Programmazione avanzata a.a.

A. De Bonis

Introduzione a Python

(IV lezione)

0

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

1

```
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
              ...
```

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)
```



```
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
```

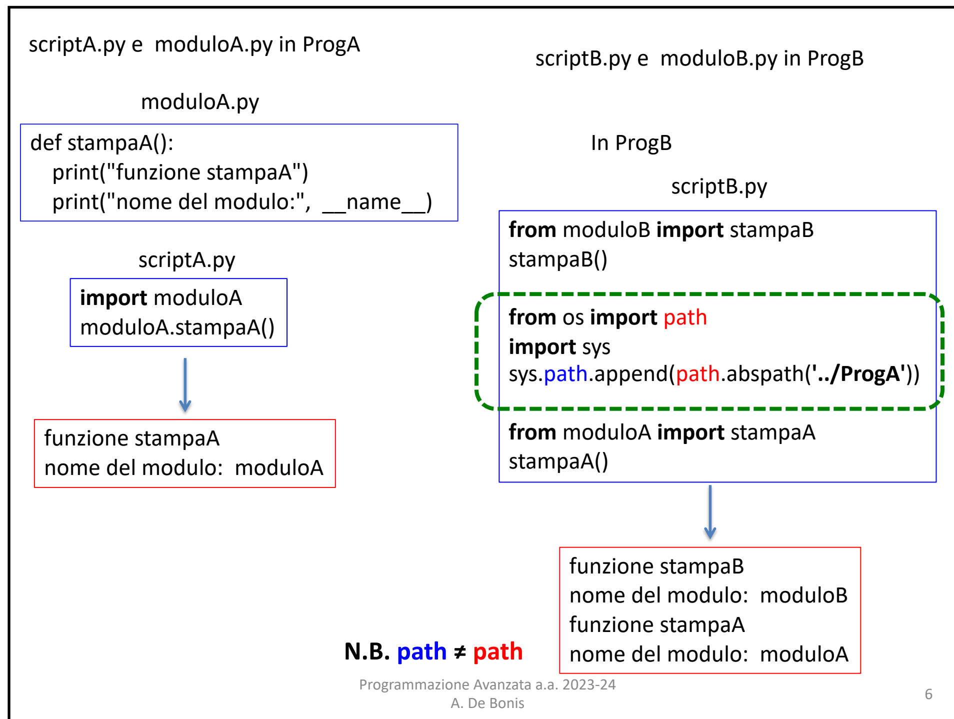
N.B. gli import relativi si basano sul nome del modulo corrente. Siccome il nome del modulo main e` sempre "__main__", i moduli usati come moduli main devono sempre usare import assoluti.

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

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 e contiene nella prima posizione la directory contenente lo script input
- sys.path è inizializzata dalle seguenti locazioni:
  - e` inizializzata da PYTHONPATH (una lista di nomi di directory con la stessa sintassi della variabile shell PATH).
  - Default dipendente dall'installazione



6

os.path

- Il modulo os.path implementa alcune utili funzioni sui pathname
- os.path.**abspath**(*path*) restituisce un versione ”assolutizzata” del pathname *path*.

7

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 di 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

Attributi di classe e attributi di istanza

- Le variabili di classe sono di solito (ma non solo) aggiunte alla classe mediante assegnamenti all'esterno delle funzioni.
- Le variabili di istanza possono essere 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)

x.b=10
print(x.b)
```

```
3
4
3
3
10
```

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


```
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 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 (slide 15)

```
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 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))
```

Programmazione avanzata a.a. 2022-23
A. De Bonis

Introduzione a Python

VI lezione

20

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 <
  - . . .

21

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'overloading 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 dei 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 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

__call__

```
class C:
    def __call__(self, *args, **kwargs):
        print('Chiamata:', args, kwargs)

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

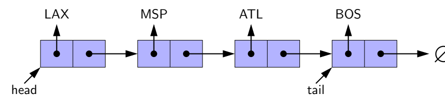
__bool__

- 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
```

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