



## **Corso di Apprendistato Python**

Mod. Python  
Docente:  
Tonino Petrulli



## Ereditarietà: `isinstance()`

Come già sapete, **un oggetto è l'incarnazione di una classe**. Ciò significa che l'oggetto è come una torta preparata con una ricetta inclusa nella classe.

Questo può generare alcuni problemi importanti.

Supponiamo di avere una torta (ad esempio, come argomento passato alla funzione). Si vuole sapere quale ricetta è stata utilizzata per realizzarla. Perché? Perché si vuole sapere cosa ci si aspetta da questa torta, ad esempio se contiene frutta secca o meno, un'informazione fondamentale per alcune persone.

Allo stesso modo, può essere fondamentale se l'oggetto ha (o non ha) determinate caratteristiche. In altre parole, **se si tratta di un oggetto di una certa classe o meno**.

Tale fatto potrebbe essere rilevato dalla funzione `isinstance()`:

`isinstance(objectName, ClassName)`

La funzione restituisce `True` se l'oggetto è un'istanza della classe, altrimenti `False`.

**Essere un'istanza di una classe significa che l'oggetto (la torta) è stato preparato utilizzando una ricetta contenuta nella classe o in una delle sue superclassi.**

Non dimenticate: se una sottoclasse contiene almeno lo stesso equipaggiamento di una qualsiasi delle sue superclassi, significa che gli oggetti della sottoclasse possono fare le stesse cose degli oggetti derivati dalla superclasse, ergo, è un'istanza della sua classe di origine e di una qualsiasi delle sue superclassi.

Testiamolo. Analizzare il codice

```
class Vehicle:  
    pass
```

```
class LandVehicle(Vehicle):  
    pass
```

```
class TrackedVehicle(LandVehicle):  
    pass
```

```
my_vehicle = Vehicle()  
my_land_vehicle = LandVehicle()  
my_tracked_vehicle = TrackedVehicle()
```

```
for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:  
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:  
        print(isinstance(obj, cls), end="\t")  
    print()
```

Abbiamo creato tre oggetti, uno per ciascuna delle classi. Quindi, utilizzando due cicli annidati, controlliamo tutte le possibili coppie oggetto-classe **per scoprire se gli oggetti sono istanze delle classi**.

Eseguire il codice.

Questo è ciò che otteniamo:

True False False

True True False

True True True

Rendiamo ancora una volta il risultato più leggibile:

↓ is an instance of →	Vehi- cle	LandVehi- cle	TrackedVehi- cle
my_vehicle	True	False	False
my_land_vehicle	True	True	False
my_tracked_vehicle	True	True	True

La tabella conferma le nostre aspettative?

## **Ereditarietà: l'operatore is**

C'è anche un operatore Python che vale la pena menzionare, poiché si riferisce direttamente agli oggetti: eccolo: `oggetto_uno` è `oggetto_due`

**L'operatore `is` verifica se due variabili (`oggetto_uno` e `oggetto_due` in questo caso) si riferiscono allo stesso oggetto.**

Non dimenticate che le **variabili non memorizzano gli oggetti stessi, ma solo gli handle che puntano alla memoria interna di Python.**

Assegnare il valore di una variabile oggetto a un'altra variabile non copia l'oggetto, ma solo il suo handle. Ecco perché un operatore come `is` può essere molto utile in particolari circostanze.

Date un'occhiata al codice

```
class SampleClass:  
    def __init__(self, val):  
        self.val = val
```

```
object_1 = SampleClass(0)  
object_2 = SampleClass(2)  
object_3 = object_1  
object_3.val += 1
```

```
print(object_1 is object_2)  
print(object_2 is object_3)  
print(object_3 is object_1)  
print(object_1.val, object_2.val, object_3.val)
```

```
string_1 = "Mary had a little "  
string_2 = "Mary had a little lamb"  
string_1 += "lamb"
```

```
print(string_1 == string_2, string_1 is string_2)
```

- c'è una classe molto semplice dotata di un semplice costruttore, che crea solo una proprietà. La classe viene utilizzata per istanziare due oggetti. Il primo viene assegnato a un'altra variabile e la sua proprietà val viene incrementata di uno.
- Successivamente, l'operatore is viene applicato tre volte per controllare tutte le possibili coppie di oggetti e vengono stampati anche tutti i valori delle proprietà val.
- l'ultima parte del codice esegue un altro esperimento. Dopo tre assegnazioni, entrambe le stringhe contengono gli stessi testi, ma **questi sono memorizzati in oggetti diversi**.

Il codice viene stampato:

Falso

Falso

Vero

1 2 1

Vero Falso

I risultati dimostrano che l'oggetto\_1 e l'oggetto\_3 sono effettivamente gli stessi oggetti, mentre la stringa\_1 e la stringa\_2 non lo sono, nonostante il loro contenuto sia lo stesso.

## **Come Python trova proprietà e metodi**

Ora vedremo come Python gestisce l'ereditarietà dei metodi.

Guardate l'esempio

```
class Super:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "My name is " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)

obj = Sub("Andy")
print(obj)
```



- esiste una classe denominata Super, che definisce il proprio costruttore utilizzato per assegnare la proprietà dell'oggetto, denominata name.
- la classe definisce anche il metodo `__str__()`, che la rende in grado di presentare la propria identità in chiaro.
- la classe viene poi utilizzata come base per creare una sottoclasse chiamata Sub. La classe Sub definisce il proprio costruttore, che richiama quello della superclasse. Si noti come è stato fatto: `Super.__init__(self, name)`.
- abbiamo nominato esplicitamente la superclasse e abbiamo indicato il metodo da invocare `__init__()`, fornendo tutti gli argomenti necessari.
- abbiamo istanziato un oggetto di classe Sub e lo abbiamo stampato.

Il codice viene emesso:

My name is Andy.

Nota: poiché non esiste un metodo `__str__()` all'interno della classe Sub, la stringa stampata deve essere prodotta all'interno della classe Super. Ciò significa che il metodo `__str__()` è stato ereditato dalla classe Sub.

## **Come Python trova proprietà e metodi: continua**

Guardate il codice

```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name + "."
```

```
class Sub(Super):
    def __init__(self, name):
        super().__init__(name)
```

```
obj = Sub("Andy")
```

```
print(obj)
```

Lo abbiamo modificato per mostrare un altro metodo di accesso a qualsiasi entità definita all'interno della superclasse.

Nell'ultimo esempio, abbiamo nominato esplicitamente la superclasse. In questo esempio, utilizziamo la funzione `super()`, che **accede alla superclasse senza bisogno di conoscerne il nome**:

```
super().__init__(nome)
```

La funzione `super()` crea un contesto in cui non è necessario (anzi, non si deve) passare l'argomento `self` al metodo invocato: ecco perché è possibile attivare il costruttore della superclasse utilizzando un solo argomento.

Nota: è possibile utilizzare questo meccanismo non solo per **invocare il costruttore della superclasse, ma anche per ottenere l'accesso a qualsiasi risorsa disponibile all'interno della superclasse**.

## Come Python trova proprietà e metodi: continua

Proviamo a fare qualcosa di simile, ma con le proprietà (più precisamente: con le **variabili di classe**).

```
# Testing properties: class variables.
```

```
class Super:
```

```
    supVar = 1
```

```
class Sub(Super):
```

```
    subVar = 2
```

```
obj = Sub()
```

```
print(obj.subVar)
```

```
print(obj.supVar)
```

Come si può notare, la classe Super definisce una variabile di classe denominata supVar e la classe Sub definisce una variabile denominata subVar.

Entrambe le variabili sono visibili all'interno dell'oggetto di classe Sub: ecco perché il codice viene visualizzato:

```
2
```

```
1
```

### **Come Python trova proprietà e metodi: continua**

Lo stesso effetto si può osservare con le **variabili di istanza**:

# Testing properties: instance variables.

```
class Super:
```

```
    def __init__(self):  
        self.supVar = 11
```

```
class Sub(Super):
```

```
    def __init__(self):  
        super().__init__()  
        self.subVar = 12
```

```
obj = Sub()
```

```
print(obj.subVar)
```

```
print(obj.supVar)
```

Il costruttore della classe Sub crea una variabile di istanza denominata subVar, mentre il costruttore di Super fa lo stesso con una variabile denominata supVar. Come in precedenza, entrambe le variabili sono accessibili dall'interno dell'oggetto di classe Sub.

L'output del programma è:

12

11

Nota: l'esistenza della variabile supVar è ovviamente condizionata dall'invocazione del costruttore della classe Super. Se la si omette, la variabile non sarà presente nell'oggetto creato

### **Come Python trova proprietà e metodi: continua**

È ora possibile formulare una dichiarazione generale che descriva il comportamento di Python.

Quando si cerca di accedere all'entità di un oggetto, Python cercherà di farlo (in questo ordine):

- trovarlo **all'interno dell'oggetto** stesso;
- Lo si trova **in tutte le classi** coinvolte nella linea di ereditarietà dell'oggetto, dal basso verso l'alto;

Se entrambe le operazioni falliscono, viene **sollevata** un'**eccezione** (AttributeError).

La prima condizione potrebbe richiedere un'ulteriore attenzione. Come è noto, tutti gli oggetti che derivano da una particolare classe possono avere diversi set di attributi e alcuni di essi possono essere aggiunti all'oggetto molto tempo dopo la sua creazione.

L'esempio che segue riassume questo aspetto in una **linea di eredità a tre livelli**.

```
class Level1:
    variable_1 = 100
    def __init__(self):
        self.var_1 = 101
    def fun_1(self):
        return 102
```

```
class Level2(Level1):
    variable_2 = 200
    def __init__(self):
        super().__init__()
        self.var_2 = 201
    def fun_2(self):
        return 202
```

```
class Level3(Level2):
    variable_3 = 300
    def __init__(self):
        super().__init__()
        self.var_3 = 301
    def fun_3(self):
        return 302
```



```
obj = Level3()
```

```
print(obj.variable_1, obj.var_1, obj.fun_1())
```

```
print(obj.variable_2, obj.var_2, obj.fun_2())
```

```
print(obj.variable_3, obj.var_3, obj.fun_3())
```

Tutti i commenti fatti finora riguardano l'**ereditarietà singola**, quando una sottoclasse ha esattamente una superclasse. Questa è la situazione più comune (e anche quella consigliata).

Python, tuttavia, offre molto di più. Nelle prossime lezioni vi mostreremo alcuni esempi di **ereditarietà multipla**.

### **Come Python trova proprietà e metodi: continua**

**L'ereditarietà multipla si verifica quando una classe ha più di una superclasse.** Sintatticamente, tale ereditarietà si presenta come un elenco di superclassi separato da virgole e inserito tra parentesi dopo il nome della nuova classe, proprio come in questo caso:

```
class SuperA:
```

```
    var_a = 10
```

```
    def fun_a(self):
```

```
        return 11
```

```
class SuperB:
```

```
    var_b = 20
```

```
    def fun_b(self):
```

```
        return 21
```

```
class Sub(SuperA, SuperB):
```

```
    pass
```

```
obj = Sub()
```

```
print(obj.var_a, obj.fun_a())
```

```
print(obj.var_b, obj.fun_b())
```

La classe Sub ha due superclassi: SuperA e SuperB. Ciò significa che la classe Sub **eredita tutti i beni offerti da SuperA e SuperB.**

Il codice viene stampato:

10 11

20 21

Ora è il momento di introdurre un nuovo termine: **overriding.**

Cosa pensate che accadrà se più di una delle superclassi definisce un'entità con un nome particolare?

## Come Python trova proprietà e metodi: continua

Analizziamo l'esempio

```
class Level1:
    var = 100
    def fun(self):
        return 101
class Level2(Level1):
    var = 200
    def fun(self):
        return 201
class Level3(Level2):
    pass
obj = Level3()
print(obj.var, obj.fun())
```

Entrambe le classi Level1 e Level2 definiscono un metodo chiamato fun() e una proprietà chiamata var. Questo significa che l'oggetto della classe Level3 potrà accedere a due copie di ciascuna entità? Assolutamente no.

**L'entità definita successivamente (nel senso dell'ereditarietà) sovrascrive la stessa entità definita**

**precedentemente.** Questo è il motivo per cui il codice produce il seguente risultato:

200 201

Come si può notare, la variabile di classe `var` e il metodo `fun()` della classe `Level2` sovrascrivono le entità con gli stessi nomi derivate dalla classe `Level1`.

Questa caratteristica può essere usata intenzionalmente per modificare i comportamenti predefiniti (o definiti in precedenza) delle classi, quando una di esse deve agire in modo diverso dai suoi antenati.

Possiamo anche dire che **Python cerca un'entità dal basso verso l'alto** e si accontenta della prima entità con il nome desiderato.

Come funziona quando una classe ha due antenati che offrono la stessa entità e si trovano sullo stesso livello? In altre parole, cosa ci si deve aspettare quando una classe emerge utilizzando l'ereditarietà multipla? Vediamo questo.

## **Come Python trova proprietà e metodi: continua**

Diamo un'occhiata all'esempio

```
class Left:
```

```
    var = "L"
```

```
    var_left = "LL"
```

```
    def fun(self):
```

```
        return "Left"
```

```
class Right:
```

```
    var = "R"
```

```
    var_right = "RR"
```

```
    def fun(self):
```

```
        return "Right"
```

```
class Sub(Left, Right):
```

```
    pass
```

```
obj = Sub()
```

```
print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

La classe Sub eredita beni da due superclassi, Left e Right (questi nomi sono intesi come significativi). Non c'è dubbio che la variabile di classe `var_right` provenga dalla classe Right e che `var_left` provenga rispettivamente da Left.

Questo è chiaro. Ma da dove viene `var`? È possibile indovinarlo? Lo stesso problema si presenta con il metodo `fun()`: sarà invocato da sinistra o da destra? Eseguiamo il programma: l'output è:

```
L LL RR Left
```

Questo dimostra che entrambi i casi poco chiari hanno una soluzione all'interno della classe Sinistra. È una premessa sufficiente per formulare una regola generale? Sì, lo è.

Possiamo dire che **Python cerca i componenti degli oggetti** nel seguente ordine:

- **all'interno dell'oggetto** stesso;
- **nelle sue superclassi**, dal basso verso l'alto;
- se c'è più di una classe su un particolare percorso di ereditarietà, Python le analizza da sinistra a destra.

Avete bisogno di qualcosa di più? Basta apportare una piccola modifica al codice, sostituendo: `class Sub(Left, Right):` con: `class Sub(Right, Left):`, quindi eseguire nuovamente il programma e vedere cosa succede.

Cosa vedete ora? Vediamo:

```
R LL RR Right
```

Vedete la stessa cosa o qualcosa di diverso?

## **Come costruire una gerarchia di classi**

Costruire una gerarchia di classi non è solo arte per l'arte.

Se si divide un problema tra le classi e si decide quali di esse debbano essere collocate in cima e quali in fondo alla gerarchia, è necessario analizzare attentamente la questione, ma prima di mostrare come farlo (e come non farlo), vogliamo evidenziare un effetto interessante. Non è nulla di straordinario (è solo una conseguenza delle regole generali presentate in precedenza), ma ricordarlo può essere fondamentale per capire come funzionano alcuni codici e come l'effetto può essere utilizzato per costruire un insieme flessibile di classi.

Date un'occhiata al codice

```
class One:
    def do_it(self):
        print("do_it from One")
    def doanything(self):
        self.do_it()
```

```
class Two(One):
    def do_it(self):
        print("do_it from Two")
```

```
one = One()
two = Two()
one.doanything()
two.doanything()
```



- ci sono due classi, chiamate Uno e Due, mentre Due deriva da Uno. Niente di speciale. Tuttavia, una cosa sembra notevole: il metodo `do_it()`.
- il metodo `do_it()` è **definito due volte**: originariamente all'interno di One e successivamente all'interno di Two. L'essenza dell'esempio sta nel fatto che viene **invocato una sola volta**, all'interno di Uno.

La domanda è: quale dei due metodi sarà invocato dalle ultime due righe del codice?

La prima invocazione sembra semplice, e in effetti lo è: invocando `doanything()` dall'oggetto chiamato uno si attiverà ovviamente il primo dei metodi.

La seconda invocazione richiede un po' di attenzione. È anche semplice, se si tiene presente come Python trova i componenti delle classi. La seconda invocazione lancerà `do_it()` nel modulo esistente all'interno della classe Two, indipendentemente dal fatto che l'invocazione avvenga all'interno della classe One.

In effetti, il codice produce il seguente risultato:

```
do_it from One
```

```
do_it from Two
```

Nota: la situazione in cui **la sottoclasse è in grado di modificare il comportamento della superclasse (proprio come nell'esempio) è chiamata polimorfismo**. La parola deriva dal greco (polys: "molti, molto" e morphe, "forma, sagoma"), il che significa che una stessa classe può assumere varie forme a seconda delle ridefinizioni effettuate da una qualsiasi delle sue sottoclassi.

Il metodo, ridefinito in una qualsiasi delle superclassi, modificando così il comportamento della superclasse, è detto **virtuale**.

In altre parole, nessuna classe è data una volta per tutte. Il comportamento di ogni classe può essere modificato in qualsiasi momento da una qualsiasi delle sue sottoclassi.

Vi mostreremo **come utilizzare il polimorfismo per estendere la flessibilità delle classi**.

## **Come costruire una gerarchia di classi: continua**

Guardate l'esempio

```
import time
class TrackedVehicle:
    def control_track(left, stop):
        pass
    def turn(left):
        control_track(left, True)
        time.sleep(0.25)
        control_track(left, False)

class WheeledVehicle:
    def turn_front_wheels(left, on):
        pass
    def turn(left):
        turn_front_wheels(left, True)
        time.sleep(0.25)
        turn_front_wheels(left, False)
```

Assomiglia a qualcosa? Sì, certo che assomiglia. Si riferisce all'esempio mostrato all'inizio del modulo, quando abbiamo parlato dei concetti generali della programmazione oggettiva.

Può sembrare strano, ma non abbiamo usato l'ereditarietà in alcun modo, solo per dimostrarvi che non ci limita e che siamo riusciti a ottenere la nostra.

Abbiamo definito due classi separate in grado di produrre due diversi tipi di veicoli terrestri. La differenza principale è nel modo in cui girano. Un veicolo a ruote gira solo le ruote anteriori (in genere). Un veicolo cingolato deve fermare uno dei binari.

- un veicolo cingolato esegue una svolta fermandosi e muovendosi su uno dei suoi binari (questo viene fatto dal metodo `control_track()`, che verrà implementato in seguito)
- un veicolo a ruote gira quando le sue ruote anteriori girano (questo viene fatto dal metodo `turn_front_wheels()`)
- il metodo `turn()` utilizza il metodo adatto a ciascun veicolo.

Riesci a capire **cosa c'è di sbagliato nel codice?**

I metodi `turn()` sono troppo simili per lasciarli in questa forma.

Ricostruiamo il codice: introdurremo una superclasse per raccogliere tutti gli aspetti simili dei veicoli di guida, spostando tutte le specificità nelle sottoclassi.

## Come costruire una gerarchia di classi: continua

Guardate di nuovo il codice

```
import time
class Vehicle:
    def change_direction(left, on):
        pass
    def turn(left):
        change_direction(left, True)
        time.sleep(0.25)
        change_direction(left, False)

class TrackedVehicle(Vehicle):
    def control_track(left, stop):
        pass
    def change_direction(left, on):
        control_track(left, on)

class WheeledVehicle(Vehicle):
    def turn_front_wheels(left, on):
        pass
    def change_direction(left, on):
        turn_front_wheels(left, on)
```

Ecco cosa abbiamo fatto:

- abbiamo definito una superclasse chiamata Veicolo, che utilizza il metodo turn() per implementare uno schema generale di svolta, mentre la svolta vera e propria è effettuata da un metodo chiamato change\_direction(); si noti: il primo metodo è vuoto, poiché metteremo tutti i dettagli nella sottoclasse (un metodo di questo tipo è spesso chiamato **metodo astratto**, poiché dimostra solo alcune possibilità che saranno istanziate in seguito)
- abbiamo definito una sottoclasse denominata TrackedVehicle (nota: è derivata dalla classe Vehicle) che istanzia il metodo change\_direction() utilizzando il metodo specifico (concreto) denominato control\_track()
- rispettivamente, la sottoclasse denominata WheeledVehicle esegue lo stesso trucco, ma utilizza il metodo turn\_front\_wheels() per forzare il veicolo a girare.

Il vantaggio più importante (tralasciando i problemi di leggibilità) è che questa forma di codice consente di implementare un nuovo algoritmo di svolta semplicemente modificando il metodo turn(), che può essere fatto in un solo punto, poiché tutti i veicoli lo rispetteranno.

Ecco come **il polimorfismo aiuta lo sviluppatore a mantenere il codice pulito e coerente.**

### **Come costruire una gerarchia di classi: continua**

L'ereditarietà non è l'unico modo per costruire classi adattabili. È possibile raggiungere gli stessi obiettivi (non sempre, ma molto spesso) utilizzando una tecnica chiamata composizione.

**La composizione è il processo di composizione di un oggetto utilizzando altri oggetti diversi.** Gli oggetti utilizzati nella composizione forniscono un insieme di caratteristiche desiderate (proprietà e/o metodi), per cui possiamo dire che agiscono come blocchi utilizzati per costruire una struttura più complessa.

Si può dire che:

- **L'ereditarietà estende le capacità di una classe** aggiungendo nuovi componenti e modificando quelli esistenti; in altre parole, la ricetta completa è contenuta nella classe stessa e in tutti i suoi antenati; l'oggetto prende tutte le proprietà della classe e le utilizza;
- **La composizione proietta una classe come un contenitore** in grado di memorizzare e utilizzare altri oggetti (derivati da altre classi) in cui ciascuno degli oggetti implementa una parte del comportamento della classe desiderata.

Illustriamo la differenza utilizzando i veicoli definiti in precedenza. L'approccio precedente ci portava a una gerarchia di classi in cui la classe più in alto era a conoscenza delle regole generali utilizzate per far girare il veicolo, ma non sapeva come controllare i componenti appropriati (ruote o cingoli).

Le sottoclassi hanno implementato questa capacità introducendo meccanismi specializzati. Facciamo (quasi) la stessa cosa, ma utilizzando la composizione. La classe, come nell'esempio precedente, sa come girare il veicolo, ma la svolta vera e propria è affidata a un oggetto specializzato, memorizzato in una proprietà chiamata controller. Il controllore è in grado di controllare il veicolo manipolando le parti del veicolo stesso.

Ecco come potrebbe apparire

```
import time
class Tracks:
    def change_direction(self, left, on):
        print("tracks: ", left, on)

class Wheels:
    def change_direction(self, left, on):
        print("wheels: ", left, on)

class Vehicle:
    def __init__(self, controller):
        self.controller = controller
    def turn(self, left):
        self.controller.change_direction(left, True)
        time.sleep(0.25)
        self.controller.change_direction(left, False)

wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())
wheeled.turn(True)
tracked.turn(False)
```



Esistono due classi, denominate Tracks e Wheels, che sanno come controllare la direzione del veicolo. Esiste anche una classe chiamata Veicolo, che può utilizzare uno qualsiasi dei controllori disponibili (i due già definiti, o qualsiasi altro definito in futuro) - il controllore stesso viene passato alla classe durante l'inizializzazione. In questo modo, la capacità di girare del veicolo è composta da un oggetto esterno, non implementato all'interno della classe Veicolo. In altre parole, abbiamo un veicolo universale sul quale possiamo installare sia i cingoli che le ruote. Il codice produce il seguente risultato:

```
wheels: True True  
wheels: True False  
tracks: False True  
tracks: False False
```

## Eredità singola vs. eredità multipla

Come già sapete, non ci sono ostacoli all'uso dell'ereditarietà multipla in Python. È possibile derivare qualsiasi nuova classe da più classi definite in precedenza.

C'è solo un "ma". Il fatto che possiate farlo non significa che dobbiate farlo.

Non dimenticate che:

- una singola classe ereditaria è sempre più semplice, più sicura, più facile da capire e da mantenere;
- L'ereditarietà multipla è sempre rischiosa, perché si hanno molte più opportunità di sbagliare nell'identificare le parti delle superclassi che influenzeranno effettivamente la nuova classe;
- L'ereditarietà multipla può rendere estremamente complicato l'override; inoltre, l'uso della funzione `super()` diventa ambiguo
- L'ereditarietà multipla viola il **principio** della **singola responsabilità** (maggiori dettagli qui: [https://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)), poiché crea una nuova classe da due (o più) classi che non fanno nulla l'una dell'altra;
- suggeriamo vivamente l'ereditarietà multipla come l'ultima delle soluzioni possibili: se si ha davvero bisogno di molte funzionalità diverse offerte da classi diverse, la composizione può essere un'alternativa migliore.

## **Che cos'è il Method Resolution Order (MRO) e perché non tutte le ereditarietà hanno senso?**

L'MRO, in generale, è un modo (si può chiamare **strategia**) con cui un particolare linguaggio di programmazione esamina la parte superiore della gerarchia di una classe per trovare il metodo di cui ha bisogno. Vale la pena sottolineare che linguaggi diversi utilizzano MRO leggermente (o addirittura completamente) diversi. Python, tuttavia, è una creatura unica da questo punto di vista e le sue abitudini sono un po' specifiche.

Mostreremo come funziona l'MRO di Python in due casi particolari, che sono chiari esempi di problemi che possono verificarsi quando si cerca di usare l'ereditarietà multipla in modo troppo avventato. Cominciamo con uno snippet che inizialmente può sembrare semplice. Guardate il codice seguente

```
class Top:
```

```
    def m_top(self):  
        print("top")
```

```
class Middle(Top):
```

```
    def m_middle(self):  
        print("middle")
```

```
class Bottom(Middle):
```

```
    def m_bottom(self):  
        print("bottom")
```

```
object = Bottom()
```

```
object.m_bottom()
```

```
object.m_middle()
```

```
object.m_top()
```

Siamo certi che se analizzerete voi stessi lo snippet, non noterete alcuna anomalia. Sì, avete perfettamente ragione: sembra chiaro e semplice e non desta alcuna preoccupazione. Se si esegue il codice, si otterrà il seguente risultato prevedibile:

bottom  
middle  
top

Finora non ci sono state sorprese. Apportiamo una piccola modifica al codice. Date un'occhiata:

```
class Top:  
    def m_top(self):  
        print("top")
```

```
class Middle(Top):  
    def m_middle(self):  
        print("middle")
```

```
class Bottom(Middle, Top):  
    def m_bottom(self):  
        print("bottom")
```

```
object = Bottom()  
object.m_bottom()  
object.m_middle()  
object.m_top()
```

Riesci a vedere la differenza? È nascosta in questa riga:

```
class Bottom(Middle, Top):
```

In questo modo esotico, abbiamo trasformato un codice molto semplice con un chiaro percorso di ereditarietà singola in un misterioso enigma di ereditarietà multipla. "È valido?", vi chiederete. Sì, lo è. "Come è possibile?", dovrete chiedervi ora, e speriamo che sentiate davvero il bisogno di porvi questa domanda.

Come si può notare, l'ordine in cui le due superclassi sono state elencate tra parentesi è conforme alla struttura del codice: la classe Middle precede la classe Top, proprio come nel percorso di ereditarietà reale.

Nonostante la sua stranezza, l'esempio è corretto e funziona come previsto, ma va detto che questa notazione non apporta alcuna nuova funzionalità o significato aggiuntivo.

Modifichiamo ancora una volta il codice: ora scambieremo entrambi i nomi delle superclassi nella definizione della classe Bottom. Ecco come appare ora

```
class Top:
    def m_top(self):
        print("top")
class Middle(Top):
    def m_middle(self):
        print("middle")
class Bottom(Top, Middle):
    def m_bottom(self):
        print("bottom")
object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

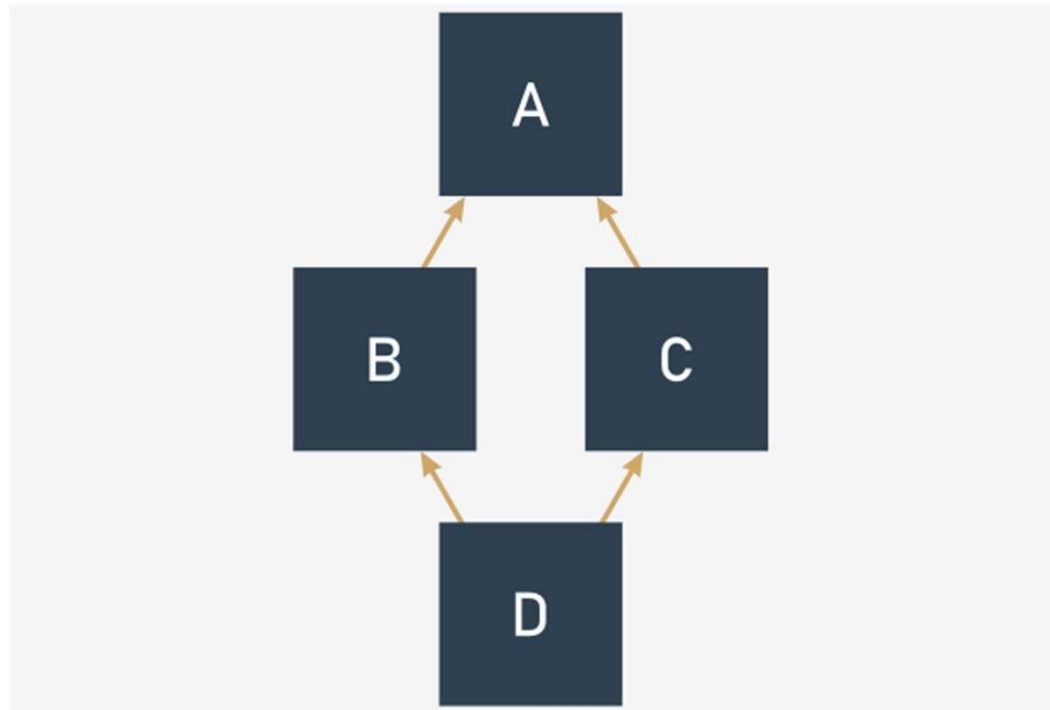
Per anticipare la sua domanda, diremo che questa modifica ha rovinato il codice e non funzionerà più. Che peccato. L'ordine che abbiamo cercato di imporre (Top, Middle) è incompatibile con il percorso di ereditarietà che deriva dalla struttura del codice. A Python non piacerà. Ecco cosa vedremo:

`TypeError: Cannot create a consistent method resolution order (MRO) for bases Top, Middle`

Pensiamo che il messaggio parli da solo. La MRO di Python non può essere piegata o violata, non solo perché è così che funziona Python, ma anche perché è una regola da rispettare.

## Il problema dei diamanti

Il secondo esempio dello spettro di problemi che possono sorgere dall'ereditarietà multipla è illustrato da un problema classico chiamato **problema del diamante**. Il nome riflette la forma del diagramma di ereditarietà: guardate l'immagine:





- c'è la superclasse più in alto, denominata A;
- esistono due sottoclassi derivate da A: B e C;
- e c'è anche la sottoclasse più in basso, denominata D, derivata da B e C (o C e B, poiché queste due varianti hanno significati diversi in Python)

Riesci a vedere il diamante?

Date un'occhiata al codice

```
class A:  
    pass
```

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

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

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

```
d = D()
```

La stessa struttura, ma espressa in Python.

Alcuni linguaggi di programmazione non consentono affatto l'ereditarietà multipla e, di conseguenza, non permettono di costruire un diamante: questa è la strada che Java e C# hanno scelto di seguire fin dalle loro origini.

Python, tuttavia, ha scelto una strada diversa: consente l'ereditarietà multipla e non si preoccupa se si scrive e si esegue codice come quello dell'editor. Ma non dimenticatevi di MRO, che è sempre al comando.

Ricostruiamo l'esempio della pagina precedente per renderlo più simile a un diamante, come qui sotto:

```
class Top:  
    def m_top(self):  
        print("top")
```

```
class Middle_Left(Top):  
    def m_middle(self):  
        print("middle_left")
```

```
class Middle_Right(Top):  
    def m_middle(self):  
        print("middle_right")
```

```
class Bottom(Middle_Left, Middle_Right):  
    def m_bottom(self):  
        print("bottom")
```

```
object = Bottom()  
object.m_bottom()  
object.m_middle()  
object.m_top()
```

Nota: entrambe le classi Middle definiscono **un metodo con lo stesso nome**: `m_middle()`.

Introduce una piccola incertezza nel nostro esempio, anche se siamo assolutamente sicuri che si possa rispondere alla seguente domanda chiave: quale dei due metodi `m_middle()` sarà effettivamente invocato quando viene eseguita la riga seguente?

```
Object.m_middle()
```

In altre parole, cosa vedrete sullo schermo: `: middle_left` or `middle_right`?

Non c'è bisogno di affrettarsi: pensateci due volte e tenete presente l'MRO di Python!

Siete pronti?

Sì, avete ragione. L'invocazione attiverà il metodo `m_middle()`, che proviene dalla classe `Middle_Left`. La spiegazione è semplice: la classe è elencata prima di `Middle_Right` nell'elenco di ereditarietà della classe `Bottom`. Se volete essere sicuri che non ci siano dubbi, provate a scambiare queste due classi nell'elenco e verificate i risultati.

Se volete provare impressioni più profonde sull'ereditarietà multipla e sulle gemme preziose, provate a modificare il nostro snippet e a dotare la classe `Top` di un altro esemplare del metodo `m_middle()` e studiate attentamente il suo comportamento.

Come potete vedere, i diamanti possono portare alcuni problemi nella vostra vita, sia quelli reali che quelli offerti da Python.

## Punti di forza

1. Un metodo chiamato `__str__()` è responsabile della **conversione del contenuto di un oggetto in una stringa (più o meno) leggibile**. È possibile ridefinirlo se si vuole che l'oggetto sia in grado di presentarsi in una forma più elegante. Per esempio:

```
class Mouse:
```

```
    def __init__(self, name):  
        self.my_name = name
```

```
    def __str__(self):  
        return self.my_name
```

```
the_mouse = Mouse('mickey')  
print(the_mouse) # Prints "mickey".
```

2. Una funzione denominata `issubclass(Class_1, Class_2)` è in grado di determinare se la Classe\_1 è una **sottoclasse** della Classe\_2. Ad esempio:

```
class Mouse:  
    pass
```

```
class LabMouse(Mouse):  
    pass
```

```
print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse)) # Prints "False True"
```

3. Una funzione chiamata `isinstance(Object, Class)` verifica se un oggetto **proviene da una classe indicata**. Ad esempio:

```
class Mouse:  
    pass
```

```
class LabMouse(Mouse):  
    pass
```

```
mickey = Mouse()  
print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse)) # Prints "True False".
```

4. Un operatore chiamato controlla se due variabili si riferiscono allo **stesso oggetto**. Ad esempio:

```
class Mouse:  
    pass
```

```
mickey = Mouse()  
minnie = Mouse()  
cloned_mickey = mickey  
print(mickey is minnie, mickey is cloned_mickey) # Prints "False True".
```

5. Una funzione senza parametri, denominata `super()`, restituisce un **riferimento alla più vicina superclasse della classe**. Ad esempio:

```
class Mouse:  
    def __str__(self):  
        return "Mouse"
```

```
class LabMouse(Mouse):  
    def __str__(self):  
        return "Laboratory " + super().__str__()
```

```
doctor_mouse = LabMouse();  
print(doctor_mouse) # Prints "Laboratory Mouse".
```

6. I metodi e le variabili di istanza e di classe definiti in una superclasse sono **automaticamente ereditati** dalle sue sottoclassi. Ad esempio:

```
class Mouse:
```

```
    Population = 0
```

```
    def __init__(self, name):
```

```
        Mouse.Population += 1
```

```
        self.name = name
```

```
    def __str__(self):
```

```
        return "Hi, my name is " + self.name
```

```
class LabMouse(Mouse):
```

```
    pass
```

```
professor_mouse = LabMouse("Professor Mouser")
```

```
print(professor_mouse, Mouse.Population) # Prints "Hi, my name is Professor Mouser 1"
```



7. Per trovare una proprietà di un oggetto/classe, Python la cerca all'interno:

- l'oggetto stesso;
- tutte le classi coinvolte nella linea di ereditarietà dell'oggetto, dal basso verso l'alto;
- se c'è più di una classe su un particolare percorso di ereditarietà, Python le analizza da sinistra a destra;
- se entrambe le operazioni falliscono, viene sollevata l'eccezione `AttributeError`.

8. Se una delle sottoclassi definisce un metodo/una variabile di classe/una variabile di istanza con lo stesso nome esistente nella superclasse, il nuovo nome **sovrascrive** qualsiasi istanza precedente del nome. Ad esempio:

```
class Mouse:
```

```
    def __init__(self, name):  
        self.name = name
```

```
    def __str__(self):  
        return "My name is " + self.name
```

```
class AncientMouse(Mouse):
```

```
    def __str__(self):  
        return "Meum nomen est " + self.name
```

```
mus = AncientMouse("Caesar") # Prints "Meum nomen est Caesar"  
print(mus)
```

## Esercizi

### Scenario

Si supponga che il seguente pezzo di codice sia stato eseguito con successo:

```
class Dog:
    kennel = 0
    def __init__(self, breed):
        self.breed = breed
        Dog.kennel += 1
    def __str__(self):
        return self.breed + " says: Woof!"

class SheepDog(Dog):
    def __str__(self):
        return super().__str__() + " Don't run away, Little Lamb!"

class GuardDog(Dog):
    def __str__(self):
        return super().__str__() + " Stay where you are, Mister Intruder!"

rocky = SheepDog("Collie")
luna = GuardDog("Dobermann")
```

Rispondete ora alle domande degli esercizi 1-4.

### **Esercizio 1**

Qual è l'output previsto per il seguente pezzo di codice?

```
print(rocky)  
print(luna)
```

### **Esercizio 2**

Qual è l'output previsto per il seguente pezzo di codice?

```
print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))  
print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))
```

### **Esercizio 3**

Qual è l'output previsto per il seguente pezzo di codice?

```
print(luna is luna, rocky is luna)  
print(rocky.kennel)
```

### **Esercizio 4**

Definire una sottoclasse di SheepDog chiamata LowlandDog e dotarla di un metodo `__str__()` che sovrascriva un metodo ereditato con lo stesso nome. Il metodo `__str__()` del nuovo dog dovrebbe restituire la stringa "Woof! I don't like mountains!" .

Rispondete ora alle domande degli esercizi 1-4.

### **Esercizio 1**

Qual è l'output previsto per il seguente pezzo di codice?

```
print(rocky)
print(luna)
```

Soluzione

Collie says: Woof! Don't run away, Little Lamb!

Dobermann says: Woof! Stay where you are, Mister Intruder!

### **Esercizio 2**

Qual è l'output previsto per il seguente pezzo di codice?

```
print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))
print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))
```

Soluzione

True False

False True

### Esercizio 3

Qual è l'output previsto per il seguente pezzo di codice?

```
print(luna is luna, rocky is luna)
print(rocky.kennel)
```

Soluzione

True False

2

### Esercizio 4

Definire una sottoclasse di SheepDog chiamata LowlandDog e dotarla di un metodo `__str__()` che sovrascriva un metodo ereditato con lo stesso nome. Il metodo `__str__()` del nuovo dog dovrebbe restituire la stringa "Woof! I don't like mountains!" .

Soluzione

```
class LowlandDog(SheepDog):
    def __str__(self):
        return Dog.__str__(self) + " I don't like mountains!"
```

## Ulteriori informazioni sulle eccezioni

Discutere di programmazione a oggetti offre un'ottima occasione per tornare a parlare di eccezioni. La natura orientata agli oggetti delle eccezioni di Python le rende uno strumento molto flessibile, in grado di adattarsi a esigenze specifiche, anche quelle che non conoscete ancora.

Prima di immergerci nell'**aspetto oggettivo delle eccezioni**, vogliamo mostrarvi alcuni aspetti sintattici e semantici di come Python tratta il blocco *try-except*, poiché offre qualcosa in più rispetto a quanto presentato finora.

La prima caratteristica che vogliamo discutere è un'ulteriore, possibile diramazione che può essere collocata all'interno (o meglio, direttamente dietro) il blocco *try-except* - è la parte di codice che inizia con *else* - proprio come nell'esempio

```
def reciprocal(n):  
    try:  
        n = 1 / n  
    except ZeroDivisionError:  
        print("Division failed")  
        return None  
    else:  
        print("Everything went fine")  
        return n  
  
print(reciprocal(2))  
print(reciprocal(0))
```

Un codice così etichettato viene eseguito quando (e solo quando) non è stata sollevata alcuna eccezione all'interno della parte try:. Possiamo dire che dopo try può essere eseguito esattamente un ramo: - o quello che inizia con except (non dimenticate che ci può essere più di un ramo di questo tipo) o quello che inizia con else.

Nota: il ramo else: deve essere collocato dopo l'ultimo ramo except.

Il codice di esempio produce il seguente risultato:

Everything went fine

0.5

Division failed

None



### **Ulteriori informazioni sulle eccezioni**

Il blocco try-except può essere esteso in un altro modo, aggiungendo una parte guidata dalla parola chiave finally (deve essere l'ultimo ramo del codice progettato per gestire le eccezioni).

Nota: queste due varianti (else e finally) non dipendono in alcun modo e possono coesistere o verificarsi indipendentemente.

Il blocco finally viene sempre eseguito (finalizza l'esecuzione del blocco try-except, da cui il nome), indipendentemente da ciò che è accaduto in precedenza, anche quando viene sollevata un'eccezione, indipendentemente dal fatto che questa sia stata gestita o meno. Guardate il codice

```
def reciprocal(n):  
    try:  
        n = 1 / n  
    except ZeroDivisionError:  
        print("Division failed")  
        n = None  
    else:  
        print("Everything went fine")  
    finally:  
        print("It's time to say goodbye")  
        return n  
  
print(reciprocal(2))  
print(reciprocal(0))
```

L'outputs:

Everything went fine

It's time to say good bye

0.5

Division failed

It's time to say good bye

None

## Le eccezioni sono classi

Tutti gli esempi precedenti si accontentavano di rilevare un tipo specifico di eccezione e di rispondere in modo appropriato. Ora ci addentreremo più a fondo e guarderemo all'interno dell'eccezione stessa.

Probabilmente non vi sorprenderà sapere che le **eccezioni sono classi**. Inoltre, quando viene sollevata un'eccezione, viene istanziato un oggetto della classe, che attraversa tutti i livelli di esecuzione del programma, alla ricerca del ramo `except` pronto ad affrontarla.

Un oggetto di questo tipo porta con sé alcune informazioni utili che possono aiutare a identificare con precisione tutti gli aspetti della situazione in sospeso. Per raggiungere questo obiettivo, Python offre una variante speciale della clausola di eccezione,

```
try:
    i = int("Hello!")
except Exception as e:
    print(e)
    print(e.__str__())
```

Come si può vedere, l'istruzione `except` è estesa e contiene una frase aggiuntiva che inizia con la parola chiave `as`, seguita da un identificatore. L'identificatore ha lo scopo di catturare l'oggetto dell'eccezione, in modo da poterne analizzare la natura e trarre le dovute conclusioni.

Nota: l'ambito dell'identificatore copre il suo ramo `except` e non va oltre. L'esempio presenta un modo molto semplice di utilizzare l'oggetto ricevuto: basta stamparlo (come si può vedere, l'output è prodotto dal metodo `__str__()` dell'oggetto) e contiene un breve messaggio che descrive il motivo. Lo stesso messaggio viene stampato se nel codice non c'è un blocco `"fitting except"` e Python è costretto a gestirlo da solo.

## Le eccezioni sono classi

Tutte le eccezioni integrate in Python formano una gerarchia di classi. Non c'è alcun ostacolo a estenderla, se lo si ritiene ragionevole. Guardate il codice

```
def print_exception_tree(thisclass, nest = 0):
    if nest > 1:
        print("  |" * (nest - 1), end="")
    if nest > 0:
        print(" +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        print_exception_tree(subclass, nest + 1)

print_exception_tree(BaseException)
```

Questo programma visualizza tutte le classi di eccezioni predefinite sotto forma di una stampa ad albero.

Poiché **un albero è un perfetto esempio di struttura di dati ricorsiva**, una ricorsione sembra essere lo strumento migliore per attraversarlo. La funzione `print_exception_tree()` prende due argomenti:

- un punto all'interno dell'albero da cui iniziare a percorrere l'albero;
- un livello di annidamento (lo utilizzeremo per costruire un disegno semplificato dei rami dell'albero)

Partiamo dalla radice dell'albero: la radice delle classi di eccezioni di Python è la classe `BaseException` (è una superclasse di tutte le altre eccezioni).

Per ciascuna delle classi incontrate, eseguire la stessa serie di operazioni:

- stampare il suo nome, preso dalla proprietà `__name__`;
- iterano attraverso l'elenco di sottoclassi fornito dal metodo `__subclasses__()` e invocano ricorsivamente la funzione `print_exception_tree()`, incrementando rispettivamente il livello di annidamento.

Si noti come sono stati disegnati i rami e le biforcazioni. La stampa non è ordinata in alcun modo: si può provare a ordinarla da soli, se si vuole una sfida. Inoltre, ci sono alcune sottili imprecisioni nel modo in cui sono presentati alcuni rami. Anche questo può essere corretto, se lo si desidera.

Ecco come si presenta:

## BaseException

- +--Eccezione

- +---TypeError

- +---StopAsyncIteration

- +---StopIteration

- +---ImportError

- +---ModuleNotFoundError

- +---ZipImportError

- Errore

- +---Errore di connessione

- +---BrokenPipeError

- +---ConnectionAbortedError

- +---ConnectionRefusedError

- +---ConnectionResetError

- +---BlockingIOError

- +---Errore di processo del bambino

- +---FileExistsError

- +---FileNotFoundError

- +---IsADirectoryError

- +---NotADirectoryError

- +---Errore interrotto

- +---PermissionError

- +---ProcessLookupError

- | | +---TimeoutError
- | | +---Operazione non supportata
- | | +---errore
- | | +---gaierror
- | | +---timeout
- | | +---Errore
- | | | +---SameFileError
- | | +---SpecialFileError
- | | +---ExecError
- | | +---Errore di lettura
- | Errore
- +---RuntimeError
- | | +---Errore di ricorsione
- | | +---NotImplementedError
- | | +---\_DeadlockError
- | | +---BrokenBarrierError
- | +---NameError
- | | +---UnboundLocalError

- | +---AttributeError
- | +---Errore di sintassi
  - | | +---Errore di indentazione
    - | | | +---TabError
- | +---LookupError
  - | | +---Errore di indice
  - | | +---KeyError
  - | | +---CodecRegistryError
- | ValoreErrore
  - | | +---UnicodeError
    - | | | +---UnicodeEncodeError
    - | | | +---UnicodeDecodeError
    - | | | +---UnicodeTranslateError
  - | | +---Operazione non supportata
- +---Errore di inserimento
  - | Errore aritmetico
    - | | +---FloatingPointError
    - | | +---OverflowError
    - | | +---ZeroDivisionError
- +---Errore di sistema
  - | | +---CodecRegistryError
- | +---Errore di riferimento



- | +---BufferError
  - | +---MemoryError
  - | Avviso
    - | | +---Avvertimento utente
    - | | +---Avviso di deprecazione
    - | | +---Avviso di deprecazione in sospeso
    - | | +---SyntaxWarning
    - | | +---RuntimeWarning
    - | | +---FutureWarning
    - | | +---ImportWarning
    - | | +---UnicodeWarning
    - | | +---BytesWarning
    - | | +---Avviso risorse
  - | Errore
- +---Verbose
  - | Errore
- +---TokenError
  - | Smettere di tokenizzare
- | Vuoto
- | Pieno
- | +---\_OptionError

+---TclError

| +---SubprocessError

| | +---CalledProcessError

| | +---TimeoutExpired

| Errore

| | +---NoSectionError

| | +---Errore di sezione duplicata

| | +---Errore di duplicazione dell'opzione

| | +---NoOptionError

| | +---InterpolazioneErrore

| | | +---InterpolazioneMancanteOpzioneErrore

| | | +---InterpolazioneSyntaxError

| | | +---InterpolazioneProfonditàErrore

| | +---ParsingError

| | | +---MissingSectionHeaderError

| +---InvalidConfigType

| +---InvalidConfigSet

| +---InvalidFgBg

| +---InvalidTheme

| +---FineBlocco

- +---BdbQuit
  - | Errore
  - | +---\_Stop
  - | +---PickleError
    - | | +---PicklingError
    - | | +---UnpicklingError
  - | +---\_GiveupOnSendfile
  - | Errore
  - | +---LZMAError
- +---Errore di registro
  - | +---ErroreDuranteImportazione
- +--Esci dal generatore
- +--Esci di sistema
- +--Interruzione della tastiera

## Anatomia dettagliata delle eccezioni

Diamo un'occhiata più da vicino all'oggetto dell'eccezione, perché ci sono alcuni elementi davvero interessanti (torneremo presto sull'argomento quando considereremo le tecniche base di input/output di Python, dato che il loro sottosistema di eccezioni estende un po' questi oggetti).

La classe `BaseException` introduce una proprietà denominata `args`. Si tratta di una **tupla che raccoglie tutti gli argomenti passati al costruttore della classe**. È vuota se il costrutto è stato invocato senza argomenti, oppure contiene un solo elemento quando il costruttore riceve un argomento (non contiamo l'argomento `self`), e così via. Abbiamo preparato una semplice funzione per stampare la proprietà `args` in modo elegante. È possibile vedere la funzione

```
def print_args(args):  
    lng = len(args)  
    if lng == 0:  
        print("")  
    elif lng == 1:  
        print(args[0])  
    else:  
        print(str(args))
```

```
try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
```

```
try:
    raise Exception("my exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
```

```
try:
    raise Exception("my", "exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
```

Abbiamo usato la funzione per stampare il contenuto della proprietà `args` in tre casi diversi, in cui l'eccezione della classe `Exception` viene sollevata in tre modi diversi. Per rendere il tutto più spettacolare, abbiamo anche stampato l'oggetto stesso, insieme al risultato dell'invocazione `__str__()`.

Il primo caso sembra di routine: c'è solo il nome `Exception` dopo la parola chiave `raise`. Ciò significa che l'oggetto di questa classe è stato creato in modo molto routinario.

Il secondo e il terzo caso possono sembrare un po' strani a prima vista, ma non c'è nulla di strano: si tratta solo di invocazioni del costruttore. Nella seconda istruzione `raise`, il costruttore viene invocato con un argomento, mentre nella terza con due.

Come si può vedere, l'output del programma riflette questa scelta, mostrando il contenuto appropriato della proprietà `args`:

: :

my exception : my exception : my exception

('my', 'exception') : ('my', 'exception') : ('my', 'exception')

## **Come creare la propria eccezione**

La gerarchia delle eccezioni non è né chiusa né finita e si può sempre estenderla se si vuole o si ha bisogno di creare un proprio mondo popolato di eccezioni.

Può essere utile quando si crea un modulo complesso che rileva errori e solleva eccezioni e si vuole che le eccezioni siano facilmente distinguibili da tutte le altre portate da Python.

Ciò avviene **definendo le proprie nuove eccezioni come sottoclassi derivate da quelle predefinite.**

Nota: se si vuole creare un'eccezione che sarà utilizzata come caso specializzato di qualsiasi eccezione incorporata, la si derivi proprio da questa. Se si vuole costruire una propria gerarchia e non si vuole che sia strettamente collegata all'albero delle eccezioni di Python, la si può derivare da una qualsiasi delle classi di eccezioni principali, come Exception.

Immaginate di aver creato una nuova aritmetica, regolata da leggi e teoremi propri. È chiaro che anche la divisione è stata ridefinita e deve comportarsi in modo diverso dalla divisione di routine. È anche chiaro che questa nuova divisione dovrebbe sollevare una propria eccezione, diversa da ZeroDivisionError, ma è ragionevole supporre che in alcune circostanze, voi (o l'utente dell'aritmetica) vogliate trattare tutte le divisioni a zero nello stesso modo.

Richieste come queste possono essere soddisfatte nel modo presentato di seguito:

```
class MyZeroDivisionError(ZeroDivisionError):
```

```
    pass
```

```
def do_the_division(mine):
```

```
    if mine:
```

```
        raise MyZeroDivisionError("some worse news")
```

```
    else:
```

```
        raise ZeroDivisionError("some bad news")
```

```
for mode in [False, True]:
```

```
    try:
```

```
        do_the_division(mode)
```

```
    except ZeroDivisionError:
```

```
        print('Division by zero')
```

```
for mode in [False, True]:
```

```
    try:
```

```
        do_the_division(mode)
```

```
    except MyZeroDivisionError:
```

```
        print('My division by zero')
```

```
    except ZeroDivisionError:
```

```
        print('Original division by zero')
```



Guardiamo il codice e analizziamolo:

- Abbiamo definito la nostra eccezione, chiamata `MyZeroDivisionError`, derivata dall'eccezione integrata `ZeroDivisionError`. Come si può notare, abbiamo deciso di non aggiungere alcun nuovo componente alla classe.

In effetti, un'eccezione di questa classe può essere trattata - a seconda del punto di vista desiderato - come un `simpleZeroDivisionError`, oppure considerata separatamente.

- La funzione `do_the_division()` solleva un'eccezione `MyZeroDivisionError` o `ZeroDivisionError`, a seconda del valore dell'argomento.

La funzione viene invocata quattro volte in totale, mentre le prime due invocazioni vengono gestite utilizzando un sol ramo escluso (quello più generale) e le ultime due con due rami diversi, in grado di distinguere le eccezioni (non dimenticate: l'ordine dei rami fa una differenza fondamentale).

### Come creare la propria eccezione: continua

Quando si costruisce un universo completamente nuovo, pieno di creature completamente nuove che non hanno nulla in comune con tutte le cose già conosciute, è meglio **costruire la propria struttura di eccezione**.

Ad esempio, se si lavora a un sistema di simulazione di grandi dimensioni destinato a modellare le attività di una pizzeria, può essere auspicabile formare una gerarchia separata di eccezioni.

Si può iniziare a costruirla **definendo un'eccezione generale come una nuova classe base** per qualsiasi altra eccezione specializzata. Lo abbiamo fatto nel modo seguente:

```
class PizzaError(Exception):  
    def __init__(self, pizza, message):  
        Exception.__init__(self, message)  
        self.pizza = pizza
```

Nota: qui raccoglieremo informazioni più specifiche di quanto non faccia una normale Exception, quindi il nostro costruttore prenderà due argomenti:

- uno che specifica una pizza come soggetto del processo,
- e una contenente una descrizione più o meno precisa del problema.

Come si può vedere, passiamo il secondo parametro al costruttore della superclasse e salviamo il primo all'interno della nostra proprietà.

Un problema più specifico (come un eccesso di formaggio) può richiedere un'eccezione più specifica. È possibile derivare la nuova classe dalla classe PizzaError già definita, come abbiamo fatto qui:

```
class TooMuchCheeseError(PizzaError):  
    def __init__(self, pizza, cheese, message):  
        PizzaError.__init__(self, pizza, message)  
        self.cheese = cheese
```

L'eccezione TooMuchCheeseError ha bisogno di più informazioni rispetto alla normale eccezione PizzaError, quindi la aggiungiamo al costruttore: il nome formaggio viene memorizzato per un'ulteriore elaborazione.

### **Come creare la propria eccezione: continua**

Guardate il codice nell'editor. Abbiamo unito le due eccezioni definite in precedenza e le abbiamo fatte funzionare in un piccolo frammento di esempio.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")
```

```
for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:  
    try:  
        make_pizza(pz, ch)  
    except TooMuchCheeseError as tmce:  
        print(tmce, ': ', tmce.cheese)  
    except PizzaError as pe:  
        print(pe, ': ', pe.pizza)
```

Una di queste viene sollevata all'interno della funzione `make_pizza()` quando viene rilevata una di queste due situazioni errate: una richiesta di pizza sbagliata o una richiesta di troppo formaggio.

Nota:

- rimuovendo il ramo che inizia con `except TooMuchCheeseError`, tutte le eccezioni che compaiono saranno classificate come `PizzaError`;
- rimuovendo il ramo che inizia con `except PizzaError`, le eccezioni `TooMuchCheeseError` rimarranno non gestite e il programma terminerà.

La soluzione precedente, sebbene elegante ed efficiente, presenta un importante punto debole. A causa del modo un po' semplicistico di dichiarare i costruttori, le nuove eccezioni non possono essere utilizzate così come sono, senza un elenco completo degli argomenti richiesti.

Elimineremo questa debolezza **impostando i valori predefiniti per tutti i parametri del costruttore**. Date un'occhiata:

```
class PizzaError(Exception):
    def __init__(self, pizza='unknown', message=""):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='unknown', cheese='>100', message=""):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")
```

```
for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:  
    try:  
        make_pizza(pz, ch)  
    except TooMuchCheeseError as tmce:  
        print(tmce, ': ', tmce.cheese)  
    except PizzaError as pe:  
        print(pe, ': ', pe.pizza)
```

Ora, se le circostanze lo permettono, è possibile utilizzare solo i nomi delle classi.

## Punti di forza

1. Il ramo else: dell'istruzione try viene eseguito quando non si verifica alcuna eccezione durante l'esecuzione del blocco try:.
2. Il ramo finally: dell'istruzione try viene **sempre** eseguito.
3. La sintassi except *Exception\_Name* as an *exception\_object*: consente di intercettare un oggetto che contiene informazioni su un'eccezione in sospeso. La proprietà dell'oggetto denominata args (una tupla) memorizza tutti gli argomenti passati al costruttore dell'oggetto.



4. Le classi di eccezioni possono essere estese per arricchirle di nuove funzionalità o per adottare i loro tratti a nuove eccezioni definite.

Ad esempio:

```
try:
    assert __name__ == "__main__"
except:
    print("fail", end=' ')
else:
    print("success", end=' ')
finally:
    print("done")
```

Il codice produce: success done.

### **Esercizio 1**

Qual è l'output previsto del seguente codice?

```
import math
try:
    print(math.sqrt(9))
except ValueError:
    print("inf")
else:
    print("fine")
```

## Esercizio 2

Qual è l'output previsto del seguente codice?

```
import math
try:
    print(math.sqrt(-9))
except ValueError:
    print("inf")
else:
    print("fine")
finally:
    print("the end")
```

ì

### Esercizio 3

Qual è l'output previsto del seguente codice?

```
import math
class NewValueError(ValueError):
    def __init__(self, name, color, state):
        self.data = (name, color, state)

try:
    raise NewValueError("Enemy warning", "Red alert", "High readiness")
except NewValueError as nve:
    for arg in nve.args:
        print(arg, end='! ')
```

### **Esercizio 1**

Qual è l'output previsto del seguente codice?

```
import math
try:
    print(math.sqrt(9))
except ValueError:
    print("inf")
else:
    print("fine")
```

Soluzione

3.0

fine

## Esercizio 2

Qual è l'output previsto del seguente codice?

```
import math
try:
    print(math.sqrt(-9))
except ValueError:
    print("inf")
else:
    print("fine")
finally:
    print("the end")
```

Soluzione

inf

the end

### Esercizio 3

Qual è l'output previsto del seguente codice?

```
import math
class NewValueError(ValueError):
    def __init__(self, name, color, state):
        self.data = (name, color, state)

try:
    raise NewValueError("Enemy warning", "Red alert", "High readiness")
except NewValueError as nve:
    for arg in nve.args:
        print(arg, end='! ')
```

Soluzione

Enemy warning! Red alert! High readiness!