

---

# Introduzione a



MONTY PYTHON'S  
FLYING  
CIRCUS

# Introduzione a Python

---

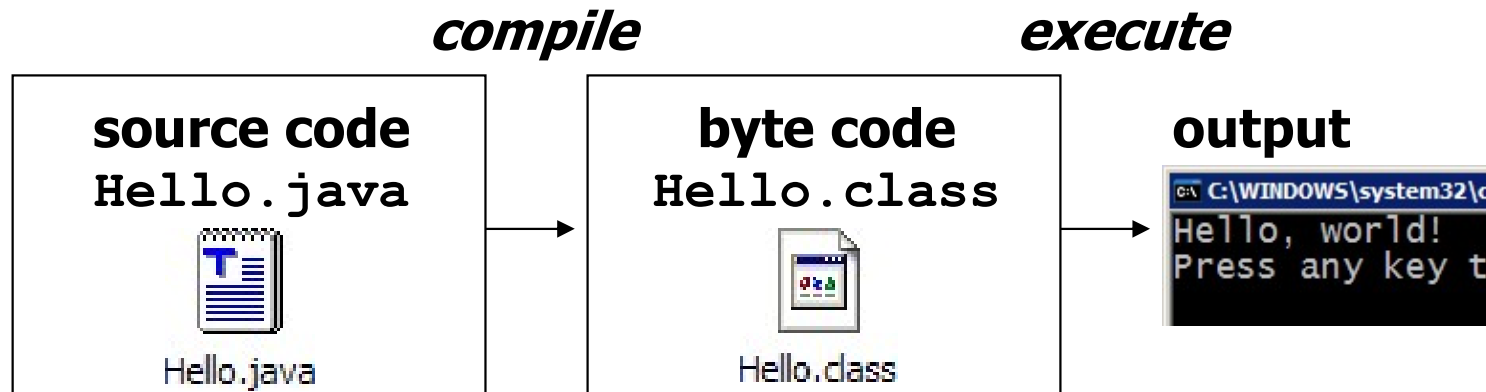
- Linguaggio di programmazione sviluppato agli inizi degli anni 90 presso il Centrum Wiskunde & Informatica (CWI)
  - Ideato da Guido van Rossum *per tenersi occupato* durante le vacanze di Natale del 1989
  - Il nome non deriva da pitone, ma dalla serie televisiva *Il circo volante dei Monty Python* amata da Van Rossum
- Sintassi snella
  - Molte funzionalità presenti nelle librerie del linguaggio



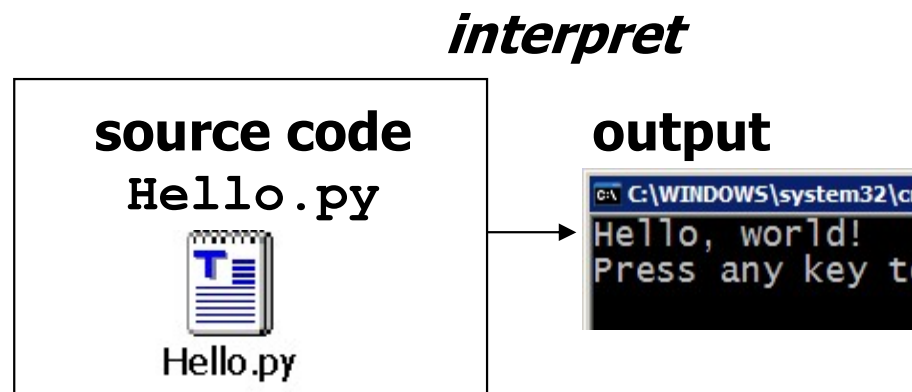
# Compilazione ed interpretazione

---

- Python non richiede che il programma sia compilato



- Python interpreta direttamente



# Il linguaggio Python

---

- È un linguaggio di programmazione:
  - Interpretato
  - Di alto livello
  - Semplice da imparare e usare
  - Potente e produttivo
  - Ottimo anche come primo linguaggio (molto simile allo pseudocodice)
  - Tipizzazione forte e dinamica
- Inoltre
  - È open source ([www.python.org](http://www.python.org))
  - È multiplatforma
  - È facilmente integrabile con C/C++ e Java

# Python: linguaggio interpretato

---

- I comandi sono eseguiti da un interprete
  - L'interprete riceve un comando, valuta il comando e restituisce il risultato del comando
- Rispetto ad altri linguaggi interpretati (e.g. Perl), fornisce maggiori funzionalità rendendolo più simile ad un linguaggio compilato, ed è più intuitivo
- Un programmatore memorizza una serie di comandi in un file di testo a cui faremo riferimento con il termine codice sorgente o script (modulo)
- Convenzionalmente il codice sorgente è memorizzato in un file con estensione `.py`
  - `ciao.py`

# Python & SNAP

---

- **Stanford Network Analysis Platform (SNAP)**: è un sistema general purpose ad alta performance creato per l'analisi e la manipolazione di reti molto grandi.
  - <http://snap.stanford.edu/index.html>
- E' in pratica una libreria di funzioni che consente di manipolare reti con milioni di nodi e bilioni di archi
- Il codice SNAP è originariamente stato scritto in C++, ma è fornita anche una interfaccia Python: **Snap.py**
  - <http://snap.stanford.edu/snappy/index.html>
- Snap.py richiede che sia installato **Python 3.x** sulla propria macchina.
  - **Seguire le istruzioni presenti su**  
<http://snap.stanford.edu/snappy/index.html>  
**per il download e l'installazione di Snap.py**

# Documentazione Python

---

- Sito ufficiale Python
  - <https://docs.python.org>
- Tutorial Python
  - <https://docs.python.org/3.x/tutorial/index.html>
- Documentazione in italiano
  - <http://docs.python.it/>
- Assicuratevi che la documentazione sia per Python 3.x

# Python

---

- Vedremo velocemente alcune delle funzionalità più comuni di Python
- Cercheremo di capirne la logica e l'utilizzo generale
- Esistono le librerie più varie scritte in python che ne facilitano l'utilizzo
- Per la gestione più specifica dei grafi farete riferimento al sito  
<http://snap.stanford.edu/snappy/index.html>



# Python: l'interprete interattivo

---

- L'interprete è un file denominato
  - “python” su Unix
  - “python.exe” su Windows
- I comandi si possono inserire direttamente dallo standard input
  - Il prompt è caratterizzato da “>>> ”
  - Se un comando si estende sulla riga successiva è preceduto da “...”
- I file sorgente Python sono file di testo, generalmente con estensione “.py”

# Python: l'interprete interattivo

---

- Avvio: aprire una shell
- Digitare `python` al prompt della shell (o la path dove si trova l'eseguibile di python)
- appare ora il prompt `>>>` pronto a ricevere comandi
  - Possiamo ora inserire qualsiasi costrutto del linguaggio e vedere immediatamente l'output:

```
>>> 3+5
```

```
8
```

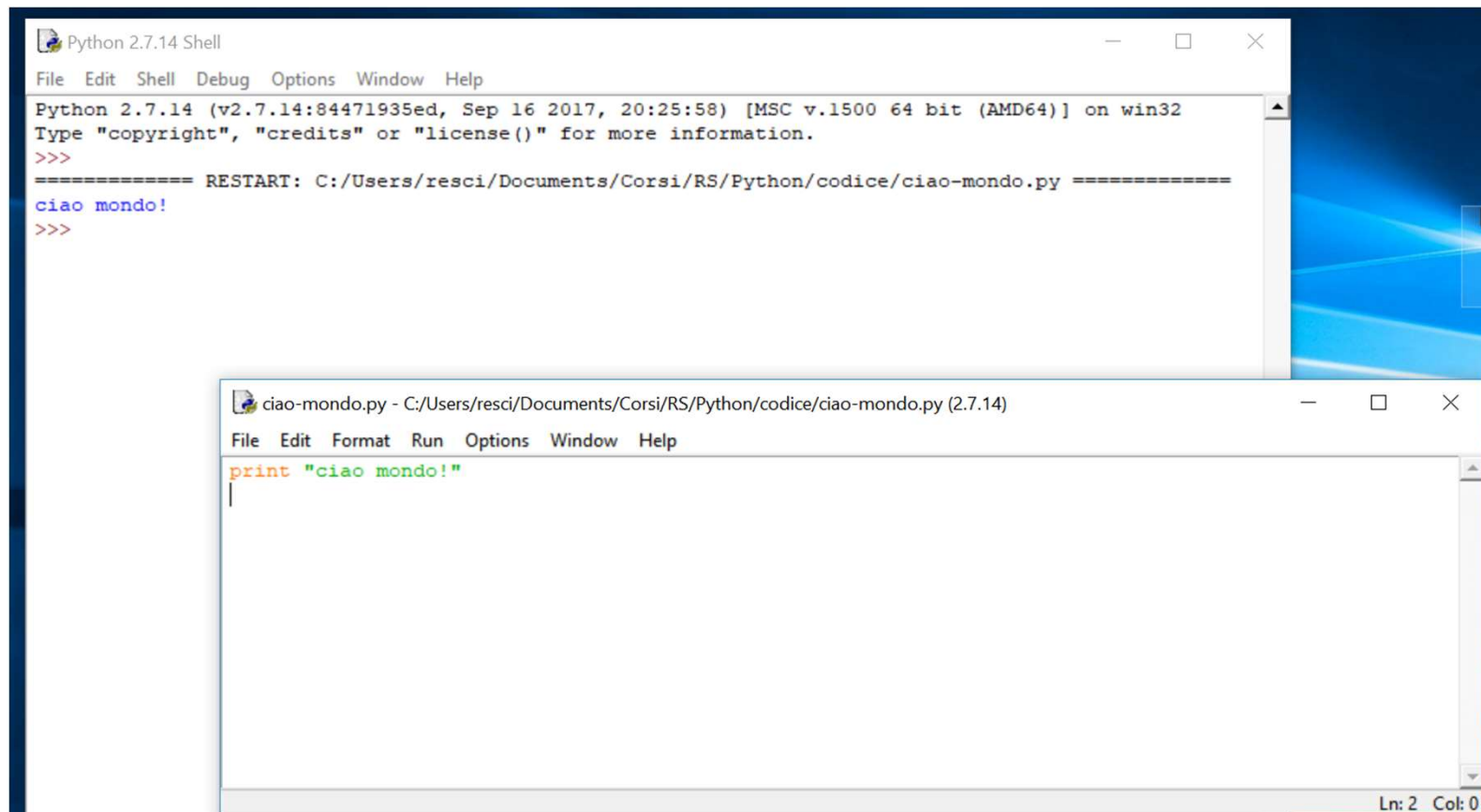
```
>>> print('Hello world!')
```

```
Hello world!
```

# IDLE

---

- IDLE
  - Ambiente di sviluppo integrato in Python
  - Avviate IDLE selezionandolo dal menu dei programmi



# File sorgente

---

- I file sorgente Python sono file di testo, generalmente con estensione .py

`python nomeProgramma.py`

- Il simbolo “#” inizia un commento che si estende fino a fine riga
- Import dei moduli richiesti dal programma
  - Subito dopo il commento introduttivo
- Definizione delle funzioni
  - Tra cui la funzione main (**non è necessaria**)
- Docstring per ogni funzione definita nel modulo
- Uso di nomi significativi

# File sorgente

---

- Le istruzioni sono separate dal fine riga e non da “;”
  - Il “;” può comunque essere usato per separare istruzioni sulla stessa riga ma è sconsigliato
- Per far continuare un’istruzione anche sulla linea successiva è necessario inserire un “\” a fine riga
- Se le parentesi non sono state chiuse correttamente Python capisce che l’istruzione si estende anche sulla riga successiva

# Funzione main

---

- Non è necessaria introdurla
  - Non succede come in C o Java dove la funzione main è invocata quando il programma è eseguito
- Il codice è organizzato meglio ed è più leggibile

Ad esempio

```
def main():  
    while not s.is_empty():  
        print(s.pop(), end=" ")  
  
if __name__ == '__main__':  
    main()
```

# Alcuni concetti introduttivi

---

- Per capire il resto della presentazione serve sapere alcune cose
  - Le funzioni vengono chiamate come in C
    - `foo(5, 3, 2)`
  - “ogg.f()” è un metodo
  - I metodi possono inizialmente essere considerati come delle funzioni applicate sull’oggetto prima del punto
  - Il seguente codice Python:
    - `"CIAO".lower()`
  - può essere pensato equivalente al seguente codice C:
    - `stringa_lower("CIAO");`

# input e output

---

- L'istruzione “print” stampa il suo argomento trasformandolo in una stringa

```
>>> print(5)
```

```
5
```

```
>>> print('Hello world')
```

```
Hello world
```

- A “print” possono essere passati più argomenti separati da un virgola. Questi sono stampati separati da uno spazio

```
>>> print(1, 2, "xxx")
```

```
1 2 xxx
```



# input e output

---

- Si può formattare l'output come il c:

```
>>> x=18; y=15
```

```
>>> print ("x=%d y=%d\n" % (x,y))
```

```
x=18 y=15
```

- Per leggere una stringa si usa input()

```
>>> x=input('Scrivi un numero:')
```

interpreta come stringa anche se si dà un numero; se si vuole dare una interpretazione diversa, anteporre il tipo

- Per leggere una numero

```
>>> x=int(input('Scrivi un numero:'))
```

# Variabili

---

- I nomi di variabili sono composti da lettere, numeri e underscore, il primo carattere non può essere un numero (come in C)
  - Sono validi:
    - “x”, “ciao”, “x13”, “x1\_y”, “\_”, “\_ciao12”
  - Non sono validi:
    - “1x”, “x-y”, “\$a”, “àñÿô”
- Le variabili non devono essere dichiarate (tipizzazione dinamica)
- Una variabile non può essere utilizzata prima che le venga assegnato un valore
- Ogni variabile può riferirsi ad un oggetto di qualsiasi tipo

# Variabili

---

Esempi:

```
>>> x=5
```

```
>>> nome="Marco"
```

- Sintetico

```
>>> inizio,fine=2,100
```

- type restituisce il tipo di una variabile

```
>>> x=[ 5, 3]
```

```
>>> type (x)
```

```
<type 'list'>
```

x è di tipo lista (verrà vista più avanti)

# Assegnamento

---

- L'assegnamento avviene attraverso l'operatore “=”
- Non è creata una copia dell'oggetto:
  - `x = y` # si riferiscono allo stesso oggetto
- Esempio:

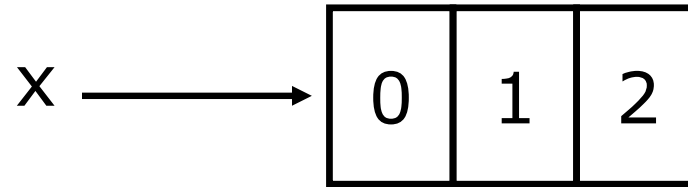
```
>>> x = [0, 1, 2]
>>> y = x
>>> x.append(3)
>>> print(y)
[0, 1, 2, 3]
```

# Assegnamento (2)

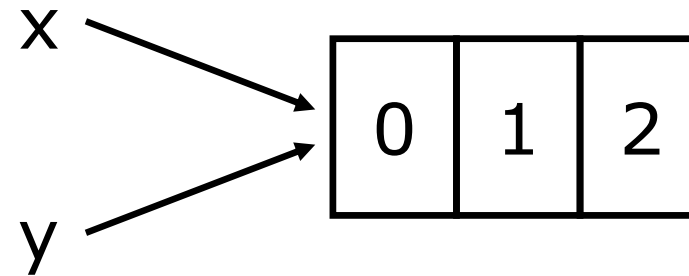
---

- Ecco quello che succede:

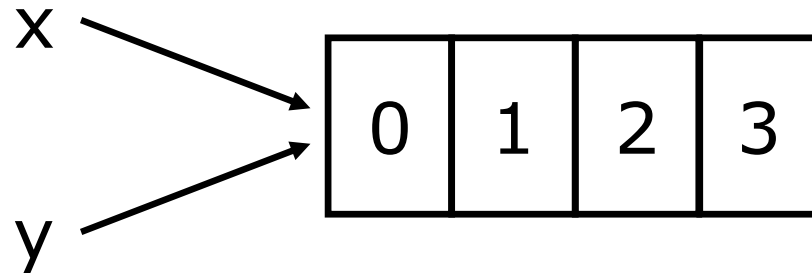
`x = [0, 1, 2]`



`y = x`



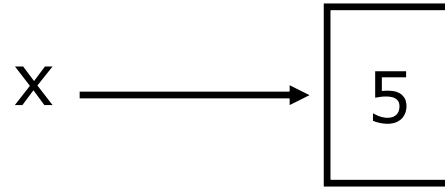
`x.append(3)`



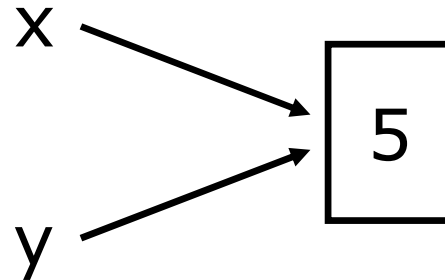
# Assegnamento (3)

---

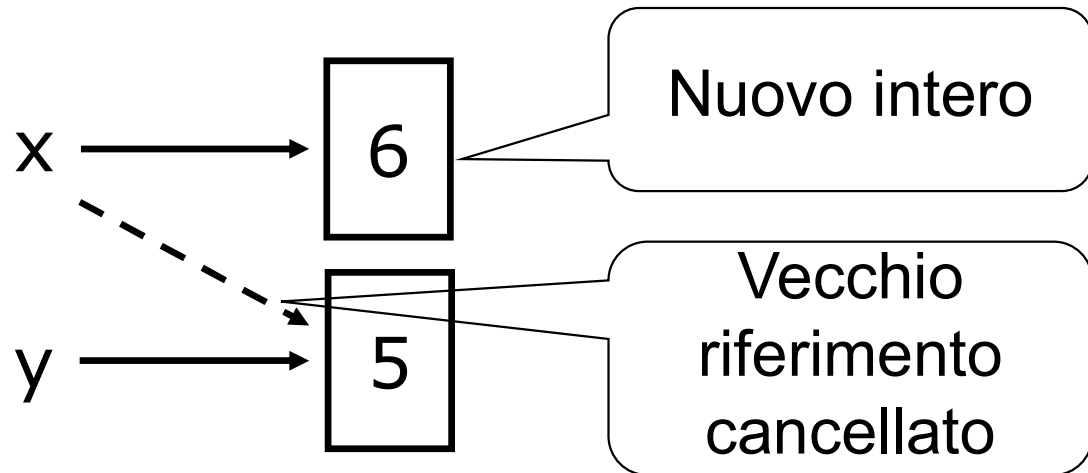
**x = 5**



**y = x**



**x = x + 1**



# Semantica Dinamica

---

- Il **tipo** di una variabile (intero, carattere, virgola mobile, ...) è basato sull'utilizzo della variabile e non deve essere specificato prima dell'utilizzo
- La variabile può essere riutilizzata nel programma e il suo tipo può cambiare in base alla necessità corrente

script

```
a = 3
print(a, type(a))
a = "casa"
print(a, type(a))
a = 4.5
print(a, type(a))
```

output

```
3 <class 'int'>
casa <class 'str'>
4.5 <class 'float'>
```

# Oggetti in Python

---

- Python è un linguaggio orientato agli oggetti e le classi sono alla base di tutti i tipi di dati
- Alcune classi predefinite in Python
  - La classe per i numeri interi **int**
  - La classe per i numeri in virgola mobile **float**
  - La classe per le stringhe **str**
- Per istanziare un oggetto invochiamo il costruttore della classe

```
t = 23.7      crea una nuova istanza  
              della classe float
```



# Oggetti mutable/immutable

---

- Oggetti il cui valore può cambiare sono chiamati *mutable*
- Una classe è *immutable* se ogni oggetto della classe una volta inizializzato non può essere modificato in seguito
- Un oggetto contenitore *immutable* che contiene un *referimento* ad un oggetto *mutable*, può cambiare quando l'oggetto contenuto cambia
  - Il contenitore è considerato *immutable* perché la collezione di oggetti che contiene non può cambiare

# Classi built-in

---

Class	Description	Immutable?
<b>bool</b>	Boolean value	✓
<b>int</b>	integer (arbitrary magnitude)	✓
<b>float</b>	floating-point number	✓
<b>list</b>	mutable sequence of objects	
<b>tuple</b>	immutable sequence of objects	✓
<b>str</b>	character string	✓
<b>set</b>	unordered set of distinct objects	
<b>frozenset</b>	immutable form of set class	✓
<b>dict</b>	associative mapping (aka dictionary)	

```
i = int(3)
print(i)
print(i.bit_length())
```



```
3
2
```

# Classe **bool**

---

- La classe **bool** è usata per rappresentare i valori booleani **True** e **False**
- Il costruttore **bool()** restituisce **False** di default
- Python permette la creazione di valori booleani a partire da valori non-booleani **bool(foo)**
- L'interpretazione dipende dal valore di foo
  - I numeri sono interpretati come **False** se uguali a 0, **True** altrimenti
  - Sequenze ed altri tipi di contenitori sono valutati **False** se sono vuoti, **True** altrimenti

```
b = bool(False)
if b == False:
    print('La variabile b è ', b)
else:
    print('La variabile b è True')
```

# Classe `int`

---

- La classe `int` è usata per rappresentare i valori interi di grandezza arbitraria
- Il costruttore `int()` restituisce `0` di default
- È possibile creare interi a partire da `stringhe` che rappresentano numeri in qualsiasi base tra 2 e 35 (2, 3, ..., 9, A, ..., Z)

```
i = int(7598234798572495792375243750235437503)
print('numero di bit: ', i.bit_length())
```

output `numero di bit: 123`

```
i = int("23", base=4)
print('la variabile vale: ', i)
```

output `la variabile vale: 11`

# Classe **float**

---

- La classe **float** è usata per rappresentare i valori razionali di grandezza arbitraria
- Il costruttore **float()** restituisce **0.0** di default
- La classe float ha vari metodi, ad esempio possiamo rappresentare il valore come rapporto di interi

```
f= 0.321123  
print(f, '=', f.as_integer_ratio())
```

```
0.321123 = (5784837692560383, 18014398509481984)
```

# Classe **float**

---

- L'istruzione `t = 23.7` crea una nuova istanza immutabile della classe **float**
- Lo stesso succede con l'istruzione `t = float(3.8)`
- `t + 4` automaticamente invoca `t.__add__(4)`
  - overloading dell'operatore `+`

```
f1 = float(3.8)  
print('operatore +: ', f1+4)
```

```
operatore +: 7.8
```

# Sequenze

---

- In Python le classi **list**, **tuple** e **str** sono tipi **sequenza**
  - Una sequenza rappresenta una collezione di valori in cui l'ordine è rilevante (non significa che gli elementi sono ordinati)
  - Ogni elemento della sequenza ha una posizione
    - Se ci sono  $n$  elementi, il primo elemento è in posizione 0, mentre l'ultimo è in posizione  $n-1$

# Oggetti iterable

---

- Un oggetto è ***iterable*** se
  - Contiene *alcuni elementi*
  - È in grado di *restituire* i suoi elementi uno alla volta
- Stesso concetto di **Iterable** in Java

```
List list = new ArrayList();  
  
for(Object o : list){  
    //Utilizza o  
}
```

Java

```
lst = list([1, 2, 3])  
  
for o in lst:  
    //Utilizza o
```

Python



# Classe **list**

---

- Un'istanza dell'oggetto lista memorizza una sequenza di oggetti
  - Una sequenza di riferimenti (puntatori) ad oggetti nella lista
- Gli elementi di una lista possono essere oggetti arbitrari (incluso l'oggetto **None**)
- Python usa i caratteri **[ ]** come delimitatori di una lista
  - `[]` lista vuota
  - `['red', 'green', 'blue']` lista con tre elementi
  - `[3, 4.9, 'casa']` lista con tre elementi

# Classe **list**

---

- Il costruttore **list**() restituisce una lista vuota di default
- Il costruttore **list**() accetta un qualsiasi parametro iterabile
  - **list**('ciao') produce una lista di singoli caratteri ['c', 'i', 'a', 'o']
- Una lista è una sequenza concettualmente simile ad un array
  - una lista di lunghezza n ha gli elementi indicizzati da 0 ad n-1
- Le liste hanno la capacità di espandersi e contrarsi secondo la necessità corrente

# Metodi di **list**

---

- `list.append(x)`
  - Aggiunge l'elemento `x` alla fine della lista `list`
- `list.extend(iterable)`
  - Estende la lista aggiungendo tutti gli elementi dell'oggetto *iterable*
  - `a.extend(b)` è equivalente a `a[len(a):] = b`
- `list.insert(i, x)`
  - Inserisce l'elemento `x` nella posizione `i`
  - `p.insert(0, x)` inserisce `x` all'inizio della lista `p`
  - `p.insert(len(p), x)` inserisce `x` alla fine della lista `p` (equivalente a `p.append(x)`)

`len(b)` restituisce  
il numero degli  
elementi in `b`

# Concatenazione di liste

---

```
a = list([1, 2, 3])
print('id =', id(a), ' a =', a)
b = list([4, 5])
print('id =', id(b), ' b =', b)
a.extend(b)
print('id =', id(a), ' a =', a)
a += b
print('id =', id(a), ' a =', a)
a = a + b
print('id =', id(a), ' a =', a)
```

```
id = 4321719112  a = [1, 2, 3]
id = 4321719176  b = [4, 5]
id = 4321719112  a = [1, 2, 3, 4, 5]
id = 4321719112  a = [1, 2, 3, 4, 5, 4, 5]
id = 4321697160  a = [1, 2, 3, 4, 5, 4, 5, 4, 5]
```

# Metodi di **list**

---

- `list.remove(x)`
  - Rimuove la prima occorrenza dell'elemento x dalla lista.  
Genera un errore se x non c'è nella lista
- `list.pop(i)`
  - Rimuove l'elemento in posizione i e lo restituisce
  - `a.pop()` rimuove l'ultimo elemento della lista
- `list.clear()`
  - Rimuove tutti gli elementi dalla lista

# Metodi di **list**

---

- `list.index(x, start, end)`
  - Restituisce l'indice della prima occorrenza di x compreso tra start ed end (opzionali)
  - L'indice è calcolato a partire dall'inizio (indice 0) della lista
- `list.count(x)`
  - Restituisce il numero di volte che x è presente nella lista
- `list.reverse()`
  - Inverte l'ordine degli elementi della lista
- `list.copy()`
  - Restituisce una copia della lista

# Esempio

---

```
l = [3, '4', 'casa']  
l.append(12)  
print('l =', l)  
d = l  
print('d =', d)  
d[3] = 90  
print('d =', d)  
print('l =', l)
```

```
l = [3, '4', 'casa', 12]  
d = [3, '4', 'casa', 12]  
d = [3, '4', 'casa', 90]  
l = [3, '4', 'casa', 90]
```

d ed l fanno riferimento  
allo stesso oggetto

```
a = [3, 4, 5, 4, 4, 6]  
print('a =', a)  
print('Indice di 4 in a:', a.index(4))  
print('Indice di 4 in a tra 3 e 6:', a.index(4, 3, 6))
```

```
a = [3, 4, 5, 4, 4, 6]  
Indice di 4 in a: 1  
Indice di 4 in a tra 3 e 6: 3
```

# Ordinare una lista

---

- `list.sort(key=None, reverse=False)`
  - Ordina gli elementi della lista, `key` e `reverse` sono opzionali
  - A `key` si assegna il nome di una funzione con un solo argomento che è usata per estrarre da ogni elemento la chiave con cui eseguire il confronto
  - A `reverse` si può assegnare il valore `True` se si vuole che gli elementi siano in ordine decrescente

```
a = [3 ,4, 5, 4, 4, 6]  
a.sort(reverse=True)  
print a
```

```
[6, 5, 4, 4, 4, 3]
```



# Classe **tuple**

---

- Fornisce una versione immutabile di una lista
- Python usa i caratteri **( )** come delimitatori di una tuple
- L'accesso agli elementi della tuple avviene come per le liste
- La tupla vuota è **()**, quella con un elemento è **(12,)**

```
t = (3 ,4, 5, '4', 4, '6')  
print('t =', t)  
print('Lunghezza t =', len(t))
```



```
t = (3, 4, 5, '4', 4, '6')  
Lunghezza t = 6
```

# tuple packing/unpacking

---

- Il packing è la creazione di una tuple
- L'**unpacking** è la creazione di variabili a partire da una tuple

```
t = (1, 's', 4)
x, y, z = t
print('t =', t, type(t))
print('x =', x, type(x))
print('y =', y, type(y))
print('z =', z, type(z))
```



```
t = (1, 's', 4) <class 'tuple'>
x = 1 <class 'int'>
y = s <class 'str'>
z = 4 <class 'int'>
```

# Ancora su mutable/immutable

---

```
lst = ['a', 1, 'casa']
tpl = (lst, 1234)
print('list  =', lst)
print('tuple =', tpl)
tpl[0] = 0
print(tpl[0])
```

```
lst.append('nuovo')
print('list  =', lst)
print('tuple =', tpl)
```

```
list  = ['a', 1, 'casa']
tuple = (['a', 1, 'casa'], 1234)
```

```
TypeError: 'tuple' object does not support item assignment
['a', 1, 'casa']
```

```
list  = ['a', 1, 'casa', 'nuovo']
tuple = (['a', 1, 'casa', 'nuovo'], 1234)
```

# Classe **str**

---

- Le stringhe (sequenze di caratteri) possono essere racchiuse da apici singoli o apici doppi
- Si usano tre apici singoli o doppi per stringhe che contengono newline (sono su più righe)
- Nei manuali dettagli sui metodi di **str**

```
s = '''Il Principe dell'Alba  
    si mette in cammino venti  
    minuti prima delle quattro.'''  
print(s)
```

```
Il Principe dell'Alba  
    si mette in cammino venti  
    minuti prima delle quattro.
```

# Classe **set**

---

- La classe **set** rappresenta la nozione matematica dell'insieme
  - Una collezione di elementi senza duplicati e senza un particolare ordine
- Può contenere **solo** istanze di oggetti **immutable**
- Si usano le parentesi graffe per indicare l'insieme { }
- L'insieme vuoto è creato con set()

```
ins = {2, 4, '4'}  
print(ins)
```



```
{2, '4', 4}
```

L'ordine dell'output dipende dalla  
rappresentazione interna di set

# Classe **set**

---

- Il costruttore **set**() accetta un qualsiasi parametro iterabile
  - `a=set('buongiorno')` → `a={'o', 'u', 'i', 'b', 'r', 'g', 'n'}`
- `len(a)` restituisce il numero di elementi di `a`
- `a.add(x)`
  - Aggiunge l'elemento `x` all'insieme `a`
- `a.remove(x)`
  - Rimuove l'elemento `x` dall'insieme `a`
- Altri metodi li vediamo in seguito
  - Dettagli sul manuale

# Classe **frozenset**

---

- È una classe immutabile del tipo **set**
  - Si può avere un set di frozenset
- Stessi metodi ed operatori di **set**
  - Si possono eseguire facilmente test di (non) appartenenza, operazioni di unione, intersezione, differenza, ...
- Dettagli maggiori quando analizzeremo gli operatori
  - Per ogni operatore esiste anche la *versione* metodo

# Classe **dict**

---

- La classe **dict** rappresenta un dizionario
  - Un insieme di coppie (chiave, valore)
  - Le chiavi devono essere distinte
  - Implementazione in Python simile a quella di set
- Il dizionario vuoto è rappresentato da **{ }**
- Un dizionario si crea inserendo nelle **{ }** una serie di coppie chiave:valore separate da virgola
  - `d = {'ga' : 'Irish', 'de' : 'German'}`
  - Alla chiave **de** è associato il valore **German**
- Il costruttore accetta una sequenza di coppie (chiave, valore) come parametro
  - `d = dict(pairs)` dove `pairs = [('ga', 'Irish'), ('de', 'German')]`.



# Esempi classe dict

---

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print('tel =', tel)
tel['irv'] = 4127
print('tel =', tel)
del tel['sape']
print('tel =', tel)
```

```
tel = {'jack': 4098, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127}
```

# Esempi classe dict


---

```
chiavi = tel.keys()
print('chiavi =', chiavi)
valori = tel.values()
print('valori =', valori)
for i in chiavi:
    print(i)
```

```
for i in tel.keys():
    print(i)
```

```
chiavi = dict_keys(['guido', 'irv', 'jack'])
valori = dict_values([4127, 4127, 4098])
guido
irv
jack
```

```
elementi = tel.items()
for k,v in elementi:
    print(k,v)
```



```
irv 4127
guido 4127
jack 4098
```

# Alcuni metodi classe **dict**

---

- **diz.clear()**
  - Rimuove tutti gli elementi da **diz**
- **diz.copy()**
  - Restituisce una copia superficiale (shallow) di **diz**
- **diz.get(k)**
  - Restituisce il valore associato alla chiave **k**
- **diz.pop(k)**
  - Rimuove la chiave **k** da **diz** e restituisce il valore ad essa associato
- **diz.update([other])**
  - Aggiorna **diz** con le coppie chiave/valore in **other**, sovrascrive chiavi esistenti
  - **update** accetta come input o un dizionario o un oggetto iterabile di coppie chiave/valore

# Esempio di update

---

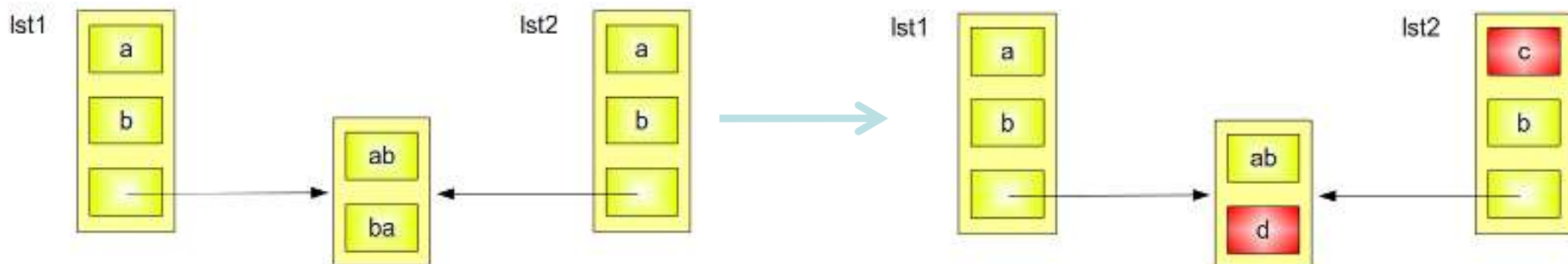
```
print('tel =', tel)
tel2 = {'guido': 1111, 'john': 666}
print('tel2 =', tel2)
tel.update(tel2)
print('tel =', tel)
tel.update([('mary', 1256)])
print('tel =', tel)
```

```
tel={'irv': 4127, 'guido': 4127, 'jack': 4098}
tel2={'guido': 1111, 'john': 666}
tel={'guido': 1111, 'john': 666, 'irv': 4127, 'jack': 4098}
tel={'guido': 1111, 'mary': 1256, 'john': 666, 'irv': 4127, 'jack': 4098}
```

# Esempio shallow/deep copy

```
lst1 = ['a', 'b', ['ab', 'ba']]
lst2 = lst1.copy()
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'd']]
lista2 = ['c', 'b', ['ab', 'd']]
```



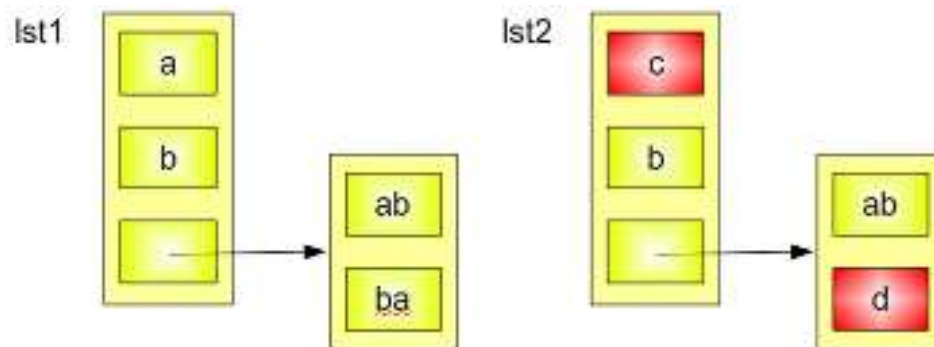
# Esempio shallow/deep copy

---

```
from copy import deepcopy
lst1 = ['a', 'b', ['ab', 'ba']]
lst2 = deepcopy(lst1)
print('lista1 =', lst1)
print('lista2 =', lst2)

lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['c', 'b', ['ab', 'd']]
```



# Espressioni ed operatori

---

- Espressioni esistenti possono essere combinate con simboli speciali o parole chiave (operatori)
- La semantica dell'operatore dipende dal tipo dei suoi operandi

```
a=3
b=4
c=a+b
print('a+b =', c)
a='ciao '
b='mondo '
c=a+b
print('a+b =', c)
```

```
a+b = 7
a+b = ciao mondo
```

# Operatori aritmetici

---

- Gli operatori aritmetici sono quelli a destra

+	addition
-	subtraction
*	multiplication
/	true division
//	integer division
%	the modulo operator

- Per gli operatori +, -, \*
  - Se entrambi gli operandi sono **int**, il risultato è **int**
  - Se uno degli operandi è **float**, il risultato è **float**
- Per la divisione **vera** /
  - Il risultato è sempre float
- Per la divisione intera //
  - Il risultato (**int**) è la parte intera della divisione

// e % definiti anche  
per numeratore o  
denominatore negativo.  
**Dettagli sul manuale**



# Operatori logici -- Operatori di uguaglianza

---

- Python supporta i seguenti operatori logici

**not** unary negation

**and** conditional and

**or** conditional or

- Python supporta i seguenti operatori di uguaglianza

**is** same identity

**is not** different identity

**==** equivalent

**!=** not equivalent

# Operatori di uguaglianza

---

- L'espressione

a **is** b

risulta vera solo se a e b sono alias dello stesso oggetto

- L'espressione

a **==** b

risulta vera anche quando gli identificatori a e b si riferiscono ad oggetti che possono essere considerati equivalenti

- Due oggetti dello stesso tipo che *contengono* gli stessi valori

# Esempio

```
lst1 = ['a', 'b', ['ab', 'ba']]
lst2 = lst1
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')
if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti identici  
Oggetti equivalenti

```
lst1 = ['a', 'b', ['ab', 'ba']]
lst2 = lst1.copy()
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')
if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti distinti  
Oggetti equivalenti

# Operatori di confronto

---

- Python supporta i seguenti operatori di confronto

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Per gli interi hanno il significato atteso
- Per le stringhe sono case-sensitive e considerano l'ordinamento lessicografico
- Per sequenze ed insiemi assumono un significato particolare (dettagli in seguito)

# Operatori bitwise

---

- Gli **interi** supportano i seguenti operatori orientati ai bit

$\sim$  bitwise complement (prefix unary operator)

$\&$  bitwise and

$|$  bitwise or

$\wedge$  bitwise exclusive-or

$\ll$  shift bits left, filling in with zeros

$\gg$  shift bits right, filling in with sign bit

# Operatori per sequenze **list**, **tuple** e **str**

---

- I tipi sequenza predefiniti in Python supportano i seguenti operatori

<code>s[j]</code>	element at index <i>j</i>
<code>s[start:stop]</code>	slice including indices [start,stop)
<code>s[start:stop:step]</code>	slice including indices start, start + step, start + 2*step, ..., up to but not equalling or stop
<code>s + t</code>	concatenation of sequences
<code>k * s</code>	shorthand for <code>s + s + s + ...</code> (k times)
<code>val in s</code>	containment check
<code>val not in s</code>	non-containment check

```
t = [2] * 7  
print(t)
```

```
[2, 2, 2, 2, 2, 2, 2]
```

```
t = 7 * [2]  
print(t)
```

# Indici negativi

---

- Le sequenze supportano anche indici negativi
- `s[-1]` si riferisce all'ultimo elemento di `s`
- `s[-2]` si riferisce al penultimo elemento di `s`
- `s[-3]` ...
  
- `s[j] = val` sostituisce il valore in posizione `j`
- **del** `s[j]` rimuove l'elemento in posizione `j`

# Confronto di sequenze

---

- Le sequenze possono essere confrontate in base all'ordine lessicografico
  - Il confronto è fatto elemento per elemento
  - Ad esempio,  $[5, 6, 9] < [5, 7]$  (True)

$s == t$	equivalent (element by element)
$s != t$	not equivalent
$s < t$	lexicographically less than
$s <= t$	lexicographically less than or equal to
$s > t$	lexicographically greater than
$s >= t$	lexicographically greater than or equal to



# Operatori per insiemi

---

- Le classi **set** e **frozenset** supportano i seguenti operatori

<code>key in s</code>	containment check
<code>key not in s</code>	non-containment check
<code>s1 == s2</code>	s1 is equivalent to s2
<code>s1 != s2</code>	s1 is not equivalent to s2
<code>s1 &lt;= s2</code>	s1 is subset of s2
<code>s1 &lt; s2</code>	s1 is proper subset of s2
<code>s1 &gt;= s2</code>	s1 is superset of s2
<code>s1 &gt; s2</code>	s1 is proper superset of s2
<code>s1   s2</code>	the union of s1 and s2
<code>s1 &amp; s2</code>	the intersection of s1 and s2
<code>s1 - s2</code>	the set of elements in s1 but not s2
<code>s1 ^ s2</code>	the set of elements in precisely one of s1 or s2

# Operatori per dizionari

---

- La classe **dict** supporta i seguenti operatori

<code>d[key]</code>	value associated with given key
<code>d[key] = value</code>	set (or reset) the value associated with given key
<code>del d[key]</code>	remove key and its associated value from dictionary
<code>key in d</code>	containment check
<code>key not in d</code>	non-containment check
<code>d1 == d2</code>	d1 is equivalent to d2
<code>d1 != d2</code>	d1 is not equivalent to d2

# Precedenza degli operatori

priorità

Operator Precedence		
	Type	Symbols
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, -expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, -
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	
11	comparisons containment	is, is not, ==, !=, <, <=, >, >= in, not in
12	logical-not	not expr
13	logical-and	and
14	logical-or	or
15	conditional	val1 if cond else val2
16	assignments	=, +=, -=, *=, etc.

# Assegnamento esteso


---

- In C o Java gli operatori binari ammettono una versione *contratta*
  - $i += 3$  è equivalente a  $i = i + 3$
- Tale caratteristica esiste anche in Python
  - Per i tipi immutable si crea un nuovo oggetto a cui si assegna un nuovo valore e l'identificatore è riassegnato al nuovo oggetto
  - Alcuni tipi di dato (e.g., list) ridefiniscono la semantica dell'operatore  $+=$

# Esempio += per **list**

---

```
alpha = [1, 2, 3]
beta = alpha
print('alpha =', alpha)
print('beta  =', beta)
beta += [4, 5]
print('beta  =', beta)
beta = beta + [6, 7]
print('beta  =', beta)
print('alpha =', alpha)
```



```
alpha = [1, 2, 3]
beta  = [1, 2, 3]
beta  = [1, 2, 3, 4, 5]
beta  = [1, 2, 3, 4, 5, 6, 7]
alpha = [1, 2, 3, 4, 5]
```

`beta += [4, 5]` estende la lista originale

Equivalente a  
`beta.extend([4,5])`

`beta = beta + [6, 7]` riassegna beta ad una nuova lista

# Chaining

---

- Assegnamento
  - In Python è permesso l'assegnamento concatenato
  - `x = y = z = 0`
- Operatori di confronto
  - In Python è permesso `1 < x + y <= 9`
  - Equivalente a `(1 < x+y) and (x + y <= 9)`,  
ma l'espressione `x+y` è calcolata una sola volta

```
x=y=5
if 3 < x+y <= 10:
    print('interno')
else:
    print('esterno')
```