



# CORSO DI PYTHON

Dott. Antonio Giovanni Lezzi

# NOTE IMPORTANTI

- C'è una sola via da percorrere per imparare a programmare, dovete: ascoltare la trattazione teorica, leggere codice, scrivere codice, porsi delle domande costruttive ed imparare dagli errori
- Per questo dovrete sempre scrivere il codice degli esempi ed eseguirlo per poi vedere che cosa succede: giocate con il codice, modificalo come volete, la cosa peggiore che può capitarvi è che il programma non funzioni
- Provate a fare esercizi sempre nuovi e porvi dei target oltrepassando i limiti
- Mai rimandare a domani quello che puoi far oggi: gli argomenti si accumulano

# CORSO PYTHON

- Introduzione, Storia e Punti di forza
- Interpreti e tecnologie
- Installazione
- Linea dei comandi ed esecuzione
- Indentazione e blocco codice
- Istruzioni e Commenti
- Variabili e Stringhe
- Input Output
- Operatori
- Condizioni e Cicli



# INTRODUZIONE

- Python è uno dei linguaggi più usati al mondo
- Grazie alla sua sintassi asciutta e potente e al supporto multiplatforma è utilizzato per moltissime tipologie di applicazioni

# INTRODUZIONE

- Gli ambiti di applicazione di questo linguaggio di programmazione sono svariati: sviluppo di siti o applicazioni Web e desktop, realizzazione di interfacce grafiche, amministrazione di sistema, calcolo scientifico e numerico, database, giochi, grafica 3D, multimedia, grafica, intelligenza artificiale e tanto altro ancora
- È un linguaggio multiplatforma, ovvero disponibile per tutti i principali sistemi operativi, ed è automaticamente incluso nelle distribuzioni Linux e nei computer Macintosh
- Inoltre fornisce tutti gli strumenti per scrivere in modo semplice programmi portabili, ovvero che si comportano alla stessa maniera se eseguiti su differenti piattaforme

# STORIA

- Nei primi anni '80 al National Research Institute for Mathematics and Computer Science (CWI) di Amsterdam, alcuni ricercatori tra cui Guido Van Rossum hanno sviluppato un linguaggio di nome ABC, molto potente ed elegante, che era diventato popolare nel mondo Unix
- Qualche anno dopo (fine anni ottanta) Guido Van Rossum ha avuto una serie di idee mirate al miglioramento di ABC, e pertanto si mise a lavorare allo sviluppo di un nuovo linguaggio: Python
- Cerca di fare mente locale su quanto ha appreso durante il periodo di lavoro su ABC. Quell'esperienza è stata piuttosto frustrante, ma alcune caratteristiche di ABC gli piacciono, tanto da pensare di usarle come fondamenti del suo nuovo linguaggio:
  - A. l'indentazione per indicare i blocchi di istruzioni annidate
  - B. nessuna dichiarazione delle variabili
  - C. stringhe e liste di lunghezza arbitraria



# STORIA

- Su queste basi inizia a scrivere in C un interprete per il suo futuro linguaggio di programmazione, che battezza con il nome di Python in onore della sua serie televisiva preferita: Monty Python's Flying Circus
- Nel 1996 scrisse come prefazione del libro "Programming Python", prima edizione, le seguenti parole:

«Più di sei anni fa, nel dicembre 1989, stavo cercando un progetto di programmazione per "hobby" che mi avrebbe dovuto tenere occupato nella settimana vicina a Natale. Il mio ufficio... sarebbe stato chiuso, ma io avevo un computer, e non molto di più. Decisi di scrivere un interprete per un nuovo linguaggio di scripting a cui avrei pensato dopo: un discendente dell'ABC, che sarebbe dovuto appartenere agli hacker di Unix. Scelsi Python come nome per il progetto, essendo leggermente irriverente (e perché sono un grande fan di Monty Python's Flying Circus).»

# STORIA

- Nel 2000 Van Rossum e il suo team si trasferiscono presso BeOpen.com e formano i BeOpen PythonLabs team, con Python giunto alla versione 1.6
- Poco tempo dopo viene rilasciata la versione 2.0, che, tra le altre cose, migliorava il linguaggio con l'aggiunta delle "list comprehension"
- Nel 2001 viene rilasciato Python 2.1, e ridefinita la licenza come "Python Software Foundation License"
- Python 2.2 fu considerato un "rilascio pulizia", e la principale novità introdotta riguardò l'unificazione dei tipi/classi
- Bisogna arrivare al Dicembre 2008 per assistere ad una vera rivoluzione, con il rilascio della versione 3.0 di Python (o "Python 3000" o "Py3k"). Questa nuova versione è molto simile alla precedente, ma ha semplificato il linguaggio e introdotto diversi miglioramenti (come ad esempio le stringhe Unicode di default). Per via di questi cambiamenti, come vedremo in questa guida, Python 3 non è compatibile con Python 2.



# STORIA

- Al momento ci sono stati altri 5 rilasci di Python 3 (fino ad arrivare a Python 3.5), che hanno aggiunto ulteriori funzionalità e nuovi moduli
- L'ultima versione di Python 2 è invece Python 2.7, che ormai riceve solo bug fix
- È utilizzato con successo in tutto il mondo da svariate aziende e organizzazioni, tra le quali Google, la NASA, YouTube, Intel, Yahoo! Groups, reddit, Spotify Ltd, OpenStack e Dropbox Inc. Quest'ultima merita una nota a parte, poiché la sua storia ci consente di evidenziare diversi punti di forza di Python.

# STORIA

- Dropbox è un software multiplatforma che offre un servizio di file hosting e sincronizzazione automatica dei file tramite web. La sua prima versione è stata rilasciata nel settembre del 2008 e in pochissimo tempo ha avuto un successo sorprendente, raggiungendo i 50 milioni di utenti nell'ottobre del 2011 e i 100 milioni l'anno successivo, come annunciato il 12 novembre del 2012 da Drew Houston, uno dei due fondatori della Dropbox Inc.
- Dopo nemmeno un mese dall'annuncio di questo incredibile risultato, il 7 dicembre 2012 Drew stupisce tutti con un'altra clamorosa notizia. Guido van Rossum, dopo aver contribuito per sette anni alle fortune di Google, si unisce al team di Dropbox

# I PUNTI DI FORZA

- **È free**

Python è completamente gratuito ed è possibile usarlo e distribuirlo senza restrizioni di copyright. Nonostante sia free, da oltre 25 anni Python ha una comunità molto attiva, e riceve costantemente miglioramenti che lo mantengono aggiornato e al passo coi tempi

- **È multi-paradigma**

Python è un linguaggio multi-paradigma, che supporta sia la programmazione procedurale (che fa uso delle funzioni), sia la programmazione ad oggetti (includendo funzionalità come l'ereditarietà singola e multipla, l'overloading degli operatori, e il duck typing). Inoltre supporta anche diversi elementi della programmazione funzionale (come iteratori e generatori)

- **È portabile**

Python è un linguaggio portabile sviluppato in ANSI C. È possibile usarlo su diverse piattaforme come: Unix, Linux, Windows, DOS, Macintosh, Sistemi Real Time, OS/2, cellulari Android e iOS. Ciò è possibile perché si tratta di un linguaggio interpretato, quindi lo stesso codice può essere eseguito su qualsiasi piattaforma purché abbia l'interprete Python installato



# I PUNTI DI FORZA

- **È facile da usare**

Python è un linguaggio di alto livello che è al tempo stesso semplice e potente. La sintassi e i diversi moduli e funzioni che sono già inclusi nel linguaggio sono consistenti, intuitivi, e facili da imparare, e il design del linguaggio si basa sul principio del least astonishment (cioè della “minor sorpresa”: il comportamento del programma coincide con quanto ci si aspetta)

- **È ricco di librerie**

Ogni installazione di Python include la standard library, cioè una collezione di oltre 200 moduli per svolgere i compiti più disparati, come ad esempio l'interazione con il sistema operativo e il filesystem, o la gestione di diversi protocolli. Inoltre, il Python Package Index consente di scaricare ed installare migliaia di moduli aggiuntivi creati e mantenuti dalla comunità

# I PUNTI DI FORZA

- **È performante**

Anche se Python è considerato un linguaggio interpretato, i programmi vengono automaticamente compilati in un formato chiamato bytecode prima di essere eseguiti. Questo formato è più compatto ed efficiente, e garantisce quindi prestazioni elevate. Inoltre, diverse strutture dati, funzioni, e moduli di Python sono implementati internamente in C per essere ancora più performanti

- **Gestisce automaticamente la memoria**

Python è un linguaggio di alto livello che adotta un meccanismo di garbage collection che si occupa automaticamente dell'allocazione e del rilascio della memoria. Questo consente al programmatore di usare variabili liberamente, senza doversi preoccupare di dichiararle e di allocare e rilasciare spazi di memoria manualmente (cosa che è invece necessaria in linguaggi di più basso livello come il C o il C++).

# IMPLEMENTAZIONI DI PYTHON

- C'è un'importante precisazione che si deve fare in merito al termine Python, infatti viene utilizzato per indicare due cose strettamente correlate, ma comunque distinte:
  - **il linguaggio Python**
  - **l'interprete Python.**
- Il linguaggio Python, come alcuni linguaggi di programmazione, è l'analogo di una lingua come può essere l'italiano o l'inglese, composto quindi da un insieme di parole, da regole di sintassi e da una semantica



# IMPLEMENTAZIONI DI PYTHON

- Il codice ottenuto dalla combinazione di questi elementi si dice che è scritto nel linguaggio di programmazione Python
- Questo codice, di per sé, non ha alcuna utilità. Diviene utile nel momento in cui si ha uno strumento che lo analizza, lo capisce e lo esegue. Questo strumento è l'interprete Python
- Quindi, quando installiamo Python o usiamo il programma python, stiamo installando o usando l'interprete, ovvero il tool che ci consente di eseguire il codice scritto nel linguaggio di programmazione Python
- L'interprete Python è, a sua volta, scritto in un linguaggio di programmazione

# INTEGRAZIONE

- È integrabile con altri linguaggi
- Oltre all'interprete classico scritto in C (e chiamato CPython), esistono anche altri interpreti che consentono l'integrazione con diversi altri linguaggi
- **IronPython** consente di utilizzare Python all'interno del framework .NET, di usarne le sue funzioni, e di interagire con altri linguaggi .NET
- Per poter invece integrare Python e Java è possibile utilizzare **Jython**
- Esistono poi altri interpreti, come **PyPy**: un'implementazione altamente performante scritta in Python.

# PYTHON PER APP

- Con Python è possibile scrivere interfacce grafiche (GUI) usando tutti i maggiori toolkit:
  - **Tkinter**: già incluso nella standard library e basato su Tcl/Tk
  - **PyQt/PySide**: permettono di utilizzare con Python il toolkit Qt (sia la versione 4 che la 5), il framework multipiattaforma storicamente realizzato da Nokia
  - **PyGtk**: basato sul popolare toolkit GTK
  - **wxPython**: un'interfaccia Python per il toolkit wxWidgets
- I programmi che usano questi toolkit sono in grado di essere eseguiti su tutte le maggiori piattaforme (Linux, Windows, Mac)



# PYTHON PER IL WEB

- Esistono svariate possibilità per lo sviluppo Web sia ad alto che a basso livello. Per realizzare siti ed applicazioni web sono disponibili diversi web framework come:
  - **Django**: uno dei framework web più popolari, che fornisce diversi strumenti per realizzare siti e webapp.
  - **Flask**: un “microframework” che permette di creare rapidamente siti semplici.
  - **Web2py**: un altro ottimo framework facile da usare.
- Sono poi disponibili diversi altri web framework che permettono la realizzazione di ogni tipologia di sito e webapp
- Il sito ufficiale di Python include un elenco di web framework (completi di una breve descrizione) e una guida che spiega come usare Python nel web

# PYTHON PER IL WEB

- La piattaforma Google App Engine permette di avviare le proprie applicazioni Web nell'infrastruttura Google, infatti App Engine ha un ambiente runtime Python dedicato, che include l'interprete Python e la libreria standard Python
- Se invece si vuole scendere più a basso livello esistono moduli della standard library come socket, httpplib, e urllib, ma anche alcuni framework a supporto della programmazione di rete
- Uno fra tutti è Twisted: un potente network engine event-driven scritto in Python, che supporta molti protocolli di rete inclusi SMTP, POP3, IMAP, SSHv2 e DNS
- È possibile usare Python anche per accedere ai database. La standard library include un'interfaccia per SQLite ed è anche possibile installare moduli per interfacciarsi con altri database (PostgreSQL, Oracle, MySQL, e altri)
- Per realizzare applicazioni scientifiche, **SciPy** fornisce un ecosistema di tool per la matematica, le scienze e l'ingegneria.

# DIFFERENZE TRA PYTHON 2 E PYTHON 3

- La seguente lista include alcuni tra i principali cambiamenti introdotti da Python 3 (che saranno comunque più chiari in avanti):
- `print` è una funzione (in Python 2 era uno statement) e va invocata usando le parentesi: `print('x')`
- `input` è stato rimosso, e `raw_input` è stato rinominato `input`
- tutte le stringhe sono ora Unicode di default, permettendo l'utilizzo di qualsiasi alfabeto
- i tipi `unicode` e `str` di Python 2 sono stati rinominati rispettivamente in `str` e `bytes`
- i tipi `long` e `int` sono stati unificati in `int`

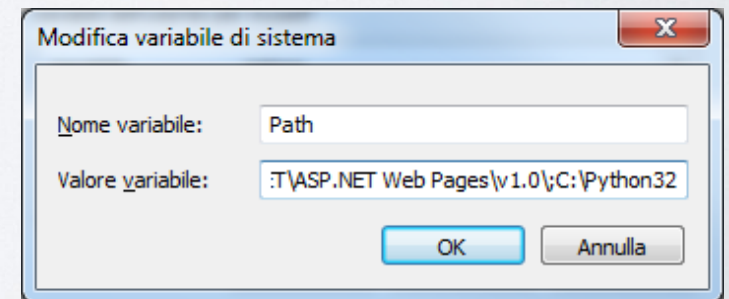


# DIFFERENZE TRA PYTHON 2 E PYTHON 3

- la divisione tra due int ora ritorna un float (per esempio:  $5 / 2 == 2.5$ )
- funzioni come range, map, e zip sono ora più efficienti
- metodi come dict.keys(), dict.values(), e dict.items() restituiscono view invece che creare nuove liste
- l'operatore <> e altre sintassi duplicate e obsolete sono state rimosse
- alcuni moduli, metodi, e funzioni sono stati rinominati per rispettare lo stile di scrittura PEP 8

# INSTALLAZIONE DI PYTHON

- Nella pagina ufficiale di download di Python è possibile trovare il file .msi per le versioni Windows a 32bit e 64bit
- Dopo l'installazione, Python di default viene collocato nel path `C:\Python32`
- A questo punto è già possibile utilizzarlo da: *Start->Tutti i programmi->Python3.2->Python*
- Se si vuole però avviare Python senza problemi da una finestra DOS, bisogna andare in: *Pannello di controllo->Sistema->Impostazioni di Sistema Avanzate->Variabili d'ambiente*
- Quindi modificare la variabile Path dalle variabili di sistema aggiungendo `C:\Python32`



# INSTALLAZIONE DI PYTHON

- Gli sviluppatori di Python mettono a disposizione un apposito installer per Windows sia in versione per le architetture a 32 bit che per quelle a 64 bit, tale package è scaricabile direttamente dal sito ufficiale del progetto (<http://www.python.it/download/>)
- Una volta terminato il download si potrà effettuare il classico doppio click sull'eseguibile scaricato; si aprirà così una schermata dove bisognerà scegliere se consentire l'accesso a Python al solo utente corrente o a tutti gli utenti definiti per il sistema
- Fatto questo basterà cliccare su "Next" per proseguire e scegliere la directory per l'installazione, ad esempio "C:\Python\" e cliccare nuovamente su "Next" per personalizzare l'installazione; ai neofiti si consiglia di lasciare inalterata l'impostazione predefinita

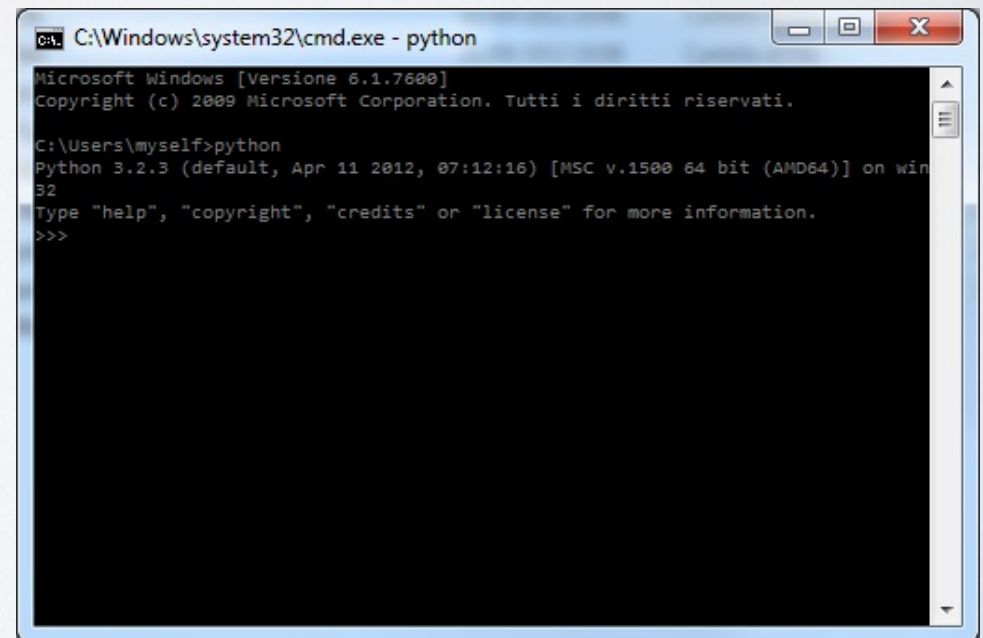


# INSTALLAZIONE DI PYTHON

- Cliccando per l'ennesima volta su "Next" si potrà passare alla fase di installazione vera e propria con la necessaria copia dei file su Windows
- il sistema di setup proseguirà senza la necessità di un intervento manuale e, una volta terminata, basterà cliccare su "Finish" per completare l'operazione
- Per verificare l'esito dell'installazione, ci si potrà ricercare sul menù "Start" e selezionare "Tutti i programmi > Python"; si aprirà così una finestra interattiva per l'interazione con Python

# INSTALLAZIONE DI PYTHON

- Se si vuole verificare che tutto è andato a buon fine, avviare il prompt dei comandi (Start->Esegui->cmd) e digitare:  
*echo %PATH%*
- quindi se si ritrova la variabile d'ambiente C:\Python è possibile avviare python digitando semplicemente python



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Versione 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Users\myself>python
Python 3.2.3 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

# INTERPRETE DI PYTHON

- Lanciando il comando python da riga di comando qualunque sia il sistema operativo, si avvia **“interprete interattivo”**, dall’aspetto del tutto simile ad un’altra riga di comando
- Appare uno specifico prompt, caratterizzato da 3 caratteri di maggiore (>>>)
- In altre parole un interprete Python è un programma che permette l’esecuzione di altri programmi



# VERSIONE DI PYTHON

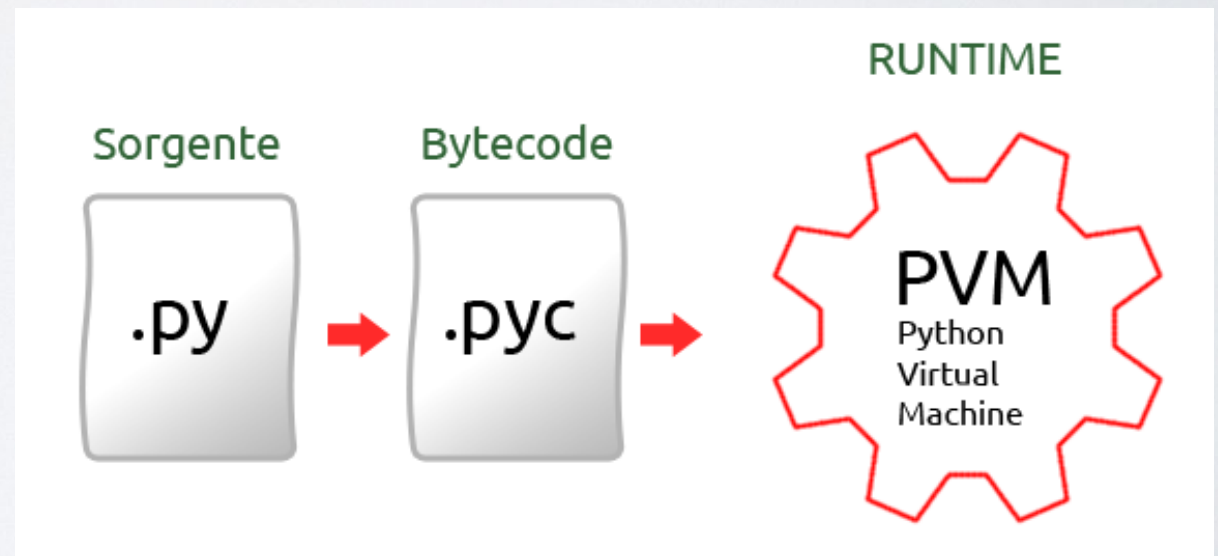
- Per controllare la versione installata di Python si accede al suo interprete e si digitano le seguenti righe:

```
import sys  
print(sys.version)
```

```
2.7.10 (default, Jul 30 2016, 18:31:42)  
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)]
```

# INTERPRETE DI PYTHON

- Ogni volta che viene invocato il comando python il codice scritto viene scansionato per token, vengono analizzati dentro una struttura ad albero che rappresenta la struttura logica del programma, in fine viene trasformata in bytecode (file .pyc o .pyo)
- Per potere eseguire questi bytecode serve un interprete bytecode che è una macchina virtuale Python(PVM)



# INTERPRETE DI PYTHON

- Python si caratterizza anche per essere un linguaggio pseudocompilato
- Ad occuparsi dell'analisi (parsing) dei codici sarà un apposito interprete messo a disposizione nativamente dal linguaggio che dovrà quindi verificare la correttezza sintattica dei sorgenti ed eseguirli
- Python non dà vita a dinamiche presenti in altri linguaggi, come per esempio a C, per i quali viene richiesta una compilazione del codice per la generazione di file eseguibili
- Tra i vantaggi della precompilazione vi è sicuramente la portabilità, infatti un sorgente scritto su una piattaforma (si ipotizzi Windows) potrà essere successivamente interpretato su un sistema operativo differente (ad esempio una distribuzione Linux) con l'unico vincolo derivante dalla versione utilizzata del linguaggio



# ESECUZIONE A RIGA DI COMANDO

- La caratteristica fondamentale è che digitando dei comandi nel prompt interattivo si ottengono subito le risposte

- ecco degli esempi:

```
>>> 5*3
```

```
15
```

```
>>> a=5
```

```
>>> b=6
```

```
>>> 2*(a+b)+3*a
```

```
37
```

# PRIMA APPLICAZIONE

- Come prima interazione con il linguaggio, una volta lanciato IDLE, digitiamo la seguente istruzione e osserviamo l'output generato:

```
>>> print ("Ciao Mondo!")  
Ciao Mondo!
```

- I costrutti basati sul comando print consentono di stampare una stringa passata come parametro (nel nostro caso "Ciao Mondo!"); come è facile notare l'interprete riconosce la stringa perché delimitata da doppi apici, l'argomento passato a print viene a sua volta delimitato grazie a delle parentesi tonde. Il commento iniziale verrà invece ignorato e non parteciperà alla generazione dell'output.
- L'uso delle parentesi è fondamentale quando si programma con il ramo 3.x del linguaggio, in precedenza infatti, come accade con il ramo 2.x, era possibile ottenere il medesimo risultato utilizzando un formato come il seguente:  

```
>>> print "Hello, World!"
```

# PRIMA APPLICAZIONE

- Tentativo di stampa senza utilizzo delle parentesi tonde per la delimitazione
- Ma si provi a lanciare tale istruzione tramite IDLE nel caso di un'installazione basata sulla versione 3.x, ciò che si otterrà in output sarà in pratica un errore:

```
>>> print "Hello, World!"
```

```
SyntaxError: Missing parentheses in call to 'print'
```

- Il messaggio generato dall'interprete appare abbastanza chiaro, per la chiamata al comando print è richiesta la delimitazione tra parentesi



# L'ESEGUIBILE .PY

- Da IDLE andiamo sul percorso di menù "File > New File", si aprirà una finestra all'interno della quale sarà possibile digitare la seguente istruzione:

```
# Il mio primo programma in Python  
print ("Ciao Mondo!")
```

- Salviamo quindi il file appena scritto con il nome di "PrimoProgrammaPython.py" e poi rechiamoci sul percorso di menù "Run > Run Module", oppure clickiamo sul tasto "F5" o ancora apriamo la Shell di Python e digitiamo:

```
>>> python PrimoProgrammaPython.py
```

- Otterremo come risposta il seguente output:

```
>>> Ciao Mondo!
```

- Con pochi semplici passaggi abbiamo creato un primo e semplice programma in Python che potrà essere eseguito su qualsiasi piattaforma che supporti il linguaggio e in cui esso sia stato installato, fermo restando le esigenze legate alla versione di riferimento di Python che deve essere la medesima

# ESECUZIONE FILE .PY

- Ma come si fa a creare ed eseguire un file Python? Per prima cosa bisogna creare un semplice file di testo e salvarlo con estensione “.py”, per esempio possiamo chiamare un file helloworld.py

- A questo punto possiamo aprire il file creato con un qualsiasi editor di testi non Wordprocessor e scrivere:

```
print('Hello World!!')
```

- Il file contiene una sola riga di codice con la funzione print(), che come risultato stamperà una stringa

- Eseguiamo il file:

```
C:\python32>python helloworld.py  
Hello World!!
```

# INTRODUZIONE AL LINGUAGGIO

- Ci sono aspetti fondamentali da tener presente nella programmazione con Python, il più importante certamente è la caratterizzazione dei blocchi di codice attraverso l'indentazione del testo
- Una pratica diversa rispetto da altri linguaggi di programmazione in cui si delimitano i blocchi con parentesi graffe, begin-end o altri costrutti
- Vedremo anche come dichiarare le variabili e commentare il codice



# INDENTAZIONE E BLOCCHI DI CODICE

- Python è stato progettato e realizzato per essere un linguaggio chiaro e leggibile, per questo è stato scelto di utilizzare l'indentazione per definire i blocchi di codice o il contenuto dei cicli di controllo
- All'inizio ci si può confondere con l'indentazione e i blocchi, e può anche capitare che gli spazi o la tabulazione, di conseguenza l'indentazione, cambiano in funzione dell'editor che si usa
- Il vantaggio è quello di avere un'alta leggibilità del codice, proprio perché si è obbligati ad indentare i sorgenti

# INDENTAZIONE E BLOCCHI DI CODICE

- Vediamo in generale come viene rappresentata l'indentazione di blocchi nel caso dei cicli di controllo:

*istruzione:*

*blocco*

*istruzione:*

*blocco*

- È molto importante fare attenzione all'indentazione perché viene considerata nel processo di parsing del programma

# INDENTAZIONE DEL CODICE

- Un blocco di codice nidificato non è delimitato da parole chiave o da parentesi graffe, bensì dal simbolo di due punti e dall'indentazione stessa del codice:

```
>>> for i in range(2): # Alla prima iterazione `i` varrà 0 e alla seconda 1
...     if i % 2 == 0:
...         print("Sono all'interno del blocco if, per cui...")
...         print(i, "è un numero pari.\n")
...         continue # Riprendi dalla prima istruzione del ciclo `for`
...     print("Il blocco if è stato saltato, per cui...")
...     print(i, "è un numero dispari\n")
```

```
Sono all'interno del blocco if, per cui...
0 è un numero pari.
Il blocco if è stato saltato, per cui...
1 è un numero dispari
```



# BLOCCO DI CODICE

- Per blocco di codice nidificato intendiamo il codice interno a una classe, a una funzione, a una istruzione if, a un ciclo for, a un ciclo while e così via
- I blocchi di codice nidificati, e solo loro, sono preceduti da una istruzione che termina con il simbolo dei due punti
- Vedremo più avanti che i blocchi nidificati sono la suite delle istruzioni composte o delle relative clausole

# BLOCCO DI CODICE

- L'indentazione deve essere la stessa per tutto il blocco, per cui il numero di caratteri di spaziatura (spazi o tabulazioni) è significativo:

```
>>> for i in range(2):  
...     print(i) # Indentazione con 4 spazi  
...     print(i + i) # Indentazione con 3 spazi...
```

*File "<stdin>", line 3*

```
    print(i + i) # Indentazione con 3 spazi...
```

*IndentationError: unindent does not match any outer indentation level*

- Nella guida dello stile stesura codice (PEP-0008) si consiglia di utilizzare quattro spazi per ogni livello di indentazione e di non mischiare mai spazi e tabulazioni

# BLOCCO DI CODICE

- Utilizzare insieme spazi e tabulazioni per indentare istruzioni nello stesso blocco è un errore:

```
>>> for i in range(2):  
...     print(i) # Indentazione con spazi  
...     print(i) # Indentazione con TAB
```

```
File "<stdin>", line 3  
    print(i) # Indentazione con TAB
```

*TabError: inconsistent use of tabs and spaces in indentation*

- L'indentazione, quindi, è un requisito del linguaggio e non una questione di stile, questo implica che tutti i programmi Python abbiano lo stesso aspetto



# SPAZI O TABULAZIONI? QUANTI?

- Le indentazioni devono essere fatte utilizzando gli Spazi
- Così è indicato anche nelle guideline ufficiali e la motivazione principale sta nel fatto che non c'è uniformità di rappresentazione delle tabulazioni tra diversi editor, il che potrebbe indurre in errore anche il più attento dei programmatori
- In genere si utilizzano indentazioni di quattro spazi, ma possiamo decidere arbitrariamente, ciò che conta è che siano coerenti i livelli di indentazione, ovvero la distanza tra le istruzioni interne al blocco di codice e quelle esterne
- Se pensate di mettere a disposizione il vostro codice ad altri sviluppatori, al di fuori di un team, è consigliabile adeguarsi ai 4 spazi canonici, mentre se si fa parte di un team di sviluppo sarebbe utile, ma è anche ovvio, che tutti condividano lo stesso stile per le indentazioni

# SPAZI O TABULAZIONI? QUANTI?

- Altra cosa sono gli spazi utilizzati all'interno delle istruzioni, per i quali invece abbiamo piena libertà, nel rispetto della sintassi
- Ad esempio, all'interno della definizione di un array possiamo utilizzare anche una indentazione di questo tipo:

```
>>> vec = [  
    'uno',  
    'dieci',  
    'cinque'  
]  
>>> print(vec)
```

```
['uno', 'dieci', 'cinque']
```

# ISTRUZIONI

- Non è necessario terminare le istruzioni con un punto e virgola, ma è sufficiente andare su una nuova linea
- Il punto e virgola è invece necessario se si vogliono inserire più istruzioni su una stessa linea:  

```
>>> a = 'Per favore, non farlo mai!'; print(a)
```
- Se potete non fatelo mai, renderebbe il codice poco leggibile



# PRIMA APPLICAZIONE

- Di default Python offre uno strumento per la programmazione denominato IDLE, esso mette a disposizione un'interfaccia grafica con la quale lanciare istruzioni basate sul linguaggio e creare file eseguibili con estensione ".py"
- Vengono inoltre fornite feature avanzate per incrementare il livello di produttività delle sessioni di sviluppo come per esempio:
- i suggerimenti per il completamento delle istruzioni
- l'evidenziazione del codice (syntax highlight) tramite una colorazione differente dei vari costrutti
- un debugger per i test sugli script
- funzionalità per la ricerca all'interno di sorgenti (anche con supporto per le espressioni regolari)
- tool per l'indentazione del listato.
- Una volta avviato IDLE si potrà lanciare immediatamente un primo comando basato su Python per cominciare ad analizzarne la sintassi di base; le istruzioni dovranno essere scritte dopo i simboli ">>>" che indicano la modalità interattiva, cioè la possibilità di comunicare con l'interprete

# COMMENTI

- Come qualsiasi linguaggio di programmazione che si rispetti, Python supporta i commenti
- Sono delle annotazioni che lo sviluppatore potrà inserire all'interno del sorgente per incrementarne il livello di leggibilità usando qualsiasi linguaggio naturale o sequenza di caratteri
- Per commentare il codice si fa uso dei caratteri appositamente concepiti per la loro delimitazione, l'interprete del linguaggio riconoscerà i commenti come tali evitando che essi entrino nel flusso di esecuzione del codice e siano visibili all'interno degli output

# COMMENTI

- È possibile distinguere tra commenti a linea singola e commenti multilinea;
- i commenti a linea singola vengono ospitati su una sola riga e introdotti tramite un delimitatore iniziale che è il carattere cancelletto (#)

```
>>> # Questo è un commento su singola linea.  
>>> # Anche questo è un commento su singola linea.  
>>> # Basta con i commenti su singola linea!
```

- I commenti multilinea, cioè quelli che si estendono su più righe, vengono invece delimitati tramite tre apici singoli posti sia in apertura che in chiusura del commento:

```
>>> '''  
    Questo è un commento  
    multilinea.  
    '''
```



# COMMENTI

- Andrebbe sottolineato che i commenti multilinea vengono sconsigliati dagli sviluppatori di Python in quanto potrebbe generare confusione nella lettura del codice
- Nelle più recenti versioni di Python l'uso della delimitazione `''' ... '''` può essere interpretata come istruzione per l'output della stringa contenuta tra gli apici
- Per le ragioni esposte è consigliabile, pertanto, l'utilizzo dei commenti linea singola anche per il multilinea, come nell'esempio seguente:

```
>>> # Questo  
    #è un commento  
    #multilinea
```

# VARIABILI

- Non è necessario definire le variabili prima di utilizzarle, non è necessario nemmeno assegnare ad esse un tipo
- Il tutto avviene implicitamente mediante l'istruzione di assegnamento `=`, un po' come in JavaScript o nel vecchio Basic
- Esistono i classici tipi di dati, comuni al linguaggio C, dal quale eredita diverse caratteristiche, e a molti altri linguaggi

# VARIABILI

- Ecco una tabella che riassume le caratteristiche principali dei tipi di dati disponibili:

<b>Tipo</b>	<b>Rappresentazione interna</b>	<b>Esempio</b>
<u>Intero</u>	32bit	1200, -56, 0
<u>Reale</u>	32 bit (tipo double del C)	1.23 3.14e-10, 4.0E210
<u>Booleano</u>	intero con 1=VERO e 0=FALSO (come in C)	0, 1
<u>Complesso</u>	coppia di numeri reali	3+4j, 5.0+4.1j, 3j
<u>Stringhe</u>	lista di caratteri	'Antonio', "l'acqua"



# VARIABILI

- Ricordiamo che le regole da seguire nella scelta dei nomi delle variabili è simile a quella dei più comuni linguaggi di programmazione, in particolare:
- Ogni variabile deve iniziare con una lettera oppure con il carattere underscore “\_”, dopodiché possono seguire lettere e numeri o il solito underscore
- Python è un linguaggio case sensitive, quindi distingue le variabili composte da caratteri minuscoli da quelle scritte con caratteri maiuscoli

# VARIABILI

- Esistono delle parole riservate che non possono essere utilizzate per i nomi delle variabili
- Esse sono le seguenti:  
***and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while***
- È possibile assegnare un valore ad una variabile mediante l'operatore “=”

# ASSEGNAMENTO MULTIPLO

- Una singolare possibilità offerta da Python è rappresentata da l'assegnamento multiplo nel quale si possono inizializzare più variabili direttamente sulla stessa riga di codice
- Ecco qualche esempio:

```
>>> a = 'viva Python'
>>> b = 3
>>> c, d = 'insegnato', 'a scuola' # assegnamento multiplo
>>> print (a,b,c,d)
viva Python 3 insegnato a scuola
```



# TIPOLOGIA DI DATI

- Python offre quattro possibilità per rappresentare i numeri: interi, razionali, complessi e booleani, come visto nelle tipologie di dati
- Nelle versioni precedenti di Python esisteva una distinzione tra intero (int) e “intero lungo” (long), da Python 3 si parla solo di int che corrisponde al vecchio long (in pratica long è stato rinominato int)
- È interessante notare come non ci sia limite ad un numero intero, purchè sia seguito da un carattere "L" o "l", in base al valore python alloca la memoria necessaria

# TIPOLOGIA DI DATI

- Quando si usa una stringa è possibile racchiudere il suo valore indifferentemente con il carattere «"» oppure «'»
- Oltre a questi tipi di dati “semplici”, Python offre la possibilità di dichiarare anche strutture dati complesse implementate in modo nativo (“built-in types” come si dice ufficialmente):

***liste, dizionari, tuple e files***

# TIPOLOGIA DI DATI

- L'aritmetica utilizzata è molto simile a quella del C, facciamo qualche esempio:

```
>>> 6+4*3
18
>>> 9/2 # la divisione tra due numeri interi restituisce un float
4.5
>>> 2j * 2j # moltiplicazione tra numeri complessi
(-4+0j)
>>> int(10.6) # conversione esplicita dal tipo float al tipo int
10
>>> float(20) # conversione esplicita dal tipo int al tipo float
20.0
```



# STRING

- Per dichiarare una stringa sia sufficiente assegnare ad una nuova variabile un testo racchiuso tra virgolette: è possibile racchiudere il suo valore indifferentemente tra apici (carattere ') o doppi apici (carattere “")
- Questo permette di superare facilmente il problema dell'utilizzo dei suddetti caratteri nel valore stesso della stringa
- Ricordo comunque che è possibile forzare l'inserimento di ognuno dei due apici semplicemente raddoppiandoli
- Per iniziare ad approfondire un po', iniziamo col dire che le stringhe possono essere considerate come sequenze di caratteri (liste), è pertanto possibile prelevare i caratteri attraverso il loro indice, tenendo presente che si parte sempre da 0

# STRING: SLICING E SUBSTRING

- Ancora più interessante è la possibilità di prelevare sottostringhe utilizzando la tecnica dello “*slicing*” (affettare)
- Questa tecnica permette di tagliare una parte della lista indicando l'indice di partenza e l'indice finale
- È utile sapere però che la sotto-stringa prelevata è quella che va dal carattere indicato del primo indice incluso, fino quello dell'indice finale escluso
- Omettendo uno dei due indici indichiamo a Python di andare fino in fondo alla stringa
- Se inseriamo un numero negativo come indice finale, Python conterà i caratteri a ritroso

# STRING: SLICING E SUBSTRING

Operatore	Descrizione	Esempio
+	concatenamento	('a'+'b')='ab'
*	ripetizione	('a'*3)='aaa'
<code>s[i]</code>	indicizzazione dei caratteri	s='abc' s[0]='a'
<code>s[i:j]</code>	slicing	s='abc' s[1:2]='bc'
<code>len(s)</code>	lunghezza	s='abc' len(s)=3
%	formattazione di stringhe	('ciao %s' % 'Antonio')='ciao Antonio'



# STRING: ESEMPIO

```
>>> y='ciao mondo'
>>> y[0] # prende la prima lettera di y tenendo presente che si parte da 0
'c'
>>> y[5]
'm'
>>> x='programmazione python'
>>> x[-2] # stampa il secondo carattere partendo da destra
'o'
>>> x[2:7] # stampa i caratteri di x compresi da 2 a 7
'ogram'
>>> x[4:] # stampa dal quarto carattere alla fine di x
'rammazione python'
>>> x[:4] # stampa i caratteri di x compresi dall'inizio al quarto
'prog'
>>> x[3:-2] # stampa dal terzo carattere (contando da 0) al penultimo di x
'grammazione pyth'
```

# STRING: METODI

- Il metodo `find()` serve a ricercare dei caratteri da una stringa (restituisce -1 se non è presente)

```
>>> s='Corso di Piton'  
>>> s.find('orso')  
1
```

- Il metodo `replace()` serve a sostituire dei caratteri da una stringa

```
>>> s.replace('Piton', 'Python')  
'Corso di Python'
```

- I metodi `upper()` e `lower()` servono invece a convertire in maiuscolo e minuscolo una stringa

```
>>> up=s.upper()  
>>> print(up)  
CORSO DI PYTHON  
>>> low=up.lower()  
>>> print(low)  
corso di python
```

# STRING, COME COSTRUIRLE

- Grazie all'operatore percentuale (%) possiamo costruire una stringa inserendo dei parametri “segnaposto” da sostituire con i valori assunti da variabili, come avviene nella istruzione printf del C

```
>>> nome = 'Antonio'  
>>> eta = 34  
>>> risultato = "%s ha %d anni" % (nome, eta)  
>>> print(risultato)  
Antonio ha 34 anni
```

- Nelle prime righe abbiamo definito le variabili e assegnato valori stringa e numerici. Nella stringa “%s ha %d anni” abbiamo inserito i parametri:
  - %s : parametro di tipo stringa
  - %d : parametro di tipo numerico



# STRINGHE

Parametro	Descrizione
%s	stringa
%c	singolo carattere
%d	numero decimale
%u	intero senza segno
%o	numero in notazione ottale
%x	numero in notazione esadecimale
%g	numero reale in notazione normale
%e	numero reale in notazione scientifica

# PRINT

- Abbiamo già trovato la funzione `print()`, serve a “stampare” in output (tipicamente sullo schermo) il valore di una variabile o di una espressione

```
>>> a = 12
>>> b = 3
>>> print(a,b,(a - b))
12 3 9
```

- La flessibilità del comando `print`, il quale concatena il valore di diversi parametri suddivisi dalla virgola e li mostra in output
- Nelle precedenti versioni di Python `print` era un comando e non una funzione e non era necessario inserire i parametri tra parentesi, mentre ora omettere le parentesi significa commettere un errore di sintassi

# PRINT

- Quando si lavora a riga di comando è sufficiente digitare il nome della variabile per stamparne il valore, questa possibilità offerta dalla modalità interattiva, consente un debug un po' più snello
- Quando si esegue un programma è necessario utilizzare il print

```
>>> x=10
>>> x
10
```



# INPUT

- Se realizziamo piccoli programmi da utilizzare da console o riga di comando, risulta utile la funzione `input`, che serve ad intercettare le sequenze inserite nello std-in, quindi tipicamente da tastiera

```
>>> valore = eval(input('Inserisci un valore numerico:'))
Inserisci un valore numerico: 5
>>> print(valore*valore)
25
>>> valore = input('Inserisci una stringa:')
Inserisci una stringa: ciao
>>> print(valore)
ciao
```

- In passato il comando `input` serviva per l'acquisizione di variabili numeriche, mentre per le sequenze e le stringhe si utilizzava `raw_input`
- Nella versione 3 di Python il comando `raw_input` diventa `input`
- Per ottenere il precedente comportamento di `input` occorre scrivere `eval(input(...))`

# ESERCIZI

- Scrivete un programma che stampa su schermo il vostro nome e cognome in due stringhe separate
- Scrivete un programma che mostra l'utilizzo delle 6 operazioni matematiche

# VARIABILI

- Le variabili sono dei costrutti comuni a quasi tutti i linguaggi di programmazione, esse nascono dall'esigenza di avere a disposizione uno spazio di memoria al quale associare dinamicamente delle informazioni; questa operazione per la memorizzazione diretta dei dati consiste in pratica nell'assegnazione di un valore alle variabili
- Per comprendere il funzionamento di tale processo di assegnazione è possibile avviare IDLE e digitare le seguenti istruzioni da salvare in un file chiamato, ad esempio, "variabili.py":

```
# Associare un valore stringa ad una variabile  
a = "Questa è una variabile"  
print(a)
```

- Nel caso specifico lo script stamperà a video la stringa *"Questa è una variabile"*



# VARIABILI

- vediamo un esempio con l'associazione ad un valore numerico:

```
# Associare un valore numerico ad una variabile  
a = 1000  
print(a)
```

- In questo caso invece, l'output prodotto dall'applicazione sarà "1000"

# VARIABILI

- .Quindi, volendo tirare le somme possiamo affermare che una variabile deve essere rappresentata da una stringa, che nel nostro esempio è il solo carattere "a"; per assegnare ad essa un valore di dovrà utilizzare il simbolo di uguaglianza ("=") che, nello specifico, agisce come operatore per l'assegnazione
- Esistono alcune regole da rispettare quando si vuole associare un nome ad una variabile che, naturalmente non deve chiamarsi forzatamente "a", esse in sostanza escludono determinate casistiche, motivo per il quale il nome di una variabile:
  - non potrà essere un numero, ad esempio non si potrà utilizzare "2" o "22" come nome di variabile
  - non potrà iniziare con un numero, per cui "2a" o "2aa" non saranno dei formati consentiti dall'interprete di Python
  - non potrà contenere spazi, quindi non si potrà adottare una variabile chiamata, per esempio, "a a" o "a b"
  - non potrà contenere segni di interpunzione, escludendo la possibilità di formati come "a.a" o "a!a"
  - non potrà contenere simboli, quindi nomi come "a€a" o "a\$a" porteranno alla generazione di un errore

# VARIABILI

- Sarà invece possibile:
  - utilizzare nomi composti da sequenze di più caratteri, ad esempio "abc", tra cui anche termini di senso compiuto ("ciao");
  - terminare il nome di una variabile con un numero, ad esempio "a2" o "a22";
  - utilizzare una stringa contenente al suo interno delle cifre, come per esempio "a2a";
  - adottare nomi composti da maiuscole e minuscole, per cui una variabile potrà essere chiamata anche "A", "AbC" o "ABC"
- Per quanto riguarda le regole dei valori utilizzati per l'assegnazione, come risulta chiaro dall'esempio precedente un valore stringa dovrà essere delimitato da doppi apici (a = "Questa è una variabile") mentre un valore numerico no (a = 2); un valore numerico delimitato da doppi apici (come per esempio a = "10") **verrà interpretato come una stringa**



# VARIABILI

- Poter comunicare all'interprete la corretta tipologia di valore (o tipo di dato) è fondamentale per il corretto funzionamento di uno script:

```
# Utilizzare i valori delle variabili  
# nei calcoli matematici  
x = 0  
y = 1000  
print(x + y)
```

# VARIABILI

- Nello script precedente viene stampata la somma tra i due valori numerici, ora scriviamo lo stesso script nel modo seguente:

```
# Utilizzare i valori numerici  
# come stringhe  
x = "0"  
y = "1000"  
print(x + y)
```

- Nel secondo caso il risultato sarà la stringa "01000", questo perché la presenza dei doppi apici forza l'interprete a riconoscere i valori assegnati alle variabili come delle stringhe e il "+" non agisce più come operatore matematico di **addizione**, ma come operatore per la **concatenazione**

# VARIABILI

- Lo sviluppatore avrà comunque la possibilità di modificare dinamicamente il tipo di dato associato ad una variabile attraverso i cosiddetti operatori di casting che sono:

- **int()**: converte una variabile al tipo di dato intero
- **str()**: converte una variabile al tipo di dato stringa
- **float()**: converte una variabile al tipo di dato float, un numero decimale

- Ecco un esempio:

```
# casting di una variabile stringa  
# in una variabile numerica intera  
x = "5"  
y = 1000  
print(int(x) + y)
```

- Teoricamente non dovrebbe essere possibile sommare i due valori assegnati alle variabili "x" e "y" dello script precedente, questo perché la prima è associata ad una stringa, la seconda ad un valore numerico; ma grazie al ricorso all'operatore int(), "x" verrà convertita in un altro valore numerico e l'output ottenuto in fase di esecuzione sarà pari a "1005"



# OPERATORI

- Gli operatori possono determinare un'assegnazione, consentono di effettuare operazioni matematiche a carico degli operandi
- Inoltre altri operatori permettono di effettuare dei confronti tra valori
- consentono di incrementare o effettuare decrementi a carico di questi ultimi
- tutto dipende dal simbolo utilizzato, dalla tipologia degli operandi impiegati e da alcune regole sintattiche

# OPERATORI: ARITMETICI

Operazione	Descrizione	Esempio
+	L'operatore di <b>addizione</b> somma i valori degli operandi.	$3 + 2 = 5$
-	L'operatore di <b>sottrazione</b> decrementa l'operando alla sinistra del simbolo di un valore pari a quello dell'operando alla destra di quest'ultimo.	$3 - 2 = 1$
*	L'operatore di <b>moltiplicazione</b> moltiplica l'operando alla sinistra del simbolo un numero di volte pari al valore dell'operando posto alla sua destra.	$3 * 2 = 6$
/	L'operatore di <b>divisione</b> divide l'operando alla sinistra del simbolo sulla base del valore associato all'operando posto alla sua destra.	$4 / 2 = 2$
%	L'operatore <b>modulo</b> restituisce il resto di una divisione.	$5 \% 2 = 1$
**	L'operatore <b>esponente</b> eleva a potenza l'operando alla sinistra del simbolo un numero di volte pari al valore dell'operando posto alla sua destra.	$8 ** 2 = 64$
//	L'operatore di <b>arrotondamento</b> restituisce il risultato di una divisione arrotondandolo al valore intero più prossimo a quello reale.	$8 // 3 = 2$ oppure $9.5 // 2 = 4.0$

# OPERATORI: ASSEGNAZIONE

Operazione	Descrizione	Esempio
=	Assegna il valore dell'operando alla sinistra del simbolo a quello alla sua destra	$z = x + y$ per cui il valore di $z$ sarà pari a 3
+=	Aggiunge l'operando di destra a quello di sinistra e assegna il risultato a quest'ultimo.	$z += x$ equivale a $z = z + x$
-=	Sottrae l'operando di destra a quello di sinistra e assegna il risultato a quest'ultimo.	$z -= x$ equivale a $z = z - x$
*=	Moltiplica l'operando di destra per quello di sinistra e assegna il risultato a quest'ultimo.	$z *= x$ equivale a $z = z * x$
/=	divide l'operando di sinistra per il valore di quello di destra e assegna il risultato al primo operando.	$z /= x$ equivale a $z = z / x$
%=	Calcola il resto dei due operatori e assegna il risultato all'operando di sinistra	$z \% = x$ equivale a $z = z \% x$
**=	Calcola un'elevazione a potenza e assegna il risultato all'operatorendo di sinistra	$z ** = x$ equivale a $z = z ** x$
//=	Restituisce l'arrotondamento di una divisione tra gli operatori e assegna il risultato all'operatorendo di sinistra	$z //= x$ equivale a $z = z // x$



# OPERATORI: CONFRONTO

Operazione	Descrizione	Esempio
==	Se il valore dei due operandi è uguale il confronto restituisce TRUE, altrimenti FALSE.	(3 == 3) restituisce TRUE (3 == 2) restituisce FALSE
!=	Se il valore dei due operandi non è uguale il confronto restituisce TRUE, altrimenti FALSE.	(3 != 2) restituisce TRUE
<>	Se il valore dei due operandi non è uguale il confronto restituisce TRUE, altrimenti FALSE. Come per l'operatore precedente	(3 <> 2) restituisce TRUE
>	Se il valore alla sinistra del simbolo è maggiore di quello alla sua destra restituisce TRUE, altrimenti FALSE.	(2 > 10) restituisce FALSE
<	Se il valore alla sinistra del simbolo è inferiore a quello alla sua destra restituisce TRUE, altrimenti FALSE	(2 < 10) restituisce TRUE
>=	Se il valore alla sinistra dei simboli è maggiore o uguale a quello alla sua destra restituisce TRUE, altrimenti FALSE.	(3 >= 5) restituisce FALSE
<=	Se il valore alla sinistra dei simboli è minore o uguale a quello alla sua destra restituisce TRUE, altrimenti FALSE.	(3 <= 5) restituisce TRUE

# OPERATORI: LOGICI

- Gli operatori logici sono dei costrutti sintattici utili al fine di legare due o più condizioni tra di loro, dove per condizione si intende una qualunque affermazione che può essere vera o falsa
- Si ipotizzi per esempio di avere una variabile "x" assegnata al valore TRUE e un'altra, "y", assegnata al valore FALSE

Operazione	Descrizione	Esempio
<i>and</i>	x and y restituisce FALSE	Restituisce TRUE se entrambi gli operatori sono TRUE.
<i>or</i>	x or y restituisce TRUE	Restituisce TRUE se almeno uno dei due operatori è TRUE.
<i>not</i>	not y restituisce TRUE	Restituisce TRUE se l'operando è FALSE.

# OPERATORI: MEMBERSHIP

- Gli operatori di membership potranno essere utilizzati per verificare se un determinato elemento fa parte o meno di un insieme, ad esempio una sequenza alfanumerica

Operazione	Descrizione	Esempio
<i>in</i>	Restituisce TRUE se un valore è presente all'interno di un insieme.	Se x = "Ciao mondo" allora print("C" in x) restituisce TRUE
<i>not in</i>	Restituisce TRUE se un valore non è presente all'interno di un insieme.	Se x = "Ciao mondo" allora print("Z" not in x) restituisce TRUE



# OPERATORI: IDENTITÀ

- Gli operatori di identità vengono utilizzati per verificare se due valori sono stati archiviati nella stessa porzione di memoria
- si tenga conto del fatto che per Python due valori "uguali" non sono necessariamente anche "identici":

Operazione	Descrizione	Esempio
<i>is</i>	Restituisce TRUE se gli operandi fanno riferimento al medesimo oggetto.	Se <code>x = "ciao"</code> e <code>y = "ciao"</code> allora <code>print(x is y)</code> restituisce TRUE perché uguali per valore e identici per tipo di dato
<i>is not</i>	Restituisce TRUE se gli operandi non fanno riferimento al medesimo oggetto	Se <code>x = 3</code> e <code>y = 3</code> allora <code>print(x is not y)</code> restituisce FALSE

# ESERCIZI

- Scrivete un programma che prenda due variabili stringa e due numeriche intere dall'utente, le concateni (unisca le due stringhe senza spazi) e le visualizzi sullo schermo, infine moltiplichi i due numeri interi su una nuova linea

# CONDIZIONI

- I costrutti condizionali sono espressioni il cui esito, cioè il risultato prodotto, dipende dalla soddisfazione o meno di una condizione precedentemente definita
- Per la definizione di un costrutto condizionale si utilizzano le keywords `if`, l'unica obbligatoria, `elif` ed `else`
- Gli esempi proposti di seguito ne evidenzieranno il funzionamento



# COSTRUTTO IF

- La keyword if ha il compito di introdurre una condizione, nel codice seguente quest'ultima si basa sull'uguaglianza di un valore passato interattivamente come parametro a quello assegnato ad una variabile:

```
# Utilizzo di if nei costrutti condizionali  
# l'applicazione restituirà in output una conferma  
# soltanto se il parametro inviato dall'utente  
# è identico al valore della variabile x
```

```
x = 8  
y = int(input("Inserisci un numero intero compreso tra 0 e 10: "))  
if y == x:  
    print("Il numero inserito è esatto")
```

- Tramite lo script proposto l'utente potrà digitare un numero che verrà impiegato come parametro per valorizzare la variabile "y", nel caso in cui tale valore dovesse essere identico a quello di "x", è cioè pari a "8", allora l'applicazione stamperà il messaggio di conferma "Il numero inserito è esatto", altrimenti nulla verrà restituito in output

# ELSE

- Il codice proposto, per quanto funzionante, è quindi incompleto, non prevede infatti alcun comportamento per i casi in cui l'uguaglianza richiesta dalla condizione introdotta da if non venga soddisfatta; a questo proposito è quindi possibile incrementare le funzionalità previste facendo ricorso alla keyword else
- else consente di definire un'istruzione alternativa che verrà eseguita quando una condizione imposta tramite if non dovesse verificarsi

```
# Utilizzo del blocco if/else nei costrutti condizionali  
# l'applicazione restituirà in output una conferma  
# se il parametro inviato è identico al valore di x  
# altrimenti verrà eseguita l'istruzione introdotta da else
```

```
x = 8  
y = int(input("Inserisci un numero da 0 a 10: "))  
if y == x:  
    print("Il numero inserito è esatto")  
else:  
    print("Il numero inserito non è esatto")
```

# ELSE

- L'istruzione introdotta da else dovrà essere preceduta da quest'ultimo seguito dal simbolo dei due punti (":"), esattamente come accade per if
- nel caso in cui il valore passato ad "y" sia diverso da "8" allora verrà stampato a video il messaggio "Il numero inserito non è esatto"
- Ora l'applicazione proposta risulta più completa rispetto a quella proposta nel primo esempio
- Cosa accadrebbe però se l'utente dovesse inserire un valore presente al di fuori dell'intervallo consentito, cioè tra "0" e "10"?
- Per questo caso specifico non è stato ancora definito alcun controllo



# ELSEIF

- elif è una parola chiave che introduce un'istruzione alternativa nei costrutti condizionali ma l'esecuzione di quest'ultima sarà vincolata ad un'ulteriore condizione:

```
# Utilizzo del blocco if/elif/else nei costrutti condizionali
# verrà restituita in output una conferma
# se il parametro inviato è identico al valore di x
# altrimenti verrà eseguito un confronto tra i valori di y e z
# o restituita l'istruzione introdotta da else
```

```
x = 8
z = 10
y = int(input("Inserisci un numero da 0 a " + str(z) + ":"))
if y == x:
    print("Il numero inserito è esatto")
elif y > z:
    print("Sono consentiti soltanto valori compresi tra 0 e " + str(z))
else:
    print("Il numero inserito non è esatto")
```

# ELSEIF

- In sostanza if introduce una condizione, per soddisfarla il valore di "y" dovrà essere uguale a quello di "x"
- nel caso in cui ciò non sia vero allora l'applicazione valuterà una seconda condizione, introdotta da elif e anch'essa seguita dal simbolo dei ":", che sarà soddisfatta soltanto se "y" dovesse avere un valore superiore a "z"
- La mancata soddisfazione di entrambe le condizioni indicate porterà invece all'esito previsto tramite else

# ELSEIF

- Dal punto di vista dell'ottimizzazione, dato che quello che si sta realizzando è un programma interattivo, potrebbe essere una buona idea fornire all'utilizzatore degli indizi che gli permettano di scoprire più velocemente l'entità del valore assegnato ad "x" diminuendo il numero di tentativi effettuati

```
# Utilizzo del blocco if/elif nei costrutti condizionali  
# verrà restituita in output una conferma  
# se il parametro inviato è identico al valore di x  
# altrimenti verrà eseguito un confronto tra i valori di y e x
```

```
x = 8  
y = int(input("Inserisci un numero da 0 a 10: "))  
if y == x:  
    print("Il numero inserito è esatto")  
elif y > x:  
    print("Il numero inserito è superiore a quello atteso")  
elif y < x:  
    print("Il numero inserito è inferiore a quello atteso")
```



# ELSEIF

- In pratica, la logica ci suggerisce che l'applicazione potrà funzionare perfettamente prevedendo soltanto tre casi:
- quello in cui il valore digitato sia esatto
- quello in cui esso sia maggiore di quello atteso
- quello in cui esso sia invece inferiore
- Prevedendo tali eventualità in altrettante condizioni non sarà necessario il ricorso ad un'alternativa introdotta con else

# COSTRUTTI CONDIZIONALI ANNIDATI

- In Python i costrutti condizionali prevedono la possibilità degli annidamenti (nested statements), questo significa che si potranno definire delle condizioni all'interno di altre condizioni. A tal proposito si analizzi il seguente esempio:

```
# Annidamento dei blocchi if/elif/else nei costrutti condizionali
x = 8
y = int(input("Inserisci un numero da 0 a 10: "))
if y == x:
    print("Il numero inserito è esatto")
elif y > x:
    print("Il numero inserito è superiore a quello atteso..")
    if (y - x) == 1:
        print("..ma ci sei andato molto vicino.")
    else:
        print("..riprova, sarai più fortunato.")
elif y < x:
    print("Il numero inserito è inferiore a quello atteso")
    if (x - y) == 1:
        print("..ma ci sei andato molto vicino.")
    else:
        print("..riprova, sarai più fortunato.")
```

# COSTRUTTI CONDIZIONALI ANNIDATI

- Il codice proposto non si limita a controllare che il valore proposto dall'input dell'utente sia uguale, maggiore o inferiore a quello atteso
- Verifica anche il grado di approssimazione di tale valore a quello esatto
- Uno scostamento unitario in eccesso o in difetto verrà notificato tramite un'apposita segnalazione aggiuntiva ("..ma ci sei andato molto vicino."), scostamenti più elevati porteranno alla stampa di un'alternativa introdotta tramite else ("..riprova, sarai più fortunato.")



# CICLO FOR

- Il ciclo (loop) for è uno dei cicli di iterazione messi a disposizione dal linguaggio Python
- Si tratta di un costrutto che consente di ripetere un'operazione un certo numero di volte, più tecnicamente "iterare una sequenza o un oggetto", fino alla soddisfazione di una determinata condizione (implicita) che terminerà il ciclo
- Per chiarire tale dinamica sarà possibile proporre un semplice esempio basato sull'impiego delle liste, queste ultime sono un tipo di dato supportato da Python che verrà analizzato nel dettaglio in seguito
- Per il momento basti sapere che una lista permette di gestire un insieme di valori di diversa natura (stringhe, interi, decimali) delimitati da parentesi quadre e separati tramite una virgola

# CICLO FOR

- Il codice proposto di seguito permetterà di ciclare e visualizzare in output tutti gli elementi presenti in una lista definita dallo sviluppatore.

```
# Ciclare gli elementi presenti in una lista tramite il  
ciclo for
```

```
# definizione della lista
```

```
fibonacci = [1,1,2,3,5,8,13,21,34,55,89,144]
```

```
# iterazione dei valori in lista
```

```
for val in fibonacci:
```

```
    # stampa dei valori iterati
```

```
    print(val)
```

# CICLO FOR

- Nell'esempio mostrato la condizione da soddisfare per la terminazione del ciclo sarà quindi quella di stampare fino all'ultimo valore presente nella lista passata come argomento, nel nostro caso "fibonacci"
- Un fattore sintattico molto importante da tenere a mente riguarda il fatto che nella digitazione di un ciclo for l'engine di Python si aspetta che l'istruzione associata al ciclo venga indentata (si noti infatti la presenza di una tabulazione prima del comando `print()`), ignorando tale regola in fase di esecuzione si assisterà alla generazione di un errore segnalato tramite la notifica:

**SyntaxError: expected an indented block**



# CICLI E ISTRUZIONI CONDIZIONALI

- E' interessante notare come il ciclo for possa essere sfruttato anche per eseguire operazioni più complesse rispetto alla semplice stampa a video degli elementi che compongono una sequenza
- A tal proposito si potrà modificare la piccola applicazione precedentemente proposta in modo che quest'ultima sommi tutti i valori presenti nella lista restituendo in output il totale ottenuto

```
# Sommare gli elementi presenti in una lista  
# tramite il ciclo for
```

```
# definizione della lista  
fibonacci = [1,1,2,3,5,8,13,21,34,55,89,144]
```

```
# variabile per l'archiviazione della somma  
# dei valori in lista  
totale = 0
```

```
# sommatoria dei valori in lista  
for val in fibonacci:  
    totale = totale + val
```

```
# stampa della somma ottenuta  
print(totale)
```

# CICLI E ISTRUZIONI CONDIZIONALI

- Nel caso specifico la condizione di terminazione del ciclo è il raggiungimento del totale tra tutti i valori degli elementi in lista, ma sarebbe possibile rendere il sorgente più articolato introducendo un controllo basato su if che escluda dalla somma determinati valori:

```
# Sommare gli elementi presenti in una lista
# tramite il ciclo for
# escludendo tutti i valori inferiori a 3

# definizione della lista
fibonacci = [1,1,2,3,5,8,13,21,34,55,89,144]

# variabile per l'archiviazione della somma dei valori in lista
totale = 0

# sommatoria dei valori in lista
for val in fibonacci:
    # controllo sul valore degli elementi sommati
    if val > 3:
        totale = totale + val
# controllo sul totale ottenuto
if totale == 369:
    # stampa della somma ottenuta
    print("Il totale della somma è " + str(totale))
else:
    # istruzione alternativa in caso di esito negativo del controllo
    print ("Valore differente da quello atteso.")
```

# CICLI E ISTRUZIONI CONDIZIONALI

- Modificando il codice utilizzato è mutata anche la condizione per la terminazione del ciclo, questa volta infatti dovranno essere sommati tutti gli elementi presenti in lista tranne quelli di valore inferiore a "3"
- L'output dell'applicazione originale era pari a "376", ma dato che nella sequenza sono presenti quattro valori inferiori o uguali a "3" ("1, 1, 2 e 3")
- Dopo le modifiche apportare il risultato sarà "369", cioè l'output iniziale meno il totale dei valori ora esclusi



# CICLO FOR APPLICATO AGLI INTERVALLI DI VALORI

- La funzione **range()** permette di non dover digitare sequenze di valori eccessivamente lunghe e consente di definire un intervallo di valori sulla base di due parametri
- Devono esser forniti un valore iniziale e un valore di terminazione
- È poi possibile passare alla funzione un terzo argomento (step) che rappresenta le posizioni che dovranno essere ignorate all'interno dell'intervallo

```
# Stampa di un intervallo di valori compresi tra 5 e 20
for val in range(5,20):
    print (val)
```

# CICLO FOR APPLICATO AGLI INTERVALLI DI VALORI

- Il risultato ottenuto sarà rappresentato da tutti i valori interni all'intervallo che va da "5" a "19", in quanto valore di terminazione "20" viene considerato invece al di fuori dell'intervallo e non verrà restituito dall'applicazione
- Detto questo, è possibile definire anche uno step che influenzerà direttamente la generazione dell'output:

```
# Stampa di un intervallo di valori compresi tra 5 e 20  
# escludendo due posizioni ad ogni iterazione
```

```
for val in range(5,20,2):  
    print (val)
```

- Questa volta la presenza del terzo parametro permetterà di escludere dal risultato un numero di posizioni pari al valore associato a tale argomento per ogni iterazione del ciclo, motivo per il quale non si otterrà più la sequenza "5 6 7 8 .. 19" ma "5 7 9 11 .. 19"

# CICLO WHILE

- Il ciclo while si differenzia dal ciclo for precedentemente descritto più a livello sintattico che funzionale
- In generale è comunque possibile affermare che mentre nel ciclo for le iterazioni proseguono fino al verificarsi di una determinata condizione (ad esempio il raggiungimento di un dato incremento di valore a carico di una variabile)
- Nel ciclo while le iterazioni continuano invece finché la condizione passata come argomento rimane vera



# CICLO WHILE

- L'applicazione seguente permette di stampare tutti i valori inferiori al parametro di condizione ("20") partendo dal valore associato ad una variabile precedentemente definita che verrà incrementata di 2 unità ad ogni iterazione del ciclo:

```
# Utilizzo del ciclo while  
# per la stampa dei valori compresi tra 10 e 20  
# vincolati all'incremento di una variabile
```

```
val = 10  
while val < 20:  
    print (val)  
    val+=2
```

# CICLO WHILE

- L'esecuzione del codice porterà alla stampa della sequenza composta dai valori "10 (valore di base della variabile sottoposta ad incremento), 12, 14, 16 e 18", non verrà invece restituito in output il valore "20" che, rappresentando la condizione di terminazione dell'incremento ed essendo superiore al più alto valore consentito, verrà giustamente considerato al di fuori dell'intervallo di valori ciclato
- Esattamente come nel caso di for, anche while può essere integrato tramite i blocchi condizionali, il codice dell'applicazione seguente introdurrà però una modalità inedita nella gestione degli output prodotti dai cicli e sottoposti ad una condizione introdotta tramite if:

```
# Utilizzo del ciclo while per la stampa dei valori compresi tra 10 e 20
# vincolati all'incremento di una variabile fino al raggiungimento del valore di
# confronto introdotto da if
val = 10
while val < 20:
    print (val)
    val+=2
    if val == 16:
        break
```

# CICLO WHILE

- Analizzando il sorgente proposto si noterà come la prima parte, quella relativa al blocco del ciclo while, sia del tutto identica al codice dell'esempio proposto in precedenza
- Questa volta però è stata implementata una seconda parte contenente una condizione per la quale l'incremento della variabile di base ("val") fino ad un valore pari a "16" porterà all'arresto del ciclo che non produrrà ulteriori iterazioni
- Si otterrà quindi come risultato la sequenza "10, 12 e 14", la presenza dell'istruzione break alla fine del codice consentirà di terminare il ciclo che, altrimenti, continuerebbe nel tentativo di incrementare la variabile fino all'ultimo valore inferiore a "20" e, non riuscendoci a causa della condizione dovuta ad if, porterebbe alla generazione di un loop infinito, cioè di un ciclo che potrebbe essere fermato soltanto forzando l'arresto dell'applicazione



# IL BLOCCO IF/ELSE NEI CICLI

- Alla luce delle nuove features descritte fino ad ora è possibile sottolineare come un altro punto in comune tra for e while sia rappresentato dalla possibilità di utilizzare un blocco condizionale quando richiesto dalla logica dell'applicazione sviluppata
- A tal proposito, si analizzi quindi un primo esempio basato sul ciclo for che sarà anche riassuntivo per molti dei costrutti analizzati fino ad ora (liste, break..)

# IL BLOCCO IF/ELSE NEI CICLI

```
# Applicazione interattiva basata su un ciclo for
# destinato a verificare la presenza di un valore digitato dall'utente
# all'interno di una lista precedentemente definita

# definizione della lista
fibonacci = [1,1,2,3,5,8,13,21,34,55,89,144]

# valorizzazione di una variabile
# tramite l'input digitato dall'utente
val = int(input("Digita un valore compreso nella sequenza di Fibonacci:
"))

# verifica del valore inviato in input
for x in fibonacci:
    if val == x:
        print(str(val) + " è un valore corretto.")
        break
else:
    print(str(val) + " non è un valore corretto.")
```

# IL BLOCCO IF/ELSE NEI CICLI

- In pratica, l'applicazione richiederà all'utente di digitare una cifra, quest'ultima dovrà corrispondere ad uno degli elementi presenti all'interno di una lista
- Nel caso in cui il valore inviato in input sia effettivamente corretto, allora verrà soddisfatta la condizione introdotta con if e il ciclo verrà interrotto tramite break, altrimenti verrà stampata l'alternativa associata ad else
- Per quanto riguarda invece il ciclo while è possibile presentare un caso forse ancora più interessante, cioè quello che prevede l'introduzione di un blocco else senza la necessità di una precedente condizione definita tramite if
- ciò è possibile perché in while è presente una condizione definita esplicitamente che dovrà essere vera per consentire il proseguimento delle iterazioni



# IL BLOCCO IF/ELSE NEI CICLI

```
# Uso del ciclo while per l'incremento del valore di una variabile  
# fino al superamento di un valore limite gestito tramite else
```

```
val = 5  
valore_limite = 50  
incremento = 10  
  
while val < valore_limite:  
    print(str(val) + " è compreso nel valore limite pari a " + str(valore_limite))  
    val = val + incremento  
else:  
    print(str(val) + " è superiore al valore limite pari a " + str(valore_limite))
```

# IL BLOCCO IF/ELSE NEI CICLI

- L'output prodotto dall'esecuzione dell'applicazione proposta sarà il seguente:

```
5 è compreso nel valore limite pari a 50
15 è compreso nel valore limite pari a 50
25 è compreso nel valore limite pari a 50
35 è compreso nel valore limite pari a 50
45 è compreso nel valore limite pari a 50
55 è superiore al valore limite pari a 50
```

- Sostanzialmente il ciclo while dell'applicazione dovrà incrementare una variabile precedentemente definita ("val") fino ad un valore inferiore a quello limite, nel caso in cui quest'ultimo dovesse essere superato l'alternativa introdotta da else si occuperà di lanciare un'apposita segnalazione

# TERMINARE UN CICLO CON BREAK

- Grazie agli esempi proposti in precedenza è stato possibile introdurre l'istruzione break
- Quando impiegato all'interno di un ciclo si occuperà di terminarlo in corrispondenza di un determinato evento, come per esempio il verificarsi di una condizione specifica o dell'eventualità opposta
- Sostanzialmente break rappresenta uno strumento attraverso il quale influenzare il flusso di esecuzione di un ciclo
- Per quanto riguarda la sintassi richiesta da Python per i costrutti basati su di esso, sarà necessario posizionare break subito dopo il corpo del loop, sia questo un ciclo for o un ciclo while
- nel caso in cui il progetto sviluppato preveda la definizione di cicli annidati (nested loops), sarà possibile inserire break dopo il corpo di un ciclo interno a quello iniziale
- Data la sua natura, è logico dedurre che break sia stato concepito per essere impiegato in associazione a condizioni introdotte tramite if, nel caso di un ciclo for, per esempio, potremmo utilizzare questa istruzione per arrestare il loop in corrispondenza del verificarsi di una condizione di identità



# TERMINARE UN CICLO CON BREAK

- A questo proposito è possibile analizzare l'esempio seguente, esso infatti riassume in poche righe di codice quello che potrebbe essere il ruolo di break nel terminare un flusso di iterazione nel caso in cui la condizione definita tramite if risulti vera.

# Uso dell'istruzione break per terminare un ciclo  
# al verificarsi di una condizione introdotta con if

```
for val in "python":  
    if val == "h":  
        break  
    print(val)
```

```
print("Basta così.")
```

# TERMINARE UN CICLO CON BREAK

- Eseguendo la piccola applicazione appena proposta si otterrà in output un risultato simile a quello mostrato di seguito:

```
p  
y  
t  
Basta così.
```

- In pratica il for dovrà occuparsi di ciclare la stringa passata come argomento ("python") stampando uno per uno i caratteri che la compongono
- Teoricamente il loop applicato su tale parametro dovrebbe dare luogo a sei iterazioni, tante quanti sono i caratteri che compongono la stringa, ma la condizione introdotta da if richiede che venga verificata l'identità tra il carattere iterato e "h", se questa condizione dovesse essere soddisfatta allora break bloccherà l'esecuzione del ciclo impedendo ulteriori iterazioni e verrà stampata la notifica prevista per questo caso ("Basta così.")
- Grazie all'istruzione di terminazione inserita in un costrutto condizionale avremo quindi soltanto tre iterazioni più un messaggio (chiaramente opzionale) da parte dell'applicazione

# GESTIRE LE ITERAZIONI DI UN CICLO CON CONTINUE

- Come già sottolineato, break risulta particolarmente utile quando si ha l'esigenza di terminare un ciclo in corrispondenza di una condizione precedentemente definita
- In alcuni casi però lo sviluppatore potrebbe non necessitare di un arresto completo del loop utilizzato e voler semplicemente evitare che quest'ultimo produca delle specifiche iterazioni
- A questo scopo è disponibile un'ulteriore istruzione denominata **continue**, essa in pratica arresta l'iterazione che soddisfa una determinata condizione ma permette il proseguimento del ciclo nella quale viene impiegata



# GESTIRE LE ITERAZIONI DI UN CICLO CON CONTINUE

# Uso dell'istruzione continue per arrestare un'iterazione  
# al verificarsi di una condizione introdotta con if

```
for val in "python":  
    if val == "h":  
        continue  
    print(val)  
  
print("Ciclo completato.")
```

# GESTIRE LE ITERAZIONI DI UN CICLO CON CONTINUE

- Una volta eseguita l'applicazione appena digitata si otterrà in output un risultato come quello presentato di seguito:

p  
y  
t  
o  
n

**Ciclo completato.**

- Anche in questo caso for avrà il compito di ciclare una stringa stampando in output uno alla volta tutti i caratteri da cui è composta, ciò però non avverrà se non in parte perché if introduce una condizione di identità
- Nel momento in cui il carattere iterato dovesse essere uguale ad "h" questo verrà ignorato e non parteciperà al risultato del ciclo, il loop però non verrà interrotto e verranno effettuate tutte le iterazioni successive previste

# FORZARE L'USCITA DAL CICLO

- È sempre possibile forzare l'uscita dal ciclo (qualsiasi ciclo) in ogni momento, grazie ad alcuni comandi:

Comando	Descrizione
<i>break</i>	permette di saltare fuori da un ciclo ignorando le restanti istruzioni da eseguire
<i>continue</i>	permette di saltare alla prima istruzione della prossima iterazione del ciclo
<i>else</i>	sia il ciclo for che il ciclo while hanno un costrutto aggiuntivo opzionale che permette di eseguire un blocco di istruzioni all'uscita dal ciclo, queste istruzioni vengono ignorate se usciamo dal ciclo con un break



# FORZARE L'USCITA DAL CICLO

- Come conseguenza le sintassi dei due cicli si estendono nel seguente modo:

```
for <contatore-del-ciclo> in <lista>:  
    <gruppo di istruzioni 1>  
else:  
    <gruppo di istruzioni 2> # in seguito a un break  
while <test>:  
    <gruppo di istruzioni 1>  
else:  
    <gruppo di istruzioni 2> # in seguito a un break
```

# FORZARE L'USCITA DAL CICLO

```
a = 0
b = 10
while a < b:
    a = a + 1
    print(a)
    if a == 5:
        break
else:
    print('sono uscito')
```

1 2 3 4

- Se impostiamo  $b < 5$  vediamo apparire la scritta “sono uscito”, altrimenti il break impedisce l'esecuzione del blocco else del while

# L'ISTRUZIONE PASS

- L'istruzione pass è un'operazione nulla, quando viene eseguita non succede nulla, è utile come segnaposto
- Vediamo un semplice esempio

```
for lettera in 'Ciao.Mondo':
```

```
    if lettera == '.':
```

```
        pass # non fa nulla
```

```
        print('_', end="")
```

```
        continue
```

```
    print(lettera, end="")
```

Ciao\_Mondo



# ESERCIZI

- Scrivete un programma che chieda all'utente di indovinare una password, ma che dia al giocatore solamente 3 possibilità, fallite le quali il programma terminerà, stampando “È troppo complicato per voi”
- Scrivete un programma che chieda due numeri. Se la somma dei due numeri supera 100, stampate “Numero troppo grande”
- Scrivete un programma che chieda all'utente il nome. Se viene inserito il vostro nome, il programma dovrà rispondere con un “Questo è un bel nome”, se il nome inserito è Mario Rossi o Giuseppe Verdi il programma dovrà rispondere con una battuta ;) mentre in tutti gli altri casi l'output del programma sarà un semplice “Tu hai un bel nome!”.

# ESERCIZI

- Riscrivete il programma 'area.py', definendo funzioni separate per l'area del quadrato, del rettangolo e del cerchio ( $3.14 * \text{raggio}^{**2}$ ). Il programma deve includere anche un'interfaccia a menu
- Scrivere una funzione che dica se una parola è palindroma ("Anna", "SOS")



# CORSO DI PYTHON

Dott. Antonio Giovanni Lezzi



# CORSO PYTHON

- Debug
- Funzioni
- Liste
- Tuple
- Lambda
- Funzioni ricorsive
- Set
- Dizionari
- Funzioni con dataset
- Comprehension

# DEBUGGING

- A volte capita che il programma scritto assume comportamenti differenti da quelli previsti
- Il **Debugging** è il processo grazie al quale portate il programma a svolgere le funzioni per cui è stato scritto correggendo gli errori inseriti nel codice
- La prima cosa da fare è pensare a cosa dovrebbe fare il programma se fosse corretto. Iniziate ad eseguire qualche test per vedere che cosa succede
- Ad esempio, diciamo che ho scritto un programma che calcola il perimetro di un rettangolo (la somma dei quattro lati) e svolgete diversi casi “a mano”
- Successivamente svolgete il programma inserendo i dati dei test per verificare se restituisce i risultati di cui ci si aspettava, in caso contrario si dovrà scoprire cosa sta facendo il programma analizzando passo dopo passo le istruzioni da eseguire

# FUNZIONI

- Le funzioni sono tra le caratteristiche più importanti dei linguaggi di programmazione
- Possiamo considerarle come delle scatole nere che:
  - prendono in ingresso alcuni parametri (o nessuno)
  - compiono una certa elaborazione dei parametri o modificano variabili o oggetti già definiti
  - restituiscono un risultato dell'elaborazione (anche se non sempre le funzioni in Python restituiscono valori)
- Sono quindi uno strumento utile per strutturare il codice in blocchi omogenei dal punto di vista logico al fine di migliorare la lettura e la manutenzione del sorgente



# FUNZIONI

- Inoltre sono un primo passo verso il riutilizzo del codice
- Una funzione creata con responsabilità definite e il necessario livello di genericità, può tornare utile in diversi progetti.
- E' possibile definire le funzioni come degli insiemi composti da una o più istruzioni necessarie per lo svolgimento di un determinato compito
- Uno dei vantaggi derivanti dall'utilizzo di questi costrutti riguarda il fatto che essi possono essere definiti una volta sola e poi utilizzati in più di un'occasione all'interno della medesima applicazione, ciò avverrà tramite un meccanismo denominato "chiamata alla funzione"
- In questo modo si eviteranno inutili ripetizioni di codice con un evidente risparmio di tempo in sede di sviluppo a cui si accompagna la possibilità di creare sorgenti più snelli, leggibili e performanti

# FUNZIONI

- Python prevede sia funzioni **native** del linguaggio che la possibilità di creare funzioni **personalizzate**
- A livello sintattico una funzione prevede un'intestazione costituita dalla parola chiave **def** seguita dal **nome della funzione** (che potrà essere stabilito arbitrariamente dallo sviluppatore ma dovrà essere univoco nel contesto dell'applicazione corrente) e da delle **parentesi tonde** che potranno contenere o meno dei **parametri** sui quali la funzione dovrà agire
- i parametri potranno essere uno o più di uno, nel caso di più parametri questi ultimi andranno separati con una virgola

```
# Intestazione di una funzione in Python  
def nome_funzione(eventuale_parametro, ..):
```

# FUNZIONI: DEFINIZIONE

- La sintassi per definire una funzione è molto semplice:

```
def nome_funzione(<lista parametri separati da virgola>)  
    <blocco istruzioni>  
    return <risultato>
```

- Nel caso in cui non prevediamo che la funzione restituisca dei valori in uscita, omettiamo semplicemente l'istruzione return:

```
def nome_funzione(<lista parametri separati da virgola>)  
    <blocco istruzioni>
```

- Quando non si specifica nessun valore di ritorno, la funzione ritornerà il valore None di default



# FUNZIONI: SENZA PARAMETRI

- Il caso più semplice di una funzione definita dall'utente in Python è quello nel quale non sono previsti né parametri da passare ad essa né un valore di ritorno; l'esempio seguente mostra la definizione di una funzione il cui unico compito è quello di stampare una stringa.

# Esempio di funzione priva di argomenti

# definizione della funzione

```
def stampa_stringa():  
    # documentazione della funzione  
    """La funzione stampa una stringa in output"""  
    # istruzione della funzione  
    print("Blah! Blah! Blah!")
```

# chiamata alla funzione

```
stampa_stringa()
```

- Da notare che le parentesi tonde che seguono il nome della funzione dovranno essere sempre presenti, sia in sede di definizione che di chiamata, anche nel caso in cui non dovessero essere necessari dei parametri.

# FUNZIONI CON PARAMETRI

- Riguardo a questi ultimi, essi svolgono sostanzialmente la funzione di placeholders ("segnalibri") , comunicando all'applicazione il numero di argomenti che dovranno essere passati alla funzione definita dall'utente
- Questi ultimi potranno essere valorizzati arbitrariamente dallo sviluppatore sulla base delle proprie esigenze e della tipologia di progetto implementato

# FUNZIONI CON PARAMETRI

- A questo proposito, la funzione presentata nell'esempio seguente ha il compito di concatenare due parametri ("nome" e "cognome") restituendoli in output all'interno di un stringa:

# Esempio di funzione con passaggio di parametri in Python

```
# definizione della funzione
# e dei parametri da elaborare
def saluto(nome, cognome):
    # documentazione della funzione
    """La funzione genera in output
       una stringa prodotta dalla concatenazione
       dei valori dei parametri passati ad essa"""
    # valore di ritorno
    print("Ti chiami " + nome + " " + cognome + ". Ciao!")

# chiamata alla funzione
saluto("Homer", "Simpson")
```

- Il risultato dell'esecuzione del codice proposto e della chiamata alla funzione "saluto" sarà quindi il seguente:

Ti chiami Homer Simpson. Ciao!



# FUNZIONI CON VALORI DI RITORNO

- Diverso il caso delle funzioni che prevedono un valore di ritorno, esso in pratica consente di uscire dal flusso di esecuzione della funzione in seguito alla chiamata dopo la restituzione di un dato specifico per la cui generazione è stata definita la funzione stessa
- L'esempio seguente mostra una funzione che accetta due parametri ("moltiplicando" e "moltiplicatore"), il valore di ritorno previsto non sarà altro che il risultato della moltiplicazione tra questi due argomenti che fungeranno da fattori

# FUNZIONI CON VALORI DI RITORNO

# Esempio di funzione con valore di ritorno

# definizione della funzione

# e dei parametri da elaborare

```
def prodotto(moltiplicando, moltiplicatore):
```

```
    # documentazione della funzione
```

```
    """La funzione restituisce
```

```
        il risultato di una moltiplicazione
```

```
        sulla base dei parametri passati ad essa"""
```

```
    # valore di ritorno
```

```
    return moltiplicando * moltiplicatore
```

# chiamata alla funzione

```
print ("Il prodotto della moltiplicazione è " + str(prodotto(90, 2)))
```

- Nel caso specifico, in fase di chiamata sono stati passati alla funzione i fattori "90" e "2" ottenendo come risultato il seguente output (ma sarebbe stato possibile utilizzare qualsiasi altra coppia di valori numerici):

**Il prodotto della moltiplicazione è 180**

# FUNZIONI: PASS

- La parola chiave `pass` è una sorta di commento che potrà essere utilizzato per comunicare all'esecutore di Python la mancata implementazione di un costrutto da completare in un momento successivo, come nell'esempio seguente:

```
def nome_funzione(eventuale_parametro, ..):  
    pass
```

- A differenza degli altri commenti `pass` non viene ignorato dall'engine del linguaggio e, anche se non è destinato a produrre alcun output, partecipa al flusso di esecuzione dell'applicazione senza dar luogo ad alcuna operazione e segnalando che la porzione di codice precedente non deve essere presa in considerazione perché incompleta.
- Molto utile nel caso di funzioni il cui sorgente deve essere ancora perfezionato, `pass` è comunque adottabile anche in altri contesti come per esempio quello di un ciclo:

```
for val in fibonacci:  
    pass
```



# GESTIRE GLI ARGOMENTI DELLE FUNZIONI

- Il numero degli argomenti: Quando si definisce una funzione, in Python è necessario che gli argomenti passati ad essa come parametri siano definiti a priori sia nella loro natura che nel loro numero
- Se per esempio si prevede che una funzione debba accettare due argomenti non si potrà passare ad essa un solo parametro

# GESTIRE GLI ARGOMENTI DELLE FUNZIONI

- A tal proposito è possibile proporre un semplice esempio basato su una funzione che prevede di elaborare due parametri.

```
# Passaggio di argomenti ad un funzione
# in esecuzione non verranno generati errori
# perché il numero di parametri passati
# è quello corretto
def dev(linguaggio,database):
    """Una semplice
       funzione che pone una domanda sulla base di
       due parametri"""
    print('Sviluppi applicazioni in ' + linguaggio + '? Utilizzi ' +
database + '?')

dev('Python','MySQL')
```

- Ora, si provi a non passare alcun parametro alla funzione, effettuando la chiamata in questo modo:

```
dev()
```

# GESTIRE GLI ARGOMENTI DELLE FUNZIONI

- L'interprete di Python rileverà la sintassi inattesa e notificherà la presenza di un errore nell'istruzione attraverso il seguente output:  
**TypeError: dev() missing 2 required positional arguments:  
'linguaggio' and 'database'**
- Nello stesso modo, si otterrebbe una notifica di errore effettuando nuovamente la chiamata con il passaggio di un singolo parametro:  
**dev('Python')**
- In questo secondo caso, infatti, l'interprete del linguaggio reagirebbe attraverso la seguente segnalazione indicando il nome dell'argomento mancante:  
**TypeError: dev() missing 1 required positional argument: 'database'**
- Nello sviluppo delle applicazioni Python sarà quindi necessario tener presente che le funzioni definite dagli utenti richiedono un numero fisso di argomenti



# FUNZIONI: ARGOMENTI PREDEFINITI

- Nell'esempio analizzato precedentemente sono stati passati due argomenti ad una funzione, tali parametri sono stati poi valorizzati successivamente
- Nell'esempio seguente verrà analizzato uno scenario diverso, quello in cui ad uno degli argomenti utilizzati viene assegnato un valore predefinito:

```
# Passaggio di argomenti ad un funzione
# con definizione di un valore predefinito
# per uno dei due parametri
def dev(linguaggio, database = "PostgreSQL"):
    """Per la chiamata alla
       funzione sarà sufficiente il passaggio di un solo argomento
       il primo è stato definito a priori"""
    print('Sviluppi applicazioni in ' + linguaggio + '? Utilizzi ' +
          database + '?')

dev('Python')
```

# FUNZIONI: ARGOMENTI PREDEFINITI

- Si noti come in questo caso basterà passare alla chiamata della funzione soltanto l'argomento non associato ad un valore predefinito, ma dato che tutti i parametri potranno essere valorizzati di default come nell'esempio seguente:

```
# Passaggio di argomenti ad un funzione con definizione di valori predefiniti
# per tutti i parametri previsti
def dev(linguaggio = 'Python', database = 'PostgreSQL'):
    """Per la chiamata alla
        funzione non verrà richiesto il passaggio di argomenti
        in quanto entrambi già definiti a priori"""
    print('Sviluppi applicazioni in ' + linguaggio + '? Utilizzi ' + database + '?')

dev()
```

- in questo caso non sarà necessario passare alcun argomento alla chiamata della funzione perché tutti i parametri sono stati già abbinati a dei valori di default al momento della sua definizione.

# FUNZIONI: OVERRIDE

- Un caso particolare riguardante la definizione di argomenti predefiniti riguarda la possibilità di effettuarne **l'override**, cioè di **sovrascriverne** i valori
- In pratica, come mostrato efficacemente dall'esempio proposto di seguito, i valori di default definiti dall'utente potranno essere sostituiti dinamicamente con altri all'atto della chiamata:

```
# Passaggio di argomenti ad un funzione
# con definizione di valori predefiniti
# e successivo override dei parametri
def dev(linguaggio = 'Python', database = 'PostgreSQL'):
    """Per la chiamata alla
       funzione sarà richiesto il passaggio di argomenti
       per sovrascrivere quelli definiti a priori"""
    print('Sviluppi applicazioni in ' + linguaggio + '? Utilizzi ' + database + '?')

dev('PHP', 'MongoDB')
```



# FUNZIONI: OVERRIDE

- In pratica, grazie alla sovrascrittura dinamica dei valori non si riceverà più in output il risultato inizialmente atteso:

**Sviluppi applicazioni in Python? Utilizzi PostgreSQL?**

- Ma un nuovo risultato basato sui nuovi valori associati agli argomenti che, come sarà possibile notare, rimarranno invece i medesimi

**Sviluppi applicazioni in PHP? Utilizzi MongoDB?**

- Si tenga conto del fatto che nel caso in cui ad un primo argomento sia stato attribuito un valore predefinito, anche tutti gli argomenti successivi (per inciso, quelli alla sua destra) dovranno essere valorizzati di default, altrimenti l'interprete di Python invierà una segnalazione di errore di tipo sintattico:

**non-default argument follows default argument**

# FUNZIONI: OVERRIDE

- esattamente come accadrebbe nel caso in cui si eseguisse il codice contenuto nell'esempio seguente:

```
# Passaggio di argomenti ad un funzione
# con definizione di un valore predefinito
# per il solo prametro iniziale
def dev(linguaggio = 'Python',database):
    """La chiamata alla
       funzione produrrà un errore perché gli argomenti
    successivi
       al primo non sono stati valorizzati di default"""
    print('Sviluppi applicazioni in ' + linguaggio + '?
    Utilizzi ' + database + '?')

dev('PostgreSQL')
```

# FUNZIONI: ARBITRARIETÀ DEGLI ARGOMENTI

- Python prevede che il numero degli argomenti sia fisso, esiste però una soluzione sintattica che consente di gestire i casi in cui non sia possibile stabilire a priori il numero di parametri da passare alla funzione
- Questo accorgimento si basa sull'utilizzo del carattere wildcard "\*" da inserire precedentemente ad un argomento che potrà essere sottoposto successivamente ad un ciclo di iterazione con il quale estrarre i valori definiti in fase di chiamata:

# Passaggio di un numero arbitrario di argomenti ad un funzione

```
def dev(*linguaggi):  
    """L'esecuzione alla funzione  
    determinerà l'estrazione di tutti i valori  
    associati al parametro passato come argomento  
    tramite "*" """  
  
    # ciclo per l'estrazione dei valori dall'argomento  
    for linguaggio in linguaggi:  
        print('Io sviluppo in', linguaggio)  
  
dev('Python', 'PHP', 'Java', 'JavaScript')
```



# LISTE

- In Python è possibile definire una lista, alla quale si potrà attribuire un nome, inserendo degli elementi tra parentesi quadre []
- Chi già lavora con altri linguaggi per la programmazione o lo sviluppo conoscerà sicuramente gli array, o "vettori", che sono delle variabili destinate a contenere ulteriori variabili
- sintatticamente e concettualmente le liste possono ricordare gli array, ma il loro funzionamento presenta delle peculiarità che le rendono differenti

# LISTE

- Di base una lista può esistere ma essere vuota, cioè non presentare alcun elemento al suo interno:
- *# Esempio di lista vuota*  
*nome\_lista = []*
- Nel caso in cui una lista non sia vuota, sarà necessario separare gli elementi che la compongono tramite una virgola

# LISTE

- E' possibile quindi definire liste che contengano elementi associati ad un solo tipo di dato:

```
# Esempio di lista contenente unicamente valori numerici
```

```
nome_lista = [15, 25, 35, 45, 55]
```

```
# Esempio di lista contenente unicamente delle stringhe
```

```
nome_lista = ['Homer', 'Bart', 'Lisa']
```

- Così come si potranno creare delle liste che presentino elementi di diversa natura:

```
# Esempio di lista contenente elementi di diverso tipo
```

```
# numeri e stringhe
```

```
nome_lista = [65, 'Homer', 7.3]
```

- E' infine consentita anche la creazione di liste annidate (nested list), cioè liste che hanno tra i propri elementi altre liste

```
# Esempio di lista annidata
```

```
nome_lista = ['Homer', 3.7, 10, [29, 39, 49]]
```



# LISTA DI INTERI

- **range()** è una funzione nativa di Python, precisamente un "immutable sequence type", concepita per generare automaticamente una lista sulla base di un intervallo di valori o di un valore numerico passato come argomento
- Essa si rivela particolarmente utile quando si devono definire liste formate da una grande quantità di elementi numerici
- Nel caso di un parametro espresso sotto forma intervallo si avrà che un'istruzione come la seguente:

```
# Utilizzo di range() per generare  
# una lista sulla base di un intervallo  
>>> range(1,8)
```

- porterà alla generazione di una lista composta da 7 elementi:  
`[1, 2, 3, 4, 5, 6, 7]`

# LISTA DI INTERI

- Tali elementi saranno il risultato della chiamata alla funzione `range()` che valuterà i due argomenti dell'intervallo restituendo una lista contenente tutti i valori interi a partire dal primo, incluso nella lista, fino ad arrivare al secondo, escluso invece dalla lista
- I due argomenti dell'intervallo potranno essere seguiti da un terzo argomento denominato `step`; esso specifica l'intervallo tra valori successivi, motivo per il quale se, per esempio, si volesse ottenere una lista composta dai soli numeri dispari compresi tra "1" e "8" si potrebbe operare in questo modo:

```
# Utilizzo di range() per generare  
# una lista sulla base di un intervallo  
# con step pari a 2  
>>> range(1,8,2)  
# lista generata  
# [1, 3, 5, 7]
```

# LISTA DI INTERI

- Quando invece si passa alla funzione un intero come nell'esempio proposto di seguito:

```
# Utilizzo di range() per generare  
# una lista sulla base di un intero  
>>> range(8)
```

- si avrà come risultato una lista popolata in questo modo:  
[0, 1, 2, 3, 4, 5, 6, 7]
- Gli elementi presenti saranno quindi 8, ponendo lo "0" come elemento e (non) valore iniziale



# LISTA: ACCEDERE AGLI ELEMENTI

- Di default le liste in Python prevedono che gli elementi interni vengano indicizzati numericamente; l'indicizzazione partirà da "0", per cui una lista composta da tre elementi avrà come indici "0", "1" e "2"
- Ne consegue che un elemento specifico di una lista potrà essere richiamato tramite il suo indice, come nell'esempio seguente:

```
# Accesso ad un elemento di una lista
# attraverso il suo indice

# definizione degli elementi in lista
>>> nome_lista = ['a', 'b', 'c', 'd', 'e']

# accesso all'elemento con indice "3"
>>> nome_lista[3]
```

# LISTA: ACCEDERE AGLI ELEMENTI

- L'elemento richiamato sarà in questo caso "d", cioè il quarto inserito in lista, questo perché "a" è associato all'indice "0", "b" a "1", "c" a "2" e di conseguenza "d" ha come indice "3"
- L'accesso agli elementi in lista potrebbe avvenire anche tramite indicizzazione negativa, dove l'ultimo elemento avrà come indice "-1", il penultimo "-2" e così via
- Una chiamata come la seguente avrà quindi il risultato di consentire l'accesso all'elemento "a"

```
# Accesso ad un elemento di una lista  
# attraverso il indice negativo
```

```
# definizione degli elementi in lista  
>>> nome_lista = ['a', 'b', 'c', 'd', 'e']
```

```
# accesso all'elemento con indice "-5"  
>>> nome_lista[-5]
```

# LISTA: ACCEDERE AGLI ELEMENTI

- E' inoltre possibile accedere agli elementi di una lista sulla base di un intervallo di valori; si faccia attenzione al fatto che i due componenti dell'intervallo corrisponderanno ai numeri indice, motivo per il quale un'espressione come questa:

```
# Accesso a più elementi di una lista  
# attraverso un intervallo
```

```
# definizione degli elementi in lista  
>>> nome_lista = ['a', 'b', 'c', 'd', 'e']
```

```
# accesso agli elementi con intervallo "0:2"  
>>> nome_lista[0:2]
```

- consentirà di accedere agli elementi "a", "b", cioè a quelli il cui indice va da "0" a "2" (escluso).
- Il simbolo dei due punti (":"), utilizzato in precedenza per la definizione dell'intervallo, prende il nome di **slicing operator**



# LISTA: ACCEDERE AGLI ELEMENTI

- Esso potrà essere utilizzato anche per altri scopi, come per esempio definire l'indice dal quale partire per l'accesso ai dati:

```
# Accesso a più elementi di una lista  
# a partire da un indice specifico
```

```
# definizione degli elementi in lista  
>>> nome_lista = ['a', 'b', 'c', 'd', 'e']
```

```
# accesso al primo elemento in lista ('a') tramite indice negativo  
# gli ultimi 4 elementi verranno esclusi  
>>> nome_lista[:-4]
```

```
# accesso a partire dal terzo elemento fino all'ultimo  
# 'c', 'd' ed 'e'  
>>> nome_lista[2:]
```

```
# accesso a tutti gli elementi in lista  
>>> nome_lista[:]
```

- Python dispone di costrutti che consentono di effettuare operazioni più avanzate rispetto al semplice accesso ai valori, come per esempio quelli dedicati alla manipolazione delle liste

# LISTA: GESTIONE

- Gli elementi presenti all'interno di una lista potranno essere modificati dinamicamente
- La differenza rispetto alle tuple che abbiamo queste funzionano in modo simile alle liste ma hanno la caratteristica di essere immutabili, mentre per quanto riguarda le liste si avrà una maggiore libertà nella manipolazione degli elementi assegnati
- Per modificare un elemento di una lista è necessario fare ricorso all'operatore "=", cioè lo stesso che consente di assegnare dei valori alle variabili; si ipotizzi per esempio di voler modificare un elemento, il "6", presente nella seguente lista:  

```
>>> nome_lista = [10, 9, 8, 6]
```
- In questo caso, dato che l'elemento che vogliamo modificare è associato all'indice "3", dovremo operare in questo modo:  

```
>>> nome_lista[3] = 7
```
- Fatto ciò dovremmo ottenere una nuova lista formata dai seguenti elementi:  

```
>>> nome_lista  
[10, 9, 8, 7]
```

# LISTA: GESTIONE

- In sostanza la sintassi per la sostituzione di un elemento in lista prevede l'assegnazione dell'elemento sostitutivo al numero indice della lista da modificare tramite l'operatore "="
- È chiaramente possibile anche l'operazione che permette di modificare più elementi di una lista alla volta, per cui un'istruzione come la seguente applicata alla lista ottenuta in precedenza:  

```
>>> nome_lista[1:4] = [11, 12, 13]
```
- Determinerà la sostituzione degli elementi che vanno dal secondo fino al quarto tra quelli presenti in lista e la lista risultante sarà la seguente:  

```
>>> nome_lista  
[10, 11, 12, 13]
```



# LISTA: AGGIUNGERE

- Il metodo `append()` consentirà di inserire un nuovo elemento in lista accodandolo a quelli già presenti; data l'ultima lista ottenuta tramite il precedente esempio, un'istruzione come la seguente:

```
>>> nome_lista.append(14)
```

- Porterebbe alla generazione della lista proposta di seguito:

```
>>> nome_lista  
[10, 11, 12, 13, 14]
```

# LISTA: AGGIUNGERE

- Nel caso in cui si vogliano invece aggiungere più elementi ad una lista con un'unica istruzione, sarà possibile utilizzare un altro metodo, **extend()**:

```
>>> nome_lista.extend([15, 16, 17, 18])
```

- che nel caso specifico dell'esempio mostrato permetterà di ottenere la lista:

```
>>> nome_lista
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18]
```

- Un altro caso interessante riguarda la concatenazione delle liste, operazione per la quale si farà ricorso all'apposito operatore di concatenazione simboleggiato com'è noto dal segno "+":

```
>>> nome_lista
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18]
```

```
>>> nome_lista + [19, 20, 21]
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
```

- L'operatore "\*" consentirà invece di definire liste formate da più elementi identici tra loro ripetuti un numero di volte pari al valore passato come moltiplicatore nell'istruzione:

```
>>> [Homer] * 4
```

```
['Homer', 'Homer', 'Homer', 'Homer']
```

# LISTA: AGGIUNGERE

- E' infine da citare il metodo denominato **insert()** con il quale si potrà aggiungere un elemento ad una lista nella posizione desiderata:

```
>>> nome_lista = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
>>> nome_lista.insert(0,9)
>>> nome_lista
[9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
```

- In sostanza si tratta di un metodo che accetta due argomenti, il primo è il numero indice (nel nostro caso "0") che dovrà essere assegnato al nuovo componente della lista, il secondo ("9" nell'esempio) è l'elemento destinato all'inserimento nella lista.



# LISTA:VERIFICA

- Uno sviluppatore potrebbe voler verificare che un determinato elemento sia effettivamente presente all'interno di una lista; per questo scopo è disponibile la parola chiave `in` che in sostanza effettua un confronto di tipo booleano tra il valore passato come argomento e gli elementi presenti in lista.
- Nel caso in cui il parametro dovesse trovare corrispondenza con un elemento in lista la verifica restituirà `TRUE`, altrimenti restituirà `FALSE`:

```
>>> nome_lista = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
>>> 14 in nome_lista
True # viene restituito "True" perché l'elemento è in lista
>>> 22 in nome_lista
False # viene restituito "False" perché l'elemento non è in lista
```

# LISTA: CANCELLARE

- Così come è possibile inserire nuovi elementi in una lista è anche possibile cancellarli utilizzando, in questo caso la parola chiave `del`; volendo rimuovere un elemento da una lista sarà necessario passare a `del` l'indice associato ad esso, come nell'esempio seguente dove ad essere eliminato sarà l'elemento con indice "5" (cioè quello con valore "15"):

**# Cancellazione di un singolo elemento da una lista**

```
>>> nome_lista  
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]  
>>> del nome_lista[5]  
>>> nome_lista  
[10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]
```

- Si noti come l'interprete di Python interverrà per riordinare gli indici della lista, motivo per il quale l'indice utilizzato come identificatore per la cancellazione non andrà perduto ma verrà riassegnato:

```
>>> nome_lista  
[10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]  
>>> nome_lista[5]  
16
```

# LISTA: CANCELLARE

- Nel caso in cui si desideri cancellare simultaneamente più elementi sarà necessario specificare l'intervallo all'interno del quale sono compresi, ad esempio:

# Cancellazione dei valori compresi in un intervallo

```
>>> nome_lista  
[10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21]  
>>> del nome_lista[2:6]  
>>> nome_lista  
[10, 11, 17, 18, 19, 20, 21]
```

- Se si desidera cancellare un elemento specifico senza richiamarlo attraverso il suo indice lo si potrà fare tramite il metodo `remove()` a cui passare il nome dell'elemento da eliminare:

# Rimozione di un elemento specifico da una lista

```
>>> nome_lista  
[10, 11, 17, 18, 19, 20, 21]  
>>> nome_lista.remove(17)  
>>> nome_lista  
[10, 11, 18, 19, 20, 21]
```



# LISTA: CANCELLARE

- Per ripulire interamente una lista dal suo contenuto è invece disponibile il metodo `clear()` a cui passare come argomento il nome della lista da "svuotare":

# Rimozione simultanea di tutti gli elementi di una lista

```
>>> nome_lista  
[10, 11, 18, 19, 20, 21]  
>>> nome_lista.clear()  
>>> nome_lista  
[]>
```

- Anche dopo aver perso tutti i suoi elementi la lista continuerà ad esistere come lista vuota, per eliminarla del tutto si dovrà fare ricorso nuovamente ad un'istruzione basata su `del`:

# Cancellazione definitiva di una lista

```
>>> del nome_lista  
# la chiamata alla lista produrrà un errore  
# in quanto la lista non esiste più  
>>> nome_lista  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'nome_lista' is not defined
```

# TUPLE

- Come anticipato, le **tuple** sono dei costrutti che, come accade per le liste, possono contenere una raccolta di elementi
- la differenza sostanziale tra le liste e le tuple sta però nel fatto che mentre nelle prime gli elementi che le costituiscono possono essere modificati, nelle tuple divengono immutabili
- La sintassi per la definizione delle tuple prevede che gli elementi da inserire in esse vengano delimitati tra parentesi tonde e siano associati ad un nome tramite assegnazione "="
- Sostanzialmente, così come già visto per le liste, le tuple possono esistere, e quindi essere state definite, ma essere nel contempo vuote e dunque non presentare alcun elemento al proprio interno:

```
# Esempio di tuple vuota  
nome_tuple = ()
```

# TUPLE

- Si faccia sempre attenzione alla regola che per assegnare un solo elemento ad una tuple impone che esso venga seguito da una virgola:

# Definizione di una tuple con un solo elemento

```
>>> nome_tuple = ('Homer')
```

```
>>> type(nome_tuple)
```

```
<class 'str'> # senza la virgola non viene identificata come una tuple
```

```
>>> nome_tuple = ('Homer',)
```

```
>>> type(nome_tuple)
```

```
<class 'tuple'>
```

- Nel caso in cui una tuple non sia vuota verrà richiesto di separare gli elementi che la costituiscono tramite una virgola, si potranno quindi creare tuple destinate a contenere elementi associati ad un unico tipo di dato:

# Esempio di tuple contenente unicamente valori numerici

```
nome_tuple = (16, 26, 36, 46, 56)
```

# Esempio di tuple contenente unicamente delle stringhe

```
nome_tuple = ('Homer', 'Abraham', 'Lisa')
```



# TUPLE

- Nello stesso modo si potranno definire delle tuple che presentino elementi associati a tipi di dato differenti.

```
# Esempio di tuple contenente elementi di diverso tipo  
# numeri e stringhe  
nome_tuple = (66, 'Marge', 6.1)
```

- Si potranno infine generare tuple annidate (nested tuple ), cioè tuple che hanno tra i propri elementi altre tuple

```
# Esempio di tuple annidata  
nome_tuple = ('Apu', 2.8, 15, (21, 31, 41))
```

- Se lo si desidera, sarà possibile anche inserire una lista come elemento di una tuple.

```
# annidamento con inserimento di una lista nella tuple  
nome_tuple = ('Homer', [7, 6, 5], ('a', 'b', 'c'))
```

- La sintassi delle tuple supporta anche la loro creazione senza l'utilizzo di parentesi tonde e, in questo caso, parliamo di un'operazione chiamata tuple packing:

```
# Tuple packing  
nome_tuple = 5, 8.7, 'Bart'
```

# TUPLE: INDICIZZAZIONE

- Così come già osservato per liste, anche le tuple prevedono che i loro elementi vengano indicizzati numericamente; l'indicizzazione partirà quindi da "0", motivo per il quale una tuple composta da soli tre elementi avrà come indici "0", "1" e "2".
- Un determinato elemento di una tuple potrà essere poi richiamato attraverso l'indice di riferimento:

```
# Accesso ad un elemento di una tuple
# attraverso il suo indice

# definizione degli elementi della tuple
>>> nome_tuple = ('z', 'y', 'x', 'w', 'v')

# accesso all'elemento con indice "2"
>>> nome_tuple[2]
```

- L'elemento richiamato sarà in questo caso "x", cioè il terzo inserito nella tuple, infatti "z" è associato all'indice "0", "y" a "1" e di conseguenza "x" avrà "2" come indice. Si noti come l'indice sia stato specificato tra parentesi quadre e non tonde, riprendendo la medesima sintassi delle liste

# TUPLE: INDICIZZAZIONE

- Una procedura alternativa per l'accesso agli elementi della tuple prevede di sfruttare l'indicizzazione negativa, in questo caso l'ultimo elemento avrà come indice "-1", il penultimo "-2" e così via, esattamente come per le liste. La chiamata seguente permetterà per esempio di accedere all'elemento "y"

```
# Accesso ad un elemento di una tuple  
# attraverso il indice negativo
```

```
# definizione degli elementi della tuple  
>>> nome_tuple = ('z', 'y', 'x', 'w', 'v')
```

```
# accesso all'elemento con indice "-4"  
>>> nome_tuple[-4]
```

- Si potrà inoltre accedere agli elementi di una lista definendo un intervallo di valori; le due componenti dell'intervallo corrisponderanno ai numeri indice, quindi un'espressione come quella proposta di seguito:

```
# Accesso a più elementi di una tuple  
# attraverso un intervallo
```

```
# definizione degli elementi della tuple  
>>> nome_tuple = ('z', 'y', 'x', 'w', 'v')
```

```
# accesso agli elementi con intervallo "0:3"  
>>> nome_lista[0:3]
```

- consentirà di accedere agli elementi "z", "y" e "x", cioè a quelli il cui indice va da "0" a "3" escludendo quest'ultimo.



# TUPLE: INDICIZZAZIONE

- Il simbolo dei due punti ":" utilizzato anche nelle liste per la definizione dell'intervallo, cioè il già noto slicing operator, potrà essere adottato anche per scopi differenti, per esempio con l'obiettivo di definire l'indice dal quale partire per accedere agli elementi della tuple:

```
# Accesso a più elementi di una tuple  
# a partire da un indice specifico
```

```
# definizione degli elementi della tuple  
>>> nome_tuple = ('z','y','x','w','v')  
# accesso agli ultimi due elementi della tuple ('z' e 'y') tramite indice negativo  
# i primi 3 elementi verranno esclusi  
>>> nome_tuple[: -3]  
# accesso a partire dal secondo elemento fino all'ultimo  
# 'y', 'x', 'v' e 'w'  
>>> nome_tuple[2:]  
# accesso a tutti gli elementi in lista  
>>> nome_tuple[:]
```

- Nel caso si voglia accedere ad un elemento di una tuple annidata all'interno di un'altra tuple, anche gli indici di riferimento dovranno essere annidati:

```
# Accesso ad un elemento di una tuple annidata  
>>> nome_tuple = ('Apu', 2.8, 15, (21, 31, 41))  
>>> nome_tuple[3][1]
```

- Il risultato della chiamata sarà uguale a "31", cioè il secondo elemento (con indice "1") della tuple che rappresenta a sua volta il quarto elemento, con indice "3", della tuple in cui essa è annidata.

# TUPLE: GESTIONE

- Come anticipato, le tuple si differenziano dalle liste soprattutto per via del fatto di essere immutabili, una volta assegnato un elemento ad una tupla non potrà essere modificato
- E' comunque possibile alterare alcune componenti di una tupla, se per esempio un elemento di una tupla è rappresentato da un tipo di dato modificabile, come nel caso di una lista, i suoi elementi annidati potranno subire delle variazioni
- Nello stesso modo si potranno effettuare delle riassegnazioni di valore a carico di una tupla

# TUPLE: IMMUTABILITÀ

- Nel caso in cui si tenti di modificare un elemento di una tupla, l'esecutore di Python impedirà il completamento di tale operazione gestendo l'eccezione e inviando un'apposita segnalazione all'utilizzatore.

- A questo proposito si definisca una tupla composta da elementi di natura differente:

# Definizione di una tupla

```
>>> nome_tupla = (6, 4, 5, [8, 7])
```

- Si avrà in questo modo una tupla composta da 4 elementi, 3 numeri interi e una lista a sua volta composta da 2 interi. Il secondo elemento della tupla, con indice uguale a "1" ha un valore pari a "4", si tenti ora di modificare questo valore da "4" a "3":

# Tentativo di modifica del valore di una tupla

```
>>> nome_tupla[1] = 3
```

- Una volta lanciata l'istruzione Python notificherà la presenza di un errore attraverso il seguente output:

```
>>> nome_tupla[1] = 3
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1>", line 1, in <module>
```

```
    nome_tupla[1] = 3
```

```
TypeError: 'tuple' object does not support item assignment
```

- L'ultima riga dell'output prodotto presenta un messaggio abbastanza chiaro: le tuple non supportano la riassegnazione degli elementi, viene così confermata l'immutabilità degli elementi di una tupla.



# TUPLE: MUTABILITÀ DI ELEMENTI ANNIDATI

- la tupla presentata nell'esempio precedente contiene un elemento mutabile, si tratta nello specifico di una lista che all'interno della tupla ha come indice "3" e come valori "8" e "7" che a loro volta hanno come indici "0" e "1".
- Le tuple consentono di modificare gli elementi annidati di un elemento mutabile, questo significa che se una tupla presenta al suo interno una lista, gli elementi di quest'ultima potranno subire dei riassegnamenti.
- Si provi quindi a modificare il valore dell'elemento della lista che ha come indice "0" da "8" a "2":

**# Riassegnazione di un elemento di una lista in una tupla**

```
>>> nome_tupla[3][0] = 2
```

- Questa volta non vi sarà alcuna notifica di errore, l'avvenuta riassegnazione potrà infatti essere confermata dalla chiamata dell'elemento riassegnato tramite il suo numero d'indice:

**# Verifica del valore dell'elemento di una lista**

```
>>> nome_tupla[3][0]
```

```
2
```

# TUPLE: RIASSEGNAZIONE

- Il nome utilizzato per definire una tupla agisce come una sorta di puntatore verso degli elementi immutabili, questo però non significa che essa non possa essere riposizionata per puntare su un nuovo gruppo di valori. Sarà infatti possibile riassegnare una tupla nel suo complesso associando ad essa valori differenti da quelli iniziali.

- Nell'esempio seguente la tupla precedentemente definita viene riassegnata utilizzando nuovi elementi:

```
# Riassegnazione di una tupla
>>> nome_tuple = ('p','y','t','h','o','n')
>>> nome_tuple
('p', 'y', 't', 'h', 'o', 'n')
```

- In questo caso l'istruzione lanciata non darà luogo ad errori, in quanto non si è tentato di assegnare nuovi valori agli elementi che compongono la tupla, per definizione immutabili, ma la tupla stessa che è stata inizializzata nuovamente.

# TUPLE: CONCATENAZIONE

- Più tuple possono essere concatenate utilizzando l'operatore "+" che agisce come operatore di concatenazione:

## # Concatenazione di due tuple

```
>>> a_tuple = (1,2,3)
>>> b_tuple = (4,5,6)
>>> c_tuple = a_tuple + b_tuple
>>> c_tuple
(1, 2, 3, 4, 5, 6)
```

- Il numero di tuple concatenabili è virtualmente illimitato, sarà inoltre possibile concatenare tuple i cui elementi appartengono a tipi di dato differenti:

## # Concatenare tuple con tipi di dati differenti

```
>>> d_tuple = ('a','b','c',[7,8])
>>> e_tuple = c_tuple + d_tuple
>>> e_tuple
(1, 2, 3, 4, 5, 6, 'a', 'b', 'c', [7, 8])
```



# TUPLE: CONCATENAZIONE

- L'interprete di Python si occuperà di distribuire gli indici interni alla tupla risultante sulla base dell'ordine con il quale le tuple sono state concatenate. Nel caso dell'ultima concatenazione effettuata abbiamo una tupla composta da 10 elementi con numero indice da "0" a "9", dato che l'ultimo elemento è una lista composta da due elementi con indice da "0" a "1" avremo per esempio:

```
>>> e_tuple[9][0]  
7
```

- Un caso particolare è quello della concatenazione di valori ripetuti che si basa sull'utilizzo dell'operatore "\*":

```
# Concatenazione di valori ripetuti  
>>> f_tuple = ('x', 'y', '7') * 3  
>>> f_tuple  
( 'x', 'y', '7', 'x', 'y', '7', 'x', 'y', '7')
```

- In questo modo sarà possibile specificare il numero di volte che uno o più elementi dovranno essere ripetuti all'interno di una tupla tramite una semplice moltiplicazione.

# TUPLE E RIMOZIONE

- Gli elementi che compongono una tupla sono immutabili e non possono essere cancellati, nel caso in cui si tenti di cancellare un elemento di una tupla attraverso la chiamata del suo numero indice si riceverà in output una notifica di errore; è invece possibile utilizzare la keyword **del** per cancellare una tupla nel suo insieme:

```
# Tentativo di cancellazione di un elemento di una tupla
```

```
>>> del f_tuple[5]
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#22>", line 1, in <module>
```

```
    del f_tuple[5]
```

```
TypeError: 'tuple' object doesn't support item deletion
```

```
# Cancellazione di una tupla
```

```
>>> del f_tuple
```

```
>>> f_tuple
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#24>", line 1, in <module>
```

```
    f_tuple
```

```
NameError: name 'f_tuple' is not defined
```

- Se provassimo a cancellare un elemento di una tupla Python ci informerebbe che tale funzionalità non è supportata, la rimozione dell'intera tupla avverrà invece regolarmente e non sarà quindi più possibile richiamare i valori di quest'ultima perché non più definita.

# FUNZIONI PER LISTE E TUUPLE

- In Python le funzioni native per le liste e le tuple sono le stesse, presentano gli stessi nomi e la stessa sintassi e prevedono gli stessi valori di ritorno
- le semplici funzioni **all()** e **any()** sono funzioni utili per verificare il contenuto di liste e tuple.
- **all()**: passando come parametro alla prima una lista, la funzione restituirà TRUE se tutti gli elementi sono TRUE oppure se la lista è vuota, cioè priva di elementi
- **any()**: restituirà TRUE se un qualsiasi elemento di una lista è TRUE o se la lista non è vuota.



# FUNZIONI PER LISTE E TUPLA

- **enumerate()** restituisce un oggetto che contiene coppie composte da indici e valori di tutti gli elementi che compongono una lista o una tupla:

# Uso della funzione enumerate()

```
l = [2, 5, 9, 10, 0]
for i in enumerate(l):
    print(i)
>>> (0, 2)
>>> (1, 5)
>>> (2, 9)
>>> (3, 10)
>>> (4, 0)
```

- Nell'esempio è stata creata una lista ed in seguito fatto il ciclo su tutti gli elementi generato dal metodo enumerate() al fine di ottenere in output tutte le coppie di indici e valori degli elementi

# FUNZIONI PER LISTE E TUPLA

- **len()** è una funzione che restituisce in output la lunghezza di una lista o di una tupla, si intende il numero di elementi presenti in una lista o in una tupla

# Contare gli elementi di una lista o di una tupla

```
>>> b_lista = ['a', 'b', 'c', 'e', 'f']
```

```
>>> len(b_lista)
```

```
5
```

```
>>> b_tuple = (1, 2, 3)
```

```
>>> len(b_tuple)
```

```
3
```

# FUNZIONI PER LISTE E TUPLA

- **max()** e **min()** sono invece due funzioni utili per ottenere come valore di ritorno rispettivamente il più grande e il più piccolo valore di una lista o di una tupla

# Ottenere il più grande e il più piccolo valore di una lista o di una tupla

```
>>> max(b_lista)
```

```
'f'
```

```
>>> min(b_lista)
```

```
'a'
```

```
>>> max(b_tuple)
```

```
3
```

```
>>> min(b_tuple)
```

```
1
```

- Nel caso di valori numerici, l'esecutore procede effettuando dei semplici confronti basati sul valore delle cifre associate agli elementi, nel caso di stringhe queste vengono considerate in ordine crescente, motivo per il quale la prima lettera dell'alfabeto risulta essere inferiore rispetto alla seconda, la seconda alla terza e così via.
- Liste o tuple composte da valori numerici e stringhe vengono considerate come formate da tipi di dato non ordinabili e, quando passate come argomento a `max()` e `min()`, l'esecutore produrrà un errore.



# FUNZIONI PER LISTE E TUPLA

- Nel caso in cui invece dei valori numerici vengano inseriti utilizzando la sintassi delle stringhe, cioè tra apici, l'interprete di Python sarà in grado di individuarne il valore e di estrarre gli elementi con maggiore e minore valore, riportando comunque il tipo di dato corretto:

```
# Ottenere il più grande e il più piccolo valore di una lista o di una tupla
```

```
# indipendentemente dal tipo di dato utilizzato
```

```
>>> c_lista = ['1', '4', '3', '7', '5', '2']
```

```
>>> max(c_lista)
```

```
'7'
```

```
>>> min(c_lista)
```

```
'1'
```

```
>>> c_tuple = (1, 4, 3, 7, 5, 2)
```

```
>>> max(c_tuple)
```

```
7
```

```
>>> min(c_tuple)
```

```
1
```

# FUNZIONI PER LISTE E TUPLA

- **sorted()** permette di riordinare gli elementi di una lista o di una tupla
- nel caso si tratti di elementi alfanumerici viene utilizzato un criterio alfabetico come avverrebbe in un qualsiasi vocabolario
- i valori numerici verranno invece riposizionati a partire dall'elemento di minor valore fino al maggiore utilizzando quindi l'ordine crescente

# Riordinare gli elementi di una lista o di una tupla

```
>>> d_lista = ['a', 'c', 'b', 'f', 'e', 'd']
```

```
>>> sorted(d_lista)
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> d_tuple = (4, 2, 12, 6, 10, 8)
```

```
>>> sorted(d_tuple)
```

```
[2, 4, 6, 8, 10, 12]
```

# FUNZIONI PER LISTE E TUPLA

- La funzione **sum()** viene utilizzata per sommare tutti i valori degli elementi numerici presenti in una lista o in una tupla
- tali elementi non potranno essere specificati adottando la sintassi delle stringhe (tra apici) o verrà generato un errore in output ("unsupported operand type(s).")

# Sommare gli elementi di una lista o di una tupla

```
>>> f_tuple = (100,200,300)
```

```
>>> sum(f_tuple)
```

```
600
```

```
>>> f_lista = [299,399,499]
```

```
>>> sum(f_lista)
```

```
1197
```

```
>>> g_lista = ['11','11','11']
```

```
>>> sum(g_lista)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#44>", line 1, in <module>
```

```
sum(g_lista)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> g_tuple = ('22','22','22')
```

```
>>> sum(g_tuple)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#47>", line 1, in <module>
```

```
sum(g_tuple)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



# FUNZIONI PER LISTE E TUPLA

- **list()** è una funzione in grado di convertire un oggetto iterabile, ad esempio una tupla, in una lista; a tal proposito si analizzi il seguente esempio:

```
# Convertire una tupla in una lista
```

```
>>> h_tuple = ('uno', 'due', 'tre')
```

```
>>> h_lista = list(h_tuple)
```

```
>>> h_lista[1] = 'quattro'
```

```
>>> h_lista
```

```
['uno', 'quattro', 'tre']
```

- Nel codice proposto viene definita una tupla i cui elementi sono per definizione immutabili, convertendo la tupla in una lista tramite `list()` è stato però possibile modificare il valore di un elemento.

# FUNZIONI PER LISTE E TUPLA

- Se vogliamo convertire la lista in una tupla utilizzando l'apposita funzione `tuple()`, gli elementi tornerebbero ad essere immutabili:

```
# Convertire una lista in una tupla
```

```
>>> i_tuple = tuple(h_lista)
```

```
>>> i_tuple[1] = 'cinque'
```

```
Traceback (most recent call last):
```

```
File "<pyshell#7>", line 1, in <module>
```

```
i_tuple[1] = 'cinque'
```

```
TypeError: 'tuple' object does not support item  
assignment
```

# FUNZIONI: ARBITRARIETÀ DEGLI ARGOMENTI

- Grazie all'adozione di "\*" il parametro passato alla funzione verrà convertito in una tuple, cioè un costrutto per alcune versi simile ad una lista i cui elementi non potranno essere modificati una volta definiti
- Tuple e liste sono due datatypes di Python che verranno approfonditi ulteriormente a breve



# FUNZIONI

- Dopo l'intestazione, la cui fine verrà marcata tramite il simbolo dei due punti (":") è possibile documentare la funzione definita tramite un particolare tipo di commento (opzionale) denominato docstring
- Esso dovrà essere delimitato da tre apici all'inizio e alla fine del testo, rispetterà la stessa indentazione delle istruzioni in cui è inserito e non parteciperà all'esecuzione della funzione, motivo per il quale sarà visibile soltanto a livello di sorgente:

```
"""Il funzionamento di questa funzione è stato documentato  
tramite docstring"""
```

# LAMBDA

- In Python le funzioni Lambda sono dei particolari costrutti sintattici derivati dal linguaggio Lisp e chiamati anche funzioni anonime
- Rispetto alle comuni funzioni definite dall'utente esse non sono associate ad un nome, da qui la caratteristica di essere "**anonime**", non vengono introdotte dalla parola chiave def, prevedendo invece la keyword **lambda**, e possono essere seguite soltanto da un'unica espressione
- Una delle caratteristiche della funzioni Lambda risiede nel fatto che esse non sono assolutamente necessarie
- Infatti qualsiasi risultato possa essere ottenuto con una funzione anonima sarà raggiungibile anche tramite una funzione definita dall'utente, ma **possono diventare estremamente comode** per la risoluzione di piccoli problemi ma sostanzialmente esse **migliorano la produttività del coding** in quanto possono essere definite "al volo", in corso d'opera, senza influire pesantemente sulla struttura di un'applicazione

# LAMBDA

- Le funzioni Lambda accettano un numero indefinito di argomenti ma, come anticipato, supportano una sola espressione; quest'ultima verrà elaborata dall'interprete di Python ai fini della generazione dell'output
- Nell'esempio seguente viene proposta una semplice funzione anonima finalizzata alla divisione del valore di una variabile ("dividendo") per un divisore dal valore costante ("5" nel caso specifico).

```
# Esempio di funzione Lambda  
# impiegata per un'operazione matematica
```

```
dividi = lambda dividendo: dividendo / 5  
  
print(dividi(10))
```

- In sostanza, nel codice il compito della funzione Lambda è quello di assegnare all'identificatore "dividi" (che non va confuso con il nome della funzione in quanto anonima) l'oggetto restituito dalla funzione, cioè il risultato dell'elaborazione dell'espressione



# LAMBDA

- Come sostenuto in precedenza, per qualsiasi funzione Lambda è possibile creare una corrispondente funzione definita dall'utente e neanche il caso proposto in precedenza farà eccezione a questa regola:

```
# Esempio di funzione definita dall'utente  
# impiegata per un'operazione matematica
```

```
def dividi(dividendo):  
    return dividendo / 5
```

```
print(dividi(10))
```

- Si noterà come, in pratica, mentre nelle funzioni definite dall'utente la chiamata alla funzione avviene tramite il loro nome, in quelle anonime viene utilizzato invece l'identificatore; detto questo però le due procedure per la chiamata risultano identiche dal punto di vista sintattico

# FUNZIONI RICORSIVE

- Il concetto della "**ricorsione**" è collegato a quello della ricorrenza, nel caso delle funzioni è possibile che una funzione richiami un'altra funzione che, a suo volta, richiamerà indirettamente la prima, ma è anche possibile che una funzione richiami direttamente se stessa
- In Python tale casistica rientra nella tipologia delle **funzioni ricorsive**, cioè funzioni per l'auto-chiamata
- A livello pratico le funzioni ricorsive presentano il vantaggio di consentire la suddivisione di un problema complesso in problemi di minore entità e quindi più facilmente risolvibili
- In alcuni casi esse mettono al riparo dalla necessità di digitare sorgenti estremamente articolati, magari particolarmente ricchi di annidamenti dovuti all'esigenza di definire lunghe procedure sequenziali, a tutto vantaggio della produttività e della leggibilità del codice.

# FUNZIONI RICORSIVE

- Nell'esempio seguente verrà presentata una semplice funzione ricorsiva inserita in un'applicazione interattiva il cui scopo è quello di ottenere il fattoriale di un numero, cioè il prodotto di quel numero per tutti i suoi antecedenti; "24" è per esempio il fattoriale di "4" in quanto risultato dell'espressione "1x2x3x4".
- Essa in sostanza eviterà allo sviluppatore di dover specificare tutte le permutazioni necessarie per ottenere lo stesso risultato, cosa non particolarmente sensata quando ci si trova a gestire un caso come quello proposto di seguito, dove l'input al programma, cioè il valore del quale si dovrà ottenere il fattoriale, verrà stabilito arbitrariamente dall'utilizzatore

```
# Utilizzo delle funzioni ricorsive
# per il calcolo del fattoriale di un numero

def fattoriale(n):
    """Il fattoriale di un numero indica il prodotto di
       quel numero per tutti i suoi antecedenti"""

    if n == 1:
        return 1
    else:
        return (n * fattoriale(n-1))

res = int(input("Inserisci un numero: "))
if res >= 1:
    print("Il fattoriale di", res, "è", fattoriale(res))
```



# FUNZIONI RICORSIVE

- Fattore necessario per il funzionamento delle funzioni ricorsive è l'esistenza di una condizione per la terminazione delle ricorrenze
- Tale condizione farà in modo che in un determinato momento la funzione non necessiti di ulteriori ricorrenze, avendo in sostanza esaurito il suo compito
- Nell'esempio proposto, dato che il valore di "n" subisce un decremento ad ogni ricorrenza ed è previsto un controllo nel caso in cui "n" sia uguale ad "1", la funzione troverà la sua condizione di terminazione al raggiungimento di quest'ultimo

# ESERCIZI

- Riscrivete il programma 'area.py', definendo funzioni separate per l'area del quadrato, del rettangolo e del cerchio ( $3.14 * \text{raggio}^{**2}$ ). Il programma deve includere anche un'interfaccia a menu
- Scrivere una funzione che dica se una parola è palindroma ("Anna", "SOS")
- Scrivere una funzione che dica se un numero è primo o no
- Scrivere una funzione che produce un istogramma esempio Istogramma([4,2,6]):  
\*\*\*\*  
\*\*  
\*\*\*\*\*
- Scrivere una funzione che produca una lista dei numeri di Fibonacci fino ad n (una funzione implementato senza ricorsione e una funzione con ricorsione)
- Scrivere una funzione che restituisce la media dei numeri passati come lista e/o come tupla

# ESERCIZI

```
def square(length):  
    return length * length  
  
def rectangle(width , height):  
    return width * height  
  
def circle(radius):  
    return 3.14 * radius ** 2  
  
def options():  
    print("Scegli:")  
    print("q) area del quadrato")  
    print("c) area del cerchio")  
    print("r) area del rettangolo")  
    print("e) quit")  
    print  
  
choice = "x"  
options()  
while choice != "e":  
    choice = input("Inserisci una scelta: ")  
    if choice == "q":  
        length = input("Lato del quadrato: ")  
        print ("l'area è ", square(length))  
        options()  
  
    elif choice == "c":  
        radius = input("Raggio della circonferenza: ")  
        print ("L'area è ", circle(radius))  
        options()  
  
    elif choice == "r":  
        width = input("Inserisci la larghezza: ")  
        height = (input("Inserisci l'altezza: "))  
        print ("l'area è ", rectangle(width, height))  
        options()  
  
    elif choice == "e":  
        print ("")  
  
    else:  
        print ("Ritenta sarai più fortunato")  
        options()
```



# ESERCIZI

```
def palindroma(str):  
    revstr = reversed(str)  
  
    if list(str) == list(revstr):  
        return True  
  
    return False  
  
def palindroma2(string):  
    result = True  
    str_len = len(string)  
    for i in range(0, int(str_len/2)):  
        if string[i] != string[str_len-i-1]:  
            result = False  
            break  
    return result
```

# ESERCIZI

```
def palindroma3(n):  
    return (n == n[::-1])  
  
def palindromaRic(string) :  
    if len(string) <= 1 :  
        return True  
    if string[0] == string[len(string) - 1] :  
        return palindromaRic(string[1:len(string) - 1])  
    else :  
        return False  
  
parola = input("Inserisci una parola:")  
print(parola[::-1])  
  
res = palindroma(parola)  
  
if res == True:  
    # check if the string is equal to its reverse  
    print("La parola è palindroma")  
else:  
    print("La parola non è palindroma")
```

```
def fib(n, a=0, b=1):  
    l = [a, b]  
    for i in range(2, n):  
        a, b = b, a + b  
        l.append(a)  
    return l
```

```
def fibonacci(n):  
    assert n >= 0  
    if n < 2:  
        return n  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```



```
def isPrime(n):  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True  
  
def primes1(n):  
    primes = [p for p in range(2,n) if  
isPrime(p)]  
    return primes  
  
print(primes1(100))
```

```
def histogram(lista):  
    for x in lista:  
        for i in range(0, x):  
            print("*", end="")  
        print("")
```

```
histogram([5,2,6])
```

```
def media(*l):  
    sum = 0  
    tot = len(l)  
    for i in range(0, tot):  
        sum += l[i]  
    mediaNum = sum/tot  
    return mediaNum  
  
print("media:", str(media(10, 30, 40, 4)))
```



# SET

- I set sono delle entità simili alle liste e alle tuple
- sono definibili come delle raccolte non ordinate di elementi in cui ciascuno di essi è unico e immutabile
- Un set è composto da elementi non duplicati che non possono essere modificati, è invece possibile modificare i set, per esempio aggiungendo o rimuovendo elementi

# SET: DEFINIZIONE

- La sintassi dei set devono esser definiti attribuendo loro un nome e associando ad essi degli elementi che dovranno essere indicati tra parentesi graffe nonché separati da virgole:

```
# Definizione di un set
```

```
>>> nome_set = {'a', 'b', 'c'}
```

- Si potranno definire set composti da elementi di tipo stringa così come set composti da elementi di tipo numerico o formati da elementi associati a diversi tipi di dato
- l'esempio seguente mostra la definizione di un set in cui sono presenti due stringhe, un intero e una tupla formata da interi:

```
# Definizione di un set composto
```

```
# da elementi di tipo di dato differente
```

```
>>> nome_set = {'homer', 120, 'lisa', (3, 5, 6)}
```

# SET

- Se è possibile utilizzare una tupla come elemento di un set lo stesso non potrà essere fatto con una lista, questo perché le liste contengono elementi modificabili; il tentativo di definire un set contenente una lista porterà alla generazione di un errore:

```
# Errore generato dall'utilizzo di elementi
>>> # modificabili in un set
>>> nome_set = {'homer', 120, 'lisa', [3, 5, 6]}
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    nome_set = {'homer', 120, 'lisa', [3, 5, 6]}
TypeError: unhashable type: 'list'
```

- Nel caso in cui invece si tenti di inserire nel set un elemento duplicato, questo verrà semplicemente reso unico dall'engine di Python che si occuperà di rimuovere dal set i doppi.

```
# Rimozione automatica dei doppi da un set
>>> nome_set = {'homer', 120, 'lisa', 'homer', (3, 5, 6)}
>>> nome_set
{120, (3, 5, 6), 'homer', 'lisa'}
```



# SET

- La funzione nativa `set()` offre una modalità alternativa per la definizione dei set, essa infatti consente di creare set accettando come argomenti gli elementi che ne faranno parte; questi ultimi però dovranno essere passati alla funzione sotto forma di un unico elemento iterabile come per esempio una tupla, questo perché `set()` ammette un solo parametro alla volta:

# Definire un set con `set()` da una tupla

```
>>> nome_set = set((1,2,3,9,5))
```

```
>>> nome_set
```

```
{1, 2, 3, 9, 5}
```

- Uno dei vantaggi derivanti dall'utilizzo di `set()` sta nel fatto che essa permette di creare un set anche a partire da un parametro iterabile modificabile come per esempio una lista:

# Definire un set con `set()` da una lista

```
>>> nome_set = set([1,2,3,9,5])
```

```
>>> nome_set
```

```
{1, 2, 3, 9, 5}
```

# SET

- Chiaramente, una volta entrati a far parte del set gli elementi della lista diventeranno immutabili
- `set()` è inoltre utilizzabile per la creazione di set vuoti da popolare eventualmente in un momento successivo:

`# Creazione di un set vuoto con set()`

```
>>> nome_set = set()
```

- E' importante tenere presente che `set()` interpreta l'argomento che gli viene passato come un gruppo di elementi, questo vuol dire che una stringa composta da più caratteri verrà suddivisa per tutti gli elementi che la compongono:

```
>>> nome_set = set('homer')
```

```
>>> nome_set
```

```
{ 'o', 'h', 'r', 'm', 'e' }
```

# SET: INSERIMENTO NUOVI ELEMENTI

- Come anticipato, i set sono delle raccolte "non ordinate", questo significa che a differenza delle liste e delle tuple i loro elementi non vengono indicizzati e non sarà possibile utilizzare un numero indice per richiamarli; sostanzialmente una sintassi come la seguente, valida per le liste e le tuple, non ha alcun senso per i set:

```
# Definizione di un set
>>> nome_set = {2,4,6,8}
# Tentativo di accesso ad un elemento di un set
# attraverso un indice
>>> nome_set[1]
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    nome_set[1]
TypeError: 'set' object does not support indexing
```

- Se si desidera aggiungere un nuovo elemento ad un set è possibile farlo tramite il metodo **add()**:

```
# Aggiungere un elemento ad un set
>>> nome_set = {100,300,400}
>>> nome_set
{400, 300, 100}
>>> nome_set.add(200)
>>> nome_set
{400, 200, 300, 100}
```



# SET: AGGIORNAMENTO

- `add()` permette però di aggiungere un solo elemento alla volta in un set, per l'aggiunta di più elementi bisogna invece ricorrere al metodo **`update()`** che consentirà il nuovo inserimento tramite tuple, liste o altre raccolte di elementi:

# Aggiungere più elementi ad un set

```
>>> nome_set.update([500,600,700])
```

```
>>> nome_set
```

```
{7, 8, 9, 500, 600, 700}
```

- `update()` è utilizzabile anche per l'inserimento di set all'interno di altri set, come nell'esempio seguente dove al set originali vengono aggiunte una lista e un set:

```
>>> nome_set.update([800,900], {1000,2000,3000})
```

```
>>> nome_set
```

```
{800, 900, 7, 8, 9, 1000, 2000, 500, 600, 3000, 700}
```

# SET: RIMOZIONE

- Python dispone di diversi metodi per la rimozione di elementi dai set, uno di questi è **discard()** al quale basterà passare come parametro l'elemento da rimuovere:

```
# Rimozione di un elemento da un set con discard()
>>> nome_set = {'c', 'i', 'a', 'o'}
>>> nome_set.discard('i')
>>> nome_set
{'o', 'c', 'a'}
```

- In alternativa sarà possibile utilizzare il metodo **remove()**:

```
# Rimozione di un elemento da un set con remove()
>>> nome_set.remove('c')
>>> nome_set
{'o', 'a'}
```

- `discard()` e `remove()` si differenziano per il fatto che il secondo genera un errore nel caso in cui l'elemento che si tenta di eliminare non sia presente nel set, il primo invece lascia semplicemente la situazione immutata.

# SET

- Particolare il caso del metodo `pop()` che quando richiamato rimuove un qualsiasi elemento del set scelto arbitrariamente dallo stesso interprete di Python, senza che l'utilizzatore possa prevedere o decidere quale debba essere:

# Rimozione di un elemento da un set con `pop()`

```
>>> nome_set = {'c', 'i', 'a', 'o'}
```

```
>>> nome_set.pop()
```

```
'o'
```

```
>>> nome_set
```

```
{'i', 'c', 'a'}
```

- Infine, per ripulire completamente un set dai suoi elementi si utilizza il metodo `clear()`:

# Rimozione di tutti gli elementi di un set

```
>>> nome_set.clear()
```

```
>>> nome_set
```

```
set()
```



# SET: OPERARE

- Le funzioni disponibili in Python per i set sono in linea di massima le stesse descritte in precedenza per le liste e le tuple, motivo per il quale non sarà necessario ripeterle
- Più interessante è analizzare alcuni utilizzi pratici dei set che si prestano in modo particolare ad operazioni di insiemistica come le unioni, le intersezioni, le differenze e le differenze simmetriche
- Considerati due set, un'**unione** sarà a sua volta un set composto dagli elementi di entrambi i set. Per determinare un'unione è possibile utilizzare l'operatore "|" in questo modo:

```
# Creare l'unione di due set
>>> a_set = {'a', 'b', 'c'}
>>> b_set = {'d', 'e', 'f'}
>>> c_set = a_set | b_set
>>> c_set
{'b', 'd', 'a', 'c', 'f', 'e'}
```

# SET: OPERARE

- In alternativa è possibile ottenere il medesimo risultato ricorrendo al metodo **union()** utilizzabile in questo modo:

# Unione di due set con union()

```
>>> a_set.union(b_set)
{'b', 'd', 'a', 'c', 'f', 'e'}
>>> b_set.union(a_set)
{'b', 'd', 'a', 'c', 'f', 'e'}
```

- In ogni caso verrà generato un set contenente gli elementi di entrambi i set coinvolti
- Nel caso in cui due set dovessero contenere elementi comuni, questi verranno inseriti una volta sola nel set risultante:

# Unione di set con elementi comuni

```
>>> a_set = {'a', 'b', 'c', 'd'}
>>> b_set = {'d', 'e', 'f'}
>>> a_set | b_set
{'b', 'd', 'a', 'c', 'f', 'e'}
```

# SET: OPERARE

- Le **intersezioni** permettono di accedere agli elementi comuni di due set, in sostanza un'intersezione permette di generare un set composto dagli elementi in comune dei set coinvolti nell'operazione
- Un primo modo di eseguire un'intersezione è quello che prevede l'utilizzo dell'operatore "&":

# Generare l'intersezione di due set

```
>>> a_set = {'2', '5', '7', '9'}  
>>> b_set = {'3', '9', '2', '8'}  
>>> a_set & b_set  
{'2', '9'}
```



# SET: OPERARE

- Anche in questo caso disponiamo di una modalità alternativa basata sull'impiego di un metodo, **intersection()**, che se applicato ai due set precedentemente definiti, permetterà di ottenere lo stesso risultato di "&":

```
# Generare l'intersezione di due set con intersection()
```

```
a_set.intersection(b_set)
```

```
{'2', '9'}
```

```
>>> b_set.intersection(a_set)
```

```
{'9', '2'}
```

- L'unica differenza riguarda la disposizione degli elementi che cambierà a seconda di quale set verrà scelto per la chiamata al metodo ("set.metodo") e quale verrà utilizzato come parametro per il passaggio al metodo (tra parentesi tonde).

# SET: DIFFERENZE

- Per spiegare come ottenere **difference** tra set si parta immediatamente da un esempio pratico: dati due set "x" ed "y", "x-y" permetterà di generare un set di elementi presenti in "x" ma non in "y", mentre "y-x" consentirà di ottenere un set di elementi presenti in "y" ma non in "x".
- Nel caso delle differenze l'operatore di riferimento è quello di sottrazione ("-") utilizzato nel modo seguente:

## # Differenza tra set

```
>>> x_set = {1,2,3,4,5,6}
>>> y_set = {4,5,6,7,8,9}
>>> x_set - y_set
{1, 2, 3}
>>> y_set - x_set
{8, 9, 7}
```

- Il metodo di riferimento è invece **difference()** che permette di ottenere lo stesso risultato di "-":

## # Differenza tra set con difference()

```
>>> x_set.difference(y_set)
{1, 2, 3}
>>> y_set.difference(x_set)
{8, 9, 7}
```

# SET: DIFFERENZE SIMMETRICHE

- Le  **differenze simmetriche**  tra set rappresentano l'operazione contraria rispetto alle semplici differenze: dati due set "x" ed "y", la loro differenza simmetrica" permetterà di generare un set di elementi presenti sia in "x" che "y" escludendo quelli non in comune tra i due.
- Per operare una differenza simmetrica si deve utilizzare l'operatore "^" dopo aver definito i due set che fungeranno da operandi:

# Differenza simmetrica tra due set

```
>>> x_set = {1,2,3,4,5}
>>> y_set = {4,5,6,7,8}
>>> x_set ^ y_set
{1, 2, 3, 7, 8, 9}
```

- Se per le differenze disponiamo del metodo **difference()** per le differenze simmetriche abbiamo l'operatore **symmetric\_difference()** che permette di ottenere gli stessi risultati di "^" attraverso la seguente sintassi:

# Differenza simmetrica tra set con symmetric\_difference()

```
>>> x_set.symmetric_difference(y_set)
{1, 2, 3, 7, 8, 9}
```



# SET: FROZENSET

- I **frozenset** sono un particolare tipo di set, o per meglio dire dei costrutti molto simili ai set, che hanno la caratteristica di non supportare le operazioni di riassegnazione
- In sostanza i frozenset sono per i set quello che le tuple rappresentano per le liste. In nessun caso sarà possibile aggiungere o rimuovere un elemento da un frozenset.
- Per definire un frozenset si utilizza la funzione **frozenset()** a cui passare degli elementi attraverso, per esempio, una lista.

# Definizione di frozenset con frozenset()

```
>>> x_frozenset = frozenset([1, 2, 3, 4, 5])
```

```
>>> y_frozenset = frozenset([3, 4, 5, 6, 7])
```

# SET: FROZENSET

- I frozenset supportano tutte le operazioni per l'insiemistica precedentemente descritte per i set:

```
# Unione tra frozenset
>>> x_frozenset | y_frozenset
frozenset({1, 2, 3, 4, 5, 6, 7})
# Intersezione tra frozenset
>>> x_frozenset & y_frozenset
frozenset({3, 4, 5})
# Differenza tra frozenset
>>> x_frozenset - y_frozenset
frozenset({1, 2})
# Differenza simmetrica tra frozenset
>>> x_frozenset ^ y_frozenset
frozenset({1, 2, 6, 7})
```

- Per ovvi motivi non sono disponibili metodi per aggiungere o rimuovere elementi dai frozenset:

```
>>> x_frozenset.add(6)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    x_frozenset.add(6)
AttributeError: 'frozenset' object has no attribute 'add'
>>> y_frozenset.remove(6)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    y_frozenset.remove(6)
AttributeError: 'frozenset' object has no attribute 'remove'
```

# DIZIONARI

- In Python i **dizionari** (**dictionary**), o semplicemente dict, sono delle raccolte non ordinate di oggetti che appartengono ai cosiddetti tipi di dato compound (o container) esattamente come le liste, le tuple e i set, ma differenza di questi i dizionari non presentano come elementi dei semplici dei valori, bensì delle coppie composte da chiavi e valori.
- Tale caratteristica è dovuta al fatto che i dizionari sono stati concepiti per reperire facilmente dei valori sulla base delle loro chiavi quando esse sono note allo sviluppatore.
- Dal punto di vista computazionale è vantaggioso utilizzare i dizionari perché sono delle tabelle hash. Come il tipo hash in Perl, le istanze di Hashtable in Java o C#, le mappe MFC per Visual C++, etc
- La sintassi per la creazione dei dizionari è semplice, infatti se ne può definire uno delimitando con le **parentesi graffe** delle coppie formate ciascuna da una chiave associata al rispettivo valore tramite **l'operatore ":" e separate da una virgola.**



# DIZIONARI

- I valori delle coppie di un dizionario possono essere associati a qualsiasi tipo di dato e essere presenti più di una volta, le chiavi invece devono essere uniche e il loro tipo immutabile

# Definizione di un dizionario

```
nome_dict = {1: 'homer', 2: 'bart'}
```

```
>>> nome_dict
```

```
{1: 'homer', 2: 'bart'}
```

- Nell'esempio proposto in precedenza è stato definito un dizionario chiamato "nome\_dict" composto da due coppie di chiavi e valori: la prima coppia ha come chiave "1" e come valore "homer", la seconda "2" come chiave e "bart" come valore.

# DIZIONARI

- Le chiavi non devono essere necessariamente numeriche così come i valori non devono essere obbligatoriamente delle stringhe:

# Definizione di un dizionario con chiavi di tipo stringa

```
nome_dict = {'nome': 'homer', 'cognome': 'simpson'}
```

```
>>> nome_dict
```

```
{'nome': 'homer', 'cognome': 'simpson'}
```

# Definizione di un dizionario con valori di tipo numerico

```
>>> nome_dict = {'primo': 1, 'secondo': 2}
```

```
>>> nome_dict
```

```
{'secondo': 2, 'primo': 1}
```

# Definizione di un dizionario con valori e chiavi di tipo differente

```
nome_dict = {'nome': 'marge', 1: [4, 6, 8]}
```

```
>>> nome_dict
```

```
{'nome': 'marge', 1: [4, 6, 8]}
```

# DIZIONARI

- Python dispone anche di una sintassi alternativa per la definizione dei dizionari che prevede l'utilizzo della funzione nativa dict()

```
# Definizione di un dizionario con dict()
```

```
>>> nome_dict = dict({1:'php',2:'java'})
```

```
>>> nome_dict
```

```
{1: 'php', 2: 'java'}
```

```
# Definizione di un dizionario con dict()
```

```
# tramite una sequenza di elementi formati da coppie
```

```
>>> nome_dict = dict([(1,'python'),(2,'javascript')])
```

```
>>> nome_dict
```

```
{1: 'python', 2: 'javascript'}
```



# DIZIONARI: ACCESSO

- Dato che i dizionari contengono coppie di chiavi e valori, il modo più semplice per accedere ad un valore è quello di fare riferimento alla relativa chiave; a tal fine si usa una sintassi che prevede di utilizzare il nome del dizionario seguito da una chiave inserita tra parentesi quadre:

# Accedere agli elementi di un dizionario

```
>>> nome_dict = {'nome': 'montgomery', 'cognome': 'burns', 'anni': 102}
>>> nome_dict['cognome']
'burns'
>>> nome_dict['anni']
102
```

- Il metodo **get()** offre una modalità alternativa per l'accesso agli elementi di un dizionario, esso accetta come parametro la chiave associata ad un determinato valore:

# Accedere agli elementi di un dizionario con get()

```
>>> nome_dict = {'nome': 'apu', 'cognome': 'Nahasapeemapetilon', 'anni': 38}
>>> nome_dict
{'anni': 38, 'cognome': 'Nahasapeemapetilon', 'nome': 'apu'}
>>> nome_dict.get('cognome')
'Nahasapeemapetilon'
>>>
```

- La differenza sostanziale tra la prima metodologia e quella basata su `get()` sta nel fatto che il metodo non restituisce nulla nel caso in cui gli venga passata una chiave inesistente come argomento, con `"nome_dict['nome_chiave_inesistente']"` verrebbe invece generato un errore.

# DIZIONARI: MODIFICHE

- Per modificare un elemento di un dizionario è sufficiente associare un nuovo valore alla chiave di una coppia:

**# Modificare un elemento di un dizionario**

```
>>> nome_dict = {1:'euro',2:'dollaro',3:'sterlina'}
>>> nome_dict
{1: 'euro', 2: 'dollaro', 3: 'sterlina'}
>>> nome_dict[2] = 'dollaro USA'
>>> nome_dict
{1: 'euro', 2: 'dollaro USA', 3: 'sterlina'}
```

- In tutte le operazioni di manipolazione dei dizionari (ad esempio modifica e rimozione) è sempre bene tenere presente che a differenza di quanto accade con le liste e le tuple, nei dizionari non abbiamo un'indicizzazione automatica da "0" a "n" operata dall'interprete di Python, ma delle chiavi definite dall'utente; si dovranno quindi utilizzare sempre queste ultime per l'accesso ai valori.

# DIZIONARI: AGGIUNTA

- Per aggiungere un elemento ad un dizionario si deve specificare una nuova chiave associando ad essa un valore utilizzando la sintassi impiegata nell'esempio seguente:

# Aggiungere un elemento ad un dizionario

```
>>> nome_dict = {'Europa': 'euro', 'USA': 'dollaro', 'UK': 'sterlina'}
>>> nome_dict
{'Europa': 'euro', 'USA': 'dollaro', 'UK': 'sterlina'}
>>> nome_dict['Svizzera'] = 'franco'
>>> nome_dict
{'Europa': 'euro', 'USA': 'dollaro', 'UK': 'sterlina', 'Svizzera': 'franco'}
```

- Quando si effettua l'inserimento di un'ulteriore coppia chiave/valore in un dizionario è necessario fare attenzione al nome utilizzato per la nuova chiave, questo perché se si dovesse indicare il nome di una chiave già presente nel dizionario il suo valore verrà aggiornato.



# DIZIONARI: RIMOZIONE

- Il metodo `pop()` consente di rimuovere una coppia chiave/valore da un dizionario accettando come parametro la chiave della coppia che si vuole eliminare; l'engine di Python si occuperà di mostrare a video il valore che sarà rimosso insieme alla chiave:

# Rimozione di un elemento da un dizionario

```
>>> nome_dict = {'a':1,'b':2,'c':3,'d':4,'e':'f'}
>>> nome_dict
{'a': 1, 'd': 4, 'b': 2, 'c': 3, 'f': 6, 'e': 5}
>>> nome_dict.pop('b')
2
>>> nome_dict
{'a': 1, 'd': 4, 'c': 3, 'f': 6, 'e': 5}
```

- Il metodo `popitem()` funziona in modo simile a `pop()` ma con la differenza che esso non accetta alcun parametro, in pratica richiamando questo metodo l'interprete di Python provvede a scegliere in modo arbitrario la coppia chiave/valore da eliminare:

# Rimozione di un elemento da un dizionario con `popitem()`

```
>>> nome_dict.popitem()
('a', 1)
>>> nome_dict
{'d': 4, 'c': 3, 'f': 6, 'e': 5}
>>>
```

# DIZIONARI: RIMOZIONE

- `clear()` è invece il metodo da utilizzare per cancellare tutto il contenuto di un dizionario tramite un'unica istruzione:

```
# Cancellare il contenuto di un dizionario con clear()
>>> nome_dict.clear()
>>> nome_dict
{}
```

- La funzione `del()` funziona infine come il metodo `pop()` quando le viene passata come argomento la chiave di una coppia, ma può essere utilizzato anche per cancellare definitivamente un dizionario:

```
# Cancellare un dizionario con del()
>>> del nome_dict
>>> nome_dict
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    nome_dict
NameError: name 'nome_dict' is not defined
```

- Tentando di richiamare un dizionario cancellato con `del()` l'interprete di Python informerà lo sviluppatore che esso non è stato definito.

# DIZIONARI: ORDINAMENTO CHIAVI

- A differenza di quanto accadeva prima di Python 3.x, non possiamo ordinare un dizionario utilizzando direttamente l'elenco ottenuto dal metodo `keys`.
- Facciamo un esempio supponiamo di voler elencare in ordine alfabetico un dizionario che rappresenta gli studenti con i rispettivi voti in una materia. Ecco cosa accadrebbe se tentiamo il **vecchio approccio**:

```
>>> diz1 = {'andrea':28, 'giuseppe':25, 'mario':21, 'elena':28}
```

```
>>> k = diz1.keys()
```

```
>>> k.sort()
```

```
Traceback (most recent call last):
```

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

```
AttributeError: 'dict_keys' object has no attribute 'sort'
```



# DIZIONARI: ORDINAMENTO CHIAVI

- Come workaround possiamo ottenere una vista ordinata convertendo in una lista l'elenco delle chiavi e applicare a quel punto il metodo sort

```
>>> k=list(k) #conversione in lista
>>> k
['mario', 'giuseppe', 'andrea', 'elena']
>>> k = sorted(k)
>>> k
['andrea', 'elena', 'giuseppe', 'mario']
>>> for elemento in k: print(elemento, diz1[elemento])
...
andrea 28
elena 28
giuseppe 25
mario 21
```

# FUNZIONI: PARAMETRI FACOLTATIVI

- Vediamo anche come passare in ingresso un insieme di valori e trasformarli in tuple (con l'operatore \*) o dizionari (con il doppio asterisco \*\*) in modo automatico, vediamo come:

```
>>> def trantupla(*tupla):  
    print(tupla)  
    print(type(tupla))
```

```
>>> trantupla('uno', 'due', 'tre')  
('uno', 'due', 'tre')  
<class 'tuple'>
```

- Vediamo i dizionari:

```
>>> def trandizionario(**dizionario):  
    print(dizionario)  
    print(type(dizionario))
```

```
>>> trandizionario(primo=1, secondo=2, terzo=3)  
{'terzo': 3, 'primo': 1, 'secondo': 2}  
<class 'dict'>
```

# COMPREHENSION PER LISTE E DIZIONARI

- Una **comprehension** è una modalità di estrazione di sottoinsiemi da liste o dizionari
- È molto potente perché ci permette di agire sugli elementi dell'insieme (lista o dizionario) e filtrarli allo stesso tempo, come se definissimo in matematica il codominio di una funzione:

`insieme = ( f(x) | x appartiene a vecchio insieme, ad una certa condizione )`

- La sintassi Python per la list-comprehension e dictionary-comprehension risulta molto potente ed espressiva:

```
lista = [espressione for variabile in altra_lista if espressione]
diz    = {espr_chiave : espr_valore for espr_chiave, espr_valore in altro_diz if espressione}
diz    = {espr_chiave : espr_valore for variabile in altra_lista if espressione}
```

- Questi strumenti permettono di compiere operazioni iterative su sequenze in maniera più immediata, flessibile ed efficiente rispetto alle consuete istruzioni cicliche che abbiamo già visto.



# COMPREHENSION PER LISTE E DIZIONARI

- Il modo migliore di comprendere questi strumenti è quello di fare degli esempi. Iniziamo dalla comprehension per liste e vedremo subito l'eleganza e la potenza nella definizione e creazione di una lista

- Esempio di conversione euro-dollaro:

```
>>> euro = [2.5, 3.7, 20.9]          #lista di valori da convertire in dollaro
>>> dollaro = [x*1.3 for x in euro]
>>> dollaro
[3.25, 4.8100000000000005, 27.169999999999998]
>>> dollaroFiltrato=[x*1.3 for x in euro if x>5.0]
>>> dollaroFiltrato
[27.169999999999998]
```

- Vediamo adesso un paio di semplici esempi di comprehension per dizionari, simili a quelli fatti per le liste.

```
>>> x=[1,2,3,4]          #data la lista x si vuole incrementare ogni valore di 100
>>> dictComp={y:y+100 for y in x}
>>> dictComp

{1: 101, 2: 102, 3: 103, 4: 104}
>>> dictCompFiltrato={y:y+100 for y in x if y>2}
>>> dictCompFiltrato
{3: 103, 4: 104}
```

# FUNZIONI PER LISTE E TUPLA

- **enumerate()** restituisce un oggetto che contiene coppie composte da indici e valori di tutti gli elementi che compongono una lista o una tupla:

# Uso della funzione enumerate()

```
>>> a_lista = ['a','b','c']
>>> a_tuple = ('x','y','z')
# coppie indice/valore della lista
>>> [(i, j) for i, j in enumerate(a_lista)]
[(0, 'a'), (1, 'b'), (2, 'c')]
# coppie indice/valore della tupla
>>> [(i, j) for i, j in enumerate(a_tuple)]
[(0, 'x'), (1, 'y'), (2, 'z')]
```

- Nell'esempio sono state create una lista e una tupla, esse sono state poi passate come argomento a `enumerate()` in modo da ottenere un oggetto per ciascuna che potesse essere iterato con il ciclo `for` al fine di ottenere in output tutte le coppie di indici e valori degli elementi di entrambe.

# ESERCIZIO

- Scrivere un programma della rubrica telefonica, programma costituito da una lista di dizionari nei quali si vogliono registrare: nome, cognome, telefono e email. Consentire tramite un menu di inserire, cancellare, ricercare un elemento presente nell'elenco o modificarlo





# CORSO DI PYTHON

Dott. Antonio Giovanni Lezzi

# CORSO PYTHON

- Moduli
- Namespaces
- Programmazione OO
- Classe e oggetti
- Metodi Set e Get
- Self
- Slot
- Ereditarietà
- Polimorfismo
- Overriding
- Ereditarietà multipla

# MODULI

- I moduli in Python non sono altro che file che possono contenere funzioni, classi
- In altre parole sono librerie che ci permettono di organizzare meglio i progetti, specie quelli di grandi dimensioni e di riutilizzare il codice
- I moduli quindi sono dei file di script (file con estensione “.py”) che possono essere richiamati da altri programmi python per riutilizzare le funzioni contenute in essi
- Creare moduli è piuttosto semplice, ma ancor più interessante è conoscere quelli già pronti tra i moltissimi già a disposizione nell'installazione di Python oppure cercare quelli che si trovano online, messi a disposizione da terze parti



# MODULI

- In “bundle” con Python abbiamo moduli che forniscono funzioni utili per risolvere diverse problematiche, come:
  - gestione delle stringhe,
  - chiamate alle funzioni del sistema operativo,
  - gestione di internet,
  - posta elettronica,
  - ...
- Online si trovano numerosissime librerie scritte da terzi per risolvere i problemi più disparati e molte di esse, in pieno stile open source, sono di utilizzo gratuito e sotto licenza GPL.

# MODULI: IMPORTARE E CREARE

- Per scrivere un modulo, basta semplicemente creare un file con estensione .py e inserirvi tutte le funzioni necessarie.
- Per utilizzare il modulo appena creato è sufficiente importarlo con il comando **import**
- Ad esempio, supponendo di voler utilizzare il modulo libreria.py, sarà sufficiente scrivere nelle prime righe del programma:

```
import libreria
```

- Dopo aver importato il modulo possiamo semplicemente richiamare le funzioni contenute in esso utilizzando la dot notation
- Digitando il nome del modulo, un punto e il nome della funzione riusciamo ad entrare nel modulo e richiamare la funzione desiderata

# MODULI: IMPORTARE E CREARE

- Ad esempio, supponendo che “libreria.py” contenga le funzioni *apri()*, *chiudi()* e *sposta(oggetto)*, possiamo richiamare le funzioni nel seguente modo:

```
libreria.apri()  
ogg=5  
libreria.sposta(ogg)  
libreria.chiudi()
```

- Se non si desidera utilizzare il nome della libreria tutte le volte che si richiama una funzione, è possibile importare anche i nomi delle funzioni direttamente
- Ad esempio, per importare la funzione *sposta(oggetto)*:

```
import libreria  
from libreria import sposta  
  
libreria.apri()  
ogg=5  
sposta(ogg)  
libreria.chiudi()
```



# MODULI: IMPORTARE E CREARE

- In questo caso la funzione `sposta` viene a far parte dell'insieme dei nomi definiti nel programma (namespace), e non necessita più del prefisso del nome del modulo (libreria)
- Se si desidera importare tutti i nomi delle funzioni di “libreria.py” all'interno del programma basta utilizzare il carattere asterisco (\*):

```
from libreria import *
```

```
apri()  
ogg=5  
sposta(ogg)  
chiudi()
```

# MODULI: COMPILAZIONE OTTIMIZZATA

- Bisogna considerare un altro vantaggio offerto dai moduli: quando si interpreta un programma python, esso crea per ogni modulo una versione “semicompilata” in un linguaggio intermedio (bytecode)
- Dopo una esecuzione, è possibile notare la presenza di tanti file con estensione .pyc (Python compiled)
- Ogni file ha lo stesso nome del relativo sorgente .py.
- In questo modo Python non dovrà interpretare tutto il codice tutte le volte, ma solamente quando viene effettuata una modifica
- L'interprete confronta ogni volta la data e l'ora del file .py con il file .pyc: se sono diverse interpreta il codice, altrimenti esegue direttamente il bytecode

# MODULI: STANDARD

- Per utilizzare tali moduli è sufficiente importarli come abbiamo visto in precedenza. Vediamo alcuni moduli standard di notevole importanza:
  - **string**: modulo che raccoglie funzioni per gestire le stringhe, ad esempio:
    - funzioni per la conversione da stringa a numero
    - funzioni per la conversione da minuscolo a maiuscolo
    - funzioni per la sostituzioni di pezzi di stringhe
  - **sys**: modulo che raccoglie funzioni per reperire informazioni di sistema, ad esempio reperire gli argomenti passati sulla linea di comando
  - **os**: modulo per eseguire operazioni del sistema operativo sottostante, ad esempio copiare, rinominare o cancellare un file
- Nella documentazione inclusa con python si trova l'elenco completo di tutti i moduli predefiniti.



# NAMESPACES

- Per meglio comprendere il meccanismo dell'importazione dei nomi delle funzioni bisogna parlare del concetto di namespace (termine già introdotto in precedenza)
- ogni modulo ha un insieme di nomi (nomi di variabili e di funzioni) che rappresentano i nomi utilizzabili al suo interno
- Questo insieme viene detto **namespace** (spazio dei nomi)
- Per trasportare i nomi da un namespace ad un altro si può usare il comando `import` che abbiamo visto

# NAMESPACES

- Grazie alla funzione `dir()` è possibile visualizzare l'elenco dei nomi presenti nel namespace corrente

```
>>> dir()  
['__builtins__', '__doc__', '__name__']
```

- Questi nomi, mostrati sopra, sono nomi definiti dall'interprete prima di cominciare
- Se io importo il modulo libreria ottengo il seguente risultato:

```
>>> from libreria import *  
>>> dir()  
['__builtins__', '__doc__', '__name__', 'apri', 'chiudi', 'sposta']
```

- Come si intuisce, adesso tutte le funzioni contenute in `libreria.py` fanno parte del namespace corrente.

# NAMESPACES

- Il concetto di namespace si applica anche alle funzioni, quando si entra in una funzione viene creato un nuovo namespace, ecco perché non sono più visibili le variabili precedenti, ma solo le variabili locali alla funzione stessa
- Infine vediamo altre due funzioni built-in:
  - **locals**
  - **globals**



# NAMESPACES

- ```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
```

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
```

# NAMESPACES

- Entrambe ritornano un dizionario, ma proviamo adesso ad importare un modulo:

```
>>> import math
>>> globals()
{'__builtins__': <module 'builtins'="" (built-in)="">, '__name__':
'__main__', '__doc_
_': None, 'math': <module 'math'="" (built-in)="">, '__package__':
None}
```

```
>>> locals()
{'__builtins__': <module 'builtins'="" (built-in)="">, '__name__':
'__main__', '__doc_
_': None, 'math': <module 'math'="" (built-in)="">, '__package__':
None}
```

- Come è possibile vedere il namespace global e locale continuano ad essere equivalenti, ma con il nuovo modulo importato

# OGGETTI BUILT-IN

- Quando un programma viene eseguito, Python, a partire dal codice, genera delle strutture dati, chiamate oggetti, sulle quali basa poi tutto il processo di elaborazione
- Gli oggetti vengono tenuti nella memoria del computer, in modo da poter essere richiamati quando il programma fa riferimento a essi
- Nel momento in cui non servono più, un particolare meccanismo, chiamato garbage collector, provvede a liberare la memoria da essi occupata
- Questa prima descrizione di un oggetto è troppo astratta per farci percepire di cosa realmente si tratti, ma non preoccupiamoci ne vedremo di più formali e concrete al momento opportuno
- Gli oggetti che costituiscono il cuore di Python, detti oggetti built-in, vengono comunemente distinti nelle seguenti categorie: core data type, funzioni built-in, classi e tipi di eccezioni built-in.



# LIBRERIA STANDARD E AL CONCETTO DI MODULO

- La libreria standard di Python è strutturata in moduli, ognuno dei quali ha un preciso ambito di utilizzo
- Ad esempio, il modulo `math` ci consente di lavorare con la matematica, il modulo `datetime` con le date e con il tempo, e via dicendo
- Per importare un modulo si utilizza la parola chiave `import`, dopodiché, come per tutti gli altri oggetti, possiamo accedere ai suoi attributi tramite il delimitatore punto:  

```
>>> import math  
>>> math.pi  
3.141592653589793  
>>> math.sin(0), math.log(1)  
(0.0, 0.0)
```

# LIBRERIA STANDARD E AL CONCETTO DI MODULO

- Possiamo utilizzare import in combinazione con la parola chiave from per importare dal modulo solo gli attributi che ci interessano, evitando così di scrivere ogni volta il nome del modulo:

```
>>> from math import pi, sin, log
```

```
>>> pi
```

```
3.141592653589793
```

```
>>> sin(0), log(1)
```

```
(0.0, 0.0)
```

- Altri moduli della libreria standard sono datetime, sys e os

# LIBRERIA STANDARD E AL CONCETTO DI MODULO

- Il modulo datetime ci consente di lavorare con le date e con il tempo:

```
>>> from datetime import datetime
```

```
>>> d = datetime.now() # È un oggetto che rappresenta il tempo  
attuale
```

```
>>> d.year, d.month, d.day # Attributi per l'anno, il mese e il giorno  
(2012, 12, 4)
```

```
>>> d.minute, d.second # Minuti e secondi  
(31, 29)
```

```
>>> (datetime.now() - d).seconds # Numero di secondi trascorsi da  
quando ho creato `d`
```

16



# LIBRERIA STANDARD E AL CONCETTO DI MODULO

- Il modulo `sys` si occupa degli oggetti legati all'interprete e all'architettura della nostra macchina:

```
>>> import sys
>>> sys.platform
'linux'
>>> sys.version # Versione di Python
'3.4.0a2 (default, Sep 10 2013, 20:16:48) \n[GCC 4.7.2]'
```

- Il modulo `os` fornisce il supporto per interagire con il sistema operativo:

```
>>> import os
>>> os.environ['USERNAME'], os.environ['SHELL'] # L'attributo `os.environ` è un dizionario
('marco', '/bin/bash')
>>> u = os.uname()
>>> u.sysname, u.version
('Linux', '#67-Ubuntu SMP Thu Sep 6 18:18:02 UTC 2012')
```

# PROGRAMMAZIONE OO

- La programmazione orientata agli oggetti (Object Oriented Programming, OOP) è un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra di loro.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la classe. Questo tipo di dato serve appunto a modellare un insieme di oggetti dello stesso tipo.
- In generale, un oggetto è caratterizzato da un insieme di attributi e da un insieme di funzionalità.

# INTRODUZIONE

## PROGRAMMAZIONE AD OGGETTI

- Il modo di pensare come programmare un applicazione è fondamentale.
- In programmazione imperativa ci mettiamo davanti il progetto e ci chiediamo cosa fare, fissando l'attenzione sull'obiettivo da raggiungere e di come raggiungerlo.
- In programmazione ad oggetti, invece, dobbiamo cercare di non personalizzare l'oggetto ma continueremo a pensare l'oggetto così come è.
- Penseremo ad un oggetto astratto, cercando di individuare quali sono le caratteristiche dell'oggetto e le sue potenzialità.
- Gli elementi caratteristici sono chiamati Attributi dell'oggetto, le potenzialità sono invece chiamate Metodi.



# ERRORI COMUNI

- Un errore che spesso si sente è l'utilizzo di Classe e Oggetto come se fossero la stessa cosa: un Oggetto è la rappresentazione reale della problematica presa in considerazione, la Classe invece è la rappresentazione astratta dell'Oggetto.
- Per esempio, se consideriamo una penna, essa nella realtà rappresenta il nostro Oggetto vero e proprio, così come è fatto: sappiamo che è una penna, il colore, il tipo, ecc...
- La Classe che rappresenterà la penna sarà, per definizione, l'astrazione dell'Oggetto. Qui ci possiamo chiedere “E cosa cambia? Non è la stessa cosa?”, la risposta è parzialmente negativa perché, attraverso questo processo di astrazione, noi andremo a determinare anche altri dettagli supplementari dell'Oggetto: se scrive o meno, livello di inchiostro, ecc.

# PROGRAMMI STRUTTURATI CONTRO OBJECT ORIENTED

La programmazione ad oggetti ha un diverso approccio alla decomposizione dei problemi:

- Con l'*approccio strutturale* ci si concentra sulla scomposizione di algoritmi in procedure.
- Nell'*approccio “a oggetti”* ci si focalizza sull'interazione di elementi (oggetti) che comunicano (scambiano messaggi) tra loro.

# CLASSE ED OGGETTI

- **Classe** è una collezione di uno o più oggetti contenenti un insieme uniforme di attributi e servizi, insieme ad una descrizione circa come creare nuovi elementi della classe stessa;
- Un **Oggetto** è dotato di stato, comportamento ed identità; la struttura ed il comportamento di oggetti simili sono definiti nelle loro classi comuni; i termini istanza ed oggetto sono intercambiabili.



# OBJECT ORIENTED PROGRAMMING

È un metodo di implementazione in cui i programmi sono organizzati attraverso un insieme di oggetti, ognuno dei quali è un'istanza di una classe. Queste classi sono tutte parte di una gerarchia di entità unite fra di loro da una relazione di ereditarietà.

Ci sono tre parti particolarmente importanti:

1. OOP utilizza un insieme di **oggetti** (non algoritmi, e gli oggetti sono la parte fondamentale della costruzione logica);
2. ogni oggetto è istanza di una **classe**;
3. ogni classe è legata alle altre attraverso una relazione detta **eredità**.

# CARATTERISTICHE OO

Si individuano solitamente nello stile Object Oriented i seguenti elementi:

- Astrazione
- Incapsulamento
- Gerarchia
- Modularità

# ASTRAZIONE

L'astrazione ci consente di evidenziare le caratteristiche fondamentali di un oggetto e di classificarlo simile ad altri dello stesso tipo e distinto da tutti gli altri tipi e quindi ci consente di tracciare confini concettuali ben definiti per descriverlo all'interno di un certo **contesto di osservazione** che ci interessa. Questo può cambiare da contesto a contesto!

È importante osservare che l'astrazione va utilizzata come strumento che permette di **focalizzare l'attenzione su una visione esterna di un oggetto**, in modo da separare quella che è l'implementazione di un comportamento dal suo ruolo nella dinamica globale di un determinato processo.



# INCAPSULAMENTO

I concetti di astrazione e di incapsulamento sono complementari:

- l'astrazione focalizza l'attenzione sul comportamento e le caratteristiche osservabili di un oggetto
- l'incapsulamento si concentra sull'implementazione che riesce a riprodurre queste caratteristiche.

L'incapsulamento è molto spesso ottenuto attraverso l'**aggregazione di informazioni**, che è il processo di nascondere tutte le informazioni private di un oggetto che non contribuiscono a definire quelle che sono le sue caratteristiche essenziali nell'interazione con le altre entità del sistema in esame.

# INCAPSULAMENTO

Tipicamente la struttura di un oggetto viene mantenuta nascosta, così come l'implementazione dei suoi metodi consentendoci di creare delle vere e proprie scatole chiuse fra i diversi tipi di astrazioni che possono essere definite per un oggetto.

L'incapsulamento è il processo di suddivisione degli elementi di un'astrazione che ne costituiscono la struttura e il comportamento; incapsulamento serve a separare l'interfaccia costruita per un certo tipo di astrazione e la sua attuazione.

# GERARCHIA

- Nella maggior parte dei problemi che si affrontano non esiste una unica astrazione da tenere in considerazione ma spesso ne esistono molteplici e la possibilità di organizzarle in gerarchie è di vitale importanza per una modellazione efficace.
- L'incapsulamento ci aiuta nella gestione di questa complessità cercando di mantenere nascosta all'utente finale la nostra astrazione e la modularità ci dà il modo per gestire le relazioni logiche dell'astrazione.
- Le gerarchie che comunemente si descrivono nella programmazione sono quelle che potremmo definire strutturali (un determinato oggetto “è un” cioè “appartiene alla tal categoria”) oppure quelli comportamentali “si comporta come un”.



# MODULARITÀ

- Il significato di modularità può essere visto come l'azione di partizionare un programma / problema in componenti che riescano a ridurre il grado di complessità.
- Nella modellazione di sistemi complessi l'astrazione in entità (e gerarchie), l'incapsulamento non bastano per rendere trattabili problemi ma è necessario ricorrere anche alla modularizzazione
- La modularizzazione è l'individuazione di gruppi di entità (classi) affini per funzionalità, area di utilizzo, comportamento e conseguente divisione del problema in sottoproblemi più facilmente affrontabili.
- Questa partizione in moduli consente anche di creare un certo numero elementi ben definiti e separati all'interno del programma stesso, ma che sia possibile connettere fra di loro per orchestrare una soluzione globale al problema in esame.

# DOMANDE DA PORSI

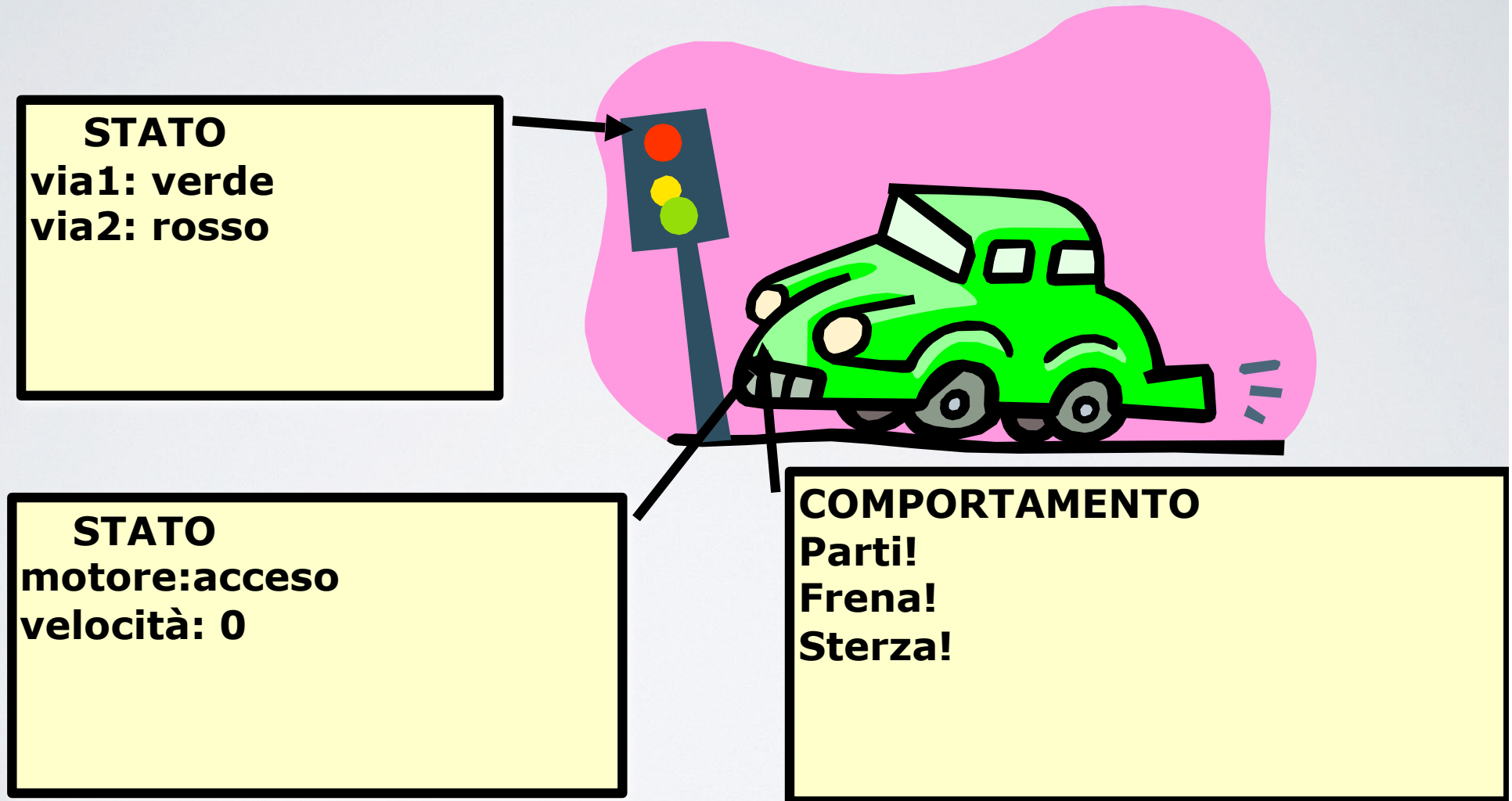
- Quali sono le entità in gioco nel nostro dominio: queste diventeranno le classi nel linguaggio di programmazione a oggetti;
- Come si relazionano tra loro le varie entità;
- Come sono distribuite le responsabilità tra le entità in gioco: chi fa cosa e come avviene la collaborazione per arrivare a raggiungere l'obiettivo prefissato;
- Architettura: organizzazione dei progetti, individuazione di componenti riusabili, di servizi generici e così via

# RIUSARE IL SOFTWARE

- A volte si incontrano classi con funzionalità simili
  - In quanto sottendono concetti semanticamente “vicini”
  - Una mountain bike assomiglia ad una bicicletta tradizionale
- È possibile creare classi disgiunte replicando le porzione di stato/comportamento condivise
  - L'approccio “Taglia&Incolla”, però, non è una strategia vincente
  - Difficoltà di manutenzione correttiva e perfetta
- Meglio “specializzare” codice funzionante
  - Sostituendo il minimo necessario



# MODELLARE LA REALTÀ





# CLASSI E OGGETTI

- Python è un linguaggio orientato agli oggetti, permette sia la programmazione tradizionale (procedurale) che il nuovo paradigma ad oggetti. Python si inquadra nei linguaggi ibridi, come il C++
- La programmazione tradizionale si è sempre basata sull'utilizzo di strutture dati (come le liste, le tuple ecc...) e su funzioni e procedure
- Questo metodo di sviluppo del software viene detto funzionale (o procedurale), con esso si organizza un intero programma in moduli che raccolgono gruppi di funzioni. Ogni funzione accede ad uno o più gruppi di dati
- I metodi di sviluppo funzionale hanno però notevoli debolezze:
  - A causa dei stretti legami tra le funzioni e i dati, si arriva ad un punto in cui ogni modifica software provoca degli effetti collaterali su altri moduli con enormi difficoltà di debug della applicazione
  - Difficoltà di riutilizzo del software, ogni volta che si vuole riciclare una funzione bisogna apportare delle modifiche strutturali per adeguarla alla nuova applicazione.



# CLASSI E OGGETTI

- La programmazione orientata agli oggetti è un modo alternativo di scomposizione di un progetto software: in essa l'unità elementare di scomposizione non è più l'operazione (la procedura) ma **l'oggetto**, inteso come modello di un'entità reale (un oggetto del mondo reale)
- Questo approccio porta ad un modo nuovo di concepire un programma: il software è ora costituito da un insieme di entità (gli oggetti) interagenti, ciascuna provvista di una struttura dati e dell'insieme di operazioni che l'oggetto è in grado di effettuare su quella struttura
- Poiché ciascun oggetto incapsula i propri dati e ne difende l'accesso diretto da parte del mondo esterno, si è certi che cambiamenti del mondo esterno non influenzeranno l'oggetto o il suo comportamento
- D'altra parte per utilizzare un oggetto basta conoscere che dati esso immagazzina e che operazioni esso fornisce per operare su questi dati, senza curarsi dell'effettiva realizzazione interna dell'oggetto stesso

# CLASSI E OGGETTI

- Questo nuovo modo di sviluppare programmi software è più vicino alla nostra realtà quotidiana, pensiamo ad un oggetto del mondo reale, ad esempio una automobile
- Una automobile ha un insieme di caratteristiche: il colore , la cilindrata ecc... inoltre essa dispone di operazioni da svolgere esclusivamente con essa, ad esempio:
  - accensione.
  - cambio della marcia.
  - parcheggio.
- è giusto correlare le sue caratteristiche e le suo operazioni in una sola entità (un solo oggetto)
- Inoltre la programmazione ad oggetti favorisce la programmazione di gruppo, poiché gli oggetti non possono dipendere dall'implementazione di altri oggetti, ma solo dalle loro operazioni, perciò un programmatore può sviluppare un oggetto senza preoccuparsi della struttura degli altri elementi che compongono il sistema

# CLASSI E OGGETTI

- I principali concetti su cui si basa questo modo di programmare:
- Una classe definisce un tipo di oggetto
- Se intendo creare un nuovo oggetto devo indicare al programma le sue caratteristiche, per fare questo devo definire una classe di appartenenza.
- In particolare la classe definisce:
  - Le variabili contenute nell'oggetto, che si chiamano **dati membri**
  - Le funzioni contenute nell'oggetto, che si chiamano **metodi**. Queste funzioni permettono di svolgere operazioni solo sui dati membri dell'oggetto stesso e ogni metodo agisce esclusivamente sui dati membri della classe di appartenenza
- Questo vincolo rappresenta la grande forza della programmazione ad oggetti, la quale costringe il programmatore ad organizzare il software per componenti riciclabili ben distinti



# CLASSI E OGGETTI

- Ad esempio, se volessi costruire un programma che conserva un archivio di **persone**, potrei creare la classe delle “**persona**”. Tale classe potrebbe avere i seguenti dati membri:
  - nome della persona
  - cognome della persona
  - indirizzo
  - telefono
  - stato civile
- Si potrebbero definire i seguenti metodi:
  - cambia l'indirizzo della persona
  - cambia il telefono della persona
  - cambia lo stato civile della persona
- Questi metodi agiscono esclusivamente sui dati membri della classe

# CLASSI E OGGETTI

- Una volta definita una classe, siamo in grado di creare tanti oggetti che riflettono le caratteristiche della classe stessa
- Infatti gli oggetti non sono altro che “istanze” della classe. Tornando al nostro esempio delle persone, possiamo creare i seguenti oggetti:
  - Antonio Lezzi
  - Mario Rossi
  - Elena Bianchi
- Queste sono tre istanze della classe persona. Ogni oggetto ha una copia dei dati membri dove conserva tutte le proprie informazioni

# CLASSI E OGGETTI

- La definizione di classe permette di realizzare l'incapsulamento, consiste nella protezione dei dati membri dagli accessi non desiderati
- Questo avviene perché dall'esterno di un oggetto si può accedere ad un dato membro solo mediante un metodo e non direttamente usando il nome del dato membro stesso
- Ad esempio, se volessi modificare l'indirizzo di una persona non posso semplicemente assegnare il nuovo indirizzo al nome della variabile (dato membro), come si farebbe nella programmazione tradizionale bensì sono obbligato ad utilizzare l'apposito metodo



# CLASSI E OGGETTI

- In realtà l'incapsulamento non è obbligatorio, si può decidere, singolarmente su ogni dato membro, se renderlo **protetto** oppure no
- Generalmente gli oggetti hanno un insieme di metodi e dati che sono resi di dominio **pubblico**, mentre altri sono inaccessibili dall'esterno
- Questo principio è conosciuto con il nome di “**information hiding**” (mascheramento dell'informazione) e fa sì che un oggetto sia diviso in due parti ben distinte: una interfaccia pubblica e una rappresentazione privata

# CLASSI E OGGETTI

- Il mascheramento dell'informazione permette di rimuovere dal campo di visibilità esterno all'oggetto alcuni metodi o dati che sono stati incapsulati nell'oggetto stesso
- L'incapsulamento e il mascheramento dell'informazione lavorano insieme per isolare una parte del programma o del sistema dalle altre parti, permettendo così al codice di essere modificato ed esteso senza il rischio di introdurre indesiderati effetti collaterali.
- Questo metodo risulta molto utile per costruire librerie software da rilasciare a terze parti, chi utilizza la libreria vede solamente l'interfaccia pubblica e ignora tutto il resto

# CLASSI IN PYTHON

- Ecco di seguito la sintassi da utilizzare per definire una classe in python:

```
class <nome classe> [(<classe madre>,...)]:
```

```
<elenco dati membro da inizializzare>
```

```
<elenco metodi>
```

- I dati membro si esprimono come le normali variabili di python. Se devo inizializzare dei dati membro faccio un semplice assegnamento, altrimenti posso definirli al momento dell'utilizzo, esattamente come avviene per le variabili normali.
- Per i metodi basta utilizzare la medesima sintassi delle funzioni con alcuni accorgimenti.
- Ogni metodo deve avere come primo parametro l'oggetto stesso, infatti ogni volta che viene richiamato il metodo, python sistema nel primo parametro il riferimento all'oggetto stesso. Questo permette di accedere ai dati membri appartenenti allo stesso oggetto, normalmente si usa chiamare questo parametro “**self**”



# CLASSI IN PYTHON

- Vediamo l'esempio delle persone scritto praticamente in codice python:

```
class persona:
    nome = ""
    cognome = ""
    indirizzo = ""
    telefono = ""
    stato_civile = ""

    def cambia_indirizzo(self, indirizzo):
        self.indirizzo = indirizzo

    def cambia_telefono(self, s):
        self.telefono = s

    def cambia_stato_civile(self, s):
        self.stato_civile = s

    def stampa(self):
        print(self.nome, self.cognome, self.indirizzo, self.stato_civile)
```

# CLASSI IN PYTHON

- L'operazione di istanziamento di una classe provoca la creazione di un oggetto
- Quando viene istanziato un nuovo oggetto della nostra classe, viene automaticamente invocata una funzione membro (detta anche "metodo") speciale, detto **costruttore**
- Spesso è necessario inizializzare i dati membri dell'oggetto nel momento della istanziamento
- Nell'esempio è utile inizializzare subito la persona con il suo nome e cognome (infatti il nome e cognome sono assegnati alla nascita e non cambiano più)
- Per fare questo la teoria della programmazione orientata agli oggetti prevede l'uso di una funzione costruttore
- La funzione denominata “**\_\_init\_\_()**”, quando all'interno di una classe viene dichiarata una funzione con questo nome, allora essa viene invocata automaticamente alla creazione dell'oggetto

# CLASSI IN PYTHON

- Vediamo come modificare la classe “persona” per inizializzare automaticamente il nome e il cognome:

```
class persona:
    indirizzo = ""
    telefono = ""
    stato_civile = ""

    def __init__(self,n,c):
        self.nome = n
        self.cognome = c

    def cambia_indirizzo(self,s):
        self.indirizzo = s

    def cambia_telefono(self,s):
        self.telefono = s

    def cambia_stato_civile(self,s):
        self.stato_civile = s

    def stampa(self):
        print(self.nome,self.cognome,self.indirizzo,self.stato_civile)
```

- Con la chiamata:

```
p1 = persona('antonio','lezzi')
p1.stampa() # risultato : antonio lezzi
```

- ottengo l'inizializzazione dei due dati membri



# CLASSI IN PYTHON

- è stato eliminato l'assegnazione della stringa vuota alle due variabili (nome, cognome) nella classe. Esse infatti sono superflue poiché le variabili vengono create all'interno della funzione `__init__`
- In precedenza abbiamo parlato di incapsulamento e subito dopo non abbiamo applicato questo principio. Infatti la prima operazione fatta sull'oggetto `p1` è stata quella di inizializzare i dati membri "nome e cognome" direttamente
- Python assume che tutti i dati membri siano pubblici e quindi modificabili dall'esterno dell'oggetto
- Per indicare di tenere protetti i dati membri e quindi realizzare veramente l'incapsulamento, si deve anteporre al nome della variabile i caratteri `"__"` ( 2 underscore)
- Così facendo nessuno può modificare il dato senza utilizzare l'apposita funzione

# CLASSI IN PYTHON

- Ad esempio, nella nostra classe “persona” è utile proteggere tutti i dati membri:

```
class persona:
    __indirizzo = ""
    __telefono = ""
    __stato_civile = ""

    def __init__(self,n,c):
        self.__nome = n
        self.__cognome = c

    def cambia_indirizzo(self,s):
        self.__indirizzo = s

    def cambia_telefono(self,s):
        self.__telefono = s

    def cambia_stato_civile(self,s):
        self.__stato_civile = s

    def stampa(self):
        print(self.__nome,self.__cognome,self.__indirizzo,self.__stato_civile)
```

# CLASSI IN PYTHON

- Se tento di assegnare direttamente “\_\_nome” e “\_\_cognome” il comando viene ignorato:

```
p1.__nome = 'prova'  
p1.stampa()  
# il risultato rimane: antonio lezzi  
via milano n. 10
```

- Esiste un altro concetto della programmazione orientata agli oggetti molto interessante, consiste nella possibilità di ridefinire gli operatori, oltre che del costruttore



# CLASSI: SET E GET

- Può essere una buona pratica dichiarare gli attributi in modalità privata e racchiudere il codice da usare per assegnare un valore ad uno degli attributi in una funzione speciale, in modo da poter fare tutti i controlli che desideriamo
- Nell'esempio che segue, sono stati resi privati gli attributi width e height, e sono state definiti due metodi per poter cambiare loro il valore

# CLASSI: SET E GET

```
class Picture():
    def __init__(self, filename, width=320, height=240, filetype='JPEG'):
        self.filename = filename
        self.setWidth(width)
        self.setHeight(height)
        self.filetype=filetype

    def setWidth(self, value):
        self.__width = value

    def setHeight(self, value):
        self.__height = value

img1=Picture('gatto.jpg', height=480)

img1.setWidth(200)
img1.__width = -10 # crea un nuovo attributo, non fa accedere a quello privato

print("Visualizzazione dell'intero dizionario:")
print(img1.__dict__)
```

# CLASSI: SET E GET

- Alla nostra classe, per i campi width e height, possiamo aggiungere anche dei metodi getter, che ci consentono di accedere agli attributi privati.
- Inoltre aggiungiamo un metodo che fornisce una rappresentazione completa dell'oggetto:



# CLASSI: SET E GET

```
class Picture():
    ....

    def getWidth(self):
        return self.__width

    def getHeight(self):
        return self.__height

    def getCompleteDescription(self):
        text = "Immagine %s, larghezza %d, altezza %d" % (
            self.filename,
            self.getWidth(),
            self.getHeight()
        )
        return text

...

print(img1.getCompleteDescription())
```

# CLASSI: SET COSA RESTITUISCE?

- Una funzione che imposta il valore di un attributo, svolto il suo compito, normalmente non deve restituire un valore o un riferimento
- Se però si fa in modo che restituisca un riferimento all'oggetto stesso (self), sarà possibile usare una sintassi più concisa per richiamare più metodi sullo stesso oggetto

# CLASSI: SET COSA RESTITUISCE?

```
def setWidth(self, value):  
    assert(isinstance(value, int))  
    assert(value>=0)  
    self.__width = value  
    return self
```

```
def setHeight(self, value):  
    assert(isinstance(value, int))  
    assert(value>=0)  
    self.__height = value  
    return self
```

```
img1.setWidth(2000).setHeight(3000)
```



# CLASSI: RIFERIMENTO A SELF

- Un chiarimento rispetto all'uso della parola chiave **self** nella lista dei parametri dei metodi può essere utile. Immaginiamo di avere una classe definita in questo modo:

```
class Picture():  
    def doSomething(self, n):  
        print("Devo fare qualcosa con il valore %d." % n)
```

- Nel momento in cui vogliamo usare il metodo doSomething() abbiamo due possibilità, perfettamente equivalenti

# CLASSI: RIFERIMENTO A SELF

- La prima è di usare il nome dell'istanza:

```
img1=Picture()  
img1.doSomething(42)    # uso il nome dell'istanza
```

- La seconda è di usare il nome della classe:

```
Picture.doSomething(img1, 42) # uso il nome della classe e passo l'istanza come  
primo parametro
```

- Il primo modo è più pratico, ma viene comunque tradotto internamente in una chiamata al metodo della classe Picture, passando il riferimento all'istanza indicata come primo parametro
- Nel codice del metodo doSomething() non c'è modo di sapere come si chiama l'istanza (ce ne potrebbero essere molte), quindi si usa un riferimento generico, che è appunto self

# CLASSI: RIFERIMENTO A SELF

- Tra l'altro, il nome self è semplicemente una convenzione. Sarebbe perfettamente lecito, seppur sconsigliabile per ovvii motivi di leggibilità, scrivere un codice come il seguente:

```
class Picture():
    def setDescription(foo, text):
        foo.description=text

    def getDescription(bar):
        return bar.description

img1=Picture()
img1.setDescription('foto del mare')
print(img1.getDescription())
```



# CLASSI IN PYTHON

- Supponiamo di creare la classe dei numeri complessi nel seguente modo:

```
class num_comp:
    parte_reale = 0
    parte_immaginaria = 0

    def somma(self, num):
        # num è un oggetto di classe num_comp
        self.parte_reale = self.parte_reale + num.parte_reale
        self.parte_immaginaria = self.parte_immaginaria +
        num.parte_immaginaria

    def stampa(self):
        print(str(self.parte_reale) + '+' + str(self.parte_immaginaria) + 'i')
```

- Un numero complesso è composto da due valori: la parte reale e la parte immaginaria, e un metodo che ne fa la somma.

# CLASSI IN PYTHON

- Per utilizzare questa classe devo scrivere il seguente programma:

```
n1 = num_comp()  
n1.parte_reale = 1  
n1.parte_immaginaria = 1  
  
n2 = num_comp()  
n2.parte_reale = 2  
n2.parte_immaginaria = 3  
  
n1.somma(n2)  
n1.stampa() # risultato 3+4i
```

- La sintassi, purtroppo, risulta piuttosto articolata. Che bello sarebbe poter scrivere  $(n1 + n2)$  !, proprio come si fa in matematica !

# CLASSI IN PYTHON

- Per poter scrivere questo è necessario rendere l'operatore "+" del python polimorfico, in modo che si accorga di lavorare con i numeri complessi e richiami la giusta funzione
- Per fare questo si deve dichiarare la classe nel seguente modo:

```
class num_comp:
    parte_reale = 0
    parte_immaginaria = 0

    def __add__(self, num):
        ris = num_comp()
        ris.parte_reale = self.parte_reale + num.parte_reale
        ris.parte_immaginaria = self.parte_immaginaria + num.parte_immaginaria
        return ris

    def stampa(self):
        print(str(self.parte_reale) + '+' + str(self.parte_immaginaria) + 'i')
```



# CLASSI IN PYTHON

- Ho dichiarato una funzione speciale, denominata “**\_\_add\_\_**”, in grado di effettuare l'overloading (la sostituzione) dell'operatore ADD (+)
- Essa crea un nuovo oggetto e somma le singole variabili. Vediamo la nuova sintassi di chiamata della funzione:

```
n1 = num_comp()  
n1.parte_reale = 1  
n1.parte_immaginaria = 1
```

```
n2 = num_comp()  
n2.parte_reale = 2  
n2.parte_immaginaria = 3
```

```
r = n1 + n2 # come in matematica !  
r.stampa() # risultato 3+4i
```

- Attraverso l'overloading degli operatori si possono creare classi molto sofisticate utilizzabili con una sintassi veramente molto elegante

# CLASSI IN PYTHON

- Di seguito vi presento una tabella con alcune funzioni speciali con le quali è possibile effettuare l'overloading

Metodi speciali	Descrizione	Esempi
<code>__init__</code>	costruttore di oggetto	<code>p1 = persona()</code>
<code>__add__</code>	operatore di somma	<code>n1+n2</code>
<code>__or__</code>	operatore OR	<code>x y</code>
<code>__len__</code>	Lunghezza	<code>len(x)</code>
<code>__cmp__</code>	confronto	<code>X==Y</code>

# ESERCIZI

- Scrivere una classe Lampadina che mi consenta di modellare il suo stato “acceso/spento”, mi consenta di compiere delle azioni “accendi e spegni” e sapere il suo stato attuale. Chiedere all’utente di inserire un valore intero N e di creare N lampadine
- Scrivere una classe Bottiglia che mi consenta di modellare il suo stato (quantità e contenuto, esempio: acqua, cocacola, Fanta). Tale classe mi consente di compiere delle azioni “bevi” o “riempi” su una bottiglia e sapere il suo stato attuale (quantità e contenuto). Chiedere all’utente di inserire un valore intero da input che indica su quale bottiglia del Distributore effettuare l’operazione desiderata, stampandone le azioni effettuate. Il Distributore mette a disposizione 5 bottiglie (le prime 3 di acqua, la quarta di CocaCola e la quinta di Fanta)





# CLASSI: NOMI SPECIALI

- Una classe può implementare certe operazioni che vengono invocate da sintassi speciali (come operazioni aritmetiche o slicing) definendo metodi con speciali nomenclature
- Questo è l'approccio all'overload degli operatori di Python, permettendo alle classi di definire il proprio comportamento rispettando altresì gli operatori del linguaggio

# CLASSI: NOMI SPECIALI

- `__init__(self [, ... ])` : Chiamata quando viene creata un'istanza. Gli argomenti vengono passati al costruttore della classe. Se una classe base ha un metodo `__init__()`, il metodo `__init__()` della classe derivata, se esiste, deve esplicitamente richiamarlo per assicurare l'appropriata inizializzazione relativa alla classe base dell'istanza
- `__del__(self )` : Chiamata quando l'istanza può essere distrutta. Viene anche chiamata distruttore. Se una classe base ha un metodo `__del__()`, il metodo `__del__()` della classe derivata, se esiste, deve esplicitamente richiamarlo per assicurare l'appropriata distruzione relativa alla classe di base dell'istanza.



# CLASSI: NOMI SPECIALI

- `__getattr__(self, name)`: Viene chiamato quando la ricerca nelle posizioni usuali di un attributo ha dato esito negativo (per esempio non è un attributo d'istanza e non c'è nemmeno nell'albero di classe per `self`). Il nome dell'attributo è `name`. Questo metodo restituisce il valore (calcolato) dell'attributo oppure solleva l'eccezione `AttributeError`
- `__setattr__(self, name, value)`: Chiamato quando viene tentato l'assegnamento di un attributo. È chiamato al posto del normale meccanismo (per esempio memorizzare il valore nel dizionario delle istanze). Il nome dell'attributo è `name`, `value` è il valore da assegnargli. Al suo interno attenzione a svolgere l'esecuzione di `'self.name = value'` — ciò causerà una chiamata ricorsiva a se stesso
- `__delattr__(self, name)`: Funziona come `__setattr__()`, ma per la cancellazione dell'attributo invece dell'assegnamento. Dovrebbe essere implementato solo se per l'oggetto è significativo `'del obj.name'`

# CLASSI: SLOT

- A volte, per questioni di performance e di risparmio di risorse, si può desiderare di non utilizzare un dizionario per immagazzinare gli attributi di un oggetto, bensì un insieme di campi predefiniti, detti slot
- Se nella definizione della classe scriviamo un'istruzione come la seguente, definiamo l'elenco degli slot
- Non sarà possibile, durante l'esecuzione, aggiungere altri attributi (nell'esempio che segue, l'elenco è una tupla, non una lista o un dizionario, e pertanto è immutabile; ma la regola varrebbe comunque)
- **`__slots__ = ('filename', 'width', 'height', 'filetype', 'quality')`**
- Già che ci siamo, vedremo nell'esempio che segue che è possibile definire anche i tipi associati a ciascuno slot, in modo da poter gestire un input controllato
- Inoltre, visto che la tupla di slot è iterabile, possiamo definire una funzione che prende in considerazione tutti i campi per fare l'input dei dati di un oggetto

# CLASSI: SLOT

```
from basic_io import *

class Picture():
    __slots__ = ('filename', 'width', 'height', 'filetype', 'quality')
    # __slots__ è una parola chiave che definisce l'elenco degli attributi
    # di una classe in modo vincolante

    __attributeTypes = {
        'filename': str,
        'width': int,
        'height': int,
        'filetype': str,
        'quality': float
    }
    # __attributeTypes è un dizionario privato in cui memorizzo i tipi
    # associati a ciascun attributo

    __attributeLabels = {
        'filename': 'nome del file',
        'width': 'larghezza',
        'height': 'altezza',
        'filetype': 'tipo di file',
        'quality': 'qualità'
    }
    # __attributeLabels è un dizionario privato in cui memorizzo la descrizione
    # associati a ciascun attributo
```



# CLASSI: SLOT

```
# costruttore
def __init__(self, filename='', width=320, height=240, filetype='JPEG', quality=1.0):
    self.filename = filename
    self.width = width
    self.height = height
    self.filetype = filetype
    self.quality = quality

def getCompleteDescription(self):
    """Fornisce una completa descrizione dell'immagine con tutti i
       nomi degli attributi e valori
       """
    fields=[]
    for attrName in self.__slots__:
        DisplayName = str(self.__attributeLabels[attrName])
        AttributeValue= str(getattr(self, attrName))
        # il metodo getattr serve per accedere al valore di un
        # attributo avendo il suo nome
        # getattr(self, 'width') corrisponde a self.width
        fields.append('%s: %s' % (DisplayName, AttributeValue))
    return ', '.join(fields)
# il metodo join applicato ad una stringa fa sì che la stringa
# venga usata come "collante" per unire tutti gli elementi della
# lista che ha come argomento
```

# CLASSI: SLOT

```
def __setattr__(self, name, value):  
    assert(isinstance(value, self.__attributeTypes[name]))  
    super().__setattr__(name, value)  
  
def inputFields(self):  
    print("*** INSERIMENTO DATI PER UNA IMMAGINE ***")  
    for field in self.__slots__:  
        self.inputField(field)  
  
def inputField(self, field):  
    datatype = self.__attributeTypes[field]  
    message = 'Inserisci un valore per "%s": ' % self.__attributeLabels[field]  
    data=checked_input(message, datatype)  
    self.__setattr__(field, data)  
  
def showFields(self):  
    print("*** VISUALIZZAZIONE DATI PER UNA IMMAGINE ***")  
    for attrName in self.__slots__:  
        self.showField(attrName)
```

# CLASSI: SLOT

```
def showField(self, attrName):  
    '''Stampa una lista di campi e valori'''  
    print('%s --> %s' % (  
  
        self.__attributeLabels[attrName],  
        str(getattr(self, attrName))  
    ))  
  
if __name__=="__main__":  
    img1=Picture()  
    img1.inputFields()  
    print(img1.getCompleteDescription())  
    img1.showFields()
```



# CLASSI E OGGETTI

- Un altro concetto importante della programmazione ad oggetti è l'ereditarietà
- Con essa è possibile definire una classe sfruttando le caratteristiche di un'altra classe (che diventa la classe madre).
- Ad esempio, se volessimo creare la classe degli studenti, dovremmo costruire una classe simile a quella delle persone, con l'aggiunta di qualche informazione in più
- Per evitare di scrivere tutto nuovamente, posso indicare al programma di ereditare tutte le caratteristiche dalla classe “persona” e aggiungere alcuni dati membri e alcuni metodi

# CLASSI E OGGETTI

- In questo modo posso affermare che gli studenti sono delle persone, quindi **ereditano** da essi le loro **caratteristiche**. Inoltre posso **aggiungere** ad esempio i seguenti **dati membri specifici** degli studenti:
  - scuola di appartenenza
  - classe di appartenenza
- Inoltre si potrebbero definire i seguenti **metodi specifici** della classe studenti:
  - cambia scuola
  - promosso

# CLASSI E OGGETTI

- L'ereditarietà permette di utilizzare anche i metodi delle classi progenitrici, così per cambiare il nome di uno studente posso semplicemente utilizzare il metodo apposito della classe persona
- Un metodo, in una classe discendente, può nascondere la visibilità di un metodo di una classe antenata, semplicemente definendo il metodo con lo stesso nome
- Lo stesso nome può così essere utilizzato in modo appropriato per oggetti che sono istanze di classi diverse
- Ereditando dalle classi antenate i metodi che operano correttamente anche nelle classi discendenti, ed eliminando quei metodi che devono agire in modo differente, il programmatore estende realmente una classe antenata senza doverla completamente ricreare



# CLASSI E OGGETTI

- La forma più comune di creazione di una classe è la **specializzazione**, mediante la quale viene creata una nuova classe poiché quella esistente è troppo generica
- In questo caso è possibile utilizzare il meccanismo dell'ereditarietà **singola**. Questo è il caso dell'esempio proposto delle persone e degli studenti
- Un altro tipico metodo di creazione è la combinazione con la quale la nuova classe è creata combinando gli oggetti di altre classi. In questo caso, se esiste, si utilizza il meccanismo dell'ereditarietà **multipla**
- Per porre in relazione gli oggetti, sono quindi utilizzati due tipi di ereditarietà: quella singola e quella multipla
- Attraverso l'ereditarietà singola, una sottoclasse può ereditare i dati membri ed i metodi da un'unica classe, mentre con l'ereditarietà multipla una sottoclasse può ereditare da più di una classe

# CLASSI E OGGETTI

- L'ereditarietà singola procura i mezzi per estendere o per perfezionare le classi, mentre l'ereditarietà multipla fornisce in più i mezzi per combinare o unire classi diverse
- Le opinioni riguardo la necessità o meno del meccanismo dell'ereditarietà multipla sono piuttosto controverse
- I contrari sostengono che si tratta di un meccanismo non strettamente necessario, piuttosto complesso e difficile da utilizzare correttamente
- Quelli a favore dicono esattamente l'opposto ed infatti sostengono che l'ereditarietà multipla è una caratteristica fondamentale di un linguaggio object oriented

# CLASSI E OGGETTI

- L'ereditarietà multipla incrementa sicuramente le capacità espressive di un linguaggio, ma comporta un costo elevato in termini di complessità della sintassi e di overhead di compilazione e di esecuzione
- Python permette l'utilizzo della ereditarietà multipla, ma non si intende discuterne ulteriormente a causa della sua complessità e del suo scarso utilizzo pratico
- I linguaggi object-oriented permettono la creazione di gerarchie di oggetti (mediante l'ereditarietà) cui sono associati metodi che hanno lo stesso nome, per operazioni che sono concettualmente simili ma che sono implementate in modo diverso per ogni classe della gerarchia
- Di conseguenza, la stessa funzione richiamata su oggetti diversi, può causare effetti completamente differenti. Questa funzionalità viene chiamata polimorfismo
- Un caso molto interessante di polimorfismo è rappresentato dall'overloading degli operatori. Con questa tecnica è possibile definire degli operatori matematici sugli oggetti che permettono di compiere operazioni sulle istanze in base al tipo di oggetto con il quale lavorano



# CLASSI IN PYTHON

```
class studente (persona): # ecco l'ereditarietà!
    scuola = ""
    classe = 0

    def cambia_scuola(self,s):
        self.scuola = s

    def promosso(self):
        if self.classe == 5:
            print('scuola finita')
        else:
            self.classe = self.classe + 1
```

# CLASSI IN PYTHON

- Se ora voglio istanziare un oggetto, è sufficiente richiamare il nome della classe come se fosse una funzione.
- Se voglio accedere ai dati membri e ai metodi basta utilizzare il punto.

```
p1 = persona() # le parentesi sono obbligatorie
p1.nome = 'antonio' # assegnazione diretta ai dati membri
p1.cognome = 'lezzi'
p1.cambia_indirizzo('via milano n. 10') # richiamo un metodo
p1.stampa() # risultato : antonio lezzi via milano n. 10
```

- p1 è un oggetto della classe persona che contiene quei valori.

# CLASSI IN PYTHON

- Notare come abbiamo assegnato ai dati membri i valori direttamente oppure attraverso gli appositi metodi
- Il metodo “cambia\_indirizzo” passando un solo parametro (il metodo ne chiede due), questo è stato possibile perché python associa al parametro “self” l’oggetto p l. Utilizzando self posso accedere direttamente ai dati membro all’interno della classe
- Se ora voglio istanziare un oggetto di tipo studente, posso utilizzare i stessi dati membri e metodi della classe progenitrice:

```
s1 = studente()  
s1.nome = 'mario'  
s1.cognome = 'rossi'  
s1.cambia_indirizzo('via pisa n. 10')  
s1.cambia_scuola('liceo')  
s1.stampa() # risultato : mario rossi via pisa n. 10
```



# CLASSI IN PYTHON

- Il risultato del metodo “stampa” non permette di visualizzare la scuola e la classe di appartenenza. Questo avviene poiché viene richiamato il metodo della classe “persona” (**classe madre**), la quale non può conoscere il valore dei dati membri della classe “studente” (**classe figlia**)
- Per risolvere questo problema ci viene in aiuto il **polimorfismo**, infatti è necessario **ridefinire** il metodo stampa costruendone una versione specifica per gli studenti

# CLASSI IN PYTHON

- Riscrivo la classe studente in questo modo:

```
class studente (persona):  
    scuola = ""  
    classe = 0  
  
    def cambia_scuola(self,s):  
        self.scuola = s  
  
    def promosso(self):  
        if self.classe == 5:  
            print("scuola finita")  
        else:  
            self.classe = self.classe + 1  
  
    def stampa(self):  
        print(self.nome,self.cognome,self.indirizzo,self.stato_civile)  
        print("scuola: "+self.scuola+" classe "+str(self.classe))
```

- Il metodo stampa è polimorfico, sembra una brutta parola, ma questo significa che assume una forma diversa in base al tipo di oggetto sul quale viene applicato

# EREDITARIETÀ E POLIMORFISMO

- La programmazione orientata agli oggetti si basa sui concetti fondamentali: ereditarietà, polimorfismo, incapsulamento
- **L'ereditarietà** nella programmazione orientata agli oggetti consiste nella possibilità di definire delle classi sulla base di altre classi già esistenti
- Immaginiamo di voler definire delle classi per la rappresentazione di semplici figure geometriche piane, di cui ci interessano il nome, l'area, il perimetro, ecc.



- Possiamo procedere definendo una classe base e poi le classi derivate, come nel seguente esempio:

```
class Shape:
    '''Un oggetto di tipo Shape rappresenta una figura geometrica
    generica'''

    @property
    def area(self):
        '''Restituisce l'area della figura'''
        pass

    @property
    def perimeter(self):
        '''Restituisce il perimetro della figura'''
        pass

    @property
    def name(self):
        return getattr(self, '_name', 'Untitled')
        # restituisce l'attributo _name se esiste, altrimenti
        'Untitled'
```

- Se poi volessimo definire le classi Rectangle e Circle potremmo farle derivare dalla classe Shape:

```
class Rectangle(Shape):  
    def __init__(self, width, height, name="A rectangle"):  
        self._width=width  
        self._height=height  
        self._name=name
```

```
@property  
def perimeter(self):  
    return 2*(self._height+self._width)
```

```
@property  
def area(self):  
    return self._height * self._width
```

```
class Circle(Shape):  
    def __init__(self, radius):  
        self._radius=radius
```

```
@property  
def perimeter(self):  
    return self._radius*math.pi*2
```

```
@property  
def area(self):  
    return math.pi * self._radius ** 2
```

- Nel programma principale potremo quindi creare delle istanze di Rectangle e Circle, e richiamare i metodi delle classi specifiche oppure quelli della classe base:

```
c=Circle(4)
print('Area del cerchio:', c.area)
print('Perimetro del cerchio:', c.perimeter)
print('Nome:', c.name)
print(type(c))
```

```
Area del cerchio: 50.2654824574
Perimetro del cerchio: 25.1327412287
Nome: Untitled
<class '__main__.Circle'>
```

```
r=Rectangle(10,20)
print('Area del rettangolo:', r.area)
print('Perimetro del rettangolo:', r.perimeter)
print('Nome:', r.name)
print(type(r))
```

```
Area del rettangolo: 200
Perimetro del rettangolo: 60
Nome: A rectangle
<class '__main__.Rectangle'>
```



# POLIMORFISMO

- Il polimorfismo fa sì che gli oggetti si comportino in maniera diversa a seconda della classe a cui appartengono, quando viene invocato uno specifico metodo (o quando viene richiesto l'accesso ad una proprietà).
- Consideriamo il seguente esempio, in cui abbiamo una lista di figure geometriche, per tutti gli elementi della quale vogliamo visualizzare nome e area:

```
shapes=[  
    Circle(5),  
    Rectangle(9,10),  
    Rectangle(11,21,'R1'),  
]  
for s in shapes:  
    print(s.name, s.area)
```

- L'output sarà, come ci aspettiamo, il seguente:

Untitled 78.5398163397

A rectangle 90

R1 231

- Si noti che viene sempre richiamato il metodo o la proprietà della classe di appartenenza dell'oggetto, mai quello della classe base

# OVERRIDING

- Quando si ridefinisce un metodo in una classe derivata, si parla di overriding
- A volte può essere utile e/o necessario richiamare comunque, nell'implementazione del metodo nella classe derivata, le azioni della classe base
- Si può procedere come nell'esempio seguente, in cui è stato aggiunto alla classe base il metodo `place()` per indicare le coordinate dell'estremo in alto a sinistra della figura geometrica
- Nelle classi `Rectangle` e `Circle` si sfrutta il metodo della classe base e poi si imposta il valore del punto centrale della figura.

```
class Shape(object):
    """
    def place(self, x, y):
        '''Imposta le coordinate dell'estremo NW della figura'''
        self._x=x
        self._y=y

    @property
    def center(self):
        '''Restituisce la tupla di coordinate del centro della figura'''
        return self._center

class Rectangle(Shape):
    """
    def place(self, x, y):
        super().place(x, y)
        self._center=(self._x+self._width/2, self._y+self._height/2)

class Circle(Shape):
    """
    def place(self, x, y):
        super().place(x, y)
        self._center=(self._x+self._radius, self._y+self._radius)
```



- Nel programma principale creeremo un'istanza e poi potremo impostarne le coordinate, visualizzando poi la tupla delle coordinate del centro:

```
c=Circle(4)
c.place(-1,12)
print('Centro del cerchio:', c.center)
```

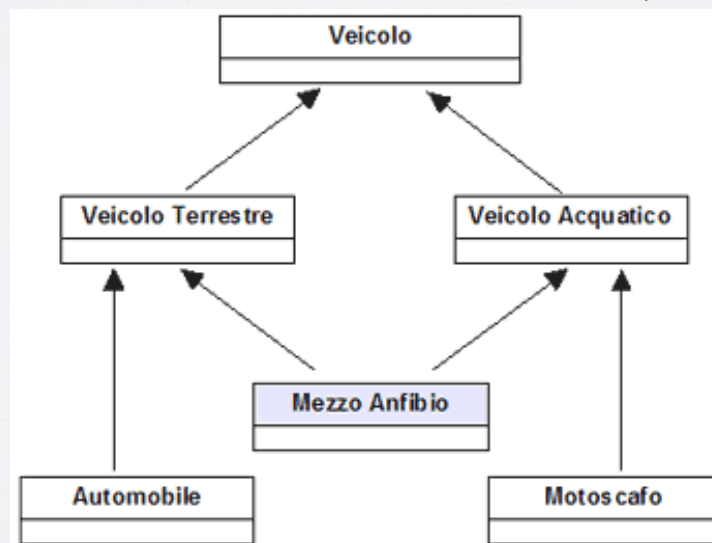
Centro del cerchio: (3, 16)

```
r=Rectangle(10,20)
r.place(4,-2)
print('Centro del rettangolo:', r.center)
```

Centro del rettangolo: (9.0, 8.0)

# EREDITARIETÀ MULTIPLA

- l'ereditarietà svolge una funzione molto importante e facilita parecchio il disegno e la codifica dei programmi
- Esistono, però, delle circostanze in cui può aver senso far derivare una classe da più classi padre: è il caso dell'ereditarietà multipla.
- Supponiamo di voler implementare una classe Mezzo Anfibia (un tipo di veicolo capace di camminare sia sulla terra che sull'acqua)
- Avvalendoci dei discorsi di astrazione dei dati ed ereditarietà fatti in precedenza potremmo tracciare il seguente diagramma di classi:



# EREDITARIETÀ MULTIPLA

- Come è facile intuire osservando la figura (in cui la gerarchia di eredità è ora rappresentata da un grafo aciclico), la classe Mezzo Anfibia, per le sue caratteristiche particolari, erediterà i metodi e le proprietà con access specifier public e protected sia dalla classe Veicolo Terrestre che dalla classe Veicolo Acquatico.
- In generale, c'è da dire che l'ereditarietà multipla pur rappresentando una notevole potenzialità nel mondo Object Oriented, favorendo notevolmente la flessibilità e il riutilizzo del codice, viene solitamente considerata un approccio da evitare a causa della complessità che può derivare da una siffatta architettura. In particolare, i due problemi principali che possono sorgere quando si utilizza l'eredità multipla sono i seguenti:
  - Ambiguità dei nomi
  - Poca efficienza nella ricerca dei metodi definiti nelle classi



# EREDITARIETÀ MULTIPLA

- Il primo problema (Ambiguità dei nomi) può verificarsi se una proprietà o un metodo ereditato è definito con lo stesso nome in tutte le classi padre di una data classe
- Ad esempio, rifacendoci all'esempio della figura precedente, se la Classe Veicolo Terrestre e la classe Veicolo Acquatico implementassero entrambi i metodi `svolta_a_dx()` e `svolta_a_sx()` ciò porterebbe ad un problema di ambiguità per la classe `Mezzo_Anfibio` nell'ereditare tali metodi da entrambe le classi.
- Il secondo problema (Poca Efficienza nella ricerca dei metodi definiti nelle classi) è causato dalla nuova struttura che assume l'architettura in questo tipo di approccio che, come detto, è adesso rappresentata da un grafo
- Con l'utilizzo di una struttura a grafo non è più possibile utilizzare la ricerca lineare (ideale sulle strutture ad albero) per l'individuazione dei metodi ma è necessario effettuare una sorta di "backtracking" sul grafo stesso.

# ESEMPIO DI EREDITARIETÀ MULTIPLA

- Nell'esempio che segue vengono definite 3 classi: A, B e C
- Le classi A e B sono indipendenti tra loro e contengono entrambe la definizione di due metodi (metodo1 e metodo2) aventi lo stesso nome sia per A che per B (la classe B definisce anche un terzo metodo: metodo3)
- La classe C, deriva sia da A che da B e definisce, a sua volta, il metodo denominato metodo1:

```
class A:
    def metodo1(self):
        print("Metodo 'metodo1' della classe A")
    def metodo2(self):
        print("Metodo 'metodo2' della classe A")
class B:
    def metodo1(self):
        print("Metodo 'metodo1' della classe B")
    def metodo2(self):
        print("Metodo 'metodo2' della classe B")
    def metodo3(self):
        print("Metodo 'metodo3' della classe B")
class C(A, B):
    def metodo1(self):
        print("Metodo 'metodo1' della classe C")
```

# ESEMPIO DI EREDITARIETÀ MULTIPLA

- Se si crea un'istanza della classe C e si invoca uno dei suoi metodi, è possibile determinare se il comportamento del metodo invocato è definito nella stessa classe C, nella classe padre A o nella classe padre B
- Le regole che determinano in modo univoco la scelta del metodo da invocare si basano su queste semplici considerazioni:
- Se la classe C (ovvero quella che implementa l'eredità multipla) definisce un metodo definito anche nelle superclassi allora verrà usata tale definizione. Nel nostro esempio, il metodo metodoI verrà invocato secondo la definizione data nella classe C
- In caso contrario, verrà utilizzata la gerarchia di ereditarietà definita per la classe C per cercare nelle sue classi padre la definizione del metodo invocato. L'ordine di ricerca dipende dalla definizione con cui è stata fissata l'ereditarietà (ovvero, nel nostro esempio si ricercherà prima nella classe padre A e poi, eventualmente, in B)



# ESEMPIO DI EREDITARIETÀ MULTIPLA

- Dunque, secondo quanto detto, dall'esempio proposto otterremo i seguenti risultati:

```
>>> obj = C()
```

```
>>> obj.metodo1()
```

```
Metodo 'metodo1' della classe C
```

```
>>> obj.metodo2()
```

```
Metodo 'metodo2' della classe A
```

```
>>> obj.metodo3()
```

```
Metodo 'metodo3' della classe B
```

# CLASSI IN PYTHON

- I sorgenti python sono raggruppati in moduli. Anche i moduli, come le classi, sono uno strumento per organizzare bene un programma e ottenere una componentistica da riutilizzare in futuro
- Si pone il problema di come organizzare le classi nei moduli. è possibile inserire più classi nello stesso modulo, si consiglia di utilizzare un modulo per ogni classe, i files sul disco equivalgono alle classi implementate e si ottiene un miglior ordine
- Una considerazione finale prima di concludere l'argomento della programmazione ad oggetti: è difficile esaurire l'argomento della programmazione ad oggetti, è molto vasto e articolato

# CLASSI IN PYTHON

- Bisogna tenere in considerazione che ancora oggi è difficile programmare bene ad oggetti per diversi motivi:
  - Ogni linguaggio di programmazione implementa in modo diverso e solo in parte le linee teoriche della programmazione ad oggetti
  - La maggior parte dei linguaggi sono ibridi e quindi non obbligano il programmatore ad usare correttamente gli oggetti
  - Non è semplice modellare un problema della vita reale ad oggetti per trasformarlo in un programma
- Python è un linguaggio ibrido, e quindi lascia libero il programmatore di scegliere se utilizzare o meno gli oggetti. Questo potrebbe essere un punto a sfavore di python rispetto ad altri linguaggio object oriented puri, come java per esempio



# ESERCIZI

- Si costruisca un sistema contenente informazioni relative ad una azienda il cui scopo è quello di profilare i propri collaboratori: dipendenti e consulenti (con le relative caratteristiche che ogni profilo possono avere), effettuare controllo accessi, gestire le commesse da assegnare al team e la gestione di un sistema di fatturazione.
- Progettare e realizzare una classe Employee(dipendente). Ciascun dipendente ha un nome (di tipo stringa) e uno stipendio (di tipo double). Scrivere un costruttore senza parametri, un costruttore con due parametri (nome e stipendio), e i metodi per conoscere nome e stipendio. Scrivere un breve programma per collaudare la classe.
- Aggiungere un metodo raiseSalary(double byPercent), che incrementi lo stipendio del dipendente secondo una certa percentuale

# ESERCIZIO

- Implementare un sistema di Ticketing Teatrale, in cui ci sono in programmazione diversi Spettacoli organizzati in diverse date da diversi gruppi. Gli Spettacoli programmati vanno in scena soltanto se sono presenti un numero sufficiente di iscritti per spettacolo (almeno 3, max 5). L'acquisto del Ticket deve esser effettuata da addetto del Botteghino che rilascia un Ticket all'utente. L'avvio dell'opera viene effettuata dalla Mascherina, che ne controlla l'accesso degli utenti e dà inizio allo spettacolo. Simulare nel Main le istanze degli oggetti in gioco, con acquisti e diversi spettacoli avviati (da simulare)

# ESERCIZIO

- Si costruisca un sistema contenente informazioni relative ad una azienda il cui scopo è quello di profilare i propri collaboratori: dipendenti e consulenti (con le relative caratteristiche che ogni profilo possono avere), effettuare controllo accessi, gestire le commesse da assegnare al team e la gestione di un sistema di fatturazione.



# ESERCIZIO

Classi:

Attributi

Impiegati: nome, cognome, stipendio -> Botteghino e Maschera: no attributi ma ereditano, diverse azioni/metodi

Spettacolo: titoloOpera, compagnia, (listaDate?)

Calendario/Direttore: data e Spettacolo (listaTicket?)

Ticket: Spettacolo, numeroTicket, Spettatore, costo e postoAssegnato, data ora Spettatore? (opzionale per valutare se implementate Ticket cartaceo o digitale)

Azioni:

Botteghino: controllo massimo di acquisti Ticket Spettacolo su Data, rimborso se numero minimo non superato (Eccezione?!)

Maschera: controllo check-in e (controllo minimo di presenze?)



# CORSO DI PYTHON

Dott. Antonio Giovanni Lezzi

# CORSO PYTHON

- Eccezioni
- Validazione dati
- Decoratori
- File
- Serializzazione
- File Binari
- Database
- Comunicazione Internet
- Richieste HTTP
- Socket
- JSON



# LE ECCEZIONI

- Il momento peggiore per un programmatore è rappresentato dal blocco del programma a causa di un errore
- Esistono due tipi di errori:
  - **Errori sintattici:** in questo caso è errato il nome di un comando o di una funzione
  - **Errori di runtime:** in questo caso i nomi sono corretti, ma c'è un errore dovuto al valore assunto da determinate variabili
- Esempi di errori di runtime sono i seguenti:
  - Divisione per zero
  - lettura di file inesistenti
- Quando si ha un errore, il programma si ferma e viene evidenziato un messaggio di errore e provoca una brusca interruzione del flusso di controllo del programma, quindi non è più possibile mantenere in vita il programma
- Questo risulta vero per gli errori sintattici, per i quali l'interprete python non sa cosa deve eseguire, ma non è completamente vero per gli errori runtime.

# LE ECCEZIONI

- Nel seguente esempio si ha un errore di runtime, in questo caso si dice che è avvenuta una ECCEZIONE, ossia un evento accidentale (in realtà in questo esempio un po' forzato ...)

```
>>> a, b = 5, 0
>>> print(a / b)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in ?
    print(a / b)
ZeroDivisionError: integer division or modulo by zero
```

- è possibile controllare questi eventi accidentali senza terminare il programma, per fare questo si utilizza un costrutto che permette di rilevare le eccezioni e passare il controllo di flusso ad un altro blocco di istruzioni

# LE ECCEZIONI

- Tale costrutto è il seguente:

**try:**

<gruppo di istruzioni sotto test>

**except:**

<gruppo di istruzioni da eseguire in caso di errore>

**else:** # opzionale

<gruppo di istruzioni2>

**finally:** # opzionale, al posto di except

<gruppo di istruzioni3>

- Tutte le istruzioni incluse nel blocco try sono tenute sotto controllo durante l'esecuzione del programma
- Se tutto va bene il blocco except viene ignorato. In caso contrario, se avviene una eccezione, si salta al blocco di istruzione che seguono la parola except



# LE ECCEZIONI

- Se incapsuliamo ogni operazione a rischio di eccezioni in un blocco try siamo in grado di reagire ad ogni errore runtime senza interrompere il programma
- Tornando all'esempio precedente si può operare in questo modo:

```
>>> a, b = 5, 0
>>> try:
    print(a / b)
except:
    print('non hai studiato matematica!')
```

```
>>> non hai studiato matematica!
```

- è possibile anche gestire tipologie diverse di eccezioni contemporaneamente, ed eseguire blocchi di istruzioni diverse a seconda del tipo di errore
- Per permettere questo è necessario fare seguire alla parola except il nome della classe di errore. Tale nome è rilevabile ad esempio dal messaggio python in caso di eccezione

# LE ECCEZIONI

- Riportando il solito esempio, per gestire in modo più puntuale l'eccezione avremo potuto scrivere nel seguente modo:

```
>>> a, b = 5, 0
>>> try:
    print(a / b)
except ZeroDivisionError:
    print('non hai studiato matematica, ma con python sei forte!')
```

**non hai studiato matematica, ma con python sei forte!**

- Nel caso si verificasse una eccezione non legata alla divisione per zero si verificherebbe ugualmente un blocco del programma, anche in presenza del blocco try
- Se si desidera eseguire un particolare gruppo di istruzioni solo nel caso in cui non avvenga nessuna eccezione, basta inserire la parola chiave else, seguito dal blocco di istruzioni desiderate
- Se si desidera far eseguire ugualmente il blocco except anche in caso di nessuna eccezione, è sufficiente sostituire la parola chiave except con la parola chiave finally

# CLASSI:VALIDAZIONE DATI

- Nel file si importa il modulo `basic_io`, che contiene il codice seguente ed effettua la verifica per qualsiasi classe, tentando la conversione



# CLASSI:VALIDAZIONE DATI

```
def checked_input(message, constraint_class):  
    """Permette l'inserimento di un valore in maniera controllata.  
    Chiede in input una stringa usando *message* come prompt, poi  
    tenta la conversione nella classe *constraint_class* e  
    restituisce il valore convertito.  
    Se viene inserito un valore non valido, viene richiesto di  
    nuovo l'inserimento.  
    """  
    while(True):  
        v=input(message)  
        try:  
            n = constraint_class(v)  
            return n  
        except ValueError as err:  
            print("Non hai inserito un valore valido...")
```

# CLASSI: VALIDAZIONE DATI

```
from Picture import *

class PictureBox(list):
    def showPictures(self):
        for i in range(len(self)):
            # NB len(self) ha senso in questo caso, visto che self è una lista
            if isinstance(self[i], Picture):
                # controlliamo, potrebbe essere None
                print(self[i].getCompleteDescription())

    def append(self, item):
        # ridefinizione del metodo append (se si vuole essere sicuri che non
        # si possano aggiungere elementi non di tipo Picture)
        assert(isinstance(item, Picture))
        super().append(item) # uso il metodo della classe padre

    def insert(self, position, item):
        # ridefinizione del metodo insert (se si vuole essere sicuri che non
        # si possano aggiungere elementi non di tipo Picture)
        assert(isinstance(item, Picture))
        super().insert(position, item) # uso il metodo della classe padre
```

# CLASSI:VALIDAZIONE DATI

```
def __setitem__(self, key, value):  
    # ridefinizione del metodo __setitem__ (se si vuole essere sicuri che non  
    # si possano cambiare elementi con oggetti non di tipo Picture, a meno  
    # che non li si imposti a None, che invece consideriamo cosa lecita)  
    assert(value is None or isinstance(value, Picture))  
    super().__setitem__(key, value) # uso il metodo della classe padre  
  
if __name__=='__main__':  
    mybox=PictureBox()  
  
    n = checked_input("Quante immagini vuoi inserire? ", int)  
  
    for i in range(n):  
        pic=Picture()  
        pic.inputFields()  
        mybox.append(pic)  
  
    mybox.showPictures()
```



# DECORATORI

- Un decoratore è una normale funzione che modifica un'altra funzione
- Quando si usa un decoratore, Python passa la funzione da decorare al decoratore, e la sostituisce con il risultato
- Facciamo un esempio senza usare la sintassi tipica dei decoratori:

```
def mio_decoratore(funzione_da_decorare):  
    #fai qualcosa con la funzione, per es.  
    funzione_da_decorare.prova = "prova"  
    return funzione_da_decorare
```

```
def molt(a, b):  
    return a * b
```

```
molt = mio_decoratore(molt)
```

# DECORATORI

- È stata definita una funzione “molt”, che rappresenta la nostra funzione da decorare, che effettua la moltiplicazione tra due numeri
- successivamente l'abbiamo sostituita con la funzione restituita da "mio\_decoratore", che altro non è che la funzione "molt" modificata
- Se proviamo a digitare:  
**molt.prova**
- otteniamo:  
**'prova'**

# DECORATORI

- Ora si procede alla scrittura del codice che usa la sintassi tipica dei decoratori:

```
def mio_decoratore(funzione_da_decorare):  
    funzione_da_decorare.prova = "prova"  
    return funzione_da_decorare
```

```
@mio_decoratore  
def molt(a, b):  
    return a * b
```

- È sufficiente mettere sopra la funzione da decorare il simbolo @ seguito dal decoratore, in questo caso "mio\_decoratore"
- Internamente Python applica la funzione decorata al decoratore e la sostituisce con il valore restituito da quest'ultimo



# DECORATORI

- un decoratore deve restituire per forza una funzione? No. Può restituire tutto quello che volete, ma la sua utilità diventa praticamente nulla. Immaginate un caso come questo:

```
def decoratore_inutile(funzione_da_decorare):  
    return True
```

```
@decoratore_inutile  
def molt(a, b):  
    return a * b
```

- Infatti "molt" non è più una funzione, ma un valore booleano, cioè True, se si provate a chiamarla si ottiene l'errore "TypeError":

```
>>>molt(5, 5)  
>>>TypeError: 'bool' object is not callable
```

- Pressoché inutile

# DECORATORI

- Un decoratore può restituire una funzione arbitraria, funzione che chiameremo "wrapper"
- Il gioco sta nel definire la funzione wrapper all'interno del decoratore, in modo che possa utilizzare le variabili presenti nel suo namespace, inclusa quindi anche la funzione da decorare che verrà passata al decoratore

```
def mio_decoratore(funzione_da_decorare):  
    def wrapper():  
        print("Sono dentro la funzione wrapper e posso accedere " \  
              "alla funzione %s" % funzione_da_decorare.__name__)  
        return funzione_da_decorare()  
  
    return wrapper
```

```
@mio_decoratore  
def foo():  
    print("Io sono la funzione da decorare")
```

```
>>>foo()  
Sono dentro la funzione wrapper e posso accedere alla funzione foo  
Io sono la funzione da decorare
```

# DECORATORI

- La funzione wrapper può fare quello che vuole alla funzione da decorare. Ma se devo passare degli argomenti alla funzione da decorare?
- Allora i parametri li passiamo alla funzione wrapper, perché quando usiamo il decoratore, la funzione wrapper sostituirà la funzione da decorare, quindi sarà lei a ricevere gli argomenti in modo da poterli usare per il proprio scopo



# DECORATORI

- Dal momento che un decoratore può lavorare con qualunque funzione, non sappiamo quanti argomenti avrà la funzione da decorare
- Il problema si risolve semplicemente facendo accettare alla funzione wrapper argomenti arbitrari non posizionali e a parola chiave, che poi passerà alla funzione da decorare

# DECORATORI

```
def mio_decoratore(funzione_da_decorare):  
    def wrapper(*args, **kwargs):  
        #Con gli argomenti possiamo fare ciò che vogliamo  
        kwargs['foo'] = "argomento modificato"  
        print("Stiamo chiamando la funzione %s con argomenti " \  
              "%s e parole chiave %s" % (funzione_da_decorare.__name__, args,  
kwargs))  
        return funzione_da_decorare(*args, **kwargs)  
  
    return wrapper
```

```
@mio_decoratore  
def molt(a, b, foo = "foo"):  
    print(foo)  
    return a * b
```

```
>>>molt(1, 2)  
Stiamo chiamando la funzione molt con argomenti (1,2) e parole chiave {'foo':  
'argomento modificato'}  
argomento modificato  
2
```

# DECORATORI

- Si potrebbe aver bisogno di personalizzare il comportamento del tuo decoratore passandogli delle opzioni
- Si ha la necessità di definire la nostra funzione decoratore dentro un'altra funzione, che chiameremo “opzioni”
- Nel momento in cui vogliamo decorare una funzione si usa la solita sintassi(`@nome_decoratore`), ma invece di usare il nostro decoratore useremo la funzione "opzioni"



# DECORATORI

```
def opzioni(valore):  
    def mio_decoratore(funzione_da_decorare):  
        funzione_da_decorare.prova = valore  
        return funzione_da_decorare  
    return mio_decoratore
```

```
@opzioni('test')  
def molt(a, b):  
    return a * b
```

```
>>>molt(5, 5)  
25  
>>>molt.prova  
'test'
```

- adesso il nostro decoratore si trova in un ambito dinamico, invece di averne uno statico

# DECORATORI

- E se insieme alle opzioni vogliamo usare anche una funzione wrapper? Il procedimento è esattamente lo stesso:

```
def opzioni(nome):  
    def mio_decoratore(funzione_da_decorare):  
        def wrapper(*args, **kwargs):  
            kwargs.update({'nome': nome})  
            print("Chiamo la funzione %s" %  
funzione_da_decorare)  
            return funzione_da_decorare(*args, **kwargs)  
        return wrapper  
    return mio_decoratore
```

```
@opzioni("pinco")  
def stampa_nome(nome = None):  
    print("Ciao ", nome)
```

# DECORATORI

- la funzione "opzioni" restituisce il decoratore, che Python userà normalmente; cioè, passerà la funzione da decorare al decoratore e verrà restituito il risultato, infatti:  

```
>>> stampa_nome  
>>> <function wrapper at 0x00BBA930>
```
- "stampa\_nome" non è la funzione decoratore (come invece si potrebbe pensare), ma bensì la funzione wrapper
- È quindi evidente che la nostra funzione da decorare ("stampa\_nome") è stata passata implicitamente per noi da Python ed è stata sostituita con il valore restituito dal decoratore
- Alcuni esempi di decoratori sono @staticmethod, @property ecc



# PROPRIETÀ

- Le proprietà sono modi per ottenere e impostare il valore di alcuni attributi (ma non solo, possono esserci proprietà calcolate) senza dover richiamare le funzioni getter e setter
- Per i programmatori Python, questa è una tecnica usata molto comunemente, considerata l'estrema semplicità di implementazione

# PROPRIETÀ

```
class Fruit(object):
    ....

    @property
    def quality(self):
        return self.__quality

    @quality.setter
    def quality(self, value):
        if 0 <= value <= 100:
            self.__quality = value
        else:
            self.__quality = 50
            self.__addAlert('bad quality: %d' % value)

def main():
    myapple = Fruit('mela granny smith')
    myapple.quality = 10
    print('quality %d' % myapple.quality)
    print('Allarmi: ', *myapple.getAlerts())
    myapple.quality = -20
    print('quality %d' % myapple.quality)
    print('Allarmi: ', *myapple.getAlerts())
```

# PROPRIETÀ

- L'output è come il seguente:

quality 10

Allarmi:

quality 50

Allarmi: bad quality: -20

- Come si vede, nonostante l'uso sia simile a quello degli attributi pubblici, in realtà vengono richiamate le funzioni specifiche impostate



# PROPRIETÀ CALCOLATE

- Alcune proprietà possono essere calcolate a partire dal valore degli attributi. Per esse si può impostare solo il getter, e non il setter. Ad esempio, si consideri il seguente codice:

```
@property
def is_good(self):
    return self.quality > 70
```

- Sarà possibile eseguire queste istruzioni:

```
myapple = Fruit('mela granny smith')
myapple.quality = 10
print(myapple.quality, myapple.is_good)
myapple.quality = 75
print(myapple.quality, myapple.is_good)
ma non questa:
myapple.is_good = False
# non funziona (manca il setter)
```

# ELIMINAZIONE DI UN ATTRIBUTO

- Gli attributi possono essere eliminati. Per farlo, è sufficiente usare la parola chiave del:

```
>>> f=Foo() # immaginiamo di avere una classe Foo.
```

```
>>> f.bar=1
```

```
>>> print(f.bar)
```

```
1
```

```
>>> del f.bar
```

```
>>> print(f.bar)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#7>", line 1, in <module>
```

```
    print(f.bar)
```

```
AttributeError: 'Foo' object has no attribute 'bar'
```

- Se usiamo le proprietà, possiamo definire una funzione che verrà invocata in occasione della cancellazione:

```
@quality.deleter
```

```
def quality(self):
```

```
    del self.__quality
```

# ELIMINAZIONE DI UN ATTRIBUTO

- Visto che consentiamo la cancellazione, sarà necessario gestire anche il getter in maniera più articolata, ad esempio con un try... except:

```
@property
def quality(self):
    '''Quality of the picture. Can be any value between 0 and 100
    (included).'''
    try:
        return self.__quality
    except AttributeError as err:
        return None
```

- Già che ci siamo, possiamo aggiungere una stringa di documentazione sulla proprietà, da poter richiamare in caso di necessità:

```
print(Fruit.quality.__doc__)
Quality of the picture. Can be any value between 0 and 100 (included).
```



# PROPRIETÀ IMPOSTATE SENZA USO DI DECORATORI

- Una proprietà può essere impostata senza uso di decoratori, tramite la funzione predefinita `property`, che permette di indicare, nell'ordine, getter, setter, deleter e docstring:

```
def getQuality(self):  
    try:  
        return self.__quality  
    except AttributeError as err:  
        return None
```

```
def setQuality(self, value):  
    if 0 <= value <= 100:  
        self.__quality = value  
    else:  
        self.__quality = 50  
        self.__addAlert('bad quality: %d' % value)
```

```
def delQuality(self):  
    print('removing quality')  
    del self.__quality
```

```
quality=property(  
    getQuality,  
    setQuality,  
    delQuality,  
    'Quality of the picture. Can be any value between 0 and 100 (included).')
```

# WITH

- L'istruzione with è usata per inglobare l'esecuzione di un blocco con metodi definiti da un gestore di contesto e permette di eseguire le istruzioni contenute nel blocco
- con questa parola chiave rende più chiaro il codice che in passato avremmo dovuto inglobare all'interno del try...finally e assicurare di creare un codice che sia resiliente e pulito nell'esecuzione del contesto

# WITH

- L'esecuzione del blocco del codice racchiuso nella clausola With avviene come segue:
- L'espressione contesto (l'espressione data dopo la with) viene valutata per ottenere un context manager
- Il metodo `__enter__()` del gestore di contesto viene richiamato
- Se un target è stato incluso nella dichiarazione With, il valore restituito da `__enter__()` viene assegnato ad esso
- L'istruzione with garantisce che se il metodo `__enter__()` restituisce valori senza la generazione di un errore, quindi `__exit__()` sarà sempre chiamata
- Così, se si verifica un errore durante l'assegnazione alla lista degli obiettivi, sarà trattato come un errore che si verifica all'interno del blocco del codice
- Si esegue il blocco del codice



# WITH

- Viene chiamato il metodo `__exit__()` del gestore contesto. Se un'eccezione è stata generata dal blocco del codice, il suo tipo, il valore e traceback vengono passati come argomenti a `__exit__()`. In caso contrario, tre argomenti sono `None`
- Se la suite è stata chiusa a causa di un'eccezione, e il valore restituito dal metodo `__exit__()` è stato falso, l'eccezione viene rilanciato. Se il valore di ritorno era vero, l'eccezione viene soppressa, e l'esecuzione continua con l'istruzione che segue l'istruzione `with`
- Se la suite era uscito per qualsiasi motivo diverso da un'eccezione, il valore restituito da `__exit__()` viene ignorato e l'esecuzione procede nella posizione dell'istruzione successiva per il tipo di uscita che è stata presa

```
with open('mydata', 'r') as f:  
    f.readline()
```

# WITH

- Implementare i metodi `__enter__` e `__exit__` consentono di utilizzare l'istanza della classe in questione facendo uso dello statement `With`
- L'idea consiste nel scrivere codice che sia più pulito possibile (per leggibilità che per esecuzione di contesto)
- Ecco un esempio di implementazione di una classe che consente la connessione a un database con la clausola `with`:

```
class DatabaseConnection(object):  
  
    def __enter__(self):  
        # esegue una connessione al database e la restituisce  
        ...  
        return self.dbconn  
  
    def __exit__(self, type, val, traceback):  
        # si assicura che la dbconnection si chiuda  
        self.dbconn.close()  
        ...
```

- Per poterlo usare non si farà altro che scrivere:

```
with DatabaseConnection() as mydbconn:  
    mydbconn.close()
```

# I FILE

- Il concetto di file è piuttosto noto. Il fatto che la gestione dei file in Python sia implementata internamente al linguaggio (built-in), garantisce una notevole velocità e semplicità nella loro gestione
- È importante poter gestire i file con istruzioni del linguaggio di programmazione, per salvare delle informazioni sul disco e renderle di conseguenza persistenti
- I file, come tutto in Python, sono oggetti e ad oggetti di tipo file afferiscono le operazioni più comuni, come aprire e chiudere un file, leggere e scrivere in sequenza byte sul medesimo file, etc
- Tutto ciò vale per **file di testo e file binari**



# I FILE: OPERAZIONI

- Ecco una tabella con le principali funzioni built-in per la gestione dei file:

Operazione	Descrizione
<code>output = open('pippo.txt', 'w')</code>	apertura di un file in scrittura
<code>input = open('dati', 'r')</code>	apertura di un file in lettura
<code>s = input.read()</code>	lettura dell'intero contenuto del file
<code>s = input.read(N)</code>	lettura di N bytes
<code>s = input.readline()</code>	lettura di una riga (per files di testo)
<code>s = input.readlines()</code>	restituisce l'intero file come lista di righe (per files di testo)
<code>output.write(s)</code>	scrive nel file i valori passati come parametri e ritorna il numero di bytes scritti
<code>output.writelines(L)</code>	scrive la lista L in righe nel file
<code>output.close(L)</code>	chiusura del file

- Le variabili `input` e `output` della tabella sono oggetti di tipo `file`.

# I FILE: ESEMPI

- Ecco un altro semplice esempio:

```
>>> miofile = open('html.txt','w') # apre il file in scrittura

>>> miofile.write('riga 1\n')      #   scrive riga 1
7
>>> miofile.write('riga 2\n')      #   scrive riga 2
7
>>> miofile.write('riga 3\n')      #   scrive riga 3
7
>>> miofile.close()                # chiude il file

>>> miofile = open('html.txt','r') # apre il file in lettura
>>> miofile.readlines()           #   legge dal file restituisce una lista di righe
['riga1\n', 'riga2\n', 'riga3\n']
```

- La funzione **open** apre il file in scrittura e ritorna un oggetto di tipo file che associamo a miofile
- Utilizziamo l'istanza creata per scrivere nel file con il metodo **write** e lo chiudiamo con **close**
- Riapriamo lo stesso file in lettura e ne stampiamo il contenuto con **readlines()**

# I FILE: APERTURA

- La funzione `open` consente altre modalità per l'apertura dei file, che possiamo passarle come secondo parametro:

Modalità	Descrizione
"r"	Apri un file di testo in lettura, equivalente a "rt", se si omette il secondo parametro di <code>open</code> è la modalità di default
"w"	Apri un file in scrittura, e ne azzeri i contenuti
"a"	Apri un file in scrittura con il puntatore del file a fine del file
"rb" e "wb"	Apri un file binario in lettura o scrittura con azzeramento del contenuto
"w+" e "w+b"	Aprono un file in modifica, rispettivamente testuale e binario, e consentendone l'aggiornamento, ma ne azzerano i contenuti
"r+" e "r+b"	Aprono un file in modifica, rispettivamente testuale e binario, e ne consentono l'aggiornamento senza azzerare i contenuti

- Dai file aperti in modalità binaria possiamo estrarre i dati come oggetti `byte` senza alcuna codifica.
- Per sapere tutto sulle modalità di apertura dei file possiamo interrogare Python:  

```
>>> print(open.__doc__)
```



# I FILE: MUOVERSI

- Come per C o altri linguaggi evoluti anche con Python possiamo spostarci all'interno dei file:

Modalità	Descrizione
<code>tell()</code>	Ritorna la posizione attuale del cursore all'interno del file
<code>seek(pos, rel)</code>	Sposta la posizione del cursore all'interno del file, con il primo parametro indichiamo il numero di byte (positivo o negativo) di cui vogliamo spostarci, il secondo parametro indica il punto di partenza ( 0=dell'inizio del file; 1=dalla posizione attuale; 2:dalla fine del file)
<code>read(N)</code>	Legge N bytes dalla posizione corrente e sposta il cursore avanti di N posizioni

- Infine per verificare lo stato del file è sufficiente stampare i dati dell'istanza corrente. Esempio:

```
>>> miofile
```

```
<_io.TextIOWrapper name='html.txt' mode='r' encoding='cp1252'>
```

- Anche se sembra scontato è utile ricordare di chiudere sempre i file (`close()`), dopo averli utilizzati

# ESEMPIO

```
def MostraAgenda(agenda):  
    print("Agenda Telefonica:")  
    for x in agenda.keys():  
        print ("Nome: ",x," \tTelefono: ",agenda[x])  
  
def AggiungiNumero(agenda,Nome,Telefono):  
    agenda[Nome] = Telefono  
  
def CercaNumero(agenda,Nome):  
    if agenda.has_key(Nome):  
        return "Numero Telefonico "+agenda[Nome]  
    else:  
        return Nome+" non trovato"  
  
def CancellaNumero(agenda,Nome):  
    if agenda.has_key(Nome):  
        del agenda[Nome]  
    else:  
        print(Nome," non trovato")
```

# ESEMPIO

```
def CaricaAgenda(agenda, NomeFile):
    in_file = open(NomeFile, "r")
    while 1:
        in_line = in_file.readline()
        if in_line == "":
            break
        in_line = in_line[:-1]
        [Nome, Telefono] = string.split(in_line, ";")
        agenda[Nome] = Telefono
    in_file.close()

def SalvaAgenda(agenda, NomeFile):
    out_file = open(NomeFile, "w")
    for x in agenda.keys():
        out_file.write(x+";"+agenda[x]+"\\n")
    out_file.close()

def print_menu():
    print('----- AGENDA -----')
    print('1. Mostra Agenda')
    print('2. Aggiungi un nuovo numero')
    print('3. Cancella un numero telefonico')
    print('4. Cerca un numero telefonico')
    print('5. Carica Agenda')
    print('6. Salva Agenda')
    print('7. Esci')
    print('-----')
```



# ESEMPIO

```
DizionarioNumeri = {}
SceltaMenu = 0
print_menu()
while SceltaMenu != 7:
    SceltaMenu = int(input("Menu (1-7):"))
    if SceltaMenu == 1:
        MostraAgenda(DizionarioNumeri)
    elif SceltaMenu == 2:
        print("Aggiungi un nome e il numero telefonico")
        Nome = input("Nome:")
        phone = input("Telefono:")
        AggiungiNumero(DizionarioNumeri, Nome, phone)
    elif SceltaMenu == 3:
        print("Rimuovi il Nome e il relativo telefono")
        Nome = input("Nome:")
        CancellaNumero(DizionarioNumeri, Nome)
    elif SceltaMenu == 4:
        print("Ricerca il numero telefonico")
        Nome = input("Nome:")
        print(CercaNumero(DizionarioNumeri, Nome))
    elif SceltaMenu == 5:
        NomeFile = input("Nome del file da caricare:")
        CaricaAgenda(DizionarioNumeri, NomeFile)
    elif SceltaMenu == 6:
        NomeFile = input("Nome del file da salvare:")
        SalvaAgenda(DizionarioNumeri, NomeFile)
    elif SceltaMenu == 7:
        pass
    else:
        print_menu()
```

# SERIALIZZAZIONE

# DESERIALIZZAZIONE

- Per serializzazione si intende un procedimento attraverso il quale dati di qualsiasi tipo presenti in memoria vengono rappresentati in una sequenza di byte che può essere poi salvata in un file per poter essere recuperata successivamente (con l'operazione inversa, detta deserializzazione).
- Possono essere serializzati valori semplici, ma anche istanze complete di classi, come si può vedere in questi due esempi, in cui per la serializzazione è stato usato il modulo pickle.

# SERIALIZZAZIONE DESERIALIZZAZIONE

```
import pickle

data=('foo', 'bar', 'baz')

f = open('mydata', 'wb')
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
f.close()

data=()

f = open('mydata', 'rb')
data = pickle.load(f)
f.close()

print(data)
```

Output:

```
('foo', 'bar', 'baz')
```



# SERIALIZZAZIONE DESERIALIZZAZIONE

In questo esempio vediamo come la serializzazione possa operare su una tupla che contiene al suo interno una tupla di istanze della classe Foo, una lista e un dizionario:

```
class Foo():
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Foo <%s>' % self.name

data = (
    Foo('abc'), Foo('def'), Foo('ghi'),
    ['uno', 'due', 'tre'],
    {'a': 'primo', 'b': 'secondo', 'c': 'terzo'}
)

f = open('mydata', 'wb')
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
f.close()
```

# SERIALIZZAZIONE

# DESERIALIZZAZIONE

```
data=()
```

```
f = open('mydata', 'rb')  
data = pickle.load(f)  
f.close()
```

```
print(data)  
print(data[0])
```

Output:

```
(<__main__.Foo object at 0xac8a98c>, <__main__.Foo object at  
0xac0ceac>, <__main__.Foo object at 0xac86d4c>, ['uno',  
'due', 'tre'], {'a': 'primo', 'c': 'terzo', 'b': 'secondo'})  
Foo «abc»
```

# SERIALIZZAZIONE DESERIALIZZAZIONE

Come per i file di testo, è possibile usare la parola chiave `with` per aprire un contesto:

```
import pickle

data=('foo', 'bar', 'baz')

with open('mydata', 'wb') as f:
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)

data=()

with open('mydata', 'rb') as f:
    data = pickle.load(f)

print(data)
```



# SERIALIZZAZIONE

# DESERIALIZZAZIONE

- **Nota sulla sicurezza:** Quando si caricano i dati memorizzati in un pickle, si deve tenere presente che in esso potrebbero essere memorizzate anche funzioni da richiamare. Ciò comporta un rischio: mai caricare dati da un pickle di fonte non fidata!

# I FILE: BINARI

- Per poter scrivere dati in un file binario è necessario trasformarli in sequenze di byte con esplicitazione del modo in cui queste sequenze sono rappresentate (ad esempio, quanti byte per un numero intero, se il numero è con o senza segno, se l'ordine dei byte è big-endian o little-endian, ecc.)
- Per poter far queste operazioni si usa il modulo struct

```
import struct
```

```
FORMAT='<2hf'
```

```
data = struct.pack(FORMAT, 30000, -12, 2.35) #packing
```

```
items = struct.unpack(FORMAT, data) #unpacking
```

```
print(items)
```

Output:

```
(30000, -12, 2.34999999046325684)
```

# I FILE: BINARI

- Il formato FORMAT, da interpretare alla luce delle informazioni presenti nella pagina struct della documentazione, significa:
  1. '<': che l'ordine dei byte è little-endian
  2. 'h': che segue un intero corto di due byte
  3. 'f': che segue un numero in virgola mobile di quattro byte
- Se si ha a che fare con stringhe, bisogna scegliere il tipo di rappresentazione della stringa e gestire la codifica e la decodifica



# I FILE: BINARI

```
name="Antonio Giovanni Lezzi"  
age=35  
FORMAT='<20sb'  
data = struct.pack(FORMAT, name.encode('UTF-8'), age)  
#packing  
items = struct.unpack(FORMAT, data) #unpacking  
print(items[0].decode('UTF-8'), items[1])
```

Output:

```
Antonio Giovanni Lez 35
```

Si noti che in questo caso la stringa viene codificata in formato UTF-8 e poi messa in una stringa di 20 caratteri (per cui il valore viene troncato).

# I FILE: BINARI IN SCRITTURA

- Un primo semplice esempio di scrittura di dati in un file binario potrebbe essere il seguente (nel quale abbiamo aggiunto l'uso di un oggetto di tipo struct.Struct)

```
import struct

class Person():
    struct_format = struct.Struct('<20sb')

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def getPacked(self):
        return self.struct_format.pack(self.name.encode('UTF-8'), self.age)

people = (Person('Mario Rossi', 20),
          Person('Giuseppe Verdi', 21),
          Person('Roberto Grigio', 22),
          Person('Elena Gialli', 23))

with open('people.dat', 'wb') as f:
    for person in people:
        f.write(person.getPacked())
```

# I FILE: BINARI IN LETTURA

- Per la lettura è sufficiente leggere il file e fare l'unpacking; abbiamo però modificato la funzione `__init__()` della classe `Person` per far sì che la stringa venga compattata, rimuovendo spazi e valori nulli finali che erano stati aggiunti in fase di impacchettamento



# I FILE: BINARI IN LETTURA

```
import struct

class Person():
    struct_format = struct.Struct('<20sb')

    def __init__(self, name, age):
        self.name = name.strip('\0x00 ')
        self.age = age

    def getPacked(self):
        return self.struct_format.pack(self.name.encode('UTF-8'), self.age)

    def __str__(self):
        return 'Nome: %s, età: %d' % (self.name, self.age)

with open('people.dat', 'rb') as f:
    while True:
        data = f.read(Person.struct_format.size)
        if not data:
            break
        name, age = Person.struct_format.unpack(data)
        p = Person(name.decode('UTF-8'), age)
        print(p)
```

# DATABASE

- Ogni linguaggio di programmazione, degno di tale nome, deve avere degli strumenti per accedere ai maggiori sistemi di gestione di base dati relazionali. Anche python dispone di strumenti atti a risolvere questa problematica.
- A tale scopo è stata sviluppata una raccolta di moduli, denominata “DB-API”.
- Tale libreria ha lo scopo di creare un interfaccia unica di accesso ai database, indipendentemente dal tipo di sistema utilizzato. Per fare questo sono stati sviluppati diversi strati:
- Uno strato unico di accesso ai dati, composto da un insieme di funzioni standard.
- Diversi drivers specifici per ogni tipo di database. A tal proposito, esistono drivers per mySQL, Informix, DB2, ODBC...

# DATABASE

- L'API DB fornisce uno standard minimo per lavorare con i database utilizzando strutture e la sintassi per quanto possibile Python. Questa API include quanto segue:
- L'importazione del modulo API.
- Acquisire una connessione con il database.
- L'emissione di istruzioni SQL e stored procedure.
- Chiusura della connessione
- Python ha un supporto integrato SQLite
- Vediamo i concetti utilizzando MySQL non ha un'interfaccia con Python 3, quindi si farà uso del modulo PyMySQL.



# PYMYSQL

- PyMySQL è un'interfaccia per la connessione a un server di database MySQL da Python
- Implementa l'API Database v2.0 di Python e contiene una libreria client nativo MySQL, il suo obiettivo è quello di essere una sostituzione a MySQLdb

# PYMYSQL, INSTALLAZIONE

- Prima di procedere ci si deve accertare di avere installato PyMySQL sulla propria macchina, per controllare l'import della libreria e la sua esecuzione produce un errore:

```
import PyMySQL
```

- Se produce il seguente errore allora la libreria PyMySQL non è stata installata:

```
Traceback (most recent call last):
```

```
  File "test.py", line 3, in <module>
```

```
    import PyMySQL
```

```
ImportError: No module named PyMySQL
```

- Scaricare il file zip dal seguente riferimento e decomprimerlo eseguendo i seguenti passi:

- <https://github.com/PyMySQL/PyMySQL>

```
cd PyMySQL*
```

```
python setup.py install
```

- **Si devono avere i privilegi di Amministrazione e PyPy installato**

# DATABASE: TRANSAZIONI

- Le transazioni sono un meccanismo che assicurano la consistenza dei dati, queste hanno le seguenti proprietà:
- **Atomicity**: Una transazione non deve esser interrotta da altre transazioni ma deve esser disturbata da ciò che può capitare
- **Consistency**: una transazione deve cominciare in uno stato consistente e lasciare il sistema consistente
- **Isolation**: I risultati intermedi di una transazione non sono visibili al di fuori delle transazione corrente
- **Durability**: una volta che la transazione è stata consegnata, i dati sono persistenti anche dopo un problema fisico di sistema
- In Python DB API 2.0 esistono due metodi che garantiscono le proprietà acide di una transazione: **Commit** e **Rollback**

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    # Execute the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()
```



# DATABASE: TRANSAZIONI

- **COMMIT**: è una operazione il quale fornisce al DBMS il segnale di finalizzare i cambiamenti, dopo questa operazione nessun cambiamento può esser ripristinato, si effettua questa operazione invocando il metodo **commit()**  
`db.commit()`
- **ROLLBACK**: nel caso in cui non si sia soddisfatti di uno o più cambiamenti e si vuole ripristinare lo stato iniziale allora si usa il metodo **rollback()**  
`db.rollback()`
- **Disconnessione**: Per chiudere la connessione di un DB si usa il metodo **close()**  
`db.close()`
- Se la connessione di un database è chiusa dall'utente con il metodo `close()`, ogni transazione esterna verrà ripristinata con il rollback dal DB. È sempre meglio invocare esplicitamente i metodi `commit` o `rollback` qualsiasi sia l'implementazione a basso livello del DB

# DATABASE: ERRORI

- Ci sono molte tipologie di errore, alcuni esempi potrebbero essere un errore di sintassi in una istruzione SQL eseguita, un errore di connessione, o chiamando il metodo fetch per un handle di istruzione già cancellato o finito
- L'API DB definisce una serie di errori che devono esistere in ogni modulo del database fornite nella tabella seguente

# DATABASE: ERRORI

Eccezione	Descrizione
<b>Warning</b>	Utilizzato per le questioni non fatali
<b>Error</b>	Classe base per gli errori
<b>InterfaceError</b>	Utilizzato per errori nel modulo di database, non il database stesso
<b>DatabaseError</b>	Utilizzato per gli errori nel database
<b>DataError</b>	Sottoclasse di DatabaseError che si riferisce a errori nei dati.
<b>OperationalError</b>	Sottoclasse di DatabaseError che si riferisce a errori come la perdita di una connessione al database. Questi errori sono generalmente al di fuori del controllo del scripter Python.
<b>IntegrityError</b>	Sottoclasse di DatabaseError per le situazioni che potrebbero danneggiare l'integrità relazionale, come i vincoli di unicità o chiavi esterne.
<b>InternalError</b>	Sottoclasse di DatabaseError che si riferisce ad errori interni al modulo database, ad esempio un cursore non attivi.
<b>ProgrammingError</b>	Sottoclasse di DatabaseError che si riferisce a errori come un nome di tabella errato e altre cose che possono tranquillamente essere attribuiti ad un errore dello sviluppatore.
<b>NotSupportedError</b>	Sottoclasse di DatabaseError che si riferisce al tentativo di chiamare la funzionalità non supportata.



# PYMYSQL, CONNESSIONE

- Prima di connettersi a un database, assicurarsi delle seguenti note:
  - aver creato un database
  - Aver creato una tabella nel database in questione
  - Aver inserito dei campi nella tabella
  - Aver creato delle utenze per poter accedere al DB
  - Di sapere utilizzare il linguaggio di interrogazione SQL e sapere un minimo di basi del funzionamento DBMS

# PYMYSQL, CONNESSIONE

- Se una connessione è stabilita con una sorgente dati, allora un oggetto Connection viene fornito dal metodo **connect()** che viene salvato in una variabile e upstate per usi futuri. Se la connessione non ha esito positivo viene fornito il valore None
- Successivamente la variabile db viene usata per creare un oggetto cursor invocando il metodo **cursor()** il quale viene usato per poter effettuare delle interrogazioni SQL.
- Al termine delle operazioni la connessione al database deve esser chiusa rilasciando tutte le risorse usate

```
import PyMySQL
```

```
# Apertura di una connessine al DB
```

```
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
```

```
# creazione di un cursor
```

```
cursor = db.cursor()
```

```
# esecuzione della query SQL usando il metodo execute()
```

```
cursor.execute("SELECT VERSION()")
```

```
# Ottiene un singolo record usando il metodo fetchone()
```

```
data = cursor.fetchone()
```

```
print("Database version : %s " % data)
```

```
# chiusura connessione dal server
```

```
db.close()
```

# PYMYSQL, CREAZIONE

- Stabilita la connessione, si è pronti per creare tabelle e/o record nel database usando il metodo `execute` per creare il cursore

- Esempio di creazione di una tabella `EMPLOYEE`

```
import PyMySQL
```

```
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )  
cursor = db.cursor()
```

```
# Cancellazione della table EMPLOYEE se già esiste usando il metodo execute()  
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
```

```
# Crea una table come da requisiti richiesti
```

```
sql = """CREATE TABLE EMPLOYEE (  
    FIRST_NAME  CHAR(20) NOT NULL,  
    LAST_NAME   CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT )"""
```

```
cursor.execute(sql)
```

```
db.close()
```



# PYMYSQL, INSERIMENTO

- Codice per creare record in una tabella

```
import PyMySQL
```

```
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )  
cursor = db.cursor()
```

```
# Preparazione di una query per inserire un record nel db
```

```
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,  
    LAST_NAME, AGE, SEX, INCOME)  
    VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
```

```
try:
```

```
    # Esegue il comando SQL
```

```
    cursor.execute(sql)
```

```
    # Committa i cambiamenti del db
```

```
    db.commit()
```

```
except:
```

```
    # Rollback nel caso ci fossero degli errori
```

```
    db.rollback()
```

```
db.close()
```

# PYMYSQL, INSERIMENTO

- Per poter creare dei record in base al contenuto delle variabili:

```
import MySQL
```

```
db = MySQL.connect("localhost","testuser","test123","TESTDB" )  
cursor = db.cursor()
```

```
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \  
    LAST_NAME, AGE, SEX, INCOME) \  
    VALUES ('%s', '%s', '%d', '%c', '%d' )" % \  
    ('Giuseppe', 'Verdi', 40, 'M', 2000)
```

```
try:  
    cursor.execute(sql)  
    db.commit()
```

```
except:  
    db.rollback()
```

```
db.close()
```

# PYMYSQL, INSERIMENTO

- Il seguente codice è una altra forma di esecuzione dello statement SQL dove si passano i parametri direttamente

```
user_id = "test123"  
password = "password"
```

```
con.execute('insert into Login values("%s", "%s")' % \  
            (user_id, password))
```



# PYMYSQL, LETTURA

- l'operazione di lettura su ogni database consiste nell'ottenere delle informazioni utili dal database
- Stabilita la connessione si è pronti per effettuare delle interrogazioni nella base di dati, si può effettuare l'invocazione del metodo **fetchone()** per ottenere un record singolo oppure **fetchall()** per ottenere diversi valori da una tabella del DB
- **fetchone()**: il metodo restituisce la prossima riga dei dati in base all'interrogazione effettuata sulla base dei dati disponibili e sui risultati ottenuti. Un result set è un oggetto ottenuto quando un cursore viene usato per interrogare una tabella
- **fetchall()**: restituisce tutte le colonne in un resultset, se alcune righe sono state già estratte allora una ulteriore invocazione del metodo restituisce le righe rimanenti dal resultset
- **rowcount**: è un attributo di sola lettura e restituisce il numero di record che sono stati generati dall'esecuzione del metodo execute()

# PYMYSQL, LETTURA

```
import PyMySQL

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
cursor = db.cursor()

sql = "SELECT * FROM EMPLOYEE \
      WHERE INCOME > '%d'" % (1000)
try:
    cursor.execute(sql)
    # Fetch tutti i dati in una lista di liste
    results = cursor.fetchall()
    for row in results:
        fname = row[0]
        lname = row[1]
        age = row[2]
        sex = row[3]
        income = row[4]
        # stampa dei dati
        print ("fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
              (fname, lname, age, sex, income ))
except:
    print ("Error: unable to fetch data")

db.close()
```

Risultato

fname=Mario, lname=Rossi, age=34, sex=M, income=2000

# PYMYSQL, AGGIORNAMENTO

L'operazione di aggiornamento significa che aggiorna uno o più record con dei nuovi valori in base a una data condizione

Nel codice di esempio di incrementa l'età se il sesso è di genere maschile

```
import PyMySQL

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

cursor = db.cursor()

sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
      WHERE SEX = '%c'" % ('M')

try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()

db.close()
```



# PYMYSQL, CANCELLAZIONE

L'operazione di cancellazione viene usata quando si vogliono cancellare dei record dal database

nel seguente esempio cancella tutti i record dalla tabella EMPLOYEE dove l'età è più grande del valore 20

```
import PyMySQL

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
cursor = db.cursor()

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()

db.close()
```

# SQLITE

- SQLite è un semplice database leggero e portatile, che risiede in un unico file con estensione .sqlite o .db (non so se esistono altre estensioni)
- In Python non abbiamo bisogno di nulla in più, in quanto è già disponibile un modulo: `sqlite3`
- Questo modulo si riferisce all'ultima versione di SQLite, che ormai è quella standard.
- Vediamo quindi di creare una semplice classe per la connessione e l'esecuzione di una query

```
sqlite3 films.sqlite
```

```
sqlite> create table film(id int, title varchar(50), description  
varchar(250));
```

```
sqlite> insert into film values(100, "Titolo 1", "Descrizione del film");
```

```
sqlite> select * from film;
```



# SQLITE

```
import sqlite3 as sq

class Conn:
    conn = None

    def __init__(self):
        self.conn = sq.connect('films.sqlite')

    def getAll(self):
        with self.conn:
            cursor = self.conn.cursor()
            cursor.execute('SELECT * FROM film')
            rows = cursor.fetchall()
            for row in rows:
                print("%s - %s" % (row[0], row[1]))

if __name__ == "__main__":
    c = Conn()
    c.getAll()
```



# COMUNICAZIONE INTERNET

- I computer in Internet comunicano scambiandosi pacchetti di dati, chiamati pacchetti IP (Internet Protocol)
- Questi pacchetti partono da un computer, attraversano vari nodi della rete (Server di rete) e arrivano a destinazione
- Per stabilire il percorso intermedio tra i due computer che vogliono comunicare si esegue un algoritmo di routing (ne esistono di vari tipi, a seconda delle esigenze e a seconda del tipo di rete).



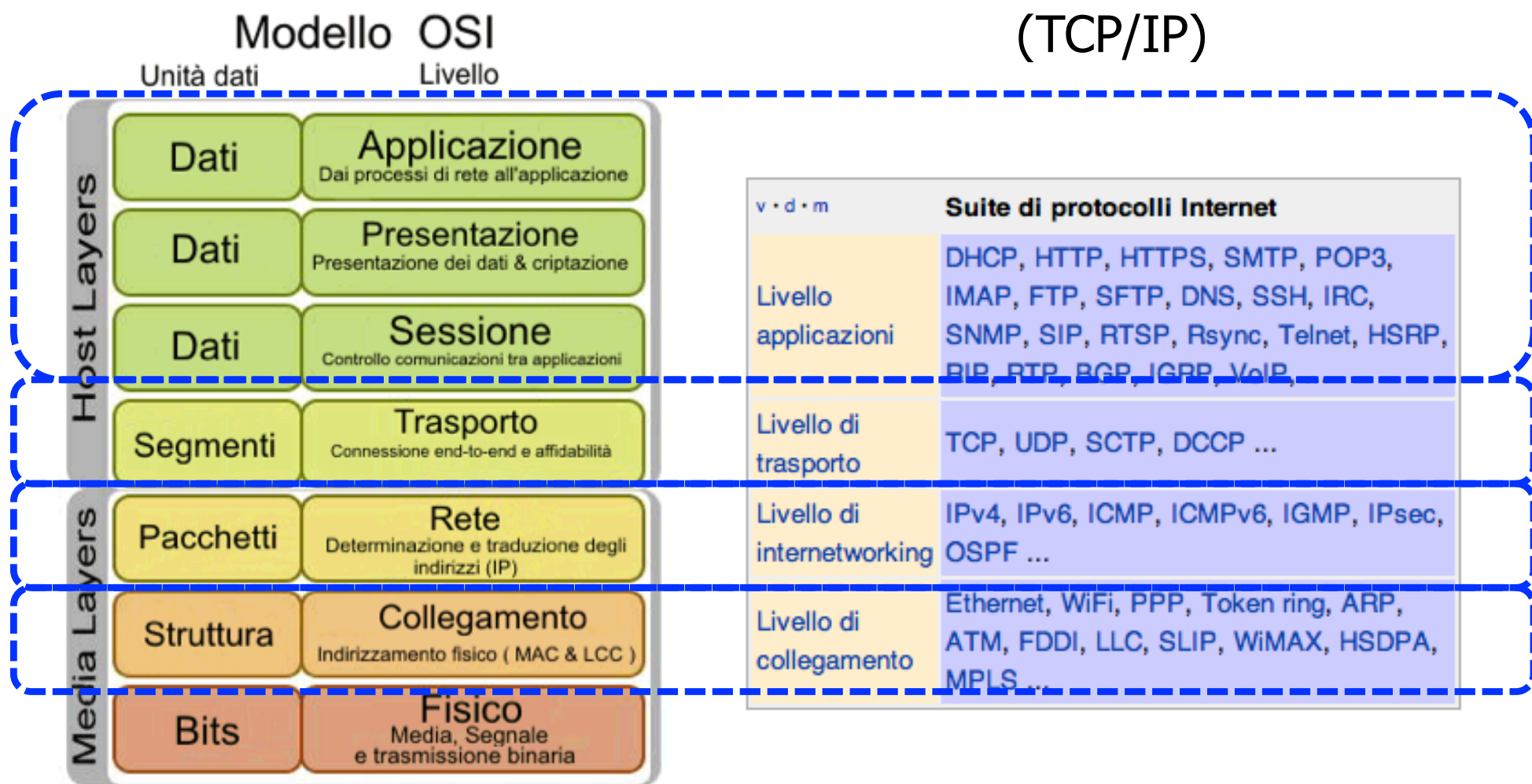
# RETE DI CALCOLATORI

- Una rete di calcolatori è un sistema che permette la condivisione di dati informativi e risorse (sia hardware sia software) tra diversi calcolatori.
- Lo scopo è di fornire dei servizi aggiuntivi per gli utenti:  
Condivisione di informazioni e risorse.
- Può essere privata o pubblica:  
Internet è la più grande rete costituita da miliardi di dispositivi in tutto il mondo.

# COMUNICAZIONE

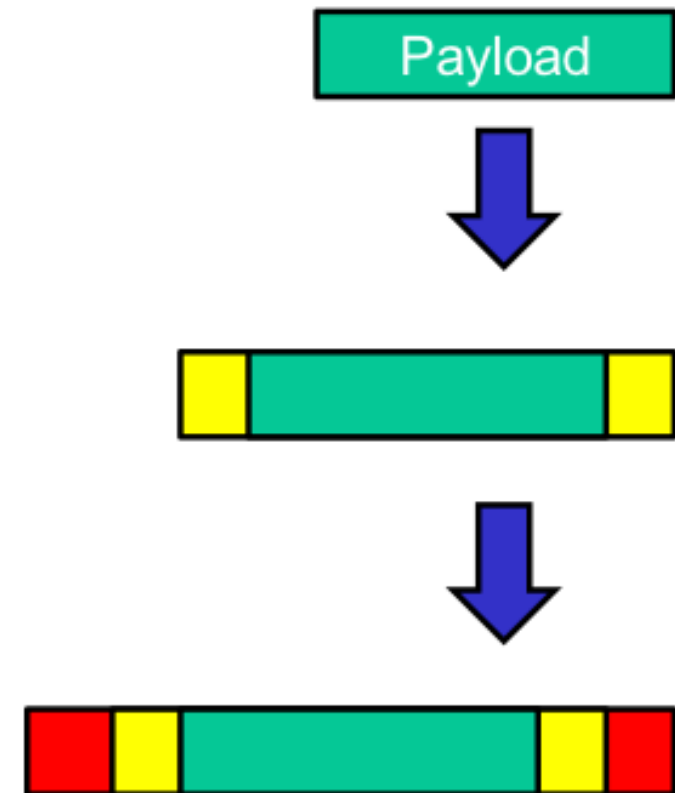
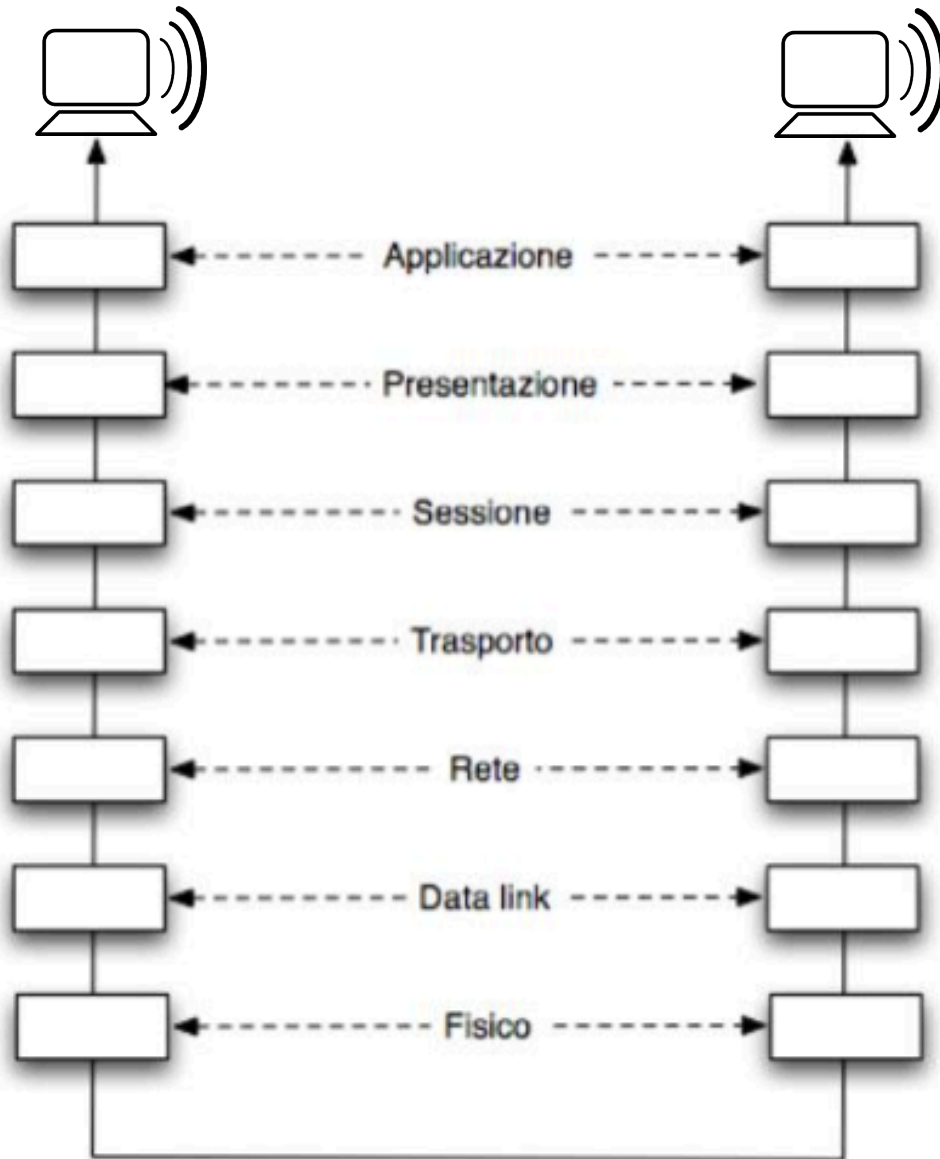
- All'interno di una rete di calcolatori la comunicazione avviene tramite scambio di informazioni (messaggi)
- Per comunicare sono necessari:
  - Un canale fisico;
  - Un linguaggio comune: protocollo di comunicazione.
- Canale fisico: Cavo telefonico, fibra ottica, onde radio, ...
- Protocollo: Insieme di regole formalmente descritte, definite al fine di favorire la comunicazione tra una o più entità.

# PILA PROTOCOLLARE





# COMUNICAZIONE A LIVELLI



# PARADIGMI DI COMUNICAZIONE

- **Client-Server:** Applicazioni di rete formate da due programmi distinti che possono essere in esecuzione in due elaboratori (host) diversi: un server e un client
  - **Server:** si mette in attesa di una richiesta da servire.
  - **Client:** effettua tale richiesta.
  - Tipicamente il client comunica con un solo server, mentre un server solitamente comunica con più client contemporaneamente.
- **Peer-to-peer:** Più host diversi comunicano tra loro comportandosi sia come client che come server.

# NETWORKING PER LE APPLICAZIONI

- Le applicazioni più diffuse per utilizzare le funzionalità di rete si appoggiano direttamente sui protocolli TCP o UDP della suite di protocolli Internet.
  - I livelli (protocolli) sottostanti sono mascherati.
- In alcuni casi si possono usare dei middleware:
  - Aggiungono un altro livello di comunicazione
  - Offrono delle funzionalità di alto livello
  - Esempio: RMI



# ALCUNI CONCETTI

- Hosts
- Internet Addresses
- Ports
- Protocols

# HOSTS

- i dispositivi connessi a Internet prendono il nome di *hosts*
- Molti hosts sono computer, ma possono anche esser considerati host router, printer, fax, macchina espresso, ecc

# INDIRIZZO IP

- Per stabilire il percorso quindi: il mittente di una comunicazione, i destinatari, i nodi intermedi, occorre che ogni computer collegato in rete abbia un nome che lo identifica univocamente, questo nome, è un numero, e si chiama indirizzo IP.



# INDIRIZZI INTERNET

- Ogni host su Internet è identificato con un unico indirizzo costituito da 4 byte definito come indirizzo: Internet Protocol cioè IP address.
- Questo indirizzo è scritto in formato di quattro cifre puntate (*dotted quad* format) come ad esempio 199.1.32.90 dove ogni byte è un intero senza segno compreso tra 0 e 255
- Ci sono circa 4 miliardi possibili indirizzi IP ma non sono allocati/assegnati efficientemente

# DOMAIN NAME SYSTEM (DNS)

- Gli indirizzi numerici sono mappati con dei nomi come *www.apple.com* o *developer.apple.com* dal DNS
- Ogni sito ha un suo software domain name server software (server) che traduce i nomi in indirizzi IP e viceversa
- DNS è un sistema distribuito

# PROTOCOLLO

- Un insieme di regole formalmente descritte, definite al fine di favorire la comunicazione tra una o più entità
- Tutte queste regole sono definite mediante specifici protocolli, dalle tipologie più varie e ciascuno con precisi compiti/finalità, a seconda delle entità interessate e il mezzo di comunicazione
- Se le due entità sono remote, si parla di protocollo di rete



# ESEMPIO PROTOCOLLO

Determinati protocolli sono gestiti da organismi quali il World Wide Web Consortium (W3C) che rispettano gli standard ISO/OSI.

Di seguito si citano i protocolli più utilizzati.

- HTTP e HTTPS - principale sistema per la trasmissione di informazioni sul web.
- Transmission Control Protocol (TCP) è uno dei principali protocolli Internet su cui si appoggiano gran parte delle applicazioni web;
- User Datagram Protocol (UDP) è usato di solito in combinazione con il protocollo IP.
- Internet Protocol (IP) - un protocollo di rete a pacchetto;

Esistono tuttavia anche protocolli non standardizzati ovvero di tipo proprietario, tipicamente adibiti a funzionalità di tecnologie altrettanto proprietarie.

# TCP (TRANSFER CONTROL PROTOCOL)

- È un protocollo di alto livello:
  - È orientato alla connessione;
  - Esegue controlli sugli errori, congestione e flusso di comunicazione
- Un destinatario TCP è identificato da un indirizzo IP (Internet Protocol) e da una porta di destinazione.
  - L'indirizzo IP identifica la macchina alla quale vogliamo collegarci.
  - Il numero della porta identifica l'applicativo (servizio) al quale vogliamo connetterci.
- Esempio: Un server web all'indirizzo 131.114.11.100 può offrire un servizio di posta elettronica e un servizio http. Il primo risponde sulla porta 110 e il secondo alla porta 80.

# URL

- Un URL o brevemente (Uniform Resource Locator) è un metodo per identificare univocamente la locazione della risorsa su internet
- Una risorsa web può essere qualsiasi cosa, una directory, un file, un oggetto in rete come ad esempio una interfaccia per fare delle query ad un database remoto, o per un motore di ricerca.
- Tramite un URL (Uniform Resource Locator) è possibile riferirsi alle risorse di Internet in modo semplice e uniforme. Si ha così a disposizione una forma intelligente e pratica per identificare o indirizzare in modo univoco le informazioni su Internet.



# URL

- I browser utilizzano gli URL per recuperare le pagine web. Java mette a disposizione alcune classi per utilizzare gli URL; sarà così possibile, ad esempio, inglobare nelle proprie applicazioni funzioni tipiche dei web browser
- Gli URL sono svariati, ognuno con un protocollo diverso, ma i più usati sono quelli che usano protocolli HTTP (HyperText transfer Protocol) e FTP (File transfer Protocol).
- La classe URL incapsula degli oggetti web, accedendovi tramite il loro indirizzo URL.
- A differenza dei web browser, l'`http://` davanti all'indirizzo è indispensabile, perché individua il protocollo dell'URL, protocollo che il Navigator e l'Explorer intuiscono anche se omissivo.

# FORMATO DI UN URL

Un URL consiste di 4 componenti:

1. il protocollo separato dal resto dai due punti (esempi tipici di protocolli sono http, ftp, news, file, ecc.);
2. il nome dell'host, o l'indirizzo IP dell'host, che è delimitato sulla sinistra da due barre (//), e sulla destra da una sola (/), oppure da due punti (:)
3. il numero di porta, separato dal nome dell'host sulla sinistra dai due punti, e sulla destra da una singola barra. Tale componente è opzionale, in quanto, come già detto, ogni protocollo ha una porta di default
4. il percorso effettivo della risorsa che richiediamo. Il percorso viene specificato come si specifica un path sotto Unix. Se non viene specificato nessun file, la maggior parte dei server HTTP aggiunge automaticamente come file di default index.html.

# MESSAGGI HTTP

- I web server permettono di ottenere informazioni come risultato di una query (interrogazione). Invece di richiedere un normale documento, si specifica nell'URL il nome di un programma, passandogli alcuni parametri che rappresentano la query vera e propria.
- In effetti tramite Python si ha più flessibilità, e i programmi vengono eseguiti dal lato client come applicazione front end. Python è possibile utilizzarlo anche lato backend tale per cui è possibile implementare dei servizi esattamente come se fossero applicazioni residenti lato web, fornendo dei servizi.
- Una query ad un particolare servizio è costituita quindi da un normale URL, con in coda alcuni parametri. La parte dell'URL che specifica i parametri inizia con un punto interrogativo (?). Ogni parametro è separato da una "e commerciale" (&), e i valori che si assegnano ai parametri sono specificati in questo modo: nome = valore (il valore è facoltativo).



# ESEMPI DI URL

`http://java.sun.com/`

`file:///Macintosh%20HD/Java/Docs/JDK%201.1.1%20docs/api/  
java.net.InetAddress.html#_top_`

`http://www.macintouch.com:80/newsrecent.shtml`

`ftp://ftp.info.apple.com/pub/`

`mailto:elharo@metalab.unc.edu`

`telnet://utopia.poly.edu`

`ftp://mp3:mp3@138.247.121.61:21000/c%3a/stuff/mp3/`

`http://elharo@java.oreilly.com/`

`http://metalab.unc.edu/nywc/comps.phtml?  
category=Choral+Works`

# MIME

- MIME è un acronimo di "Multipurpose Internet Mail Extensions".
- inizialmente utilizzato per inviare messaggi di email ma adottato in seguito anche per HTTP
- Quando un browser invia una richiesta al server web, si invia anche un header MIME

# HEADER DI RICHIESTE HTTP

GET /python/images/cup.gif HTTP/1.0

Connection: Keep-Alive

User-Agent: Mozilla/3.01 (Macintosh; I; PPC)

Host: www.oreilly.com:80

Accept: image/gif, image/x-xbitmap, image/jpeg, \*/\*



# RISPOSTA DEL SERVER

- Quando un server web risponde a una richiesta di un browser un header ed un messaggio MIME viene inviato, ecco un esempio di risposta:

```
HTTP/1.0 200 OK
Server: Netscape-Enterprise/2.01
Date: Sat, 02 Aug 1997 07:52:46 GMT
Accept-ranges: bytes
Last-modified: Tue, 29 Jul 1997 15:06:46 GMT
Content-length: 2810
Content-type: text/html
```

# RISPOSTA COMPLETA

GET python/images/index.html HTTP/1.0

HTTP/1.1 302 Moved Temporarily

Date: Mon, 04 Aug 1997 14:21:27 GMT

Server: Apache/1.2b7

Location: <http://www.python.com/python/images/index.html>

Connection: close

Content-type: text/html

<HTML><HEAD>

<TITLE>302 Moved Temporarily</TITLE>

</HEAD><BODY>

<H1>Moved Temporarily</H1>

The document has moved <A HREF="http://www.macfaq.com/macfaq/index.html">here</A>.<P>

</BODY></HTML>

# SERVIZI WEB E RICHIESTE REST

- I servizi web sono essenzialmente funzionalità rese disponibili in rete da servizi remoti dai quali, client distribuiti nel mondo possono recuperare informazioni, richiedere elaborazioni e altro
- Le richieste sul protocollo HTTP possono essere di diverso tipo e per il servizio REST avremmo:
- GET: per leggere dati da remoto senza apportare modifiche
- PUT: per richiedere la modifica dei dati pre-esistenti
- POST: per inviare dati verso il servizio con lo scopo di richiederne l'inserimento nella base dati
- DELETE: per cancellare il dato



# RICHIESTE REST

- POST, GET, PUT, DELETE richiamano gli stessi quattro concetti espressi dai metodi CRUD dei database (create, read, update e delete)
- in questo senso i servizi REST possono essere visti come un modo per gestire un database.

# CODICI DI STATO

- i codici di stato contenuti nella risposta e forniti dal server dobbiamo sapere:
- 200: OK
- 400: Bad Request
- 403: Forbidden
- 404: Not found
- 500: Internal Server error

# RETE

- Python fornisce due livelli di accesso ai servizi di rete
- Ad un livello basso, è possibile accedere al supporto di base prese nel sistema operativo sottostante, che permette di implementare i client ed i server per entrambi i protocolli orientati alla connessione e senza connessione
- Python ha anche librerie che forniscono l'accesso di livello superiore per i protocolli di rete a livello di applicazione specifici, come FTP, HTTP, e così via



# RETE

- Vi sono dei moduli che forniscono il supporto per l'accesso a Internet e il controllo del web browser:

```
>>> from urllib.request import urlopen
```

```
>>> for line in urlopen('http://www.python.org/community/diversity/'):
    text = line.decode('utf-8')
    if '<title>' in text:
        print(text)
```

```
<title>Diversity</title>
```

```
>>> import webbrowser
```

```
>>> url = 'http://www.python.org/dev/peps/pep-0001/'
```

```
>>> webbrowser.open(url) # Apre un url usando il browser di default
True
```

# RETE: DOWNLOAD FILE

- Esistono diversi modi per fare il download dei file tramite Python

- Ecco un semplice esempio con urllib

```
import urllib.request  
urllib.request.urlretrieve('http://www.sito.it/file.txt', 'file.txt')
```

# RETE: RICHIESTE HTTP REST

- L'HTTP (HyperText Transfer Protocol) è il protocollo principale al Livello Applicazioni (i livelli sono quattro: Applicazioni, Trasporto, Internetworking, Collegamento) usato per la trasmissione di informazioni sul web
- Generalmente, un Server HTTP utilizza il protocollo TCP (Transmission Control Protocol) sulla porta 80
- L'HTTP si basa su un meccanismo: richiesta/risposta (client/server). Il client esegue una richiesta e il server restituisce la risposta
- È importante tenere in mente che una applicazione può agire sia da client che da server.



# RETE: RICHIESTE HTTP REST

```
import urllib.request
response = urllib.request.urlopen('http://python.org/')
html = response.read()

print(html)
```

```
import urllib.parse
import urllib.request

url = 'http://www.server.com/register'
values = {'name' : 'Antonio Lezzi',
          'language' : 'Python' }

data = urllib.parse.urlencode(values)
print(data)
req = urllib.request.Request(url, data)
response = urllib.request.urlopen(req)
the_page = response.read()
```

# RETE: RICHIESTE HTTP REST

```
import urllib.parse
import urllib.request

url = 'http://www.server.com/register'
values = {'name' : 'Antonio Lezzi',
          'language' : 'Python' }
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
headers = { 'User-Agent' : user_agent }

data = urllib.parse.urlencode(values)
req = urllib.request.Request(url, data, headers)
response = urllib.request.urlopen(req)
the_page = response.read()

req = urllib.request.Request('http://www.pretend_server.org')
try:
    urllib.request.urlopen(req)
except urllib.error.URLError as e:
    print(e.reason)
```

# RETE: RICHIESTE HTTP REST

- La risposta ottenuta contiene tre parti:
  - Riga di stato
  - Header
  - Body
- La riga di stato consiste in un codice di tre cifre:
  - 1xx : Messaggi informativi
  - 2xx : La richiesta è stata effettuata con successo
  - 3xx : Non c'è risposta immediata, ma la richiesta è sensata
  - 4xx : La richiesta non può essere effettuata perché errata
  - 5xx : La richiesta non può essere effettuata per un errore nel server



# RETE: RICHIESTE HTTP REST

- Gli header più comuni sono due:
  - Server: indica tipo e versione del server
  - Content-Type: indica il tipo di contenuto restituito. Alcuni fra questi:
    - **1 text/html**
    - **2 text/plain**
    - **3 text/xml**
    - **4 image/jpeg**
- Il body contiene il contenuto della risposta.

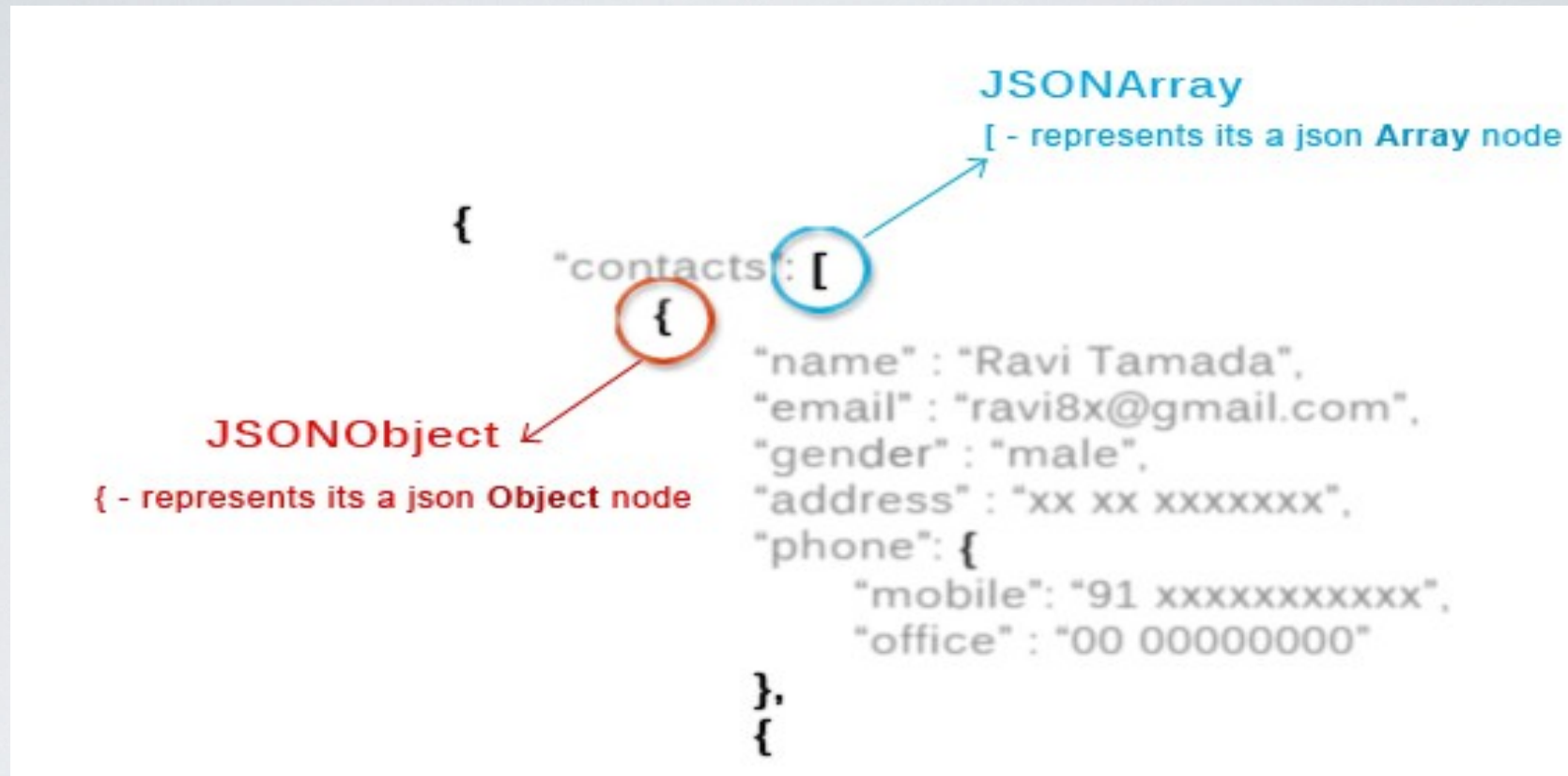
# GESTIONE DEL FORMATO JSON

- Il JSON (Javascript Object Notation) è un formato stringa per la rappresentazione di dati organizzati in oggetti e array.
- Negli ultimi anni, soprattutto grazie alla sua semplicità e al suo impiego in Ajax, ha acquisito una notevolissima popolarità a discapito di XML

# FORMATI JSON

- I tipi di dati supportati da questo formato sono:
- **booleani** (true e false)
- **numeri**: interi, reali, virgola mobile
- **stringhe** racchiuse da doppi apici "
- **array**: sequenze ordinate di valori, separati da virgole e racchiusi in parentesi quadre [ ]
- **array associativi**: sequenze coppie chiave-valore separate da virgole racchiuse in parentesi graffe





# STRUTTURA FORMATO SON

Notazione array e oggetti

# ESEMPIO JSON

- esempio Json di un array di oggetti:

```
[  
  {  
    "nome": "Roberto",  
    "eta": 52  
  },  
  {  
    "nome": "Antonio",  
    "eta": 32  
  },  
  {  
    "nome": "Francesca",  
    "eta": 40  
  }  
]
```

```
import http.client
import json

class Service(object):
    conn = None

    def __init__(self):
        self.conn = http.client.HTTPConnection("www.sito.it")

    def listLogFile(self):
        self.conn.request(method = "GET", url = "/service/file.php")
        res = self.conn.getresponse()
        jsonStr = str(res.read().decode("utf-8"))
        data = json.loads(jsonStr)

        self.conn.close()

        return data
```



# RETE: RICHIESTE HTTP REST

- Tale funzione può essere ingegnerizzata per usarla come di seguito:

- **GET**

```
risposta = requestHTTP("GET", url_sito, url_home_page)
```

- **POST**

```
risposta = requestHTTP("POST", url_sito, url_login,  
                        user = username,  
                        passwd = password,  
                        api_type = "json")
```

- **Nota:** consultare sempre l'API di un sito per accertarsi che la richiesta venga effettuata correttamente

# RETE: SOCKET

- Sockets sono i punti finali di un canale di comunicazione bidirezionale
- Con i socket si può comunicare all'interno di un processo, tra i processi sulla stessa macchina, o tra processi su diversi continenti
- Sockets possono essere attuate per un certo numero di diversi tipi di canali: socket di dominio Unix, TCP, UDP, e così via
- La libreria socket fornisce classi specifiche per la gestione dei servizi di trasporto comuni e un'interfaccia generica per gestire il resto

# RETE: SOCKET

- In informatica e telecomunicazioni una rete di calcolatori è un sistema o particolare tipo di rete di telecomunicazioni che permette lo scambio o condivisione di dati informativi e risorse (sia hardware sia software) tra diversi calcolatori



# RETE: SOCKET

- Per creare un socket, è necessario utilizzare il metodo `socket.socket ()` disponibile nel modulo `socket`, che ha la sintassi generale:  
`s = socket.socket (socket_family, socket_type, protocollo = 0)`
- Ecco la descrizione dei parametri:
  - **socket\_family**: Questo può essere `AF_UNIX` o `AF_INET`
  - **socket\_type**: Questo può essere `SOCK_STREAM` o `SOCK_DGRAM`
  - **Protocollo**: Questo è di solito lasciato fuori, inadempiente a 0
- Una volta che si ha l'oggetto `socket`, quindi è possibile utilizzare le funzioni necessarie per creare il programma client o server

# RETE: SOCKET

## Server

Metodo	Descrizione
s.bind()	Si lega indirizzo (nome host, numero di coppie della porta) al socket
s.listen()	Prevede l'iscrizione e iniziare ad ascoltare al TCP
s.accept()	Accetta passivamente la connessione client TCP, in attesa fino a quando il collegamento arriva (blocco)

## Client

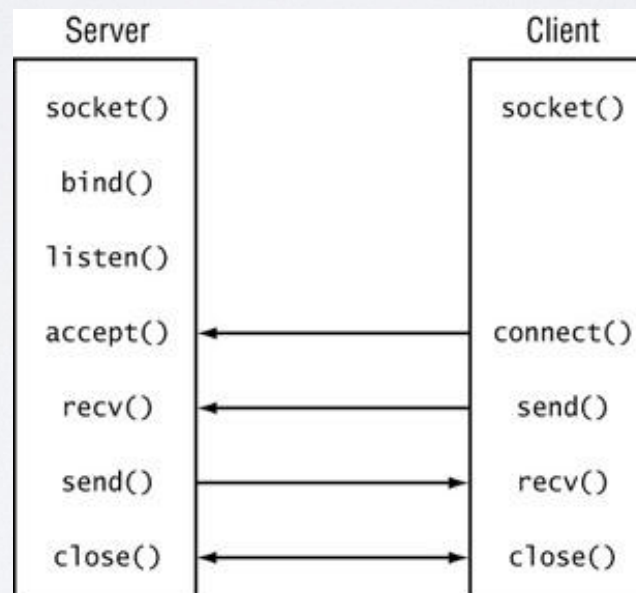
Metodo	Descrizione
s.connect()	Questo metodo avvia attivamente connessione al server TCP

## Comuni

Metodo	Descrizione
s.recv()	Riceve un messaggio TCP
s.send()	Trasmette un messaggio TCP
s.recvfrom()	Riceve il messaggio UDP
s.sendto()	Trasmette un messaggio UDP
s.close()	Chiude il socket
socket.gethostname()	Restituisce il nome host

# RETE: SOCKET

- L'immagine mostra la struttura principale del Server e del Client per una connessione TCP (Transmission Control Protocol)





# RETE: SOCKET SERVER

- La prima cosa da fare è importare la libreria socket:

```
import socket
```

- Quindi creiamo una socket per il server:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- La costante `socket.AF_INET` serve a specificare il dominio sul quale il network funzionerà. In questo caso la nostra socket sta definendo il proprio dominio sotto un IPv4. Altri domini sono: `socket.AF_INET6` e `socket.AF_UNIX`.

- La costante `socket.SOCK_STREAM` indica che stiamo creando una socket per una connessione TCP (per UDP usare: `socket.SOCK_DGRAM`)

- I due step successivi sono i seguenti:

```
server_socket.bind(('', 5000))
```

```
server_socket.listen(5)
```

# RETE: SOCKET SERVER

- La funzione `bind(address, port)` fissa la socket all'indirizzo e la porta specificati
- Per indicare "se stessi" si può semplicemente scrivere `"` oppure `"localhost"`
- In caso non si conosce l'URI dell'host, ma si conosce l'indirizzo IP possiamo utilizzare la funzione `socket.gethostbyaddr( indirizzo_IP)`
- La funzione `listen(n)` dove `n` è un number intero da 0 a 5, posiziona la socket in stato di ascolto. `n` server a specificare il numero di connessioni in coda
- Nota: il parametro da passare a `bind` è uno: una tupla di due elementi

# RETE: SOCKET SERVER

- Quindi accettiamo la connessione dal client e creiamo una socket specifica per la connessione TCP con il client  
`client_socket, address = server_socket.accept()`
- A questo punto creiamo un loop infinito, così da comunicare ripetutamente con il client

```
while 1:
    #Riceviamo i dati inviati dal client.
    data = client_socket.recv(512)
    print "Ricevuto:" , data

    #Controlliamo che il client non abbia inviato un comando di chiusura.
    if ( data == 'q' or data == 'Q'):
        client_socket.close()
        break;

    #Se non è un comando di chiusura facciamo qualcosa con i dati ricevuti.
    else:
        #FARE QUALCOSA CON I DATI.
        #Inviare i dati processati dal server.
        client_socket.send (data)
```



# RETE: SOCKET CLIENT

- Le funzioni sono identiche a quelle del server, solo che il client non ascolta per connessioni in entrata.
- Di seguito è il codice per il client.

```
import socket
```

```
#il parametro da passare a connect è uno: una tupla di due elementi.
```

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
client_socket.connect(("", 5000))
```

```
while 1:
```

```
    data = raw_input ( "Invia( Digita q or Q per chiudere):" )
```

```
    if (data <> 'Q' and data <> 'q'):
```

```
        client_socket.send(data)
```

```
        data = client_socket.recv(512)
```

```
        print("Ricevuto:" , data)
```

```
    else:
```

```
        client_socket.send(data)
```

```
        client_socket.close()
```

```
        break
```

# RETE: CLIENT-SERVER UDP

- Lo User Datagram Protocol (UDP) è il protocollo principale (insieme a quello TCP) a livello di trasporto nella comunicazione fra hosts
- L'UDP è :
  1. connectionless: un messaggio non viene mandato a un altro host tramite un apposito stream di dati stabilito in precedenza. I dati quindi possono facilmente essere persi senza che chi li invia ne venga a conoscenza
  2. I dati ricevuti non sono necessariamente in ordine. Un file di dimensione  $A$ , infatti, viene suddiviso in  $n$  pacchetti di dimensione  $A/n$  e ognuno di essi viene inviato separatamente. Per ricostruire  $A$ , tuttavia, i pacchetti devono essere "uniti" nell'ordine esatto
  3. Rapido (ottimo per l'invio di Audio/Video in streaming)
  4. Supporta il Broadcast (dati inviati a tutti nel network locale) e il Multicasting (dati inviati a tutti coloro che lo richiedono)
  5. Non supporta il controllo di congestione della rete

# RETE: CLIENT-SERVER UDP

- Possiamo ben notare che questo protocollo presenta alcuni pregi, ma anche molti difetti. Allora perché preoccuparsi tanto?
- L'UDP, a differenza del TCP, è molto più flessibile. Volendo è possibile costruire un proprio protocollo TCP partendo dall'UDP. In poche parole l'UDP è flessibile (personalizzabile).
- Un pacchetto dati UDP (Datagram) contiene le seguenti principali informazioni:
  1. Indirizzo mittente (IP + Porta) (Facoltativo)
  2. Indirizzo destinatario (IP + Porta)
  3. Lunghezza del pacchetto in ottetti
  4. I dati da inviare



# RETE: CLIENT-SERVER UDP

# Mittente

```
import socket
```

```
# Indirizzo destinatario (in questo caso è lo stesso host del mittente)
```

```
IP_UDP = "127.0.0.1" # "localhost" o ""
```

```
PORTA_UDP = 5005
```

```
MESSAGGIO = "Un saluto a tutti"
```

```
# AF_INET --> Uso del protocollo IP /IPv4
```

```
# usare AF_INET6 per IPv6 se disponibile
```

```
# SOCK_DGRAM --> Crea una socket di tipo Datagram (UDP)
```

```
sock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
```

```
# Invia il messaggio all'indirizzo specificato
```

```
# L'indirizzo è sotto forma di una TUPLA.
```

```
sock.sendto(MESSAGGIO, (IP_UDP, PORTA_UDP))
```

# RETE: CLIENT-SERVER UDP

# Ricevente

```
import socket
```

```
IP_UDP = "127.0.0.1"
```

```
PORTA_UDP = 5005
```

```
sock = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
```

```
# L'indirizzo specificato è vincolato per la socket sock.
```

```
sock.bind( (UDP_IP,UDP_PORT) )
```

```
# Ascolta ripetutamente per dei dati.
```

```
while True:
```

```
    # Riceve 1024 bytes alla volta.
```

```
    messaggio, indirizzo = sock.recvfrom( 1024 )
```

```
    print("Messaggio ricevuto :", messaggio)
```

# RETE: MODULI

Lista dei più importanti moduli in Python

Protocollo	Funzioni	Porta	Modulo Python
HTTP	Pagine web	80	httplib, urllib, xmlrpclib
NNTP	News	119	nntplib
FTP	Trasferimento file	20	ftplib, urllib
SMTP	Invio di email	25	smtpplib
POP3	Richiesta email	110	poplib
IMAP4	Richiesta email	143	imaplib
Telnet	Linea comandi	23	telnetlib



# RETE: FTP

- Per chi volesse usare FTP in Python si possono far due scelte: usare la libreria `ftplib`, dedicata per il trasferimento dei file sotto il protocollo FTP oppure usare una normalissima connessione TCP (opzione più complessa)
- Quando si trasferisce un file vi sono sempre due parti: chi manda e chi riceve.

```
import ftplib # Importa il modulo FTP
session = ftplib.FTP('nomeserver.it', 'login', 'passord') #
Connessione al server FTP
myfile = open('test.txt', 'rb') # Apre il file da inviare
session.storbinary('filetest.txt', myfile) # Invia il file
myfile.close() # Chiude il file
session.quit() # Chiude la sessione FTP
```

# RETE:TELNET

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

# ESERCIZIO

- Proseguire con l'implementazione del Gestionale di Teatro usando i dati generati dai seguenti servizi web:
- <http://letus.vfdns.org/java/teatro/teatro.json>
- <http://letus.vfdns.org/java/teatro/spettacoli.json>
- <http://letus.vfdns.org/java/teatro/programmazione.json>
- <http://letus.vfdns.org/java/teatro/venditabiglietti.json>
- In seguito memorizzare tutti i dati in sqlite e mettere a disposizione una interfaccia che consenta di acquistare biglietti o presentarsi per entrare a teatro a vedere lo spettacolo acquistato