

Introduzione al linguaggio Python

Prof. Roberto Canonico

Corso di Reti di Calcolatori

a.a. 2023-2024



Obiettivi

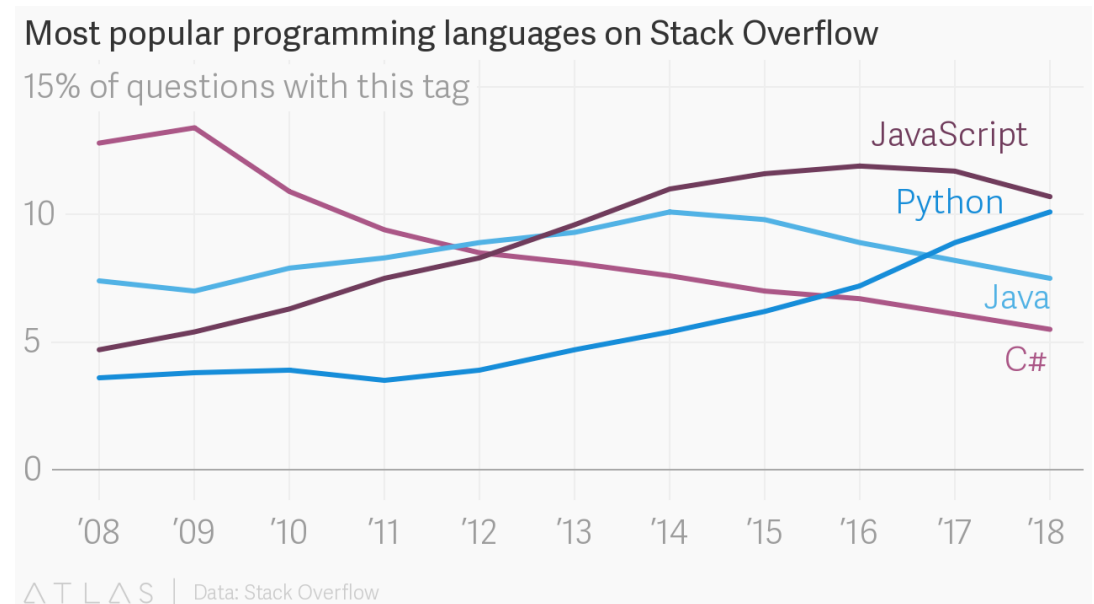


- Introdurre un linguaggio di programmazione la cui popolarità negli ultimi anni è in continua ascesa
- Mostrare come si programma in un **linguaggio interpretato**
- Spingere ad imparare nuovi linguaggi di programmazione
 - **Caveat:** saranno mostrati prevalentemente gli aspetti sintattici del linguaggio, tralasciando gli 'internals', cioè i meccanismi utilizzati dall'interprete per eseguire il codice Python ed implementare le astrazioni del linguaggio

Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	🌐 🖥️ ⚙️	100.0
2	Java	🌐 📱 🖥️	95.4
3	C	📱 🖥️ ⚙️	94.7
4	C++	📱 🖥️ ⚙️	92.4
5	JavaScript	🌐	88.1
6	C#	🌐 📱 🖥️ ⚙️	82.4

Fonte: IEEE Spectrum,
[Interactive: The Top Programming Languages 2021](https://www.theatlantic.com/charts/BJIUYlxqQ)



Fonte: <https://www.theatlantic.com/charts/BJIUYlxqQ>
su dati raccolti dallo *StackOverflow Developer Survey 2018*
<https://insights.stackoverflow.com/survey/2018/>

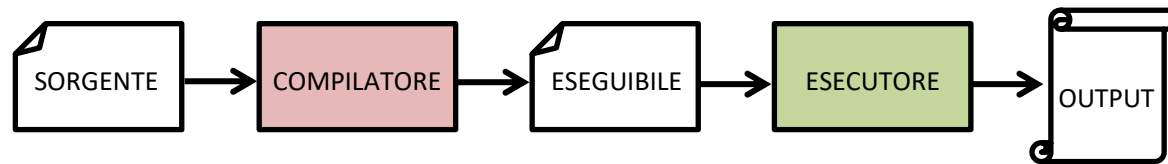
- Python è un linguaggio di programmazione:
 - di alto livello e general-purpose
 - ideato dall'informatico olandese Guido van Rossum all'inizio degli anni novanta
 - interpretato
 - interprete open-source multiplatforma
 - con tipizzazione *forte e dinamica*
 - il controllo dei tipi viene eseguito a runtime
 - supporta il paradigma object-oriented
 - ampiamente utilizzato per sviluppare applicazioni di scripting, scientifiche, distribuite, e per system testing
- Oggi se ne utilizza la versione 3
 - Alcuni programmi sviluppati nella precedente versione 2 sono ancora in uso
 - Le due versioni non sono compatibili
 - L'interprete è scaricabile dal web: <https://www.python.org/downloads/>



Differenza tra linguaggi interpretati e compilati

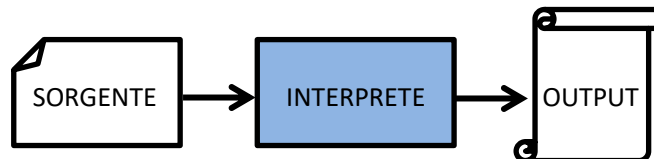
Linguaggi compilati: es. C, C++

- Il programmatore scrive il **codice sorgente** in un file di testo
- Il **compilatore** traduce il codice sorgente e produce un **codice eseguibile** (linguaggio macchina)
- L'esecutore carica il codice eseguibile nella memoria del computer e lo esegue



Linguaggi interpretati: es. Python

- Il programmatore scrive il **codice sorgente** in un file di testo
- L' **interprete** traduce (ad ogni esecuzione) il codice sorgente, trasformandolo in istruzioni del linguaggio macchina che vengono eseguite





Interprete Python: modalità interattiva

- Lanciando il programma eseguibile **python** si esegue l'interprete Python in **modalità interattiva**
- Si ottiene un prompt dal quale si possono eseguire singoli statement Python
- In modalità interattiva i comandi vanno digitati da tastiera

```
C:\Users\Roberto>python
```

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:37:02) [MSC v.1924 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> 4+7
```

```
11
```

```
>>> print("Hello world!")
```

```
Hello world!
```

```
>>> exit()
```

```
C:\Users\Roberto>
```

- L'interprete tiene memoria dei comandi eseguiti precedentemente, che possono essere richiamati al prompt usando i tasti freccia-sù e freccia-giù della tastiera



Script Python

- Un programma Python è scritto in un file sorgente testuale
- I file sorgente Python sono anche detti **script** perché possono essere eseguiti direttamente dall'interprete
- Per convenzione, gli script Python sono salvati in file con estensione **.py**
- Uno script Python può essere scritto con un normale editor di testo oppure con appositi tool o IDE
 - Thonny
 - Visual Studio Code
 - Atom
 - PyCharm
 - ...



Esecuzione di script Python

- L'esecuzione del programma si ottiene avviando l'interprete Python e fornendo come input il *path* dello script Python

- In Windows:

```
C:\Users\Roberto\Python> python prova.py
```

- In Linux, se il file script è eseguibile:

```
chmod prova.py a+x
```

e la prima riga dello script è:

```
#!/usr/bin/python
```

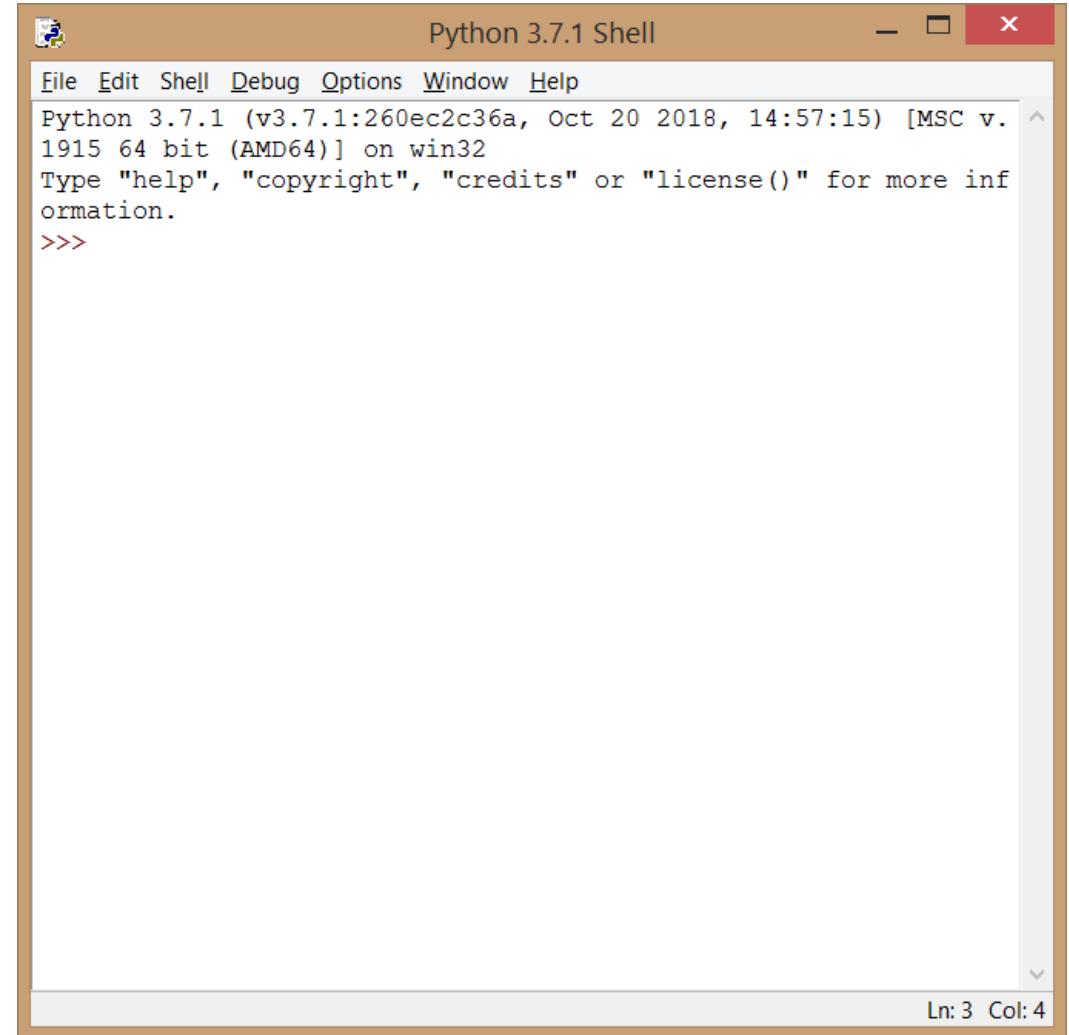
è possibile eseguire direttamente lo script dal prompt dei comandi:

```
user@computer:~$ prova.py
```

Per questa lezione useremo l'ambiente Thonny (<https://thonny.org/>)

L'ambiente IDLE

- IDLE è un ambiente minimale per lo sviluppo di programmi Python distribuito con lo stesso interprete Python
- IDLE sta per *Integrated Development and Learning Environment for Python*
 - IDLE consente sia l'esecuzione interattiva di comandi Python dal prompt dell'interprete che l'editing e l'esecuzione di uno script salvato su file



The screenshot shows a window titled "Python 3.7.1 Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the following text: "Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v. 1915 64 bit (AMD64)] on win32", "Type 'help', 'copyright', 'credits' or 'license()' for more information.", and a red prompt ">>>". The status bar at the bottom right shows "Ln: 3 Col: 4".

L'ambiente di sviluppo Thonny

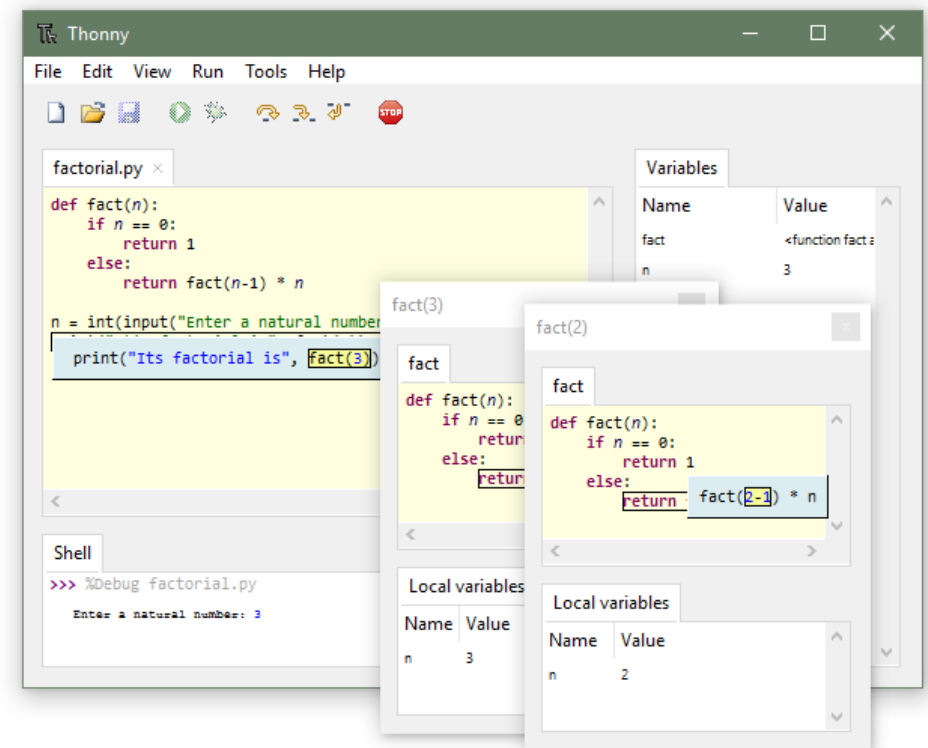


- Thonny è un semplice IDE per lo sviluppo di programmi Python
 - IDE = *Integrated Development Environment*
 - Installer di Thonny disponibile per Windows, MacOS e Linux su <https://thonny.org>
 - Sorgenti disponibili su GitHub
 - Thonny supporta Python 3 e non Python 2
 - Ha un interprete Python 3 distribuito con l'IDE ma può usare anche un altro interprete installato sulla macchina

Thonny
Python IDE for beginners



Download version **3.0.8** for
[Windows](#) • [Mac](#) • [Linux](#)



Python: aspetti lessicali

- Un programma Python è una sequenza di linee di testo
- L'unità di esecuzione è detta uno **statement**
- Una sequenza di caratteri preceduta dal carattere *hash* (#), se posta ad inizio di linea o al termine di uno statement, è considerata un commento ed ignorata dall'interprete
- Una singola linea di testo può contenere più statement separati da punto e virgola

```
a = 5; b = 8; c = 3
```

- Uno statement termina con la fine di una riga, a meno che la riga non termini con il carattere \

```
# Esempio di statement su due linee di testo  
a = 3 + 5 + 7 \  
+ 4 + 2 + 1
```

Python: indentazione



- Python non usa parentesi per delimitare blocchi di codice, a tale scopo, Python usa le regole di indentazione
 - Per indentazione si intendono gli spazi (o caratteri di tabulazione) a sinistra del primo carattere dello statement
 - In una sequenza di statement, uno statement deve avere la stessa indentazione del precedente, altrimenti si genera un errore in esecuzione

```
>>> a = 1
>>>   b = 2
File "<stdin>", line 1
b = 2
^
IndentationError: unexpected indent
```

Sul concetto di oggetto in Python

- Un oggetto è un'entità caratterizzata da un insieme di attributi e metodi
 - Si assumono noti i concetti della programmazione ad oggetti
- In Python "tutto è oggetto"
 - Anche valori di tipi come `int`, `float`, `string` sono "oggetti"
 - Esistono attributi e metodi predefiniti associati a questi "tipi"
 - La funzione predefinita `dir()` restituisce una lista di attributi e metodi di un qualunque oggetto

```
>>> dir(-5)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Oggetti: invocazione di metodi e attributi

- Chiamata di metodo: `oggetto.nome_metodo(argomenti)`

- Esempi:

```
>>> (-5).__abs__()  
5  
>>> ('Pippo').upper()  
'PIPPO'
```

- Attributi: `oggetto.nome_attributo`

- Esempi:

```
>>> (3+5j).real  
3.0  
>>> (3+5j).imag  
5.0
```

Numeri in Python



- Il linguaggio Python supporta nativamente tre diversi tipi numerici:

`int` `float` `complex`

- Se nella rappresentazione è presente il punto decimale, il numero si intende `float`, altrimenti si intende di tipo `int`
- Il tipo `int` può rappresentare numeri interi con rappresentazione illimitata (limitata cioè solo dalla capacità di memoria del calcolatore...)
- I numeri interi sono rappresentati nel codice Python:
 - in notazione decimale, di default Es. 64
 - in notazione ottale se la sequenza di cifre inizia con `0o` Es. 0o20
 - in notazione esadecimale se la sequenza di cifre inizia con `0x` o `0X` Es. 0x40
 - in notazione binaria se la sequenza di cifre inizia con `0b` Es. 0b0110
- Il valore di un numero `float` può essere rappresentato nel codice Python:
 - sia in notazione con la virgola (es. 0.123) sia in notazione esponenziale (es. 1.23E-1)
- Il tipo `complex` rappresenta numeri complessi
 - la parte reale ed il coefficiente immaginario sono numeri di tipo `real`
 - `a = 3 + 5j; print(a.real, a.imag)` produce l'output: **3.0 5.0**

Numeri in Python (2)



- La sequenza di cifre mediante le quali si rappresenta un valore numerico NON deve contenere spazi
- Il carattere . è usato come separatore tra la parte intera e la parte decimale; quando usato, il numero è trattato come float
- NON è ammesso l'uso del carattere , come separatore di cifre
 - Quindi, non è ammesso scrivere il numero «un milione» come 1,000,000
- Dalla versione 3.6 di Python, è ammesso l'uso del carattere _ come separatore di cifre
 - Può essere usato per aumentare la leggibilità
 - Il carattere _ viene ignorato nella valutazione del valore numerico, la sua posizione non ha alcun significato specifico
 - Quindi, il numero intero «un milione» può essere scritto come:
 - 1000000
 - 1_000_000
 - 1_000_0_00



Numeri in Python: segno

- Un numero (`int` o `float`) ha un segno positivo o negativo
- Un numero `float` ha anche un segno (positivo o negativo) per l'esponente
- Se omesso, il segno è considerato positivo
- Il segno `+` o `-` può essere opzionalmente separato dalla sequenza di cifre mediante spazi
- E' possibile scrivere una sequenza di segni (eventualmente separati da spazi) prima della sequenza di cifre
 - In questo caso, la scrittura viene valutata come un'espressione
 - Ad esempio:
 - `+ -1` o `+ -1` equivalgono a `+ (-1)` e corrisponde al valore `-1`
 - `- +1` o `- +1` equivalgono a `- (+1)` e corrisponde al valore `-1`
 - `--1` equivale a `- (-1)` e corrisponde al valore `+1`
 - `-+-1` equivale a `- (+ (-1))` e corrisponde al valore `+1`
 - `---1` equivale a `- (- (-1))` e corrisponde al valore `-1`

Operatori aritmetici

- In Python esistono i classici operatori aritmetici per le operazioni fondamentali:

+	somma	//	divisione intera
-	sottrazione	%	resto modulo
*	prodotto	**	elevamento a potenza
/	divisione		

- Non esistono gli operatori aritmetici unari ++ e -- del C/C++
- Se gli operandi sono interi, il risultato di un'operazione aritmetica può essere un intero oppure un `float` (nel caso di una divisione tra due interi non divisibili)
- Se almeno uno degli operandi è `float`, il risultato è un `float`
- Valgono le usuali regole di priorità tra gli operatori
- Attraverso le parentesi si modifica l'ordine di applicazione degli operatori
- L'operatore `**` effettua l'associazione a destra, cioè $2 ** 3 ** 2 = 2 ** (3 ** 2)$



Operatori aritmetici: priorità

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR

Priorità
decrescente



Operators	Meaning
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Espressioni aritmetiche



```
>>> 3/2
1.5
>>> 3//2
1
>>> 3.0/2
1.5
>>> 3.0//2
1.0
>>> 3**2
9
>>> (3 + 2) * 7
35
```



La classe `bool`

- In Python la classe `bool` ha due valori: **True** e **False**
- E' possibile associare ad una variabile un valore `bool`
- **IMPORTANTE:** In Python si considerano **False** i seguenti valori:
 - **None**
 - **False**
 - zero, di un qualunque tipo numerico: 0, 0L, 0.0, 0j
 - strutture dati vuote:
 - `''` *stringa vuota*
 - `()` *tupla vuota*
 - `[]` *lista vuota*
 - `{}` *dizionario vuoto*

```
>>> print(type(True))  
<class 'bool'>  
>>> print(type(False))  
<class 'bool'>
```



Sul concetto di variabile in Python (1)

- In linguaggi come C, C++, Java, che adottano un controllo del tipo dei dati statico (a tempo di compilazione) una variabile è un "contenitore di memoria" atto a mantenere un dato di un certo tipo
- **In Python, il modello di programmazione è significativamente diverso:**
 - il linguaggio effettua un controllo dinamico dei tipi (a tempo di esecuzione o *runtime*)
 - l'operatore di assegnazione semplicemente associa un nome ad un oggetto:
$$a = 5$$
 - ***a* è *un nome*** per l'oggetto 5



Sul concetto di variabile in Python (2)

- Lo stesso "nome" può essere associato (*binding*) in momenti successivi ad oggetti diversi, anche di tipi diversi:

```
a = 5; a = "Pippo"
```

- Questi **nomi** sono tipicamente chiamati "**variabili**" in Python
- **Un nome non può essere usato prima che sia stato associato ad un valore**

```
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

- **Un oggetto esiste fintanto che esiste un nome che fa riferimento ad esso**

Tipizzazione dinamica

- In Python non si effettua la dichiarazione esplicita del tipo di una variabile
- L'interprete determina a runtime il tipo di una variabile in base al valore assegnatole
- Il tipo associato ad una variabile può cambiare nel corso dell'esecuzione
 - Si parla pertanto di ***tipizzazione dinamica***
- La funzione predefinita **`type()`** restituisce la classe di una variabile

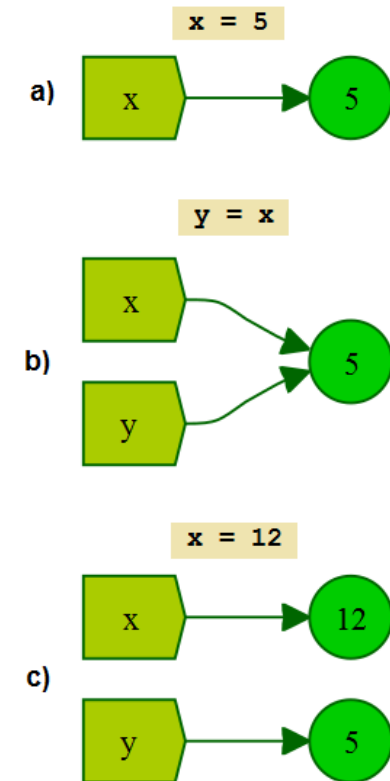
```
>>> x = "Pippo"; print(type(x))
<class 'str'>
>>> x = 123; print(type(x))
<class 'int'>
>>> x = 3.14; print(type(x))
<class 'float'>
>>> x = 3+5j; print(type(x))
<class 'complex'>
```

Statement di assegnazione

- In generale, l'operatore di assegnazione crea un'associazione (***binding***) tra un nome ed un 'oggetto'
- La funzione predefinita `id(my_var)` restituisce un identificativo associato all'area di memoria che contiene il valore associato al nome `my_var`
- Si consideri quanto illustrato nell'esempio seguente

```
>>> x = 5
>>> y = x
>>> print(id(x), id(y))
30696568 30696568
>>> x = 12; print(x)
12
>>> print(y)
5
>>> print(id(x), id(y))
30696544 30696568
```

- L'esempio mostra che, nel momento in cui un nome (nell'esempio `x`) viene assegnato ad un nuovo valore (`x = 12`), cambia l'area di memoria associata al nome
 - ***Questo è molto diverso rispetto a quanto succede in C/C++***





La funzione `print` per l'output a video

- In Python 3 `print()` è una funzione che stampa sullo standard output in sequenza gli argomenti che le sono passati separati da uno spazio e termina andando a capo
- Possono essere specificati i parametri opzionali ***end*** e ***sep***
- Con il parametro opzionale ***end*** si specifica il carattere (o i caratteri) da appendere alla fine della riga. Il valore di default di *end* è `"\n"`
- Con il parametro opzionale ***sep*** si specifica il carattere (o i caratteri) che separano i vari argomenti della print. Il valore di default di *sep* è `" "` (1 spazio)



Esempi d'uso di `print()`

```
>>> print(1, 12, 99)
```

```
1 12 99
```

```
>>> print("A",end=""); print("B")
```

```
AB
```

```
>>> print("A",end=" "); print("B")
```

```
A B
```

```
>>> print("A",end="--"); print("B")
```

```
A--B
```

```
>>> print("A", "B", sep="")
```

```
AB
```

```
>>> print("A", "B", sep="-")
```

```
A-B
```



Funzioni di conversione

- E' possibile associare il valore di un dato ad un valore "corrispondente" di un altro tipo numerico mediante apposite funzioni predefinite
 - `int()` restituisce un `int` a partire da una stringa di caratteri o un `float`
 - `float()` restituisce un `float` a partire da una stringa di caratteri o un `int`
 - `int(x, b)` restituisce un `int` a partire dalla stringa `x` interpretata come sequenza di cifre in base `b`
- Di seguito si mostrano alcuni esempi

```
>>> int('2014')
2014
>>> int (3.141592)
3
>>> float ('1.99')
1.99
>>> float (5)
5.0
```

```
>>> int('20',8)
16
>>> int('20',16)
32
>>> int('aa')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'aa'
>>> int('aa',16)
170
```

Funzioni per l'input da tastiera

- In Python 3 la funzione `input()` consente di leggere una stringa dallo standard input (tastiera)

```
x = input("Digita una stringa e premi ENTER: ")
```

- Se si desidera acquisire da tastiera un numero intero occorre convertire la stringa ritornata da `input()` con la funzione di conversione `int()`

```
x = int(input("Digita un numero e premi ENTER: "))
```

- Se l'utente digita una stringa che non rappresenta un numero, si genera un errore a runtime

```
>>> x = int(input("Digita un numero e premi ENTER: "))
Digita un numero e premi ENTER: xxx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'xxx'
>>>
```



Compound statement

- I **compound statement** (if, while, for, try, def, ...) "contengono" una sequenza (*suite* o *body*) di statement elementari
- Il body di un compound statement è formato da linee con la stessa indentazione allineate più a destra rispetto all'istruzione che "le contiene"
 - Python raccomanda di usare 4 caratteri spazio per ciascun livello di indentazione e di non usare il carattere tab

```
#valore assoluto
a = int(input("Digita un numero intero e premi ENTER: "))
if (a >= 0):
    b = a
else:
    b = -a

print("b=", b)
```



Statement `if`

- `if (cond)` esegue una sequenza di istruzioni se `cond` ha valore diverso da **False**
- Si consideri il seguente esempio:

```
>>> a=2
>>> if (a):
...     print("eseguo if")
... else:
...     print("eseguo else")
...
eseguo if
>>> if (a==True):
...     print("eseguo if")
... else:
...     print("eseguo else")
...
eseguo else
```



Statement `if` (2)

```
a = int(input("Digita un numero intero e premi ENTER: "))
# la funzione int puo' generare un errore
# se l'input da tastiera non e' corretto
# in questo caso il programma termina
print("Hai digitato",end=" ")
if (a > 0):
    print("un numero POSITIVO ( >0)")
elif (a < 0):
    print("un numero NEGATIVO ( <0)")
else:
    print("ZERO")
```

- Le parentesi intorno alla condizione non sono necessarie
- La parte **elif** e la parte **else** sono facoltative
- Si possono aggiungere un numero di arbitrario di **elif**

Operatori and, or, not

Operatore	Valore restituito
<code>a or b</code>	<code>b</code> , se <code>a</code> è False o un valore assimilato a False <code>a</code> , altrimenti
<code>a and b</code>	<code>a</code> , se <code>a</code> è False o un valore assimilato a False <code>b</code> , altrimenti
<code>not a</code>	True , se <code>a</code> è False o un valore assimilato a False False , altrimenti

- Ordine di priorità decrescente: `not`, `and`, `or`

`not a and b or c` equivale a: `((not a) and b) or c`

- `not` ha priorità minore di altri operatori non-booleani

`not a == b` equivale a: `not (a==b)`

```
>>> True or False
True
>>> True and False
False
>>> not True
False
>>> not False
True
```

```
>>> 3 or 5
3
>>> 0 or 5
5
>>> True or 5
True
>>> False or 5
5
```

```
>>> 3 and 5
5
>>> 0 and 5
0
>>> True and 5
5
>>> False and 5
False
```


Espressioni di confronto

- Un'espressione di confronto restituisce un valore di tipo `bool`
- Usa gli operatori di confronto:
`==` `<` `<=` `>=` `>` `!=` `is` `is not`
- I valori booleani `False` e `True` sono uguali agli interi 0 ed 1 rispettivamente
- Se necessario, nel calcolare un'espressione di confronto vengono eseguite conversioni di tipo `int` \rightarrow `float`

```
>>> False == 0
True
>>> False == 1
False
>>> False == 2
False
>>> True == 0
False
>>> True == 1
True
>>> True == 2
False
```

```
>>> 1 == 2 - 1
True
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

Statement **pass**

- Lo statement **pass** rappresenta un'istruzione che non produce effetto
- Lo si usa a volte negli esempi di codice o quando si scrive un codice incompleto
- Esempio d'uso di **pass**:

```
if (a > 0) :  
    pass  
b = 5
```

equivale al seguente statement C/C++

```
if (a > 0) {} /* Se (a > 0) fai niente */  
b = 5;      /* Istruzione successiva eseguita in qualsiasi caso */
```



Statement **while**

- **while** esegue un ciclo finché una condizione è vera

```
a = input("Digita qualsiasi cosa e poi ENTER (QUIT per terminare): ")  
while (a != "QUIT"):  
    print("Hai digitato: ", a)  
    a = input("Digita qualsiasi cosa e poi ENTER (QUIT per terminare): ")  
  
print("Hai digitato QUIT ed il programma termina.")
```



Statement `while` (2)

- Lo statement **`break`** nel corpo di un ciclo fa uscire dal ciclo
- Lo statement **`continue`** nel corpo di un ciclo salta all'iterazione successiva

```
conta = 0
while (True):
    a = int(input("Digita un numero e premi ENTER (0 per terminare): "))
    if (a > 0):
        print("Hai digitato un numero positivo : viene contato")
    elif (a < 0):
        print("Hai digitato un numero negativo : NON viene contato")
        continue
    else:
        print("Hai digitato 0 ed il programma termina.")
        break
    conta = conta + 1

print("Hai digitato %d numeri positivi." % conta)
```



Funzioni

- Una funzione è un sottoprogramma al quale possono essere passati dei parametri (*argomenti*) e che eventualmente restituisce un valore
- Per definire una funzione si usa lo statement **def**
- Una funzione che non prevede argomenti ha una lista di argomenti vuota ()
- Per ritornare al chiamante un oggetto **x** si usa lo statement **return x**
- Un oggetto speciale **None** è ritornato da una funzione se:
 - il flusso di esecuzione della funzione termina senza aver eseguito un'istruzione **return**
 - lo statement **return** viene eseguito senza argomenti

```
def funzione_1():  
    pass  
  
def funzione_2(x):  
    return x
```

```
ret = funzione_1()  
print("Valore di ritorno =", ret)  
  
ret = funzione_2(1)  
print("Valore di ritorno =", ret)  
ret = funzione_2("PIPPO")  
print("Valore di ritorno =", ret)
```

```
Valore di ritorno = None  
Valore di ritorno = 1  
Valore di ritorno = PIPPO
```



Funzioni: esempio

```
def max(x, y):  
    print("x =", x)  
    print("y =", y)  
    if (x >= y):  
        return x  
    else:  
        return y  
  
ret = max(1, 1)  
print("Valore di ritorno =", ret)  
ret = max(2, 1)  
print("Valore di ritorno =", ret)  
s1 = "PIPPO"  
s2 = "PIPPONE"  
s3 = "PLUTO"  
ret = max(s1, s2)  
print("Valore di ritorno =", ret)  
ret = max(s2, s3)  
print("Valore di ritorno =", ret)
```

```
x = 1  
y = 1  
Valore di ritorno = 1  
x = 2  
y = 1  
Valore di ritorno = 2  
x = PIPPO  
y = PIPPONE  
Valore di ritorno = PIPPONE  
x = PIPPONE  
y = PLUTO  
Valore di ritorno = PLUTO
```

Funzioni: argomenti, variabili locali e globali

- Un nome di variabile usato in una funzione è trattato come una variabile locale
 - Il valore assegnato ad una variabile locale non si riflette sul valore assegnato ad una variabile con lo stesso nome definita all'esterno della funzione
- Gli argomenti di una funzione sono trattati alla stregua di variabili locali

```
def f1(x):  
    a = 3      # variabile locale  
    x += 1     # argomento  
    print("In f1:")  
    print("a =", a, "x =", x)  
  
a = 99  
x = 1  
print("Prima di eseguire f1:")  
print("a =", a, "x =", x)  
f1(x)  
print("Dopo aver eseguito f1:")  
print("a =", a, "x =", x)
```

```
Prima di eseguire f1:  
a = 99 x = 1  
In f1:  
a = 3 x = 2  
Dopo aver eseguito f1:  
a = 99 x = 1
```

Funzioni: argomenti, variabili locali e globali

- Per modificare in una funzione il valore associato ad un nome di variabile definito esternamente, si usa lo statement `global`

```
def f2(x):  
    global a  
    a = 3      # variabile globale  
    x += 1     # argomento  
    print("In f1:")  
    print("a =", a, "x =", x)  
  
a = 99  
x = 1  
print("Prima di eseguire f2:")  
print("a =", a, "x =", x)  
f2(x)  
print("Dopo aver eseguito f2:")  
print("a =", a, "x =", x)
```

```
Prima di eseguire f2:  
a = 99 x = 1  
In f2:  
a = 3 x = 2  
Dopo aver eseguito f2:  
a = 3 x = 1
```


Funzioni: argomenti con valori di default

- Per ogni argomento di una funzione si può specificare un valore di default
 - Quando ad un argomento è assegnato un valore di default, nella chiamata della funzione, un parametro può essere specificato con la notazione `nome = valore`
 - In tal modo i parametri possono essere passati in ordine diverso da quello con cui sono elencati nella definizione della funzione

```
def f(x, y = 5, z = 3):  
    print("In f:", end=" ")  
    print("x =", x, "y =", y, "z =", z)
```

```
f(0, 1, 2)  
f(0)  
f(8, 1)  
f(7, z = 2)  
f(6, y = 2)  
f(0, z = 2, y = 1)
```

```
In f: x = 0 y = 1 z = 2  
In f: x = 0 y = 5 z = 3  
In f: x = 8 y = 1 z = 3  
In f: x = 7 y = 5 z = 2  
In f: x = 6 y = 2 z = 3  
In f: x = 0 y = 1 z = 2
```



Strutture dati container in Python

- Uno degli aspetti peculiari del linguaggio Python è la ricchezza di strutture dati definite dal linguaggio
- In particolare, il linguaggio offre al programmatore diverse strutture dati di tipo container atte a contenere oggetti di vario tipo
 - liste, tuple, dizionari, stringhe, set
 - il modulo `collections` definisce ulteriori tipi container
- Le varie strutture dati differiscono in vari aspetti:
 - il modo con il quale si può accedere agli oggetti contenuti
 - la possibilità di iterare sugli elementi contenuti nella struttura dati
 - l'eventuale ordinamento definito tra gli elementi contenuti
 - la possibilità o meno di modificare gli elementi presenti nella struttura dati una volta che sia stata "costruita"
 - si parla di strutture dati **mutabili** o **immutabili**
 - liste, dizionari, set sono mutabili mentre
 - stringhe e tuple sono immutabili



Stringhe di caratteri

- In Python una stringa è una sequenza ordinata di caratteri
- E' possibile definire una stringa di caratteri usando come delimitatori:
 - apici 'esempio'
 - doppi apici "esempio"
- Tre apici ''' e tre doppi apici """ sono usati per stringhe su più linee
- Caratteri speciali possono essere inseriti in una stringa mediante sequenze di escape
- La concatenazione di stringhe si effettua con l'operatore +
- L'operatore * consente la concatenazione di una stringa a se stessa per n volte
- E' possibile identificare mediante un indice intero i singoli caratteri di una stringa s
 - s[0] indica il primo, s[1] il secondo, ecc. ...
- E' possibile anche usare valori negativi per l'indice:
 - s[-1] indica l'ultimo carattere, s[-2] il penultimo, ecc. ...

In Python le stringhe sono immutabili

`s = "pippo"; s[0] = 'P'` produce un errore:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**TypeError: 'str' object does not support
item assignment**

Non è possibile modificare un carattere di una stringa in maniera diretta



Stringhe di caratteri: esempi

```
>>> print('riga 1\nriga 2')
riga 1
riga 2
>>> nome = "Roberto"
>>> print("Ciao " + nome)
Ciao Roberto
>>> print("Ciao"*3)
CiaoCiaoCiao
```

```
>>> s = "ABCDEFGHJIJ"
>>> print(s[0],s[1],sep="")
AB
>>> print(s[9])
J
>>> print(s[10])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(s[-1],s[-2],sep="")
JI
>>> print(s[-10])
A
```

```
>>> s1 = "ABC"; s1[0] = 'X'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
item assignment
```



Stringhe: *slicing*

- E' possibile costruire una nuova stringa prendendo dei "pezzi" di una stringa formati da caratteri consecutivi
- Si usa la notazione con indici, specificando l'indice *i* del primo carattere (compreso) e quello *j* dell'ultimo (escluso) separati dal carattere : (due punti)

`s[i:j]`

- Se l'estremo sinistro *i* è omissso: `s[:j]`
la sottostringa inizia dal primo carattere di *s*
- Se l'estremo destro *j* è omissso: `s[i:]`
la sottostringa termina con l'ultimo carattere di *s*

```
>>> s = "Qui, Quo, Qua"
>>> s[0:3]
'Qui'
>>> s[:3]
'Qui'
>>> s[3:3]
''
>>> s[3:4]
','
```

```
>>> s = "Qui, Quo, Qua"
>>> s[5:8]
'Quo'
>>> s[-3: -1]
'Qu'
>>> s[-3:]
'Qua'
>>>
```

Stringhe: *slicing* (2)

- Nella forma più generale, l'operazione di *slicing* è determinata da tre numeri interi indicati tra parentesi quadre e separati dal simbolo :

`s[i:j:step]`

- Il terzo numero (**step**) può essere positivo o negativo
- La stringa prodotta come risultato dello *slicing* è costituita dai caratteri di **s**, a partire da quello di indice **i**, individuati dal valore di indice che si ottiene a partire dal valore precedente mediante un incremento di **step**, fino ad arrivare all'elemento di indice **j** (escluso)
- Es.: `s = "Ciao"; s[0:3:2] → "Ca"`
- Se **step** è positivo, per non avere una stringa vuota, dovrà essere $i < j$
- Se **step** è negativo, per non avere una stringa vuota, dovrà essere $i > j$
- Es.: `s = "Ciao"; s[3:0:-1] → "oai"`



Combinazione di stringhe e variabili

- E' possibile combinare una stringa di testo fatta da una parte costante specificata tra apici ed una parte definita attraverso i nomi di variabili
- Come si può fare in C per la funzione printf, ciò si realizza inserendo nella stringa costante il simbolo % come *placeholder*, seguito da dei codici che servono a definire la formattazione dell'output prodotto
- I valori da sostituire ai placeholder sono indicati a destra della stringa dopo %
- Esempio con un solo valore:

```
>>> import math
>>> print("The value of PI is approximately %5.3 f." % math.pi)
The value of PI is approximately 3.142.
```

- Esempio con due valori:

```
>>> x =18; y=15
>>> print("x=%d y=%d" % (x,y))
x=18 y=15
```



Esempio: verifica se una stringa è palindroma

```
>>> s = "INGEGNI"
>>> s[::-1]
'INGEGNI'
>>> is_palindrome = s == s[::-1]
>>> is_palindrome
True

>>> s = "PIPPO"
>>> s[::-1]
'OPPIP'
>>> is_palindrome = s == s[::-1]
>>> is_palindrome
False
```


Stringhe: metodi (1)

Una stringa è un oggetto sul quale si possono invocare dei metodi predefiniti

- `s.find(sub_str)` restituisce l'indice della posizione della prima occorrenza della sottostringa `sub_str` nella stringa `s`
- `s.find(sub_str, start)` restituisce l'indice della posizione della prima occorrenza della sottostringa `sub_str` nella stringa `s`, cominciando la ricerca dal carattere di indice `start`
- `find()` restituisce -1 se la sottostringa `sub_str` non è trovata
- `s.split(sep)` restituisce una lista di sottostringhe di `s` separate nella stringa originaria dal separatore `sep`. Se `sep` non esiste in `s`, `split()` restituisce una lista con il solo elemento `s`

```
>>> s = "Qui, Quo, Qua"
>>> s.find(", ")
3
>>> s.find(", ", 0)
3
>>> s.find(", ", 3)
3
>>> s.find(", ", 4)
8
>>> s.find("X")
-1
```

```
>>> s = "Qui, Quo, Qua"
>>> s.split(", ")
['Qui', ' Quo', ' Qua']
>>> s.split(", ")
['Qui', 'Quo', 'Qua']
>>> s.split("X")
['Qui, Quo, Qua']
```

Stringhe: metodi (2)

- `s.strip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità sinistra e destra
- `s.rstrip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità destra
- `s.lstrip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità sinistra
- `s.startswith(x)` restituisce `True` se la stringa `s` inizia con la sottostringa `x`, `False` altrimenti
- `s.endswith(x)` restituisce `True` se la stringa `s` termina con la sottostringa `x`, `False` altrimenti
- `s.upper(x)` restituisce una stringa in cui i caratteri di `s` che sono lettere minuscole sono convertiti in maiuscole, gli altri caratteri sono lasciati inalterati
- `s.lower(x)` restituisce una stringa in cui i caratteri di `s` che sono lettere maiuscole sono convertiti in minuscole, gli altri caratteri sono lasciati inalterati

```
>>> s = " Qui,\tQuo,\tQua\t\n"
>>> print(s)
    Qui,      Quo,      Qua
>>> s.strip()
'Qui,\tQuo,\tQua'
```

```
>>> s = " Qui,\tQuo,\tQua\t\n"
>>> s.rstrip()
' Qui,\tQuo,\tQua'
>>> s.lstrip()
'Qui,\tQuo,\tQua\t\n'
```

```
>>> s = "Qui, Quo, Qua"
>>> s.startswith('Qui')
True
>>> s.endswith('Qua')
True
>>> s.upper()
'QUI, QUO, QUA'
>>> s.lower()
'qui, quo, qua'
```



Liste

- Le liste sono strutture dati ordinate che possono contenere oggetti di tipi differenti
- Il seguente statement crea una lista associata al nome **l**:

```
l = [1, 2, "Pippo"]
```

- La scrittura **[]** indica una lista vuota
- Sulle liste si possono applicare gli operatori **+** e ***** come per le stringhe
- La funzione **len(l)** restituisce il numero di elementi di una lista **l**
- Gli elementi di una lista possono essere individuati e modificati tramite un indice come per gli array in C/C++
- La scrittura **l[i:j]** indica la lista costituita dagli elementi di **l** compresi tra quello di indice **i** (compreso) a quello di indice **j** (escluso)
 - Se **i** non è specificato, si intende **i=0**
 - Se **j** non è specificato, si intende **j=len(l)**

Liste: esempi



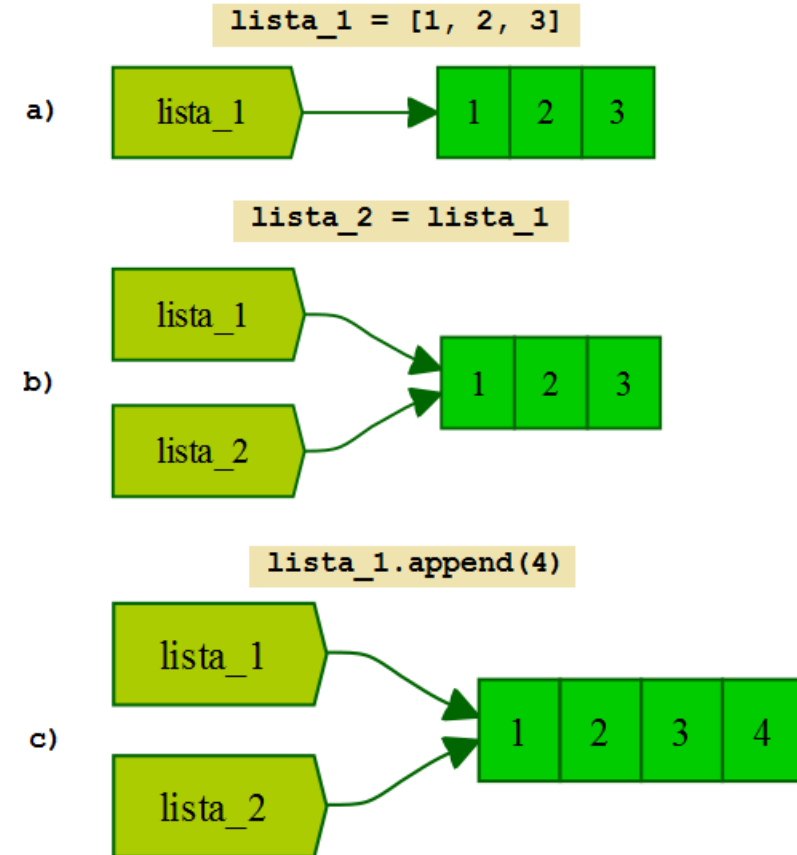
```
>>> l = [1, 2]
>>> m = ["Pippo", 3, 4]
>>> l + m
[1, 2, 'Pippo', 3, 4]
>>> l*4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> m[0]
Pippo
>>> m[0] = 99; print(m)
[99, 3, 4]
>>> len(m)
3
```

```
>>> n = l + m[1:]; print(n)
[1, 2, 3, 4]
>>> n[1:2]
[2]
>>> n[1:3]
[2, 3]
>>> n[1:4]
[2, 3, 4]
>>> n[1:5]
[2, 3, 4]
>>> n[5:6]
[]
```

Le liste sono oggetti mutabili

```
>>> lista_1 = [1, 2, 3]
>>> lista_2 = lista_1
>>> print(id(lista_1), id(lista_2))
32782280 32782280
>>> lista_1.append(4)
>>> print(lista_1)
[1, 2, 3, 4]
>>> print(lista_2)
[1, 2, 3, 4]
>>> print(id(lista_1), id(lista_2))
32782280 32782280
```

- L'esempio mostra che l'invocazione del metodo **append()** su un oggetto di tipo lista (nell'esempio **lista_1**) non cambia l'area di memoria associata al nome
 - Di conseguenza, la modifica operata su **lista_1** si riflette anche su **lista_2**



Liste: metodi



Una lista è un oggetto sul quale si possono invocare dei metodi predefiniti

- `l.append(obj)` aggiunge `obj` "in coda" alla lista `l`
- `l.extend(l1)` estende `l` aggiungendo "in coda" gli elementi della lista `l1`
- `l.insert(index, obj)` aggiunge `obj` in `l` prima della posizione indicata da `index`
- `l.pop(index)` rimuove da `l` l'oggetto nella posizione `index` e lo restituisce
- `l.remove(obj)` rimuove la prima occorrenza di `obj` nella lista `l`
- `l.reverse()` dispone gli elementi della lista `l` in ordine inverso
- `l.sort()` dispone gli elementi della lista `l` in ordine crescente
- `l.index(obj)` restituisce l'indice della prima occorrenza di `obj` nella lista `l`
- `l.count(obj)` restituisce il numero di occorrenze di `obj` nella lista `l`

Liste: metodi - esempi



```
>>> l1 = [6, 7, 8, 9]
>>> l2 = [1, 2, 3, 4, 5]
>>> l1.append(10); print(l1)
[6, 7, 8, 9, 10]
>>> l1.extend(l2); print(l1)
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(0, "START"); print(l1)
['START', 6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(6, "x"); print(l1)
['START', 6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.pop(0); print(l1)
'START'
[6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.remove("x"); print(l1)
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
```

```
...
>>> l1.reverse(); print(l1)
[5, 4, 3, 2, 1, 10, 9, 8, 7, 6]
>>> l1.sort(); print(l1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l1.pop(-1)
10
>>> l1.index(1)
0
>>> l1.index(9)
8
>>> l1.index(10)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 10 is not in list
```



Liste: altre funzioni predefinite

- `len(x)` restituisce il numero di elementi di una lista x (si può applicare anche a tuple e stringhe)
- `max(l)` restituisce l'elemento massimo di una lista l
- `sorted(l)` restituisce una lista con gli elementi della lista l ordinati in ordine crescente

```
>>> len([0, 1, 2, 3])
4
>>> max([1.0, 2.5, -2.3, 1.3])
2.5
>>> sorted([3, 2, 1])
[1, 2, 3]
```




La funzione range()

- `range(n)` restituisce una lista(*) formata dai numeri interi compresi tra 0 ed n-1
- `range(m, n, step)` restituisce una lista(*) formata dai numeri interi compresi tra m (incluso) ed n (escluso), con un incremento di step

(*) In Python 2, range() restituiva effettivamente una lista
In Python 3, range() restituisce un oggetto della classe iterabile **range**

```
>>> a = range(5)
>>> print(a)
range(0, 5)
>>> print(type(range(5)))
<class 'range'>
```



Statement **for**

- Lo statement **for** esegue una sequenza di istruzioni (corpo del ciclo) per tutti gli elementi di una struttura dati **iterabile**
- La sintassi è:

```
for iterating_var in sequence:  
    corpo_del_ciclo
```

- Esempio:

```
nomi = ['Antonio', 'Mario', 'Giuseppe', 'Francesco']  
for nome in nomi:  
    print('Nome :', nome)
```

- **break** nel corpo del ciclo fa uscire dal ciclo
- **continue** nel corpo del ciclo salta all'iterazione successiva

Statement `for` con iterazione su elementi di range



- E' possibile iterare sugli elementi (numeri interi) di un oggetto range come nell'esempio seguente:

```
nomi = ['Antonio', 'Mario', 'Giuseppe', 'Francesco']  
for index in range(len(nomi)):  
    print('Nome :', nomi[index])
```



Tuple

- Le tuple sono strutture dati ordinate che possono contenere oggetti di tipi differenti
- Il seguente statement crea una tupla associata al nome `t`:

`t = (1, 2, "Pippo")`

- `()` indica una tupla vuota, `(a,)` indica una tupla con il solo elemento `a`
- Sulle tuple si possono applicare gli operatori `+` e `*` come per le stringhe
- La funzione `len(t)` restituisce il numero di elementi di una tupla `t`
- Gli elementi di una tupla possono essere individuati (ma non modificati) tramite un indice come per gli array in C/C++
- La scrittura `t[i:j]` indica la tupla costituita dagli elementi di `t` compresi tra quello di indice `i` (compreso) a quello di indice `j` (escluso)
 - Se `i` non è specificato, si intende `i=0`
 - Se `j` non è specificato, si intende `j=len(t)`

Tuple: esempi



```
>>> t1 = (1, 2)
>>> t2 = ("Pippo", 3, 4)
>>> t = t1 + t2; print(t)
(1, 2, 'Pippo', 3, 4)
>>> len(t)
5
>>> t[0]
1
>>> t[2]
'Pippo'
```

```
>>> t[0] = 99
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
>>> t[1:3]
(2, 'Pippo')
>>> t[1:1]
()
>>> t[5:5]
()
```

Creazione di liste mediante `for`

- Lo statement `for` può essere usato per creare liste i cui elementi sono generati da un'espressione valutata iterativamente
- Esempi

```
>>> l1 = [ (2*x + 1) for x in range(0,10) ]
>>> print(l1)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> l2 = [ (x, x **2) for x in range(0,11) ]
>>> print(l2)
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64),
(9, 81), (10, 100)]
>>> l3 = [ chr(i) for i in range(ord('a'),ord('z')+1) ]
>>> print(l3)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```



Dizionari

- Un dizionario è una struttura dati contenitore di coppie (chiave, valore) in cui ciascun valore è identificato univocamente da una "chiave"
- Il seguente statement crea un dizionario associato al nome **d**:
d = {chiave1: val1, chiave2: val2, chiave3: val3}
- L'accesso agli elementi di un dizionario avviene fornendo il valore della chiave:
d[key]
- Se il valore di chiave non esiste nel dizionario, si produce un errore
- I dizionari sono strutture dati mutabili, il valore della chiave è immutabile
- L'operatore **in** restituisce **True** se una chiave è presente in un dizionario



Dizionari: esempi

```
>>> d = { "NA": "Napoli",      "AV": "Avellino",
          "BN": "Benevento", "CE": "Caserta",
          "SA": "Salerno"}

>>> d["BN"]
'Benevento'

>>> d["MI"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'MI'

>>> d[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

```
>>> d["NA"] = "Naples"
>>> d["NA"]
'Naples'

>>> "NA" in d
True

>>> "AV" in d
True

>>> "MI" in d
False
```




Dizionari: metodi

Una lista un oggetto sul quale si possono invocare dei metodi predefiniti

- `d.clear()` elimina tutti gli elementi del dizionario `d`
- `d.copy()` restituisce una copia del dizionario `d`
- `d.has_key(key)` restituisce `True` se in `d` esiste la chiave `key`
- `d.items()` restituisce una lista(*) con le tuple (chiave, valore) in `d`
- `d.keys()` restituisce una lista(*) con le chiavi in `d`
- `d.values()` restituisce una lista(*) con i valori in `d`
- `d.update(d2)` aggiunge al contenuto di `d` quello di `d2`
- `d.get(key, val)` restituisce il valore associato a `key`, altrimenti `val`
- `d.get(key)` restituisce il valore associato a `key`, altrimenti `None`

(*) In Python 2, questi metodi restituivano effettivamente delle liste
In Python 3, questi metodi restituiscono oggetti delle classi iterabili i **dict_items**, **dict_keys**, e **dict_values**
Per ottenere una lista da questi oggetti, si può usare la funzione predefinita **list()**

Dizionari: esempi d'uso dei metodi



```
>>> d = { "NA": "Napoli", "AV": "Avellino", "BN": "Benevento", "CE": "Caserta", "SA": "Salerno" }
>>> d.has_key("NA")
True
>>> d.has_key("MI")
False
>>> d.items()
dict_items([('NA', 'Napoli'), ('AV', 'Avellino'), ('BN', 'Benevento'), ('CE', 'Caserta'), ('SA', 'Salerno')])
>>> d.keys()
['NA', 'BN', 'SA', 'CE', 'AV']
>>> d.values()
dict_values(['Napoli', 'Avellino', 'Benevento', 'Caserta', 'Salerno'])
>>> print(d.get("NA"))
Napoli
>>> print(d.get("MI"))
None
>>> d.update( { "NA": "Naples", "MI": "Milano" } )
>>> print(d)
{'NA': 'Naples', 'AV': 'Avellino', 'BN': 'Benevento', 'CE': 'Caserta', 'SA': 'Salerno', 'MI': 'Milano'}
```



Gestione delle eccezioni

- Un'**eccezione** è un errore che si produce a tempo di esecuzione (*runtime*)
- Quando si verifica un errore, di regola il programma è terminato
- In alcune circostanze è possibile prevedere il verificarsi di un errore
 - ad es. perché l'input fornito dall'utente non è corretto
- Con `try` è possibile "catturare" un evento di errore prodotto da uno statement
- Esempio (due versioni)

```
prompt = "Inserisci un numero intero: "  
warning = "Non hai inserito un numero \\  
intero valido. Riprova."  
while True:  
    try:  
        x = int(input(prompt))  
        break  
    except ValueError:  
        print(warning)  
print("Hai inserito il numero", x)
```

```
prompt = "Inserisci un numero intero: "  
warning = "Non hai inserito un numero \\  
intero valido. Riprova."  
while True:  
    try:  
        x = int(input(prompt))  
    except ValueError:  
        print(warning)  
        continue  
    break  
print("Hai inserito il numero", x)
```



Il concetto di namespace

- Un ***namespace*** è un contesto logico nel quale ciascun identificatore è unico
 - Ad esempio, in uno script non possono esistere due variabili globali con lo stesso nome
- In namespace differenti possono esistere oggetti differenti con lo stesso nome
 - All'interno di una funzione può esistere una variabile locale con lo stesso nome di una variabile globale: una funzione ha un suo proprio namespace locale

Moduli e package in Python

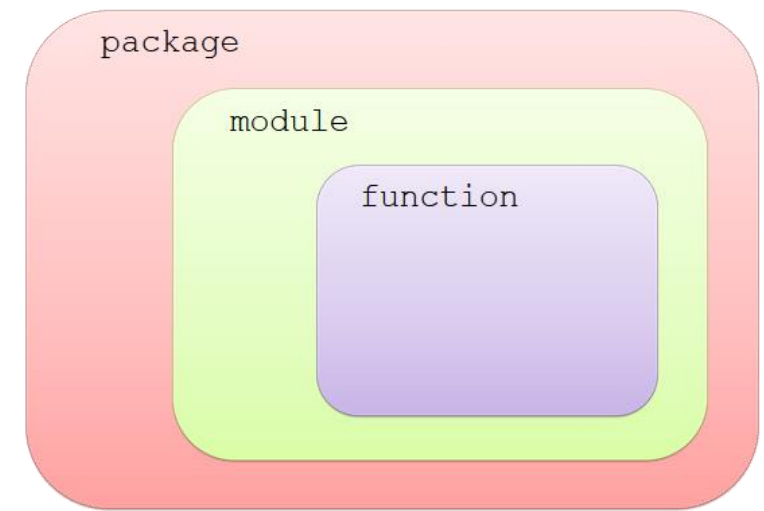
- Un **modulo** è un file nel quale si trovano funzioni e statement Python
- Un **package** è una collezione di moduli i cui file sono collocati nella stessa directory
- Esistono sia moduli distribuiti insieme all'interprete (*built-in*) che moduli distribuiti separatamente e resi disponibili all'interprete mediante un tool di installazione
- L'interprete Python rende già disponibili al programmatore molte librerie che implementano funzioni di utilità (si parla di *standard library* del linguaggio)

`time, datetime, string, os,`

`sys, getopt, fileinput, ...`

- Molte altre librerie Python possono essere scaricate ed installate con il tool **pip**

`numpy, matplotlib, ...`





Uso di librerie in Python: `import`

- Lo statement

```
import my_lib
```

dice all'interprete di rendere visibile (nello script in cui si trova) tutto ciò che è visibile a livello globale nel modulo `my_lib.py`

- In questo modo si possono usare variabili e funzioni definite in una libreria
- I nomi di variabili e funzioni della libreria sono associati al **namespace** `my_lib`
- Il nome `n` definito in `my_lib.py` dovrà essere riferito come `my_lib.n` in uno script che fa `import my_lib`
- Se lo script `my_lib.py` contiene statement eseguibili, essi sono eseguiti nel momento in cui è eseguito l'import
- Il file libreria `my_lib.py` si può trovare:
 - o nella stessa cartella dove si trova lo script che fa l'import
 - o in una cartella prevista dall'interprete (es. in `c:\Python38\Lib\site-packages`)

Uso di `import`: esempio

```
# File: my_lib.py
currentYear = 2018
def currentAge(birthYear):
    global currentYear
    return (currentYear-birthYear)

print("Libreria my_lib caricata")
```

- L'esecuzione di `my_lib_test.py` produce come output:

```
Libreria my_lib caricata
Eta' dei miei amici nell' anno 2018
Giovanni2: eta' 42
Mario: eta' 46
Andrea: eta' 40
Luigi: eta' 50
Giovanni: eta' 53
```

```
# File: my_lib_test.py
import my_lib
# NB: le chiavi devono essere uniche
annoNascita_amici = {
    'Mario': 1972,
    'Giovanni': 1965,
    'Giovanni2': 1976,
    'Andrea': 1978,
    'Luigi': 1968 }
print("Eta' dei miei amici nell'anno ", \
    my_lib.currentYear)

for amico in annoNascita_amici:
    print("%15s: eta'" % amico, end=" ")
    print(my_lib.currentAge(
        annoNascita_amici[amico]))
```



Uso di librerie in Python: `from-import`

- Lo statement

```
from my_lib import *
```

dice all'interprete di rendere visibili nel namespace dello script corrente tutti i nomi definiti nel modulo `my_lib.py`

- In questo caso, il nome `n` definito in `my_lib.py` potrà essere direttamente riferito come `n` nello script che fa l'import
- Occorre fare attenzione a possibili conflitti di nomi definiti in file differenti

Uso di `from-import`: esempio



```
# File: my_lib.py
currentYear = 2018
def currentAge(birthYear):
    global currentYear
    return (currentYear-birthYear)

print("Libreria my_lib caricata")
```

- L'esecuzione di `my_lib_test.py` produce come output:

```
Libreria my_lib caricata
Eta' dei miei amici nell' anno 2018
Giovanni2: eta' 42
Mario: eta' 46
Andrea: eta' 40
Luigi: eta' 50
Giovanni: eta' 53
```

```
# File: my_lib_test_2.py
from my_lib import *
# NB: le chiavi devono essere uniche
annoNascita_amici = {
    'Mario': 1972,
    'Giovanni': 1965,
    'Giovanni2': 1976,
    'Andrea': 1978,
    'Luigi': 1968 }
print("Eta' dei miei amici nell'anno", \
    currentYear)

for amico in annoNascita_amici:
    print("%15s: eta'" % amico, end=" ")
    print(currentAge(
        annoNascita_amici[amico]))
```

- L'output prodotto da questo programma è identico a quello prodotto dal precedente
- Si noti che i nomi `currentYear` e `currentAge` sono adesso nel namespace principale
 - La notazione `my_lib.currentYear` e `my_lib.currentAge` produrrebbe un errore



Programmazione ad oggetti in Python



Il concetto di classe in Python

- Una classe è un modello di oggetti costituito da attributi e metodi
- Relativamente agli attributi, occorre distinguere tra
 - attributi di classe, condivisi da tutte le istanze della classe
 - attributi di istanza, specifici per ciascuna istanza della classe
- Una classe `MyClass` è definita mediante uno statement `class`

```
class MyClass:  
    statement_1  
    statement_2  
    statement_3  
    ...
```

- Lo statement `class` crea un nuovo namespace `MyClass`
- Gli statement che costituiscono il corpo di `class` sono eseguiti
- Tipicamente, il corpo di `class` è costituito da statement del tipo:
 - `nome = valore` assegnazioni per la inizializzazione di attributi di classe
 - `def nome_metodo:` definizione di funzioni membro (metodi)
- Gli attributi di classe sono condivisi tra tutte le istanze della classe e sono identificati con la scrittura `MyClass.nome`

Costruttore e altri metodi

- Nella definizione di una classe, il metodo `__init__` ha la funzione di **costruttore**
- Esso viene eseguito quando si crea un'istanza di una classe mediante un'istruzione:

```
nome_oggetto = MyClass(param1, param2, ...)
```

- Nella definizione dei metodi della classe, incluso `__init__`, il primo argomento deve essere `self`, un riferimento all'istanza che è poi passato implicitamente all'atto dell'invocazione del metodo

```
def __init__(self, param1, param2, ...):
```

- Nel codice che costituisce il corpo dei metodi, incluso il costruttore, per fare riferimento agli attributi (variabili di istanza) si usa la notazione

```
self.nome_attributo
```

- La scrittura `nome_oggetto.f()` equivale a `MyClass.f(nome_oggetto)`
- La funzione predefinita `isinstance(obj, myClass)` restituisce `True` se `obj` è istanza della classe `myClass`, `False` altrimenti



Classi: esempi

```
>>> class MyClass1:  
...     pass  
...  
>>> class MyClass2:  
...     pass  
...  
>>> obj1 = MyClass1()  
>>> obj2 = MyClass2()  
>>>
```

```
>>> isinstance(obj1, MyClass1)  
True  
>>> isinstance(obj2, MyClass1)  
False  
>>> isinstance(obj1, MyClass2)  
False  
>>> isinstance(obj2, MyClass2)  
True
```

Attributi di istanza

- In Python si chiamano **attributi di istanza** quelle informazioni che costituiscono lo stato di ciascuna specifica istanza di una classe
 - Due istanze della stessa classe avranno valori diversi degli attributi di istanza
- Si tratta di variabili il cui valore è inizializzato nel costruttore `__init__` e che possono essere riferite con la notazione:
nome_oggetto.nome_attributo
- Nel costruttore, un attributo di istanza è riferito come **self.nome_attributo**

```
class Automobile:
    def __init__(self, nuova_targa):
        self.targa = nuova_targa

myCar = Automobile("NA3131")
print(myCar.targa)
```

Incapsulamento

- In Python non esistono meccanismi che consentano dichiarare un attributo pubblico o privato
- Per impedire la modifica diretta degli attributi di un oggetto, in Python si può dare a questi attributi “privati” un nome che inizia con doppio underscore
- Questa nomenclatura rende visibile l’attributo solo all’interno della classe
 - Per rendere accessibile il valore dell’attributo nascosto occorrerà definire un metodo che restituisce il valore dell’attributo o ne consenta la modifica

```
class Automobile:  
    def __init__(self, nuova_targa):  
        self.__targa = nuova_targa  
    def get_targa(self):  
        return self.__targa  
    def set_targa(self, nuova_targa):  
        self.__targa = nuova_targa
```

```
myCar = Automobile("NA3131")  
print(myCar.get_targa())  
myCar.set_targa("MI4321")  
print(myCar.get_targa())
```



Attributi di classe

- In Python si chiamano **attributi di classe** quelle informazioni che sono uguali per tutte le istanze di una classe
- Si tratta di variabili il cui valore è inizializzato nella definizione della classe
- All'esterno della classe, un attributo di classe sarà riferito con la notazione:

nome_classe.nome_attributo

```
class Automobile:  
    numero_ruote = 4  
    def __init__(self, nuova_targa):  
        self.targa = nuova_targa  
  
print(Automobile.numero_ruote)
```




Esempio di classe in Python

```
import math

class Cerchio:
    def __init__(self, c, r):
        self.centro = c
        self.raggio = r

    def area(self):
        return math.pi * self.raggio**2

c1 = Cerchio((0, 0), 5)
print("Il cerchio c1 ha raggio", c1.raggio)
print("Il cerchio c1 ha centro", c1.centro)
print("Il cerchio c1 ha area", c1.area())
print("c1 e' di tipo", type(c1))
```

- L'output prodotto è:

```
Il cerchio c1 ha raggio 5
Il cerchio c1 ha centro (0, 0)
Il cerchio c1 ha area 78.53981633974483
c1 e' di tipo <class '__main__.Cerchio'>
```

Ereditarietà

- Una classe può essere definita per derivazione da una classe base (**ereditarietà**)
- Una classe derivata può ridefinire (**overriding**) tutti i metodi di una classe base
- I tipi built-in del linguaggio non possono essere usati come classi base dal programmatore
- Per definire una classe `DerivedClass` come derivata da una classe `BaseClass` si usa la sintassi:

```
class DerivedClass(BaseClass):  
    statement_1  
    statement_2  
    statement_3  
    ...
```

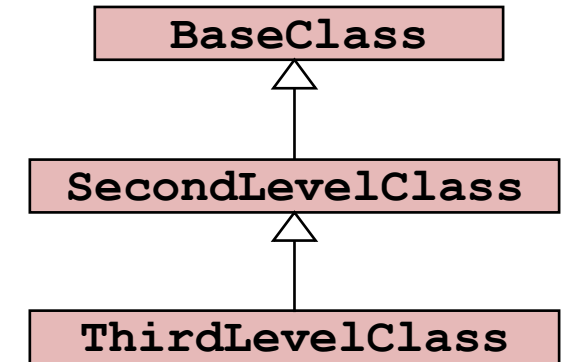
- In un metodo `f` della classe derivata `DerivedClass` si può invocare il metodo omologo della classe base con il nome `BaseClass.f`
- Una classe può essere fatta derivare da una classe a sua volta derivata
 - in questo modo si possono realizzare gerarchie di ereditarietà multi-livello
- E' anche possibile realizzare l'ereditarietà multipla definendo una classe come

```
class DerivedClass(BaseClass1, BaseClass2, ...):
```

Gerarchie di ereditarietà

- Una classe può essere fatta derivare da una classe a sua volta derivata
- In questo modo si possono realizzare gerarchie di ereditarietà multi-livello
- Una qualsiasi classe deriva dalla classe predefinita `object`

```
class BaseClass:  
    pass  
class SecondLevelClass(BaseClass):  
    pass  
class ThirdLevelClass(SecondLevelClass):  
    pass
```



- La funzione predefinita `issubclass(class1, class2)` restituisce:
 - `True` se `class1` è derivata da `class2` o da una sua sottoclasse
 - `False` altrimenti
- Con riferimento alle classi dell'esempio precedente:
 - `issubclass(SecondLevelClass, BaseClass)` restituisce `True`
 - `issubclass(ThirdLevelClass, BaseClass)` restituisce `True`
 - `issubclass(ThirdLevelClass, SecondLevelClass)` restituisce `True`

Ereditarietà multipla in Python

- Per la risoluzione di un qualsiasi nome in una classe (attributi o metodi), Python usa il seguente ordinamento di ricerca:
 1. dentro la stessa classe dell'oggetto;
 2. nelle sue classi "padre", dalla più diretta a salire verso l'alto nella gerarchia;
 3. se ci sono più classi "padre" allo stesso livello (*ereditarietà multipla*), la ricerca tra le classi viene fatta da sinistra verso destra, nell'ordine con cui sono state elencate nella definizione della classe

```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"

class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass

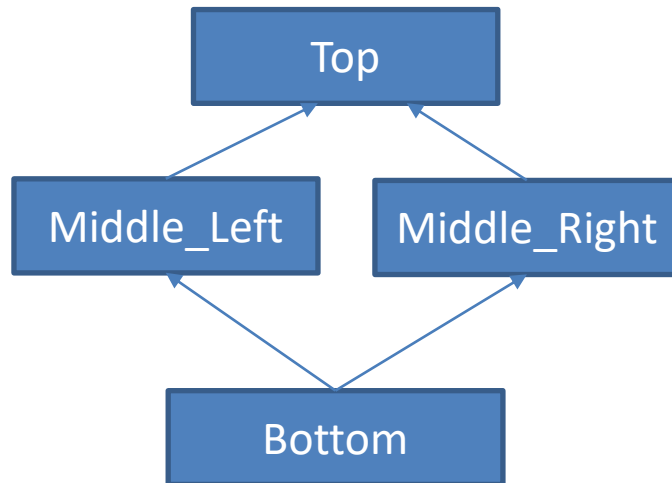
obj = Sub()

print(obj.var_left, obj.var_right, \
      obj.var, obj.fun())
```

```
LL RR L Left
```

Ereditarietà multipla e diamond problem

- A quale metodo si riferisce `object.m_middle()` ?



- Risposta: a `Middle_Left.m_middle()`

```
class Top:
    def m_top(self):
        print("top")

class Middle_Left(Top):
    def m_middle(self):
        print("middle_left")

class Middle_Right(Top):
    def m_middle(self):
        print("middle_right")

class Bottom(Middle_Left, Middle_Right):
    def m_bottom(self):
        print("bottom")

object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

```
bottom
middle_left
top
```



Polimorfismo in Python

- In Python ogni oggetto mantiene informazione sulla propria classe di appartenenza
- Se un metodo è ridefinito in una classe derivata, quando si invoca quel metodo su un oggetto della classe derivata l'interprete eseguirà il codice corrispondente

```
v1 = AutoPrivata()

v2 = AutoDaNoleggio()

v3 = Taxi()

veicoli = [v1, v2, v3]

for veicolo in veicoli:
    veicolo.avvia()
```

```
class Automobile:
    def avvia(self):
        print("Avvio automobile")

class AutoPrivata(Automobile):
    def avvia(self):
        print("Avvio auto privata")

class AutoDaNoleggio(Automobile):
    def avvia(self):
        print("Avvio auto da noleggio")

class Taxi(Automobile):
    def avvia(self):
        print("Avvio taxi")
```

Avvio auto privata
Avvio auto da noleggio
Avvio taxi



Esempi di programmi Python



Esempio #1: calcolo numeri primi

- Calcola i numeri primi compresi tra 1 e 1000 mediante una funzione

```
def primes(up_to):  
    primes = []  
    for n in range(2, up_to + 1):  
        is_prime = True  
        for divisor in range(2, n):  
            if n % divisor == 0:  
                is_prime = False  
                break  
        if is_prime:  
            primes.append(n)  
    return primes  
  
print(primes(1000))
```

- Output prodotto:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,  
79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163  
, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251  
, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349  
, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443  
, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557  
, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647  
, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757  
, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863  
, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983  
, 991, 997]
```




Esempio #2: manipolazione di una stringa

- Siccome le stringhe sono oggetti *immutable*, quando occorre eseguire una manipolazione di una stringa, il programmatore ne deve creare una nuova
- La funzione mostrata nell'esempio che segue restituisce una nuova stringa a partire da una stringa fornita come primo argomento, sostituendo tutte le occorrenze del carattere fornito come secondo argomento con il carattere fornito come terzo argomento (che per default è '~')

```
def replace(origin_string, char_to_replace, new_char = '~'):
    new_string = ''
    for i in range(len(origin_string)):
        if (origin_string[i] == char_to_replace):
            new_string += new_char
        else:
            new_string += origin_string[i]
    return new_string
```

```
a = "Qui, Quo e Qua sono nipoti di Paperino"
print(a)
a = replace(a, ' ', '_')
print(a)
a = replace(a, '~'); print(a)
```

```
Qui, Quo e Qua sono nipoti di Paperino
Qui,_Quo_e_Qua_sono_nipoti_di_Paperino
Qui,~Quo~e~Qua~sono~nipoti~di~Paperino
```



Esempio #3: calcolo media cumulativa

- Si vuole calcolare la media di un insieme di campioni **samples**
 - Si assume di non conoscere a priori il numero di campioni finale
 - Detto **n-1** il numero di campioni considerati fino ad un certo momento, e detta **avg** la media degli n-1 campioni considerati fino a quel momento, l'aggiunta di un n-esimo campione **new_sample** comporta un aggiornamento della media:

$$\text{avg} = (\text{avg} * (\text{n}-1) + \text{new_sample}) / \text{n}$$

```
samples = [24, 28, 22, 18, 30]
n = 0
avg = 0
for new_sample in samples:
    n += 1
    avg = (avg * (n-1) + new_sample) / n
    print("current sample:", new_sample, " - average: %.2f"%avg)

print("samples:", samples)
print("average: %.2f" % avg)
```

```
current sample: 24 - average: 24.00
current sample: 28 - average: 26.00
current sample: 22 - average: 24.67
current sample: 18 - average: 23.00
current sample: 30 - average: 24.40
samples: [24, 28, 22, 18, 30]
average: 24.40
```

Esempio #4: analisi di dati da file CSV (1)

- Scrivere un programma Python che estragga da un file dati testuale in formato CSV (*comma separated values*) le temperature medie giornaliere e successivamente rappresenti la temperatura giornaliera con un diagramma
- La struttura del file `daily_temperature.csv` è la seguente:

```
Day;Temperature
2019-01-01;4.597416666666665
2019-01-02;4.088999999999999
2019-01-03;1.774333333333334
2019-01-04;0.9607500000000001
2019-01-05;2.254583333333333
....
```

- Ogni riga contiene una sequenza di due dati separati da punto e virgola
- Il significato dei dati è descritto nella prima riga del file (che va ignorata nella elaborazione)

Esempio #4: analisi di dati da file CSV (2)



- Prima parte:

```
filename = "daily_temperature.csv"
csv_file = open(filename, "r", encoding="latin-1")
lines = csv_file.readlines()[1:] # ignora la prima riga

daily_temp = {}

for line in lines:
    line = line.rstrip('\n')
    data_item = line.split(';')
    date = data_item[0]
    temp = float(data_item[1])
    daily_temp.update({date: temp})

csv_file.close()
....
```

Esempio #4: analisi di dati da file CSV (3)



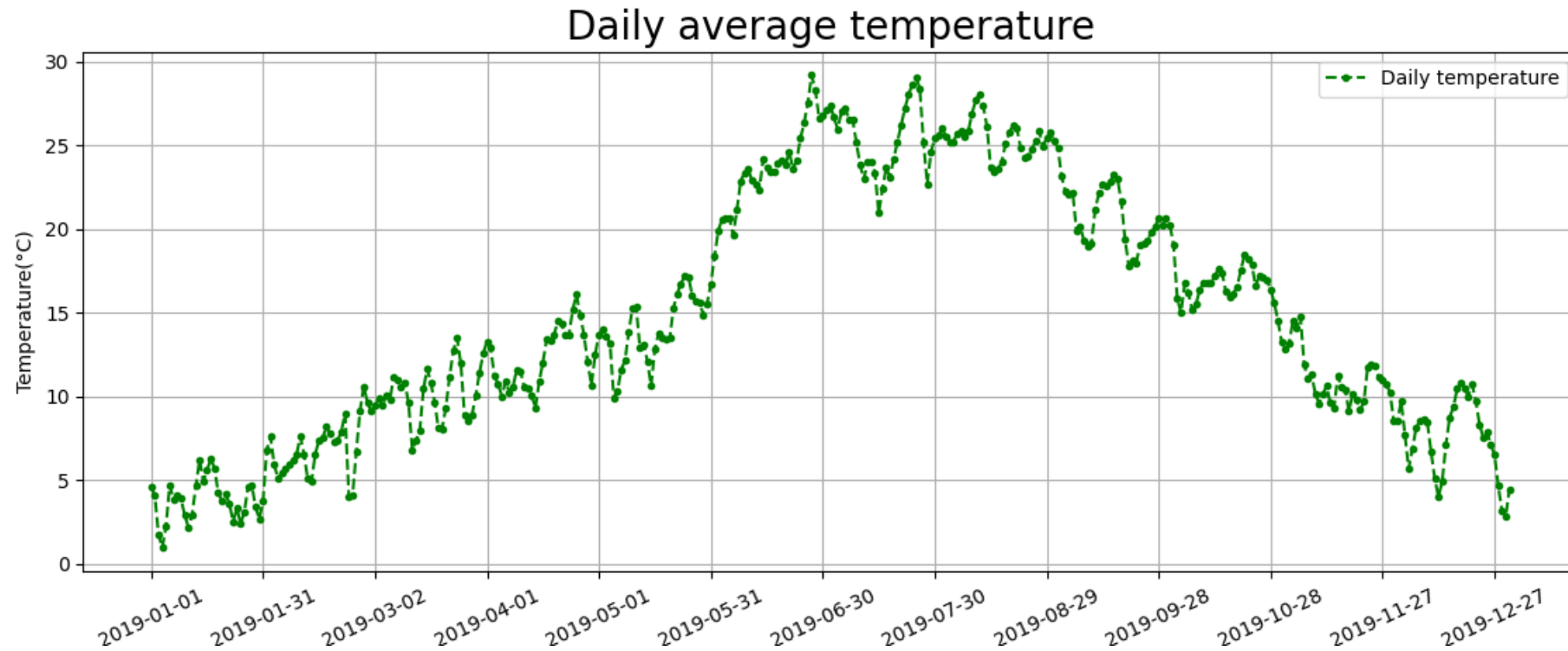
- Seconda parte:

```
....  
import matplotlib.pyplot as plt  
  
ticks = list(daily_temp.keys())[::30]  
plt.plot(daily_temp.keys(), daily_temp.values(),  
         color = 'g', linestyle = 'dashed', marker = '.',  
         label = "Daily temperature")  
  
plt.xticks(ticks, rotation = 25)  
plt.xlabel('Dates')  
plt.ylabel('Temperature(°C)')  
plt.title('Daily average temperature', fontsize = 20)  
plt.grid()  
plt.legend()  
plt.show()
```

Esempio #4: analisi di dati da file CSV (4)



- Il programma produce come output la figura seguente:



Esempio #5: grafico di una funzione



Plot di una funzione

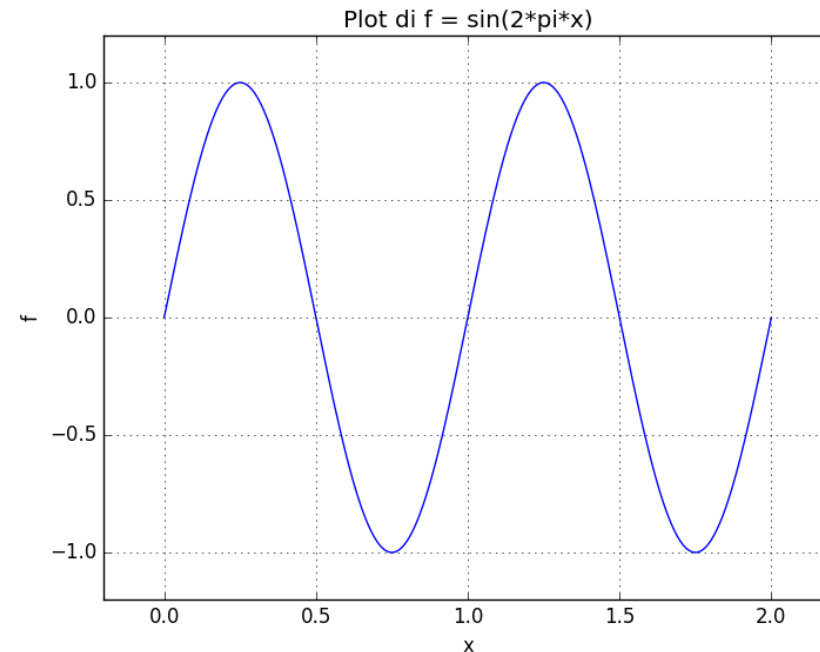
```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.0, 2.01, 0.01)
y = np.sin(2*np.pi*x)

plt.plot(x, y)
plt.margins(0.1)

plt.xlabel('x')
plt.ylabel('f')
plt.title('Plot di f = sin(2*pi*x)')
plt.grid(True)
plt.savefig("test.png")
plt.show()
```

Output prodotto (test.png)



Questo esempio utilizza i package matplotlib e numpy che vanno installati separatamente

numpy: le funzioni `arange` e `sin`

- `np.arange(start, end, step)` restituisce una lista di numeri `float` prodotti a partire da `start` con incremento `step` fino al valore massimo `end` (escluso)

```
[ 0.  0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11
 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23
 0.24 0.25 0.26 0.27 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35
 0.36 0.37 0.38 0.39 0.4  0.41 0.42 0.43 0.44 0.45 0.46 0.47
 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59
 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69 0.7  0.71
 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95
 0.96 0.97 0.98 0.99 1.   1.01 1.02 1.03 1.04 1.05 1.06 1.07
 1.08 1.09 1.1  1.11 1.12 1.13 1.14 1.15 1.16 1.17 1.18 1.19
 1.2  1.21 1.22 1.23 1.24 1.25 1.26 1.27 1.28 1.29 1.3  1.31
 1.32 1.33 1.34 1.35 1.36 1.37 1.38 1.39 1.4  1.41 1.42 1.43
 1.44 1.45 1.46 1.47 1.48 1.49 1.5  1.51 1.52 1.53 1.54 1.55
 1.56 1.57 1.58 1.59 1.6  1.61 1.62 1.63 1.64 1.65 1.66 1.67
 1.68 1.69 1.7  1.71 1.72 1.73 1.74 1.75 1.76 1.77 1.78 1.79
 1.8  1.81 1.82 1.83 1.84 1.85 1.86 1.87 1.88 1.89 1.9  1.91
 1.92 1.93 1.94 1.95 1.96 1.97 1.98 1.99 2. ]
```

```
[ 0.00000000e+00  6.27905195e-02  1.25333234e-01  1.87381315e-01
 2.48689887e-01  3.09016994e-01  3.68124553e-01  4.25779292e-01
 4.81753674e-01  5.35826795e-01  5.87785252e-01  6.37423990e-01
 6.84547106e-01  7.28968627e-01  7.70513243e-01  8.09016994e-01
 8.44327926e-01  8.76306680e-01  9.04827052e-01  9.29776486e-01
 9.51056516e-01  9.68583161e-01  9.82287251e-01  9.92114701e-01
 ...]
```

- `np.sin(2*np.pi*x)` (essendo `x` una lista) restituisce una lista di numeri `float` prodotti applicando la funzione `sin(2*np.pi*x)` a ciascun elemento di `x`