Programmazione avanzata a.a. 2020-21

A. De Bonis

Introduzione a Python
(V e VI parte)

0

Namespace

- Quando si utilizza un identificativo si attiva un processo chiamato risoluzione del nome (*name resolution*) per determinare il valore associato all'identificativo
- Quando si associa un valore ad un identificativo tale associazione è fatta all'interno di uno scope
- Il **namespace** (spazio dei nomi) gestisce tutti i nomi definiti in uno scope (ambito)

1

Namespace

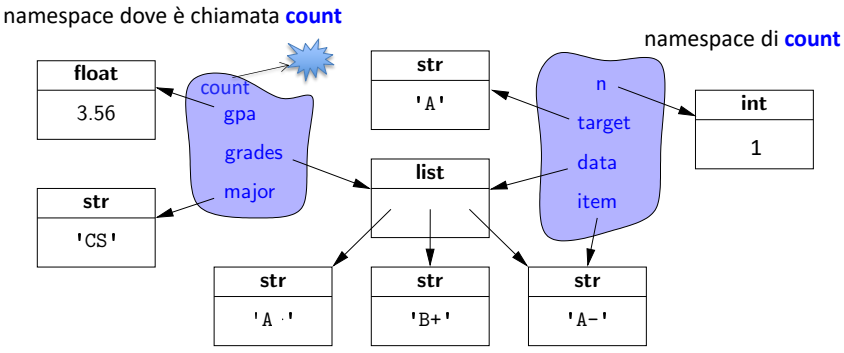
- Python implementa il namespace tramite un dizionario che mappa ogni identificativo al suo valore
- Uno scope può contenere al suo interno altri scope
- Non c'è nessuna relazione tra due identificatori che hanno lo stesso nome in due namespace differenti
- Tramite le funzioni dir() e vars() si può conoscere il contenuto del namespace dove sono invocate
  - dir elenca gli identificatori nel namespace
  - vars visualizza tutto il dizionario

2

Esempio

```
grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'
count(grades, 'A')
```

```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n
```



3

Esempio

```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n

grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'

count(grades, 'A')

print(dir())
```

```
[
'__annotations__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__',
'__spec__', 'count', 'gpa', 'grades', 'major'
]
```


4

I moduli in Python

- Un modulo è un particolare script Python
  - È uno script che può essere utilizzato in un altro script
  - Uno script incluso in un altro script è chiamato modulo
- Sono utili per decomporre un programma di grande dimensione in più file, oppure per riutilizzare codice scritto precedentemente
  - Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) attraverso il comando import
  - Il nome di un modulo è il nome del file script (esclusa l'estensione '.py')
  - All'interno di un modulo si può accedere al suo nome tramite la variabile globale __name__

5

Moduli esistenti

- Esistono vari moduli già disponibili in Python
  - Alcuni utili moduli sono i seguenti

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

6

Utilizzare i moduli

- All'interno di un modulo/script si può accedere al nome del modulo/script tramite l'identificatore __name__
- Per utilizzare un modulo deve essere incluso tramite l'istruzione **import**
  - **import** math
- Per far riferimento ad una funzione del modulo importato bisogna far riferimento tramite il nome qualificato completamente
  - math.gdc(7,21)

7

Utilizzare i moduli

- Con l'istruzione from si possono importare singole funzioni a cui possiamo far riferimento direttamente con il loro nome
  - **from** math **import** sqrt
  - **from** math **import** sqrt, floor

```
import math
print(math.gcd(7,21))

from math import sqrt
print(sqrt(3))
```

→

```
7


1.7320508075688772
```

from math **import** * tutte le funzioni di math sono importate

8

Caricamento moduli

- Ogni volta che un modulo è caricato in uno script è eseguito
- Il modulo può contenere funzioni e codice *libero*
- Le funzioni sono *interpretate,* il codice libero è eseguito
- Lo script che importa (eventualmente) altri moduli ed è eseguito per primo è chiamato dall'interprete Python __main__
- Per evitare che del codice *libero* in un modulo sia eseguito quando il modulo è importato dobbiamo inserire un controllo nel modulo sul nome del modulo stesso. Se il nome del modulo è __main__ allora il codice libero è eseguito; altrimenti il codice non viene eseguito.

9

La variabile __name__

- Ogni volta che un modulo è importato, Python crea una variabile per il modulo chiamata __name__ e salva il nome del modulo in questa variabile.
- Il nome di un modulo è il nome del suo file .py senza l'estensione .py.
- Supponiamo di importare il modulo contenuto nel file test.py. La variabile __name__ per il modulo importato test ha valore "test".
- Supponiamo che il modulo test.py contenga del codice libero. Se prima di questo codice inseriamo il controllo **if** __name__ == **'__main__'**: allora il codice libero viene eseguito se e solo se __name__ ha valore __main__ . Di conseguenza, se importiamo il modulo test allora il suddetto codice libero non è eseguito.
- Ogni volta che un file .py è eseguito Python crea una variabile per il programma chiamata __name__ e pone il suo valore uguale a "__main__" . Di conseguenza se eseguiamo test.py come se fosse un programma allora il valore della sua variabile __name__ è __main__ e il codice libero dopo l'if viene eseguito.

10

Esempio

testNoIfMain.py

```
def modifica(lista):
    lista.append('nuovo')


lst = [1, 'due']
print('lista =', lst)
modifica(lst)
print('lista =', lst)
```

esecuzione testNoIfMain.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```

Stesso comportamento se eseguiti entrambi come programmi

test.py

```
def modifica(lista):
    lista.append('nuovo')

if __name__ == '__main__':
    lst = [1, 'due']
    print('lista =', lst)
    modifica(lst)
    print('lista =', lst)
```

esecuzione test.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```


11

Esempio

importUNO.py

```
import test
lista = [3,9]
print(lista)
test.modifica(lista)
print(lista)
```

esecuzione importUNO.py

```
[3, 9]
[3, 9, 'nuovo']
```

In questo caso l'if presente in test.py evita che vengano eseguite le linee di codice libero presenti in test.py

importDUE.py

```
import testNoIfMain
lista = [3,9]
print(lista)
testNoMain.modifica(lista)
print(lista)
```

esecuzione importDUE.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
[3, 9]
[3, 9, 'nuovo']
```

In questo caso vengono eseguite anche le linee di codice libero di testNoIfMain.py perché non sono precedute dall'if

12

package

- Modo per strutturare codice Python in moduli, cartelle e sotto-cartelle
- Il package è una collezione di moduli
  - Il package è una cartella in cui, oltre ai moduli o subpackage, è presente il file __init.py__ che contiene istruzioni di inizializzazione del package (può essere anche vuoto)
  - __init.py__ serve ad indicare a Python di trattare la cartella come un package

13

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

In uno script presente nella cartella che contiene **sound**

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7)
```

```
from sound.effects import echo
echo.echofilter(input, output, delay=0.7)
```

```
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7)
```


14

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Per importare moduli in surround.py si usa un import relativo

```
from . import echo
from .. import formats
from ..filters import equalizer
```

N.B. gli import relativi si basano sul nome del modulo corrente. Siccome il nome del modulo main e` sempre "__main__", i moduli usati come moduli main devono sempre usare import assoluti.

15

Importare moduli tra package

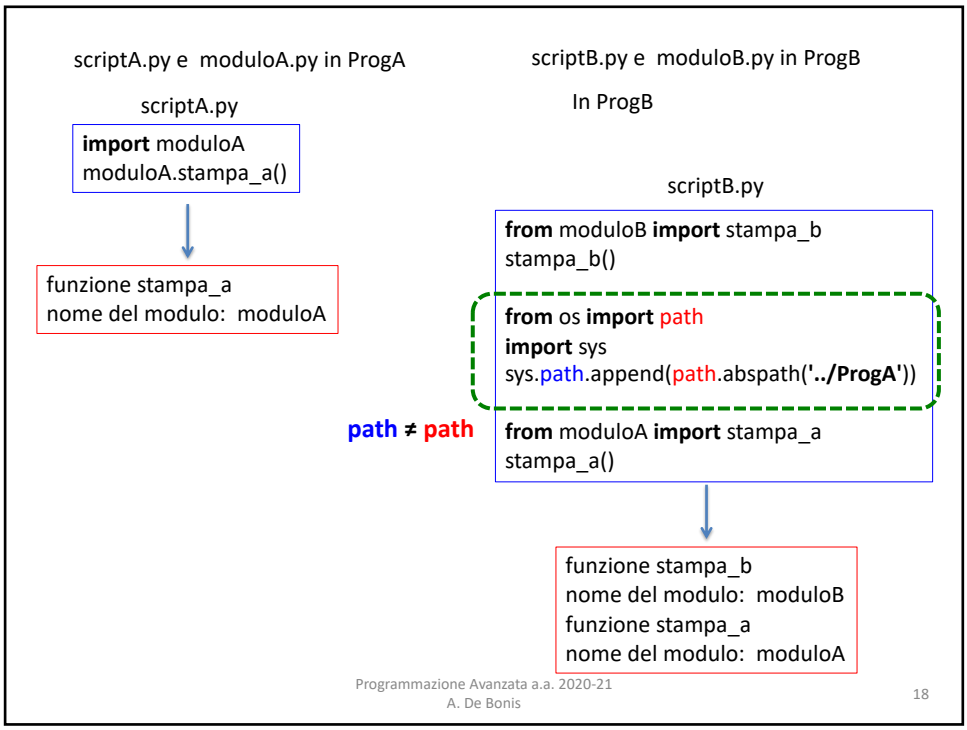
- Lo script che importa il modulo deve conoscere la posizione del modulo da importare
  - Non è necessario quando
    - il modulo è un modulo di Python
    - il modulo è stato installato
  - La variabile sys.path è una lista di stringhe che determina il percorso di ricerca dell'interprete Python per i moduli
  - Occorre aggiungere a sys.path il percorso assoluto che contiene il modulo da importare

16

Importare moduli tra package

- Quando il modulo miomodulo è importato l'interprete prima cerca un modulo built-in con quel nome. Se non lo trova, cerca un file miomodulo.py nella lista di directory date dalla variabile **sys.path**
- sys. path e` una lista di stringhe che specifica il percorso di ricerca di un modulo. sys.path è inizializzata dalle seguenti locazioni:
  - Contiene nella prima posizione la directory contenente lo script input
  - e` inizializzata da PYTHONPATH (una lista di nomi di directory con la stessa sintassi della variabile shell PATH).
  - Default dipendente dall'installazione

17



18

Programmazione avanzata

OOP

19

Python e OOP

- Python supporta tutte le caratteristiche standard della OOP
  - Derivazione multipla
  - Una classe derivata può sovrascrivere qualsiasi metodo della classe base
- Tutti i membri di una classe (dati e metodi) **sono pubblici**

20

Ereditarietà

- Le superclassi di una classe vengono elencate tra le parentesi nell'intestazione della classe
- Le superclassi potrebbero trovarsi in altri moduli
  - Esempio: supponiamo che FirstClass sia nel modulo modulename

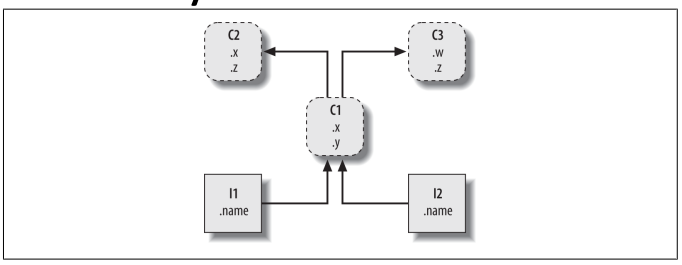
```
from modulename import FirstClass
class SecondClass(FirstClass):
    def display(self): ...
```

oppure

```
import modulename
class SecondClass(modulename.FirstClass):
    def display(self): ...
```


21

Python e OOP



- I1.w viene risolto in C3.w
- Python cerca l'attributo nell'oggetto e poi risale man mano nelle classi sopra di esso dal basso verso l'alto e da sinistra verso destra
  - I2.z viene risolto in C2.z

22

Classi in Python

- In Python in una classe possiamo avere
  - variabili istanza (dette anche membri dati)
  - variabili di classe
    - condivise tra tutte le istanze della classe
  - metodi (detti anche membri funzione)
    - metodi specifici della classe
    - overloading di operatori
- Per far riferimento ad una variabile di istanza si fa precedere l'identificatore dalla parola chiave **self**
  - se non esiste una variabile di istanza con lo stesso nome, **self** puo` essere usato anche per far riferimento ad una variabile di classe

23

Attributi di classe e attributi di istanza

- Le variabili di classe sono di solito aggiunte alla classe mediante assegnamenti all'esterno delle funzioni
- Le variabili di istanza sono aggiunte all'istanza mediante assegnamenti effettuati all'interno di funzioni che hanno self tra gli argomenti.

24

Attributi di classe e attributi di istanza

```
class myClass:
      a=3
      def method(self):
            self.a=4


x=myClass()
print(x.a)
x.method()
print(x.a)
y=myClass()
print(y.a)
print(myClass.a)
```

```
3
4
3
3
```


25

Costruttori in Python

- Nelle classi Python ci può essere un solo costruttore chiamato **__init__**
- Per simulare differenti costruttori si possono usare
  - parametri inizializzati di default
  - numero di parametri variabile
  - parametri keyword
- Se __init__ non è fornito né dalla classe né da nessuna delle classi più in alto nella gerarchia delle classi allora vengono create istanze vuote

26

```
class MyClass:
    common = []

    def __init__(self, *args):
        self.L = []
        for val in args:
            self.L.append(val)
            self.common.append(val)
        #oppure
        #MyClass.common.append(val)

    def __str__(self):
        return str(self.L)

    def out(self):
        for val in self.common:
            print(val, end=' ')
        print()
```

Variabile di classe

```
var_a = MyClass()
var_b = MyClass(3, 4)
var_c = MyClass(5, 6)
print(var_a)
print(var_b)
print(var_c)
var_a.out()
var_b.out()
var_c.out()
```

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


27

Metodi di una classe

- Tutti i metodi di istanza della classe hanno come primo parametro **self** che rappresenta l'istanza dell'oggetto su cui è chiamato il metodo
  - self è un riferimento esplicito all'oggetto su cui andare ad operare
    - Simile a this in Java

a istanza di una classe A
func metodo della classe A
a.func(b) è convertito in A.func(a,b)

A è considerato un namespace

Assegnamenti dinamici

- Data un'istanza della classe è possibile aggiungere e/o rimuovere dinamicamente membri all'istanza stessa
- Possiamo aggiungere anche variabili di classe

```
def add_var():
    var_a.nuovo = 3
    print('nuovo attributo: ', var_a.nuovo)
    try:
        print('nuovo attributo: ', var_b.nuovo)
    except Exception as e: print(e)

    MyClass.nuovo = 0
    try:
        print('nuovo attributo: ', var_b.nuovo)
    except Exception as e: print(e)
```

```
nuovo attributo:  3

'MyClass' object has no attribute 'nuovo'

nuovo attributo:  0
```

Per cancellare un attributo si usa **del**
del var_a.nuovo

Ulteriori esempi

la classe MyClass ha il metodo out

```
x = MyClass()
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

16

```
MyClass.common = []
x = MyClass([1,'x','das'])
xout = x.out
xout()
```

[1, 'x', 'das']

30

Altro sui metodi

- I metodi di istanza di una classe possono chiamare altri metodi di istanza della stessa classe utilizzando **self**

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

- I metodi di una classe possono essere definiti fuori la classe stessa

```
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'Ciao Mondo!'

c = C()
print(c.f(2,3))
```


31

Overloading di operatori

- In Python è possibile fornire, per la classe che si sta definendo, una propria definizione degli operatori
  - overloading degli operatori
- È sufficiente definire i metodi corrispondenti agli operatori
  - __add__ corrisponde a +
  - __lt__ corrisponde a <
  - . . .

32

Overloading di operatori

- In una classe Python implementiamo l'overloading degli operatori fornendo i metodi con nomi speciali (__X__) corrispondenti all'operatore.
- Tali metodi vengono richiamati automaticamente se un'istanza della classe appare in operazioni built-in
  - Ad esempio, se la classe di un oggetto ha un metodo __add__ quel metodo __add__ è invocato ogni volta che l'oggetto appare in un'espressione con +
- Le classi possono effettuare l'overriding della maggior parte degli operatori built-in
- Non ci sono default per questi metodi. Se una classe non definisce questi metodi allora l'operazione corrispondente non è supportata
  - nel caso venga usata un operazione non supportata viene lanciata un'eccezione.

33

Overloading di operatori

- L'overloading degli operatori permette di usare le istanze delle nostre classi come se fossero di tipi built-in.
- Ciò permette ad altri programmatori Python di interfacciarsi in modo più naturale con il nostro codice
- Quando ciò non è necessario (ad esempio nello sviluppo di applicazioni) è preferibile non ricorrere all'overloading e utilizzare nomi più appropriati e consoni all'uso che si fa di quegli operatori nell'ambito dell'applicazione.

34

Operatori

- Un operatore può essere applicato a due istanze di tipi diversi, come nel caso: a + b
  - a istanza di una classe A
  - b istanza di una classe B
- Se A non implementa __add__ Python controlla se B implementa __radd__ e lo esegue
  - Permette di definire una semantica differente a seconda se l'operando sia un operando a sinistra o a destra dell'operatore

```
a = int(3)
b = int(2)
print(a.__pow__(b))
print(a.__rpow__(b))
```

→

```
9
8
```


35

Operatori

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	

36

Operatori

abs(a)	a.__abs__()
a < b	a.__lt__(b)
a <= b	a.__le__(b)
a > b	a.__gt__(b)
a >= b	a.__ge__(b)
a == b	a.__eq__(b)
a != b	a.__ne__(b)
v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k,v)
del a[k]	a.__delitem__(k)
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)

Possiamo definire l'operatore di chiamata a funzione per una classe

```
def __call__(self, *args, **kwargs):
    print(args)
```

```
n = MyClass()
n(3,4,5,6)
```

→ (3, 4, 5, 6)

37

Operatori __i*__

- Implementano gli operatori +=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=
- Possiamo implementarli come vogliamo, ma per preservare la semantica dell'operatore dovrebbero
  - Modificare self
  - Restituire il risultato dell'operazione (self o risultato equivalente)
- Il risultato restituito è assegnato all'identificativo a sinistra dell'operando

```
a += b      è equivalente a
a.__iadd__(b) e a
a=a.__iadd__(b)
```


Overload di non-operatori

- Nella definizione della classe si può specificare l'overloading di alcune funzioni built-in di Python
  - Specificare come queste funzioni devono operare quando ricevono in input un'istanza della classe
- Funzioni built-in
  - len
  - str
  - bool

```
foo = F()
if foo:  è trasformato in
if foo.__bool__() che è trasformato in
F.__bool(foo)__
```

Non-Operatori

len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

Python deriva alcuni automaticamente la definizione di alcuni metodi dalla definizione di altri

40

__call__

- Se all'interno di una classe è definito il medodo __call__ allora le istanze della classe diventano callable
- __call__ viene invocato ogni volta che usiamo il nome di un'istanza della classe come se fosse il nome di una funzione

41

__call__

```
class C:
      def __call__(self, *pargs, **kargs):
            print('Chiamata:', pargs, kargs)

x=C()
x(1, 2, 3)
x(1, 2, 3, x=4, y=5)
```

```
Chiamata: (1, 2, 3) {}
Chiamata: (1, 2, 3) {'y': 5, 'x': 4}
```


42

__call__

```
class Prod:
      def __init__(self, value):
            self.value = value
      def __call__(self, other):
            return self.value * other

x = Prod(2)
print(x(3))
```

```
6
```

Posso usare l'istanza x di Prod come se fosse una funzione ma allo stesso tempo posso utilizzare lo stato interno di x per definire quello che fa la funzione

43

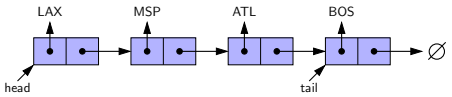
__bool__

- Ogni oggetto è vero o falso in Python
- Quando si codifica una classe si possono definire metodi che restituiscono True o False per le istanze della classe
- Non è necessario implementare __bool__
  - se __bool__ non è implementato nella classe (o in una superclasse) allora Python usa il metodo __len__ per dedurre il valore Booleano dell'oggetto (si dice che il metodo __bool__ è implicato)
  - Se nessuno dei due metodi è implementato, l'oggetto è considerato vero.

__iter__

- Il metodo __iter__ restituisce un iteratore per un oggetto contenitore
  - Gli oggetti iteratori hanno anch'essi bisogno del metodo __iter__ per poter restituire se stessi
- Il for invoca automaticamente __iter__ e crea una variabile temporanea senza nome per immagazzinare l'iteratore durante il loop.
- se in una classe __len__ e __getitem__ sono implementati, Python fornisce automaticamente __iter__ per quella classe
- Se presente __iter__, allora è fornito anche il metodo __contains__ automaticamente

Esempio di Lista Lincata




```
class LinkedList:
   class Node:
      def __init__(self, element, next):
         self._element = element
         self._next = next

   def __init__(self):
      self._head = None
      self._tail = None
      self._size = 0
```

```
def add_head(self, element):
   newNode = self.Node(element, self._head)
   if self._size == 0:
      self._tail = newNode
   self._head = newNode
   self._size += 1

def add_tail(self, element):
   newNode = self.Node(element, None)
   if self._size == 0:
      self._head = newNode
   else:
      self._tail._next = newNode
   self._tail = newNode
   self._size += 1
```


46

```
def __len__(self):
      return self._size

def __getitem__(self, j):
   cnt = 0
   #Consideriamo anche indici
   #negativi alla Python
   if j<0: j = self._size+j
   if  j < 0 or j >= self._size:
     raise IndexError()
   current = self._head
   while current != None:
      if cnt == j:
         return current._element
      else:
         current = current._next
         cnt += 1
```

```
def __str__(self):
   toReturn = '<'
   current = self._head
   while current != None:
      toReturn += str(current._element)
      current = current._next
      if current != None:
         toReturn += ', '
   toReturn += '>'
   return toReturn
```

Automaticamente implementati da Python

__bool__
__iter__
__contains__

47

```
from LinkedList import LinkedList
lst = [1, 3, 5, 6]
lista = LinkedList()
for val in lst:
    lista.add_head(val)

print(lista)
if lista:
    print('lista piena')
else:
    print('lista vuota')

print(lista[1])
print(lista[-1])

if 5 in lista:
    print('presente')
else:
    print('assente')

for val in lista:
    print(val, end=' ')
```

```
<6, 5, 3, 1>

lista piena


5
1


presente


6 5 3 1
```


48

Esercizio

3. Scrivere la classe MyDictionary che implementa gli operatori di dict riportati di seguito. MyDictionary deve avere **solo** una variabile di istanza e questa deve essere di tipo lista. Per rappresentare le coppie, dovete usare la classe MyPair che ha due variabili di istanza (key e value) e i metodi getKey, getValue, setKey, setValue .

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

49