

Programmazione Avanzata

Python

Francesco Abate

Introduzione

È importante sapere, prima di iniziare, che è necessario installare Python e che utilizzeremo come ambiente di sviluppo PyCharm.

Python è un linguaggio interpretato, quindi le istruzioni vengono eseguite così come descritte nel codice sorgente, il quale sarà un file con estensione “.py”. L’installazione di Python, in realtà, non contiene solo un interprete, bensì anche un compilatore: Python compila i moduli utilizzati nel caso si avvii il programma, in modo da generare una nuova rappresentazione del codice a basso livello detta bytecode, la quale viene letta ed eseguita dalla Python Virtual Machine (PVM).

Convenzioni e identificatori

Python adotta delle convenzioni, le quali sono le seguenti:

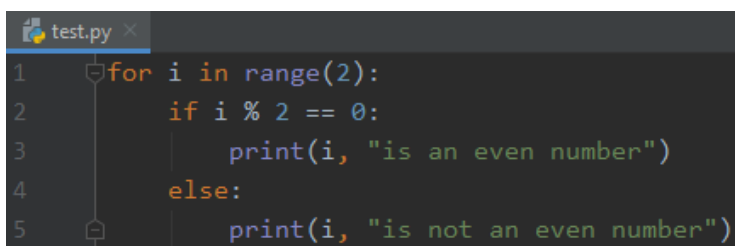
- Nomi di funzioni, metodi e di variabili iniziano sempre con la lettera minuscola;
- Nomi di classi iniziano con la lettera maiuscola;
- In ogni caso si utilizza la notazione CamelCase (ogni parola comincia con la maiuscola, ad esempio testDomain);
- Nomi di costanti sono scritti interamente in maiuscolo.

Python stabilisce alcune regole riguardo i nomi utilizzati per gli identificatori (nomi di variabili, di costanti, di funzioni, ecc.), quali sono:

- Gli identificatori sono case sensitive;
- Gli identificatori possono essere composti da lettere, numeri e underscore;
- Un identificatore non può iniziare con un numero e non può essere una delle parole riservate.

Indentazione del codice

In Python un blocco di codice non è delimitato da parole chiavi o da parentesi graffe, bensì dal simbolo di due punti e dall’indentazione del codice.



```
test.py x
1  for i in range(2):
2      if i % 2 == 0:
3          print(i, "is an even number")
4      else:
5          print(i, "is not an even number")
```

È importante sapere che l’indentazione deve essere la stessa per tutto il blocco, per cui il numero di caratteri di spaziatura è significativo. Ovviamente utilizzare gli spazi al posto della tabulazione è un errore. L’indentazione è un requisito e non una questione di stile, quindi tutti i programmi scritti in Python avranno lo stesso aspetto.

Come ormai avremmo notato, Python non utilizza i punti e virgola per terminare le istruzioni, bensì è sufficiente andare su una nuova linea. L’unico caso in cui è necessario l’utilizzo dei punti e virgola è nel caso si vogliano inserire molteplici istruzioni su una sola riga, ma è un pessimo stile di programmazione ed è altamente sconsigliato.

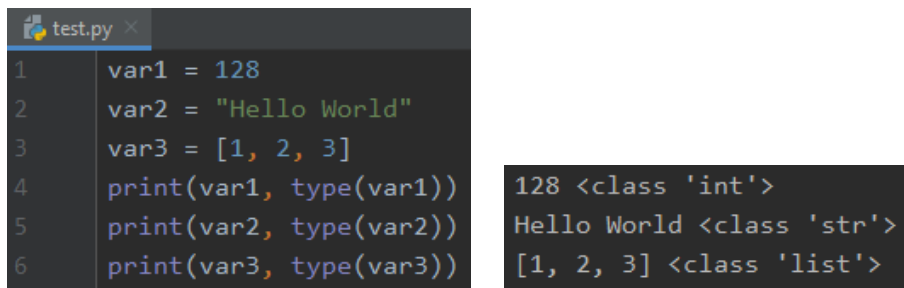
Oggetti, core data type ed etichette

Quando un programma viene eseguito, Python genera delle strutture dati chiamate oggetti, sulle quali basa tutto il processo di elaborazione. Tali oggetti vengono conservati nella memoria RAM del computer, in modo da poter esser richiamati quando il programma fa riferimento ad essi: nel caso non servano più, un particolare meccanismo chiamato garbage collector provvederà a liberare la memoria da essi occupata.

Python offre un insieme di tipi built-in chiamato core data type, divisibile in quattro categorie:

- Numeri: interi (int), floating point (float), booleani (bool), complessi (complex);
- Insiemi: set e frozenset;
- Sequenze: stringhe (str e byte), liste (list) e tuple (tuple);
- Dizionari: dict.

I tipi del core data type sono anche detti classi: danno la possibilità di istanziare oggetti specifici di quel tipo, chiamati appunto istanze. Il tipo di un oggetto è ottenibile tramite la funzione `type(var)`.



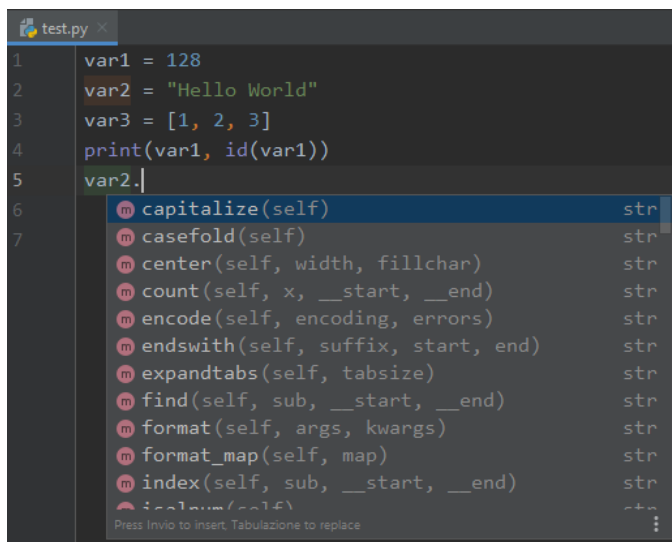
The image shows a code editor window titled 'test.py' with the following code:

```
1 var1 = 128
2 var2 = "Hello World"
3 var3 = [1, 2, 3]
4 print(var1, type(var1))
5 print(var2, type(var2))
6 print(var3, type(var3))
```

To the right of the code editor, the output of the script is displayed:

```
128 <class 'int'>
Hello World <class 'str'>
[1, 2, 3] <class 'list'>
```

Altri elementi caratteristici degli oggetti sono l'identità e gli attributi: l'identità distingue in maniera assolutamente univoca un determinato oggetto, mentre gli attributi sono strettamente legati al tipo di oggetto. È possibile accedere all'identità tramite la funzione `id(var)`, mentre è possibile accedere agli attributi tramite il delimitatore punto.



The image shows a code editor window titled 'test.py' with the following code:

```
1 var1 = 128
2 var2 = "Hello World"
3 var3 = [1, 2, 3]
4 print(var1, id(var1))
5 var2.
```

Below the code editor, a list of attributes for the string object 'Hello World' is displayed:

- capitalize(self) str
- casefold(self) str
- center(self, width, fillchar) str
- count(self, x, __start, __end) str
- encode(self, encoding, errors) str
- endswith(self, suffix, start, end) str
- expandtabs(self, tabsize) str
- find(self, sub, __start, __end) str
- format(self, args, kwargs) str
- format_map(self, map) str
- index(self, sub, __start, __end) str
- isalnum(self) str

Precisando, spesso si dice che ogni cosa in Python è un oggetto. In realtà, come anticipato poco fa, essi vengono generati quando il programma è in esecuzione. Gli oggetti vengono “puntati” dalle etichette, le quali sono degli identificativi con il quale si farà riferimento agli oggetti (comunemente

vengono anche chiamate “variabili”). L’etichetta “punta” ad un oggetto, quindi è un identificativo che permette di accedere in qualche modo ad un oggetto.

```
test.py x
1 var = 44
2 var = 99
3 print(var)
```

Nell’esempio soprastante: var punta all’oggetto 44 di tipo int; var punta nel successivo rigo all’oggetto 99 di tipo int, il quale verrà mandato in output grazie alla funzione print(var). L’oggetto 44 non è referenziato da nessun’altra etichetta, quindi verrà automaticamente cancellato dalla memoria. Un’operazione di assegnamento ad un’etichetta già esistente non modifica mai l’oggetto a cui essa precedentemente puntava: ciò è sempre vero, anche se quell’oggetto è mutabile.

Mutabilità ed immutabilità

Gli oggetti, come sappiamo, sono caratterizzati da una classe: una classe è detta mutabile se il valore dell’oggetto può cambiare, mentre è detta immutabile se il valore dell’oggetto non può essere modificato in seguito all’inizializzazione.

Gli oggetti mutabili possono essere modificati tramite i loro metodi e tramite gli augmented assignment.

```
test.py x
1 myList = [1, 2, 3]
2 print(id(myList)) # output: 25642488
3 myList.append(4)
4 print(id(myList)) # output: 25642488
5 myList += [5]
6 print(id(myList)) # output: 25642488
7 myList *= 2
8 print(id(myList)) # output: 25642488
9 print(myList) # output: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Gli oggetti immutabili, invece, non hanno metodi che consentono di modificarli: neppure gli augmented assignment possono farlo! Nel caso si usino questi ultimi, si creeranno nuovi oggetti!

```
test.py x
1 myTuple = (1, 2, 3)
2 print(id(myTuple)) # output: 10171576
3 myTuple += (4, 5)
4 print(id(myTuple)) # output: 9761008 -> it's a new object!
5 print(myTuple) # output: (1, 2, 3, 4, 5)
```

Consideriamo ora il caso in cui un oggetto immutabile contenga un oggetto mutabile, come una tupla contenente un set ed una lista (la tupla è immutabile, il set e la lista sono mutabili):

```
test.py x
1 myTuple = ({1, 2}, [1, 2, 3])
2 print(id(myTuple))
3 print(id(myTuple[0]), id(myTuple[1]))
4 # output:
5 # tuple id: 29046704
6 # set id: 29015376
7 # list id: 28198392
```

In tal caso, l'immutabilità consiste nel non poter modificare i riferimenti ad una tupla, ovvero nel non poter effettuare nuovi assegnamenti.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

La classe bool, int e float

La classe bool è utilizzata per rappresentare i valori booleani. Il costruttore è bool(value), dove value può assumere uno dei due valori: True o False; nel caso non venga passato alcun argomento, viene ritornato False di default.

La classe int è usata per rappresentare i valori interi di grandezza arbitraria. Il costruttore è int(value), dove value è un qualsiasi numero intero; nel caso non venga passato alcun argomento, viene ritornato 0 di default. È possibile creare interi partendo da stringhe, le quali rappresentano numeri in qualsiasi base tra 2 e 35 utilizzando il seguente costruttore: int(value, base), dove value è una stringa e base è la base da utilizzare per la conversione.

```
test.py x
1 i = int("23", base=4)
```

La classe float è utilizzata per rappresentare i valori floating point in doppia precisione. Il costruttore è float(value), dove value è un qualsiasi numero in doppia precisione; nel caso non venga passato alcun argomento, viene ritornato 0.0 di default.

Oggetti iterabili

Python permette l'utilizzo e la gestione di oggetti iterabili, i quali non sono altro che oggetti contenenti una collezione di altri oggetti: un esempio sono le liste e le tuple. Gli oggetti iterabili sono anche detti sequenze o collezioni. Se un oggetto sequenza possiede N oggetti, la posizione degli elementi va da 0 ad N-1.

Sugli oggetti iterabili è possibile effettuare apposite operazioni, quali sono lo spaccettamento, l'iterazione e il test di appartenenza.

Lo spaccettamento permette di associare delle variabili a degli elementi di un oggetto iterabile in maniera molto rapida, come nel seguente esempio:

```
test.py x
1 a, b, c = (1, 2, 3)
2 print(a) # a contains 1
```

È importante sapere che il dizionario (il quale vedremo successivamente) non ha memoria riguardo l'ordine di inserimento, quindi lo spaccettamento avverrà in ordine casuale.

L'iterazione, invece, è la classica operazione che permette di "scansionare" uno per uno tutti gli oggetti di un oggetto sequenza, di solito realizzata tramite un for.

Il test di appartenenza, invece, consente di verificare se degli elementi sono presenti o meno in un oggetto sequenza. Sarà più chiaro con i seguenti esempi:

```
test.py x
1 # First test
2 if "orl" in "World":
3     print("Success!")
4 else:
5     print("Failure :(")
6 # Second test
7 if (1, 2) in ["a", "b", 1, (1, 2)]:
8     print("Success!")
9 else:
10    print("Failure :(")
11 # Third test
12 if "Pippo" not in {"age": 23, "name": "Pippo"}:
13     print("Success!")
14 else:
15     print("Failure :(")
```

La classe list, tuple, str, set, frozenset e dict

La classe list non è altro che un oggetto contenente una sequenza di puntatori ad oggetti ed utilizza i caratteri [] come delimitatori. Il costruttore è list(iterable), dove iterable è un oggetto sequenza; nel caso non venga passato alcun argomento, viene ritornata una lista vuota. Le liste hanno la capacità di espandersi e contrarsi secondo le necessità del programmatore, quindi offre metodi riguardo l'inserimento e la rimozione degli oggetti. È possibile accedere ad un elemento della lista utilizzando la seguente sintassi: myList[index].

[Clicca qui per i metodi della classe list.](#)

La classe tuple non è altro che la versione immutabile della lista ed utilizza i caratteri () come delimitatori. Il costruttore è tuple(iterable), dove iterable è un oggetto sequenza; nel caso non venga passato alcun argomento, viene ritornata una tupla vuota. Attenzione: una tupla con un

unico elemento è indicata con (item,). È possibile accedere ad un elemento della tupla utilizzando la seguente sintassi: myTuple[index].

[Clicca qui per i metodi della classe tuple.](#)

La classe str offre la possibilità di creare stringhe, nonché sequenze di caratteri che possono essere racchiuse tra apici singoli o doppi. Se la stringa è posta su un'unica linea basta un solo apice, mentre se è multi-line si necessita di tre apici.

[Clicca qui per i metodi della classe str.](#)

La classe set rappresenta una collezione di elementi immutabili senza duplicati e senza un particolare ordine ed utilizza i caratteri { } come delimitatori. Il costruttore è set(iterable), dove iterable è un oggetto sequenza; nel caso non venga passato alcun argomento, viene ritornato un set vuoto. La classe frozenset, invece, offre una versione immutabile del tipo set ed utilizza gli stessi metodi della classe set.

[Clicca qui per i metodi della classe set.](#)

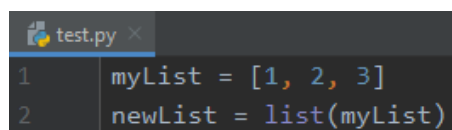
La classe dict rappresenta un dizionario, quindi un insieme di coppie chiave e valore, dove le chiavi devono essere univoche tra loro. Il costruttore è dict(pairs), dove pairs è una lista di coppie (key, value); nel caso non venga passato alcun argomento, viene ritornato un dict vuoto. Tale classe utilizza i caratteri { } come delimitatori. Nel caso non si utilizzi il costruttore, è importante notare che le coppie devono essere dichiarate nel seguente modo: myDict = {"key": "value"}.

[Clicca qui per i metodi della classe dict.](#)

Shallow copy e deep copy (collezioni)

Come ben sappiamo, gli assegnamenti in Python non creano reali copie di oggetti, bensì fanno puntare le variabili a quel determinato oggetto. Per gli oggetti mutabili o per le collezioni di oggetti mutabili si potrebbe star cercando un modo per creare copie reali o cloni di tali oggetti. Essenzialmente, alcune volte si potrebbero volere delle copie di oggetti in modo da non intaccare gli originali.

Iniziamo a vedere come copiare le collezioni: le collezioni mutabili come lists, dicts e sets possono essere copiate richiamando il loro costruttore, come nell'esempio:



```
test.py x
1 myList = [1, 2, 3]
2 newList = list(myList)
```

Questo metodo funziona solo per le collezioni built-in e crea solo shallow copies. Per le collezioni c'è un'importante differenza tra shallow copy e deep copy:

- La shallow copy applicata alle collezioni permette di creare una nuova collezione popolandola con le referenze agli oggetti contenuti dall'originaria collezione. Sostanzialmente, la shallow copy agisce ad un solo livello: il processo di copia non è ricorsivo e non creerà copie anche per gli oggetti contenuti dalla lista.
- La deep copy applicata alle collezioni effettua una copia ricorsiva degli oggetti: ciò significa che permette di creare una nuova collezione popolandola con copie di oggetti contenuti

dall'originaria collezione, effettuando una copia ricorsiva e non ad un solo livello. Ciò può essere pericoloso siccome potrebbe esserci qualche ciclo tra riferimenti e la deep copy andrebbe in loop infinito.

Nel seguente esempio vediamo un esempio di shallow copy:

```
test.py x
1 myList = [[1, 2], [4, 5], [3, 6]]
2 newList = list(myList)
3 newList.append([7, 8])
4 newList[1][0] = 10
5 print(myList)
6 print(newList)
```

```
[[1, 2], [10, 5], [3, 6]]
[[1, 2], [10, 5], [3, 6], [7, 8]]
```

Perché il 10 cambia in myList nonostante sia stato modificato solo in newList? Ebbene, sia myList[1] che newList[1] puntano alla stessa lista [4, 5], quindi qualsiasi modifica verrà registrata in qualsiasi variabile che punti ad essa: ciò perché la shallow copy ne copia, appunto, i puntatori!

Notiamo ora questo nuovo esempio, contenente una modifica diversa:

```
test.py x
1 myList = [[1, 2], [4, 5], [3, 6]]
2 newList = list(myList)
3 newList.append([7, 8])
4 newList[1] = [4, 10]
5 print(myList)
6 print(newList)
```

```
[[1, 2], [4, 5], [3, 6]]
[[1, 2], [4, 10], [3, 6], [7, 8]]
```

Come mai, questa volta, [4, 10] non è cambiato in myList? Ebbene, come possiamo vedere è stato modificato il puntatore di newList[1]: in tal modo, newList[1] punterà ad una collezione diversa rispetto a quella puntata da myList[1].

Nel seguente esempio, invece, vediamo un esempio di deep copy. Prima di procedere, è necessario importare la funzione deepcopy dal modulo copy:

```
test.py x
1 from copy import deepcopy
2 myList = [[1, 2], [4, 5], [3, 6]]
3 newList = deepcopy(myList)
4 newList.append([7, 8])
5 newList[1][0] = 10
6 print(myList)
7 print(newList)
```

```
[[1, 2], [4, 5], [3, 6]]
[[1, 2], [10, 5], [3, 6], [7, 8]]
```

Come possiamo ben vedere, la modifica viene apportata solamente in newList e non anche in myList, a differenza della shallow copy: questo perché, appunto, la deep copy ricrea tutti gli oggetti in maniera ricorsiva, a differenza della shallow copy che ne copia solo i puntatori al primo livello.

Operatori aritmetici

Addizione	+	Effettua l'addizione	a+b
Sottrazione	-	Effettua la sottrazione	a-b
Moltiplicazione	*	Effettua la moltiplicazione	a*b
Elevamento a potenza	**	Effettua l'elevamento a potenza	a**b
Divisione vera	/	Effettua la divisione con risultato float	a/b
Divisione intera	//	Effettua la divisione con parte intera (int)	a//b
Divisione per modulo	%	Effettua la divisione per modulo	a%b

Operatori logici

and	Ritorna true se entrambi gli operandi sono veri
or	Ritorna true se almeno uno degli operandi è vero
not	Ritorna false se l'operando è vero; ritorna true se l'operando è falso

Operatori di uguaglianza

is	Ritorna true se entrambe le variabili puntano allo stesso oggetto
is not	Ritorna true se le variabili non puntano allo stesso oggetto
==	Ritorna true se l'oggetto è uguale, ma non per forza il riferimento
!=	Ritorna true se l'oggetto è diverso

Operatori di confronto

<	a<b -> true se a minore di b
<=	a<=b -> true se a minore o uguale di b
>	a>b -> true se a maggiore di b
>=	a>=b -> true se a maggiore o uguale di b

Operatori di appartenenza

in	a in b -> true se a compare in b
not in	a not in b -> true se a non compare in b

Operatori di assegnamento

a = b	Assegna il riferimento di b ad a
a, b = b, a	Scambio dei valori di a e b
a, b, c = 1, 2, 3	Assegna valori in sequenza
a = 1, 2, 3	Crea una tupla di oggetti
c, d, e = a	Assegna i valori di a alle variabili scelte. Funziona se a è una collezione
a = b = 3	Riferisce più variabili allo stesso oggetto

Operazioni per sequenze

Sia s una sequenza:

$s + t$	Concatena le sequenze (t sequenza)
$s * n$	Ripete la sequenza n volte
$s[i]$	Ritorna l'elemento alla posizione i
$s[i:k]$	Ritorna gli elementi dalla posizione i alla posizione $k-1$
$s[i:k:r]$	Ritorna gli elementi da i a $k-1$ contandone r alla volta
$\text{len}(s)$	Ritorna il numero di elementi appartenenti alla sequenza
$\text{min}(s)$	Ritorna il minimo elemento presente nella sequenza
$\text{max}(s)$	Ritorna il massimo elemento presente nella sequenza

Operazioni per sequenze mutabili

Sia s una sequenza:

$s[i] = x$	Associa il riferimento di x alla posizione i
$s[i:k] = x$	Modifica gli elementi da i a $k-1$
$\text{del } s[i:k:r]$	Elimina gli elementi seguendo gli indici indicati

Operazioni per insiemi

Le classi `set` e `frozenset` supportano i seguenti operatori. Siano $s1$ ed $s2$ due sequenze:

$s1 == s2$	Ritorna <code>true</code> se $s1$ è equivalente ad $s2$
$s1 != s2$	Ritorna <code>true</code> se $s1$ non è equivalente ad $s2$
$s1 \leq s2$	Ritorna <code>true</code> se $s1$ è sottoinsieme di $s2$
$s1 < s2$	Ritorna <code>true</code> se $s1$ è sottoinsieme stretto di $s2$
$s1 \geq s2$	Ritorna <code>true</code> se $s2$ è sottoinsieme di $s1$
$s1 > s2$	Ritorna <code>true</code> se $s2$ è sottoinsieme stretto di $s1$
$s1 s2$	Unione di $s1$ ed $s2$
$s1 \& s2$	Intersezione di $s1$ ed $s2$
$s1 - s2$	Set di elementi in $s1$ ma non in $s2$

Costrutto IF

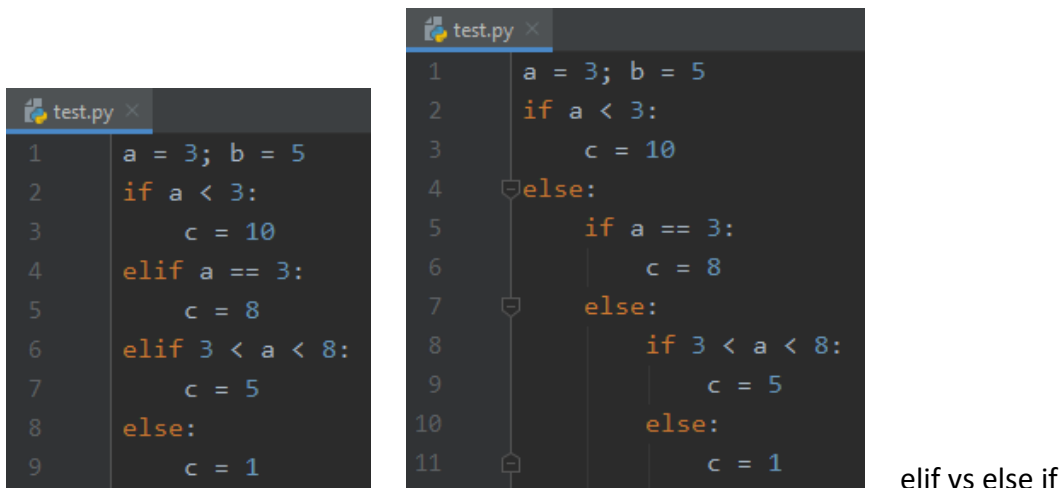
Il costrutto IF permette di intraprendere azioni diverse sulla base del risultato di un test di verità, detto anche "condizione".

In generale, il costrutto IF segue la seguente sintassi:

```
test.py x
1  a = 3; b = 5
2  if a < b:
3      c = 10
4  else:
5      c = 5
```

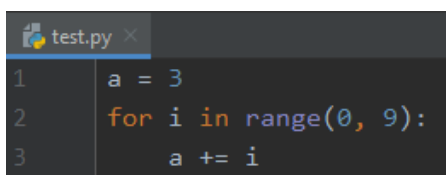
$a < b$ è la condizione del costrutto IF

Sostanzialmente, se la condizione è vera verrà eseguito il blocco di codice posto dopo l'if; se la condizione è falsa verrà eseguito il blocco di codice posto dopo l'else. L'else non è obbligatorio, quindi nel caso la condizione risulti falsa sarà possibile non eseguire nulla, data la mancanza dell'else. È possibile controllare molteplici condizioni in due casi: usando molteplici if concatenati tra loro; usando gli elif. Gli elif non fanno altro che sostituire l'else if, come mostrato nell'immagine:

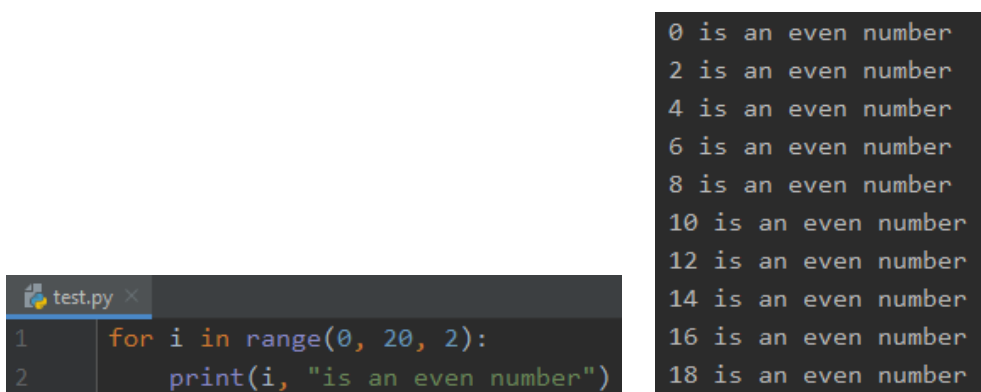


Costrutto FOR

Il costrutto FOR consente di eseguire delle iterazioni utilizzando la seguente sintassi:



Sostanzialmente si segue il modello “for *element* in *iterable*”, quindi un elemento incrementerà o decremerà seguendo come condizione un elemento iterabile o una funzione. Nel caso visto sopra, viene utilizzata la funzione range per scegliere i valori che la variabile i assumerà, in tal caso da 0 ad 8. La funzione range(n, m) genera una lista di interi compresi tra n ed m-1 e permette al for di controllare che i assuma un valore compreso nella lista di interi creata dalla funzione. La funzione range assume anche un altro parametro, opzionale, quale è range(n, m, increment), dove increment regola quanto incrementi la variabile i ad ogni iterazione: se non specificato, la variabile i aumenta di 1 ad ogni iterazione.



Un altro esempio è il seguente: conta le occorrenze della parola "Hello" in un array di parole. Come possiamo vedere, la variabile `i` scorre grazie alla funzione `range` entro 0 e la grandezza dell'array.

```
test.py x
1 # Count the occurrences of the word "Hello"
2 wordToSearch = "Hello"
3 dictionary = ["Hello", "Hi", "Goodbye", "Hello"]
4 counter = 0
5 for i in range(0, len(dictionary)):
6     if dictionary[i] == wordToSearch:
7         counter += 1
8 print("There are", counter, "occurrences")
```

Sostanzialmente, quindi, `range` non fa altro che produrre una lista di interi sul quale l'indice scorrerà. Da ciò possiamo dedurre che "iterabile" non sarà solamente la funzione `range`, bensì potrà essere un qualsiasi oggetto iterabile. Vediamo un esempio:

```
test.py x
1 # Count the occurrences of the word "Hello"
2 dictionary = ["Hello", "Hi", "Goodbye", "Hello"]
3 counter = 0
4 for actualWord in dictionary:
5     if actualWord == "Hello":
6         counter += 1
7 print("There are", counter, "occurrences")
```

Nell'esempio soprastante, si scorre una lista di oggetti: viene preso ogni oggetto della lista, uno per ogni iterazione, e conservato nella variabile "element". Il `for` terminerà quando "element" scansionerà l'intera lista.

```
test.py x
1 for actualNumber in [2, 4, 8, 16, 32, 64]:
2     if actualNumber % 3 == 0:
3         print("There is a multiple of 3")
4         break
5 else:
6     print("There are no multiples of 3")
```

È possibile accodare un `else` al `for`: esso verrà eseguito nel caso il `for` venga eseguito senza interruzioni dovute ad un eventuale `break`. Quindi: se il `for` si conclude senza un `break`, allora eseguirà anche il suo blocco `else`.

Costrutto WHILE

Il costrutto `WHILE` permette di eseguire cicli finiti utilizzando una determinata condizione di uscita e cicli infiniti utilizzando una condizione sempre vera, quindi `while true`. Sostanzialmente segue la seguente sintassi:

```

test.py x
1 a = 3; b = 5
2 while a <= b:
3     print("a value:", a)
4     a += 1

```

```

a value: 3
a value: 4
a value: 5
I'm out

```

$a \leq b$ è la condizione

Finchè la condizione scelta sarà rispettata, while continuerà ad eseguire le istruzioni date in loop. Ovviamente è necessario fare qualcosa, tra le istruzioni, al fine di poter poi uscire dal ciclo: nel caso soprastante, viene aumentata la variabile a in modo tale che diventi maggiore di b.

Istruzioni break e continue

Le istruzioni break e continue si utilizzano nei cicli for e while:

- L'istruzione break termina immediatamente un ciclo for o while;
- L'istruzione continue interrompe la corrente iterazione di un ciclo for o while per poi riprendere la prossima iterazione, verificando sempre la condizione del ciclo.

Comprensione di lista (FOR abbreviato)

La comprensione di lista è un costrutto sintattico che agevola il programmatore nella creazione di una lista partendo dalla scansione di un'altra lista. La sintassi è la seguente:

[expression for item in list if condition]

La sintassi della comprensione di lista è equivalente alla seguente:

```

for item in list:
    if condition:
        expression

```

Vediamo ora tutti gli elementi della comprensione di lista, in modo da averne chiaro il funzionamento:

- for item in list – item conserverà l'oggetto preso da list in base all'iterazione effettuata;
- condition – se la condizione viene rispettata, allora verrà eseguita expression;
- expression – funzione/operazione eseguita nel caso la condition sia rispettata.

Vediamo un paio di esempi:

```

test.py x
1 # filter of even numbers
2 numberList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 evenList = [number for number in numberList if number % 2 == 0]
4 print(evenList)

```

```

test.py x
1 # power list
2 numberList = [1, 2, 3, 4, 5]
3 powerList = [number * number for number in numberList]
4 print(powerList)

```

```

test.py x
1 # power list - new method
2 myList = [1, 2, 3, 4, 5]
3 powList = [myList[i]**2 for i in range(0, len(myList))]
4 print(powList)

```

Nel caso si necessiti di una lista di coppie, è possibile realizzare una doppia comprensione di lista come nel successivo esempio:

```

test.py x
1 # combine the numbers creating couples without duplicates
2 firstList = [1, 2, 3]
3 secondList = [2, 3, 4, 5]
4 newList = [(x, y) for x in firstList for y in secondList if x != y]
5 print(newList)

```

Funzioni

Una funzione è un blocco di codice eseguito solo quando essa viene richiamata. È possibile passare parametri alla funzione, in modo da poter elaborare istruzioni utilizzando tali parametri. Una funzione ritorna, molto spesso, un valore.

Le funzioni vengono dichiarate con la keyword `def`, utilizzando la seguente sintassi:

```

test.py x
1 def myFunction(parameter):
2     newParameter = parameter ** 2
3     return newParameter

```

Ovviamente, dopo aver dichiarato la funzione è possibile assegnarle tutte le istruzioni che si vogliano. Per utilizzarla basta richiamarla durante la normale esecuzione:

```

test.py x
1 def myFunction(parameter):
2     newParameter = parameter ** 2
3     return newParameter
4
5
6 myVar = 10
7 myValue = myFunction(myVar)
8 print(myValue)

```

Come possiamo ben vedere, è possibile passare dei parametri tramite le parentesi della funzione, poste dopo il nome (vengono detti anche argomenti). Si possono passare quanti parametri si vogliano, separandoli con una semplice virgola. È possibile passare anche dei parametri con valori di default, in modo tale che se non vengano passati alla funzione abbiano già dei valori prefissati. Vediamo un esempio:

```

test.py x
1 def myFunction(parameter=10):
2     newParameter = parameter ** 2
3     return newParameter

```

È importante sapere che i parametri vengono passati alla funzione effettuando una copia per valore, quindi non ne viene passato il puntatore; eccezione per le sequenze: viene passato il loro puntatore, quindi qualsiasi modifica ad esse viene registrata anche al di fuori della funzione. Sostanzialmente, quindi, se si passa un valore e lo si modifica nella funzione, all'esterno di essa non verrà riportata la modifica: ciò, però, è possibile grazie alle variabili globali, dichiarandole `global` all'interno della funzione:

```

test.py x
1 def myFunction():
2     global myValue
3     myValue = 5
4
5
6 def anotherFunction(value):
7     value = 100
8
9
10 myValue = 10
11 myFunction()
12 anotherFunction(myValue)
13 print("My value:", myValue)

```

My value: 5

Come anticipato, non c'è un limite ai parametri passabili ad una funzione. È possibile dichiarare funzioni che ricevano un numero variabile di argomenti, usando la seguente sintassi:

```

test.py x
1 def myFunction(first, second, third, *theRest):
2     print(first)
3     print(second)
4     print(third)
5     for i in theRest:
6         print(i)
7
8
9 myFunction(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

```

Se un parametro possiede, precedentemente al nome, una star (*), allora verrà visto come un insieme variabile di argomenti, i quali verranno inseriti in una lista. In tal caso, il parametro `theRest` verrà visto come una lista contenente gli elementi da 4 a 10.

L'operatore star (*) permette di spaccettare un tipo iterabile, come nel seguente esempio:

```

test.py x
1 def customSum(*values: int):
2     mySum = 0
3     for i in values:
4         mySum += i
5     return mySum
6
7
8 valueList = [1, 2, 3, 4, 5]
9 outSum = customSum(*valueList)
10 print(outSum)

```

La variabile valueList punta ad una lista, la quale verrà spaccettata grazie allo star quando sarà passata come parametro: in questo modo valueList non viene passato come una lista, bensì come molteplici parametri (sarebbe l'equivalente di customSum(1, 2, 3, 4, 5)). È vero, nella funzione values viene trattato come una lista: questo perché *values, come visto poco fa, è un insieme variabile di argomenti, i quali verranno inseriti in una lista.

Vediamo un altro esempio, probabilmente molto più semplice da capire:

```

test.py x
1 myList = [1, 2, 3, 4, 5]
2 print(*myList)

```

1 2 3 4 5

Come possiamo ben vedere non viene stampata la lista: quest'ultima viene iterata ritornando uno alla volta i propri elementi.

I parametri visti fin ora sono detti parametri posizionali. Vediamo ora i parametri keyword, i quali sono preceduti dalla doppia star (**) e indicano tipi mapping (collezioni di coppie chiave-valore, ad esempio i dict). I parametri keyword sono sempre posti come ultimi parametri, quindi dopo essi non è possibile passare altri parametri posizionali. Si fa prima a vedere un esempio.

Nell'esempio posto nella prossima pagina notiamo la funzione myOperation accettare come parametri due posizionali ed una keyword. La keyword operation verrà vista come un dict, quindi sarà possibile utilizzare le funzioni per i dizionari su tale parametro. Notiamo che con operation.get("action") non facciamo altro che prendere il parametro action passato alla funzione durante il suo richiamo nel main: qui capiamo perché la keyword deve essere l'ultimo parametro della funzione. Essendo la keyword un dizionario, essa conterrà molteplici coppie passate come argomenti nel richiamo della funzione: se si potessero passare argomenti posizionali, non si riuscirebbe a capire quali argomenti saranno del dizionario e quali no!

Sostanzialmente, quindi, si passano alla funzione coppie chiave-valore utilizzando la seguente sintassi: key="value" in caso di string value, key=value in caso di number value. Tali coppie, nonostante possano sembrare argomenti diversi, verranno raggruppate in un unico dizionario, quale è operation.

Ovviamente una funzione può accettare un'unica keyword con doppia star.


```
test.py x
1 def myOperation(first, second, **operation):
2     if operation.get("action") == "mult":
3         print("Multiplication result:", first * second)
4     elif operation.get("action") == "div":
5         if first >= second:
6             print("Division result:", first / second)
7         else:
8             print("Division result:", second / first)
9     if operation.get("flag") == "watch":
10        if operation.get("action") == "mult":
11            print(first, "x", second)
12        elif operation.get("action") == "div":
13            if first >= second:
14                print(first, ":", second)
15            else:
16                print(second, ":", first)
17    elif operation.get("flag") == "noWatch":
18        print("You can't see the operation")
19
20
21 myOperation(25, 5, action="div", flag="noWatch")
22 myOperation(4, 8, action="mult", flag="watch")
```

```
Division result: 5.0
You can't see the operation
Multiplication result: 32
4 x 8
```

Altra nota da fare riguardo i parametri keyword: anche i parametri con valori di default sono considerati keyword, quindi dopo essi non è possibile passare parametri posizionali.

Bisogna, inoltre, fare attenzione a ciò che si passa al richiamo della funzione. La funzione può essere definita senza parametri keyword, quindi è possibile usare parametri posizionali a piacimento senza alcun dubbio; il problema sorge nel caso si faccia un assegnamento del genere nel richiamo della funzione:

```
test.py x
1 def myFunction(a, b, c):
2     return a+b+c
3
4
5 myFunction(5, 10, 20)
6 myFunction(5, a=10, 8)
```

Problema con assegnamento keyword

L'assegnamento `a=10` viene preso come un parametro chiave-valore (ergo come una keyword), quindi si pensa si stia assegnando valori ad un dizionario. Ciò è assolutamente da evitare siccome, come già detto, i parametri keyword devono essere posti alla fine dell'elenco dei parametri. Nell'ultimo richiamo a funzione, quell'8 risulta in qualche modo essere un parametro chiave-valore riferente al dizionario "b".

Ricapitolando, quindi, i parametri keyword devono essere posti sempre alla fine dell'elenco dei parametri di funzione.

Ovviamente è possibile utilizzare tutti i tipi di parametri finora visti contemporaneamente:

```

test.py x
1 def myFunction(arg1, arg2, *otherArgs, **dictArg):
2     print(arg1)
3     print(arg2)
4     print(otherArgs)
5     print(dictArg)

```

```

1
2
(3, 4, 5, 6)
{'a': 7, 'b': 8, 'c': 9, 'd': 10}

```

Può risultare, a volte, necessario specificare il tipo di un parametro, al fine da rispettarlo durante il passaggio nel richiamo della funzione: ciò è possibile seguendo la sintassi `arg: type`.

È possibile specificare anche il tipo di ritorno della funzione, come vediamo nell'esempio:

```

test.py x
1 def mySum(first: int, second: int) -> int:
2     return first + second
3
4
5 def myDiv(first: int, second: int) -> float:
6     return first / second

```

Finora il parametro è sempre stato un valore o una sequenza.. ma è possibile passare come argomento una funzione da eseguire? Ebbene si!

```

test.py x
1 def run(func, a, b):
2     return func(a, b)
3
4
5 def mySum(a, b):
6     return a + b
7
8
9 def mySub(a, b):
10    return a - b
11
12
13 result = run(mySum, 5, 10)
14 print(result)

```

15

Altra caratteristica delle funzioni è la possibilità di poter documentare la funzione utilizzando tre virgolette, in modo da spiegare cosa essa faccia. È possibile ritornare la stringa di documentazione assegnata alla funzione con `functionName.__doc__`

```

test.py x
1 def run(func, a, b):
2     """ It run a function """
3     return func(a, b)

```

```

print(run.__doc__)

```

Funzioni anonime (espressioni lambda)

Mentre con la keyword `def` definiamo un oggetto di tipo funzione, con la keyword `lambda` definiamo una funzione senza alcun identificativo: tali funzioni non hanno un nome, per tale motivo vengono chiamate funzioni anonime. Se usate bene, le espressioni `lambda` migliorano la lettura del codice: difatti, vengono usate per semplici istruzioni.

La sintassi è la seguente:

```
test.py x
1 mySum = lambda a, b: a + b
2 calculated = mySum(5, 10)
3 print(calculated)
```

L'utilità delle espressioni `lambda` si riscontra quando si ha la necessità di passare una breve funzione come parametro: in alcuni casi è sufficiente specificare, appunto, una `lambda`:

```
test.py x
1 def operation(function, a: int, b: int):
2     return function(a, b)
3
4
5 first = 20
6 second = 30
7 result = operation(lambda a, b: a * b, first, second)
8 print(result)
```

Invece di definire una nuova funzione da passare ad `operation`, si fa prima a definire una `lambda`.

Formattazione dell'output

La funzione `print` riceve un numero variabile di parametri da stampare e due parametri keyword opzionali, quali sono `sep` ed `end`. Il parametro `sep` stabilisce un carattere da posizionare nella concatenazione di variabili, mentre il parametro `end` stabilisce un carattere da posizionare alla fine della stringa.

```
test.py x
1 print("Hello my friend", "how are", "you", sep="*", end="?\n")
2 print("I'm okay", "thank you", sep=",", end="!")
```

Hello my friend*how are*you?
I'm okay,thank you!

La `print`, come ben sappiamo, ovviamente accetta delle stringhe come argomento: in tali stringhe è possibile specificare delle parentesi graffe con un intero all'interno. Tali parentesi verranno sostituite da dei valori stabiliti dalla funzione `format`, la quale non fa altro che creare un dict con i valori che sostituiranno le parentesi.

```
test.py x
1 print("Hello {1}, how are {0}?".format("you", "Mark"))
```

In questo primo caso, vengono usati gli interi per indicare la posizione del valore da usare. Ergo, l'output sarà: "Hello Mark, how are you?"

Vediamo ora, invece, la funzione `format` con la creazione di un dict in argomento:

```
test.py x
1 print("Hello {name}, how are {subject}?".format(name="Mark", subject="you"))
```

Come possiamo ben vedere, non è obbligatorio utilizzare gli indici per indicare il valore da utilizzare: è possibile utilizzare la chiave del dict creato nella funzione `format`. Ovviamente è anche possibile creare un dict precedentemente per poi passarlo direttamente alla funzione `format`:

```
test.py x
1 myDict = {"name": "Mark", "subject": "you"}
2 print("Hello {name}, how are {subject}?".format(**myDict))
```

Input

È possibile chiedere all'utente dei valori in input da tastiera tramite l'apposita funzione `input`, la quale restituirà una stringa contenente ciò che è stato scritto da tastiera. L'inserimento termina con la pressione di invio, quindi con il carattere di newline: quest'ultimo non viene inserito nella stringa letta!

```
test.py x
1 strValue = input("Insert a string: ")
2 intValue = int(input("Insert a number: ")) # cast
3 print("String:", strValue)
4 print("Integer:", intValue)
```

```
Insert a string: Hello
Insert a number: 25
String: Hello
Integer: 25
```

Lettura e scrittura di file

Python permette di interagire con i files tramite un file object, ricavabile tramite la funzione `open(filename, permit)`. Il parametro `filename` specifica il file da aprire/creare (se il file si trova in una cartella diversa è necessario specificare il path), mentre il parametro `permit` specifica i permessi sul file, quindi specifica cosa potremo fare e cosa no.

Il parametro `permit` è una stringa, la quale può assumere uno dei seguenti valori:

- "r", modalità di sola lettura;
- "w", modalità di sola scrittura. Se il file non esiste lo crea; se il file già esiste, il suo contenuto viene cancellato;
- "a", modalità di append. Se il file non esiste lo crea; se il file già esiste, il suo contenuto non viene cancellato e i nuovi contenuti vengono accodati alla fine del file;
- "r+", modalità di lettura e scrittura. Se il file non esiste non lo crea; se il file già esiste, il suo contenuto non viene cancellato e i nuovi contenuti vengono accodati alla fine del file;
- "w+", modalità di lettura e scrittura. Se il file non esiste lo crea; se il file già esiste, il suo contenuto non viene cancellato e i nuovi contenuti vengono accodati alla fine del file.

Sull'oggetto file ritornato dalla funzione `open` è possibile richiamare alcuni metodi, tra i quali `write` e `read`. Tali metodi ci consentono, rispettivamente, di scrivere e leggere il contenuto del file. Il metodo `write` prende come argomento una stringa e restituisce il numero di caratteri appena scritti su file, mentre il metodo `read` restituisce ciò che è scritto su file (è presente anche un

parametro opzionale per il metodo read che permette di specificare il numero di caratteri da leggere).

È importante sapere una cosa: quando si scrive una stringa su file, essa non viene immediatamente immessa su file, bensì viene conservata in un buffer e viene realmente scritta solamente quando quest'ultimo viene svuotato. Il buffer può essere svuotato tramite il metodo flush richiamabile su file oppure tramite la chiusura del file tramite il metodo close. Ovviamente quando un file non serve più è sempre necessario chiuderlo con il metodo close.

```
test.py x
1 file = open("testFile.txt", "w")
2 file.write("Hello")
3 print(open("testFile.txt").read())
4 print("World")
5 file.close()
```

World

Versione senza flush

```
test.py x
1 file = open("testFile.txt", "w")
2 file.write("Hello")
3 file.flush()
4 print(open("testFile.txt").read())
5 print("World")
6 file.close()
```

Hello
World

Versione con flush

Una volta raggiunta la fine del file, il metodo read restituirà sempre una stringa vuota. È possibile riposizionarsi nel file tramite il metodo seek: tale metodo, oltre ad impostare la nuova attuale posizione, svuota il buffer e restituisce la posizione impostata.

```
test.py x
1 file = open("testFile.txt", "r")
2 print(file.read())
3 print(file.read())
4 file.seek(0) # start of file
5 print(file.read())
6 file.close()
```

Hello

Hello

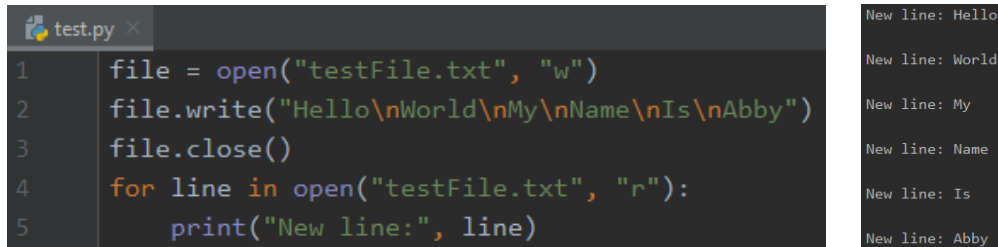
Per conoscere l'attuale posizione nel file si utilizza il metodo tell, il quale, come seek, svuota il buffer quando viene chiamato. Entrambi i metodi misurano la posizione in byte e non conteggiando il numero dei caratteri. Il metodo seek prende anche un secondo argomento opzionale, il quale permette di variare la posizione attuale nel file in base ad alcuni criteri. Sia variation il primo parametro e position il secondo parametro della seek, quest'ultimo può assumere tre valori:

- 0 -> Imposta il riferimento all'inizio del file, quindi ci si sposterà di variation byte partendo dalla posizione 0;
- 1 -> Il riferimento rimane la posizione attuale, quindi ci si sposterà di variation byte partendo dalla posizione attuale;

- 2 -> Imposta il riferimento alla fine del file, quindi ci si sposterà di variation byte partendo dalla fine del file.

[Clicca qui per la guida completa sui files e per la lista completa dei metodi.](#)

È bene sapere che i file objects sono degli iteratori, quindi è possibile iterare sulle linee di un file:



The screenshot shows a Python script in a file named 'test.py'. The script opens a file 'testFile.txt' in write mode ('w'), writes the string 'Hello\nWorld\nMy\nName\nIs\nAbby' to it, and then closes the file. It then opens the same file in read mode ('r') and iterates over each line, printing 'New line:' followed by the line content. The output on the right shows the result of this iteration: 'New line: Hello', 'New line: World', 'New line: My', 'New line: Name', 'New line: Is', and 'New line: Abby'.

```
1 file = open("testFile.txt", "w")
2 file.write("Hello\nWorld\nMy\nName\nIs\nAbby")
3 file.close()
4 for line in open("testFile.txt", "r"):
5     print("New line:", line)
```

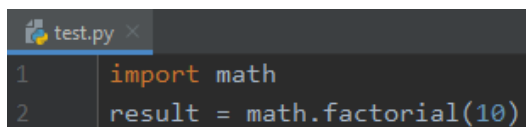
New line: Hello
New line: World
New line: My
New line: Name
New line: Is
New line: Abby

Moduli

La libreria standard di Python è strutturata in moduli, ciascuno dei quali è, sostanzialmente, un contenitore di attributi: prendendo come esempio il modulo math, esso offre classi, funzioni e costanti che forniscono il supporto per la matematica.

In pratica, quindi, sono utili per decomporre un programma di grandi dimensioni in più file. Particolarità dei moduli è quella di offrire la possibilità di poter riusare classi e funzioni in nuovi progetti. Per utilizzare un modulo, codesto deve essere incluso e può esser fatto in diversi modi:

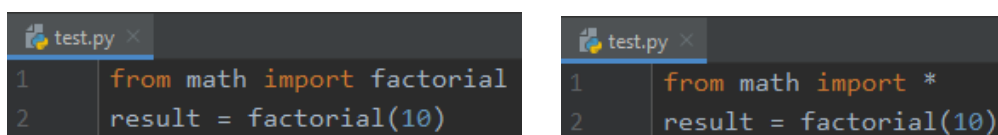
Se si utilizza l'istruzione `import moduleName`, si importa l'intero `moduleName`; quando dovranno essere utilizzate le funzioni rilette a tale modulo, ebbene sarà necessario richiamare il modulo e il metodo scelto, come nell'esempio:



The screenshot shows a Python script in a file named 'test.py'. The script imports the 'math' module and then uses 'math.factorial(10)' to calculate the factorial of 10.

```
1 import math
2 result = math.factorial(10)
```

Nel caso si voglia evitare di richiamare ogni volta il nome del modulo, è possibile utilizzare la sintassi del prossimo esempio. In tal modo, vengono importate direttamente le funzioni del modulo, senza dover richiamare quest'ultimo. Nell'esempio di sinistra viene mostrato l'import di un'unica funzione, mentre nell'esempio di destra viene mostrato l'import di tutte le funzioni appartenenti al modulo math:



The two screenshots show Python scripts in files named 'test.py'. The left screenshot shows the import of a specific function 'factorial' from the 'math' module, followed by its use. The right screenshot shows the import of all functions from the 'math' module using an asterisk, followed by the use of 'factorial'.

```
1 from math import factorial
2 result = factorial(10)
```

```
1 from math import *
2 result = factorial(10)
```

È possibile raggruppare molteplici moduli in un package, il quale non è altro che una cartella per riorganizzare al meglio i files. Il package deve contenere un file, anche vuoto, chiamato `__init__.py`, il quale permetterà di riconoscere la cartella come raccolta di moduli.

Lo script che importerà un determinato modulo appartenente ad un determinato package deve conoscere la posizione del modulo. È possibile arrivare a quel determinato modulo nei seguenti modi:

- Se il modulo da includere è innestato in dei package posti a livelli “inferiori”, basterà specificare il path tramite dei punti, come nei seguenti esempi:
 - `import sound.effects.echo` -> `sound/effects/echo.py`
 - `from sound.effects.surround import *` -> `sound/effects/surround.py`
- Se il modulo da includere è innestato in dei package posti a livelli “superiori”, basterà specificare il path risalendo tramite punti concatenati, come nei seguenti esempi (specifichiamo il fatto che si parta dalla cartella effects, tale che `sound/effects/*.py`):
 - `from .. import formats` -> PERCORSO: `effects-sound-formats`
 - `formats` path: `sound/formats/*.py`
 - `from ..filters import equalizer` -> PERCORSO: `effects-sound-filters-equalizer.py`
 - `equalizer.py` path: `sound/filters/equalizer.py`
 - `from . import echo` -> PERCORSO: `effects`
 - `echo.py` path: `sound/effects/echo.py`

Notiamo, quindi, che se presente un punto si cerca nello stesso package; se presenti più punti, si sale ai package di livelli più alti nella gerarchia.

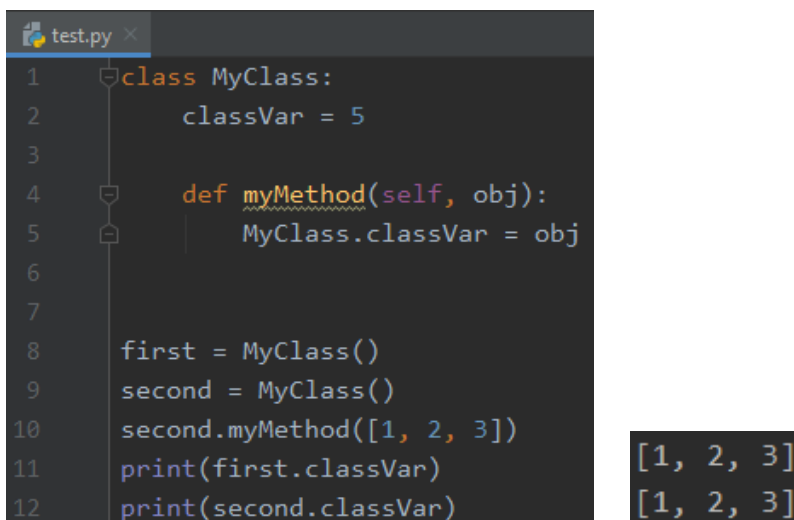
Object Oriented Programming, introduzione

Python supporta tutte le caratteristiche della Object Oriented Programming introducendo alcune particolarità, quali sono il fatto che una classe possa derivare da molteplici classi, il fatto che una classe derivata possa sovrascrivere qualsiasi metodo della classe padre, il fatto che tutti gli attributi e metodi di una classe sono pubblici.

Classi

Se un oggetto creato con l’istruzione `def` è detto funzione, allora uno creato tramite l’istruzione `class` è detto classe. Le classi sono gli elementi alla base della programmazione orientata agli oggetti e permettono di definire tipi di dato: tutti i tipi di dato visti finora (`int`, `float`, `str`, `list`, ecc.) sono delle classi. Una qualsiasi classe è composta da metodi, variabili di classe e variabili di istanza.

È importante la distinzione tra variabili di classe e variabili di istanza: gli assegnamenti fatti al di fuori dei metodi creano degli attributi di classe, quindi condivisi da tutte le istanze. Ciò significa che se una classe o un’istanza modifica l’attributo di classe, tale modifica viene vista sia dalla classe che da tutte le sue istanze.



```

1 class MyClass:
2     classVar = 5
3
4     def myMethod(self, obj):
5         MyClass.classVar = obj
6
7
8 first = MyClass()
9 second = MyClass()
10 second.myMethod([1, 2, 3])
11 print(first.classVar)
12 print(second.classVar)

```

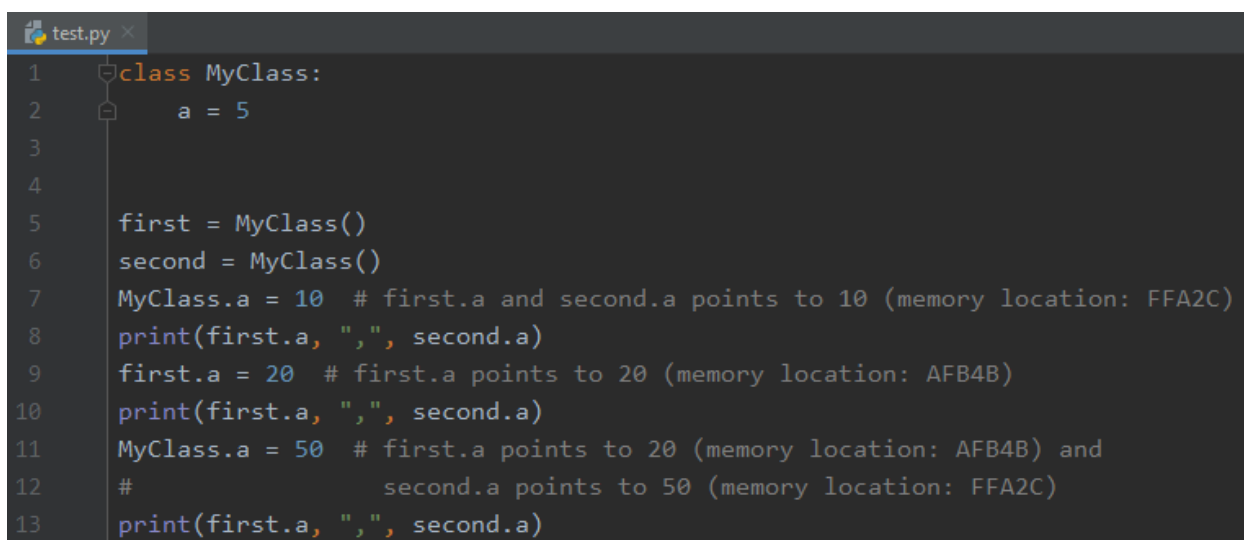
[1, 2, 3]
[1, 2, 3]

L'attributo di classe può esser modificato solamente cambiando il valore all'attributo richiamato sulla classe: `ClassName.attribute = newValue`

Sostanzialmente, quindi, una variabile di classe non è altro che un puntatore condiviso da tutte le istanze di quella classe: è possibile modificare il valore di tale puntatore usando la sintassi vista sopra, altrimenti si modificherebbe il puntatore, come vedremo tra poco.

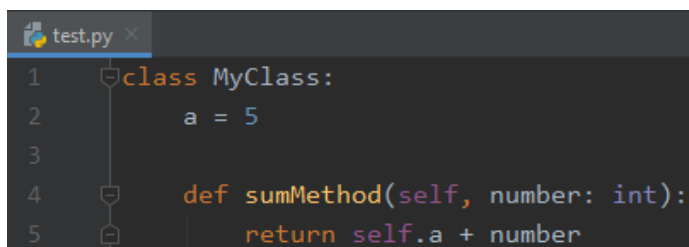
È importante sapere che nel caso si cambi il valore di un attributo di classe richiamandolo da un'istanza, codesto cambiamento verrà rilevato solamente dall'istanza sul quale è stato effettuato: il puntatore viene modificato, quindi quella variabile non punterà più a quella zona di memoria, perdendo il riferimento all'originale puntatore di classe.

Quindi, gli assegnamenti fatti tramite istanza creano o modificano le variabili di istanza, ovvero particolari di quella istanza e non condivisi né con altre istanze né con la classe.



```
1 class MyClass:
2     a = 5
3
4
5     first = MyClass()
6     second = MyClass()
7     MyClass.a = 10 # first.a and second.a points to 10 (memory location: FFA2C)
8     print(first.a, ",", second.a)
9     first.a = 20 # first.a points to 20 (memory location: AFB4B)
10    print(first.a, ",", second.a)
11    MyClass.a = 50 # first.a points to 20 (memory location: AFB4B) and
12    #                 second.a points to 50 (memory location: FFA2C)
13    print(first.a, ",", second.a)
```

Riguardo i metodi, invece, sono dichiarabili sempre nello stesso modo e sono utilizzabili solo riferendosi alla classe o alle sue istanze. Ogni metodo della classe passa l'istanza in modo automatico: quando un metodo è chiamato tramite istanza, questa viene passata al metodo come primo argomento, anche se noi non lo vediamo. Ciò accade anche con i metodi senza argomenti. Nella dichiarazione, il passaggio dell'istanza è ben visibile tramite l'utilizzo del comando `self`.



```
1 class MyClass:
2     a = 5
3
4     def sumMethod(self, number: int):
5         return self.a + number
```

Ogni classe è caratterizzata da unico costruttore, il quale può esser specificato dal programmatore tramite la funzione `__init__` (altrimenti resterebbe quello di default, il quale non effettua alcuna operazione). Il costruttore serve ad inizializzare le variabili di istanza e si comporta come una vera e propria funzione, quindi accetta in ingresso argomenti che potranno esser assegnati alle variabili di istanza. Il fatto che `__init__` sia presente in qualsiasi classe è dovuto dal fatto che ogni classe è discendente della classe `object`.


```

test.py x
1 class MyClass:
2
3     def __init__(self, value):
4         self.container = value

```

Inoltre, è importante specificare che per accedere ad una variabile o ad un metodo della classe all'interno di un'istanza, è necessario sempre richiamarla/o sul self, come nel seguente esempio:

```

test.py x
1 class MyClass:
2
3     def __init__(self, value):
4         self.container = value
5
6     def double(self):
7         self.container *= 2
8
9     def printDoubleContainer(self):
10        self.double()
11        print(self.container)
12
13
14 instance = MyClass(10)
15 instance.printDoubleContainer()

```

Output: 20

Operatori magici

Python attribuisce un significato speciale ad alcuni attributi che iniziano e finiscono con due underscore, i quali vengono detti operatori magici. Riferendoci a ciò che abbiamo visto sulle classi, l'inizializzatore `__init__` è un operatore magico. Gli operatori magici sono dichiarati nella superclasse `object`: trovandosi al vertice della gerarchia di ereditarietà, i suoi attributi vengono ereditati da tutte le classi e quindi appartengono a tutti gli oggetti. La classe `object` non fa altro che definire delle azioni standard da compiere quando le classi non effettuano l'overriding degli operatori speciali. Quindi, se vogliamo personalizzare i compiti assegnati agli operatori speciali, dobbiamo farne l'overriding, implementando quindi il comportamento che ci interessa sovrascrivendo quello fornito da `object`. È più corretto, però, parlare di overloading, siccome è possibile modificare il tipo, il numero di parametri ed il tipo di ritorno degli operatori magici.

Nella prossima pagina vediamo un chiaro esempio di overloading di operatori magici. Vengono ridefiniti `__setitem__` e `__getitem__`, con parametri diversi rispetto a quelli dell'operatore originario. Ovviamente, se richiamato l'operatore magico sull'istanza della classe, verrà eseguito l'operatore appartenente a quella classe e non quello appartenente alla superclasse, siccome è stato sovrascritto. Ovviamente è possibile creare nuovi operatori magici.

Nella prossima pagina vediamo sia la versione con overloading e sia la versione con la dichiarazione di un nuovo parametro magico.

```

1 class MyClass:
2
3     vector = []
4
5     # __setitem__(self, key, value)
6     def __setitem__(self, obj):
7         self.vector.append(obj)
8
9     # __getitem__(self, item)
10    def __getitem__(self, position):
11        return self.vector[position]
12
13
14    instance = MyClass()
15    instance.__setitem__("a")
16    instance.__setitem__("b")
17    instance.__setitem__("c")
18    item = instance.__getitem__(1)
19    print(item)

```

```

1 class MyClass:
2
3     vector = []
4
5     # __additem__ is a new magic operator
6     def __additem__(self, obj):
7         self.vector.append(obj)
8
9     # __getitem__(self, item)
10    def __getitem__(self, position):
11        return self.vector[position]
12
13
14    instance = MyClass()
15    instance.__additem__("a")
16    instance.__additem__("b")
17    instance.__additem__("c")
18    item = instance.__getitem__(1)
19    print(item)

```

Alcuni oggetti possono, quindi, utilizzare operatori commutativi per effettuare determinati compiti. Perché commutativi? Perché esiste una variante che svolge l'operazione "al contrario": tali operatori hanno il nome anticipato dalla lettera r, la quale sta per right. Si fa prima a vedere un esempio:

```

1 a = int(10)
2 b = int(5)
3 print(a.__sub__(b)) # a - b
4 print(a.__rsub__(b)) # b - a

```

Output: 5; -5

Gli operatori right sono, quindi, la controparte dei normali operatori. Non tutti gli operatori hanno una controparte!

Alcuni attributi particolarmente importanti da vedere sono `__call__` ed `__init__`, il quale è stato già visto nel capitolo "Classi": sostanzialmente è il costruttore della classe e permette di associare delle variabili di istanza. Ora concentriamoci su `__call__`.

Se una classe definisce l'operatore magico `__call__`, allora le istanze di tale classe diventano oggetti callable, nel senso che potranno esser trattati come dei metodi e una loro chiamata corrisponde alla chiamata dell'operatore `__call__`.

```

1 class Test:
2     def __call__(self, value):
3         print("You've called the call operator!")
4         print("Here's your value:", value)
5
6
7 var = Test()
8 var(5)

```

```

You've called the call operator!
Here's your value: 5

```

Ereditarietà

L'ereditarietà è il principale meccanismo di riutilizzo del codice nella OOP, grazie al quale è possibile far ereditare ad una classe gli attributi e metodi di un'altra classe, in modo da non dover scrivere lo stesso codice più volte. Se una classe ha variabili e metodi comuni ad un'altra classe, è bene usufruire dell'ereditarietà in modo da non riscrivere codice, evitando lunghi tempi di mantenimento del software, siccome ogni modifica dovrebbe esser replicata anche nel codice superfluo. Una classe eredita attributi e metodi da un'altra classe passandola come argomento nella dichiarazione:

```
test.py x
1 class A:
2     aValue = 10
3
4
5 class B(A):
6     bValue = 20
7
8
9 var = B()
10 print(var.aValue, var.bValue)
```

Output: 10, 20

Richiamare metodi sull'istanza riferita alla classe B significa utilizzare, appunto, metodi della classe B: ma se essi esistono solo nella classe padre A? Allora verranno richiamati nella classe A. Ciò ci porta a notare che in caso di overriding/overloading, verrà data priorità alla classe nativa dell'istanza, in questo caso B. Nel caso, comunque, si voglia utilizzare un metodo della superclasse nonostante sia stato sovrascritto, si utilizza la funzione `super`, come da esempio:

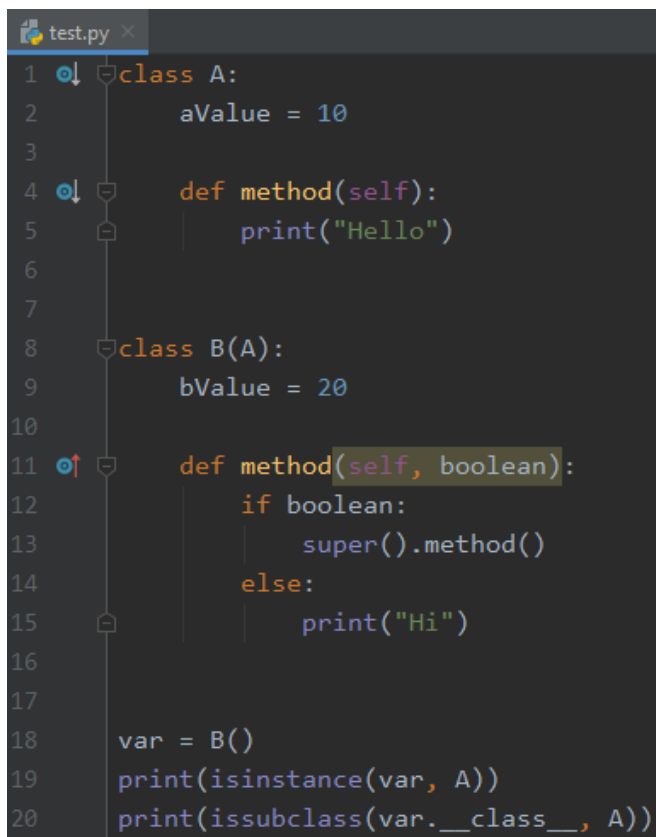
```
test.py x
1 class A:
2     aValue = 10
3
4     def method(self):
5         print("Hello")
6
7
8 class B(A):
9     bValue = 20
10
11     def method(self, boolean):
12         if boolean:
13             super().method()
14         else:
15             print("Hi")
16
17
18 var = B()
19 var.method(True)
```

Output: Hello

Non è possibile, però, utilizzare `super` al di fuori dei metodi della classe: nel `main` non è in alcun modo possibile utilizzare `super`.

Python supporta l'ereditarietà multipla, ovvero una classe può ereditare da più classi: ciò può portare a grande confusione, siccome nel caso uno stesso metodo si trovi in più classi ci porterebbe a domandarci quale metodo di quale classe verrà utilizzato. Ebbene, vengono controllate le classi in base all'ordine in cui vengono passate. Facciamo un esempio: bisogna chiamare la funzione `foo()`, dichiarata nelle superclassi `A1`, `A2` e `A3`; verrà utilizzato il metodo `foo()` appartenente alla superclasse `A1` siccome è la prima ad esser stata passata come padre nell'argomento di classe. Nel caso non si voglia usare `foo()` della classe `A1`, bisognerà specificare il nome della classe da utilizzare nel seguente modo: `ClassName.foo()`.

Può capitare, a volte, di dover controllare se un oggetto sia istanza di una determinata classe o superclasse; altre volte può essere utile sapere se una classe è sottoclasse di un'altra: ciò è reso possibile dai due seguenti metodi:



```
1 class A:
2     aValue = 10
3
4     def method(self):
5         print("Hello")
6
7
8 class B(A):
9     bValue = 20
10
11     def method(self, boolean):
12         if boolean:
13             super().method()
14         else:
15             print("Hi")
16
17
18 var = B()
19 print(isinstance(var, A))
20 print(issubclass(var.__class__, A))
```

Output: True, True

`isinstance` controlla se un determinato oggetto sia istanza di una determinata classe (anche superclasse); `issubclass` controlla se una determinata classe sia sottoclasse di un'altra classe.

Iteratori

Il protocollo di iterazione permette di recuperare uno per volta gli elementi da un oggetto iterabile, evitando che questo venga caricato in memoria. Consideriamo un normalissimo ciclo `FOR`, il quale compie una serie di passi in modo da recuperare gli elementi uno per volta. Come primo passo crea un iteratore `it`, dopodichè chiama il metodo `it.__next__()` per ogni iterazione da

eseguire, in modo da ottenere tutti gli elementi uno per volta. Quando l'iteratore non ha più elementi da restituire, lancia un'eccezione di tipo `StopIteration`.

È possibile emulare tale sistema creando un iteratore. Come? Tramite una funzione.

```
test.py x
1 def createIterator(number):
2     for k in range(0, number):
3         yield k
```

La funzione appena vista restituisce un tipo iteratore, il quale conterrà una lista di elementi iterabili secondo alcuni metodi. Tali elementi vengono inseriti nell'iteratore grazie al comando `yield` (l'uso di `yield` vieta l'uso del `return`). Una volta tornato l'iteratore, è possibile scorrere gli elementi uno per volta tramite il metodo `__next__()`

```
test.py x
1 def createIterator(number):
2     for k in range(0, number):
3         yield k
4
5
6 a = createIterator(5)
7 print(a.__next__())
8 print(a.__next__())
```

Output: 0, 1

Inoltre, siccome tale funzione torna un iteratore, sarà possibile passarlo al FOR:

```
test.py x
1 def getIterator(minVar, maxVar):
2     for item in range(minVar, maxVar):
3         if item % 2 == 0:
4             yield item
5
6
7 for k in getIterator(2, 11):
8     print(k)
```

2
4
6
8
10

Nell'esempio soprastante possiamo notare che è possibile aggiungere elementi all'iteratore anche seguendo determinati criteri! Sostanzialmente è possibile selezionare in qualsiasi modo elementi da passare all'iteratore, l'importante è passarli tramite l'istruzione `yield`.

Superclassi astratte

Può capitare di voler definire una classe generica che non possa, per qualche motivo, implementare determinati metodi. Facciamo un esempio: poniamo di voler creare la classe `FiguraGeometrica`, con varie sottoclassi come `Rettangolo`, `Cerchio`, e così via; tutte le sottoclassi hanno in comune molteplici metodi, come il calcolo dell'area, del perimetro, eccetera, però `FiguraGeometrica` non sa definirne l'implementazione, siccome varierebbe da sottoclasse a sottoclasse. Si ricorre, quindi, ad una tipologia di classi che sollevano un'eccezione nel caso la

sottoclasse tenti di utilizzare un metodo della superclasse non implementato: tale tipologia è detta superclasse astratte. Il comando `raise` permette di sollevare un'eccezione, la quale può essere gestibile, in modo da non causare l'interruzione forzata del programma (vedremo più avanti).

```
test.py x
1 class AbstractClass:
2
3     def abstractMethod(self):
4         raise NotImplementedError("Method not implemented")
5
6
7 class SubClass(AbstractClass):
8     pass
9
10
11 a = SubClass()
12 a.abstractMethod()
```

```
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 12, in <module>
    a.abstractMethod()
  File "D:/Local Workspace/Python-Workspace/test.py", line 4, in abstractMethod
    raise NotImplementedError("Method not implemented")
NotImplementedError: Method not implemented
```

Altro modo riguarda l'utilizzo delle `assert`, le quali terminano il programma in base a determinate condizioni:

```
test.py x
1 class AbstractClass:
2
3     def abstractMethod(self):
4         assert False, "Method not implemented"
5
6
7 class SubClass(AbstractClass):
8     pass
9
10
11 a = SubClass()
12 a.abstractMethod()
```

```
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 12, in <module>
    a.abstractMethod()
  File "D:/Local Workspace/Python-Workspace/test.py", line 4, in abstractMethod
    assert False, "Method not implemented"
AssertionError: Method not implemented
```

L'ideale, però, è che sia le classi astratte che quelle che ereditano da esse e non implementano i metodi astratti delle classi base non siano istanziabili, in modo da non correre il rischio di chiamare qualche metodo non implementato. Il modulo `abc` della libreria standard ci consente di ottenere questo comportamento: riguardo la dichiarazione della classe, sarà necessario specificare come parametro `metaclass=ABCMeta`, il quale assicura che il costruttore della classe lanci l'eccezione quando si tenta di istanziare tale classe astratta; `@abstractmethod` è un decoratore, indica che non si fornisce un'implementazione del metodo astratto e che le classi derivate devono implementarlo. Per usare tali funzioni, è necessario importare `ABCMeta` e `abstractmethod`:

```
test.py x
1  from abc import ABCMeta, abstractmethod
2
3
4  class AbstractClass(metaclass=ABCMeta):
5
6      @abstractmethod
7      def abstractMethod(self):
8          pass
```

Metodi statici, metodi di classe e metodi di istanza

Abbiamo parlato finora in maniera molto generale dei metodi, ma è bene precisare il fatto che esistano tre tipologie di metodi: metodi di classe, metodi di istanza e metodi statici. Ai metodi di classe viene passato implicitamente come primo argomento la classe, mentre a quelli di istanza viene passata l'istanza stessa. Di solito i metodi operano sull'istanza, quindi nella maggior parte dei casi incontreremo quelli. Vediamo nel dettaglio tutte le categorie nominate:

I metodi di classe sono dei metodi a cui viene implicitamente passata la classe come primo argomento: se vogliamo che un metodo `foo` sia di classe, dobbiamo renderlo tale decorandolo con la classe built-in `classmethod`, la quale lo modifica rendendolo, appunto, di classe.

```
test.py x
1  class TestClass:
2
3      variable = 10
4
5      @classmethod
6      def myMethod(cls, newValue):
7          cls.variable = newValue
```

Per convenzione la classe si indica con `cls`

Durante il richiamo, la classe viene passata a prescindere, anche se non la si definisce.

```
var = TestClass()
var.myMethod(20)
print(var.variable)
```

La classe è passata implicitamente, quindi si può fare a meno di definirla.

I metodi statici non sono di classe ma vengono qualificati tramite essa. A questi metodi non viene passato alcun argomento, per cui sono semplici funzioni che vivono nello scope locale della classe.

È possibile e preferibile rendere un metodo statico in modo esplicito, decorandolo con la classe built-in `staticmethod`: in tal modo, sarà statico a prescindere da come esso viene qualificato.

```
test.py x
1 class TestClass:
2
3     @staticmethod
4     def myMethod():
5         print("That's a static method!")
6
7
8 var = TestClass()
9 var.myMethod()
```

Un metodo di istanza, invece, accetta come parametro l'istanza stessa, quindi `self`, implicitamente. Sostanzialmente, sono i metodi finora visti in cui si passa `self` come parametro.

```
test.py x
1 class TestClass:
2
3     variable = 10
4
5     def myMethod(self, newValue):
6         self.variable = newValue
```

Le eccezioni

Gli errori in Python si verificano sotto forma di errori di sintassi oppure di eccezioni. Gli errori di sintassi vengono rilevati durante la compilazione del programma in bytecode, mentre le eccezioni sono errori che si presentano a runtime. Nel caso si sollevi un'eccezione, l'esecuzione del programma termina e viene mostrato un messaggio di errore, composto dal traceback, dal tipo di eccezione e dalla causa dell'errore: il traceback tiene traccia delle linee di codice coinvolte nell'errore, partendo dalla meno recente fino ad arrivare alla più recente.

Le eccezioni non sono altro che istanze di determinate classi, come `IndexError`, `TypeError`, `NameError`; hanno un attributo tupla chiamato `args` che contiene gli argomenti passati alla classe al momento della creazione dell'istanza. Nella maggior parte dei casi, un'eccezione viene creata passando alla classe una stringa che descrive la causa dell'errore, la quale verrà mostrata in output nel caso l'eccezione venga sollevata dal programma.

La gestione delle eccezioni avviene tramite il costrutto `try` ed `except`:

```
test.py x
1 myList = [0, 1, 2]
2 try:
3     var = myList[3]
4 except IndexError:
5     print("Wow, there's an exception!")
```


Nel caso nel blocco try venga sollevata un'eccezione, si prosegue entrando nel blocco except, eseguendo le istruzioni che conterrà. Se, invece, non viene sollevata alcuna eccezione nel blocco try, allora il blocco except non verrà eseguito. Ovviamente il blocco try rileverà qualsiasi eccezione lanciata, ma verrà gestita dal blocco except solo se l'eccezione presentata è inclusa tra le eccezioni specificate dal blocco except. Una volta eseguito il blocco except, il programma continuerà il suo normale funzionamento.

La clausola except può specificare molteplici eccezioni per lo stesso blocco di codice, specificandole in delle parentesi con la seguente sintassi:

```
except (IndexError, ValueError):  
    print("Wow, there's an exception!")
```

In alternativa è possibile utilizzare molteplici clausole except, in modo da associare diversi blocchi di codice per ogni eccezione presentata:

```
except IndexError:  
    print("Wow, there's an exception!")  
except ValueError:  
    print("Another exception!")
```

È, inoltre, possibile catturare l'eccezione e associarla ad un'etichetta, in modo da poterla gestire come un'oggetto:

```
except IndexError as myException:  
    print("Error:", myException)
```

Oltre alle clausole try ed except esiste anche la clausola finally, la quale è molto utile quando ci si vuole assicurare che alcune azioni vengano eseguite, qualunque cosa accada. Non è altro che un blocco di codice che viene eseguito a prescindere dal fatto che venga sollevata o meno un'eccezione:

```
test.py x  
1 try:  
2     int("Hahaha i'll destroy this program")  
3 except IndexError:  
4     print("Index error!")  
5 finally:  
6     print("The IndexError exception is a joke")
```

```
The IndexError exception is a joke  
Traceback (most recent call last):  
  File "D:/Local Workspace/Python-Workspace/test.py", line 2, in <module>  
    int("Hahaha i'll destroy this program")  
ValueError: invalid literal for int() with base 10: "Hahaha i'll destroy this program"
```

La clausola else (posta dopo il blocco except) permette di specificare un blocco di codice eseguito solo nel caso nessuna eccezione venga sollevata dal blocco try.

```

test.py x
1  try:
2      var = int(5)
3  except IndexError:
4      print("I will not enter here")
5  else:
6      print("I've skipped the except block")

```

È possibile sollevare eccezioni manualmente tramite due istruzioni: `raise` ed `assert`. L'istruzione `raise` permette di sollevare qualsiasi eccezione, mentre l'istruzione `assert` permette di sollevare solamente errori di tipo `AssertionError`.

L'istruzione `raise` solleva una qualsiasi eccezione, con possibilità di passare come argomento all'eccezione una stringa, la quale indicherà il messaggio di errore dell'eccezione:

```

test.py x
1  var = int(input("Insert a number: "))
2  if var >= 10:
3      raise ValueError("The number must be lower than 10")

```

```

Insert a number: 12
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 3, in <module>
    raise ValueError("The number must be lower than 10")
ValueError: The number must be lower than 10

```

L'istruzione `assert`, invece, solleva solamente eccezioni di tipo `AssertionError` quando la sua espressione di test è valutata come `false`: sostanzialmente è un test che se da come risultato `false` solleva l'eccezione:

```

test.py x
1  var = int(input("Insert a number: "))
2  try:
3      assert var % 2 == 0
4  except AssertionError as catchException:
5      print("Error!")

```

```

Insert a number: 7
Error!

```

L'istruzione `assert`, quindi, viene utilizzata solamente per il test di poche e semplici istruzioni; il sollevamento di tale eccezione dipende dalla variabile globale `__debug__`, la quale permetterà il sollevamento delle eccezioni `AssertionError` solo se posta a `true`.

I decorator (generale)

I decorator sono funzioni "speciali" che permettono di aggiungere nuove funzionalità ad una funzione o ad una classe. Una funzione o classe utilizzata per decorare un oggetto è detta *decoratore*: in particolare, quando l'oggetto da decorare è una funzione, il decoratore verrà detto "decoratore di funzione". Analogamente, quando l'oggetto da decorare è una classe, il decoratore verrà detto "decoratore di classe". Nel caso, quindi, si voglia decorare un oggetto (che sia una

funzione o una classe), basta specificare il decoratore utilizzando la @ nella linea che precede la definizione della funzione o della classe:

È possibile specificare un nuovo decoratore, il quale dovrà ritornare una funzione, la quale verrà eseguita prima che venga eseguito il blocco di codice a cui il decoratore si riferisce quando utilizzato:

```
test.py x
1 def myDecorator(func):
2     pass
3
4
5 @myDecorator
6 def myFunction():
7     pass
```

Facendo riferimento al capitolo dove si parla di metodi astratti, statici e di classe, ebbene le funzioni specificate con @ erano proprio dei decoratori. I decoratori possono esser specificati anche in cascata, assegnandone molteplici allo stesso oggetto.

I decoratori di funzione prendono come argomento una funzione, la quale sarà quella decorata, per poi restituire una funzione wrapper: la funzione wrapper contiene istruzioni, è un involucro contenente codice aggiuntivo che verrà eseguito prima dell'esecuzione della funzione decorata.

```
test.py x
1 def doubleValues(function):
2     def wrapper(*args, **kwargs):
3         newList = list()
4         for item in args:
5             item *= 2
6             newList.append(item)
7         args = newList
8         return function(*args, **kwargs)
9     return wrapper
10
11
12 @doubleValues
13 def add(a, b):
14     return a + b
15
16
17 print(add(5, 6))
```

Nell'esempio soprastante, specifichiamo un decoratore che raddoppi i parametri in ingresso della funzione add: si definisce un wrapper che accetta gli argomenti della funzione add (quali sono args e kwargs) e li moltiplica per due. Dal wrapper viene ritornata la funzione decorata con i parametri modificati; dal decoratore doubleValues viene ritornata la funzione wrapper, la quale verrà eseguita prima di eseguire la vera e propria add.

Sostanzialmente, quindi, viene eseguito prima il wrapper, per poi passare all'esecuzione della funzione decorata.

È possibile, inoltre, associare il decoratore sottoforma di funzione, in modo da poter anche passare argomenti:

```
def add(a, b):  
    return a + b  
  
add = doubleValues(add)  
print(add(5, 6))
```

Oltre ai decoratori di funzione esistono anche i decoratori di classe: aggiungono nuove funzioni e nuove caratteristiche alle classi, quindi anche alle istanze di esse. I decoratori di classe non prendono come argomento una funzione, bensì una classe, permettendo modifiche in ogni aspetto e funzione: è addirittura possibile modificare il costruttore della classe decorata.

```
CounterWithDecorator.py x  
1 def dec_counterClass(decoratedClass):  
2     decoratedClass.numberOfInstances = 0  
3     decoratedClass.oldInit = decoratedClass.__init__  
4  
5     def moddedInit(self, *args, **kwargs):  
6         decoratedClass.numberOfInstances += 1  
7         decoratedClass.oldInit(self, *args, **kwargs)  
8  
9         decoratedClass.__init__ = moddedInit  
10  
11     return decoratedClass  
12  
13  
14 @dec_counterClass  
15 class Counter:  
16     var = 0  
17  
18  
19     a = Counter()  
20     b = Counter()  
21     c = Counter()  
22     print(a.numberOfInstances)
```

Nell'esempio soprastante, definiamo il decoratore passando come argomento una classe: come possiamo ben vedere, è possibile associare variabili ed effettuare nuove operazioni. È importante sapere che nel caso si vada ad intaccare il costruttore della classe originaria, come nel caso soprastante, è altamente consigliabile salvarsi l'init originario (cioè l'init della classe decorata) in una variabile, in modo da eseguirlo successivamente: riferendoci al caso di sopra, vogliamo fare in modo che la classe, alla creazione di una propria istanza, incrementi un contatore, tenendo conto

di tutte le istanze della classe. È chiaro si debba incrementare il contatore nell'init, ma come fare? Ebbene si stabilisce una nuova funzione init (in questo caso moddedInit) che incrementi il contatore e che esegua il vecchio init, salvato in una variabile (in questo caso oldInit); successivamente tale moddedInit viene salvato nella variabile init originaria.

È lecito, quindi, chiederci: perché è necessario salvarsi l'init originario? Nell'esempio soprastante è inutile, siccome la classe Counter non fa nulla di suo, il costruttore è standard e vuoto; il fatto è che i decorator, in genere, non sono stabiliti per un'unica classe, bensì dovrebbero poter essere utilizzati su qualsiasi altra classe. Se non si va a salvare l'init originario, nel caso la classe decorata abbia un init contenente istruzioni, queste ultime andranno perse: per ovviare tale problema viene conservato il vecchio init in una variabile ed eseguito nel nuovo init, stabilito dal decoratore.

Proprietà di istanza (getter, setter, deleter)

Un'istanza, come ben sappiamo, è composta da variabili e funzioni, quali possono essere di istanza o di classe. Per qualche ragione si può avere la necessità di variabili di istanza o classe che possano essere modificate solo con alcuni parametri o che possano essere utilizzate solo in alcune condizioni: sostanzialmente, si vuole vietare il libero accesso a quella variabile. Si introducono, quindi, le variabili private, indicate con un carattere underscore posto prima del nome della variabile, le quali vietano l'accesso dall'esterno. In realtà le variabili private non esistono, l'uso dell'underscore è una convenzione adottata al fine di indicare il non utilizzo di tali variabili all'esterno: l'utilizzo, comunque, non lancerebbe eccezioni, ma sarebbe logicamente errato.

Nel caso queste variabili debbano essere utilizzate e/o modificate solo in alcune condizioni, non basta renderle private; si utilizzano apposite funzioni in grado di restituire, modificare ed eliminare quel valore: tali funzioni vengono scritte dal programmatore, il quale potrà regolamentare la restituzione, la modifica e l'eliminazione della variabile.

```
test.py x
1  class MyClass:
2
3      def __init__(self):
4          self._variable = 0
5
6      def getter(self):
7          return self._variable
8
9      def setter(self, value):
10         self._variable = value
11
12     def deleter(self):
13         del self._variable
14
15     variable = property(getter, setter, deleter, "Documentation string")
16
17
18     var = MyClass()
19     var.variable = 10
20     print(var.variable)
```

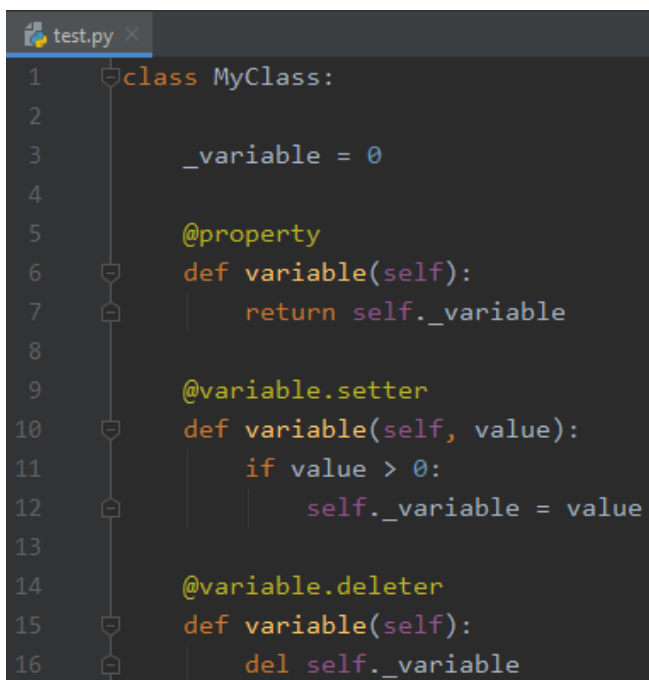
Nell'esempio soprastante vediamo denotata una variabile privata chiamata `_variable`, la quale viene settata a 0 dal costruttore. Essendo `_variable` inaccessibile (per convenzione) dall'esterno della classe, vengono stabiliti dei metodi per permetterne l'accesso, la modifica e l'eliminazione. È vero, l'esempio non rende l'utilità, ma basta sapere che in quei metodi è possibile inserire qualsiasi controllo o operazione si voglia! Per fare un esempio, si può permettere una modifica alla variabile solo se il nuovo valore è maggiore di 0:

```
def setter(self, value):
    if value > 0:
        self._variable = value
```

In tal modo, grazie a tali metodi, la modifica alla variabile privata `_variable` è permessa solo in determinati casi scelti dal programmatore. Altra cosa interessante, al di fuori della classe non vengono utilizzati i metodi `getter`, `setter` e `deleter`: grazie alla funzione `property`, una variabile con lo stesso nome della variabile privata, ma senza underscore, potrà esser utilizzata come una normale variabile ma facendo riferimento alla privata, utilizzando le associazioni al posto delle funzioni. Cosa significa? Ebbene, se alla nuova variabile `variable` viene fatto un assegnamento, verrà automaticamente invocato il metodo `getter` scritto per la variabile privata. Nell'esempio visto, viene effettuata l'associazione `var.variable = 10`: non viene associato un valore alla variabile `variable`, bensì viene richiamato il `setter` che è stato scritto, associato grazie alla funzione `property`, e il nuovo valore viene conservato nella variabile privata associata.

In questo modo, quindi, la variabile privata potrà esser restituita, modificata ed eliminata solo secondo i criteri stabili dal programmatore.

Nel caso non si vogliano definire funzioni con nomi propri, è possibile definire funzioni con il nome della nuova variabile pubblica che abbiano le stesse funzioni della `getter`, `setter` e `deleter`, utilizzando i decorator:



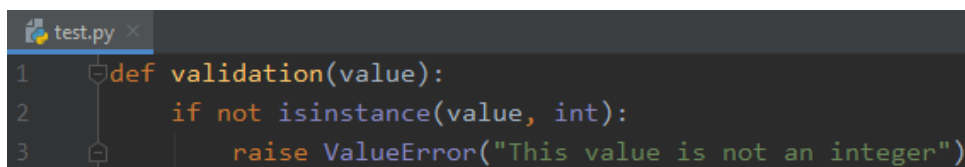
```
test.py x
1 class MyClass:
2
3     _variable = 0
4
5     @property
6     def variable(self):
7         return self._variable
8
9     @variable.setter
10    def variable(self, value):
11        if value > 0:
12            self._variable = value
13
14    @variable.deleter
15    def variable(self):
16        del self._variable
```

Come possiamo ben vedere, le tre funzioni hanno lo stesso nome, quale è quello della nuova variabile pubblica. Il getter avrà come descrittore `@property`, il setter `@variable.setter` e il deleter `@variable.deleter`. Nella normale esecuzione, la variabile viene gestita come visto nel primo esempio. Entrambi i modi sono equivalenti, non c'è alcuna differenza circa il funzionamento.

Decoratore @ensure di classe

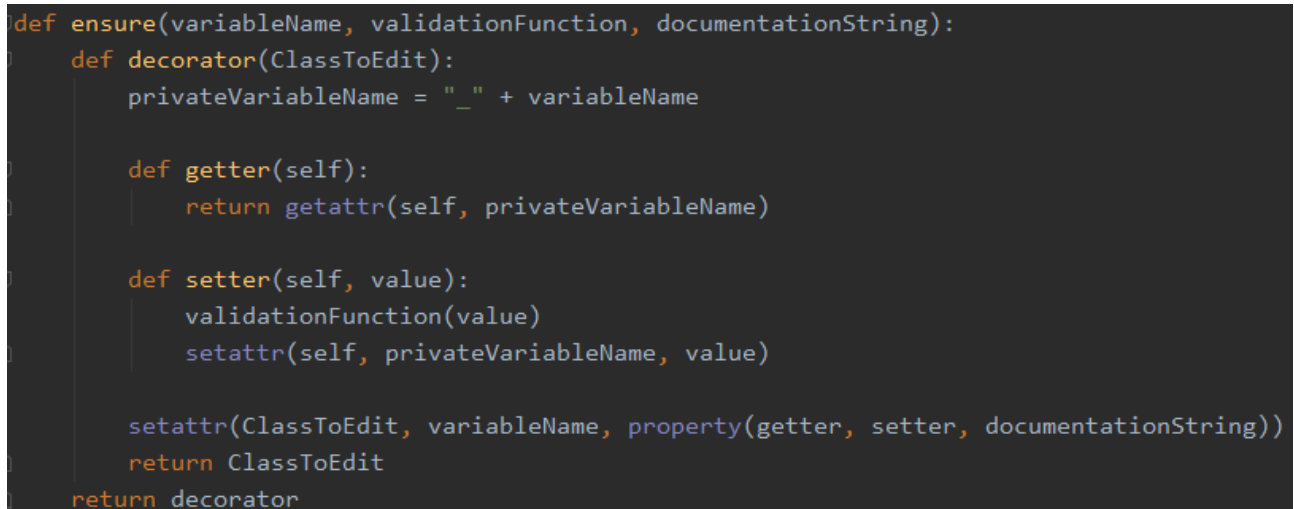
Una classe è composta da molteplici variabili, delle quali si sente spesso l'esigenza dei metodi getter, setter e deleter: non è possibile scrivere tanto codice per dei metodi riferiti a singole variabili. Per fortuna, Python permette la creazione di decoratori che evitino la scrittura di cotanto codice, automatizzando ciò che abbiamo visto nel capitolo precedente. I decoratori che permettono ciò sono detti ensure, non sono altro che funzioni che permettono di creare variabili di classe con metodi getter e setter, permettendo anche il rispetto delle condizioni.

La prima cosa da fare è stabilire una funzione che controlli se le condizioni per effettuare la getter o la setter siano rispettate:



```
test.py x
1 def validation(value):
2     if not isinstance(value, int):
3         raise ValueError("This value is not an integer")
```

Creata la funzione, si sviluppa il decoratore ensure:



```
def ensure(variableName, validationFunction, documentationString):
    def decorator(ClassToEdit):
        privateVariableName = "_" + variableName

        def getter(self):
            return getattr(self, privateVariableName)

        def setter(self, value):
            validationFunction(value)
            setattr(self, privateVariableName, value)

        setattr(ClassToEdit, variableName, property(getter, setter, documentationString))
        return ClassToEdit
    return decorator
```

Il decoratore ensure accetta come parametri il nome della variabile da stabilire per la classe decorata, la funzione di validazione, una stringa di documentazione per documentare la variabile. Il decoratore ensure, nel suo interno, realizza un decoratore di classe: tale decoratore di classe fornirà una variabile contenente il nome della variabile privata, un metodo getter, un metodo setter e un attributo di classe con nome della variabile privata senza underscore, il quale permetterà di accedere alla variabile privata nel modo in cui abbiamo visto sopra. È importante specificare perché si utilizzano `getattr` e `setattr`: non è possibile accedere alla variabile tramite `var.privateVariableName` siccome quest'ultima contiene il nome della variabile privata. Le funzioni `getattr` e `setattr` permettono il get ed il set di variabili passando il loro nome.

Realizzato, quindi, il decoratore ensure, è possibile applicarlo alla classe desiderata:

```

test.py x
1 def validation(value):
2     if not isinstance(value, int):
3         raise ValueError("This value is not an integer")
4
5
6 def ensure(variableName, validationFunction, documentationString):
7     def decorator(ClassToEdit):
8         privateVariableName = "_" + variableName
9         setattr(ClassToEdit, privateVariableName, 0)
10
11         def getter(self):
12             return getattr(self, privateVariableName)
13
14         def setter(self, value):
15             validationFunction(value)
16             setattr(self, privateVariableName, value)
17
18         setattr(ClassToEdit, variableName, property(getter, setter, documentationString))
19         return ClassToEdit
20     return decorator
21
22
23 @ensure("myVariable", validation, "Documentation")
24 class MyClass:
25     pass
26
27
28 myVar = MyClass()
29 myVar.myVariable = 20
30 print(myVar.myVariable)

```

La classe MyClass, quindi, otterrà grazie all'ensure l'attributo `_myVariable`, al quale sarà possibile accedere nei soliti modi visti finora grazie all'implementazione nascosta del getter e del setter.

Decoratori o superclassi?

Spesso può capitare di voler creare una classe con opportuni metodi e dati per poi crearne sottoclassi in modo da sfruttare l'ereditarietà: i decorator possono evitare la duplicazione di metodi e dati e permettono un funzionamento simile a quello dell'ereditarietà, nel caso i metodi ereditati o i dati non vengano mai modificati nelle sottoclassi.

Vediamo prima un esempio di superclasse:

```

test.py x
1 class Superclass:
2     def __init__(self):
3         self.variable = None
4
5     def change(self, value):
6         self.variable = value

```


Tale superclasse sarà ovviamente ereditabile da molteplici sottoclassi con la sintassi precedentemente vista, cioè `class MyClass(Superclass): ...`

Se le sottoclassi non re-implementeranno i metodi della superclasse, può risultare comodo usare i decorator di classe per fornire tali caratteristiche. Vediamo l'equivalente in versione decoratore:

```
test.py x
1  def superclass(Class):
2      setattr(Class, "variable", None)
3
4      def change(self, value):
5          setattr(self, "variable", value)
6
7      setattr(Class, "change", change)
8      return Class
```

Usando il decoratore, è possibile stabilire delle caratteristiche (che siano metodi o attributi) alla classe, le quali non potranno, però, esser sovrascritte.

Singleton Pattern

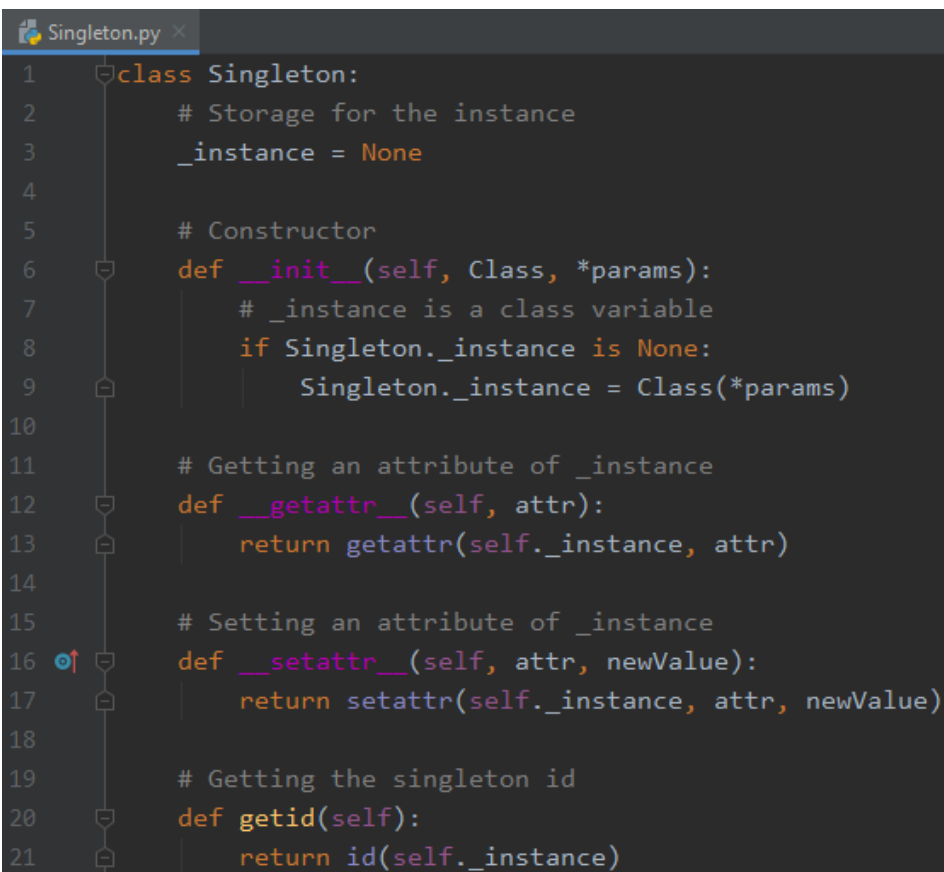
Il Singleton Pattern è ciò che si usa quando si ha bisogno di un'unica istanza di una determinata classe nell'intero programma. Sostanzialmente, si dichiara una classe Singleton che fa da wrapper per un'oggetto di classe la cui classe è specificata internamente, il quale sarà unico in tutto il programma: la classe Singleton avrà un attributo privato `_instance`, il quale conterrà l'istanza che sarà unica. Tramite appositi metodi sarà possibile accedere agli attributi dell'oggetto contenuto in `_instance`.

```
Singleton.py x
1  class Singleton:
2      class InternalClass:
3          internalVar = None
4
5          def __init__(self):
6              internalVar = 0
7
8      _instance = None
9
10     def __init__(self):
11         if Singleton._instance is None:
12             Singleton._instance = Singleton.InternalClass()
13
14     def __getattr__(self, attrName):
15         return getattr(self._instance, attrName, None)
16
17     def __setattr__(self, attrName, value):
18         return setattr(self._instance, attrName, value)
```

Il funzionamento è il seguente:

```
mySingleton = Singleton()
mySingleton.__setattr__("attributeA", 20)
mySingleton.__setattr__("attributeB", 40)
anotherSingleton = Singleton()
print(anotherSingleton.__getattr__("attributeA"))
```

Altro modo di scrivere il singleton è il seguente: al momento della sua istanziazione, il singleton accetta come parametro una classe e dei parametri, i quali inizializzeranno la classe passata come parametro. È decisamente comodo siccome è possibile usare il singleton come classe esterna in qualsiasi programma, siccome basta importarla e diventa compatibile con qualsiasi altra classe anziché scrivere un'apposita classe in un apposito singleton. Il problema è che passando la classe dall'esterno, essa è comunque istanziabile più volte tramite il suo costruttore: il singleton deve impedire ciò, perciò la classe viene specificata privata proprio per evitare che essa venga istanziata molteplici volte. Vediamo comunque tale modello:

A screenshot of a Python IDE window titled 'Singleton.py'. The code defines a class 'Singleton' with several methods. Line 1: 'class Singleton:'. Line 2: '# Storage for the instance'. Line 3: '_instance = None'. Line 4: (blank). Line 5: '# Constructor'. Line 6: 'def __init__(self, Class, *params):'. Line 7: '# _instance is a class variable'. Line 8: 'if Singleton._instance is None:'. Line 9: 'Singleton._instance = Class(*params)'. Line 10: (blank). Line 11: '# Getting an attribute of _instance'. Line 12: 'def __getattr__(self, attr):'. Line 13: 'return getattr(self._instance, attr)'. Line 14: (blank). Line 15: '# Setting an attribute of _instance'. Line 16: 'def __setattr__(self, attr, newValue):'. Line 17: 'return setattr(self._instance, attr, newValue)'. Line 18: (blank). Line 19: '# Getting the singleton id'. Line 20: 'def getid(self):'. Line 21: 'return id(self._instance)'.

```
1 class Singleton:
2     # Storage for the instance
3     _instance = None
4
5     # Constructor
6     def __init__(self, Class, *params):
7         # _instance is a class variable
8         if Singleton._instance is None:
9             Singleton._instance = Class(*params)
10
11     # Getting an attribute of _instance
12     def __getattr__(self, attr):
13         return getattr(self._instance, attr)
14
15     # Setting an attribute of _instance
16     def __setattr__(self, attr, newValue):
17         return setattr(self._instance, attr, newValue)
18
19     # Getting the singleton id
20     def getid(self):
21         return id(self._instance)
```

Il funzionamento è il seguente:

```
mySingleton = Singleton(MyClass)
mySingleton.__setattr__("anotherAttr", 20)
print(mySingleton.__getattr__("myAttr"))
print(mySingleton.__getattr__("anotherAttr"))
print(mySingleton.getid())
anotherSingleton = Singleton(MyClass)
print(anotherSingleton.getid())
```

10
20
30743792
30743792

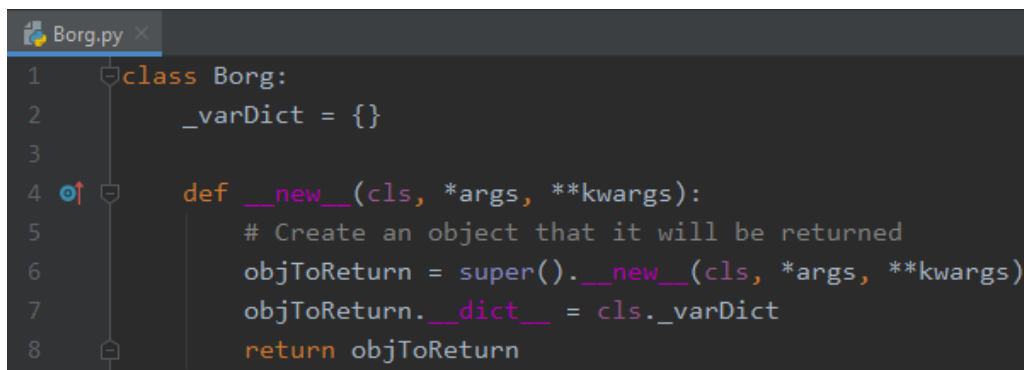
Grazie al costruttore di questo nuovo singleton, la classe Singleton provvederà alla creazione di un oggetto della classe da noi scelta solamente nel caso non sia stato già creato un altro singleton.

Borg Pattern

Il Borg Pattern permette di istanziare più oggetti della stessa classe accomunando le variabili di istanza di ogni oggetto in tutti gli altri: per essere più precisi, se all'oggetto obj (istanza di Borg) viene associato un attributo attr, quest'ultimo sarà variabile di istanza di ogni oggetto di classe Borg. Prima di spiegare come funzioni, è bene introdurre la variabile di istanza `__dict__` e la funzione `__new__`.

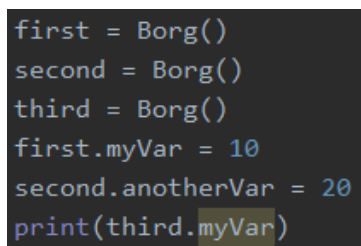
La funzione `__new__` istanzia un oggetto vuoto, il quale verrà inizializzato tramite la funzione `__init__`; la variabile di istanza `__dict__` è un dizionario contenente tutti gli attributi di quella determinata istanza.

Il concetto di borg è proprio quello di cambiare il `__dict__` di default quando si istanzia l'oggetto con la `__new__`: si stabilisce un dizionario di classe che conterrà tutte le variabili di istanza, in modo tale che siano comuni a tutte le istanze della classe Borg.



```
Borg.py x
1 class Borg:
2     _varDict = {}
3
4     def __new__(cls, *args, **kwargs):
5         # Create an object that it will be returned
6         objToReturn = super().__new__(cls, *args, **kwargs)
7         objToReturn.__dict__ = cls._varDict
8         return objToReturn
```

Il funzionamento è il seguente:



```
first = Borg()
second = Borg()
third = Borg()
first.myVar = 10
second.anotherVar = 20
print(third.myVar)
```

Adapter Pattern

L'Adapter Pattern è una tecnica che permette ad una classe di adattarsi ad un'altra classe tramite una semplice interfaccia, senza cambiare il proprio codice o la classe utilizzata. Tale pattern risulta utile quando si vuole utilizzare una classe in un contesto per il quale essa non è stata designata.

Sostanzialmente, risulta utile quando molteplici classi possono effettuare un'operazione categorizzabile risultando un'operazione generica. Per rendere meglio l'idea sarebbe meglio vedere un esempio: poniamo di avere 3 classi, quali sono Computer, Synthesizer e Human; il computer può eseguire un programma, il sintetizzatore può suonare una canzone elettronica e l'umano può parlare. Tutte e tre le classi, quindi, possono eseguire un'azione, la quale potrebbe esser categorizzabile con la funzione `execute()`.

Stabiliamo le tre classi:

```
test.py x
1 class Computer:
2     def __init__(self, name):
3         self.name = name
4
5     def execute(self):
6         return self.name + " is executing a program"
7
8
9 class Synthesizer:
10    def __init__(self, name):
11        self.name = name
12
13    def play(self):
14        return self.name + " is playing a song"
15
16
17 class Human:
18    def __init__(self, name):
19        self.name = name
20
21    def speak(self):
22        return self.name + " is speaking"
```

Poniamo caso di avere un array di istanze di tali classi, senza sapere però precisamente a quale classe appartiene una certa istanza; per semplicità può risultare utile richiamare su ogni oggetto il metodo `execute`, anziché capire se richiamare `play` o `speak`. Peggio ancora, magari il programmatore sa dell'esistenza del metodo `execute` ma non di `play` e `speak`. È possibile fare in modo che alla chiamata di `execute` vengano eseguite `play` o `speak`, in base al tipo di oggetto: ciò viene fatto tramite gli Adapter.

```
Adapter.py x
1 class Adapter:
2     def __init__(self, obj, dictMethods):
3         self.obj = obj
4         self.__dict__.update(dictMethods)
```

ADAPTER VERSIONE 1

L'adapter prende come argomenti un oggetto e un dizionario. L'oggetto viene conservato nella variabile di istanza, mentre il dizionario è una coppia attributo-metodo, in modo da salvare nel dizionario degli attributi un attributo `execute` che contenga il metodo da richiamare.

Difatti, come possiamo ben vedere nella successiva immagine, viene creato un oggetto `Synthesizer` ed un `Human`; vengono passati al costruttore dell'adapter e vengono create due istanze della classe `Adapter`, dove all'attributo `execute` viene associata la corrispondente funzione da eseguire. Stampando l'attributo `execute` dell'adapter, difatti, viene eseguita la funzione associata e stampato il risultato.

```
myList = [Computer("MyPC")]
mySynth = Synthesizer("MySynth")
myHuman = Human("MyHuman")
var1 = Adapter(mySynth, dict(execute=mySynth.play()))
var2 = Adapter(myHuman, dict(execute=myHuman.speak()))
myList.append(var1)
myList.append(var2)
print(myList[1].execute)
```

```
MySynth is playing a song
```

Altro modo per implementare l'adapter è tramite l'ereditarietà: viene specificata una classe Adapter che eredita da una delle classi da generalizzare e specifica un metodo da invocare in base alla classe dell'istanza passata come parametro all'init:

```
class Adapter(Computer):
    def __init__(self, instance):
        self.instance = instance

    def execute(self):
        if isinstance(self.instance, Synthesizer):
            return self.instance.play()
        if isinstance(self.instance, Human):
            return self.instance.speak()
```

ADAPTER VERSIONE 2

Come possiamo ben vedere, in base alla classe dell'oggetto passato come parametro viene richiamato l'apposito metodo.

```
myList = [Computer("MyPC")]
mySynth = Synthesizer("MySynth")
myHuman = Human("MyHuman")
var1 = Adapter(mySynth)
var2 = Adapter(myHuman)
myList.append(var1)
myList.append(var2)
print(myList[1].execute())
print(myList[2].execute())
```

```
MySynth is playing a song
MyHuman is speaking
```

Altro modo di scrivere l'adapter tramite ereditarietà consiste nel creare una classe che erediti due classi: la classe associata e la classe avente il metodo generalizzato. Si fa prima a capire con un esempio (mostrato nella pagina successiva): viene stabilita una classe adapter riferita alla classe da generalizzare (ad esempio, HumanAdapter si riferisce alla classe Human) ed eredita sia la classe da generalizzare che la classe con il metodo generalizzato (in tal caso Computer). Tale adapter eredita da Computer perché sovrascriverà il metodo execute, il quale sarebbe il metodo generalizzato; eredita da Human perché l'esecuzione del metodo generalizzato execute ritorna il valore di ritorno della funzione associata ad Human, quindi speak.

```
class SynthesizerAdapter(Synthesizer, Computer):
    def __init__(self, obj):
        self.obj = obj

    def execute(self):
        return self.obj.play()

class HumanAdapter(Human, Computer):
    def __init__(self, obj):
        self.obj = obj

    def execute(self):
        return self.obj.speak()
```

ADAPTER VERSIONE 3

```
myList = [Computer("MyPC")]
mySynth = Synthesizer("MySynth")
myHuman = Human("MyHuman")
var1 = SynthesizerAdapter(mySynth)
var2 = HumanAdapter(myHuman)
myList.append(var1)
myList.append(var2)
print(myList[1].execute())
print(myList[2].execute())
```

```
MySynth is playing a song
MyHuman is speaking
```

Proxy Pattern

Il Proxy Pattern fornisce una classe surrogata che nasconde la classe che svolge effettivamente il lavoro: quando si invoca un metodo della classe surrogata, viene utilizzato il metodo della classe nascosta. Sostanzialmente, quindi, tale pattern non fa altro che delegare il proprio lavoro ad un altro oggetto e viene utilizzato in quattro casi:

- Remove proxy: è un proxy contenente un oggetto presente in un diverso spazio di indirizzi. Un esempio davvero pratico è la libreria RPyC (Remote Python Call) che permette di creare oggetti su un server e i server delegati su uno o più client;
- Virtual proxy: è un proxy che permette la creazione di oggetti “pesanti” solo se davvero necessari, tramite la lazy initialization (inizializzazione che tarda la creazione di un oggetto finché quest’ultimo non serve per davvero);
- Protection proxy: è un proxy che limita al client il pieno accesso all’oggetto delegato;
- Smart reference: è un proxy che aggiunge nuove funzioni all’accesso dell’oggetto.

Nell’esempio che vedremo, il proxy fa da interfaccia completa all’oggetto che effettuerà le operazioni. Per interfaccia completa si intende che implementa tutte le funzioni dell’oggetto interno. Ovviamente il proxy può non implementare tutte le funzioni dell’oggetto interno, in modo da limitare le operazioni eseguibili dall’esterno.

È, inoltre, possibile creare un proxy che implementi ogni singola funzione dell’oggetto interno senza rendere esplicita ogni interfaccia.

```

test.py x
1 class InternalClass:
2     def method(self):
3         print("Hello World, my ID is", id(self))
4
5
6 class Proxy:
7     def __init__(self):
8         self._internalObject = InternalClass()
9
10    def method(self):
11        self._internalObject.method()
12
13
14    proxyA = Proxy()
15    proxyB = Proxy()
16    proxyA.method()
17    proxyB.method()

```

Proxy esplicito

```

test.py x
1 class InternalClass:
2     def method(self):
3         print("Hello World, my ID is", id(self))
4
5
6 class GenericProxy:
7     def __init__(self):
8         self._internalObject = InternalClass()
9
10    def __getattr__(self, attrName):
11        return getattr(self._internalObject, attrName, None)
12
13
14    proxyA = GenericProxy()
15    proxyB = GenericProxy()
16    proxyA.method()
17    proxyB.method()

```

Proxy generico

È, inoltre, importante parlare dei virtual proxy, detti anche proxy cumulativi. Con i proxy cumulativi vengono richiamate funzioni ma non vengono eseguite: vengono conservate ed eseguite tutte insieme con l'esecuzione di un particolare metodo. Tale proxy viene utilizzato quando devono esser fatte molteplici operazioni ma non utili in quel preciso momento: si salvano, quindi, i richiami a funzione, in modo da eseguirli non appena serviranno, tramite il richiamo di un'altra funzione. Per fare un esempio, se si richiama `functA` e `functB`, esse non vengono eseguite, bensì vengono salvate in una lista; all'esecuzione del metodo `functExecute`, `functA` e `functB` vengono eseguite. Vediamo un'implementazione di tale proxy:

```
HeavyProxy.py x
1 class HeavyProxy:
2     def __init__(self, cls, *args):
3         self._object = cls(*args)
4         self._params = list()
5
6     def __getattr__(self, attrName):
7         return getattr(self._object, attrName, None)
8
9     def callMethod(self, methodName, *args):
10        callingMethod = getattr(self._object, methodName, None)
11        self._params.append((callingMethod, *args))
12
13    def execute(self):
14        for itemMethod in self._params:
15            callingMethod, *methodParams = itemMethod
16            callingMethod(*methodParams)
```

Aggiornato al 22/10/2019 alle ore 12:37

Sommario

Convenzioni e identificatori

Indentazione del codice

Oggetti, core data type ed etichette

Mutabilità ed immutabilità

La classe bool, int e float

Oggetti iterabili

La classe list, tuple, str, set, frozenset e dict

Shallow copy e deep copy (collezioni)

Operatori ed operazioni

Costrutto IF

Costrutto FOR

Costrutto WHILE

Istruzioni break e continue

Comprensione di lista (FOR abbreviato)

Funzioni

Funzioni anonime (espressioni lambda)

Formattazione dell'output

Input

Lettura e scrittura da file

Moduli

Classi

Operatori magici

Ereditarietà

Iteratori

Superclassi astratte

Metodi statici, metodi di classe e metodi di istanza

Le eccezioni

I decorator (generale)

Proprietà di istanza (getter, setter, deleter)

Decoratore @ensure di classe

Decoratori o superclassi?

Singleton Pattern

Borg Pattern

Adapter Pattern

Proxy Pattern