



Corso di Apprendistato Python

Mod. Python
Docente:
Tonino Petrulli



I concetti di base dell'approccio orientato agli oggetti

Facciamo un passo al di fuori della programmazione informatica e dei computer in generale e parliamo di problemi di programmazione a oggetti.

Quasi tutti i programmi e le tecniche utilizzate finora rientrano nello stile di programmazione procedurale. È vero che avete fatto uso di alcuni oggetti incorporati, ma quando ci riferiamo ad essi, ci limitiamo a citare il minimo indispensabile.

Lo stile di programmazione procedurale è stato l'approccio dominante allo sviluppo del software per decenni di informatica ed è ancora in uso oggi. Inoltre, non è destinato a scomparire in futuro, poiché funziona molto bene per determinati tipi di progetti (in genere, non quelli molto complessi e non quelli di grandi dimensioni, ma ci sono molte eccezioni a questa regola).

L'approccio a oggetti è piuttosto giovane (molto più giovane dell'approccio procedurale) ed è particolarmente utile se applicato a progetti grandi e complessi realizzati da grandi team composti da molti sviluppatori.

Questo tipo di comprensione della struttura di un progetto facilita molti compiti importanti, ad esempio la suddivisione del progetto in parti piccole e indipendenti e lo sviluppo indipendente di diversi elementi del progetto.

Python è uno strumento universale per la programmazione a oggetti e procedurale.

Può essere utilizzato con successo in entrambi gli ambiti.

Inoltre, è possibile creare molte applicazioni utili, anche se non si conosce nulla di classi e oggetti, ma bisogna tenere presente che alcuni problemi (ad esempio, la gestione dell'interfaccia grafica) possono richiedere un approccio rigoroso agli oggetti.

Fortunatamente, la programmazione a oggetti è relativamente semplice.



Approccio procedurale e approccio orientato agli oggetti

Nell'**approccio procedurale**, è possibile distinguere due mondi diversi e completamente separati: **il mondo dei dati e il mondo del codice**. Il mondo dei dati è popolato da variabili di vario tipo, mentre il mondo del codice è abitato da codice raggruppato in moduli e funzioni.

Le funzioni possono utilizzare i dati, ma non viceversa. Inoltre, le funzioni sono in grado di abusare dei dati, cioè di utilizzare il valore in modo non autorizzato (ad esempio, quando la funzione sine riceve come parametro il saldo di un conto bancario).

In passato abbiamo detto che i dati non possono utilizzare le funzioni. Ma è del tutto vero? Esistono alcuni tipi speciali di dati che possono utilizzare le funzioni?

Sì, ci sono: quelli denominati metodi. Si tratta di funzioni che vengono invocate dall'interno dei dati, non accanto ad essi. Se riuscite a capire questa distinzione, avete fatto il primo passo verso la programmazione a oggetti.

L'**approccio a oggetti** suggerisce un modo di pensare completamente diverso. I dati e il codice sono racchiusi nello stesso mondo, diviso in classi.

Ogni **classe è come una ricetta che può essere utilizzata quando si vuole creare un oggetto utile** (da qui il nome dell'approccio). Si possono produrre tutti gli oggetti necessari per risolvere il problema.

Ogni oggetto ha un insieme di caratteristiche (chiamate proprietà o attributi - useremo entrambi i termini come sinonimi) ed è in grado di eseguire un insieme di attività (chiamate metodi).

Le ricette possono essere modificate se sono inadeguate per scopi specifici e, di fatto, si possono creare nuove classi. Queste nuove classi ereditano proprietà e metodi dalle originali e di solito ne aggiungono di nuove, creando nuovi strumenti più specifici.

Gli oggetti sono incarnazioni di idee espresse nelle classi, come il cheesecake sul piatto è un'incarnazione dell'idea espressa in una ricetta stampata in un vecchio libro di cucina.

Gli oggetti interagiscono tra loro, scambiando dati o attivando i loro metodi. Una classe (e quindi i suoi oggetti) costruita correttamente è in grado di proteggere i dati sensibili e di nasconderli da modifiche non autorizzate.

Non esiste un confine netto tra dati e codice: essi vivono come un tutt'uno negli oggetti.

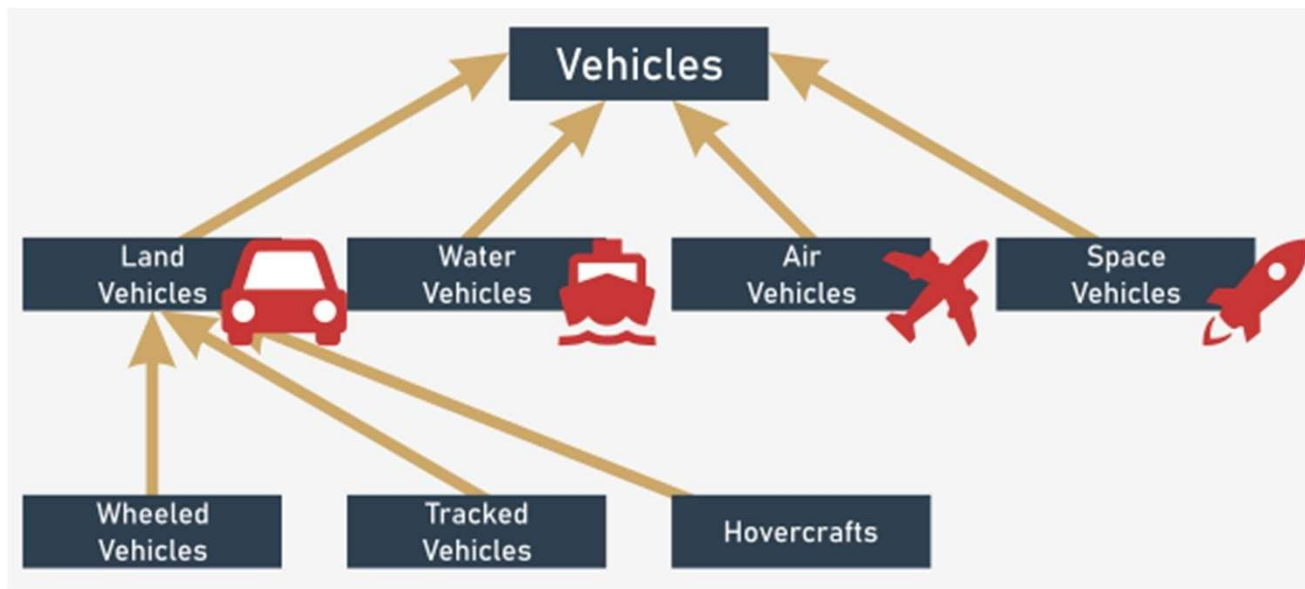
Tutti questi concetti non sono astratti come si potrebbe sospettare a prima vista. Al contrario, sono tutti tratti da esperienze reali e quindi sono estremamente utili nella programmazione informatica: non creano vita artificiale, ma **riflettono fatti, relazioni e circostanze reali**.



Gerarchie di classi

La parola *classe* ha molti significati, ma non tutti sono compatibili con le idee che vogliamo discutere qui. La *classe* di cui ci occupiamo è come una *categoria*, come risultato di somiglianze precisamente definite.

Cercheremo di evidenziare alcune classi che sono un buon esempio di questo concetto.



Consideriamo per un momento i veicoli. Tutti i veicoli esistenti (e quelli che ancora non esistono) sono **accomunati da un'unica, importante caratteristica**: la capacità di muoversi. Si potrebbe obiettare che anche un cane si muove; un cane è un veicolo? No, non lo è. Dobbiamo migliorare la definizione, cioè arricchirla con altri criteri, distinguendo i veicoli dagli altri esseri e creando un legame più forte. Prendiamo in considerazione le seguenti circostanze: i veicoli sono entità create artificialmente e utilizzate per il trasporto, mosse da forze della natura e dirette (guidate) dall'uomo.

In base a questa definizione, un cane non è un veicolo.

La classe dei *veicoli* è molto ampia. Troppo ampia. Dobbiamo quindi definire alcune **classi** più **specializzate**. Le classi specializzate sono le **sottoclassi**. La classe *veicoli* sarà una **superclasse** per tutte.

Nota: **la gerarchia cresce dall'alto verso il basso, come le radici di un albero, non i rami**. La classe più generale e più ampia si trova sempre in cima (la superclasse), mentre i suoi discendenti si trovano in basso (le sottoclassi). A questo punto, probabilmente si possono indicare alcune potenziali sottoclassi della superclasse *Veicoli*. Le classificazioni possibili sono molte. Abbiamo scelto le sottoclassi in base all'ambiente e diciamo che ci sono (almeno) quattro sottoclassi:

- veicoli terrestri;
- veicoli acquatici;
- veicoli aerei;
- veicoli spaziali.

In questo esempio, discuteremo solo la prima sottoclasse, quella dei veicoli terrestri. Se lo si desidera, è possibile continuare con le altre classi.

I veicoli terrestri possono essere ulteriormente suddivisi, a seconda del metodo con cui impattano sul terreno.

Possiamo quindi elencare:

- veicoli su ruote;
- veicoli cingolati;
- Hovercraft.

La gerarchia creata è illustrata nella figura.

Si noti la direzione delle frecce, che puntano sempre alla superclasse. La classe di primo livello è un'eccezione: non ha una propria superclasse.

Gerarchie di classi: continua

Un altro esempio è la gerarchia del regno tassonomico degli animali.

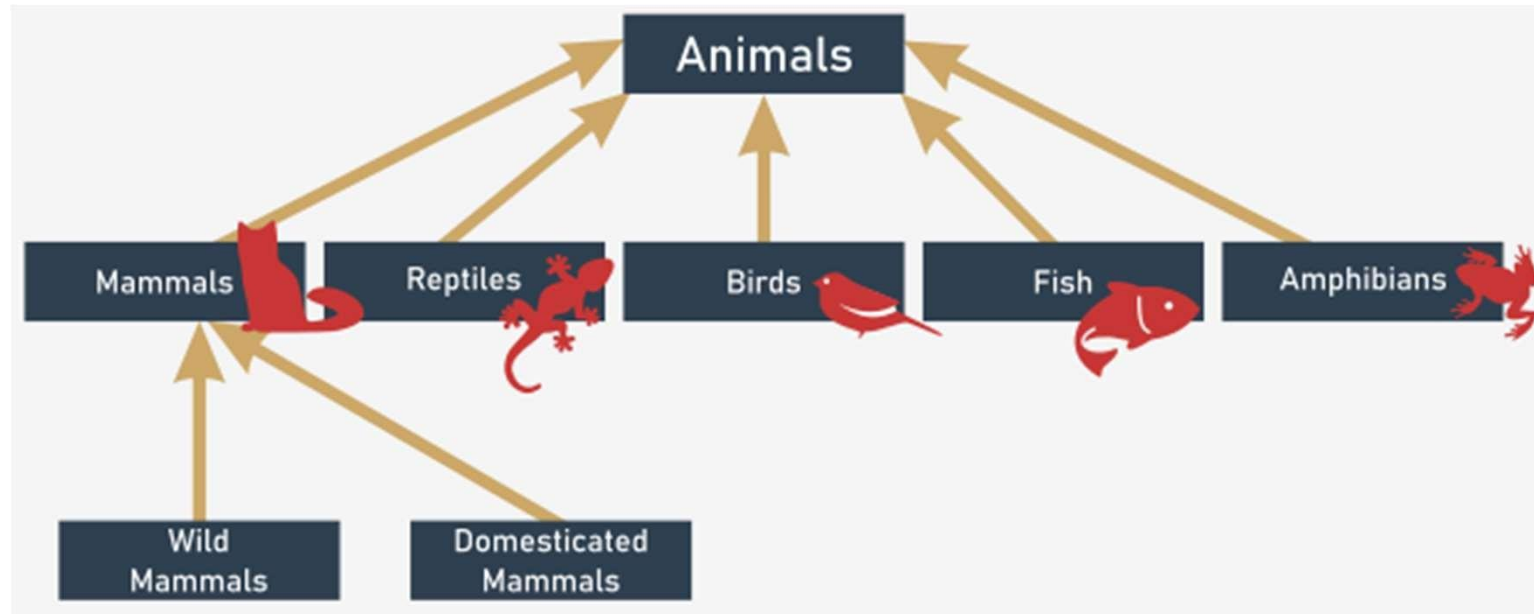
Possiamo dire che tutti gli *animali* (la nostra classe di primo livello) possono essere suddivisi in cinque sottoclassi:

- mammiferi;
- rettili;
- uccelli;
- pesce;
- anfibi.

Prendiamo il primo per un'ulteriore analisi.

Abbiamo identificato le seguenti sottoclassi:

- mammiferi selvatici;
- mammiferi addomesticati.



Provate a estendere la gerarchia come volete e trovate il posto giusto per gli esseri umani.

Che cos'è un oggetto?

Una classe (tra le altre definizioni) è un **insieme di oggetti**. Un oggetto è **un essere appartenente a una classe**.

Un oggetto è **un'incarnazione dei requisiti, dei tratti e delle qualità assegnati a una classe specifica**. Può

sembrare semplice, ma è importante notare le seguenti circostanze. Le classi formano una gerarchia.

Ciò può significare che un oggetto appartenente a una classe specifica appartiene a tutte le superclassi allo stesso tempo. Può anche significare che un oggetto appartenente a una superclasse non può appartenere a nessuna delle sue sottoclassi.

Per esempio: qualsiasi auto personale è un oggetto appartenente alla classe *veicoli su ruote*. Ciò significa anche che la stessa auto appartiene a tutte le superclassi della sua classe di origine; quindi, è anche un membro della classe *veicoli*.

Il cane (o il gatto) è un oggetto incluso nella classe *mammiferi domestici*, il che significa esplicitamente che è incluso anche nella classe *animali*.

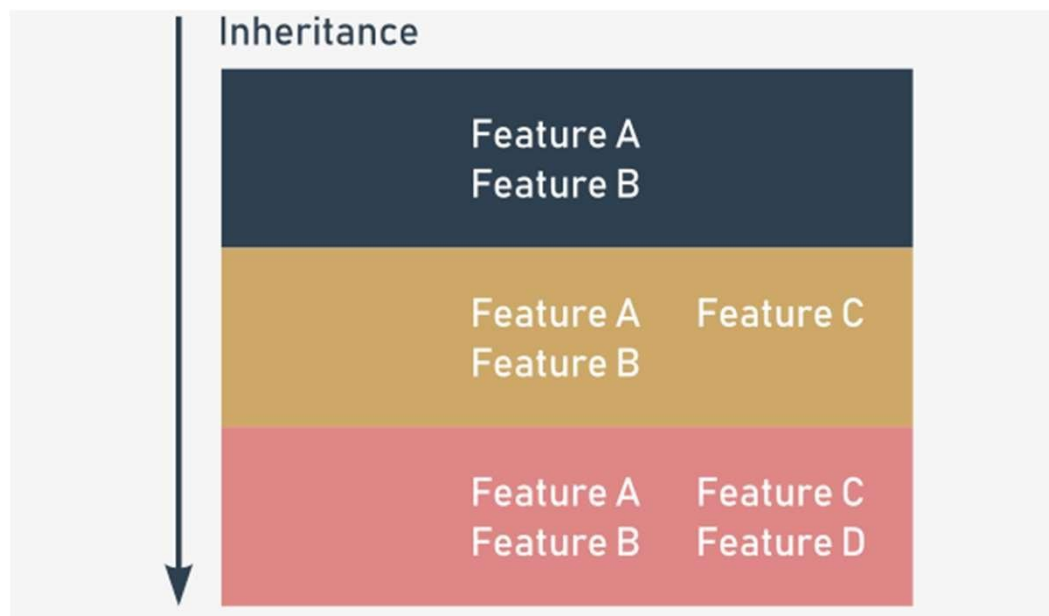
Ogni **sottoclasse è più specializzata** (o più specifica) della sua superclasse. Viceversa, ogni **superclasse è più generale** (più astratta) di qualsiasi sua sottoclasse.

Si noti che abbiamo dato per scontato che una classe possa avere una sola superclasse: questo non è sempre vero, ma ne parleremo più avanti.

Eredità

Definiamo uno dei concetti fondamentali della programmazione a oggetti, l'**ereditarietà**. Qualsiasi oggetto legato a un livello specifico della gerarchia delle classi **eredita tutti i tratti (nonché i requisiti e le qualità) definiti all'interno di qualsiasi superclasse**.

La classe di origine dell'oggetto può definire nuovi tratti (oltre a requisiti e qualità) che saranno ereditati da tutte le sue sottoclassi.



Non dovrete avere problemi ad associare questa regola a esempi specifici, sia che si applichi agli animali, sia che si applichi ai veicoli.

Che cosa ha un oggetto?

La convenzione di programmazione a oggetti presuppone che **ogni oggetto esistente possa essere dotato di tre gruppi di attributi**:

- un oggetto ha un **nome** che lo identifica in modo univoco all'interno del suo spazio dei nomi (sebbene possano esistere anche oggetti anonimi)
- un oggetto ha un **insieme di proprietà individuali** che lo rendono originale, unico o eccezionale (anche se è possibile che alcuni oggetti non abbiano alcuna proprietà)
- un oggetto ha un **insieme di abilità per svolgere attività specifiche**, in grado di modificare l'oggetto stesso o alcuni degli altri oggetti.

Esiste un suggerimento (anche se non funziona sempre) che può aiutarvi a identificare una qualsiasi delle tre sfere di cui sopra. Ogni volta che si descrive un oggetto e si usa:

- un sostantivo - probabilmente si definisce il nome dell'oggetto;
- un aggettivo - probabilmente si definisce la proprietà dell'oggetto;
- un verbo - probabilmente si definisce l'attività dell'oggetto.

Due frasi esemplificative dovrebbero servire da buon esempio:

- Una Cadillac rosa è andata velocemente.
Nome oggetto = Cadillac
Classe di appartenenza = Veicoli a ruote
Proprietà = Colore (rosa)
Attività = Andare (velocemente)
- Rudolph è un gatto di grandi dimensioni che dorme tutto il giorno.
Nome oggetto = Rudolph
Classe di appartenenza = Gatto
Proprietà = Taglia (grande)
Attività = Dormire (tutto il giorno)

**Rudolph is a large cat
who sleeps all day** → **Verb** → **Sleep (all day)**

Object



La vostra prima lezione

La programmazione a oggetti è **l'arte di definire ed espandere le classi**. Una classe è un modello di una parte molto specifica della realtà, che riflette proprietà e attività presenti nel mondo reale.

Le classi definite all'inizio sono troppo generali e imprecise per coprire il maggior numero possibile di casi reali. Non c'è alcun ostacolo alla definizione di nuove sottoclassi più precise. Esse ereditano tutto dalla loro superclasse, per cui il lavoro svolto per la sua creazione non andrà sprecato.

La nuova classe può aggiungere nuove proprietà e nuove attività e quindi può essere più utile in applicazioni specifiche. Ovviamente, può essere utilizzata come superclasse per un numero qualsiasi di sottoclassi appena create.

Il processo non deve necessariamente avere una fine. Si possono creare tutte le classi necessarie.

La classe definita non ha nulla a che fare con l'oggetto: **l'esistenza di una classe non implica la creazione automatica di alcun oggetto compatibile**. La classe stessa non è in grado di creare un oggetto: dovete crearlo voi stessi e Python vi permette di farlo.

È il momento di definire la classe più semplice e di creare un oggetto. Guardate l'esempio qui sotto:

```
classe TheSimplestClass:  
    passaggio
```

Abbiamo definito una classe. La classe è piuttosto povera: non ha né proprietà né attività. In realtà è **vuota**, ma questo non ha importanza per ora. Più semplice è la classe, meglio è per i nostri scopi.

La definizione inizia con la parola chiave class. La parola chiave è seguita da un **identificatore che darà il nome alla classe** (attenzione: non bisogna confonderlo con il nome dell'oggetto, sono due cose diverse).

Successivamente, si aggiungono i **due punti** (:), poiché le classi, come le funzioni, formano il proprio blocco annidato. Il contenuto del blocco definisce tutte le proprietà e le attività della classe.

La parola chiave pass riempie la classe di nulla. Non contiene metodi o proprietà.

Il primo oggetto

La nuova classe definita diventa uno strumento in grado di creare nuovi oggetti. Lo strumento deve essere usato esplicitamente, su richiesta.

Si immagini di voler creare un oggetto (esattamente uno) della classe TheSimplestClass.

Per fare ciò, è necessario assegnare una variabile per memorizzare l'oggetto appena creato di quella classe e creare un oggetto allo stesso tempo.

Si procede nel modo seguente:

```
mio_primo_oggetto = TheSimplestClass()
```

Nota:

- il nome della classe cerca di far credere che sia una funzione - riuscite a vederlo? Ne parleremo presto;
- l'oggetto appena creato è dotato di tutto ciò che la classe porta con sé; poiché questa classe è completamente vuota, anche l'oggetto è vuoto.

L'atto di creare un oggetto della classe selezionata è chiamato anche **istanziamento** (poiché l'oggetto diventa un'**istanza della classe**).

Lasciamo da parte le classi per un breve momento, perché ora vi diremo qualche parola sugli *stack*. Sappiamo che il concetto di classi e oggetti potrebbe non essere ancora del tutto chiaro. Non preoccupatevi, vi spiegheremo tutto molto presto.

Punti di forza

1. Una **classe** è un'idea (più o meno astratta) che può essere usata per creare un certo numero di incarnazioni - una tale incarnazione è chiamata **oggetto**.

2. Quando una classe deriva da un'altra classe, la loro relazione prende il nome di **ereditarietà**. La classe che deriva da un'altra classe prende il nome di **sottoclasse**. Il secondo lato di questa relazione è chiamato **superclasse**.

Un modo per presentare tale relazione è un **diagramma di ereditarietà**, in cui:

Le superclassi sono sempre presentate **sopra le** loro sottoclassi;

Le relazioni tra le classi sono indicate come frecce dirette **dalla sottoclasse verso la sua superclasse**.

3. Gli oggetti sono dotati di:

- un **nome** che li identifichi e ci permetta di distinguerli;
- un insieme di **proprietà** (l'insieme può essere vuoto)
- un insieme di **metodi** (può essere anche vuoto)

4. Per definire una classe Python, è necessario utilizzare la parola chiave `class`. Ad esempio:

```
class This_Is_A_Class:
```

```
    istruzioni
```

5. Per creare un oggetto della classe precedentemente definita, è necessario utilizzare la classe come se fosse una funzione. Ad esempio:

```
this_is_an_object = This_Is_A_Class()
```

Esercizio 1

Se ipotizziamo che pitoni, vipere e cobra siano sottoclassi della stessa superclasse, come la chiamereste?

Esercizio 2

Provate a nominare alcune sottoclassi di classi pitone.

Esercizio 3

Potete chiamare una delle vostre classi solo "classe"?

Esercizio 1

Se ipotizziamo che pitoni, vipere e cobra siano sottoclassi della stessa superclasse, come la chiamereste?

Soluzione

Serpente, rettile, vertebrato, animale: tutte queste risposte sono accettabili.

Esercizio 2

Provate a nominare alcune sottoclassi di classi pitoni.

Soluzione

Pitone indiano, pitone delle rocce africano, pitone palla, pitone birmano: l'elenco è lungo.

Esercizio 3

Potete chiamare una delle vostre classi solo "classe"?

Soluzione

No, non è possibile: la classe è una parola chiave!

Che cos'è una pila?

Una pila è una struttura sviluppata per memorizzare i dati in un modo molto specifico. Immaginate una pila di monete. Non è possibile mettere una moneta in un altro posto se non in cima alla pila.

Allo stesso modo, non è possibile prendere una moneta dalla pila da un punto diverso dalla cima della pila. Se si vuole ottenere la moneta che si trova in basso, è necessario rimuovere tutte le monete dai livelli superiori.

Il nome alternativo di una pila (ma solo nella terminologia informatica) è **LIFO**.

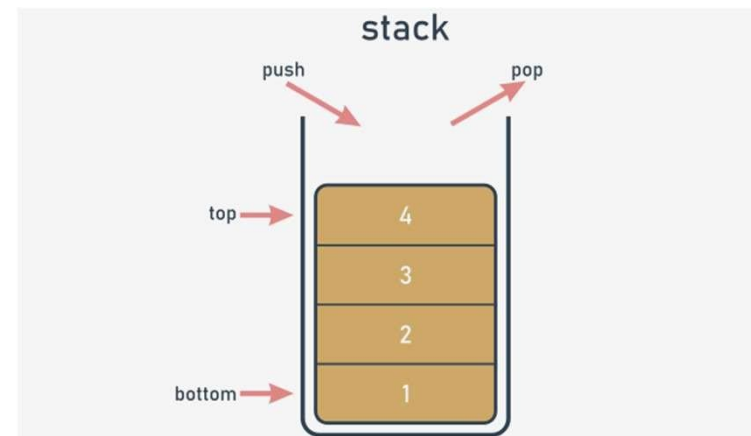
È un'abbreviazione per una descrizione molto chiara del comportamento dello stack: **Last In - First Out**. La moneta che è entrata per ultima nella pila uscirà per prima.

Una pila è un oggetto con due operazioni elementari, convenzionalmente chiamate **push** (quando un nuovo elemento viene messo in cima) e **pop** (quando un elemento esistente viene tolto dalla cima).

Gli stack sono utilizzati molto spesso in molti algoritmi classici ed è difficile immaginare l'implementazione di molti strumenti ampiamente utilizzati senza l'uso di stack.

Implementiamo una pila in Python. Si tratterà di una pila molto semplice e vi mostreremo come realizzarla con due approcci indipendenti: procedurale e oggettivo.

Cominciamo con il primo.



La pila - l'approccio procedurale

Per prima cosa, bisogna decidere come memorizzare i valori che arriveranno nello stack. Sugeriamo di utilizzare il metodo più semplice e di **impiegare una lista** per questo lavoro. Supponiamo che la dimensione della pila non sia limitata in alcun modo. Supponiamo inoltre che l'ultimo elemento della lista contenga l'elemento superiore. La pila stessa è già stata creata:

```
stack = []
```

Siamo pronti a **definire una funzione che mette un valore in pila**. Ecco i suoi presupposti:

- il nome della funzione è push;
- la funzione riceve un parametro (questo è il valore da mettere in pila)
- la funzione non restituisce nulla;
- la funzione aggiunge il valore del parametro alla fine della pila;

Ecco come abbiamo fatto: date un'occhiata:

```
def push(val):  
    stack.append(val)
```

Ora è il momento di una **funzione che prenda un valore dallo stack**. Ecco come si può fare:

- il nome della funzione è pop;
- la funzione non riceve alcun parametro;
- la funzione restituisce il valore preso dallo stack
- la funzione legge il valore dalla cima della pila e lo rimuove.

La funzione è qui:

```
def pop():  
    val = stack[-1]  
    del stack[-1]  
    return val
```

Nota: la funzione non controlla se ci sono elementi nella pila.

Assembliamo tutti i pezzi per mettere in moto la pila. Il **programma completo** inserisce tre numeri nella pila, li estrae e stampa i loro valori sullo schermo. Lo si può vedere nella finestra dell'editor.

Il programma visualizza sullo schermo il seguente testo:

1

2

3

```
stack = []
```

```
def push(val):  
    stack.append(val)
```

```
def pop():  
    val = stack[-1]  
    del stack[-1]  
    return val
```

```
push(3)  
spingere(2)  
spingere(1)
```

```
print(pop())  
print(pop())  
print(pop())
```


Lo stack: l'approccio procedurale contro l'approccio orientato agli oggetti

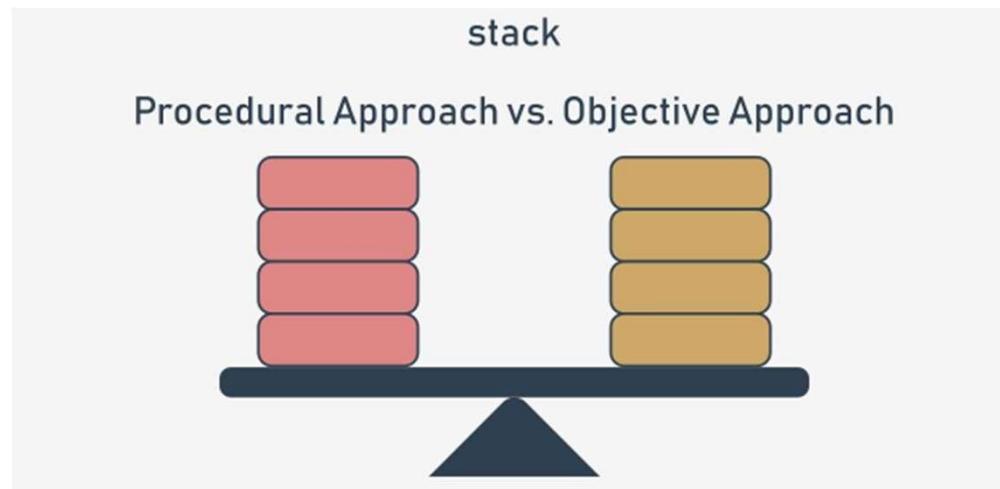
Lo stack procedurale è pronto. Naturalmente, ci sono alcuni punti deboli e l'implementazione potrebbe essere migliorata in molti modi (sfruttare le eccezioni per funzionare è una buona idea), ma in generale lo stack è completamente implementato e si può usare se necessario.

Ma più spesso lo si usa, più svantaggi si incontrano. Eccone alcuni:

- la variabile essenziale (la lista dello stack) è altamente **vulnerabile**; chiunque può modificarla in modo incontrollabile, distruggendo di fatto lo stack; questo non significa che sia stato fatto in modo malevolo, anzi, può accadere come risultato di una disattenzione, ad esempio quando qualcuno confonde i nomi delle variabili; immaginate di aver scritto per sbaglio qualcosa del genere:
- `stack[0] = 0`
- Il funzionamento della pila sarà completamente disorganizzato;
- Può anche accadere che un giorno si abbia bisogno di più di uno stack; si dovrà creare un'altra lista per la memorizzazione dello stack e probabilmente anche altre funzioni push e pop;
- Può anche accadere che non siano necessarie solo le funzioni push e pop, ma anche altre comodità; si possono certamente implementare, ma provate a immaginare cosa accadrebbe se aveste decine di stack implementati separatamente.

L'approccio oggettivo offre soluzioni per ciascuno dei problemi sopra citati. Per prima cosa nominiamoli:

- La capacità di nascondere (proteggere) i valori selezionati da accessi non autorizzati si chiama **incapsulamento**; **i valori incapsulati non possono essere né accessibili né modificati se si desidera utilizzarli in modo esclusivo**;
- Quando si ha una classe che implementa tutti i comportamenti necessari per lo stack, si possono produrre tutti gli stack che si vogliono; non è necessario copiare o replicare alcuna parte del codice;
- la possibilità di arricchire lo stack con nuove funzioni deriva dall'ereditarietà; si può creare una nuova classe (una sottoclasse) che eredita tutti i tratti esistenti dalla superclasse e ne aggiunge di nuovi.



Scriviamo ora una nuova implementazione di stack da zero. Questa volta utilizzeremo l'approccio per obiettivi, guidandovi passo dopo passo nel mondo della programmazione a oggetti.

La pila - l'approccio a oggetti

Naturalmente, l'idea principale rimane la stessa. Utilizzeremo un elenco come memoria dello stack. Dobbiamo solo sapere come inserire l'elenco nella classe.

Partiamo dall'inizio assoluto: è così che inizia la pila degli obiettivi:

classe Stack:

Ora ci aspettiamo due cose:

- vogliamo che la classe abbia **una proprietà come memoria dello stack** - dobbiamo **"installare" una lista all'interno di ogni oggetto della classe** (nota: ogni oggetto deve avere la propria lista - la lista non deve essere condivisa tra diversi stack)
- allora vogliamo che **l'elenco sia nascosto** alla vista degli utenti della classe.

Come si fa?

A differenza di altri linguaggi di programmazione, Python non consente di dichiarare una proprietà come questa. È invece necessario aggiungere una dichiarazione o un'istruzione specifica. Le proprietà devono essere aggiunte manualmente alla classe.

Come si può garantire che tale attività avvenga ogni volta che viene creato un nuovo stack?

C'è un modo semplice per farlo: bisogna **dotare la classe di una funzione specifica**, la cui specificità è duale:

- deve essere nominato in modo rigoroso;
- viene invocato implicitamente, quando viene creato il nuovo oggetto.

Una funzione di questo tipo è chiamata **costruttore**, poiché il suo scopo generale è quello di **costruire un nuovo oggetto**. Il costruttore deve sapere tutto sulla struttura dell'oggetto e deve eseguire tutte le inizializzazioni necessarie.

Aggiungiamo un costruttore molto semplice alla nuova classe. Date un'occhiata allo snippet:

```
class Stack:  
    def __init__(self):  
        print("Ciao!")
```

```
stack_object = Stack()
```

E ora:

- il nome del costruttore è sempre `__init__`;
- deve avere **almeno un parametro** (ne parleremo più avanti); il parametro è usato per rappresentare l'oggetto appena creato - si può usare il parametro per manipolare l'oggetto e per arricchirlo con le proprietà necessarie; lo si userà presto;
- nota: il parametro obbligatorio di solito si chiama `self`; è solo **una convenzione, ma è bene seguirla**: semplifica il processo di lettura e comprensione del codice.

Il codice è nell'editor. Eseguitelo ora.

Ecco il suo risultato:

Ciao!

Nota: non c'è traccia dell'invocazione del costruttore all'interno del codice. È stato invocato implicitamente e automaticamente. Utilizziamolo ora.

```
class Stack: # Definizione della classe Stack.  
    def __init__(self): # Definizione della funzione costruttore.  
        print("Ciao!")
```

```
stack_object = Stack() # Istanziare l'oggetto.
```

La pila - l'approccio a oggetti: continua

Qualsiasi modifica apportata all'interno del costruttore che modifichi lo stato del parametro self si rifletterà nell'oggetto appena creato.

Ciò significa che è possibile aggiungere qualsiasi proprietà all'oggetto e la proprietà rimarrà lì fino a quando l'oggetto non terminerà la sua vita o la proprietà verrà esplicitamente rimossa.

Aggiungiamo ora una sola proprietà al nuovo oggetto: una lista per lo stack. Lo chiameremo stack_list.

Proprio come qui:

```
class Stack:
```

```
    def __init__(self):  
        self.stack_list = []
```

```
stack_object = Stack()
```

```
print(len(stack_object.stack_list))
```

Nota:

- abbiamo usato la **notazione a punti**, proprio come quando si invocano i metodi; questa è la convenzione generale per accedere alle proprietà di un oggetto: bisogna nominare l'oggetto, mettere un punto (.) dopo di esso e specificare il nome della proprietà desiderata; non usare le parentesi! Non si vuole invocare un metodo, ma **accedere a una proprietà**;
- se si imposta il valore di una proprietà per la prima volta (come nel costruttore), la si sta creando; da quel momento in poi, l'oggetto ha la proprietà ed è pronto a usare il suo valore;
- abbiamo fatto qualcosa di più nel codice - abbiamo cercato di accedere alla proprietà `stack_list` dall'esterno della classe subito dopo la creazione dell'oggetto; vogliamo controllare la lunghezza attuale dello stack - ci siamo riusciti?

Sì, è così: il codice produce il seguente risultato:

0

Questo non è ciò che vogliamo dallo stack. Preferiamo che `stack_list` sia **nascosto al mondo esterno**. È possibile?

Sì, ed è semplice, ma non molto intuitivo.

La pila - l'approccio a oggetti: continua

Date un'occhiata: abbiamo aggiunto due trattini bassi prima del nome `stack_list`, niente di più:

```
class Stack:
```

```
    def __init__(self):  
        self.__stack_list = []
```

```
stack_object = Stack()  
print(len(stack_object.__stack_list))
```

La modifica invalida il programma.

Perché?

Quando un componente di una classe ha un **nome che inizia con due trattini bassi (__)**, diventa **privato**: ciò significa che vi si può accedere solo dall'interno della classe.

Non è visibile dal mondo esterno. Questo è il modo in cui Python implementa il concetto di **incapsulamento**.

Eseguire il programma per verificare le nostre ipotesi: dovrebbe essere sollevata un'eccezione

`AttributeError`.

L'approccio a oggetti: uno stack da zero

Ora è il momento delle due funzioni (metodi) che implementano le operazioni di *push* e *pop*. Python presuppone che una funzione di questo tipo (un'attività di classe) sia **immersa nel corpo della classe**, proprio come un costruttore.

Vogliamo invocare queste funzioni per spingere e togliere i valori. Ciò significa che entrambe devono essere accessibili all'utente di ogni classe (a differenza dell'elenco costruito in precedenza, che è nascosto agli utenti della classe ordinaria).

Un componente di questo tipo è chiamato **pubblico**, quindi **non è possibile iniziare il suo nome con due (o più) trattini bassi**. C'è un altro requisito: **il nome non deve avere più di un trattino basso**. Poiché nessun trattino basso soddisfa pienamente il requisito, si può ritenere che il nome sia accettabile.

Le funzioni sono semplici. Date un'occhiata:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]

        del self.__stack_list[-1]
        return val
```

```
stack_object = Stack()
stack_object.push(3)
stack_object.push(2)
stack_object.push(1)
print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

Tuttavia, c'è qualcosa di molto strano nel codice. Le funzioni sembrano familiari, ma hanno più parametri rispetto alle loro controparti procedurali.

In questo caso, entrambe le funzioni hanno un parametro chiamato `self` nella prima posizione dell'elenco dei parametri.

È necessario? Sì, lo è.

Tutti i metodi devono avere questo parametro. Ha lo stesso ruolo del primo parametro del costruttore.

Consente al metodo di accedere alle entità (proprietà e attività/metodi) svolte dall'oggetto reale. Non è possibile ometterlo. Ogni volta che Python invoca un metodo, invia implicitamente l'oggetto corrente come primo argomento.

Ciò significa che un **metodo è obbligato ad avere almeno un parametro, che viene utilizzato da Python stesso**, senza che l'utente abbia alcuna influenza su di esso.

Se il metodo non ha bisogno di alcun parametro, questo deve essere specificato comunque. Se è stato progettato per elaborare un solo parametro, è necessario specificarne due, ma il ruolo del primo è sempre lo stesso.

C'è un'altra cosa che richiede una spiegazione: il modo in cui i metodi vengono invocati all'interno della variabile `__stack_list`.

Fortunatamente, è molto più semplice di quanto sembri:

- il primo stadio consegna l'oggetto come un tutto → sé;
- poi, è necessario raggiungere l'elenco `__stack_list` → `self.__stack_list`;
- con `__stack_list` pronta per essere utilizzata, si può eseguire il terzo e ultimo passo → `self.__stack_list.append(val)`.

La dichiarazione della classe è completa e tutti i suoi componenti sono stati elencati. La classe è pronta per essere utilizzata.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val
```

```
stack_object = Stack()
stack_object.push(3)
stack_object.push(2)
stack_object.push(1)
print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

L'approccio a oggetti: uno stack da zero

Avere una classe di questo tipo apre alcune nuove possibilità. Ad esempio, è possibile avere più di uno stack che si comporta allo stesso modo. Ogni stack avrà la sua copia di dati privati, ma utilizzerà lo stesso insieme di metodi. Questo è esattamente ciò che vogliamo per questo esempio.

Analizzare il codice:

```
class Stack:
```

```
    def __init__(self):
```

```
        self.__stack_list = []
```

```
    def push(self, val):
```

```
        self.__stack_list.append(val)
```

```
    def pop(self):
```

```
        val = self.__stack_list[-1]
```

```
        del self.__stack_list[-1]
```

```
        restituire val
```

```
stack_object_1 = Stack()
```

```
stack_object_2 = Stack()
```

```
stack_object_1.push(3)
```

```
stack_object_2.push(stack_object_1.pop())
```

```
print(stack_object_2.pop())
```

Ci sono **due pile create dalla stessa classe di base**. Funzionano **in modo indipendente**. È possibile crearne altri, se lo si desidera.

Eseguite il codice nell'editor e vedete cosa succede. Eseguite i vostri esperimenti.

L'approccio a oggetti: una pila da zero (continua)

Analizzate lo snippet qui sotto: abbiamo creato tre oggetti della classe Stack. Poi, li abbiamo messi in relazione tra loro. Cercate di prevedere il valore che verrà visualizzato sullo schermo.

```
class Stack:
    def __init__(self):
        self.__stack_list = []
    def push(self, val):
        self.__stack_list.append(val)
    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        restituire val

little_stack = Stack ()
another_stack = Stack()
funny_stack = Stack()
little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)
print(funny_stack.pop())
```

Qual è il risultato? Eseguite il programma e verificate se avevate ragione.

L'approccio a oggetti: una pila da zero (continua)

Ora andiamo un po' oltre. **Aggiungiamo una nuova classe per gestire le pile.**

La nuova classe deve essere in grado di **valutare la somma di tutti gli elementi attualmente memorizzati nello stack.**

Non vogliamo modificare lo stack precedentemente definito. È già abbastanza buono nelle sue applicazioni e non vogliamo che venga modificato in alcun modo. Vogliamo un nuovo stack con nuove funzionalità. In altre parole, vogliamo costruire una sottoclasse della classe Stack già esistente.

Il primo passo è semplice: basta **definire una nuova sottoclasse che punta alla classe che sarà usata come superclasse.**

Ecco come si presenta:

```
class AddingStack(Stack):
```

La classe non definisce ancora alcun nuovo componente, ma ciò non significa che sia vuota. **Riceve tutti i componenti definiti dalla sua superclasse**: il nome della superclasse è scritto prima dei due punti, direttamente dopo il nome della nuova classe.

Questo è ciò che vogliamo dal nuovo stack:

- vogliamo che il metodo push non solo spinga il valore sullo stack, ma aggiunga anche il valore alla variabile sum;
- vogliamo che la funzione pop non solo estragga il valore dallo stack, ma anche che sottragga il valore dalla variabile sum.

Per prima cosa, aggiungiamo una nuova variabile alla classe. Sarà una **variabile privata**, come lo stack list. Non vogliamo che nessuno manipoli il valore della somma.

Come già sapete, l'aggiunta di una nuova proprietà alla classe avviene tramite il costruttore. Sapete già come farlo, ma c'è qualcosa di veramente intrigante all'interno del costruttore. Date un'occhiata:

```
class AddingStack(Stack):
```

```
    def __init__(self):  
        Stack.__init__(self)  
        self.__sum = 0
```

La seconda riga del corpo del costruttore crea una proprietà chiamata `__sum`, che memorizzerà il totale di tutti i valori dello stack.

Ma la linea che la precede sembra diversa. A cosa serve? È davvero necessaria? Sì, lo è.

Contrariamente a molti altri linguaggi, Python obbliga a **invocare esplicitamente il costruttore di una superclasse**. Omettere questo punto avrà effetti dannosi: l'oggetto sarà privato dell'elenco `__stack_list`. Una pila di questo tipo non funzionerà correttamente.

Questo è l'unico momento in cui è possibile invocare esplicitamente uno dei costruttori disponibili: può essere fatto all'interno del costruttore della sottoclasse.

Si noti la sintassi:

- si specifica il nome della superclasse (questa è la classe di cui si vuole eseguire il costruttore)
- si mette un punto (.) dopo di esso;
- si specifica il nome del costruttore;
- si deve puntare all'oggetto (l'istanza della classe) che deve essere inizializzato dal costruttore - questo è il motivo per cui si deve specificare l'argomento e usare la variabile `self`; si noti: **invocare qualsiasi metodo (compresi i costruttori) dall'esterno della classe non richiede mai di mettere l'argomento `self` nella lista degli argomenti** - invocare un metodo dall'interno della classe richiede l'uso esplicito dell'argomento `self`, che deve essere messo per primo nella lista.

Nota: in genere è consigliabile invocare il costruttore della superclasse prima di qualsiasi altra inizializzazione che si voglia eseguire all'interno della sottoclasse. Questa è la regola che abbiamo seguito nello snippet.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val
```

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

L'approccio a oggetti: una pila da zero (continua)

In secondo luogo, aggiungiamo due metodi. Ma vi chiediamo: è davvero un'aggiunta? Abbiamo già questi metodi nella superclasse. Possiamo fare qualcosa di simile?

Sì, è possibile. Significa che **cambieremo la funzionalità dei metodi, non i loro nomi**. Possiamo dire più precisamente che l'interfaccia (il modo in cui vengono gestiti gli oggetti) della classe rimane la stessa quando si cambia contemporaneamente l'implementazione.

Cominciamo con l'implementazione della funzione push. Ecco cosa ci aspettiamo da essa:

- per aggiungere il valore alla variabile `__sum`;
- per inserire il valore nella pila.

Nota: la seconda attività è già stata implementata nella superclasse, quindi possiamo usarla. Inoltre, dobbiamo usarla, perché non c'è altro modo per accedere alla variabile `__stackList`.

Ecco come appare il metodo push nella sottoclasse:

```
def push(self, val):  
    self.__sum += val  
    Stack.push(self, val)
```

Si noti il modo in cui abbiamo invocato la precedente implementazione del metodo push (quella disponibile nella superclasse):

- dobbiamo specificare il nome della superclasse; questo è necessario per indicare chiaramente la classe che contiene il metodo, per evitare di confonderlo con qualsiasi altra funzione con lo stesso nome;
- dobbiamo specificare l'oggetto target e passarlo come primo argomento (in questo contesto non viene aggiunto implicitamente all'invocazione).

Diciamo che il metodo push è stato sovrascritto: lo stesso nome della superclasse rappresenta ora una funzionalità diversa.

L'approccio a oggetti: una pila da zero (continua)

Questa è la nuova funzione pop:

```
def pop(self):  
    val = Stack.pop(self)  
    self.__sum -= val  
    return val
```

Finora abbiamo definito la variabile `__sum`, ma non abbiamo fornito un metodo per ottenere il suo valore.

Sembra che sia nascosta. Come possiamo rivelarla e farlo in modo da proteggerla dalle modifiche?

Dobbiamo definire un nuovo metodo. Lo chiameremo `get_sum`. Il suo unico compito sarà quello di **restituire il valore** `__sum`.

Eccola qui:

```
def get_sum(self):  
    return self.__sum
```


Quindi, diamo un'occhiata al programma nell'editor. Il codice completo della classe è presente. Ora possiamo verificarne il funzionamento e lo facciamo con l'aiuto di pochissime righe di codice aggiuntive. Come si può vedere, aggiungiamo cinque valori successivi allo stack, stampiamo la loro somma e li togliamo tutti dallo stack.

Ok, questa è stata una breve introduzione alla programmazione a oggetti di Python. Presto vi parleremo di tutto questo in modo più dettagliato.

classe Stack:

```
def __init__(self):
```

```
    self.__stack_list = []
```

```
def push(self, val):
```

```
    self.__stack_list.append(val)
```

```
def pop(self):
```

```
    val = self.__stack_list[-1]
```

```
    del self.__stack_list[-1]
```

```
    return val
```

```
classe AddingStack(Stack):
```

```
    def __init__(self):
```

```
        Stack.__init__(self)
```

```
        self.__sum = 0
```

```
    def get_sum(self):
```

```
        return self.__sum
```

```
    def push(self, val):
```

```
        self.__sum += val
```

```
        Stack.push(self, val)
```

```
    def pop(self):
```

```
        val = Stack.pop(self)
```

```
        self.__sum -= val
```

```
        return val
```

```
stack_object = AddingStack()
```

```
for i in range(5):
```

```
    stack_object.push(i)
```

```
print(stack_object.get_sum())
```

```
for i in range(5):
```

```
    print(stack_object.pop())
```

Punti di forza

1. Uno **stack** è un oggetto progettato per memorizzare i dati utilizzando il modello **LIFO**. Lo stack di solito esegue almeno due operazioni, denominate **push()** e **pop()**.
2. L'implementazione dello stack in un modello procedurale solleva diversi problemi che possono essere risolti con le tecniche offerte dall'**OOP** (Object Oriented Programming):
3. Un **metodo** di classe è in realtà una funzione dichiarata all'interno della classe e in grado di accedere a tutti i componenti della classe.
4. La parte della classe Python responsabile della creazione di nuovi oggetti si chiama **costruttore** ed è implementata come un metodo dal nome `__init__`.
5. Ogni dichiarazione di metodo della classe deve contenere almeno un parametro (sempre il primo), solitamente indicato come `self`, che viene utilizzato dagli oggetti per identificarsi.
6. Se si vuole nascondere uno dei componenti di una classe al mondo esterno, si deve iniziare il suo nome con `__`. Tali componenti sono chiamati **privati**.

Esercizio 1

Supponendo che esista una classe chiamata Snakes, scrivete la prima riga della dichiarazione di classe di Python, esprimendo il fatto che la nuova classe è in realtà una sottoclasse di Snake.

Esercizio 2

Manca qualcosa nella seguente dichiarazione: cosa?

```
class Serpenti:  
    def __init__():  
        self.sound = 'Sssssss'
```

Esercizio 3

Modificare il codice per garantire che la proprietà velenosa sia privata.

```
class Serpenti:  
    def __init__(self):  
        self.venomous = Vero
```

Esercizio 1

Supponendo che esista una classe chiamata Snakes, scrivete la prima riga della dichiarazione di classe di Python, esprimendo il fatto che la nuova classe è in realtà una sottoclasse di Snake.

Soluzione

```
class Python(Serpenti):
```

Esercizio 2

Manca qualcosa nella seguente dichiarazione: cosa?

```
class Serpenti:  
    def __init__():  
        self.sound = 'Sssssss'
```

soluzione

Il costruttore `__init__()` manca del parametro obbligatorio (dovremmo chiamarlo `self` per essere conformi agli standard).

Esercizio 3

Modificare il codice per garantire che la proprietà velenosa sia privata.

classe Serpenti:

```
def __init__(self):  
    self.venomous = Vero
```

soluzione

Il codice dovrebbe apparire come segue:

classe Serpenti:

```
def __init__(self):  
    self.__venomous = Vero
```

LAB-36

Tempo stimato

20-45 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le competenze dello studente nella definizione delle classi;
- utilizzando le classi esistenti per creare nuove classi dotate di nuove funzionalità.

Scenario

Di recente abbiamo mostrato come estendere le possibilità di *Stack* definendo una nuova classe (cioè una sottoclasse) che mantiene tutti i tratti ereditati e ne aggiunge di nuovi.

Il vostro compito è quello di estendere il comportamento della classe *Stack* in modo tale che la classe sia in grado di contare tutti gli elementi che vengono spinti e rimossi (supponiamo che il conteggio dei pop sia sufficiente).

Utilizzate la classe *Stack* che abbiamo fornito nell'editor.

Seguire i suggerimenti:

- introdurre una proprietà destinata al conteggio delle operazioni pop e denominarla in modo da garantirne l'occultamento;
- inizializzarlo a zero nel costruttore;
- fornire un metodo che restituisca il valore attualmente assegnato al contatore (chiamarlo `get_counter()`).

Completate il codice nell'editor. Eseguitelo per verificare se il codice produce 100.

Completate il codice nell'editor. Eseguitelo per verificare se il codice produce 100.

```
def __init__(self):
    self.__stk = []
def push(self, val):
    self.__stk.append(val)
def pop(self):
    val = self.__stk[-1]
    del self.__stk[-1]
    return val
```

classe CountingStack(Stack):

```
def __init__(self):
    #
    # Riempire il costruttore con le azioni appropriate.
    #

def get_counter(self):
    #
    # Presentare al mondo il valore attuale del contatore.
    #
```

```
def pop(self):  
    #  
    # Fare il pop e aggiornare il contatore.  
    #
```

```
stk = CountingStack()  
for i in range(100):  
    stk.push(i)  
    stk.pop()  
print(stk.get_counter())
```

LAB-37

Tempo stimato

20-45 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le capacità dello studente nel definire le classi da zero;
- implementare strutture di dati standard come classi.

Scenario

Come già sapete, una *pila* è una struttura di dati che realizza il modello LIFO (Last In - First Out). È facile e ci siete già perfettamente abituati.

Proviamo ora qualcosa di nuovo. Una *coda* è un modello di dati caratterizzato dal termine **FIFO: First In - First Out**. Nota: una normale coda (linea) che conoscete nei negozi o negli uffici postali funziona esattamente nello stesso modo: il cliente che è arrivato per primo viene servito per primo.

Il compito è quello di implementare la classe Queue con due operazioni di base:

- `put(elemento)`, che mette un elemento alla fine della coda;
- `get()`, che prende un elemento dalla parte anteriore della coda e lo restituisce come risultato (la coda non può essere vuota per poterlo eseguire con successo).

Seguire i suggerimenti:

- utilizzare un elenco come memoria (proprio come abbiamo fatto con la pila)
- `put()` deve aggiungere elementi all'inizio dell'elenco, mentre `get()` deve rimuovere gli elementi dalla fine dell'elenco;
- definire una nuova eccezione chiamata `QueueError` (scegliere un'eccezione da cui derivare) e sollevarla quando `get()` tenta di operare su un elenco vuoto.

Completate il codice che vi abbiamo fornito nell'editor. Eseguite per verificare se il risultato è simile al nostro.

Risultato previsto

1

cane

Falso

Errore di coda

```
class QueueError(???): # Scegliere la classe base per la nuova eccezione.
```

```
#
```

```
# Scrivere il codice qui
```

```
#
```

```
class Coda:
```

```
    def __init__(self):
```

```
        #
```

```
        # Scrivere il codice qui
```

```
        #
```

```
    def put(self, elem):
```

```
        #
```

```
        # Scrivere il codice qui
```

```
        #
```

```
    def get(self):
```

```
        #
```

```
        # Scrivere il codice qui
```

```
        #
```

```
que = coda()
que.put(1)
que.put("cane")
que.put(False)
provare:
    per i in range(4):
        print(que.get())
eccetto:
    print("Errore di coda")
```

LAB-38

Tempo stimato

15-30 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le competenze dello studente nella definizione delle sottoclassi;
- aggiungere una nuova funzionalità a una classe esistente.

Scenario

Il vostro compito è quello di estendere leggermente le capacità della classe Queue. Vogliamo che abbia un metodo senza parametri che restituisca True se la coda è vuota e False altrimenti.

Completate il codice che vi abbiamo fornito nell'editor. Eseguitelo per verificare se produce un risultato simile al nostro.

Di seguito è possibile copiare il codice utilizzato nel laboratorio precedente

Risultato previsto

1

cane

Falso

Coda vuota

```
class QueueError(???):
    passaggio
class Coda:
    #
    # Codice del laboratorio precedente.
    #
class SuperQueue(Coda):
    #
    # Scrivere qui il nuovo codice.
    #
que = SuperQueue()
que.put(1)
que.put("cane")
que.put(False)
for i in range(4):
    if not que.isempty():
        print(que.get())
    else:
        print("Coda vuota")
```


Variabili d'istanza

In generale, una classe può essere dotata di due diversi tipi di dati per formare le sue proprietà. Uno di questi è già stato visto quando abbiamo visto le pile.

Questo tipo di proprietà di classe esiste quando e solo quando viene esplicitamente creata e aggiunta a un oggetto. Come si sa, ciò può essere fatto durante l'inizializzazione dell'oggetto, eseguita dal costruttore.

Inoltre, può essere fatto in qualsiasi momento della vita dell'oggetto. Inoltre, qualsiasi proprietà esistente può essere rimossa in qualsiasi momento.

Questo approccio ha alcune importanti conseguenze:

- oggetti diversi della stessa classe **possono possedere insiemi diversi di proprietà**;
- Ci deve essere un modo per **verificare in modo sicuro se un oggetto specifico possiede la proprietà** che si vuole utilizzare (a meno che non si voglia provocare un'eccezione, ma vale sempre la pena considerarlo).
- Ogni oggetto **ha un proprio insieme di proprietà**, che non interferiscono in alcun modo l'una con l'altra.

Tali variabili (proprietà) sono chiamate **variabili di istanza**.

Il termine *istanza* suggerisce che sono strettamente collegati agli oggetti (che sono istanze di classe), non alle classi stesse. Vediamoli più da vicino.

Ecco un esempio:

```
class EsempioClasse:  
    def __init__(self, val = 1):  
        self.first = val
```

```
    def set_second(self, val):  
        self.second = val
```

```
esempio_oggetto_1 = EsempioClasse()  
esempio_oggetto_2 = EsempioClasse(2)
```

```
esempio_oggetto_2.set_second(3)
```

```
esempio_oggetto_3 = EsempioClasse(4)  
esempio_oggetto_3.terzo = 5
```

```
print(esempio_oggetto_1.__dict__)  
print(esempio_oggetto_2.__dict__)  
print(esempio_oggetto_3.__dict__)
```

Prima di entrare nel dettaglio, è necessaria un'ulteriore spiegazione. Osservate le ultime tre righe del codice. Gli oggetti Python, quando vengono creati, sono dotati di un **piccolo insieme di proprietà e metodi predefiniti**. Ogni oggetto li ha, che li si voglia o meno. Una di queste è una variabile chiamata `__dict__` (è un dizionario). La variabile contiene i nomi e i valori di tutte le proprietà (variabili) di cui l'oggetto è attualmente dotato. Utilizziamola per presentare in modo sicuro il contenuto di un oggetto.

Ora immergiamoci nel codice:

- la classe denominata `ExampleClass` ha un costruttore che **crea incondizionatamente una variabile di istanza** denominata `first` e la imposta con il valore passato attraverso il primo argomento (dal punto di vista dell'utente della classe) o il secondo argomento (dal punto di vista del costruttore); si noti il valore predefinito del parametro - ogni trucco che si può fare con un normale parametro di funzione può essere applicato anche ai metodi;
- la classe ha anche un **metodo che crea un'altra variabile di istanza**, denominata `second`;
- abbiamo creato tre oggetti della classe `ExampleClass`, ma tutte queste istanze differiscono tra loro:

`esempio_oggetto_1` ha solo la proprietà denominata `prima`;

`esempio_oggetto_2` ha due proprietà: `prima` e `seconda`;

`esempio_oggetto_3` è stato arricchito con una proprietà di nome `terzo al volo`, al di fuori del codice della classe: questo è possibile e del tutto lecito.

L'output del programma mostra chiaramente che le nostre ipotesi sono corrette: eccolo:

```
{'first': 1}
```

```
{'secondo': 3, 'first': 2}
```

```
{'terzo': 5, 'primo': 4}
```

C'è un'ulteriore conclusione che deve essere enunciata: **la modifica di una variabile di istanza di un qualsiasi oggetto non ha alcun impatto su tutti gli altri oggetti**. Le variabili di istanza sono perfettamente isolate le une dalle altre.

Variabili d'istanza: continua

Guardate l'esempio modificato nell'editor.

È quasi identico al precedente. L'unica differenza è nei nomi delle proprietà. Abbiamo **aggiunto due trattini bassi (__)** davanti ad esse.

Questa aggiunta rende **privata** la variabile di istanza, che diventa inaccessibile dal mondo esterno. Il comportamento effettivo di questi nomi è un po' più complicato, quindi eseguiamo il programma. Questo è l'output:

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

Riuscite a vedere questi strani nomi pieni di trattini bassi? Da dove vengono?

Quando Python vede che si vuole aggiungere una variabile di istanza a un oggetto e che lo si fa all'interno di uno qualsiasi dei metodi dell'oggetto, **distrugge l'operazione** nel modo seguente:

- mette il nome della classe prima del proprio nome;
- mette un trattino basso aggiuntivo all'inizio.

Ecco perché `__first` diventa `_ExampleClass__first`.

Il nome è ora completamente accessibile dall'esterno della classe. È possibile eseguire un codice come questo:

```
print(esempio_oggetto_1._Classe_Esempio__prima)
```

e si otterrà un risultato valido senza errori o eccezioni.

Come si può notare, rendere privata una proprietà è limitato.

La manipolazione non funziona se si aggiunge una variabile di istanza privata al di fuori del codice della classe.
In questo caso, si comporterà come qualsiasi altra proprietà ordinaria.

Codice:

```
class EsempioClasse:
    def __init__(self, val = 1):
        self.__first = val

    def set_second(self, val = 2):
        self.__second = val

esempio_oggetto_1 = EsempioClasse()
esempio_oggetto_2 = EsempioClasse(2)
esempio_oggetto_2.set_second(3)
esempio_oggetto_3 = EsempioClasse(4)
esempio_oggetto_3.__terzo = 5

print(esempio_oggetto_1.__dict__)
print(esempio_oggetto_2.__dict__)
print(esempio_oggetto_3.__dict__)
```

Variabili di classe

Una variabile di classe è **una proprietà che esiste in una sola copia ed è memorizzata al di fuori di qualsiasi oggetto**.

Nota: non esiste una variabile di istanza se non c'è un oggetto nella classe; una variabile di classe esiste in una copia anche se non ci sono oggetti nella classe.

Le variabili di classe sono create in modo diverso dalle loro sorelle di istanza. L'esempio vi dirà di più:

```
class EsempioClasse:
    contatore = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1
```

```
esempio_oggetto_1 = EsempioClasse()
esempio_oggetto_2 = EsempioClasse(2)
esempio_oggetto_3 = EsempioClasse(4)
```

```
print(esempio_oggetto_1.__dict__, esempio_oggetto_1.counter)
print(esempio_oggetto_2.__dict__, esempio_oggetto_2.counter)
print(esempio_oggetto_3.__dict__, esempio_oggetto_3.counter)
```

Guarda:

- c'è un'assegnazione nella prima riga della definizione della classe: essa imposta la variabile denominata counter a 0; l'inizializzazione della variabile all'interno della classe, ma al di fuori di qualsiasi suo metodo, rende la variabile una variabile di classe;
- L'accesso a tale variabile è identico a quello di qualsiasi attributo di istanza: lo si può vedere nel corpo del costruttore; come si può notare, il costruttore incrementa la variabile di uno; in effetti, la variabile conta tutti gli oggetti creati.

L'esecuzione del codice provoca il seguente risultato:

```
{'_ExampleClass__first': 1} 3
```

```
{'_ExampleClass__first': 2} 3
```

```
{'_ExampleClass__first': 4} 3
```

Dall'esempio emergono due importanti conclusioni:

- **Le** variabili di classe **non sono mostrate nel __dict__ di un oggetto** (questo è naturale, perché le variabili di classe non sono parti di un oggetto), ma si può sempre provare a cercare la variabile con lo stesso nome, ma a livello di classe - lo mostreremo molto presto;
- una variabile di classe **presenta sempre lo stesso valore** in tutte le istanze della classe (oggetti)

Variabili di classe: continua

La manipolazione del nome di una variabile di classe ha gli stessi effetti di quelle già conosciute.

Guardate l'esempio nell'editor. Riuscite a indovinare l'output?

Eseguite il programma e verificate se le vostre previsioni erano corrette. Tutto funziona come previsto, vero?

Codice:

```
classe EsempioClasse:
```

```
    __counter = 0
```

```
    def __init__(self, val = 1):
```

```
        self.__first = val
```

```
        ExampleClass.__counter += 1
```

```
esempio_oggetto_1 = EsempioClasse()
```

```
esempio_oggetto_2 = EsempioClasse(2)
```

```
esempio_oggetto_3 = EsempioClasse(4)
```

```
print(esempio_oggetto_1.__dict__, esempio_oggetto_1._ExampleClass__counter)
```

```
print(esempio_oggetto_2.__dict__, esempio_oggetto_2._ExampleClass__counter)
```

```
print(esempio_oggetto_3.__dict__, esempio_oggetto_3._ExampleClass__counter)
```

Variabili di classe: continua

Abbiamo già detto che le variabili di classe esistono anche quando non è stata creata alcuna istanza di classe (oggetto).

Ora cogliamo l'occasione per mostrare **la differenza tra queste due variabili** `__dict__`, quella della classe e quella dell'oggetto.

Guardate il codice nell'editor. La prova è lì.

Guardiamo più da vicino:

1. Definiamo una classe chiamata `ClasseEsempio`;
2. La classe definisce una variabile di classe, denominata `varia`;
3. Il costruttore della classe imposta la variabile con il valore del parametro;
4. Il nome della variabile è l'aspetto più importante dell'esempio perché:
 - Cambiando l'assegnazione in `self.varia = val`, si creerebbe una variabile di istanza con lo stesso nome di quella della classe;
 - Cambiando l'assegnazione in `varia = val` si opererebbe sulla variabile locale di un metodo; (si consiglia vivamente di testare entrambi i casi sopra descritti, in modo da ricordare più facilmente la differenza).
5. La prima riga del codice fuori classe stampa il valore dell'attributo `ExampleClass.varia`; si noti che il valore viene utilizzato prima che venga istanziato il primo oggetto della classe.

Eseguite il codice nell'editor e verificate il risultato.

Come si può vedere, il `__dict__` della classe contiene molti più dati rispetto alla controparte dell'oggetto. La maggior parte di essi è inutile: quello che vogliamo controllare con attenzione mostra il valore attuale della variabile.

Si noti che il `__dict__` dell'oggetto è vuoto: l'oggetto non ha variabili di istanza.

```
class EsempioClasse:
    varia = 1
    def __init__(self, val):
        EsempioClasse.varia = val

print(ExampleClass.__dict__)
esempio_oggetto = EsempioClasse(2)

print(ExampleClass.__dict__)
print(oggetto_esempio.__dict__)
```

Verifica dell'esistenza di un attributo

L'atteggiamento di Python nei confronti dell'istanziamento degli oggetti solleva un problema importante: a differenza di altri linguaggi di programmazione, **non ci si può aspettare che tutti gli oggetti della stessa classe abbiano lo stesso insieme di proprietà.**

Proprio come nell'esempio dell'editor. Guardate con attenzione.

L'oggetto creato dal costruttore può avere solo uno di due possibili attributi: a o b.

L'esecuzione del codice produrrà il seguente risultato:

1

Traceback (ultima chiamata):

```
File ".main.py", riga 11, in  
    print(esempio_oggetto.b)
```

AttributeError: l'oggetto 'ExampleClass' non ha l'attributo 'b'.

Come si può vedere, l'accesso a un attributo di un oggetto (classe) inesistente causa un'eccezione AttributeError.

classe EsempioClasse:

```
def __init__(self, val):  
    if val % 2 != 0:  
        self.a = 1  
    else:  
        self.b = 1
```

```
esempio_oggetto = EsempioClasse(1)  
print(esempio_oggetto.a)  
print(esempio_oggetto.b)
```

Verifica dell'esistenza di un attributo: continua

L'istruzione try-except consente di evitare problemi con proprietà inesistenti.

È facile: guardate il codice nell'editor.

Come si può vedere, questa azione non è molto sofisticata. In sostanza, abbiamo semplicemente nascosto il problema sotto il tappeto.

Fortunatamente esiste un altro modo per affrontare il problema.

Python fornisce una **funzione in grado di verificare in modo sicuro se un oggetto/classe contiene una proprietà specificata**. La funzione si chiama `hasattr` e si aspetta che le vengano passati due argomenti:

- la classe o l'oggetto da controllare;
- il nome della proprietà di cui deve essere segnalata l'esistenza (nota: deve essere una stringa contenente il nome dell'attributo, non il solo nome)

La funzione restituisce Vero o Falso.

Ecco come potete utilizzarlo:

```
class ExampleClass:
```

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

```
        if val % 2 != 0:
```

```
            self.a = 1
```

```
        else:
```

```
            self.b = 1
```

```
example_object = ExampleClass(1)
```

```
print(example_object.a)
```

```
if hasattr(example_object, 'b'):
```

```
    print(example_object.b)
```

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1
```

```
example_object = ExampleClass(1)
print(example_object.a)
```

```
try:
    print(example_object.b)
except AttributeError:
    pass
```

Verifica dell'esistenza di un attributo: continua

Non dimenticate che la funzione `hasattr()` può operare anche sulle classi. Si può usare **per scoprire se una variabile di classe è disponibile**, proprio come nell'esempio dell'editor.

La funzione restituisce `True` se la classe specificata contiene un determinato attributo e `False` altrimenti.

Riuscite a indovinare l'output del codice? Eseguite per verificare le vostre ipotesi.

Un altro esempio: guardate il codice qui sotto e cercate di prevedere il suo risultato:

```
class ExampleClass:
```

```
    a = 1
```

```
    def __init__(self):
```

```
        self.b = 2
```

```
example_object = ExampleClass()
```

```
print(hasattr(example_object, 'b'))
```

```
print(hasattr(example_object, 'a'))
```

```
print(hasattr(ExampleClass, 'b'))
```

```
print(hasattr(ExampleClass, 'a'))
```


Avete avuto successo? Eseguite il codice per verificare le vostre previsioni.

Bene, siamo arrivati alla fine di questa sezione. Nella prossima sezione parleremo dei metodi, poiché i metodi guidano gli oggetti e li rendono attivi.

```
class ExampleClass:
```

```
    attr = 1
```

```
print(hasattr(ExampleClass, 'attr'))
```

```
print(hasattr(ExampleClass, 'prop'))
```

Punti di forza

1. Una **variabile di istanza** è una proprietà la cui esistenza dipende dalla creazione di un oggetto. Ogni oggetto può avere un insieme diverso di variabili di istanza.
Inoltre, possono essere aggiunte e rimosse liberamente dagli oggetti durante la loro vita. Tutte le variabili di istanza degli oggetti sono memorizzate all'interno di un dizionario dedicato, chiamato `__dict__`, contenuto in ogni oggetto separatamente.
2. Una variabile di istanza può essere privata quando il suo nome inizia con `_`, ma non dimenticate che una proprietà di questo tipo è comunque accessibile dall'esterno della classe usando un **nome manipolato** costruito come `_ClassName__PrivatePropertyName`.
3. Una **variabile di classe** è una proprietà che esiste esattamente in una copia e non ha bisogno di alcun oggetto creato per essere accessibile. Tali variabili non vengono mostrate come contenuto di `__dict__`.
Tutte le variabili di una classe sono memorizzate all'interno di un dizionario dedicato, chiamato `__dict__`, contenuto in ogni classe separatamente.
4. Una funzione chiamata `hasattr()` può essere usata per determinare se un oggetto/classe contiene una proprietà specificata.
Ad esempio:

```
class Sample:
    gamma = 0 # Class variable.
    def __init__(self):
        self.alpha = 1 # Instance variable.
        self.__delta = 3 # Private instance variable.
```

```
obj = Sample()
obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)
print(obj.__dict__)
```

The code outputs:

```
{'alpha': 1, '_Sample__delta': 3, 'beta': 2}
```

Metodi in dettaglio

Riassumiamo tutti i fatti relativi all'uso dei metodi nelle classi Python.

Come già sapete, un **metodo è una funzione incorporata in una classe**.

C'è un requisito fondamentale: un **metodo deve avere almeno un parametro** (non esistono metodi senza parametri: un metodo può essere invocato senza argomenti, ma non dichiarato senza parametri).

Il primo (o unico) parametro è solitamente chiamato self. Suggeriamo di seguire questa convenzione: è comunemente usata, e si rischia di avere qualche sorpresa usando altri nomi.

Il nome self suggerisce lo scopo del parametro: **identifica l'oggetto per il quale viene invocato il metodo**.

Se state per invocare un metodo, non dovete passare l'argomento per il parametro self: Python lo imposterà per voi.

L'esempio nell'editor mostra la differenza.

```
class Classy:
```

```
    def method(self):  
        print("method")
```

```
obj = Classy()
```

```
obj.method()
```

Il codice viene emesso:

```
method
```

Si noti il modo in cui abbiamo creato l'oggetto: abbiamo **trattato il nome della classe come una funzione**, restituendo un oggetto della classe appena istanziato.

Se si desidera che il metodo accetti parametri diversi da self, è necessario:

- inserirli dopo self nella definizione del metodo;
- consegnarli durante l'invocazione senza specificare self (come in precedenza)

```
class Classy:  
    def method(self, par):  
        print("method:", par)
```

```
obj = Classy()  
obj.method(1)  
obj.method(2)  
obj.method(3)
```

The code outputs:

```
method: 1  
method: 2  
method: 3
```

Metodi in dettaglio: continua

Il parametro self viene utilizzato **per ottenere l'accesso alle variabili di istanza e di classe dell'oggetto**.

L'esempio mostra entrambi i modi di utilizzare il self:

```
class Classy:  
    varia = 2  
    def method(self):  
        print(self.varia, self.var)
```

```
obj = Classy()  
obj.var = 3  
obj.method()
```

The code outputs:

2 3

Il parametro self viene utilizzato anche **per invocare altri metodi dell'oggetto/classe dall'interno della classe**.

```
class Classy:  
    def other(self):  
        print("other")  
  
    def method(self):  
        print("method")  
        self.other()
```

```
obj = Classy()  
obj.method()
```

The code outputs:
method
other

Metodi in dettaglio: continua

Se si dà un nome a un metodo come questo: `__init__`, non sarà un metodo normale, ma un **costruttore**.

Se una classe ha un costruttore, questo viene invocato automaticamente e implicitamente quando l'oggetto della classe viene istanziato.

Il costruttore:

- è **obbligato ad avere il parametro** `self` (viene impostato automaticamente, come al solito);
- **può (ma non è necessario) avere più parametri** del solo `self`; se ciò accade, il modo in cui il nome della classe viene usato per creare l'oggetto deve riflettere la definizione `__init__`;
- **può essere usato per impostare l'oggetto**, cioè per inizializzare correttamente il suo stato interno, creare variabili di istanza, istanziare altri oggetti se è necessaria la loro esistenza, ecc.

L'esempio mostra un costruttore molto semplice.

```
class Classy:  
    def __init__(self, value):  
        self.var = value
```

Eseguirlo. Il codice viene visualizzato:

object

Si noti che il costruttore:

- **non può restituire un valore**, poiché è progettato per restituire un oggetto appena creato e nient'altro;
- **non può essere invocato direttamente né dall'oggetto né dall'interno della classe** (è possibile invocare un costruttore da una qualsiasi sottoclasse dell'oggetto, ma di questo argomento parleremo più avanti).

Metodi in dettaglio: continua

Poiché `__init__` è un metodo e un metodo è una funzione, si possono fare gli stessi trucchi con i costruttori/metodi che si fanno con le normali funzioni.

L'esempio nell'editor mostra come definire un costruttore con un valore di argomento predefinito. Provatelo.

The code outputs:

object

None

uscita

Tutto ciò che abbiamo detto sulla **manipolazione dei nomi delle proprietà** si applica anche ai nomi dei metodi: un metodo il cui nome inizia con `__` è (parzialmente) nascosto.

L'esempio mostra questo effetto:

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("hidden")

obj = Classy()
obj.visible()
try:
    obj.__hidden()
except:
    print("failed")
obj._Classy__hidden()
```

The code outputs:

```
visible
failed
hidden
```

Eseguire il programma e testarlo.

```
class Classy:  
    def __init__(self, value = None):  
        self.var = value
```

```
obj_1 = Classy("object")  
obj_2 = Classy()
```

```
print(obj_1.var)  
print(obj_2.var)
```

La vita interna di classi e oggetti

Ogni classe e ogni oggetto Python è dotato di una serie di attributi utili che possono essere utilizzati per esaminare le sue capacità.

Una di queste la conoscete già: è la proprietà `__dict__`.

Osserviamo come tratta i metodi: guardiamo il codice nell'editor.

Eseguitelo per vedere cosa produce. Controllare attentamente l'output.

Trovare tutti i metodi e gli attributi definiti. Individuare il contesto in cui esistono: all'interno dell'oggetto o della classe.

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2
    def method(self):
        pass
    def __hidden(self):
        pass
obj = Classy()
print(obj.__dict__)
print(Classy.__dict__)
```

La vita interna di classi e oggetti: continua

`__dict__` è un dizionario. Un'altra proprietà built-in degna di nota è `__name__`, che è una stringa.

La proprietà contiene **il nome della classe**. Non è nulla di particolare, solo una stringa.

Nota: l'attributo `__name__` è assente dall'oggetto, **esiste solo all'interno delle classi**.

Se si vuole **trovare la classe di un particolare oggetto**, si può usare una funzione chiamata `type()`, che è in grado (tra le altre cose) di trovare una classe che è stata usata per istanziare qualsiasi oggetto.

Guardate il codice nell'editor, eseguitelo e verificate voi stessi.

```
class Classy:
    pass
print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

Output:

```
Classy
Classy
```

Si noti che un'affermazione come questa:

```
print(obj.__name__)
```

causerà un errore.

La vita interna di classi e oggetti: continua

Anche `__module__` è una stringa: **memorizza il nome del modulo che contiene la definizione della classe.**

Verifichiamo: eseguiamo il codice nell'editor.

```
class Classy:
    pass
print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

Output:

```
__main__
__main__
```

Come si sa, qualsiasi modulo chiamato `__main__` non è un modulo, ma il **file in esecuzione**.

La vita interna di classi e oggetti: continua

`__bases__` è una tupla. La **tupla contiene le classi** (non i nomi delle classi) che sono superclassi dirette della classe.

L'ordine è lo stesso utilizzato all'interno della definizione della classe.

Mostreremo solo un esempio molto elementare, per evidenziare **il funzionamento dell'ereditarietà**.

Inoltre, vi mostreremo come utilizzare questo attributo quando discuteremo degli aspetti oggettivi delle eccezioni.

Nota: **solo le classi hanno questo attributo**, gli oggetti no.

Abbiamo definito una funzione chiamata `printbases()`, pensata per presentare in modo chiaro il contenuto della tupla.

Guardate il codice nell'editor. Analizzatelo ed eseguitelo.

```
class SuperOne:
    pass
class SuperTwo:
    pass
class Sub(SuperOne, SuperTwo):
    pass
def printBases(cls):
    print('( ', end='')
    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')
printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

Verrà visualizzato:

(oggetto)

(oggetto)

(SuperOne SuperTwo)

Nota: **una classe senza superclassi esplicite punta a object** (una classe Python predefinita) come suo antenato diretto.

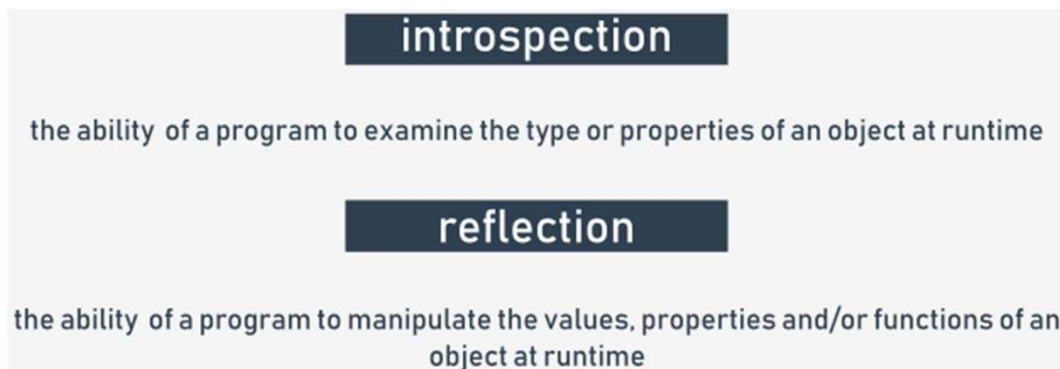
Riflessione e introspezione

Tutti questi mezzi permettono al programmatore Python di svolgere due importanti attività specifiche di molti linguaggi obiettivo. Esse sono:

introspezione, ovvero la capacità di un programma di esaminare il tipo o le proprietà di un oggetto in fase di esecuzione;

reflection, che fa un ulteriore passo avanti ed è la capacità di un programma di manipolare i valori, le proprietà e/o le funzioni di un oggetto in fase di esecuzione.

In altre parole, non è necessario conoscere la definizione completa di classe/oggetto per manipolare l'oggetto, poiché l'oggetto e/o la sua classe contengono i metadati che consentono di riconoscerne le caratteristiche durante l'esecuzione del programma.



Indagine sulle classi

Cosa si può scoprire sulle classi in Python? La risposta è semplice: tutto.

Sia la riflessione che l'introspezione consentono al programmatore di fare qualsiasi cosa con qualsiasi oggetto, indipendentemente dalla sua provenienza.

Analizzare il codice nell'editor.

```
class MyClass:  
    pass
```

```
obj = MyClass()  
obj.a = 1  
obj.b = 2  
obj.i = 3  
obj.ireal = 3.5  
obj.integer = 4  
obj.z = 5
```

```
def incIntsI(obj):  
    for name in obj.__dict__.keys():  
        if name.startswith('i'):  
            val = getattr(obj, name)  
            if isinstance(val, int):  
                setattr(obj, name, val + 1)
```

```
print(obj.__dict__)  
incIntsI(obj)  
print(obj.__dict__)
```

La funzione `incIntsI()` prende un oggetto di qualsiasi classe, ne analizza il contenuto per trovare tutti gli attributi interi con nomi che iniziano per `i` e li incrementa di uno.

Impossibile? Niente affatto!

Ecco come funziona:

riga 1: definire una classe molto semplice...

righe da 5 a 11: ... e riempirlo con alcuni attributi;

riga 14: questa è la nostra funzione!

riga 15: scansiona l'attributo `__dict__`, cercando tutti i nomi degli attributi;

riga 16: se un nome inizia per `i`...

riga 17: ... usare la funzione `getattr()` per ottenere il suo valore corrente; nota: `getattr()` prende due argomenti: un oggetto e il nome della sua proprietà (come stringa) e restituisce il valore dell'attributo corrente;

riga 18: controlla se il valore è di tipo intero e utilizza la funzione `isinstance()` a tale scopo (ne parleremo più avanti);

riga 19: se la verifica è andata a buon fine, incrementare il valore della proprietà utilizzando la funzione `setattr()`; la funzione accetta tre argomenti: un oggetto, il nome della proprietà (come stringa) e il nuovo valore della proprietà.

Output:

```
{'a': 1, 'intero': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
```

```
{'a': 1, 'intero': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

Punti di forza

1. Un metodo è una funzione incorporata in una classe. Il primo (o unico) parametro di ogni metodo è solitamente chiamato `self`, che ha lo scopo di identificare l'oggetto per il quale il metodo viene invocato, al fine di accedere alle proprietà dell'oggetto o invocarne i metodi.
2. Se una classe contiene un **costruttore** (un metodo chiamato `__init__`), questo non può restituire alcun valore e non può essere invocato direttamente.
3. Tutte le classi (ma non gli oggetti) contengono una proprietà chiamata `__name__`, che memorizza il nome della classe. Inoltre, una proprietà chiamata `__module__` memorizza il nome del modulo in cui la classe è stata dichiarata, mentre la proprietà chiamata `__bases__` è una tupla contenente le superclassi di una classe.

Ad esempio:

```
class Sample:
    def __init__(self):
        self.name = Sample.__name__
    def myself(self):
        print("My name is " + self.name + " living in a " + Sample.__module__)
obj = Sample()
obj.myself()
```

outputs:

My name is Sample living in a `__main__`

Esercizio 1

La dichiarazione della classe Snake è riportata di seguito. Arricchire la classe con un metodo chiamato `increment()`, aggiungendo 1 alla proprietà `victims`.

```
class Snake:  
    def __init__(self):  
        self.victims = 0
```

Esercizio 2

Ridefinire il costruttore della classe Snake in modo che abbia un parametro per inizializzare il campo `victims` con un valore passato all'oggetto durante la costruzione.

Esercizio 3

È possibile prevedere l'output del seguente codice?

```
class Snake:  
    pass
```

```
class Python(Snake):  
    pass
```

```
print(Python.__name__, 'is a', Snake.__name__)  
print(Python.__bases__[0].__name__, 'can be a', Python.__name__)
```


Esercizio 1

La dichiarazione della classe Snake è riportata di seguito. Arricchire la classe con un metodo chiamato `increment()`, aggiungendo 1 alla proprietà

```
class Snake:  
    def __init__(self):  
        self.victims = 0
```

Soluzione

```
class Snake:  
    def __init__(self):  
        self.victims = 0  
  
    def increment(self):  
        self.victims += 1
```

Esercizio 2

Ridefinire il costruttore della classe Snake in modo che abbia un parametro per inizializzare il campo victims con un valore passato all'oggetto durante la costruzione.

Soluzione

```
class Snake:
```

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

Esercizio 3

Soluzione

Python is a Snake

Snake can be a Python

LAB-39

Tempo stimato

30-60 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le capacità dello studente nel definire le classi da zero;
- definire e utilizzare le variabili di istanza;
- definire e utilizzare i metodi.

Scenario

Abbiamo bisogno di una classe in grado di contare i secondi. Facile? Non tanto quanto si possa pensare, perché avremo delle aspettative specifiche.

Leggete con attenzione perché la classe che state per scrivere sarà utilizzata per lanciare razzi che trasportano missioni internazionali su Marte. È una grande responsabilità. Contiamo su di voi!

La classe si chiamerà Timer. Il suo costruttore accetta tre argomenti che rappresentano le **ore** (un valore compreso nell'intervallo [0..23] - useremo l'ora militare), i **minuti** (dall'intervallo [0..59]) e i **secondi** (dall'intervallo [0..59]).

Zero è il valore predefinito per tutti i parametri di cui sopra. Non è necessario eseguire alcun controllo di convalida.

La classe stessa dovrebbe fornire le seguenti strutture:

- Gli oggetti della classe devono essere "stampabili", cioè devono essere in grado di convertirsi implicitamente in stringhe della forma seguente: "hh:mm:ss", con l'aggiunta di zeri iniziali quando uno qualsiasi dei valori è inferiore a 10;
- la classe dovrebbe essere dotata di metodi senza parametri chiamati `next_second()` e `previous_second()`, che incrementano il tempo memorizzato all'interno degli oggetti rispettivamente di +1/-1 secondo.

Utilizzate i seguenti suggerimenti:

- tutte le proprietà dell'oggetto devono essere private;
- considerare di scrivere una funzione separata (non un metodo!) per formattare la stringa dell'ora.

Completare il modello che abbiamo fornito nell'editor. Eseguite il vostro codice e verificate se l'output è uguale al nostro.

Risultato previsto

23:59:59

00:00:00

23:59:59

```
class Timer:
    def __init__( ??? ):
        # Write code here

    def __str__(self):
        # Write code here

    def next_second(self):
        # Write code here

    def prev_second(self):
        # Write code here

timer = Timer(23, 59, 59)
print(timer)
timer.next_second()
print(timer)
timer.prev_second()
print(timer)
```

LAB-40

Tempo stimato

30-60 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le capacità dello studente nel definire le classi da zero;
- definire e utilizzare le variabili di istanza;
- definire e utilizzare i metodi.

Scenario

Il vostro compito è quello di implementare una classe chiamata Weeker. Sì, i vostri occhi non vi ingannano: questo nome deriva dal fatto che gli oggetti di questa classe saranno in grado di memorizzare e manipolare i giorni della settimana.

Il costruttore della classe accetta un argomento: una stringa. La stringa rappresenta il nome del giorno della settimana e gli unici valori accettabili devono provenire dal seguente insieme:

lun mar mer gio ven sab dom

L'invocazione del costruttore con un argomento al di fuori di questo insieme dovrebbe sollevare l'eccezione `WeekDayError` (definitela voi stessi; non preoccupatevi, parleremo presto della natura oggettiva delle eccezioni).

La classe deve fornire le seguenti funzioni:

- Gli oggetti della classe devono essere "stampabili", cioè devono essere in grado di convertirsi implicitamente in stringhe della stessa forma degli argomenti del costruttore;
- la classe dovrebbe essere dotata di metodi a un parametro chiamati `add_days(n)` e `subtract_days(n)`, con **n** che è un numero intero e che aggiornano il giorno della settimana memorizzato all'interno dell'oggetto nel modo che riflette il cambiamento della data del numero di giorni indicato, in avanti o indietro.
- tutte le proprietà dell'oggetto devono essere private;

Completate il modello che vi abbiamo fornito nell'editor, eseguite il vostro codice e verificate se il vostro risultato è uguale al nostro.

Risultato previsto

Lun

Mar

Sole

Mi dispiace, non posso soddisfare la sua richiesta.

```
class WeekDayError(Exception):
```

```
    pass
```

```
class Weeker:
```

```
    # Write code here.
```

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

```
        # Write code here
```

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

```
        # Write code here.
```

```
    def add_days(self, n):
```

```
        # Write code here.
```

```
    def subtract_days(self, n):
```

```
        # Write code here.
```

```
try:
```

```
    weekday = Weeker('Mon')
```

```
    print(weekday)
```

```
    weekday.add_days(15)
```

```
    print(weekday)
```

```
    weekday.subtract_days(23)
```

```
    print(weekday)
```

```
    weekday = Weeker('Monday')
```

```
except WeekDayError:
```

```
    print("Sorry, I can't serve your request.")
```


LAB-41

Tempo stimato

30-60 minuti

Livello di difficoltà

Facile/Medio

Obiettivi

- migliorare le capacità dello studente nel definire le classi da zero;
- definire e utilizzare le variabili di istanza;
- definire e utilizzare i metodi.

Scenario

Visitiamo un luogo molto speciale: un piano con il sistema di coordinate cartesiane (potete approfondire questo concetto qui: https://en.wikipedia.org/wiki/Cartesian_coordinate_system).

Ogni punto situato sul piano può essere descritto come una coppia di coordinate chiamate abitualmente **x** e **y**. Ci aspettiamo che siate in grado di scrivere una classe Python che memorizzi entrambe le coordinate come numeri float. Inoltre, vogliamo che gli oggetti di questa classe valutino le distanze tra due punti qualsiasi del piano.

Il compito è piuttosto facile se si utilizza la funzione `hypot()` (disponibile attraverso il modulo di *matematica*) che valuta la lunghezza dell'ipotenusa di un triangolo rettangolo (maggiori dettagli qui:

<https://en.wikipedia.org/wiki/Hypotenuse>) e qui: <https://docs.python.org/3.7/library/math.html#trigonometric-functions>).

Ecco come immaginiamo la classe:

- si chiama Punto;
- il suo costruttore accetta due argomenti (rispettivamente **x** e **y**), entrambi predefiniti a zero;
- tutte le proprietà devono essere private;
- la classe contiene due metodi senza parametri, chiamati `getx()` e `gety()`, che restituiscono ciascuna delle due coordinate (le coordinate sono memorizzate privatamente, quindi non è possibile accedervi direttamente dall'interno dell'oggetto);
- la classe fornisce un metodo chiamato `distanza_da_xy(x,y)`, che calcola e restituisce la distanza tra il punto memorizzato all'interno dell'oggetto e l'altro punto dato come coppia di float;
- la classe fornisce un metodo chiamato `distanza_da_punto(punto)`, che calcola la distanza (come il metodo precedente), ma la posizione dell'altro punto è data da un altro oggetto della classe Point;

Completate il modello che vi abbiamo fornito nell'editor, eseguite il vostro codice e verificate se il vostro risultato è uguale al nostro.

Risultato previsto

1.4142135623730951

1.4142135623730951

```
import math
class Point:
    def __init__(self, x=0.0, y=0.0):
        # Write code here

    def getx(self):
        # Write code here

    def gety(self):
        # Write code here

    def distance_from_xy(self, x, y):
        # Write code here

    def distance_from_point(self, point):
        # Write code here

point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))
```

LAB-42

Tempo stimato

30-60 minuti

Livello di difficoltà

Medio

Obiettivi

- migliorare le capacità dello studente nel definire le classi da zero;
- utilizzando la composizione.

Scenario

Ora incorporeremo la classe Punto (vedi Laboratorio 3.4.1.14) all'interno di un'altra classe. Inoltre, metteremo tre punti in una classe, il che ci permetterà di definire un triangolo. Come possiamo farlo?

- La nuova classe si chiamerà Triangolo e questo è l'elenco delle nostre aspettative:
- il costruttore accetta tre argomenti, tutti oggetti della classe Point;
- i punti sono memorizzati all'interno dell'oggetto come elenco privato;

la classe fornisce un metodo senza parametri chiamato perimetro(), che calcola il perimetro del triangolo descritto dai tre punti; il perimetro è la somma delle lunghezze di tutte le gambe (lo citiamo per dovere di cronaca, anche se siamo sicuri che lo conoscete perfettamente).

Completate il modello che vi abbiamo fornito nell'editor. Eseguite il vostro codice e verificate se il perimetro valutato è uguale al nostro.

Risultato previsto

3.414213562373095

```
import math
```

```
class Point:
```

```
    def __init__(self, x=0.0, y=0.0):  
        self.__x = x  
        self.__y = y
```

```
class Triangle:
```

```
    def __init__(self, vertice1, vertice2, vertice3):  
        #  
        # Write code here  
        #
```

```
    def perimeter(self):  
        #  
        # Write code here  
        #
```

```
triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))  
print(triangle.perimeter())
```

Eredità: perché e come?

Prima di iniziare a parlare di ereditarietà, vogliamo presentare un nuovo, pratico meccanismo utilizzato dalle classi e dagli oggetti di Python: si tratta del **modo in cui l'oggetto è in grado di presentarsi**.

Cominciamo con un esempio. Guardate il codice nell'editor.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy
sun = Star("Sun", "Milky Way")
print(sun)
```

Il programma stampa una sola riga di testo, che nel nostro caso è questa:

```
<__main__.Star object at 0x7f1074cc7c50>
```

Se si esegue lo stesso codice sul proprio computer, si vedrà qualcosa di molto simile, anche se il numero esadecimale (la sottostringa che inizia con 0x) sarà diverso, poiché si tratta di un identificatore interno di oggetto usato da Python ed è improbabile che appaia lo stesso quando lo stesso codice viene eseguito in un ambiente diverso.

Come si può notare, la stampa qui riportata non è molto utile e qualcosa di più specifico, o semplicemente più bello, potrebbe essere preferibile.

Fortunatamente, Python offre proprio questa funzione.

Eredità: perché e come?

Quando Python ha bisogno di una classe/oggetto da presentare come stringa (mettere un oggetto come argomento nell'invocazione della funzione `print()` soddisfa questa condizione), cerca di invocare un metodo chiamato `__str__()` dall'oggetto e di usare la stringa che restituisce.

Il metodo predefinito `__str__()` restituisce la stringa precedente - brutto e poco informativo. È possibile cambiarlo semplicemente **definendo un proprio metodo con questo nome**.

L'abbiamo appena fatto: guardate il codice nell'editor.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy
    def __str__(self):
        return self.name + ' in ' + self.galaxy
```

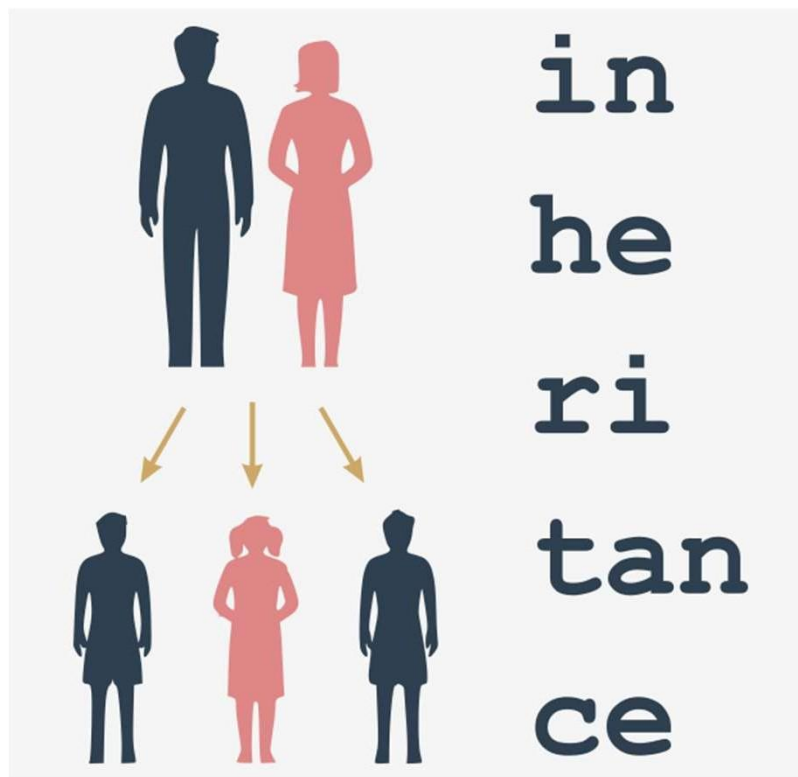
```
sun = Star("Sun", "Milky Way")
print(sun)
```

Questo nuovo metodo `__str__()` crea una stringa composta dai nomi delle stelle e delle galassie - niente di speciale, ma ora i risultati di stampa hanno un aspetto migliore, non è vero?

Riuscite a indovinare l'output? Eseguite il codice per verificare se avete indovinato.

Eredità: perché e come?

Il termine eredità è più antico della programmazione informatica e descrive la pratica comune di passare diversi beni da una persona a un'altra alla sua morte. Il termine, se riferito alla programmazione informatica, ha un significato completamente diverso.



L'ereditarietà è una pratica comune (nella programmazione a oggetti) che consiste nel **passare attributi e metodi dalla superclasse (definita ed esistente) a una nuova classe, chiamata sottoclasse.**

In altre parole, l'ereditarietà è **un modo per costruire una nuova classe, non da zero, ma utilizzando un repertorio di tratti già definito.** La nuova classe eredita (e questa è la chiave) tutti gli equipaggiamenti già esistenti, ma è in grado di aggiungerne di nuovi, se necessario.

Grazie a ciò, è possibile **costruire classi più specializzate (più concrete)** utilizzando alcuni insiemi di regole e comportamenti generali predefiniti.

Il fattore più importante del processo è la relazione tra la superclasse e tutte le sue sottoclassi (nota: se B è una sottoclasse di A e C è una sottoclasse di B , significa anche che C è una sottoclasse di A , poiché la relazione è completamente transitiva).

Qui viene presentato un esempio molto semplice di **ereditarietà a due livelli:**

```
class Vehicle:
```

```
    pass
```

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

```
    pass
```

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

```
    pass
```

Tutte le classi presentate sono vuote per il momento, poiché mostreremo come funzionano le relazioni reciproche tra le superclassi e le sottoclassi. Le riempiremo presto di contenuti.

Possiamo dire che:

- La classe Veicolo è la superclasse delle classi Veicolo terrestre e Veicolo cingolato;
- La classe LandVehicle è una sottoclasse di Vehicle e allo stesso tempo una superclasse di TrackedVehicle;
- La classe TrackedVehicle è una sottoclasse delle classi Vehicle e LandVehicle.

Lo capiamo semplicemente leggendo il codice (in altre parole, lo sappiamo perché lo vediamo).

Python conosce la stessa cosa? È possibile chiedere a Python informazioni in merito? Sì, è possibile.

Ereditarietà: `issubclass()`

Python offre una funzione in grado di **identificare una relazione tra due classi** e, sebbene la sua diagnosi non sia complessa, è in grado di **verificare se una particolare classe è una sottoclasse di qualsiasi altra classe**.

Ecco come si presenta:

```
issubclass(ClasseUno, ClasseDue)
```

La funzione restituisce `True` se `ClassOne` è una sottoclasse di `ClassTwo` e `False` altrimenti.

Vediamolo in azione: potrebbe sorprendervi. Guardate il codice nell'editor. Leggetelo con attenzione.

```
class Vehicle:
    pass
class LandVehicle(Vehicle):
    pass
class TrackedVehicle(LandVehicle):
    pass
for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Ci sono due cicli annidati. Il loro scopo è **controllare tutte le possibili coppie ordinate di classi e stampare i risultati del controllo per determinare se la coppia corrisponde alla relazione sottoclasse-superclasse**.

Eseguire il codice. Il programma produce il seguente output:

```
True False False
True True False
True True True
```

Rendiamo il risultato più leggibile:

↓ è una sottoclasse di →	Veicolo	Veicolo terrestre	Veicolo cingolato
Veicolo	Vero	Falso	Falso
Veicolo terrestre	Vero	Vero	Falso
Veicolo cingolato	Vero	Vero	Vero

C'è un'osservazione importante da fare: **ogni classe è considerata una sottoclasse di se stessa.**