Programmazione avanzata a.a. 2019-20

A. De Bonis

Introduzione a Python

IV parte

Programmazione avanzata

OOP

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**

Ereditarietà

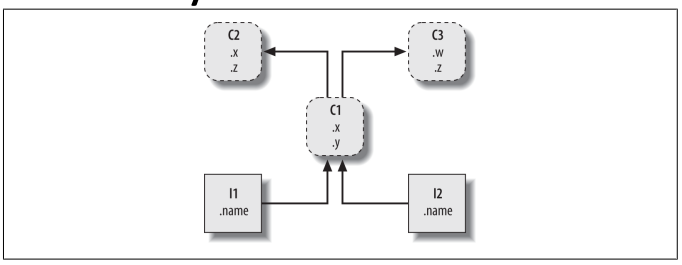
- Le superclassi di una classe vengono elencate tra le parentesi nell'intestazione della classe
- Le superclassi potrebbero trovarsi in altri moduli
  - Esempio: supponiamo che FirstClass sia nel modulo modulename

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

oppure

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

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

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 o di classe si fa precedere l'identificatore dalla parola chiave **self**

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.

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
```

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 a né dalla classe né da nessuna delle classi più in alto nella gerarchia delle classi allora vengono create istanze vuote


```
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
```

Metodi di una classe

- Tutti i metodi della classe hanno come primo parametro **self** che rappresenta l'istanza dell'oggetto su cui è chiamato il metodo
  - In ogni metodo c'è un riferimento esplicito all'oggetto su cui andare ad operare
    - Simile a this in Java
  - Anche le variabili istanza e quelle di classe sono precedute da **self**

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

```
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']

Altro sui metodi

- I metodi di una classe possono chiamare altri metodi 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))
```

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 <
  - . . .

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.

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.

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
```

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__()	

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)

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

__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

__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}
```


__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

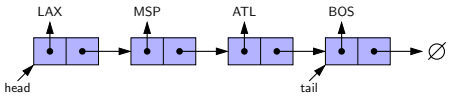
__bool__

- 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
```



```
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__

```
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
```


Ereditarietà

- Supportata da Python come segue

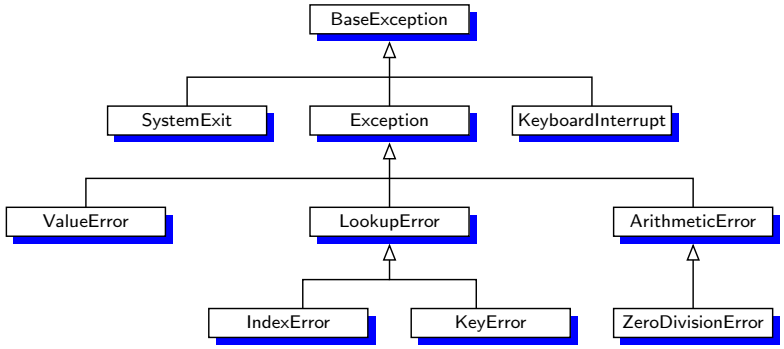
```
class DerivedClassName(BaseClassName):
    <statement-1>
    . . .
    <statement-N>
```

- 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):
```

Gerarchia delle eccezioni in Python

Piccola porzione della gerarchia



Ereditarietà

- Le classi derivate possono
  - aggiungere variabili 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>
```

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))
```

Base.__init__(self,a,b)

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

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

Utilizzo metodi classe base

- Per invocare metodi definiti nella classe base si usa la funzione super()
  - super().nomemetodo(argomenti)
- Oppure si usa

  BaseClassName.nome_metodo(self, argomenti)
  - Funziona quando BaseClassName è accessibile come BaseClassName nello scope globale

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)

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
```

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
```


Ereditarietà Multipla

- Nella classe derivata la ricerca degli attributi ereditati da una classe genitore avviene in modalità
  - 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 è trovato si procede con Base2 e così via

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 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'>)
```

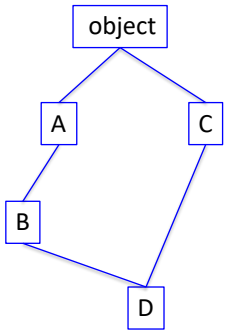


```
class A():
  pass

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'>)
```

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))

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**

Ordine differente rispetto a mro

- Assumiamo che le classi A, B e C definiscono 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


```
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
```

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


```
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**

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
```


Generatori

- Un generatore è un modo semplice ed immediato per creare un iteratore
  - I metodi **__iter__()** e **__next__()** sono creati automaticamente
- La sintassi per definire un 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**

Generatori

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

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

0 1 2 3 4 5 6 7 8 9

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

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**

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

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

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!
```

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 possible assegnare valori a __debug__. Il suo valore è determinato all'inizio dell'interpretazione del codice.
- Maggiori dettagli su assert e raise in seguito

Assert statement

mod.py

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


X=Super()
X.delegate()
```

```
$ 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!
```

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!
```


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
```

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

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 metaclasse 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'istanziazione di sottoclassi che non implementano i metodi dichiarati astratti)

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

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 appproccio 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.