

Programmazione avanzata a.a. 2024-25

A. De Bonis

## Introduzione a Python (V e VI parte)

26

## 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

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

27

27

```

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

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

28

## 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

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

29

29

## 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

Programmazione Avanzata a.a. 2024-25  
 A. De Bonis

30

30

## Ulteriori esempi

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

16

la classe MyClass ha il  
 metodo **out**

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

[1, 'x', 'das']

Programmazione Avanzata a.a. 2024-25  
 A. De Bonis

31

31

## Altro sui metodi

- I metodi di istanza di una classe possono chiamare altri metodi di istanza della stessa classe utilizzando **self**
- I metodi di una classe possono essere definiti fuori la classe stessa

```
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)
```

```
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))
```

32

## 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 `<`
  - ...

33

## 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.

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

34

34

## 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.

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

35

35

## 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

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

36

36

## Operatori

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

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

37

37

## Operatori

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

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)`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

38

38

## 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)
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

39

39

## 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)
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

40

40

## 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

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

41

41

## \_\_call\_\_

- Se all'interno di una classe è definito il metodo `__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

42

## \_\_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}
```

43

## \_\_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

44

## \_\_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 implicito)
  - Se nessuno dei due metodi è implementato, l'oggetto è considerato vero.

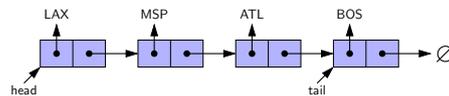
45

## \_\_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

46

## 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
  
```

47

```

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\_\_

48

```

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

49

## 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

50

## Ereditarietà

- Supportata da Python come segue

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

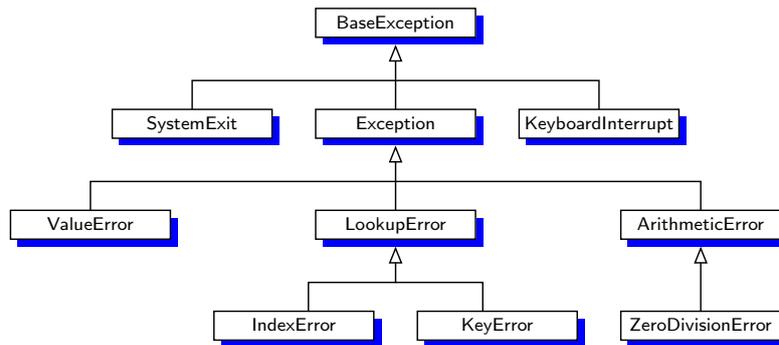
- BaseClassName** deve essere definita nello scope che contiene la definizione della classe derivata **DerivedClassName**
- Si possono usare classi base definite in altri moduli

```
class DerivedClassName(modname.BaseClassName):
```

51

## Gerarchia delle eccezioni in Python

Piccola porzione della gerarchia



Programmazione Avanzata a.a. 2024-25  
A. De Bonis

52

52

## Ereditarietà

- Le classi derivate possono
  - aggiungere variabili di istanza
  - sovrascrivere i metodi della classe base
  - accedere ai metodi e variabili della classe base
- Python supporta l'ereditarietà multipla

```

class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    <statement-N>
  
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

53

53

## Esempio

```
class Base:
    def __init__(self, a, b):
        self._a = a
        self._b = b

    def stampa(self):
        print('<{0}>,<{1}>'.format(self._a, self._b))
```

```
class Derivata(Base):
    def __init__(self, a, b, c):
        super().__init__(a, b)
        self._c = c

    def stampa(self):
        print('{{{0}}}-{{{1}}}'.format(self._a, self._b))
```

```
b = Base(1, 3)
b.stampa()
d = Derivata('a', 'b', 9)
d.stampa()
```

<1>,<3>  
[a]-[b]

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

54

54

## Utilizzo metodi classe base

- Per invocare metodi di istanza definiti nella classe base si usa la funzione `super()`
  - `super().nome_metodo(argomenti)`
- Oppure si usa `BaseClassName.nome_metodo(self, argomenti)`
  - Funziona quando `BaseClassName` è accessibile come `BaseClassName` nello scope globale

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

55

55

## super()

- Serve per accedere ai metodi della classe base che sono stati sovrascritti con la derivazione
  - `super()` restituisce un riferimento ad un oggetto
- Un altro modo per far riferimento, tramite `super`, a metodi sovrascritti è
  - `super(DerivedClassName, self).nome_metodo(parametri)`

56

## super()

```
class base:
    def f(self):
        print("base")

class der(base):
    def f(self):
        print("der")

    def g(self):
        self.f()
        super().f()
        super(der,self).f()
        base.f(self)

class derder(der):
    def f(self):
        print("derder")
    def h(self):
        self.f()
        super().f()
        super(derder,self).f()
        super(der,self).f()
```

```
x=der()
x.g()
y=derder()
y.h()
```

```
der
base
base
base
derder
der
der
base
```

57

## super()

```

class base:
    def __init__(self,v):
        self.a=v
    def f(self):
        print("base --", "a =",self.a)

class der(base):
    def f(self):
        print("der -- ", "a =",self.a)

class derder(der):
    def f(self):
        print("derder -- ", "a =",self.a)

x=der(10)
super(der,x).f()
print()
y=derder(20)
super(derder,y).f()
super(der,y).f()

```

```

base -- a = 10

der -- a = 20
base -- a = 20

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

58

## Ereditarietà Multipla

- Nella classe derivata la ricerca degli attributi ereditati da una classe genitore avviene
  - dal basso verso l'alto e da sinistra verso destra

**class DerivedClassName**(Base1, Base2, Base3):

- Se un attributo non è trovato in **DerivedClassName** lo si cerca in Base1, dopo (ricorsivamente) nelle classi base di Base1 e, se non viene trovato si procede con Base2 e così via

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

59

59

## Ereditarietà Multipla

- L'attributo `__mro__` di una classe contiene l'elenco delle classi in cui si cerca il metodo che è stato invocato su un'istanza della classe
  - Le classi sono esaminate secondo l'ordine indicato in `__mro__` (mro: Method Resolution Order)
    - L'attributo dipende da come è stata definita la classe
    - Il metodo `mro()` è invocato quando si crea un'istanza della classe. Questo metodo può essere sovrascritto (in una metaclass) per modificare l'ordine in cui vengono cercati i metodi nelle classi che formano la gerarchia. L'ordine stabilito da `mro()` è memorizzato in `__mro__`.
    - L'attributo è a sola lettura

```
class D(A,B,C):
```

`D.__mro__`

```
(<class '__main__.D'>, <class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>)
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

60

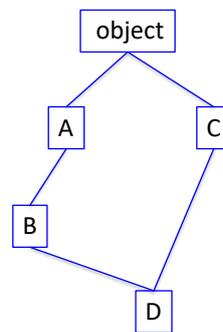
60

```
class A():
    pass
```

```
class B(A):
    pass
```

```
class C():
    pass
```

```
class D(B,C):
    pass
```



`D.__mro__`

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>)
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

61

61

## Attributo `__bases__`

- Contiene la tupla delle classi base di una classe
  - Accessibile in lettura/scrittura
  - Modificando `__bases__` l'attributo `__mro__` è *ricomputato*
- Per modificare `__bases__` si usa la funzione `setattr`
  - `setattr(Derivata, '__bases__', (Base2, Base1))`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

62

62

## Funzioni built-in

- `isinstance`(ist, classe) serve per verificare il tipo di un'istanza di una classe

```
f = 3.4
print(isinstance(f, float)) → True
```

- `issubclass`(x,y) serve per verificare se x è una sottoclasse di y

```
class A(): pass
class B(A): pass
class C(): pass
class D(B,C): pass
```

```
issubclass(A,C) → False
```

```
issubclass(D,C) → True
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

63

63

## Ordine differente rispetto a **mro**

- Assumiamo che le classi A, B e C definiscano il metodo **metodo\_base** e che D sia derivata da A, B e C ( `class D(A,B,C):` pass) e sia d un'istanza di D
- Se si esegue `d.metodo_base(parametri)`, allora è eseguito **metodo\_base** definito nella classe A
- Per invocare **metodo\_base** definito in un'altra classe base si deve far riferimento direttamente alla classe base specifica

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

64

64

```

class A():
    def __init__(self, a, val):
        self._a = a
        self._val = val

    def stampa(self):
        print('a =', self._a, 'val =',self._val)

class B():
    def __init__(self, b, val):
        self._b = b
        self._val = val

    def stampa(self):
        print('b =', self._b, 'val =',self._val)

class C():
    def __init__(self, c, val):
        self._c = c
        self._val = val

    def stampa(self):
        print('c =', self._c, 'val =',self._val)

class D(A,B,C):
    def __init__(self, a, b, c, val):
        A.__init__(self, a, val)
        B.__init__(self, b, 2*val)
        C.__init__(self, c, 3*val)

    def stampa(self):
        C.stampa(self)
        B.stampa(self)
        A.stampa(self)

```

`d = D(1,2,3, 123)`  
`d.stampa()`

↓

```

c = 3 val = 369
b = 2 val = 369
a = 1 val = 369

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

65

65

## Iteratori

- Se una classe supporta l'iteratore possiamo ottenere un riferimento ad esso tramite la funzione `iter()`
  - Si invoca `iter` su un'istanza della classe
- Per ottenere il prossimo elemento nella classe invochiamo `next()` sull'iteratore ottenuto
- Viene lanciata un'eccezione quando non ci sono più elementi nell'istanza della classe

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

66

66

```
lista = [85,23,59]
print(lista)
it = iter(lista)
print(it)
print(next(it))
print(next(it))
print(next(it))
```



```
[89, 23, 59]
<list_iterator object at 0x10217aa58>
89
23
59
```

```
lista = [59, 42, 90]
print(lista)
it = iter(lista)
print(it)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```



```
[59, 42, 90]
<list_iterator object at 0x10ad62908>
59
42
90
Traceback (most recent call last):
  File "/Users/adb/Documents/r.py", line 8, in
    <module>
      print(next(it))
StopIteration
```

← **eccezione**

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

67

67

## Gestire l'eccezione

```

lista=[59, 42, 90]
print(lista)
it = iter(lista)
print(it)

while True:
    try:
        print(next(it))
    except Exception as e:
        break

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

68

68

## Generatori

- Una funzione generatore è un modo semplice ed immediato per creare un iteratore (detto generatore)
  - I metodi del generatore `__iter__()` e `__next__()` sono creati automaticamente
  - `__iter__()` : restituisce l'iteratore stesso
  - `__next__()` : viene utilizzato nei cicli per ottenere il prossimo elemento
- La sintassi per definire una funzione generatore è simile a quella usata per definire una funzione, ma al posto di `return` si usa `yield`
- Quando si incontra un `yield` l'esecuzione del generatore è sospesa, viene restituito il valore indicato da `yield`
- Ogni volta che si chiama `next()`, il generatore riparte da dove l'esecuzione era stata sospesa
  - Si parte dall'istruzione successiva a `yield`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

69

69

## Generatori

```
def gen_range(n):
    k=0
    while k<n:
        yield k
        k += 1
```

```
ite=gen_range(10)
while(True):
    try:
        i=next(ite)
        print(i, end=' ')
    except Exception as e:
        break
```

```
for i in gen_range(10):
    print(i, end=' ')
```

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

Si tratta di una sorta di funzione che genera una sequenza di valori restituiti uno per volta tramite **yield**

Nel generatore non possono coesistere **yield** e **return**

Programmazione Avanzata a.a. 2024-25  
A. De Bonis 70

70

## Generatori

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

for char in reverse('programmazione'):
    print(char, end="")
```

enoizammargorp

Programmazione Avanzata a.a. 2024-25  
A. De Bonis 71

71

## Superclassi astratte

- Una superclasse astratta è una classe il cui comportamento è in parte specificato dalle sottoclassi
- Se un metodo che deve essere definito dalle sottoclassi non è definito in una sottoclasse allora Python lancia un'eccezione quando effettua la ricerca del metodo nella gerarchia delle classi

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

72

72

## Superclassi astratte

- A volte i programmatori, per rendere più evidente ciò che deve essere specificato nelle sottoclassi, usano degli **statement assert** o degli statement raise che lanciano l'eccezione `NotImplementedError`

Uso di assert nella funzione che deve essere fornita dalle sottoclassi

```
>>> class Super:
...     def delegate(self):
...         self.action()
...     def action(self):
...         assert False, 'action must be defined!'
...
>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

73

73

## Assert statement

- Rappresentano un modo per inserire asserzioni per il debugging in un programma
  - `assert expression`  
è equivalente a
    - `if __debug__:`  
    `if not expression: raise AssertionError`
  - `assert expression1, expression2`  
è equivalente a
    - `if __debug__:`  
    `if not expression1: raise AssertionError(expression2)`
- Negli if in alto `AssertionError` è l'eccezione built-in lanciata quando uno statement `assert` fallisce e `__debug__` è una variabile built-in
  - `__debug__` è normalmente `True` ed è `False` quando si usa l'opzione `-O` per richiedere l'ottimizzazione in fase di compilazione. Il generatore di codice non genera alcun codice per lo statement `assert` quando è specificata l'opzione `-O`.
- Non è possibile assegnare valori a `__debug__`. Il suo valore è determinato all'inizio dell'interpretazione del codice.
- Maggiori dettagli su `assert` e `raise` in seguito

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

74

74

## Assert statement

```

class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'
X=Super()
X.delegate()

```

mod.py

```

$ python3 -O mod.py
$ python3 mod.py
Traceback (most recent call last):
  File "mod.py", line 8, in <module>
    X.delegate()
  File "mod.py", line 3, in delegate
    self.action()
  File "mod.py", line 5, in action
    assert False, 'action must be defined!'
AssertionError: action must be defined!

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

75

75

## Superclassi astratte

- A volte i programmatori, per rendere più evidente ciò che deve essere specificato nelle sottoclassi, usano degli **statement assert** o **degli statement raise** che lanciano l'eccezione `NotImplementedError`

Uso di `raise` nella funzione che deve essere fornita dalle sottoclassi

```
>>>class Super:
...     def delegate(self):
...         self.action()
...     def action(self):
...         raise NotImplementedError('action must be defined!')
>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

76

76

## Superclassi astratte

- L'eccezione sarà lanciata anche se il metodo `delegate()` è invocato su istanze di una sottoclasse di `Super` a meno che la sottoclasse non fornisca il metodo `action()` che rimpiazza quello della superclasse

```
>>> class Sub(Super): pass
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!
>>> class Sub(Super):
...     def action(self): print('spam')
>>> X = Sub()
>>> X.delegate()
spam
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

77

77

## Abstract Base Class (ABC)

- Rappresenta un ulteriore strumento per definire superclassi astratte
- Python, tramite il modulo `abc`, fornisce il supporto per definire formalmente una classe di base astratta

78

<https://github.com/python/cpython/blob/3.6/Lib/abc.py>

```
from abc import ABCMeta, abstractmethod # need these definitions

class Sequence(metaclass=ABCMeta):
    """Our own version of collections.Sequence abstract base class."""

    @abstractmethod
    def __len__(self):
        """Return the length of the sequence."""

    @abstractmethod
    def __getitem__(self, j):
        """Return the element at index j of the sequence."""
```

Una metaclassa fornisce un modello per la definizione della classe stessa, `ABCMeta` assicura che il costruttore della classe lanci un'eccezione quando si tenta di istanziare la classe astratta

`@abstractmethod` è un decoratore, indica che non si fornisce un'implementazione del metodo (il metodo è astratto) e le classi derivate devono implementarlo (Python impone ciò impedendo l'istanziamento di sottoclassi che non implementano i metodi dichiarati astratti)

79

### metodi comuni a tutte le sequenze

```
def __contains__(self, val):
    """Return True if val found in the sequence; False otherwise."""
    for j in range(len(self)):
        if self[j] == val:           # found match
            return True
    return False

def index(self, val):
    """Return leftmost index at which val is found (or raise ValueError)."""
    for j in range(len(self)):
        if self[j] == val:         # leftmost match
            return j
    raise ValueError('value not in sequence') # never found a match

def count(self, val):
    """Return the number of elements equal to given value."""
    k = 0
    for j in range(len(self)):
        if self[j] == val:         # found a match
            k += 1
    return k
```

i metodi `__contains__`, `index` e `count` non si basano su nessuna assunzione di come è realizzata l'istanza `self`

80

## Abstract Base Class (ABC)

- Sebbene quest'ultima tecnica per creare superclassi astratte richieda più codice e la conoscenza di strumenti più avanzati, un vantaggio di questo approccio è che gli errori che scaturiscono dall'assenza di metodi si verificano quando tentiamo di creare un'istanza della classe e non più tardi quando tentiamo di invocare il metodo mancante.

81