = input("Cosa devo stampare?")
Stampa_valore_input(x)
a=2
b=4
# Quando la funzione rende un valore, si puo' mettere a destra dell'operatore di assegnazione. Assegno
l'output della funzione Stampa_somma alla variabile
t = Stampa_Somma(a,b)
print("La somma delle variabili a e b risulta: ")
print(t)
```

Confronto tra Funzione produttiva e non P. (2)

main.py saved

```
1  #Definizione di funzione che non restituisce niente (si ha solo la stampa nel prompt) con nessun input
2  def Stampa_valore():
3      print("Testo deciso dalla funzione!")
4  #Definizione di funzione che non restituisce niente  con un valore in input
5  def Stampa_valore_input(x):
6      print("Stampo il valore passato dal prompt: "+x)
7  #Definizione di una funzione che ha come input due valori (x e y) e restituisce un valore in output (z)
8  def Stampa_Somma(x,y):
9      z= x+y
10     return(z)
11
12 Stampa_valore()
13 x = input("Cosa devo stampare?")
14
15 Stampa_valore_input(x)
16 a=2
17 b=4
18 # Quando la funzione rende un valore, si puo' mettere a destra dell'operatore di assegnazione. Assegno l'output della funzione Stampa_somma alla variabile
19 t = Stampa_Somma(a,b)
20 print("La somma delle variabili a e b risulta: ")
21 print(t)
```

```
Testo deciso dalla funzione!
Cosa devo stampare? Ciao, stampa questa frase!
Stampo il valore passato dal prompt:  Ciao, stampa questa frase!
La somma delle variabili a e b risulta:
6
>
```

Valori di ritorno (1)

```
#import della libreria math
import math
#definizione delle funzione
def AreaDelCerchio(Raggio):
  temp = math.pi * Raggio**2
  return temp

#definizione della variabile r nel main
r=4
#chiamata della funzione con input r definito nel main
#si assegna alla variabile z l'output della funzione
z = AreaDelCerchio(r)
print(z)

#attenzione la variabile Raggio NON è definita nel main!!!
#Raggio ha valore solo nel contesto della definizione della funzione
AreaDelCerchio(...). Quindi la chiamata seguente da' errore!
#AreaDelCerchio(Raggio)
```

Valori di ritorno (2)

main.py saved

```
1  #import della libreria math
2  import math
3  #definizione delle funzione
4  def AreaDelCerchio(Raggio):
5    temp = math.pi * Raggio**2
6    return temp
7
8  #definizione della variabile r nel main
9  r=4
10 #chiamata della funzione con input r definito nel main
11 #si assegna alla variabile z l'output della funzione
12 z = AreaDelCerchio(r)
13 print(z)
14
15 #attenzione la variabile Raggio NON è definita nel main!!!
16 #Raggio ha valore solo nel contesto della definizione della funzione
   AreaDelCerchio(...). Quindi la chiamata seguente da' errore!
17
18 #AreaDelCerchio(Raggio)
```

```
50.26548245743669
>
```

Valori di ritorno (3)

```
#import della libreria math
import math
#definizione delle funzione
def AreaDelCerchio(Raggio):
  temp = math.pi * Raggio**2
  return temp

[…]
#attenzione la variabile Raggio NON è
definita nel main!!!
#Raggio ha valore solo nel contesto della
definizione della funzione (variabile locale)
AreaDelCerchio(...). Quindi la chiamata
seguente da' errore!

AreaDelCerchio(Raggio)
```

```
Traceback (most recent call last):
  File "main.py", line 18, in <module>
    AreaDelCerchio(Raggio)
NameError: name 'Raggio' is not defined
```

Funzioni Produttive con valori di ritorno multipli (1)

```
#definizione di funzione con valore di ritorno multiplo
def ValoreAssoluto(x):
  if x < 0:
    return -x
  else:
    return x
#definizione della variabile a nel main
a = -10
#chiamata e dalla funzione e stampa del suo valore di ritorno
print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

E' una buona idea assicurarci che ogni ramificazione possibile porti ad un'uscita dalla funzione con un'istruzione di return

main.py saved

```
1  #definizione di funzione con valore di ritorno multiplo
2  def ValoreAssoluto(x):
3    if x < 0:
4      return -x
5    else:
6      return x
7
8  a = -10
9  print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

Funzioni Produttive con valori di ritorno multipli (2)

```
#definizione di funzione con valore di ritorno multiplo
def ValoreAssoluto(x):
    if x < 0:
        return -x
    elif x > 0:
        return x

a = -10
print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

E’ una buona idea assicurarci che ognuna delle ramificazioni possibili porti ad un’uscita dalla funzione con un’istruzione di return

- Questo programma non è corretto in quanto non è prevista un’uscita con return nel caso x sia 0.
- Se a=0, allora il valore di ritorno sarà un valore speciale chiamato None

Funzioni Produttive con valori di ritorno multipli (3)

- Questo programma non è corretto in quanto non ` e prevista un'uscita con return nel caso x sia 0.
- Se a=0, allora il valore di ritorno sarà un valore speciale chiamato None

E' una buona idea assicurarci che ognuna delle ramificazioni possibili porti ad un'uscita dalla funzione con un'istruzione di return

main.py saved

```python
#definizione di funzione con valore di ritorno multiplo
def ValoreAssoluto(x):
    if x < 0:
        return -x
    elif x > 0:
        return x

a = 0
print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

```
Il valore assoluto di 0 è: None
```

Funzioni Produttive con valori di ritorno multipli (4)

```
#definizione di funzione con valore di ritorno multiplo
def ValoreAssoluto(x):
    if x < 0:
        return -x
    elif x >= 0:
        return x

a = 0
print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

E' una buona idea assicurarci che ognuna delle ramificazioni possibili porti ad un'uscita dalla funzione con un'istruzione di return

main.py saved

```
1  #definizione di funzione con valore di ritorno multiplo
2  def ValoreAssoluto(x):
3      if x < 0:
4          return -x
5      elif x >= 0:
6          return x
7
8  a = 0
9  print("Il valore assoluto di "+str(a)+" è: "+str(ValoreAssoluto(a)))
```

```
Il valore assoluto di 0 è: 0
```

Esercizi

Esercizi:

scrivi una funzione *Confronto* che ritorna 1 se x>y, 0 se x==y e -1 se x<y

Scrivere una funzione Stampa_maggiore t.c.:

- Stampa x, se x > y
- Stampa y, se y >x
- Stampa 'Le variabili hanno lo stesso valore pari a: '- valore

Sviluppo incrementale (1)

- Scopo dello sviluppo incrementale: evitare lunghe sessioni di debug, aggiungendo e testando continuamente piccole parti di codice alla volta
- Una buona tecnica per affrontare lo sviluppo di algoritmi complessi è ricorrere allo **sviluppo incrementale**
- Supponiamo di voler trovare la distanza tra due punti A e B conoscendone le coordinate:
  - A (x1,y1) e B(x2,y2)
- Con il teorema di Pitagora sappiamo che la distanza si calcola come:

$$distanza = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Sviluppo incrementale (2)

- Con il teorema di Pitagora sappiamo che la distanza si calcola come:

$$distanza = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Si scompone il problema in sotto-problemi
  - distanza = [(distanza_tra_due_punti(x1,x2)^2 + (distanza_tra_due_puntiy1,y2)^2]^(1/2)
- Si definisce la funzione distanza_tra_due_punti(a1,a2):

```
def distanza_tra_due_punti(a1,a2):
    return a1-a2
```

- Si definisce una funzione valore_quadrato(x):

```
def valore_quadrato(x):
    return x**2
```

Sviluppo incrementale (3)

- Con il teorema di Pitagora sappiamo che la distanza si calcola come:

$$distanza = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Si definisce la funzione distanza, richiamando le funzioni precedenti

```
#import della libreria math
import math

def distanza_tra_due_punti(a1,a2):
  return a1-a2

def valore_quadrato(x):
  return x**2

def distanza(a1,a2, b1, b2):
  A = distanza_tra_due_punti(a1,a2)
  B = distanza_tra_due_punti(b1,b2)
  #uso della funzione sqrt della libreria math importata in precedenza
  return(math.sqrt(int(valore_quadrato(A)) + int (valore_quadrato(B))))
```

COMPOSIZIONE DI FUNZIONI

Sviluppo incrementale (4)

- Con il teorema di Pitagora sappiamo che la distanza si calcola come:

$$distanza = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Nel main si assegnano le coordinate ai punti X(x1,y1) e Y(x2,y2):

```
#coordinate punto X
x1= 1
y1 = 1
#coordinate punto Y
x2= 4
y2 = 3
```

- Si richiamano le funzioni:

```
print("Distanza tra x1 e x2: "+str(distanza_tra_due_punti(x1,x2)))
print("Distanza tra y1 e y2: "+str(distanza_tra_due_punti(y1,y2)))
#distanza fra due punti
print()
print("Distanza tra X e Y: "+str(distanza(x1,x2, y1, y2)))
```

Sviluppo incrementale (5)

Programma completo

```
import math #import della libreria math
#definizione delle funzioni
def distanza_tra_due_punti(a1,a2):
  return a1-a2
def valore_quadrato(x):
  return x**2
def distanza(a1,a2, b1, b2):
  A=distanza_tra_due_punti(a1,a2)
  B =distanza_tra_due_punti(b1,b2)
  return(math.sqrt(int(valore_quadrato(A)) + int (valore_quadrato(B))))
#main
#coordinate punto X
x1= 1
y1 = 1
#coordinate punto Y
x2= 4
y2 = 3
#chiamata delle funzioni e stampa dei risultati
print("Distanza tra x1 e x2: "+str(distanza_tra_due_punti(x1,x2)))
print("Distanza tra y1 e y2: "+str(distanza_tra_due_punti(y1,y2)))
#distanza fra due punti
print()
print("Distanza tra X e Y: "+str(distanza(x1,x2, y1, y2)))
```

Alcune regole di buona programmazione

- Scomposizione del problema in sotto-problemi
- Iniziare con un programma funzionante facendo piccoli cambiamenti (in modo da scoprire facilmente dove siano localizzati gli eventuali errori)
- Usare variabili temporanee per memorizzare i valori intermedi, così da poterli stampare e controllare
- Quando il programma funziona perfettamente, rimuovere le istruzioni temporanee
- Consolidare le istruzioni in espressioni composite (attenzione a non rendere il programma illeggibile!)

Funzioni booleane (1)

- Le funzioni possono anche ritornare valori booleani (vero/falso e in Python.. True/False)

```
def Divisibile(x, y):
   if x % y == 0:
      return True # x e’ divisibile per y: ritorna vero
   else:
      return False # x non e’ divisibile per y: ritorna false

a = 20
b = 6
result = Divisibile(a,b)
print("Valore booleno restituito dalla funzione: ")
print(result)
```

Funzioni booleane (2)

- Le funzioni possono anche ritornare valori booleani (vero o falso)
- I valori booleani non sono sempre facilmente interpretabili dall’utente finale, quindi è possibile tradurli in frasi (in questo modo il risultato è ‘human readable’):

```
def Divisibile(x, y):
   if x % y == 0:
      return True # x e’ divisibile per y: ritorna
vero
   else:
      return False # x non e’ divisibile per y:
ritorna false

a = 20
b = 6
result = Divisibile(a,b)
print("Valore booleno restituito dalla funzione: ")
print(result)
print("Traduzione leggibile per l'utente finale: ")
if(Divisibile(a,b) == 1):
   print(str(a)+" è divisibile per "+str(b))
else:
   print(str(a)+" NON è divisibile per "+str(b))
```

Funzioni booleane (3)

main.py

```python
def Divisibile(x, y):
    if x % y == 0:
        return True # x e’ divisibile per y: ritorna vero
    else:
        return False # x non e’ divisibile per y: ritorna false

a = 20
b = 6
result = Divisibile(a,b)
print("Valore booleno restituito dalla funzione: ")
print(result)

print("Traduzione leggibile per l'utente finale: ")
if(Divisibile(a,b) == 1):
    print(str(a)+" è divisibile per "+str(b))
else:
    print(str(a)+" NON è divisibile per "+str(b))
```

```
Valore booleno restituito dalla funzione:
False
Traduzione leggibile per l'utente finale:
20 NON è divisibile per 6
```

Iterazione

Istruzione while (1)

- Si usa il concetto di iterazione quando è necessario ripetere la stessa operazione più volte
- “Finchè (while) n è più grande di 0, stampa il valore di n e poi diminuiscilo di 1. Quando arrivi a 0 stampa la stringa Partenza!”

```python
#definizione della funzione
def ContoAllaRovescia(n):
        while n > 0:
                print(n)
                n = n-1
        print("Partenza!")

x = 10
#chiamata della funzione
ContoAllaRovescia(x)
```

main.py saved

```python
1  #definizione della funzione
2  def ContoAllaRovescia(n):
3    while n > 0:
4      print(n)
5      n = n-1
6    print("Partenza!")
7
8  x = 10
9  #chiamata della funzione
10 ContoAllaRovescia(x)
11
```

```
10
9
8
7
6
5
4
3
2
1
Partenza!
```

Istruzione while (2)

- Si usa il concetto di iterazione quando è necessario ripetere la stessa operazione più volte

```
#definizione della funzione
def ContoAllaRovescia(n):
        while n > 0:
                print(n)
                n = n-1
        print("Partenza!")

x = 10
#chiamata della funzione
ContoAllaRovescia(x)
```

- lettura più formale del ciclo while:
  - passo 1) Si valuta la condizione controllando se essa è vera (True) o falsa (False)
  - passo 2) Se la condizione è falsa, il flusso esce dal ciclo while e continua l'esecuzione dalla prima istruzione che segue il ciclo (X=10)
  - passo 3) Se la condizione è vera, si eseguono tutte le istruzioni nel corpo del while e si torna al passo 1

Istruzione while (3)

Si usa il concetto di iterazione quando è necessario ripetere la stessa operazione più volte

```
#definizione della funzione
def ContoAllaRovescia(n):
        while n > 0:
                print(n)
                n = n-1
        print("Partenza!")

x = 10
#chiamata della funzione
ContoAllaRovescia(x)
```

- Il corpo del ciclo while consiste di **tutte le istruzioni che seguono l'intestazione e che hanno la stessa indentazione**
- se la condizione è falsa al primo controllo, le istruzioni del corpo non sono mai eseguite
- Il corpo del ciclo dovrebbe cambiare il valore di una o più variabili così che la condizione possa prima o poi diventare falsa e far terminare il ciclo. In caso contrario il ciclo si ripeterebbe all'infinito, determinando un ciclo infinito

Tabelle

Esempio: tabella dei logaritmi

Il programma seguente stampa una sequenza di valori nella colonna di sinistra e il loro logaritmo in quella di destra

La stringa '\t' rappresenta la tabulazione

Si ricorda che la funzione log usa il logaritmo dei numeri naturali e.

```
import math
x = 1.0
while x < 10.0:
    print (x, '\t', math.log(x))
x = x + 1.0
```

```
1.0     0.0
2.0     0.693147180559453
3.0     1.09861228866811098
4.0     1.38629436111198906
5.0     1.6094379124341003
6.0     1.791759469228055
7.0     1.9459101490553132
8.0     2.0794415416798357
9.0     2.1972245773362196
>
```

Tabelle e concetto di tabulazione

- A mano a mano che caratteri e stringhe sono mostrati sullo schermo un marcatore invisibile chiamato cursore tiene traccia di dove andrà stampato il carattere successivo. Dopo un'istruzione print() il cursore normalmente si posiziona all'inizio della riga successiva
- Il carattere di tabulazione sposta il cursore a destra finchè quest'ultimo raggiunge una delle posizione di stop delle tabulazioni.
- Queste posizioni si ripetono a distanze regolari, tipicamente ogni 4 o 8 caratteri.
- Le tabulazioni sono utili per allineare in modo semplice le colonne di testo
- Il carattere di tabulazione fa in modo che la posizione della seconda colonna non dipenda dal numero di cifre del valore nella prima.

Backslash e escape

- Il carattere di backslash ’\’ in ’\t’ indica l’inizio di una sequenza di escape
- Le sequenze di escape sono usate per rappresentare caratteri invisibili come la tabulazione (’\t’) e il ritorno a capo (’\n’)
- Queste sequenze possono comparire in qualsiasi punto di una stringa:

```
print (x, '\t', math.log(x))
```

- Esercizio: scrivi una stringa singola che quando stampata

produca

questo

risultato.

esercizio

print('produca','\n', '\t','questo','\n','\t','\t','risultato')

main.py saved

```
1  print('produca','\n', '\t','questo','\n','\t','\t','risultato')
```

```
produca
         questo
                 risultato
```

Tabelle bidimensionali (1)

- Una tabella bidimensionale è una tabella dove leggi un valore all'intersezione tra una riga ed una colonna, la tabella della moltiplicazione ne è un buon esempio
- Supponiamo di voler stampare la tabella della moltiplicazione per i numeri da 1 a 6
- Si inizia stampando i multipli di 2

```
i = 1
while i <= 6:
    print(2*i)
    i = i + 1
```

… print()

- A partire da Python 3 print() è una funzione e prende degli argomenti opzionali
- Uno di questi si chiama `end`, e serve per specificare come deve terminare la stampa.
- Per default la stampa termina con end='\n'

#codice equivalente al precedente:

```
i = 1
while i <= 6:
    print(2*i, end='\n')
    i = i + 1
```

Se non si vuole andare a capo ad ogni ciclo, si può inserire una terminazione diversa, ad esempio end= ' '

Tabelle bidimensionali (2)

main.py saved

```
1  i = 1
2  while i <= 6:
3      print(2*i)
4      i = i + 1
5
```

```
2
4
6
8
10
12
>
```

main.py saved

```
1  i = 1
2  while i <= 6:
3      print(2*i, end= )
4      i = i + 1
5
```

```
2
4
6
8
10
```

```
print(value, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)
```

string appended after the last value, default a newline.

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.

Guida online
Da cui si notano i parametri opzionali di print()

Tabelle bidimensionali (3)

main.py

```
1   i = 1
2   while i <= 6:
3       print(2*i,end='\n')
4       i = i + 1
5
```

```
2
4
6
8
10
12
```

Cambiando il carattere di separazione , la stampa varia.
Il parametro di default (ovvero se non si usa parametro aggiuntivo) è \n

main.py

```
1   i = 1
2   while i <= 6:
3       print(2*i, end=' ')
4       i = i + 1
5
```

```
2 4 6 8 10 12
```

Incapsulamento e generalizzazione (1)

- L'incapsulamento è il processo di inserire un pezzo di codice all'interno di una funzione così da poter chiamare le funzioni
- **Generalizzare** significa prendere qualcosa di specifico per farlo diventare generale: nel nostro caso prendere il programma che calcola i multipli di 2 e fargli calcolare i multipli di un qualsiasi numero intero

```
def StampaMultipli(n):
        i = 1
        while i <= 6:
                print (n*i, '\t', end='')
                i = i + 1

x =5
StampaMultipli(5)
```

main.py   saved

```
1  def StampaMultipli(n):
2    i = 1
3    while i <= 6:
4      print (n*i, '\t', end=' ')
5      i = i + 1
6
7  x =5
8  StampaMultipli(5)
9
```

```
5    10    15    20    25    30    ›
```

Incapsulamento e generalizzazione (2)

- Se chiamiamo la funzione con l'argomento 2 otteniamo lo stesso risultato di prima. Con l'argomento 3 il risultato è:
  - 3 6 9 12 15 18
- Con l'argomento 4:
  - 4 8 12 16 20 24
- Per stampare la tabella delle moltiplicazioni basta richiamare la funzione 'StampaMultipli(n)' in un ciclo while:

```
i = 1
while i <= 6:
      StampaMultipli(i)
      i = i + 1
      print('\n')
```

Incapsulamento e generalizzazione (3)

NOTA: per stampare una tabella sono quindi necessari due cicli:

Uno, quello del main() per le righe e l'altro , quello contenuto nel corpo della funzione StampaMultipli, per le colonne

main.py saved

```
def StampaMultipli(n):
    i = 1
    while i <= 6:
        print (n*i, '\t', end='')
        i = i + 1

i = 1
while i <= 6:
    StampaMultipli(i)
    i = i + 1
    print('\n')
```

```
1    2    3    4    5    6

2    4    6    8    10   12

3    6    9    12   15   18

4    8    12   16   20   24

5    10   15   20   25   30

6    12   18   24   30   36
```

Variabili locali e globali

- Si ricorda che: le variabili create all'interno della definizione di una funzione sono variabili locali, hanno validità solo all'interno della funzione (nelle righe di codice che servono per definire la funzione)
- Non è possibile accedere al valore di una variabile locale al di fuori della funzione che la ospita
- Si possono usare variabili con lo stesso nome (i) all'interno di un programma trattandole come variabili distinte, sempre se non si trovano all'interno di una stessa funzione

main.py saved

```
1  def StampaMultipli(n):
2      i = 1
3      while i <= 6:
4          print (n*i, '\t', end='')
5          i = i + 1
6
7  i = 1
8  while i <= 6:
9      StampaMultipli(i)
10     i = i + 1
11     print('\n')
```

Generalizzazione ulteriore (1)

- Supponiamo di volere estendere la tabella delle moltiplicazioni ad una dimensione generica e non solo ad una 6X6

```
def StampaMultipli(n):
    i = 1
while i <= 6:
    print (n*i, '\t', end=‘’)
    i = i + 1
def tabellaMoltiplicazioneGenerica(n):
    i=1
    while i <= n:
        StampaMultipli(i)
        i = i + 1
        print('\n’)

n=10
tabellaMoltiplicazioneGenerica(n)
```

Generalizzazione ulteriore (2)

main.py saved

```
1   def StampaMultipli(n):
2       i = 1
3       while i <= 6:
4           print (n*i, '\t', end='')
5           i = i + 1
6
7   def tabellaMoltiplicazioneGenerica(n):
8       i=1
9       while i <= n:
10          StampaMultipli(i)
11          i = i + 1
12          print('\n')
13
14
15  n=10
16  tabellaMoltiplicazioneGenerica(n)
17
```

```
1   2   3   4   5   6

2   4   6   8   10  12

3   6   9   12  15  18

4   8   12  16  20  24

5   10  15  20  25  30

6   12  18  24  30  36

7   14  21  28  35  42

8   16  24  32  40  48

9   18  27  36  45  54

10  20  30  40  50  60

>
```

Altra generalizzazione (1)

- Se si vogliono generalizzare anche le colonne…

```
def StampaMultipli(n, col):
    i = 1
    while i <= col:
        print (n*i, '\t', end=‘’)
        i = i + 1

def tabellaMoltiplicazioneGenerica(n, col):
    i=1
    while i <= n:
        StampaMultipli(i, col)
        i = i + 1
        print('\n')

n=10
tabellaMoltiplicazioneGenerica(n, n)
```

Altra generalizzazione (2)

main.py saved

```python
def StampaMultipli(n, col):
    i = 1
    while i <= col:
        print (n*i, '\t', end='')
        i = i + 1

def tabellaMoltiplicazioneGenerica(n, col):
    i=1
    while i <= n:
        StampaMultipli(i, col)
        i = i + 1
        print('\n')


n=10
tabellaMoltiplicazioneGenerica(n, n)
```

```
1	2	3	4	5	6	7	8	9	10

2	4	6	8	10	12	14	16	18	20

3	6	9	12	15	18	21	24	27	30

4	8	12	16	20	24	28	32	36	40

5	10	15	20	25	30	35	40	45	50

6	12	18	24	30	36	42	48	54	60

7	14	21	28	35	42	49	56	63	70

8	16	24	32	40	48	56	64	72	80

9	18	27	36	45	54	63	72	81	90

10	20	30	40	50	60	70	80	90	100

>
```

Stringhe

Riprendiamo il concetto di stringa

- Cosa si è già visto:
  - Una stringa è una sequenza di caratteri
  - In python le stringhe sono sequenze di caratteri UNICODE
  - la funzione print() prende in ingresso una stringa
  - Esistono degli operatori di formato per le stringhe
  - La funzione str() prende in ingresso ad esempio un numero per restituire come output una stringa
  - E’ possibile ricorrere all’uso degli indici per individuare i singoli caratteri uno alla volta

Stampare: Uso di print

- Si possono usare sia i doppi che i singoli apici
- Stampare stringhe:
  - print("Hello World")
  - print('Hello \n World')
  - print("La temperatura di oggi è 10 °Centigradi")
  - print("La temperatura di oggi è:\n 10 °Centigradi")
- Ci sono alcuni simboli speciali che permetto di formattare la stampa in uscita
  - Ad esempio \n manda a capo

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> print("La temperatura di oggi è 10 °Centigradi")
La temperatura di oggi è 10 °Centigradi
>>> print("La temperatura di oggi è:\n 10 °Centigradi")
La temperatura di oggi è:
 10 °Centigradi
>>> print('Hello World')
Hello World
>>> print('Hello \n World ')
Hello
 World
>>> print('La temperatura di oggi è:\n 10 °Centigradi')
La temperatura di oggi è:
 10 °Centigradi
>>>
```

Operatore di formato per stringhe (1)

Sintassi:

stringaContenteInformazioniDiFormato % (valore1, valore2, …, valoreN)

- **NOTA:** Se il valore a sinistra del simbolo % è una stringa, allora il simbolo % diviene operatore di formato

Esempio:

#cifre decimali da considerare

```
print("Numero di oggetti: %d Peso totale: %.3f" % (quantita, peso))
```

Stringa con informazioni di formato, ovvero contiene uno o più indicatori di formato (%d %.3f)

Prompt dei comandi - python

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> quantita = 748
>>> peso = 235.746432
>>> print("Numero di oggetti: %d Peso totale: %.3f" % (quantita, peso))
Numero di oggetti: 748 Peso totale: 235.746
>>>
```

Nella stringa con le info di formato ci sono due indicatori, quindi qui si inseriscono le due variabili a cui gli indicatotri fanno riferimento

Operatore di formato per stringhe (3)

https://docs.python.it/html/lib/typesseq-strings.html

Conversione	**Significato**
d	Numero intero decimale con segno.
i	Numero intero decimale con segno.
o	Ottale senza segno.
u	Decimale senza segno.
x	Esadecimale senza segno (minuscolo).
X	Esadecimale senza segno (maiuscolo).
e	Numero in virgola mobile, in formato esponenziale (minuscolo).
E	Numero in virgola mobile, in formato esponenziale (maiuscolo).
f	Decimale in virgola mobile.
F	Decimale in virgola mobile.
g	Lo stesso di "`e`" se l'esponente è più grande di -4 o minore della precisione, "`f`" altrimenti.
G	Lo stesso di "`E`" se l'esponente è più grande di -4 o minore della precisione, "`F`" altrimenti.
c	Carattere singolo (accetta interi o stringhe di singoli caratteri).
r	Stringa (converte ogni oggetto Python usando `repr()`).
s	Stringa (converte ogni oggetto Python usando `str()`).
%	Nessun argomento viene convertito, riporta un carattere "`%`" nel risultato.

Operatore di formato per stringhe (4)

- Oltre alla possibilità di specificare non solo il formato e il numero di cifre decimali da considerare, è anche possibile specificare la larghezza del campo:

peso = 235.746432

print("Numero di oggetti: %d Peso totale: %10.3f" % (quantita, peso))

peso ->

			2	3	5	.	7	4	6

- In questo caso vengono usati 10 caratteri in totale, i primi 3 sono spazi vuoti

Prompt dei comandi - python

```
>>> print("Numero di oggetti: %d Peso totale: %3.3f" % (quantita, peso))
Numero di oggetti: 748 Peso totale: 235.746
>>> print("Numero di oggetti: %d Peso totale: %30.3f" % (quantita, peso))
Numero di oggetti: 748 Peso totale:                        235.746
>>>
```

Operatore di formato per stringhe (5)

Alcuni esempi…..

Indicatore	Esempio di visualizzazione	Descrizione
"%d"	2 4	Numeri interi
"%5d"	2 4	Aggiunti spazi a sx in modo che ampiezza totale = 5
"%05d"	0 0 0 2 4	Aggiunti 0 a sx in modo che ampiezza totale = 5
"%f"	1 . 2 4 8 1 6	Numeri in virgola mobile
"%.2f"	1 . 2 4	Visualizza due cifre dopo il punto decimale
"%7.2f"	1 . 2 4	Aggiunti spazi a sx in modo che ampiezza totale = 7
"%s"	H e l l o	Stringhe
"%-9s"	H e l l o	Ampiezza tot. 9, stringa allineata a destra
"%-9s"	H e l l o	Ampiezza tot. 9, stringa allineata a sinistra
"%d%%"	2 4 %	Per visualizzare un segno di percentuale si usa %%
"%+6d"	+ 2 4	Usando il segno +, i numeri pos. Si visualizzano con +

Stampare stringhe e numeri (1)

- Quando si vuole stampare più di una stringa alla volta si dice che si effettua una operazione di concatenazione
- L'operatore di concatenazione in Python è '+'
- Se si vogliono stampare due stringhe:
  - str1= 'Hello '
  - str2 ='World!'
  - print(str1+str2)
- Se si vogliono stampare due numeri:
  - a=2
  - b=19
  - print(a+b) # in questo caso il + è
    # l'operatore di addizione!!
- Cosa succede se si vogliono stampare insieme stringhe e numeri?

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> str1='Hello '
>>> str2='World!'
>>> print(str1+str2)
Hello World!
>>> a=2
>>> b=19
>>> print(a+b)
21
>>>
```

Conversione tra numeri e stringhe (1)

- Si è visto che a volte si rende necessario convertire in stringa un valore numerico
- Non è possibile infatti concatenare una stringa con numero
- La funzione str() converte in stringa un numero intero o un numero in virgola mobile (il concetto di funzione verrà approfondito più avanti):

  str(1432) converte il valore numerico 1432 nella stringa '1432'
- Quindi è possibile scrivere la seguente riga di codice, senza che Python dia errore:

```
print("Anno: "+str(1432))
```

- La funzione str() può essere usata anche per convertire in stringa un valore in virgola mobile:

```
print("Anno: "+str(14.032))
```

Conversione tra numeri e stringhe (2)

- Viceversa, per trasformare i numeri interi e i numeri in virgola mobile in stringhe, si usano rispettivamente le funzioni int() e float()

id = int(‘1734’)

price = float(’17.34’)

- Questa conversione è particolarmente utile quando le stringhe vengono fornite dall’utente del programma come dati di ingresso e devono poi essere convertite in numeri

value = float(’17x56’)

- Genera invece un errore perché la lettera x non puo’ far parte di un letterale in virgola mobile
- Eventuali spazi presenti vengono ignorati:

value = int(’   1756’) # assegna il valore intero 1756 alla variabile value

Conversione tra numeri e stringhe (3)

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (A
MD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> id = int('1734')
>>> print(id)
1734
>>> price=("17.32")
>>> print(price)
17.32
>>> value = float("14x44")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '14x44'
>>> value = int("    3333")
>>> print(value)
3333
>>>
```

Concatenazione stringhe e numeri

Tipi in phyton:

- Numeri
- Stringhe
- Etc.

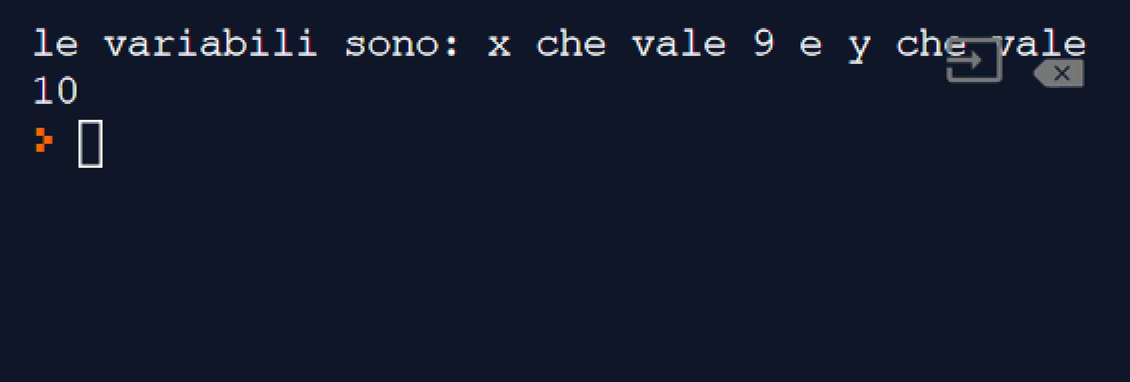
Per concatenare stringhe e numeri è necessario trasformare i numeri in stringhe perché i tipi devo essere uguali

Si usa a funzione str():

- var ="le variabili sono: "+str(x)+" e "+str(y)

main.py saved

```
 9   x = 9
10   y = 10
11   var ="le variabili sono: x che vale "+str
     (x)+" e y che vale "+str(y)
12   print(var)
```



Le stringhe sono sequenze di caratteri UNICODE… (1)

- Le tabelle dei codici Unicode sono disponibili sul sito http://www.unicode.org/charts.
- I primi caratteri di Unicode sono esattamente gli stessi della codifica ASCII, in modo da mantenere la compatibilità con il sistema preesistente
- In python le stringhe sono sequenze di caratteri UNICODE
- Cosa significa? Come possiamo verifcare?
- Esistono due funzioni: chr() e ord() che eseguono rispettivamente la conversione da numero a carattere e da carattere a numero basandosi esattamente sulla tabella UNICODE
- Facciamo degli esempi..
- Cosa ci si aspetta con i seguenti comandi?

chr(65)

ord(‘A’)

Dec	Sym	Dec	Char	Dec	Char	Dec	Char
0	NUL	32		64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g

Le stringhe sono sequenze di caratteri UNICODE… (2)

Prompt dei comandi - python

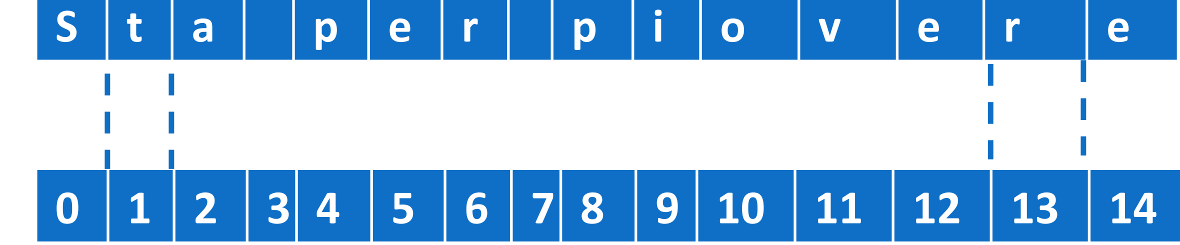
```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50
6 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for mor
n.
>>> chr(65)
'A'
>>> ord('A')
65
>>> ord('a')
97
>>> chr(64)
'@'
>>>
```

Dec	Sym	Dec	Char	Dec	Char	Dec	Char
0	NUL	32		64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g

Accesso al singolo carattere (1)

- Le Stringhe sono caratteri UNICODE
- E’ possibile accedere ai singoli caratteri contenuti in una stringa mediante la loro posizione al suo interno: tale posizione viene detta *indice* del carattere
- Il primo carattere ha indice 0, il secondo ha indice 1 e così via
- Si accede al singolo carattere con una speciale notazione a indice, racchiudendo la sua posizione tra parentesi quadre
- Se ad esempio si definisce:

stringa = ‘Sta per piovere’



Allora, gli enunciati seguenti stampano

rispettivamente la prima e l’ultima lettera della stringa:

```
print(stringa[0])

print(stringa[14])
```

Accesso al singolo carattere (2)

Prompt dei comandi - python

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.191
6 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more informatio
n.
>>> stringa='Sta per piovere'
>>> print(stringa[0])
S
>>> print(stringa[14])
e
>>> print(stringa[15])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> pos = len(stringa) -1
>>> print(pos)
14
>>>
```

Se si tenta si tenta di stampare il quindicesimo carattere (che NON costituisce la stringa), Python da’ errore ‘index out of range’

Per determinare la posizione dell’ultimo indice si usa la funzione len()

Stringhe e tipi di dati composti

- Fino ad ora sono stati usati i seguenti tipi di dati:
  - Int, float, string
- Le stringhe sono tipi di dati diversi rispetto a int e float, sono n tipo di dato composto, ovvero formato da una serie di caratteri
- Le stringhe sono sequenze di caratteri
- I tipi di dati che sono fatti di elementi più piccoli sono detti tipi di dati composti
- A seconda di ciò che stiamo facendo possiamo avere la necessità di trattare un tipo composto come fosse una singola entità o possiamo voler agire sulle sue singole parti
- Questa duplice possibilità e molto utile
- L’operatore porzione seleziona dei caratteri da una stringa:

```
stringa = "Firenze"

print(stringa[1])
```

Indice

```
stringa = "Firenze"
print(stringa[1])
```

```
main.py    saved
1   stringa = "Firenze"
2   print(stringa[1])
```

```
i
>
```

- L'espressione stringa[1] seleziona il carattere numero 1 dalla stringa
- Quando stampiamo tale carattere abbiamo però una sorpresa: si visualizza la 'i', questo perché gli indici in Python partono da 0
- Per selezionare il primo carattere è necessario quindi scrivere:

```
stringa = "Firenze"
print(stringa[0])
```

```
main.py    saved
1   stringa = "Firenze"
2   print(stringa[0])
```

```
F
>
```

Lunghezza

- La funzione len ritorna il numero di caratteri di una stringa:

```
stringa = "Firenze"
print(len(stringa))
```

- Per ottenere l'ultimo carattere di una stringa potresti essere tentato di fare qualcosa di simile a:

```
stringa = "Firenze"
print(stringa[len(stringa)])    # ERRORE!!
```

- È necessario tenere presente però che **gli indici iniziano da 0**:

```
stringa = "Firenze"
print(stringa[len(stringa)-1])
```

main.py saved

```
1    stringa = "Firenze"
2    print(stringa[len(stringa)-1])
```

e

Stringhe: leggere un carattere per volta (1)

- Molti tipi di elaborazione comportano un'azione su una stringa un carattere per volta
- Spesso queste elaborazioni iniziano dal primo carattere, selezionano un carattere per volta e continuano fino al completamento della stringa
- Questo tipo di elaborazione è definita elaborazione trasversale o attraversamento, in quanto attraversa la stringa dall'inizio alla fine
- Un modo per implementare un'elaborazione trasversale è quello di usare un ciclo while

Stringhe: leggere un carattere per volta - esempio

```
Frutto='Limone'
Indice = 0
while Indice < len(Frutto):
        Lettera = Frutto[Indice]
        print(Lettera)
        Indice = Indice + 1
```

main.py saved

```
1  Frutto='Limone'
2
3  Indice = 0
4  while Indice < len(Frutto):
5    Lettera = Frutto[Indice]
6    print(Lettera)
7    Indice = Indice + 1
8
```

```
L
i
m
o
n
e
```

Esercizio

Esercizio: scrivere una funzione che prende una stringa come argomento e la stampa un carattere per riga partendo dall'ultimo carattere.

Stringhe: leggere un carattere per volta (2)

- Usare un indice per attraversare un insieme di valori è un'operazione così comune che Python fornisce una sintassi ancora più semplice: **il ciclo for**

```
Frutto='Limone'
for Indice in Frutto:
    print(Indice)
```

```
main.py                saved
1  Frutto='Limone'
2  for Indice in Frutto:
3    print(Indice)
4
5
```

```
L
i
m
o
n
e
>
```

- Ad ogni ciclo, Indice assume il valore del carattere successivo della stringa Frutto, così Frutto viene attraversata completamente finché non rimangono più caratteri da analizzare

Porzioni di stringa (slicing) (1)

- Un segmento di stringa è chiamato porzione (o slice). La selezione di una porzione è simile alla selezione di un carattere (ed è detta slicing):

```
stringa = 'Sta per piovere'
print(stringa[0:3])
print(stringa[4:7])
print(stringa[8:15])
```

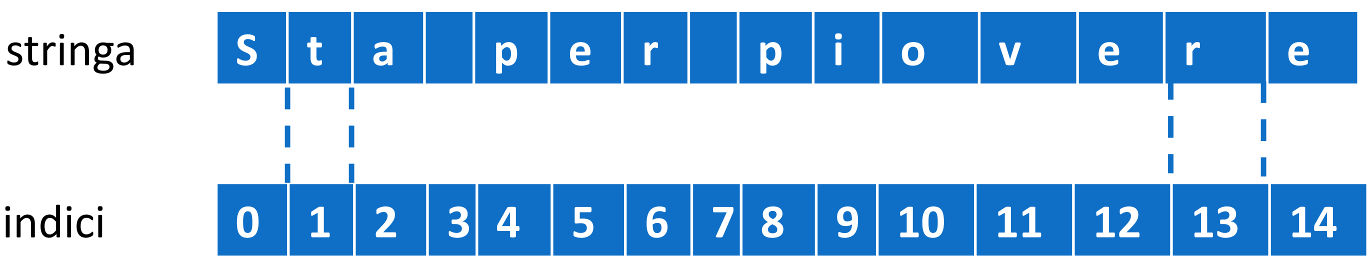
main.py saved

```
1    stringa = 'Sta per piovere'
2    print(stringa[0:3])
3    print(stringa[4:7])
4    print(stringa[8:15])
```

```
Sta
per
piovere
>
```

- L'operatore [n:m] ritorna la stringa a partire dall' "n-esimo" carattere incluso fino all' "m-esimo" escluso. Questo comportamento non è intuitivo, e per comprenderlo è meglio immaginare i puntatori tra i caratteri, come nel diagramma seguente

Porzioni di stringa (slicing) (2)



- Se non è specificato il primo indice (prima dei due punti :) la porzione parte dall'inizio della stringa. Senza il secondo indice la porzione finisce con il termine della stringa:

```
print(stringa[:7])
print(stringa[4:])
```

```
main.py    saved
1   stringa = 'Sta per piovere'
2   print(stringa[0:3])
3   print(stringa[4:7])
4   print(stringa[8:15])
5
6   print(stringa[:7])
7   print(stringa[4:])
```

```
Sta
per
piovere
Sta per
per piovere
```

- Cosa significa la seguente?

```
print(stringa[:])
```

Confronto fra stringhe (1)

- Gli operatori di confronto operano anche sulle stringhe. Per vedere se due stringhe sono uguali:

```
frutto = 'Limone'
if frutto == 'Limone’:
    print('stai parlando di un frutto!')
```

- Altri operatori di confronto sono utili per mettere le parole in ordine alfabetico:

```
frutto = 'Limone'
frutto1 = 'Arancia'
frutto2 = 'Uva'
confronta = frutto2

if confronta < 'Limone’:
    print('la parola '+confronta+' precede "Limone"')
elif confronta == 'Limone’:
    print('stai parlando di un frutto!')
else:
    print('la parola '+confronta+' posticipa "Limone"')
```

Confronto fra stringhe (2)

- Attenzione alle maiuscole e minuscole!!! Pyhton è case sensitive:

  LIMone < Limone < limone

- Un modo pratico per aggirare il problema è quello di convertire le stringhe ad un formato standard (tutto maiuscole o tutto minuscole) prima di effettuare il confronto:

```
frutto1 = 'ARAncia'
confronta = frutto1.lower()
print(confronta)
```

main.py saved

```
1  frutto1 = 'ARAncia'
2  confronta = frutto1.lower()
3  print(confronta)
4  confronta = frutto1.upper()
5  print(confronta)
```

```
arancia
ARANCIA
```

Esercizio completo

```
frutto = 'limone'
frutto1 = 'Arancia'
frutto2 = 'Uva'
frutto3 = 'Limone'
confronta = frutto1.lower()
print(confronta)

if confronta < 'Limone’:
        print('la parola '+confronta+' precede "Limone"')
elif confronta == 'Limone’:
        print('stai parlando di un frutto!')
else:
        print('la parola '+confronta+' posticipa "Limone"')
```

Le stringhe sono immutabili

- Non è possibile cambiare un carattere di una stringa con assegnazione ad esempio si potrebbe pensare di farlo:

```
main.py   saved
1  stringa = 'Stringa iniziale'
2  stringa[0] = 'C'
3  print(stringa)
```

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    stringa[0] = 'C'
TypeError: 'str' object does not support item assignment
```

- MA Pthyon da' errore!!!
- Al massimo è possibile effettuare l'operazione di concatenazione tra stringhe:

```
main.py   saved
1  stringa1 = 'Stringa iniziale'
2  stringa2 = '... con
   concatenzazione'
3  print(stringa1+stringa2)
```

```
Stringa iniziale... con concatenzazione
```

Ricerca: Funzione Trova

```
def Trova(stringa, carattere):
	indice = 0
	while indice < len(stringa):
		if stringa[indice] == carattere:
			return indice
		indice = indice + 1
		return -1
```

- La funzione Trova, cerca in una stringa la posizione (indice) corrispondente al carattere cercato e ne restituisce l’indice.
- Se il carattere non esiste rende -1
- Come si effettua la verifica di funzionamento della funzione appena definita?

Ricerca: chiamata della funzione Trova (1)

```
stringa1 = 'Stringa iniziale'

stringa2 = '... con concatenzazione'

stringa = stringa1+stringa2

print(stringa)

def Trova(stringa, carattere):
	indice = 0
	while indice < len(stringa):
		if stringa[indice] == carattere:
			return indice
		indice = indice + 1
	return -1

if (Trova(stringa,'c') != -1):
	print('Trovato il carattere! Ha indice: '+ str(Trova(stringa,'c')))
else:
	print('Il carattere cercato non esiste nella stringa!')
```

Ricerca: chiamata della funzione Trova (2)

main.py saved

```
1   stringa1 = 'Stringa iniziale'
2   stringa2 = '... con concatenzazione'
3   stringa = stringa1+stringa2
4   print(stringa)
5
6   def Trova(stringa, carattere):
7     indice = 0
8     while indice < len(stringa):
9       if stringa[indice] == carattere:
10        return indice
11      indice = indice + 1
12    return -1
13
14  if (Trova(stringa,'c') != -1):
15    print('Trovato il carattere! Ha indice: '+ str(Trova(stringa,'c')))
16  else:
17    print('Il carattere cercato non esiste nella stringa!')
```

```
Stringa iniziale... con concatenzazione
Trovato il carattere! Ha indice: 20
>
```

Si cerca ‘c’, la funzione rende l’indice.

Nota: si usa str(…) per convertire l’indice che è di tipo int a stringa, per poi stampare…

Ricerca: chiamata della funzione Trova (3)

main.py saved

```python
stringa1 = 'Stringa iniziale'
stringa2 = '... con concatenzazione'
stringa = stringa1+stringa2
print(stringa)

def Trova(stringa, carattere):
    indice = 0
    while indice < len(stringa):
        if stringa[indice] == carattere:
            return indice
        indice = indice + 1
    return -1

if (Trova(stringa,'C') != -1):
    print('Trovato il carattere! Ha indice: '+ str(Trova(stringa,'c')))
else:
    print('Il carattere cercato non esiste nella stringa!')
```

```
Stringa iniziale... con concatenzazione
Il carattere cercato non esiste nella stringa!
>
```

Si cerca ‘C’, carattere non presente nella stringa , si usa il valore di ritorno (-1) delle funzione Trova per stampare una frase comprensibile per l’utente finale (human readable)

L'Operatore in

- La parola in è un operatore Booleano che confronta due stringhe e restituisce True se la prima stringa è una sottostringa della seconda, False in caso contrario:

Prompt dei comandi - python

```
C:\Users\disit>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 'a' in 'Stringa di prova!  '
True
>>> 's' in 'Stringa di prova!  '
False
>>> 'S' in 'Stringa di prova!  '
True
>>> ' ' in 'Stringa di prova!  '
True
>>>
```

L’Operatore in: cicli e contatori (1)

- Come contare il numero di volte in cui un carattere compare in una stringa di testo:

```
for carattere in stringa:
    if carattere == ‘c’:
        conteggio = conteggio+1
```

- La variabile conteggio deve essere inizializzata a 0 e poi incrementata ogni volta che viene trovata una ’c’.
- Al termine del ciclo la variabile conteggio contiene il risultato, ovvero il numero totale di lettere a nella stringa

L'Operatore in: Cicli e contatori (2)

```
stringa1 = 'Stringa iniziale'
stringa2 = '... con concatenazione'
stringa = stringa1+stringa2
print(stringa)
conteggio = 0
cerca_carattere = 'c'

for carattere in stringa:
        if carattere == cerca_carattere:
        conteggio = conteggio+1

if(conteggio != 0):
        print(conteggio)
else:
        print('Il carattere cercato NON compare nella stringa!')
```

L'Operatore in: Cicli e contatori (3)

main.py saved

```python
stringa1 = 'Stringa iniziale'
stringa2 = '... con concatenazione'
stringa = stringa1+stringa2
print(stringa)
conteggio = 0
cerca_carattere = 'c'

for carattere in stringa:
  if carattere == cerca_carattere:
    conteggio = conteggio+1

if(conteggio != 0):
  print(conteggio)
else:
  print('Il carattere cercato NON compare nella stringa!')
```

```
Stringa iniziale... con concatenazione
Il carattere cercato NON compare nella stringa!
>
```

Operazioni con le stringhe: metodi (1)

find() - str.find(str, beg = 0 end = len(string)), con:

- str – This specifies the string to be searched
- beg – This is the starting index, by default its 0
- end – This is the ending index, by default its equal to the lenght of the string

```
stringa = 'Stringa iniziale...'

posizione = stringa.find("a") #rende indice della prima ‘a’

posizione = stringa.find("nga") #cerca la stringa 'nga' e rende indice del primo
carattere della stringa che sta cercando, ovvero indice di 'n'

posizione = stringa.find("a",10) #cerca carattere a dalla posizione con indice 10 in
poi, trova la seconda 'a'

posizione = stringa.find("a",1,12) #cerca carattere 'a' dalla posizione con indice dal 1
al 12, trova la prima 'a'

posizione = stringa.find("a",10,12 ) #cerca carattere 'a' dalla posizione con indice dal
10 al 12, non trova nessuna 'a' e rende -1
```

L'Operatore in: esempio

- Definiamo una funzione che stampa tutte le lettere di parola1 che compaiono anche in parola2:

```
def in_entrambe (parola1, parola2):
    for lettera in parola1:
        if lettera in parola2:
        print(lettera)

in_entrambe('carote', 'patate’)
```

main.py saving...

```
1
2   def in_entrambe (parola1, parola2):
3     for lettera in parola1:
4       if lettera in parola2:
5         print(lettera)
6
7   in_entrambe('carote', 'patate')
```

```
a
t
e
```

Metodo find()

main.py saved

```
1   stringa = 'Stringa iniziale...'
2   print(stringa)
3
4   posizione = stringa.find("a") #rende indice della prima a=6
5   print(posizione)
6
7   posizione = stringa.find("nga") #cerca la stringa 'nga' e rende
    indice del primo carattare della string che sta cercando, ovvero
    indice di 'n'
8   print(posizione)
9
10  posizione = stringa.find("a",10) #cerca carattere a dalla posizione
    con indice 10 in poi, trova la seconda 'a'
11  print(posizione)
12
13  posizione = stringa.find("a",1,12) #cerca carattere 'a' dalla
    posizione con indice dal 1 al 12, trova la prima 'a'
14  print(posizione)
15
16  posizione = stringa.find("a",10,12 ) #cerca carattere 'a' dalla
    posizione con indice dal 10 al 12, non trova nessuna 'a' e rende -1
17  print(posizione)
```

```
Stringa iniziale...
6
4
13
6
-1
```

Operazioni con le stringhe: metodi (2)

len(string): rende il numero dei caratteri della strigna passata in input

```
stringa = 'Stringa iniziale...'
print(len(stringa))
```

lower(string): rende la stringa in lower case

```
stringa = 'Stringa iniziale...'
print(stringa.lower())
```

upper(): rende la stringa in upper case

```
stringa = 'Stringa iniziale...'
print(stringa.upper())
```

main.py saved

```
1  stringa = 'Stringa iniziale...'
2  print(len(stringa))
3  print(stringa.lower())
4  print(stringa.upper())
```

```
19
stringa iniziale...
STRINGA INIZIALE...
```

Operazioni con le stringhe: metodi (3)

strip(): rimuove gli spazi bianchi in cima e in fondo alla stringa in esame

```
stringa = ' Stringa iniziale... '
print(stringa.strip())
```

replace(): sostitusce una stringa (o una sotto-stringa con un’altra)

```
stringa = ' Stringa iniziale... '
stringa2 = stringa.strip()
print(stringa2.replace('iniziale...', 'finale!'))
```

main.py saved

```
1   stringa = '    Stringa iniziale...   '
2   print(stringa.strip())
3   stringa2 = stringa.strip()
4   print(stringa2.replace('iniziale...', 'finale!'))
```

```
Stringa iniziale...
Stringa finale!
```

Operazioni con le stringhe: metodi (4)

split(): divide la stringa iniziale in due sottostringhe (lista...) in base al separatore scelto

```
stringa = 'Salve, la contatteremo...'
stringa2 = stringa.split(",")
print(stringa2)
stringa3 = stringa.split("Salve, ")
print(stringa3)
print(stringa3[1])
```

main.py saved

```
1   stringa = 'Salve, la contatteremo...'
2   stringa2 = stringa.split(",")
3   print(stringa2)
4   stringa3 = stringa.split("Salve, ")
5   print(stringa3)
6   print(stringa3[1])
```

```
['Salve', ' la contatteremo...']
['', 'la contatteremo...']
la contatteremo...
```

Operazioni con le stringhe: metodi (5)

splitlines(): Split a string into a list where each line is a list item

```
x = stringa.splitlines()
print(x)
```

NOTA: con split() scegliamo noi il separatore, con splitlines(), il separatore è lo spazio

main.py saved

```
1   stringa = 'Salve, la contatteremo...'
2   stringa2 = stringa.split(",")
3   print(stringa2)
4   stringa3 = stringa.split("Salve, ")
5   print(stringa3)
6   print(stringa3[1])
7   print()
8   x = stringa.splitlines()
9   x = stringa.split()
10  print(x)
```

```
['Salve', ' la contatteremo...']
['', 'la contatteremo...']
la contatteremo...

['Salve,', 'la', 'contatteremo...']
```

Operazioni con le stringhe: metodi (6)

format()

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

main.py saved

```
1   age = 36
2   txt = "My name is John, and I am {}"
3   print(txt+ age)
4
```

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(txt+ age)
TypeError: can only concatenate str (not "int") to str
```

main.py saved

```
1   age = 36
2   txt = "My name is John, and I am {}"
3   print(txt.format(age))
```

```
My name is John, and I am 36
```

Stringhe: uso di escape

- Per Inserire caratteri ‘particolari’ in una stringa, si ricorre all’escape: si usa il backslah \ seguito dal carattere che si vuole inserire

```
txt = 'Il cestino e\' pieno di: \nfrutta. Tra cui:\n\t1 Arancia\n\t2 limoni\t...'
print(txt)
```

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace

main.py saved

```
1   txt = 'Il cestino e\' pieno di: \nfrutta. Tra cui:\n\t1
    Arancia\n\t2 limoni\t...'
2   print(txt)
```

```
Il cestino e' pieno di:
frutta. Tra cui:
	1 Arancia
	2 limoni	...
```

Fondamenti di Informatica

AA 2019/2020

Eng. Ph.D. Michela Paolucci

DISIT Lab http://www.disit.dinfo.unifi.it
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it