

Programmazione avanzata a.a. 2023-24  
A. De Bonis

## Introduzione a Python

### V lezione

51

## Ereditarietà

- Supportata da Python come segue

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

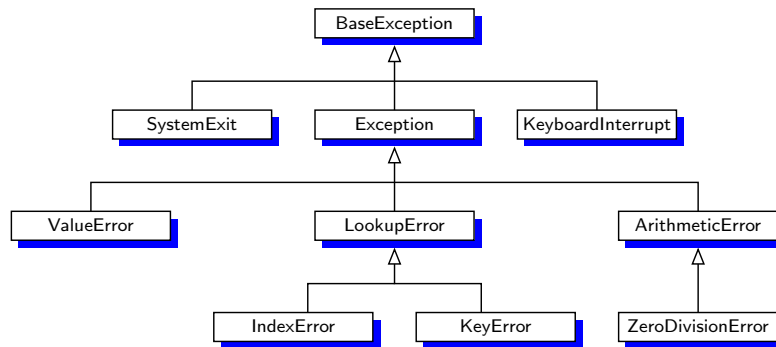
- `BaseClassName` deve essere definita nello scope che contiene la definizione della classe derivata `DerivedClassName`
- Si possono usare classi base definite in altri moduli

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

52

## Gerarchia delle eccezioni in Python

Piccola porzione della gerarchia



Programmazione Avanzata a.a. 2023-24  
A. De Bonis

53

53

## 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. 2023-24  
A. De Bonis

54

54

## 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. 2023-24  
A. De Bonis

55

55

## Utilizzo metodi classe base

- Per invocare metodi di istanza 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

Programmazione Avanzata a.a. 2023-24  
A. De Bonis

56

56

## 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)`

57

## 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
```

58

## 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. 2023-24  
A. De Bonis

59

## 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. 2023-24  
A. De Bonis

60

## 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. 2023-24  
A. De Bonis

61

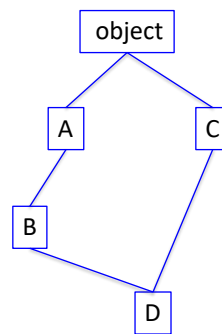
61

```
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. 2023-24  
A. De Bonis

62

62

```

class A: pass
class C: pass
class F: pass
class B(A): pass
class G(F): pass
class H(F): pass
class D(B,C,G,H): pass
print(D.__mro__)

```

```

graph TD
    object --> A
    object --> C
    object --> F
    A --> B
    F --> G
    F --> H
    B --> D
    C --> D
    G --> D
    H --> D

```

**D.\_\_mro\_\_**  
(<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.G'>, <class '\_\_main\_\_.H'>, <class '\_\_main\_\_.F'>, <class 'object'>)

Programmazione Avanzata a.a. 2023-24  
A. De Bonis

63

```

class A: pass
class C: pass
class F: pass
class B(A): pass
class L(A): pass
class G(F): pass
class H(F): pass
class D(B,L,C,G,H): pass
print(D.__mro__)

```

```

graph TD
    object --> A
    object --> C
    object --> F
    A --> B
    A --> L
    F --> G
    F --> H
    B --> D
    L --> D
    C --> D
    G --> D
    H --> D

```

**D.\_\_mro\_\_**  
(<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.L'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.G'>, <class '\_\_main\_\_.H'>, <class '\_\_main\_\_.F'>, <class 'object'>)

Programmazione Avanzata a.a. 2023-24  
A. De Bonis

64

```

class A: pass
class C: pass
class F: pass
class B(A): pass
class L(A): pass
class G(F): pass
class H(F): pass
class D(B,C,G,H,L): pass
print(D.__mro__)

```

```

graph TD
    object[object] --> A[A]
    object --> C[C]
    object --> F[F]
    A --> B[B]
    A --> L[L]
    F --> G[G]
    F --> H[H]
    B --> D[D]
    C --> D[D]
    G --> D[D]
    H --> D[D]
    L --> D[D]

```

**D.\_\_mro\_\_**

```

(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.G'>,
<class '__main__.H'>, <class '__main__.F'>, <class '__main__.L'>, <class '__main__.A'>,
<class 'object'>)

```

Programmazione Avanzata a.a. 2023-24  
A. De Bonis

65

## 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__` qui usiamo la funzione `setattr`
  - `setattr(Derivata, '__bases__', (Base2, Base1))`

66



## 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 definiscano il metodo `metodo_base`
  - D sia derivata da A, B e C ( `class D(A,B,C): pass` )
  - d sia 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)

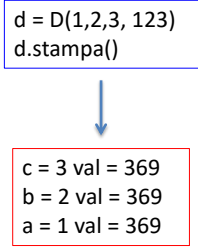
class C():
    def __init__(self, c, val):
        self._c = c
        self._val = val

    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)

```



Programmazione Avanzata a.a. 2023-24  
A. De Bonis

69

**Programmazione avanzata a.a. 2022-23**  
**A. De Bonis**  
**Introduzione a Python**  
**VII lezione**

70

## Iteratori

- Se una classe supporta l'iteratore possiamo ottenere un riferimento ad esso tramite la funzione `iter()`
  - Si invoca `iter` passando come argomento 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. 2023-24  
A. De Bonis

71

71

```
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. 2023-24  
A. De Bonis

72

72

## 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. 2023-24  
A. De Bonis

73

73

## Gestire l'eccezione

- Ad una clausola `try` possono essere associate più clausole `except`.
- Una clausola `except` può includere più eccezioni (all'interno di una tupla)
  - esempio: `... except (RuntimeError, TypeError, NameError):`  
`... pass`
- Una clausola `except` che include un'eccezione di un certo tipo `A` può gestire solo eccezioni di tipo `A` o di sottoclassi di `A`.
- L'ultima clausola `except` può omettere i nomi delle eccezioni. Questo uso di `except` deve essere fatto con molta cautela perché potrebbe nascondere un vero errore di programmazione. Può essere usato per stampare un messaggio di errore e lanciare nuovamente l'eccezione.
- Lo statement `try ... except` ha una clausola `else` opzionale che, quando presente, deve seguire tutte le clausole `except`. È utile per inserire codice che deve essere eseguito quando la clausola `try` non lancia un'eccezione.
- Lo statement `try ... except` può contenere una clausola `finally` che viene eseguita immediatamente prima che venga completato lo statement `try ... except`. Questa clausola viene eseguita sia nel caso in cui il `try` produca un'eccezione sia nel caso non venga prodotta un'eccezione. (vedere esempi nella documentazione)

Programmazione Avanzata a.a. 2023-24  
A. De Bonis

74

74