

Programmazione Avanzata

Python

Francesco Abate

Francesco Abate / frekkanzer2

GitHub Python repository: <https://github.com/frekkanzer2/PythonTests>

[Click here for direct link](#)

Ultimo aggiornamento della guida: 09/12/2019 alle ore 03:41

Sommario

Introduzione	6
Convenzioni e identificatori.....	6
Indentazione del codice.....	6
Oggetti, core data type ed etichette	7
Mutabilità ed immutabilità.....	8
Le classi bool, int e float	9
Oggetti iterabili.....	9
Le classi list, tuple, str, set, frozenset e dict	10
Shallow copy e deep copy (collezioni).....	11
Operatori aritmetici.....	13
Operatori logici	13
Operatori di uguaglianza	13
Operatori di confronto	13
Operatori di appartenenza	13
Operatori di assegnamento.....	13
Operazioni per sequenze.....	14
Operazioni per sequenze mutabili	14
Operazioni per insiemi.....	14
Costrutto IF.....	14
Costrutto FOR	15
Costrutto WHILE	16
Istruzioni break e continue.....	17
Comprensione di lista (FOR abbreviato).....	17
Funzioni	18
Funzioni anonime (espressioni lambda).....	23
Formattazione dell'output	23
Input	24
Lettura e scrittura di file	24
Moduli	26
Introduzione alla Object Oriented Programming.....	27
Classi	27
Operatori magici.....	29
Ereditarietà	31
Iteratori.....	32

Superclassi astratte	33
Metodi statici, metodi di classe e metodi di istanza	35
Le eccezioni.....	36
I decoratori (generale).....	38
Proprietà di istanza (getter, setter, deleter)	41
Decoratore @ensure di classe.....	43
Decoratori o superclassi?	44
Singleton Pattern.....	45
Borg Pattern	47
Adapter Pattern.....	47
Proxy Pattern	50
Chain of Responsibility Pattern	52
Chain of Responsibility: Conventional Chain.....	52
Chain of Responsibility: Coroutine-Based Chain	54
State Pattern.....	56
Mediator Pattern.....	60
Mediator Pattern: mediatore convenzionale.....	60
Mediator Pattern: mediatore basato su coroutine.....	63
Template Method Pattern.....	65
Observer Pattern	68
Facade Pattern.....	71
Context Managers	72
Flyweight Pattern	75
Prototype Pattern.....	76
Concorrenza.....	77
Pacchetto multiprocessing	78
Concorrenza CPU-bound	80
Utilizzare i future e il multiprocessing.....	82
Approfondimento: dizionari e __dict__	85
Approfondimento: decoratori e funzioni	89
Approfondimento: closures.....	95
Approfondimento: common gotchas	97
Argomenti default mutabili	97
Chiusure con late binding.....	97
Approfondimento: iteratori, iterabili e generatori.....	100
Approfondimento: lettura e scrittura su files.....	104

Approfondimento: coroutines.....	108
Approfondimento: la magia di __slots__	111
Approfondimento: Process e ProcessPoolExecutor	112
Approfondimento: Thread e ThreadPoolExecutor	116

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:

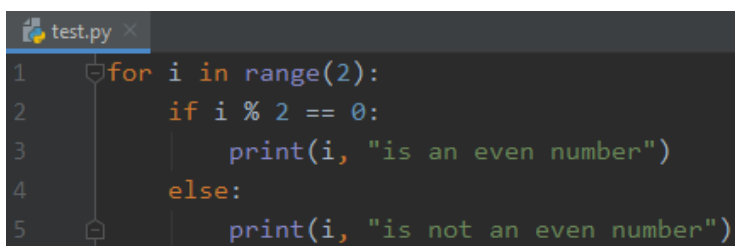
- Nomi di funzioni, metodi e di variabili iniziano sempre con la lettera minuscola;
- Nomi di classi iniziano con la lettera maiuscola;
- In ogni caso si utilizza la notazione CamelCase (ogni parola comincia con la maiuscola, ad esempio testDomain);
- Nomi di costanti sono scritti interamente in maiuscolo.

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.

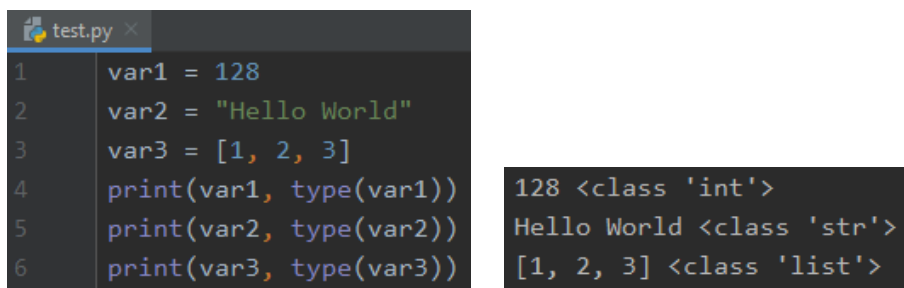
Oggetti, core data type ed etichette

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)`.



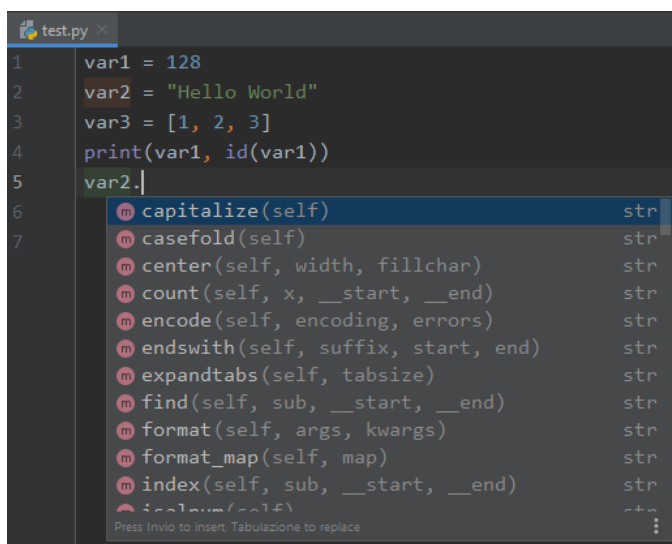
The image shows a code editor window titled 'test.py' with the following code:

```
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.
```

A popup menu is visible over the code, showing a list of attributes for the string object 'var2'. The attributes listed are:

- 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

At the bottom of the popup, it says: "Press Enter to insert. Tabulazione to replace".

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)	

Le classi *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 :(")
```

Le classi 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.](#)

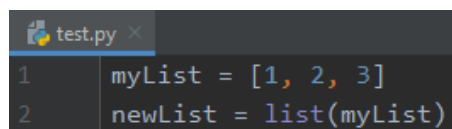
La classe dict rappresenta un dizionario, quindi un insieme di coppie chiave e valore, dove le chiavi devono essere univoche tra loro. Il costruttore è dict(pairs), dove pairs è una lista di coppie (key, value); nel caso non venga passato alcun argomento, viene ritornato un dict vuoto. Tale classe utilizza 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
$len(s)$	Ritorna il numero di elementi appartenenti alla sequenza
$min(s)$	Ritorna il minimo elemento presente nella sequenza
$max(s)$	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 $k-1$
$del\ s[i:k:r]$	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 $s2$
$s1 <= s2$	Ritorna <code>true</code> se $s1$ è sottoinsieme di $s2$
$s1 < s2$	Ritorna <code>true</code> se $s1$ è sottoinsieme stretto di $s2$
$s1 >= s2$	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

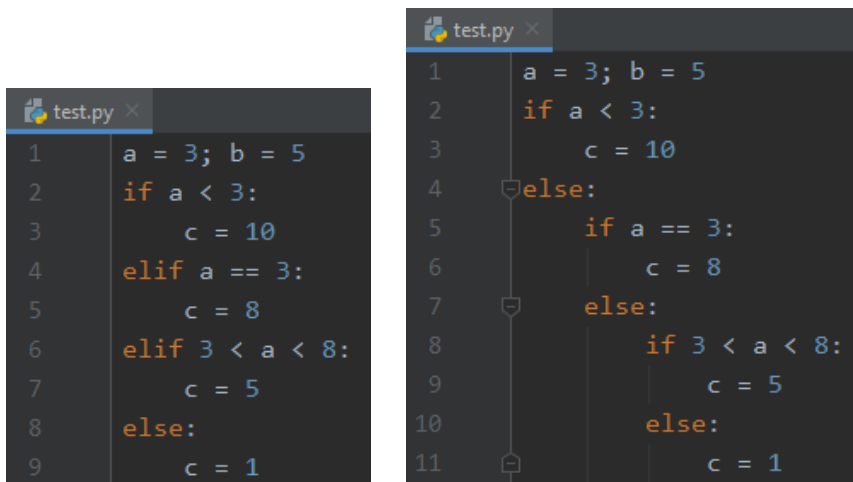
Il costrutto IF permette di intraprendere azioni diverse sulla base del risultato di un test di verità, detto anche "condizione".

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:



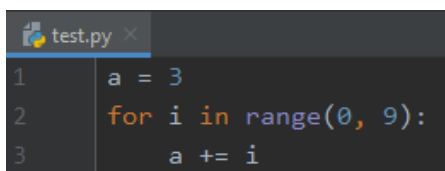
```
test.py x
1 a = 3; b = 5
2 if a < 3:
3     c = 10
4 elif a == 3:
5     c = 8
6 elif 3 < a < 8:
7     c = 5
8 else:
9     c = 1

test.py x
1 a = 3; b = 5
2 if a < 3:
3     c = 10
4 else:
5     if a == 3:
6         c = 8
7     else:
8         if 3 < a < 8:
9             c = 5
10        else:
11            c = 1
```

elif vs else if

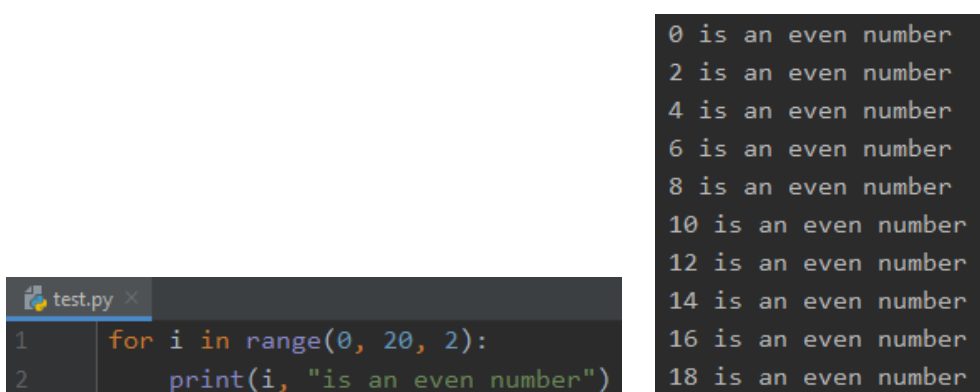
Costrutto FOR

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



```
test.py x
1 a = 3
2 for i in range(0, 9):
3     a += i
```

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.



```
test.py x
1 for i in range(0, 20, 2):
2     print(i, "is an even number")

0 is an even number
2 is an even number
4 is an even number
6 is an even number
8 is an even number
10 is an even number
12 is an even number
14 is an even number
16 is an even number
18 is an even number
```

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: infatti, 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 iterabili, quindi è possibile iterare sulle linee di un file:



```
test.py x
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)
```

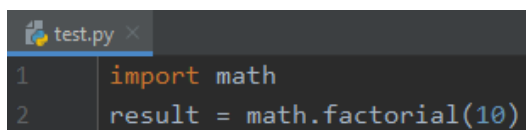
New line: Hello
New line: World
New line: My
New line: Name
New line: Is
New line: Abby

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.

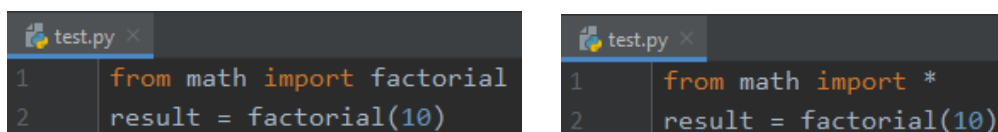
In pratica, quindi, sono utili per decomporre un programma di grandi dimensioni in più file. Particolarità dei moduli è quella di offrire la possibilità di poter riusare classi e funzioni in nuovi progetti. Per utilizzare un modulo, codesto deve essere incluso e può esser fatto in diversi modi:

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



```
test.py x
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:



```
test.py x
1 from math import factorial
2 result = factorial(10)
```

```
test.py x
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.

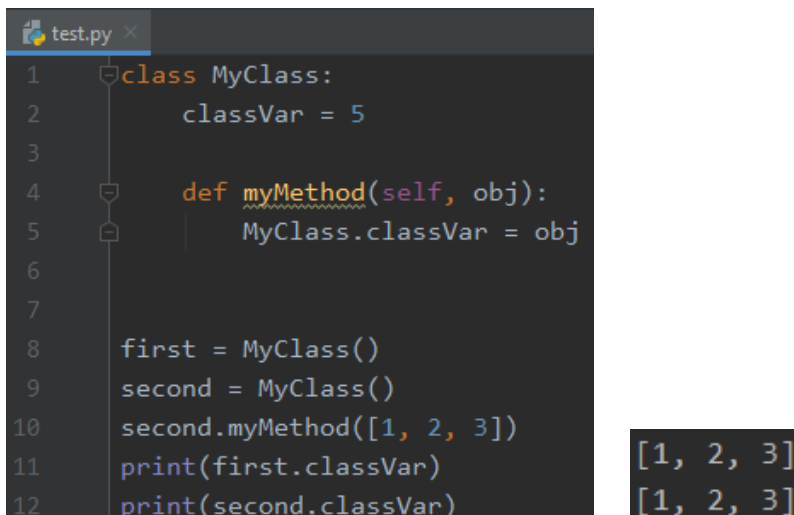
Introduzione alla Object Oriented Programming

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]

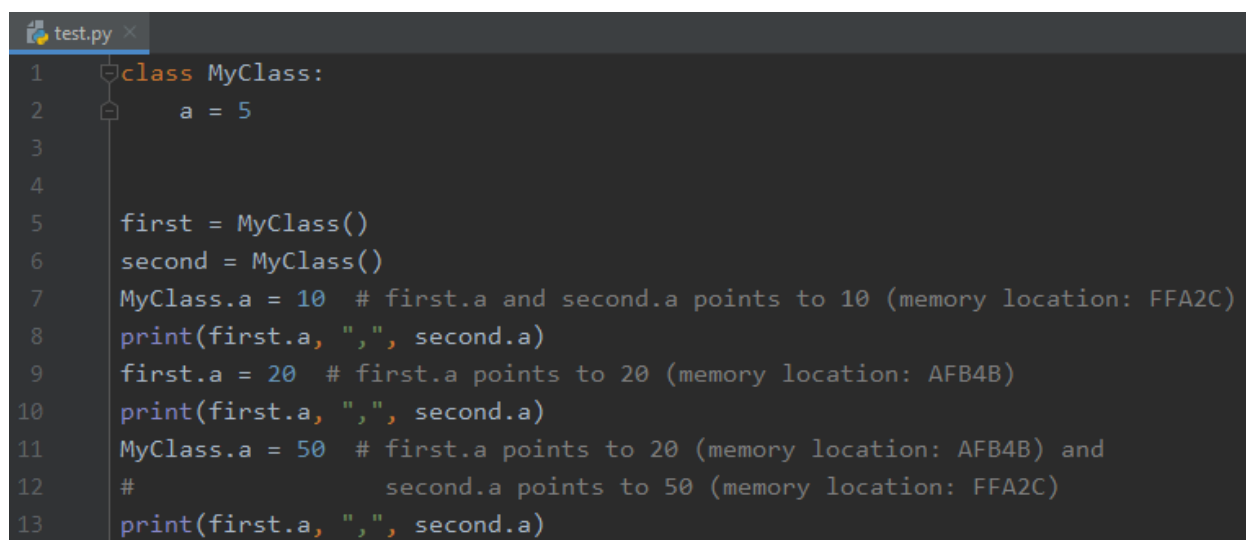
[1, 2, 3]

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.

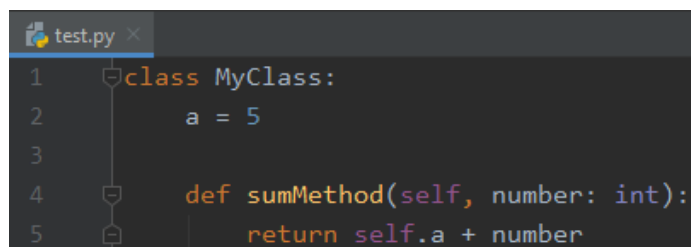
È importante sapere che nel caso si cambi il valore di un attributo di classe richiamandolo da un'istanza, codesto cambiamento verrà rilevato solamente dall'istanza sul quale è stato effettuato: il puntatore viene modificato, quindi quella variabile non punterà più a quella zona di memoria, perdendo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

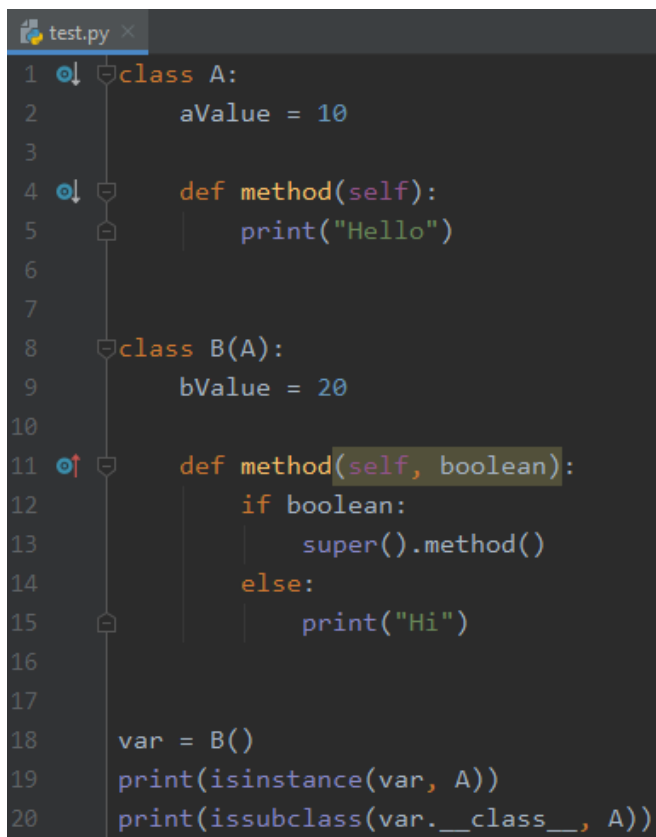
```
test.py x
1 class A:
2     aValue = 10
3
4     def method(self):
5         print("Hello")
6
7
8 class B(A):
9     bValue = 20
10
11     def method(self, boolean):
12         if boolean:
13             super().method()
14         else:
15             print("Hi")
16
17
18 var = B()
19 var.method(True)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

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`.

Attenzione: ciò che è stato illustrato in tale capitolo sono iteratori, siamo tutti d'accordo, ma questo particolare iteratore visto viene anche detto generatore siccome fa uso dello `yield`. Onde capire a fondo iteratori e generatori, è altamente consigliato approfondire leggendo il capitolo "Approfondimento: iteratori, iterabili e generatori".

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`

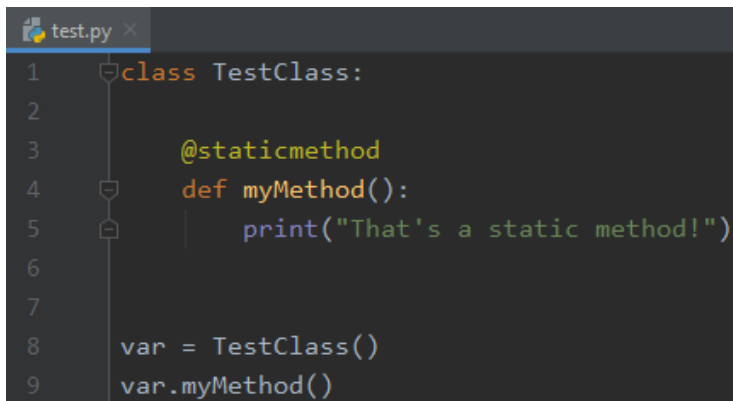
Durante il richiamo, la classe viene passata a prescindere, anche se non la si definisce.

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

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

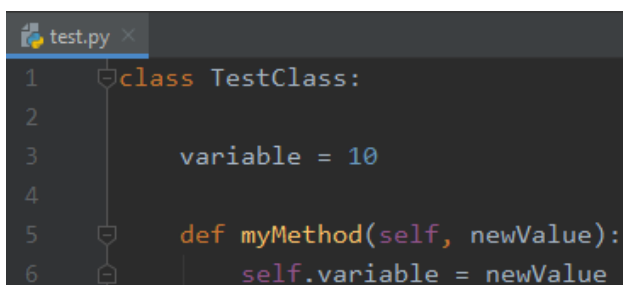
I metodi statici non sono di classe ma vengono qualificati tramite essa. A questi metodi non viene passato alcun argomento, per cui sono semplici funzioni che vivono nello scope locale della classe.

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 decoratori (generale)

I decoratori 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 decorator. I decorator possono esser specificati anche in cascata, assegnandone molteplici allo stesso oggetto.

I decorator 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.

ATTENZIONE: i decorator sono un argomento decisamente ostile. Per comprenderlo a meglio, è altamente consigliato leggere il capitolo "Approfondimento: decorator e funzioni".

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)
```

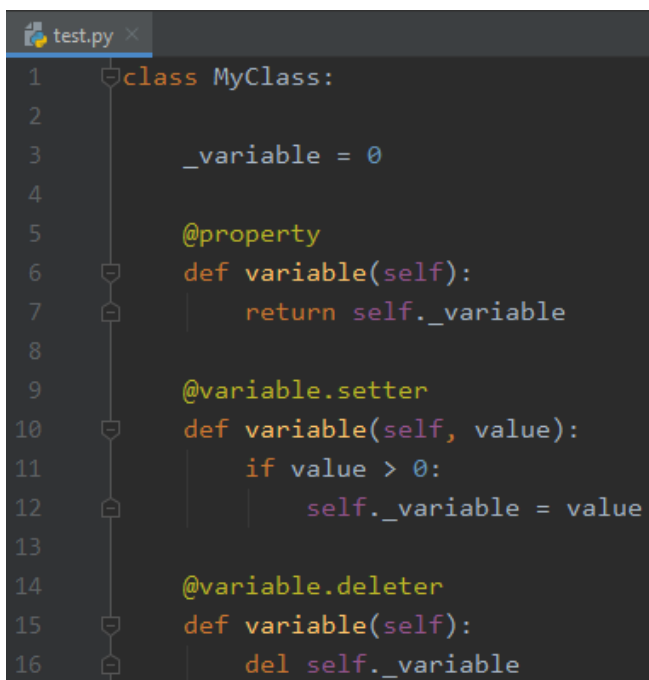
Nell'esempio soprastante vediamo denotata una variabile privata chiamata `_variable`, la quale viene settata a 0 dal costruttore. Essendo `_variable` inaccessibile (per convenzione) dall'esterno della classe, vengono stabiliti dei metodi per permetterne l'accesso, la modifica e l'eliminazione. È vero, 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:



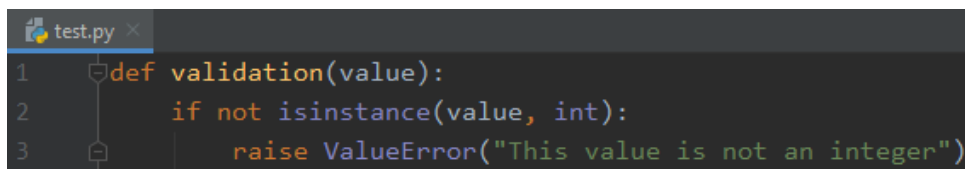
```
test.py x
1 class MyClass:
2
3     _variable = 0
4
5     @property
6     def variable(self):
7         return self._variable
8
9     @variable.setter
10    def variable(self, value):
11        if value > 0:
12            self._variable = value
13
14    @variable.deleter
15    def variable(self):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

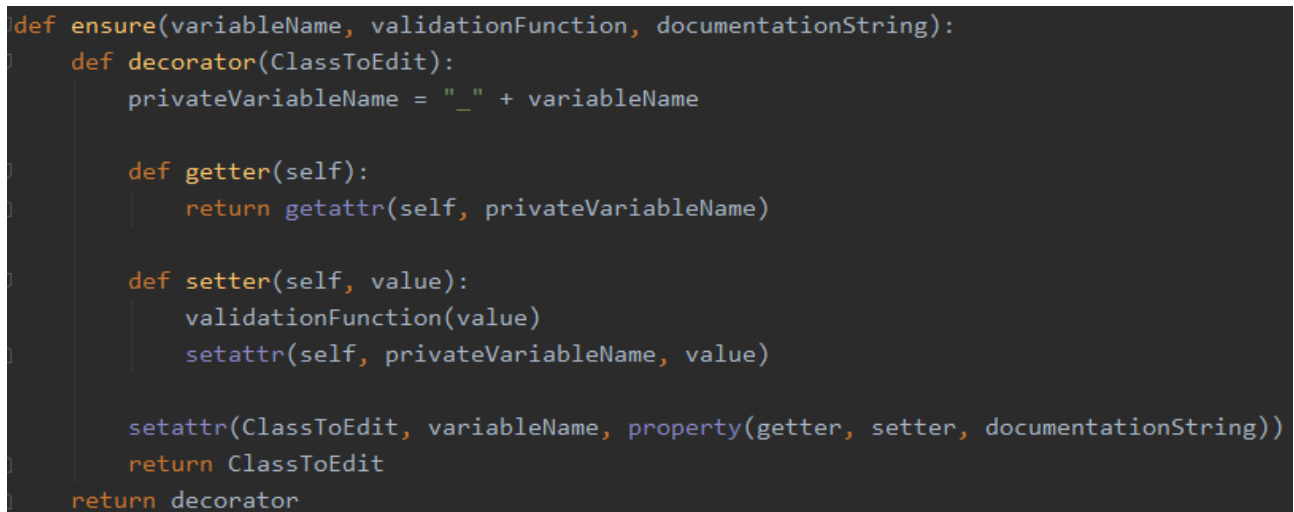
Una classe è composta da molteplici variabili, delle quali si sente spesso l'esigenza dei metodi getter, setter e deleter: non è possibile scrivere tanto codice per dei metodi riferiti a singole variabili. Per fortuna, Python permette la creazione di decoratori che evitino la scrittura di cotanto codice, 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

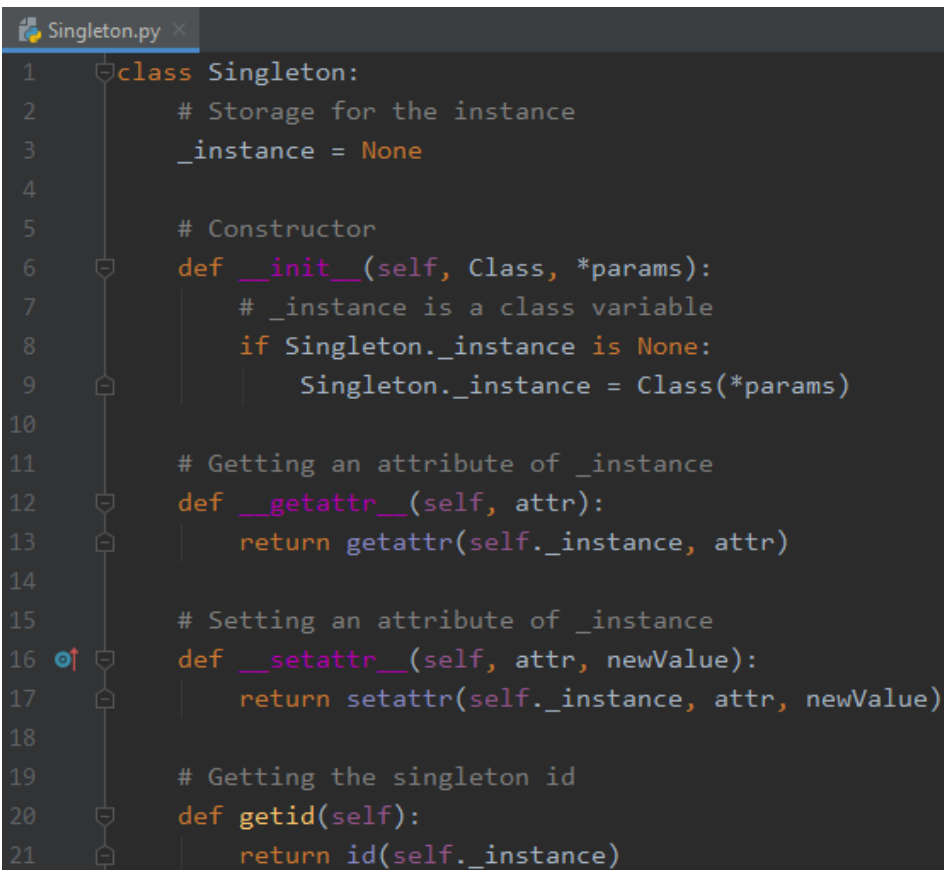
Il Singleton Pattern è ciò che si usa quando si ha bisogno di un'unica istanza di una determinata classe nell'intero programma. Sostanzialmente, si dichiara una classe Singleton che fa da wrapper per un'oggetto di classe la cui classe è specificata internamente, il quale sarà unico in tutto il programma: 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:



```
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

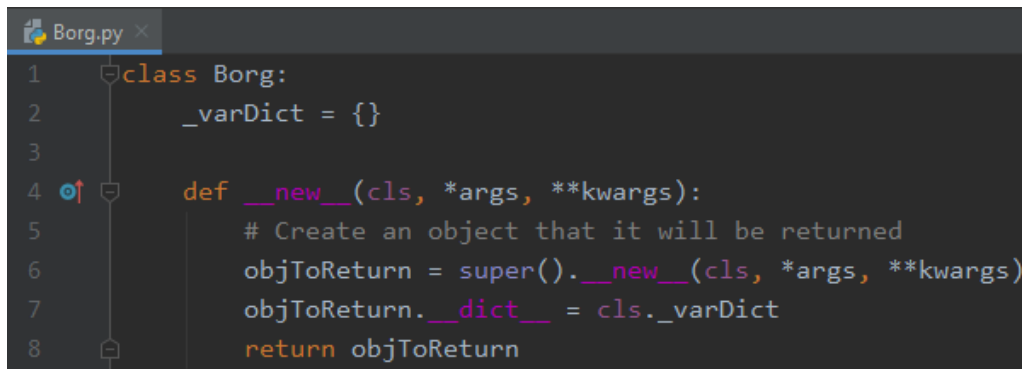
Grazie al costruttore di questo nuovo singleton, la classe Singleton provvederà alla creazione di un oggetto della classe da noi scelta solamente nel caso non sia stato già creato un altro singleton.

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 funziona, è bene introdurre la variabile di istanza `__dict__` e la funzione `__new__`.

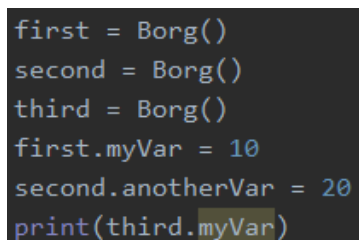
La funzione `__new__` istanzia un oggetto vuoto, il quale verrà inizializzato tramite la funzione `__init__`; la variabile di istanza `__dict__` è un dizionario contenente tutti gli attributi di quella determinata istanza.

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.



```
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)

```

Chain of Responsibility Pattern

Il pattern Chain of Responsibility è utilizzato per separare il codice che effettua una determinata richiesta dal codice che effettivamente la elabora. Sostanzialmente, la richiesta viene passata in una catena di destinatari e viaggia finché non trova il corretto destinatario: il primo destinatario può elaborare la richiesta o passarla al prossimo nella catena; il secondo destinatario può effettuare le stesse scelte del primo e così via, finché non si raggiunge l'ultimo destinatario che può decidere se scartare la richiesta o lanciare un'eccezione.

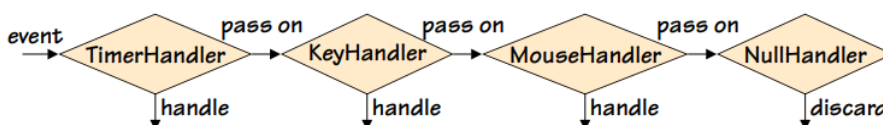
Immaginiamo di avere un'interfaccia utente che riceva eventi da gestire: alcuni di questi eventi provengono dall'interazione dell'utente, come la pressione del mouse o di un tasto sulla tastiera, mentre altri eventi provengono direttamente dal sistema, come i timer events. Tali eventi dovranno essere gestiti da alcuni handlers, posti in delle catene. Vedremo due tipi di catene: il primo tipo sarà una catena di handlers (Conventional Chain), il secondo tipo adotterà un approccio basato sulle pipeline usando la coroutines (Coroutine-Based Chain).

Chain of Responsibility: Conventional Chain

In questa sezione vedremo la Conventional Chain composta da handlers di eventi, dove ad ogni evento corrisponde una propria event-handling class.

```
handler1 = TimerHandler(KeyHandler(MouseHandler(NullHandler())))
```

Sostanzialmente, tale catena segue la logica sottostante:



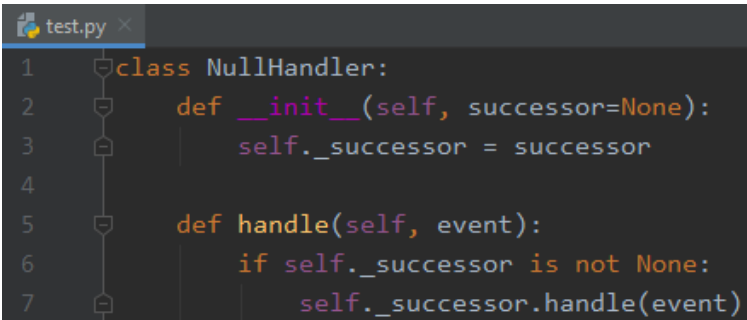
L'ordine con cui si creano gli handlers non importa siccome ognuno gestisce un proprio tipo di eventi. Nel seguente esempio viene eseguito un while true in cui ad ogni evento viene fatto un

controllo: se l'evento è di tipo TERMINATE allora non si gestiscono più gli eventi, altrimenti l'evento si passa alla catena di handlers per essere gestito.

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    handler1.handle(event)
```

Event è un iteratore di eventi

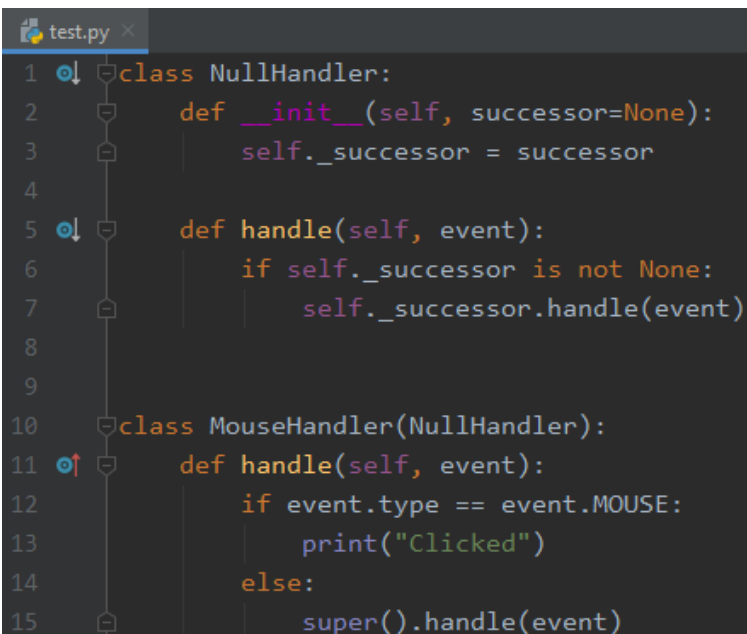
Notiamo che nella catena è presente un particolare handler: NullHandler.



```
test.py x
1 class NullHandler:
2     def __init__(self, successor=None):
3         self._successor = successor
4
5     def handle(self, event):
6         if self._successor is not None:
7             self._successor.handle(event)
```

NullHandler serve come infrastruttura per gli handler e, sostanzialmente, non gestisce alcun evento. Il suo compito è il seguente: nel caso venga istanziata un'istanza di tipo NullHandler con tanto di parametro successore, la gestione dell'evento passa al parametro successore (il quale sarà un altro handler), altrimenti tale evento non viene gestito.

Ciò ci fa capire che la classe NullHandler sarà superclasse di tanti altri handler, come il seguente che vedremo: MouseHandler.

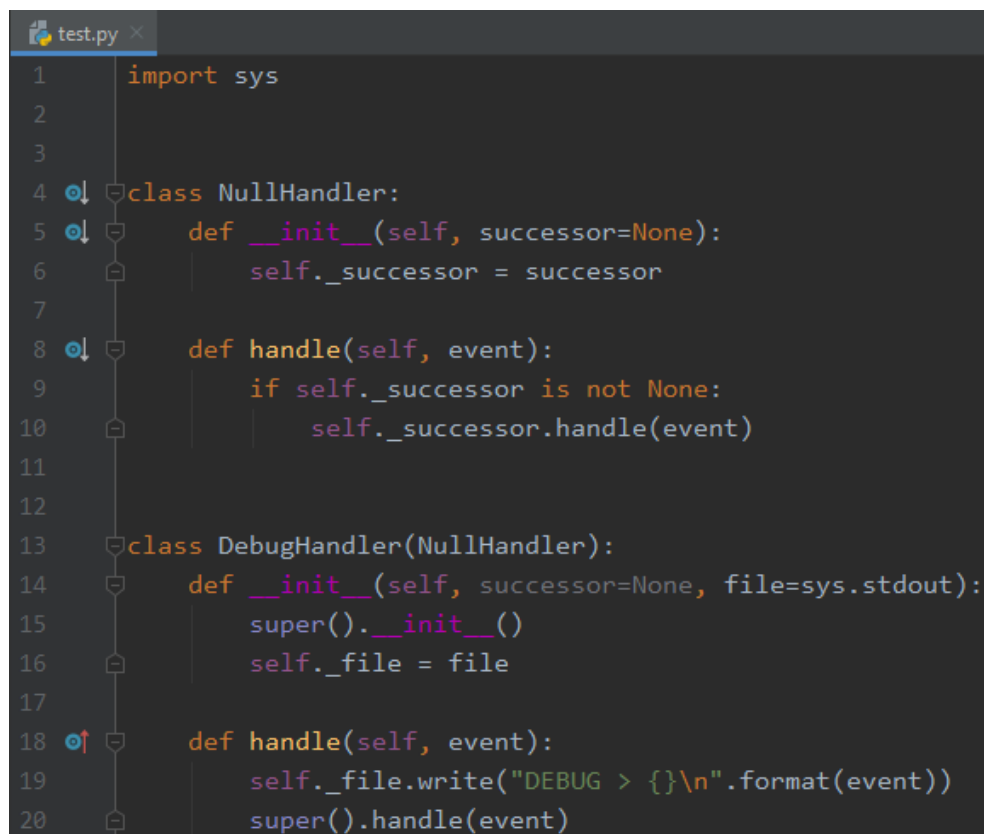


```
test.py x
1 class NullHandler:
2     def __init__(self, successor=None):
3         self._successor = successor
4
5     def handle(self, event):
6         if self._successor is not None:
7             self._successor.handle(event)
8
9
10 class MouseHandler(NullHandler):
11     def handle(self, event):
12         if event.type == event.MOUSE:
13             print("Clicked")
14         else:
15             super().handle(event)
```

Notiamo come MouseHandler estenda NullHandler: l'init non viene reimplementato siccome quello della superclasse crea con successo il successore senza alcun problema, mentre viene reimplementato il metodo handle. La reimplementazione del metodo handle offre la gestione dell'evento nel caso sia del tipo supportato dall'handler (in questo caso, l'evento dovrà essere di tipo MOUSE per esser gestito); nel caso l'evento non sia del tipo supportato dall'handler, la sua

gestione passa all'handler successivo specificato nella variabile `successore`, richiamando tramite `super()` la funzione `handle` della superclasse, la quale passa il controllo dell'evento al prossimo handler. Ovviamente tutti gli altri handler avranno la stessa struttura, cambierà solo il tipo supportato e come verrà eventualmente gestito.

Altro tipo di handler è il `DebugHandler`, il quale estende `NullHandler`: è differente dai normali handler siccome non gestisce eventi, bensì crea dei report su file quando accade un evento. È necessario che tale handler sia il primo della catena.



```
1  import sys
2
3
4  class NullHandler:
5      def __init__(self, successor=None):
6          self._successor = successor
7
8      def handle(self, event):
9          if self._successor is not None:
10             self._successor.handle(event)
11
12
13  class DebugHandler(NullHandler):
14      def __init__(self, successor=None, file=sys.stdout):
15          super().__init__()
16          self._file = file
17
18      def handle(self, event):
19          self._file.write("DEBUG > {}\n".format(event))
20          super().handle(event)
```

Chain of Responsibility: Coroutine-Based Chain

Come ben sappiamo, un generatore è una funzione che ha una o più espressioni `yield` al posto dell'istruzione `return`. Una coroutine usa anch'essa l'espressione `yield`, ma con un differente comportamento: viene eseguito un loop infinito e si sospende ogni volta che si raggiunge uno `yield`, in attesa di un valore da gestire con quest'ultima espressione. Se alla coroutine viene passato un valore, quest'ultimo verrà gestito dalla `yield` e continuerà a ciclare, fino ad arrivare nuovamente alla `yield` che attenderà un nuovo valore. I valori vengono inseriti nella coroutine tramite i metodi `send()` o `throw()`.

In Python qualsiasi funzione o metodo che usi `yield` viene considerato un generatore; usando il decoratore `@coroutine` e usando un ciclo infinito, possiamo trasformare un generatore in una coroutine.

Nella prossima pagina vediamo il codice del decoratore `@coroutine`.

```

test.py x
1 def coroutine(function):
2     def wrapper(*args, **kwargs):
3         generator = function(*args, **kwargs)
4         next(generator)
5         return generator
6     return wrapper

```

Il wrapper chiama la funzione generatore e conserva nella variabile generator il generatore ritornato dall'invocazione. Conservato il generatore, verrà fatto avanzare tramite la next per farlo arrivare alla prima espressione yield. Viene ritornato, quindi, il generatore bloccato sulla yield, considerato come coroutine siccome sarà pronto a ricevere un valore. Se mandiamo un valore alla coroutine utilizzando il metodo send(value), tale valore verrà ricevuto ed elaborato, per poi riprendere l'esecuzione.

Siccome dalla coroutine è possibile mandare e ricevere valori, è possibile creare delle pipelines, utilizzabili per gestire anche le catene di handler. Di conseguenza, non si sentirà il bisogno di associare una variabile per il successore siccome la catena verrà gestita dal generatore.

Nel seguente esempio, creiamo una coroutine utilizzando chiamate a funzioni innestate:

```
pipeline = key_handler(mouse_handler(timer_handler()))
```

Ogni funzione chiamata è una coroutine, dove ognuna si esegue fino alla prima espressione yield. Invece di passare un NullHandler, all'ultimo handler non viene passato nulla. Nel seguente esempio viene eseguito un while true in cui ad ogni evento viene fatto un controllo: se l'evento è di tipo TERMINATE allora non si gestiscono più gli eventi, altrimenti l'evento si passa alla pipeline di coroutines per esser gestito.

```

while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    pipeline.send(event)

```

Vediamo ora come vengono gestite le coroutine a catena: la coroutine accetta un successore coroutine, in modo tale che se l'evento non si possa gestire nell'attuale coroutine, viene mandato alla prossima; il decoratore @coroutine assicura che key_handler venga eseguito fino alla prima yield expression: quindi quando la pipeline viene creata, la funzione ha raggiunto la sua espressione yield e rimane bloccata finché yield non riceve un valore (ovviamente viene bloccata solo la coroutine, non l'intero programma). Ovviamente, ogni valore mandato alla coroutine sarà sempre un evento in questo esempio. Se il tipo dell'evento passato alla coroutine è uguale al tipo di evento che la coroutine in questione dovrebbe gestire, allora l'evento viene gestito, altrimenti viene passato alla prossima coroutine. Nel caso una prossima coroutine non esista (quindi nel caso l'argomento successor sia None), l'evento viene scartato. Una volta gestito o scartato l'evento, la coroutine ritorna all'inizio del loop while True e si blocca di nuovo all'istruzione yield, attendendo un nuovo valore tramite il metodo send(event).

```

test.py x
1  def coroutine(function):
2      def wrapper(*args, **kwargs):
3          generator = function(*args, **kwargs)
4          next(generator)
5          return generator
6
7      return wrapper
8
9
10 @coroutine
11 def key_handler(successor=None):
12     while True:
13         event = (yield)
14         if event.kind == event.KEYPRESS:
15             print("Pressed")
16         elif successor is not None:
17             successor.send(event)

```

Come per la Conventional Chain, anche con le coroutine abbiamo un gestore debug:

```

test.py x
1  import sys
2
3
4  def coroutine(function):
5      def wrapper(*args, **kwargs):
6          generator = function(*args, **kwargs)
7          next(generator)
8          return generator
9
10     return wrapper
11
12
13 @coroutine
14 def debug_handler(successor, file=sys.stdout):
15     while True:
16         event = (yield)
17         file.write("DEBUG > {}".format(event))
18         successor.send(event)

```

La funzione `debug_handler()` attende un evento, ne stampa i dettagli e lo trasmette al prossimo handler.

State Pattern

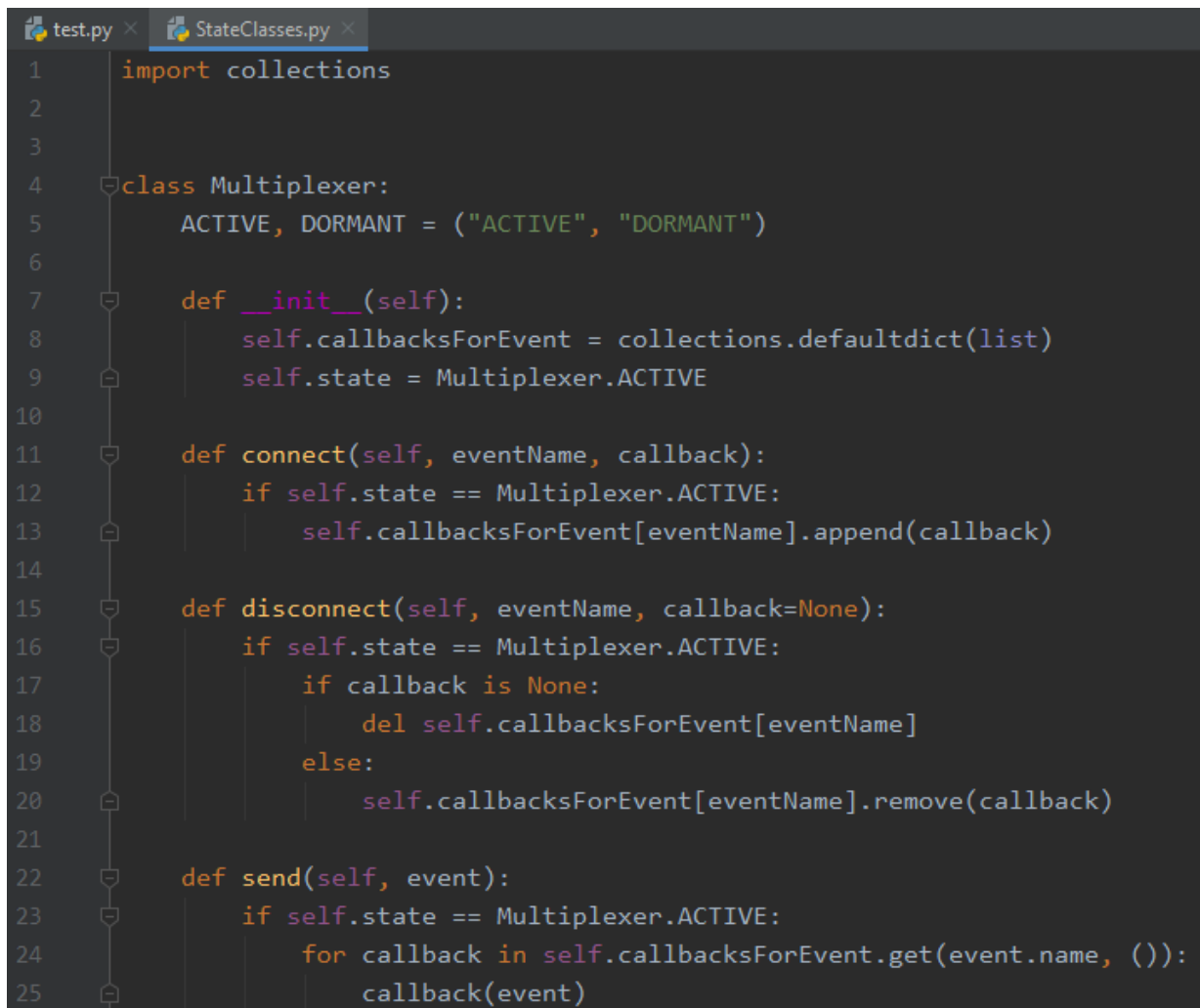
Lo State Pattern viene utilizzato per gestire oggetti il cui comportamento dipende da un determinato stato. Per illustrare il funzionamento di tale pattern, utilizzeremo una classe multiplexer possedente due stati: quando il multiplexer è nello stato attivo, può “accettare

connessioni”, cioè coppie (evento, callback) dove callback è una funzione richiamabile, altrimenti non succede nulla. Una volta stabilite le dovute connessioni, ogni volta che viene inviato un evento al multiplexer, le funzioni callback associate vengono invocate.

Vediamo un esempio: inviando un certo numero di eventi random al multiplexer, vengono stampati i conteggi degli eventi ricevuti dalle callback. Il multiplexer cambia stato ogni 100 eventi ricevuti, quindi il passaggio di 300 eventi randomici avrebbe i seguenti effetti:

```
After 100 active events: cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events: cars=303 vans=83 trucks=30 total=416
```

Iniziamo a vedere il codice del Multiplexer, il quale verrà comunque spiegato più avanti:



```
1  import collections
2
3
4  class Multiplexer:
5      ACTIVE, DORMANT = ("ACTIVE", "DORMANT")
6
7      def __init__(self):
8          self.callbacksForEvent = collections.defaultdict(list)
9          self.state = Multiplexer.ACTIVE
10
11     def connect(self, eventName, callback):
12         if self.state == Multiplexer.ACTIVE:
13             self.callbacksForEvent[eventName].append(callback)
14
15     def disconnect(self, eventName, callback=None):
16         if self.state == Multiplexer.ACTIVE:
17             if callback is None:
18                 del self.callbacksForEvent[eventName]
19             else:
20                 self.callbacksForEvent[eventName].remove(callback)
21
22     def send(self, event):
23         if self.state == Multiplexer.ACTIVE:
24             for callback in self.callbacksForEvent.get(event.name, ()):
25                 callback(event)
```

Per ora ci interessa sapere solo un paio di cose: riguardo il costruttore di Multiplexer, viene conservato un dizionario che, nel caso non contenga la key cercata, ritorna un determinato valore di default passato come parametro, in tal caso una lista vuota; riguardo il metodo connect, nel caso lo stato sia attivo viene accodato il metodo di callback alla lista associata alla key del dizionario, quindi all’evento (perché l’evento è la key del dizionario). In poche parole, Multiplexer mantiene salvate le funzioni di callback in base all’evento in questione.

Altre due classi da vedere onde capire il funzionamento del programma sono le seguenti: Event e Counter.

```
28 class Event:
29     def __init__(self, name, count=1):
30         if not name.isidentifier():
31             raise ValueError("names must be valid identifiers")
32         self.name = name
33         self.count = count
34
35
36 class Counter:
37     def __init__(self, *names):
38         self.anonymous = not bool(names)
39         if self.anonymous:
40             self.count = 0
41         else:
42             for name in names:
43                 if not name.isidentifier():
44                     raise ValueError("names must be valid identifiers")
45                 setattr(self, name, 0)
46
47     def __call__(self, event):
48         if self.anonymous:
49             self.count += event.count
50         else:
51             count = getattr(self, event.name)
52             setattr(self, event.name, count + event.count)
```

Parleremo delle classi appena viste tra poco, ora vediamo il funzionamento del programma. Sia chiaro che, per comodità, tali classi sono state inserite in un modulo apposito, onde mantenere il codice ordinato.

Iniziamo col vedere la prima parte del programma:

```
test.py x StateClasses.py x
1 from StateClasses import *
2
3 totalCounter = Counter()
4 carCounter = Counter("cars")
5 commercialCounter = Counter("vans", "trucks")
6 instanceMultiplexer = Multiplexer()
7 coupleEventCallback = [("cars", carCounter),
8                         ("vans", commercialCounter),
9                         ("trucks", commercialCounter)]
10 for eventName, eventCallback in coupleEventCallback:
11     instanceMultiplexer.connect(eventName, eventCallback)
12     instanceMultiplexer.connect(eventName, totalCounter)
```

Si inizia creando due contatori, i quali sono callable siccome scrivono al loro interno il metodo magico `__call__()`, quindi possono essere usati come callback. Sostanzialmente, ogni contatore mantiene un conteggio indipendente per nome fornito, eccetto il `totalCounter` che mantiene un singolo conteggio globale. Si passa poi alla creazione del multiplexer, all'inizio attivo per default, per poi creare una lista di tuple, dove ogni tupla non è altro che una coppia (evento, callback), la quale verrà scansionata per connettere ogni evento all'associata funzione di callback: in tal caso, nel for, ogni evento viene collegato sia alla propria funzione di callback sia al `totalCounter`, il quale verrà collegato ad ogni evento della lista di tuple. Difatti, `carCounter()` sarà associata all'evento cars, la funzione `commercialCounter()` sarà associata agli eventi vans e trucks, mentre la funzione `totalCounter()` sarà associata a tutti e tre gli eventi.

Vediamo ora la seconda parte del programma:

```
13     for event in generate_random_events(100):
14         instanceMultiplexer.send(event)
15     print("After 100 active events: cars={}, vans={}, trucks={}, total={}"
16           .format(carCounter.cars, commercialCounter.vans,
17                 commercialCounter.trucks, totalCounter.count))
```

In questa porzione di codice generiamo un centinaio di eventi casuali e li inviamo uno per uno al multiplexer tramite il metodo `send(event)`: se, per esempio, un evento è "cars", il multiplexer richiamerà le funzioni `carCounter()` e `totalCounter()`, passando l'evento come unico argomento per ogni chiamata (controllare il metodo `send` della classe `Multiplexer`). Similmente, ovviamente, se l'evento è "vans" o "trucks", vengono richiamate le funzioni `commercialCounter()` e `totalCounter()`.

Ora torniamo alla classe `Counter` vista prima e analizziamola passo passo:

```
class Counter:
    def __init__(self, *names):
        self.anonymous = not bool(names)
        if self.anonymous:
            self.count = 0
        else:
            for name in names:
                if not name.isidentifier():
                    raise ValueError("names must be valid identifiers")
                setattr(self, name, 0)

    def __call__(self, event):
        if self.anonymous:
            self.count += event.count
        else:
            count = getattr(self, event.name)
            setattr(self, event.name, count + event.count)
```

Il costruttore controlla se vengono forniti dei nomi quando la classe viene istanziata: in caso negativo, viene creato un attributo di istanza contatore; in caso positivo, per ogni nome passato come argomento viene creato un attributo di istanza contenente un contatore. Sostanzialmente, quindi, per ogni nome passato come argomento al costruttore di `Counter`, viene creato un proprio

contatore. Per esempio, all'istanza carCounter viene assegnato un attributo self.cars e all'istanza commercialCounter vengono associati gli attributi self.vans e self.trucks.

Quando viene richiamata un'istanza di Counter, la chiamata viene passata al metodo magico `__call__()`: se il contatore non è associato ad un evento preciso (quindi anonimo, come totalCounter) viene incrementato, altrimenti si recupera l'attributo contatore corrispondente all'evento, per poi aggiornare il valore dell'attributo con la somma del vecchio conteggio più quello del nuovo evento.

Ora passiamo alla classe Event:

```
class Event:
    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```

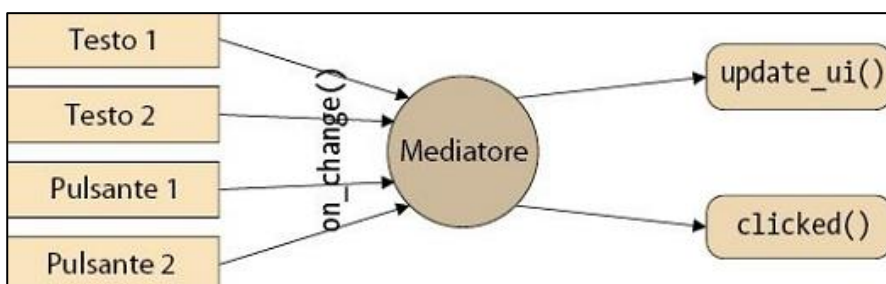
La classe event, come possiamo ben vedere, fa solo da wrapper di dati, quindi conserva per ogni eventName un contatore associato.

Concludendo, quindi, lo scopo di codesto esempio è quello di mostrare cosa può fare lo State Pattern: attribuendo uno stato ad un oggetto, le sue funzionalità cambieranno in base allo stato che esso assume.

Mediator Pattern

Il Mediator Pattern permette la creazione di un oggetto detto mediatore in grado di ottenere interazioni tra oggetti che non hanno una diretta conoscenza tra loro. Tale pattern è particolarmente utilizzato per la programmazione di GUI. L'esempio che vedremo riguarderà proprio le GUI, precisamente un form, e ne vedremo due implementazioni: la prima sarà convenzionale, la seconda utilizzerà le coroutine.

Sostanzialmente, il ruolo del mediatore in tale ambito è il seguente:



Mediator Pattern: mediatore convenzionale

In tale paragrafo vediamo la creazione di un mediatore convenzionale, ovvero di una classe in grado di gestire le interazioni, in tal caso di un form.

La classe Form ha due widget per l'inserimento di testo, destinati al nome e alla mail dell'utente, e due pulsanti, quali sono OK e Cancel.

```

class Form:
    def __init__(self):
        self.create_widgets()
        self.create_mediator()

    def create_widgets(self):
        self.nameText = Text()
        self.emailText = Text()
        self.okButton = Button("OK")
        self.cancelButton = Button("Cancel")

    def create_mediator(self):
        self.mediator = Mediator(((self.nameText, self.update_ui),
                                   (self.emailText, self.update_ui),
                                   (self.okButton, self.clicked),
                                   (self.cancelButton, self.clicked)))
        self.update_ui()

    def update_ui(self, widget=None):
        self.okButton.enabled = (bool(self.nameText.text) and
                                bool(self.emailText.text))

    def clicked(self, widget):
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")

```

Nel costruttore della classe Form, oltre al creare i due widget per l'inserimento del testo e i due pulsanti, viene creato un mediatore, il quale riceve una o più coppie (widget, callable), che descrivono le relazioni che il mediatore deve supportare: sostanzialmente, ad ogni item (in questo caso ad ogni widget) viene associata una funzione che verrà richiamata dopo una determinata azione su quel determinato item. Ad esempio, se il testo di uno dei widget per l'inserimento del testo cambia, allora viene richiamata la funzione `update_ui`, mentre se viene cliccato uno dei due pulsanti viene richiamata la funzione `clicked`. Una volta creato il mediatore, viene invocata la `update_ui` per inizializzare il form: quest'ultimo metodo disabilita il pulsante OK se almeno uno dei due widget per l'inserimento del testo non contiene del testo, altrimenti lo abilita. Naturalmente tale metodo deve essere invocato ogni volta che il testo dei widget cambia. Il metodo `clicked`, invece, deve essere invocato ogni volta che si clicca un pulsante.

Riguardo la classe Mediator, possiede un dizionario le cui chiavi sono widget e i cui valori sono liste di uno o più callable: a questo scopo viene utilizzato il dizionario di default, già visto nel capitolo riguardante lo State Pattern (pagina 56). Quando si accede ad un elemento di un dizionario di default, se l'elemento non è presente nel dizionario, viene creato e aggiunto il valore di default fornito al costruttore del dizionario (in tal caso è un oggetto list, quindi verrebbe creata una lista vuota). Nel costruttore di Mediator, una volta creato il dizionario di default, vengono iterate le

coppie passategli in `widgetCallablePairs`, salvandole nel dizionario come coppie (`widget`, `callable`), dove ogni `widget` può anche avere più `callables`.

```
class Mediator:
    def __init__(self, widgetCallablePairs):
        self.callablesForWidget = collections.defaultdict(list)
        for widget, caller in widgetCallablePairs:
            self.callablesForWidget[widget].append(caller)
            widget.mediator = self

    def on_change(self, widget):
        callables = self.callablesForWidget.get(widget)
        if callables is not None:
            for caller in callables:
                caller(widget)
        else:
            raise AttributeError("No on_change method registered for {}".format(widget))
```

Per ogni `widget`, i metodi vengono salvati nel dizionario nell'ordine in cui appaiono nelle coppie; se l'ordine non ci interessasse, potremmo utilizzare il `set` anziché la `list`. Inoltre, ad ogni `widget` iterato viene associato il mediatore.

Ogni oggetto mediato deve essere sottoclasse di `Mediated`:

```
class Mediated:
    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

Ogni volta che un oggetto mediato presenta un cambiamento dello stato, viene richiamato il proprio metodo `on_change`, il quale richiamerà il metodo `on_change` del mediatore associato: tale metodo recupera i `callables` associati a quel determinato `widget` e li esegue uno ad uno. Poiché la classe `Mediated` non viene modificata ed intaccata da nessun'altra classe, sarebbe stato possibile renderla come decoratore (decoratori o superclassi? Pagina 43).

Vediamo nella prossima pagina, giusto per completezza, le classi `Text` e `Button`.

Come ben vedremo, ad un cambiamento di `Text` o alla pressione di `Button` viene invocato il metodo `on_change`. Sostanzialmente, quindi, alla presenza di un evento su un oggetto a cui è legato, il Mediatore non fa altro che compiere azioni di qualunque tipo.

```

class Text(Mediated):
    def __init__(self, text=""):
        super().__init__()
        self._text = text

    @property
    def text(self):
        return self._text

    @text.setter
    def text(self, newText):
        if self.text != newText:
            self._text = newText
            self.on_change()

class Button(Mediated):
    def __init__(self, text=""):
        super().__init__()
        self.enabled = True
        self.text = text

    def click(self):
        if self.enabled:
            self.on_change()

```

Concludendo, ad un'azione effettuata su un oggetto mediato (quindi collegato in qualche modo ad un mediatore) corrisponde la gestione di esso da parte del mediatore, il quale cercherà nel suo dict di oggetti mediati come chiave l'oggetto su cui è stata effettuata l'azione, per poi invocare tutte le funzioni ad esso associate, poste come values del dizionario.

Mediator Pattern: mediatore basato su coroutine

Un mediatore può essere considerato come una pipeline che riceve messaggi per poi passarli agli oggetti interessati: ciò può esser realizzabile tramite le coroutine. La logica del precedente capitolo consisteva nell'associare coppie di widget e metodi ad un mediatore, e ogni volta che il widget notificava un'azione, il mediatore richiamava i metodi associati tramite le coppie. In tal caso, ad ogni widget viene associato un mediatore, il quale non è altro che una pipeline di coroutine: ad ogni azione sul widget, esso viene inviato nella pipeline, in modo tale che la coroutine possa scegliere che azione intraprendere in risposta ad un cambiamento del widget.

In questo capitolo alcuni codici non verranno mostrati siccome saranno uguali a quelli del capitolo precedente, quindi nel caso non venga mostrata qualche classe è bene fare riferimento al capitolo precedente.

Partiamo dalla creazione del form, nella fase in cui si crea il mediatore: nella versione con coroutine non abbiamo bisogno di una classe Mediator, bensì si crea direttamente una pipeline di coroutine utilizzando due funzioni: `_update_ui_mediator` e `_clicked_mediator`, entrambi metodi di Form sui quali torneremo tra poco.

```
def create_mediator(self):
    self.mediatorPipeline = self._update_ui_mediator(
        self._clicked_mediator()
    )
    for widget in (self.nameText, self.emailText,
                  self.okButton, self.cancelButton):
        widget.mediator = self.mediatorPipeline
    self.mediatorPipeline.send(None)
```

I metodi con cui si crea la pipeline sono i seguenti (ci torneremo comunque tra poco):

```
@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)

@coroutine
def _clicked_mediator(self, successor=None):
    while True:
        widget = (yield)
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
        elif successor is not None:
            successor.send(widget)
```

Ottenuta la pipeline, la si collega all'attributo mediator di ogni widget, per poi inviare None: poiché nessun widget è None, non scattano azioni specifiche ma vengono eseguite azioni nel form, in tal caso l'abilitazione/disabilitazione del pulsante OK in `_update_ui_mediator`.

Analizziamo ora le due coroutine di tale esempio partendo da `_update_ui_mediator`:

```
@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)
```

In tale coroutine, ogni volta che un widget riporta un cambiamento, viene passato alla pipeline tramite `send`, catturato dallo `yield` e conservato nella variabile `widget`. Catturato il widget, viene

abilitato/disabilitato il pulsante OK a prescindere da quale widget si stia gestendo: lo stato abilitato/disabilitato dipende dal contenuto dei widget di inserimento del testo. Dopo la gestione del pulsante, viene controllato se esiste una coroutine come successore: in tal caso, la gestione del widget prosegue nella prossima coroutine della pipeline trasmettendo il widget tramite la funzione `send(widget)`.

Analizziamo ora la coroutine `_clicked_mediator`:

```
@coroutine
def _clicked_mediator(self, successor=None):
    while True:
        widget = (yield)
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
        elif successor is not None:
            successor.send(widget)
```

Tale coroutine si occupa solamente del click sui pulsanti OK e Cancel: l'istruzione `yield` cattura il widget passato tramite la `send` nella precedente coroutine e lo gestisce. Ovviamente, anche in tale coroutine, se esiste una coroutine successore, la gestione del widget viene passata tramite `send(widget)`.

Le classi `Text` e `Button` sono identiche al capitolo precedente, ma la classe `Mediated` ha subito un piccolo cambiamento: nel caso venga richiamato il metodo `on_change`, viene inviato il widget soggetto ad azione/cambiamento alla pipeline di coroutine che fa da mediatore.

```
class Mediated:
    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.send(self)
```

Il Mediator Pattern può anche esser variato in modo da fornire funzionalità di multiplexing, cioè comunicazioni molti a molti tra oggetti: lo State Pattern, difatti, offre tale funzionalità.

Concludendo, ad un'azione effettuata su un oggetto mediato (quindi gestibile da mediatore) corrisponde la gestione di esso da parte del mediatore, il quale non è altro che una catena di coroutine che effettuerà controlli ed invocazioni di funzioni in base a quale oggetto mediato è stato ricevuto.

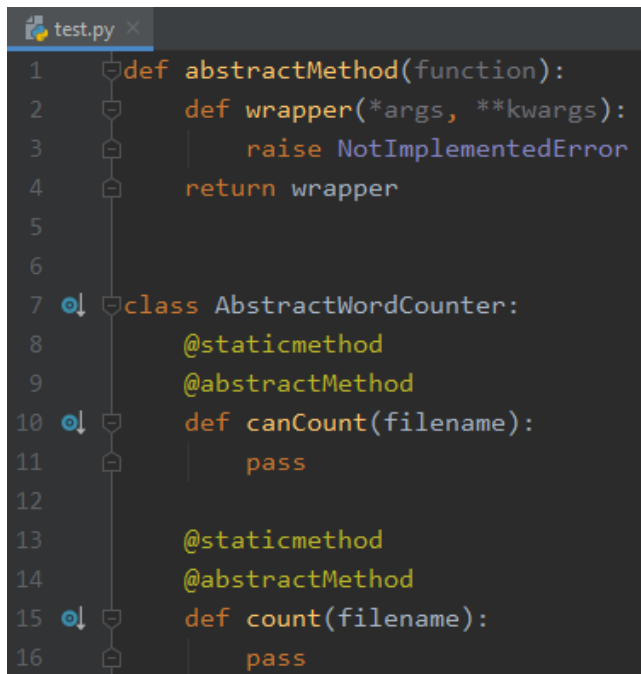
Template Method Pattern

Il Template Method Pattern consente di definire un algoritmo affidandone l'esecuzione ad opportune sottoclassi. Illustreremo tale pattern tramite tre classi:

- Una superclasse `AbstractWordCounter`, la quale fornisce due metodi:

- `canCount(filename)` indica se è possibile contare le parole nel file indicato;
- `count(filename)` conta le parole presenti nel file indicato;
- Due sottoclassi, `TextWordCounter` ed `HtmlWordCounter`, di cui una conterà le parole presenti nei file di testo, mentre l'altra conterà le parole presenti nei file HTML.

Iniziamo con il vedere la superclasse `AbstractWordCounter`:



```

1  def abstractMethod(function):
2      def wrapper(*args, **kwargs):
3          raise NotImplementedError
4      return wrapper
5
6
7  class AbstractWordCounter:
8      @staticmethod
9      @abstractMethod
10     def canCount(filename):
11         pass
12
13     @staticmethod
14     @abstractMethod
15     def count(filename):
16         pass

```

Come già detto, tale classe deve fornire i metodi `canCounter` e `count` ma, essendo una classe astratta, dovrà fornirli senza implementazione. Un semplice `pass` al metodo non basta, altrimenti il richiamo del metodo non svolgerebbe nulla, quindi applichiamo il decoratore `abstractMethod`, il quale permette di lanciare un'eccezione `NotImplementedError` nel caso venga richiamato il metodo astratto.

Quindi, tale classe non fa altro che fornire l'interfaccia per il conteggio delle parole, i cui metodi devono essere reimplementati dalle sottoclassi.

Inoltre, notiamo che ogni metodo definito nella superclasse viene dichiarato come statico: questo perché, visto ciò che si deve fare, non si vede l'utilità di salvare alcuno stato nelle istanze o nella classe. Ovviamente sarebbe stato facile rendere i metodi non statici per poi usare le istanze, se solo avessimo la necessità di istanziare visto ciò che dovranno fare queste classi.

Sostanzialmente, la logica dietro tale classe è: se la classe può effettuare il conteggio di parole sul file indicato, allora si esegue il conteggio e si restituisce il valore; se la classe non può effettuare il conteggio di parole, allora viene restituito 0.

Vediamo ora le sottoclassi che implementano i metodi forniti dalla superclasse astratta appena vista:

```

19 class TextWordCounter(AbstractWordCounter):
20     @staticmethod
21     def canCount(filename):
22         return filename.lower().endswith(".txt")
23
24     @staticmethod
25     def count(filename):
26         if TextWordCounter.canCount(filename):
27             counter = 0
28             with open(filename, "r") as f:
29                 for line in f:
30                     arrayOfWords = line.split()
31                     for word in arrayOfWords:
32                         counter += 1
33             return counter
34         return 0

```

```

37 class HtmlWordCounter(AbstractWordCounter):
38     @staticmethod
39     def canCount(filename):
40         return filename.lower().endswith((".htm", ".html"))
41
42     @staticmethod
43     def count(filename):
44         if HtmlWordCounter.canCount(filename):
45             counter = 0
46             with open(filename, "r") as f:
47                 for line in f:
48                     arrayOfWords = line.split()
49                     for word in arrayOfWords:
50                         counter += 1
51             return counter
52         return 0

```

La classe per il conteggio delle parole nei file HTML è pressoché identica a quella del comune file, quindi il conteggio delle parole sarà banale, includendo anche il conteggio dei tag: ciò è stato fatto per semplificare il tutto, siccome lo scopo di questo capitolo non è creare un contatore di parole nei file HTML, bensì comprendere il Template Method Pattern.

Vediamo ora il programma di testing:

```

myFilename = "testFile.txt"
if TextWordCounter.canCount(myFilename):
    print("There are", TextWordCounter.count(myFilename), "words")
elif HtmlWordCounter.canCount(myFilename):
    print("There are", HtmlWordCounter.count(myFilename), "words")

```

There are 14 words

Il Template Method Pattern, concludendo, è estremamente semplice: data una superclasse astratta che definisce funzioni per la risoluzione di un problema, allora si avranno delle sottoclassi che

implementeranno le funzioni della superclasse, in modo da affidare la risoluzione del problema a quest'ultime. In parole povere, si stabilisce una classe astratta che definisce funzioni generiche e delle sottoclassi che definiranno le precise implementazioni delle funzioni della classe astratta. È, inoltre, importante il fatto che la classe astratta non debba esser utilizzata in esecuzione, siccome l'esecuzione del programma dovrà esser condotta dalle sue implementazioni, quali sono le sottoclassi.

Observer Pattern

L'Observer Pattern supporta relazioni di dipendenza molti a molti tra oggetti, tali che quando un oggetto cambia di stato, tutti gli oggetti in relazione vengono in un certo senso influenzati. Il paradigma con cui possiamo confrontare tale pattern è MVC, in cui abbiamo il modello per i dati (Model), viste per visualizzare i dati (View) e gestori di operazioni e dati (Controller).

In Observer, il modello gestisce i dati, gli osservatori (le view) visualizzano i dati in modo da renderli comprensibili per l'utente e i controller mediano tra input e modello, quindi rendono l'input in comandi per il modello o gli osservatori. In parole povere, le view osservano il modello, mentre il modello è l'oggetto dell'osservatore.

Consideriamo un modello che rappresenta un valore con un minimo e un massimo, come una scrollbar; due osservatori per il modello: uno per l'output del valore del modello ed uno per mantenere la cronologia delle modifiche. Consideriamo, inoltre, una classe Observer per fornire la funzionalità di aggiungere, rimuovere e notificare gli osservatori (le view).

Sostanzialmente, la classe Observer mantiene un insieme di oggetti osservatori ed è caratterizzata da funzioni capaci di gestire gli osservatori:

```
test.py x
1  from datetime import datetime
2  from itertools import chain
3
4
5  class Observer:
6      def __init__(self):
7          self.views = set()
8
9      def addObserver(self, view, *otherViews):
10         for obs in chain(view, *otherViews):
11             self.views.add(obs)
12             obs.update(self)
13
14         def discardObserver(self, view):
15             self.views.discard(view)
16
17         def notifyObserver(self):
18             for obs in self.views:
19                 obs.update(self)
```

La classe Observer è progettata per essere ereditata da modelli che vogliano supportare l'osservazione. Tale classe mantiene un insieme di oggetti di osservazione, gestibili con appropriati metodi. Prima di vedere i metodi con cui si gestiscono gli osservatori, vediamo il modello che eredita Observer e le due view.

```
22 class SliderModel(Observer):
23     def __init__(self, minimum, value, maximum):
24         super().__init__()
25         self.minimum = minimum
26         self._value = value
27         self.maximum = maximum
28
29     @property
30     def value(self):
31         return self._value
32
33     @value.setter
34     def value(self, newValue):
35         if self.minimum < newValue < self.maximum:
36             self._value = newValue
37             self.notifyObserver()
```

In questo esempio utilizziamo uno SliderModel, il quale eredita Observer acquisendo un insieme di osservatori inizialmente vuoto ed i metodi per la gestione degli osservatori, implementati da Observer. Ogni volta che il modello cambia il proprio valore, vengono notificate tutte le viste tramite il metodo notifyObserver, il quale vedremo tra poco.

```
39 class HistoryView:
40     def __init__(self):
41         self.data = []
42
43     def update(self, model):
44         self.data.append((model.value, str(datetime.now())))
```

La vista HistoryView è un osservatore del modello, fornisce un metodo update che verrà richiamato ad ogni cambiamento di un qualsiasi model collegato. Ciò vale anche per LiveView.

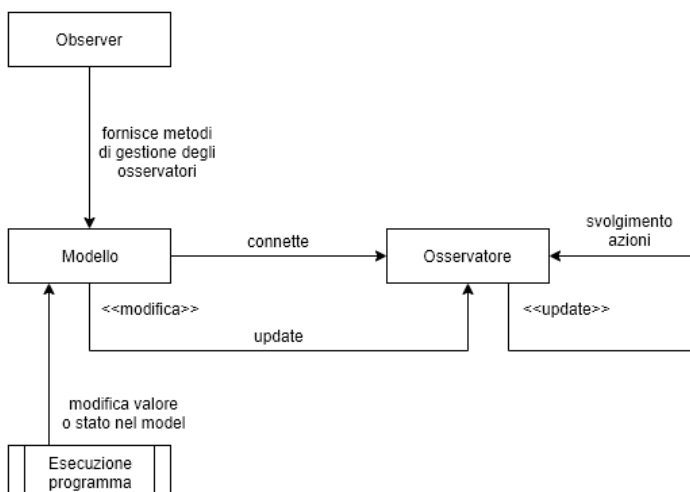
```
47 class LiveView:
48     def __init__(self, length=40):
49         self.length = length
50
51     def update(self, model):
52         tippingPoint = round(model.value * self.length /
53                               (model.maximum - model.minimum))
54         td = '<td style="background-color: {}">&nbsp;</td>'
55         html = ['<table style="font-family: monospace" border="0"><tr>']
56         html.extend(td.format("darkblue") * tippingPoint)
57         html.extend(td.format("cyan") * (self.length - tippingPoint))
58         html.append("<td{}</td></tr></table>".format(model.value))
59         print("".join(html))
```

Avendo compreso, in maniera generale, il funzionamento del modello e degli osservatori, vediamo per bene i metodi della classe Observer:

```
test.py x
1  from datetime import datetime
2  from itertools import chain
3
4
5  class Observer:
6      def __init__(self):
7          self.views = set()
8
9      def addObserver(self, view, *otherViews):
10         for obs in chain(view, *otherViews):
11             self.views.add(obs)
12             obs.update(self)
13
14         def discardObserver(self, view):
15             self.views.discard(view)
16
17         def notifyObserver(self):
18             for obs in self.views:
19                 obs.update(self)
```

Il metodo addObserver permette di aggiungere uno o più osservatori alla lista di osservatori collegati al modello che estende Observer; oltre all'aggiunta, ne segue l'update dell'osservatore aggiunto. Il metodo discardObserver rimuove un determinato osservatore dalla lista degli osservatori collegati al modello che estende Observer. Il metodo notifyObserver effettua la update su tutti gli osservatori collegati al modello che estende Observer; tale metodo viene richiamato ogni volta in cui viene variato un parametro del modello.

Ricapitolando, quindi: il modello estende Observer, la quale classe offre la gestione degli osservatori, permettendo, quindi, al modello di gestire e rimanere collegato agli osservatori; ad ogni modifica o variazione di stato del modello, gli osservatori collegati intervengono effettuando determinate operazioni.



Funzionamento di Observer su diagramma

Facade Pattern

Il Facade Pattern viene utilizzato per rappresentare un'interfaccia semplificata per un sottosistema la cui interfaccia è troppo complessa.

Un esempio è la libreria di Python che fornisce moduli per gestire file compressi: supponiamo di voler utilizzare un'unica interfaccia per poter usufruire delle funzionalità dei moduli. Una soluzione consiste nell'utilizzare il Facade Pattern per fornire una semplicissima interfaccia che demandi la maggior parte del lavoro alla libreria standard.

Vedremo, però, un esempio banale, che faccia comprendere quanto utile sia Facade.

Poniamo di avere due classi che svolgono determinate funzioni:

```
test.py x
1  class MyClass1:
2      def __init__(self, value):
3          self.internal1 = value
4
5      def myFunction1(self):
6          print(self.internal1)
7
8
9  class MyClass2:
10     def __init__(self, value):
11         self.internal2 = value
12
13     def myFunction2(self):
14         print(self.internal2)
15
16     def myFunction3(self):
17         print(self.internal2 * 2)
```

Per qualche motivo risulta complicato eseguire determinate operazioni su tali classi, come l'esecuzione di tutte le funzioni di tali classi. Si stabilisce, quindi, una classe che implementi il Facade Pattern:

```
20  class Facade:
21     def __init__(self, value1, value2):
22         self.item1 = MyClass1(value1)
23         self.item2 = MyClass2(value2)
24
25     def executeAll(self):
26         self.item1.myFunction1()
27         self.item2.myFunction2()
28         self.item2.myFunction3()
29
30     def executeDouble(self):
31         self.item2.myFunction3()
```

Grazie ad una semplice invocazione su questa nuova classe, svolgeremo l'operazione che ci risultava complessa con estrema facilità. Sostanzialmente, quindi, tale pattern tende solamente a ridurre la complessità tra interfacce, in modo da poter eseguire molteplici compiti con pochi richiami di funzione, grazie ad un'interfaccia molto più semplificata rispetto a quella di partenza.

34	<code>myVar = Facade(10, 20)</code>	10
35	<code>myVar.executeAll()</code>	20
36	<code>print("-----")</code>	40
37	<code>myVar.executeDouble()</code>	40

Context Managers

I context managers permettono di allocare e rilasciare risorse in maniera decisamente semplificata. Il classico esempio di context manager riguarda l'utilizzo di un file, il quale viene aperto utilizzando la keyword "with":

```
test.py x
1 with open("testFile.txt", "w") as openedFile:
2     openedFile.write("Hello!")
```

Il blocco di codice sopra mostrato non fa altro che aprire un file, scriverci dentro e chiuderlo. Può sembrare confusionario siccome manca la close: ebbene, le risorse vengono automaticamente rilasciate una volta concluso il blocco with. Difatti, il codice appena visto corrisponde al seguente:

```
test.py x
1 openedFile = open("testFile.txt", "w")
2 openedFile.write("Hello!")
3 openedFile.close()
```

Quindi, il blocco di codice con il with è un context manager, ma non è finita qui siccome è possibile implementare il proprio context manager come una classe.

Una classe context manager deve forzatamente definire i metodi magici `__enter__` ed `__exit__`. La logica dietro tali metodi sta nel fatto che `__enter__` deve poter ritornare l'oggetto allocato, mentre `__exit__` deve poter liberare la memoria riservata all'oggetto allocato. L'allocazione dell'oggetto e la sua inizializzazione avvengono nell'init.

```
test.py x
1 class File:
2     def __init__(self, filename, method):
3         self.obj_file = open(filename, method)
4
5     def __enter__(self):
6         return self.obj_file
7
8     def __exit__(self, exc_type, exc_val, exc_tb):
9         self.obj_file.close()
10
11
12 with File("testFile.txt", "r") as myFile:
13     print(myFile.readline())
```

Hello! That script works!

Definiti i due metodi `__enter__` ed `__exit__`, è possibile testare il nostro context manager utilizzando la keyword `with`.

Vediamo cosa succede precisamente con il costrutto `with`:

1. Viene eseguita l'istruzione `"with File(...)"`, quindi viene aperto un blocco `with` e viene eseguita la `init` della classe `File`, la quale, nell'esempio sopra visto, apre un file e salva il riferimento in una variabile di istanza;
2. Viene eseguita l'istruzione `"as myFile"`, quindi viene eseguito il metodo `__enter__`, il quale deve poter restituire la variabile di istanza a cui farà riferimento `myFile`, in questo caso `self.obj_file`. Sostanzialmente, quindi, quando tratteremo `myFile` è come se stessimo trattando la variabile di istanza `self.obj_file`;
3. Vengono eseguite le istruzioni interne al blocco `with`;
4. Concluso il blocco `with`, viene eseguito il metodo `__exit__`, il quale deve liberare la memoria allocata nell'`init`. Sostanzialmente, in questo esempio, non fa altro che eseguire una `close`.

Ricapitolando, quindi:

1. Viene aperto il blocco `with` e viene eseguito l'`init` della classe specificata;
2. Viene salvato l'oggetto allocato in una variabile del blocco `with`, la quale può essere utilizzata per gestire l'oggetto allocato;
3. Vengono eseguite le istruzioni interne al blocco `with`;
4. Concluso il blocco `with`, viene liberata la memoria riservata all'oggetto allocato.

Nel caso lo abbiate notato, stiamo tralasciando gli argomenti del metodo `__exit__`: questo perché non servono riguardo la gestione dell'allocazione e la liberazione della memoria, bensì servono a gestire eventuali eccezioni. Se si manifesta un'eccezione, Python passa il tipo, il valore ed il `traceback` dell'eccezione come parametri del metodo `__exit__`, in modo tale che quest'ultimo metodo possa decidere come chiudere comunque il file e se sarà necessario qualche altro eventuale passaggio.

Nel caso, quindi, si presenti qualche eccezione durante la gestione dell'oggetto allocato nel blocco `with`: se l'eccezione viene gestita con successo dal metodo `__exit__`, allora viene ritornato `True`; se l'eccezione non viene gestita con successo dal metodo `__exit__`, l'eccezione viene passata e lanciata dal `with` statement. La gestione dell'eccezione è davvero semplice, basta apportare una semplice modifica: basta controllare che uno degli argomenti del metodo `__exit__` non sia `None`. Nel caso uno degli argomenti di `__exit__` sia `None`, allora è possibile specificare eventuali istruzioni da eseguire nel caso l'eccezione di presenti per poi ritornare `True`; se uno degli argomenti di `__exit__` non è `None`, allora basta eseguire la semplice liberazione della memoria. Ovviamente viene ritornato `True` nel caso si voglia gestire l'eccezione: nel caso si voglia lasciar gestire l'eccezione al `with` statement, basta tornare qualsiasi cosa che non sia `True`. Sì, era stato già detto prima, ma ciò ci fa capire che siamo noi a decidere il valore di ritorno!

Nella prossima pagina vediamo un esempio riguardante la corretta gestione dell'eccezione ed un esempio riguardante il passaggio della gestione dell'eccezione al `with` statement.

```

1 class File:
2     def __init__(self, filename, method):
3         self.obj_file = open(filename, method)
4
5     def __enter__(self):
6         return self.obj_file
7
8     def __exit__(self, exc_type, exc_val, exc_tb):
9         if exc_type is not None:
10            print("Exception handled!")
11            self.obj_file.close()
12            return True
13        else:
14            self.obj_file.close()
15
16
17 with File("testFile.txt", "r") as myFile:
18     print(myFile.invalidMethod())

```

Exception handled!

```

1 class File:
2     def __init__(self, filename, method):
3         self.obj_file = open(filename, method)
4
5     def __enter__(self):
6         return self.obj_file
7
8     def __exit__(self, exc_type, exc_val, exc_tb):
9         if exc_type is not None:
10            print("Exception not handled!")
11            self.obj_file.close()
12            return False
13        else:
14            self.obj_file.close()
15
16
17 with File("testFile.txt", "r") as myFile:
18     print(myFile.invalidMethod())

```

```

Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 18, in <module>
    print(myFile.invalidMethod())
Exception not handled!
AttributeError: '_io.TextIOWrapper' object has no attribute 'invalidMethod'

```

È possibile fare qualsiasi controllo nel metodo exit ed è possibile personalizzarlo a proprio piacimento. Un esempio un pochino più avanzato consiste nel controllare addirittura il tipo di eccezione presentata, in modo da decidere cosa fare in quel caso:

```
test.py x
1 class File:
2     def __init__(self, filename, method):
3         self.obj_file = open(filename, method)
4
5     def __enter__(self):
6         return self.obj_file
7
8     def __exit__(self, exc_type, exc_val, exc_tb):
9         if exc_type is AttributeError:
10            print("Exception handled! It was an {}".format(exc_type))
11            print("Correcting execution flux...")
12            print(myFile.readline())
13            print("Done. Closing file...")
14            self.obj_file.close()
15            print("File closed.")
16            return True
17        elif exc_type is not AttributeError:
18            print("I can't handle this exception! Fatal error!")
19            self.obj_file.close()
20            return False
21        else:
22            self.obj_file.close()
23
24
25 with File("testFile.txt", "r") as myFile:
26     print(myFile.invalidMethod())
```

Exception handled! It was an <class 'AttributeError'>
Correcting execution flux...
Hello! That script works!
Done. Closing file...
File closed.

Flyweight Pattern

Il Flyweight Pattern è progettato per la gestione di molteplici oggetti relativamente piccoli di cui molti sono duplicati di altri. Tale pattern è implementato rappresentando ogni oggetto univoco una sola volta tramite un dizionario, in cui ogni oggetto univoco è registrato come un valore identificato da una chiave univoca.

Tale pattern è utile in particolare quando si deve ricreare più volte la stessa cosa (come con le pagine web, quando bisogna creare più pagine con lo stesso font), ma risulta più efficiente riutilizzare la singola risorsa (nel caso delle pagine web non si ricrea più volte il font, bensì lo si ricava da un appropriato file, in modo tale che ogni font venga creato una sola volta, indipendentemente dal numero di volta in cui è usato).

In alcune situazioni potremmo avere un gran numero di oggetti non necessariamente piccoli, di cui la maggior parte sono diversi dagli altri. Un modo facile per ridurre lo spazio occupato in memoria è quello di utilizzare `__slots__` (vedi approfondimento, se necessario).

Il seguente esempio mostra una semplice classe `Point` che conserva le variabili per la raffigurazione tridimensionale senza l'utilizzo del dict, in modo da risparmiare spazio in RAM. In tal modo, quindi, nessun `Point` avrà il proprio dict privato. Tuttavia, quindi, ciò a significare che non è possibile aggiungere attributi ai singoli punti, siccome la memoria sarà stata allocata staticamente.

```
test.py x
1 class Point:
2     __slots__ = ("x", "y", "z", "color")
3
4     def __init__(self, x, y, z, color=None):
5         self.x = x
6         self.y = y
7         self.z = z
8         self.color = color
```

Sostanzialmente, quindi, tale pattern serve a migliorare le prestazioni del programma, o meglio la memoria RAM richiesta per l'allocazione di determinati oggetti.

Prototype Pattern

Il Prototype Pattern è utilizzato per creare nuovi oggetti creando un oggetto originale e poi modificando il clone. In parole povere, si tratta di creare un oggetto, clonarlo e modificare quest'ultimo.

Python offre molteplici modi per clonare un oggetto, ma il Prototype Pattern predilige l'uso della `deepcopy`:

```
test.py x
1 import copy
2
3
4 class Point:
5     __slots__ = ("x", "y")
6
7     def __init__(self, x, y):
8         self.x = x
9         self.y = y
10
11
12 original = Point(1, 2)
13 prototype = copy.deepcopy(original)
14 prototype.x = 6
15 prototype.y = 12
16 print("x = {}, y = {}".format(prototype.x, prototype.y))
```

Concorrenza

Finora abbiamo visto script capaci di eseguire un determinato insieme di istruzioni in sequenza, quindi di eseguire un'operazione alla volta con lo svantaggio di avere un rallentamento del programma nel caso un'operazione sia particolarmente pesante e lenta, siccome il resto del programma dovrà comunque attendere la sua esecuzione. Una soluzione a tale problema può essere il rendere ogni operazione una coroutine, in modo da rendere indipendente l'esecuzione di ogni operazione: nel caso una coroutine sia particolarmente lenta, non intaccherà le altre, le quali continueranno a lavorare minimizzando gli idle time. Come fa, però, la coroutine ad essere concorrente? Ebbene, ogni coroutine ha un ciclo while infinito che non viene eseguito sequenzialmente con il normale flusso di istruzioni. È pur sempre una tecnica ostile, soprattutto da vedere nel caso il codice sia tanto, quindi non è la miglior scelta da seguire quando si parla di concorrenza.

Nonostante non sia conveniente utilizzare le coroutine, ci siamo comunque fatti un'idea di cosa significhi eseguire operazioni in concorrenza. In Python, la concorrenza può essere implementata in due modi: la principale distinzione è data dall'accesso diretto (ad esempio attraverso memoria condivisa) o indiretto ai dati (ad esempio utilizzando la comunicazione tra processi).

La concorrenza basata sui thread, detta anche multithreading, si ha quando diversi thread di esecuzione operano all'interno dello stesso processo di sistema, accedendo a dati condivisi tramite un accesso alla memoria condivisa. La concorrenza basata sui processi, detta anche multiprocessing, si ha quando più processi distinti tra loro vengono eseguiti in modo indipendente. I processi concorrenti tipicamente condividono i dati tra loro mediante IPC, anche se possono comunque utilizzare la memoria condivisa nel caso il linguaggio o la libreria lo permettano. Python supporta entrambi i tipi di concorrenza appena visti, ma è bene sapere che i thread vengono utilizzati solo per convenzione, mentre i processi vengono usati perché sono di alto livello.

È importante sapere che a causa del GIL (Global Interpreter Lock) l'interprete Python può essere in esecuzione solamente su uno dei core del processore, negando la produzione di notevoli incrementi prestazionali nel caso si usino i thread: questo perché si utilizzano molteplici thread in un solo processo, quindi il GIL rimane in esecuzione su un solo core e non si ha l'incremento prestazionale. Se l'elaborazione è CPU-bound, una soluzione può essere quella di utilizzare Cython, che è essenzialmente Python con costrutti aggiuntivi compilati in C, portando le performance fino a 100 volte migliori. Cython tende ad aggirare il GIL in qualche modo, ma è meglio evitarlo del tutto scegliendo di utilizzare il modulo multiprocessing: tale modulo non utilizza i thread, bensì usa processi separati, ognuno dei quali utilizza la propria istanza indipendente dell'interprete Python, senza creare conflitti.

Con Python è sempre bene evitare, se possibile, scrivere programmi concorrenti. Difatti, è altamente consigliato scrivere sempre un programma non concorrente dato che è più semplice da scrivere e testare, anziché partire direttamente programmando in concorrente; una volta creato il programma non concorrente, può essere che risulti sufficientemente veloce così com'è. In caso contrario, può essere utilizzato comunque per un confronto con la versione concorrente, in modo da misurare il guadagno in prestazioni e capire se ne è valsa la pena o meno.

In generale, comunque, si consiglia il multiprocessing per programmi CPU-bound, mentre si consiglia il multithread per programmi I/O-bound. In ogni caso, non conta solamente il tipo di concorrenza, bensì anche il livello:

- Concorrenza a basso livello: concorrenza che utilizza operazioni atomiche, le quali sono azioni non interrompibili che si completano del tutto oppure per nulla. Questo tipo di concorrenza è adatto a chi scrive librerie e non a chi scrive applicazioni;
- Concorrenza a livello intermedio: concorrenza che non utilizza operazioni atomiche, bensì utilizza lock espliciti. Questo è il livello di concorrenza supportato dalla maggior parte dei linguaggi di programmazione e viene generalmente utilizzato per lo sviluppo di applicazioni. Questo livello di concorrenza offre classi come `threading.Semaphore`, `threading.Lock` e `multiprocessing.Lock` per la gestione della concorrenza;
- Concorrenza ad alto livello: è un livello di concorrenza che non prevede né operazioni atomiche e né lock espliciti.

Gli approcci di livello intermedio sono semplici da utilizzare ma decisamente soggetti ad alti rischi di errori logici, come deadlock e problematiche varie riguardanti la concorrenza, i quali non discuteremo.

Piuttosto, il problema fondamentale riguarda la condivisione dei dati. I dati condivisi mutabili devono essere protetti mediante lock per garantire che tutti gli accessi siano sincroni. Inoltre, quando più processi o thread tentano di accedere agli stessi dati condivisi, tutti tranne uno vengono bloccati e rimangono inattivi, in attesa che quella risorsa si sblocchi: ciò significa che quando è attivo un lock, l'applicazione permette l'accesso ai dati come se fosse sequenziale, quindi lasciando che un solo thread o un solo processo possa accederci. Il problema, quindi, è presentato dai lock, i quali possono bloccare il programma e renderlo decisamente lento. A questo punto rimangono due scelte: o i lock vengono utilizzati il meno possibili e per tempi decisamente brevi, o non si condividono dati mutabili, in modo da evitare totalmente l'utilizzo dei lock. In quest'ultimo caso, una soluzione alternativa ai lock consiste nell'utilizzare una struttura dati che preveda l'accesso concorrente, come il modulo `queue` che offre code thread-safe, mentre per il multiprocessing ci sono classi apposite come `multiprocessing.JoinableQueue` e `multiprocessing.Queue`. Tali code, quindi, forniscono sia una singola sorgente di job per tutti i thread e tutti i processi, sia un'unica destinazione dei risultati.

Pacchetto multiprocessing

Un oggetto `multiprocessing.Process` rappresenta un'attività svolta in un processo separato ed è caratterizzato da molteplici metodi, alcuni dei quali sono:

- `run()` – metodo che rappresenta l'attività del processo;
- `start()` – metodo che dà inizio all'attività del processo, deve essere invocato al più una volta per un oggetto processo e fa in modo che il metodo `run()` dell'oggetto venga invocato in un processo separato;
- `join()` – metodo che permette allo scope dove si invoca di attendere la fine dell'esecuzione del processo su cui è stato invocato il metodo.

Il costruttore dell'oggetto `multiprocessing.Process` è il seguente:

`multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={ }, daemon=None)`

Notiamo che ogni argomento del costruttore è una keyword, la quale analizziamo:

- `group` deve essere sempre `None` in quanto è presente solo per ragioni di compatibilità con `threading.Thread` di cui il `multiprocessing.Process` condivide l'interfaccia;
- `target` è l'oggetto callable invocato da `run()`: se rimane `None`, non verrà invocato alcun metodo;
- `name` è il nome del processo;
- `args` è la tupla di argomenti da passare a `target`, cioè l'oggetto callable;
- `kwargs` è un dizionario di argomenti keyword da passare a `target`, cioè l'oggetto callable;
- `daemon` è un booleano che indica se il processo deve essere creato come regolare processo oppure come processo daemon. Se il valore di `daemon` è `None`, allora viene ereditato dal processo invocante.

In base alla piattaforma, `multiprocessing` supporta tre modalità per dare inizio ad un processo. Noi vedremo solamente due di questi metodi:

- `spawn()` – il processo padre lancia un nuovo processo per eseguire l'interprete Python. Il processo figlio eredita solo le risorse necessarie per eseguire il metodo `run()`. Questo modo di iniziare i processi è molto lento se confrontato a `fork`, ma è disponibile sia per Unix che per Windows;
- `fork()` – il processo padre utilizza la `fork` per fare il fork dell'interprete Python. Il processo figlio sarà praticamente uguale al processo padre, quindi ne eredita risorse e caratteristiche. Tale metodo è disponibile solo su Unix, dove rappresenta il metodo di default per iniziare i processi.

Come già anticipato nel capitolo precedente, è possibile che dati o processi vogliano accedere agli stessi dati: è possibile usufruire di due classi, quali sono `Queue` e `JoinableQueue`, le quali saranno l'unica fonte di task/job per tutti i thread o processi e un'unica destinazione per i risultati.

Tali code sono, appunto, condivise dai processi, quindi qualsiasi oggetto conservato è accessibile da qualsiasi processo. Vediamo solo alcuni dei metodi di `multiprocessing.Queue`, i quali restituiscono un output non affidabile per via della semantica del `multithreading/multiprocessing`:

- `qsize()` – restituisce la dimensione approssimata della coda;
- `empty()` – restituisce `True` se la coda è vuota, `False` altrimenti;
- `full()` – restituisce `True` se la coda è piena, `False` altrimenti;
- `put(obj, block, timeout)` – inserisce `obj` nella coda. Se l'argomento opzionale `block` è `True` e `timeout` è `None`, si blocca finché non si rende disponibile uno slot della coda (accade ovviamente quando la coda è piena). Se, invece, `timeout` è un numero positivo, si blocca per quanti secondi sono stati specificati con `timeout`, per poi lanciare l'eccezione `queue.Full` nel caso non si renda disponibile alcuno slot della coda entro quel lasso di tempo. Se `block` è `False`, l'oggetto viene inserito solo se è immediatamente libero uno slot della coda, altrimenti viene lanciata l'eccezione `queue.Full`. Ricapitolando, quindi:

- Block = True -> attende che la coda liberi uno slot per inserire l'oggetto al massimo per timeout secondi, se specificato. Se lo slot non si libera entro timeout secondi, viene lanciata un'eccezione queue.Full;
- Block = False -> tenta immediatamente l'inserimento: se la coda è piena lancia un'eccezione queue.Full.
- put_nowait(obj) – equivalente a put(obj, False);
- get(block, timeout) – rimuove e restituisce un oggetto dalla coda. Se l'argomento opzionale block è True e timeout è None, si blocca fin quando un elemento non sarà disponibile, quindi finché non ci sarà un elemento da ritornare. Se, invece, timeout è un numero positivo, allora il metodo si blocca e attende un elemento disponibile fin quando non scadrà il tempo impostato come timeout. Se block è False, viene restituito un elemento solo se immediatamente disponibile, altrimenti viene lanciata l'eccezione queue.Empty. Ricapitolando, quindi:
 - Block = True -> attende che ci sia un elemento nella coda per restituirlo al massimo per timeout secondi, se specificato. Se non sarà presente un elemento da restituire entro timeout secondi, viene lanciata un'eccezione queue.Empty;
 - Block = False -> tenta immediatamente la restituzione dell'elemento: se la coda è vuota lancia un'eccezione queue.Empty.
- get_nowait() – equivalente a get(False);

La classe multiprocessing.JoinableQueue è una sottoclasse di Queue che ha in aggiunta i metodi task_done() e join():

- task_done() – indica che un task precedentemente inserito in coda è stato completato. Per ciascuna get() utilizzata per prelevare un task, deve essere effettuata una chiamata a task_done() per informare la coda riguardo il completamento del task. Un join() bloccato si sblocca quando tutti i task sono stati completati e dopo che è stata ricevuta una chiamata a task_done() per ogni task precedentemente inserito in coda. Si ha un ValueError se task_done() è invocato un numero di volte maggiore degli elementi in coda;
- join() – causa un blocco sullo scope nel quale viene invocato tale metodo fin quando gli elementi della coda non sono stati tutti prelevati e processati. Il conteggio dei task incompleti incrementa ogni volta in cui viene aggiunto un elemento alla coda e viene decrementato ogni volta che viene invocato task_done(). Quando il conteggio dei task incompleti va a zero, join() si sblocca.

Sostanzialmente, quindi, la JoinableQueue si comporta in base ai task che contiene: se contiene task non processati allora rende bloccato lo scope, altrimenti lo sblocca.

Concorrenza CPU-bound

Supponiamo di voler effettuare un'operazione decisamente pesante CPU-bound, come il ridimensionamento di immagini. Essendo il ridimensionamento un'operazione CPU-bound, ci si aspetta performance migliori dal multiprocessing. Nella prossima pagina riportiamo una tabella con i dati riferiti al ridimensionamento di 56 immagini per una dimensione complessiva di 331MB, dove si mostra il tipo di concorrenza adoperata, i secondi utilizzati per il completamento dell'operazione ed il guadagno in prestazioni.

Concorrenza	Secondi	Incremento velocità
Nessuna	784	Riferimento
4 coroutine	781	1,00×
4 thread con un pool di thread	1339	0,59×
4 processi con una coda	206	3,81×
4 processi con un pool di processi	201	3,90×

I risultati dimostrano quanto l'utilizzo dei processi sia efficiente su operazioni CPU-bound. C'è, però, da fare una nota: l'avvio di nuovi processi è molto più dispendioso su Windows che sulla maggior parte degli altri OS: fortunatamente, le code ed i pool di Python utilizzano pool di processi persistenti, in modo da evitare ripetutamente fastidiosi costi di avvio dei processi.

È bene specificare, inoltre, che con i programmi CPU-bound si tende ad utilizzare tanti processi quanti sono i core della macchina; con i programmi I/O-bound, invece, si tende ad utilizzare un multiplo del numero di core (2x, 3x, 4x, ...) a seconda della larghezza della banda di rete.

Prendendo proprio l'esempio del rescaling dell'immagine, vediamo la funzione `scale(...)` e commentiamola:

```
def scale(size, smooth, source, target, concurrency):
    canceled = False
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(size, smooth, jobs, results, concurrency)
    todo = add_jobs(source, target, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt: # Potrebbe non funzionare su Windows
        Qttrac.report("canceling...")
        canceled = True
    copied = scaled = 0
    while not results.empty(): # Sicuro perché tutti i job sono terminati
        result = results.get_nowait()
        copied += result.copied
        scaled += result.scaled
    return Summary(todo, copied, scaled, canceled)
```

La funzione inizia creando una coda `JoinableQueue` composta dai jobs da eseguire. Una coda `JoinableQueue` è una coda per la quale è possibile attendere finché essa non sarà vuota. Successivamente, viene creata una normale `Queue`, quindi non joinable, nella quale verranno conservati i risultati. Fatto ciò, la funzione crea i processi per eseguire il lavoro: saranno pronti per essere utilizzati ma bloccati, siccome la coda dei jobs è vuota e non sapranno cosa eseguire. Si prosegue, quindi, aggiungendo dei jobs alla `JoinableQueue` per poi effettuare una `join()`, la quale bloccherà lo scope attuale finché tutti i processi non avranno concluso i jobs della `JoinableQueue`. Il restante codice non lo vedremo perché tratta i risultati e poco ci interessa.

Piuttosto, sarebbe interessante vedere la funzione `create_processes()`, la quale si occupa di creare i processi:

```
def create_processes(size, smooth, jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(size,
            smooth, jobs, results))
        process.daemon = True
        process.start()
```

Tale funzione si occupa della creazione dei processi per eseguire il lavoro. A ciascun processo viene data come parametro la funzione worker, la quale si occupa del reale ridimensionamento dell'immagine. Inoltre, come possiamo ben vedere, viene passata anche la coda dei jobs e la coda dei risultati. Ad ogni processo creato, viene settato a True l'attributo daemon: quando il processo principale termina, termina in modo corretto anche tutti i propri processi daemon. La differenza dai normali processi è che con la terminazione del processo principale può capitare che continui a lavorare qualche altro processo, il quale diventerà un processo zombie siccome rimarrà sempre in esecuzione.

Ci interessa ora sapere come viene gestita la coda dei jobs e dei results, anche perché, magari, non si è ancora compreso cosa comprenda la coda dei jobs. Vediamo, quindi, la funzione worker:

```
def worker(size, smooth, jobs, results):
    while True:
        try:
            sourceImage, targetImage = jobs.get()
            try:
                result = scale_one(size, smooth, sourceImage, targetImage)
                Qtrac.report("{} {}".format("copied" if result.copied else
                    "scaled", os.path.basename(result.name)))
                results.put(result)
            except Image.Error as err: Qtrac.report(str(err), True)
        finally:
            jobs.task_done()
```

Il worker esegue un ciclo infinito e in ciascuna iterazione tenta di ottenere un job da eseguire dalla coda dei jobs condivisa. Essendo il processo un tipo daemon, è sicuro utilizzare cicli infiniti siccome termineranno insieme alla terminazione del processo principale. Ricevuto un job, il quale non è altro che l'immagine su cui operare, si inserisce il risultato nella coda dei risultati e si comunica alla coda di jobs che il task è stato concluso. Da come viene trattato, comunque, notiamo che un job non è altro che un qualsiasi oggetto su cui può operare un processo: in tal caso, non è altro che una coppia di elementi.

Come viene gestito tutto ciò, ora, è finalmente chiaro: i processi si istanziano tramite la classe Process (è possibile crearne anche sottoclassi), si passa la funzione da eseguire ed eventuali argomenti, comprendenti magari una coda di jobs da trattare e condivisa da tutti i processi istanziati.

PS: il discorso fatto finora vale anche per i thread, i quali trattiamo solamente nell'approfondimento siccome fare un altro capitolo su questo discorso è fondamentalmente inutile. I threads vanno usati per programmi I/O-bound, ottenendo ottimi miglioramenti prestazionali. Non tratteremo, appunto, l'I/O-bound perché la gestione dei thread è praticamente uguale a quella dei processi, con alcuni cambiamenti sulla compatibilità sui sistemi operativi.

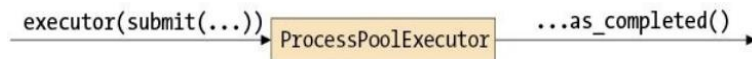
Utilizzare i future e il multiprocessing

Col tempo è stato introdotto il modulo concurrent.futures, il quale costituisce un mezzo di alto livello per realizzare concorrenza utilizzando più thread e più processi. I future sono istanze della classe concurrent.futures.Future ed eseguono i callables in maniera asincrona; si creano richiamando il metodo concurrent.futures.Executor.submit() e possono segnalare il loro stato (interrotto, in esecuzione, completato) e il risultato o eccezione prodotta. La classe Executor non

può essere utilizzata siccome è una classe astratta, mentre possiamo utilizzare una delle sue due sottoclassi:

- `concurrent.futures.ProcessPoolExecutor()` produce una concorrenza basata sull'utilizzo di più processi. L'uso di un pool significa che ogni Future utilizzato con esso può eseguire e restituire solamente oggetti pickleable.
- `concurrent.futures.ThreadPoolExecutor()`, invece, non ha la restrizione di `ProcessPoolExecutor` e realizza la concorrenza utilizzando i thread.

L'utilizzo di un pool è molto più semplice che utilizzare le code.



Vediamo l'utilizzo del pool nel seguente esempio:

```
def scale(size, smooth, source, target, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
        max_workers=concurrency) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, smooth, sourceImage,
                                    targetImage)
            futures.add(future)
    summary = wait_for(futures)
    if summary.canceled:
        executor.shutdown()
    return summary
```

Si inizia creando un set di futures, per poi creare un pool di processi: dietro le quinte, quest'ultimo creerà una serie di processi worker. Il loro numero viene determinato in modo non rigoroso, ma può comunque esser direttamente specificato proprio come in questo caso. Ora che si ha un executor per il pool di processi, si itera sulle operazioni restituite dalla funzione `get_jobs()`: il metodo `submit()` accetta un worker e argomenti opzionali per poi restituire un Future che verrà aggiunto al set di futures stabilito all'inizio. Sostanzialmente, la `submit(func, *args, **kwargs)` non farà altro che eseguire `func(*args, **kwargs)` e restituisce un oggetto Future che rappresenta l'esecuzione del callable. Il pool inizierà a lavorare non appena avrà un future su cui operare. Creati tutti i futures ed aggiunti al set precedentemente creato, si attende la loro esecuzione e terminazione tramite la funzione `wait_for()`, la quale rimane bloccata finché non verranno ultimati tutti i futures. La funzione `wait_for()` è realizzata tramite la funzione `concurrent.futures.as_completed(future)`, la quale attua un bloccaggio fino a quando un future non termina o viene annullato, per poi restituire il future stesso.

Se il worker callable ha generato un'eccezione, essa viene restituita tramite il metodo `Future.exception()`, altrimenti il metodo restituisce `None`.

PS: Come nel capitolo precedente, abbiamo trattato solo i processi e non i thread per il semplice fatto che vengono gestiti nello stesso ed identico modo. In ogni caso, è consigliabile dare un'occhiata al capitolo riguardante l'approfondimento sul pooling dei threads.

Conclusione programma base

Introduzione approfondimenti

Approfondimento: dizionari e `__dict__`

In questo capitolo approfondiremo i dizionari e le loro funzioni e la variabile built-in `__dict__`.

I dizionari sono ciò che in altri linguaggi vengono chiamati *hashmap* o *associative arrays*, permettono di lavorare con coppie (chiave, valore), dove la chiave è un identificativo univoco e il valore è il dato associato a quella determinata chiave.

I dizionari sono chiamati così proprio perché sono paragonabili ai dizionari della vita reale: si cerca una parola (chiave) e si trova la definizione associata (valore). Vediamo un esempio:

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4 print(student)
5 print(student["name"])
6 print(student["courses"])
7 print(student["courses"][1])

{'name': 'John', 'age': 20, 'courses': ['math', 'programming']}
John
['math', 'programming']
programming
```

Definiamo un oggetto `student` tramite la definizione di `dict` con le brackets `{ ... }` in cui specifichiamo tre chiavi: `name`, `age` e `courses`, le quali vengono specificate come stringhe. Ogni chiave specificata ha un valore, il quale può essere un qualsiasi oggetto: alla chiave `name` viene associata una stringa; alla chiave `age` viene associato un intero; alla chiave `courses` viene associata una lista. Andando a stampare valori, notiamo come sia possibile accedere a determinati valori del dizionario, cioè specificando tra le brackets `{ ... }` la chiave da controllare.

Ovviamente, come i valori assunti dalle chiavi, anche le chiavi stesse possono assumere un qualsiasi tipo:

```
test.py x
1 student = {1: "John",
2           2: 20,
3           3: ["math", "programming"]}
4 print(student)
5 print(student[1])
6 print(student[2])
7 print(student[3][1])
```

Altro modo per accedere ai valori del dizionario è tramite il metodo `get(key)`, il quale offre un paio di cambiamenti rispetto al classico modo di ottenere valori: nel caso la chiave non esista, con `get` viene ritornato `None`, mentre con il classico modo viene lanciata un'eccezione; è possibile specificare un valore di default nel caso la chiave non esista, in modo da ritornare il valore di default anziché `None`, mentre con il classico modo ciò non è possibile.

È possibile aggiungere nuove coppie (chiave, valore) al dizionario con la seguente sintassi:

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4
5 student["phone"] = "111 222 3333"
6 print(student["phone"])
```

Nota: il warning è dovuto dal fatto che l'aggiunta della nuova coppia (chiave, valore) può esser fatta direttamente nel costruttore del dizionario.

Altro modo di aggiungere coppie (chiave, valore) è tramite il metodo update della classe dict, il quale vuole come argomento un nuovo dizionario, nel quale inseriremo le coppie da inserire o sovrascrivere:

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4 student.update({"name": "Jane", "phone": "111 222 3333"})
5 print(student["name"], student["phone"]) Jane 111 222 3333
```

Per rimuovere una coppia (chiave, valore), invece, basta utilizzare l'istruzione del, la quale oltre ad eliminare la coppia ritorna il valore di essa, il quale può essere salvato in una variabile.

Illustriamo ora due metodi per tornare tutte le chiavi e tutti i valori presenti nel dizionario: keys() e values():

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4 print(student.keys())
5 print(student.values())
dict_keys(['name', 'age', 'courses'])
dict_values(['John', 20, ['math', 'programming']])
```

È inoltre possibile stampare tutte le coppie tramite il metodo items().

Nel caso si voglia scansionare il dizionario tramite un ciclo for, è bene sapere che nel caso si usi la seguente sintassi verranno ottenute solo le chiavi appartenenti ad esso:

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4 for item in student:
5     print(item)
name
age
courses
```

Nel caso, invece, si vogliano ottenere i valori delle chiavi, è necessario scansionare la lista di valori, quindi la lista ricavata tramite il metodo `values()`. Se si vogliono ottenere le coppie (chiave, valore), invece, è possibile ciclare con un `for` con doppio attributo:

```
test.py x
1 student = {"name": "John",
2           "age": 20,
3           "courses": ["math", "programming"]}
4 for key, value in student.items():
5     print(key, value)
```

```
name John
age 20
courses ['math', 'programming']
```

Avendo compreso, quindi, il funzionamento dei dizionari, possiamo finalmente approdare su una variabile built-in, posseduta da qualsiasi istanza: si parla della variabile `__dict__`, la quale conserva un dizionario contenente una serie di coppie (chiave, valore) dove la chiave è un attributo dell'istanza, mentre il valore è ciò che contiene quel determinato attributo.

```
test.py x
1 class Student:
2
3     def greetings(self):
4         print("Hello! I'm", self.name, "and i'm", self.age, "years old.")
5
6     def __init__(self, name, age):
7         self.name = name
8         self.age = age
9         self.funct = self.greetings
10
11
12 myStudent = Student("Francesco", 21)
13 print(myStudent.__dict__)
```

```
{'name': 'Francesco', 'age': 21, 'funct': <bound method Student.greetings of <__main__.Student object at 0x013E0E50>>}
```

Ovviamente su tale dizionario è possibile operare come su qualsiasi altro dizionario, quindi sarà possibile aggiungere nuovi attributi dinamicamente, modificarli e rimuoverli.

Vediamo nella prossima pagina un esempio di inserimento, di modifica e di rimozione.

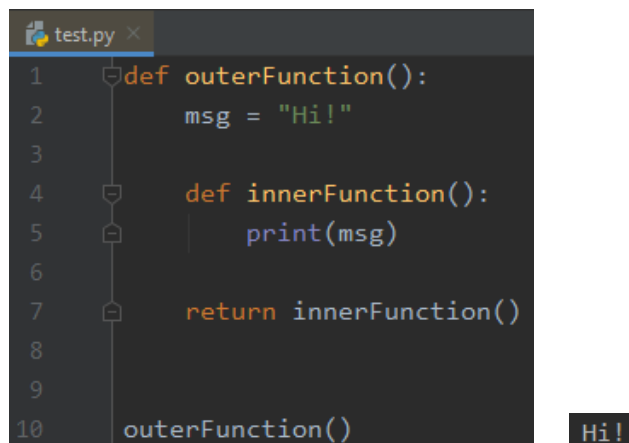
Come ben vedremo, è possibile aggiungere, modificare e rimuovere dinamicamente gli attributi dell'istanza su cui si sta lavorando.

```
test.py x
1 class Student:
2
3     def greetings(self):
4         print("Hello! I'm", self.name, "and i'm", self.age, "years old.")
5
6     def __init__(self, name, age):
7         self.name = name
8         self.age = age
9         self.funct = self.greetings
10
11
12 myStudent = Student("Francesco", 21)
13 myStudent.__dict__["phone"] = "111 222 3333"
14 myStudent.__dict__.update({"name": "Jane", "nation": "Italy"})
15 del myStudent.__dict__["age"]
16 for key, value in myStudent.__dict__.items():
17     print("Key: {} | Value: {}".format(key, value))
```


Approfondimento: decoratori e funzioni

In questo capitolo capiremo come funzionano i decoratori in relazioni a funzioni e classi, in modo da comprendere a fondo il loro funzionamento e la loro gestione. Verranno mostrati molteplici esempi, siccome la pratica in tale capitolo sarà fondamentale.

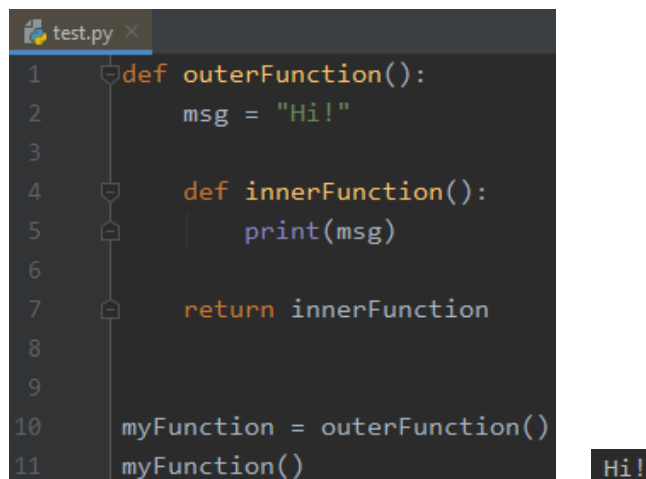
Partiamo dal seguente esempio:

A screenshot of a code editor window titled 'test.py'. The code defines an 'outerFunction()' which sets a variable 'msg' to 'Hi!' and contains an inner function 'innerFunction()' that prints 'msg'. The 'outerFunction()' returns the 'innerFunction()'. At the bottom, 'outerFunction()' is called, and the output 'Hi!' is shown in a terminal window.

```
1 def outerFunction():
2     msg = "Hi!"
3
4     def innerFunction():
5         print(msg)
6
7     return innerFunction()
8
9
10 outerFunction()
```

Abbiamo una funzione `outerFunction` che racchiude una funzione interna chiamata `innerFunction`, la quale non fa altro che stampare `msg`, variabile della `outerFunction`. Come ben sappiamo, la `innerFunction` riesce a vedere la variabile `msg` essendo racchiusa in una funzione esterna, quindi può accedere e gestire tale valore. La `outerFunction` non fa altro che restituire l'esecuzione della `innerFunction`.

Apportiamo ora un paio di modifiche al sorgente appena visto:

A screenshot of a code editor window titled 'test.py'. The code is similar to the previous example, but the 'outerFunction()' returns the 'innerFunction' object instead of calling it. In the main code, 'myFunction' is assigned the result of 'outerFunction()' and then called. The output 'Hi!' is shown in a terminal window.

```
1 def outerFunction():
2     msg = "Hi!"
3
4     def innerFunction():
5         print(msg)
6
7     return innerFunction
8
9
10 myFunction = outerFunction()
11 myFunction()
```

Questa volta notiamo che non viene restituita l'esecuzione della `innerFunction`, bensì la funzione vera e propria, la quale conserviamo nella variabile `myFunction` per poi eseguirla. Il risultato è, ovviamente, sempre lo stesso. Rispetto a prima, quindi, non abbiamo fatto altro che ritornare la funzione interna per poi eseguirla.

Nel prossimo esempio introduciamo gli argomenti:

```
test.py x
1 def outerFunction(message):
2     msg = message
3
4     def innerFunction():
5         print(msg)
6
7     return innerFunction
8
9
10 hiFunction = outerFunction("Hi!")
11 byeFunction = outerFunction("Bye")
12 hiFunction()
13 byeFunction()
```

Hi!
Bye

Ora possiamo passare un argomento alla funzione, in modo da lavorare su quest'ultimo. Vi ricorda qualcosa? Se no, eccovi un piccolo suggerimento: la funzione! Nel caso non vi sia ancora del tutto chiaro andiamo avanti, vi sarà chiaro tra pochissimo!

Riscriviamo la funzione esterna sfruttando il parametro passato come argomento direttamente all'interno della funzione interna:

```
test.py x
1 def outerFunction(message):
2     def innerFunction():
3         print(message)
4
5     return innerFunction
6
7
8 hiFunction = outerFunction("Hi!")
9 byeFunction = outerFunction("Bye")
10 hiFunction()
11 byeFunction()
```

Hi!
Bye

Ora sembra proprio uguale a qualcosa di già visto... Sarà per caso ciò?

```
test.py x
1 def myDecorator(message):
2     def wrapper():
3         print(message)
4
5     return wrapper
6
7
8 hiFunction = myDecorator("Hi!")
9 byeFunction = myDecorator("Bye")
10 hiFunction()
11 byeFunction()
```

Hi!
Bye

Ma guarda un po', sono proprio i decoratori! E, incredibilmente, funzionano proprio come una funzione! Sarà perché, magari, sono proprio delle funzioni?

Facciamo un passo avanti:

```
test.py x
1 def myDecorator(originalFunction):
2     def wrapper():
3         return originalFunction()
4
5     return wrapper
6
7
8 def myFunction():
9     print("I'm a function!")
10
11
12 decoratedFunction = myDecorator(myFunction)
13 decoratedFunction()
```

I'm a function!

Abbiamo stabilito una funzione myFunction, la quale non fa altro che stampare una stringa. La nostra funzione principale, quale è myDecorator, non fa altro che ritornare una funzione la cui esecuzione eseguirà la funzione passata come argomento. Confusionario? Forse sarebbe meglio essere un pelino più concreti:

La funzione myDecorator torna una funzione, quale è wrapper; l'esecuzione di wrapper ritorna l'esecuzione di originalFunction, che era argomento di myDecorator. Difatti, in decoratedFunction ci sarà la funzione wrapper, la quale viene eseguita nella riga sottostante tramite l'esecuzione di decoratedFunction.

Sì, è vero, si poteva eseguire direttamente myFunction per avere lo stesso risultato: il fatto è che il decoratore ci permette di aggiungere nuove funzioni, proprio come nel prossimo esempio:

```
test.py x
1 def myDecorator(originalFunction):
2     def wrapper():
3         print("I'm a decorated function!")
4         return originalFunction()
5
6     return wrapper
7
8
9 def myFunction():
10    print("I'm a function!")
11
12
13 decoratedFunction = myDecorator(myFunction)
14 decoratedFunction()
```

I'm a decorated function!
I'm a function!

Ora si rende decisamente meglio l'idea di decoratore, cioè il fatto che vengano aggiunte nuove funzioni alla funzione che viene eseguita.

A questo punto possiamo evitare l'utilizzo della variabile `decoratedFunction`: il risultato dell'esecuzione del decorator viene associato a `myFunction`, in modo tale che essa venga eseguita sempre con le nuove funzionalità offerte dal decoratore:

```
test.py x
1 def myDecorator(originalFunction):
2     def wrapper():
3         print("I'm a decorated function!")
4         return originalFunction()
5
6     return wrapper
7
8
9 def myFunction():
10    print("I'm a function!")
11
12
13 myFunction = myDecorator(myFunction)
14 myFunction()
```

```
I'm a decorated function!
I'm a function!
```

E, ovviamente, ben sappiamo che la notazione vista sopra è uguale alla seguente:

```
test.py x
1 def myDecorator(originalFunction):
2     def wrapper():
3         print("I'm a decorated function!")
4         return originalFunction()
5
6     return wrapper
7
8
9 @myDecorator
10 def myFunction():
11    print("I'm a function!")
12
13
14 myFunction()
```

```
I'm a decorated function!
I'm a function!
```

Avendo finalmente capito come funziona il decoratore più semplice del mondo, continueremo ad utilizzare la notazione appena vista anche nei prossimi esempi.

Iniziamo ora ad intaccare anche gli argomenti della funzione, altrimenti dei decorator si potrebbe fare davvero a meno:

```

test.py x
1  def myDecorator(originalFunction):
2      def wrapper():
3          print("I'm a decorated function!")
4          return originalFunction()
5
6      return wrapper
7
8
9  @myDecorator
10 def myFunction(first, second):
11     print("I'm a function!", first, second)
12
13
14 myFunction()

```

Stavolta la funzione vuole due argomenti, quali sono first e second. Si accettano scommesse: quale sarà il risultato dell'invocazione di myFunction soggetta a decoratore? Una fantastica eccezione!

```

I'm a decorated function!
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 14, in <module>
    myFunction()
  File "D:/Local Workspace/Python-Workspace/test.py", line 4, in wrapper
    return originalFunction()
TypeError: myFunction() missing 2 required positional arguments: 'first' and 'second'

```

Notiamo come il decoratore viene eseguito: la stringa da stampare viene stampata senza problemi, mentre l'esecuzione di originalFunction fallisce, perché? È molto semplice, nell'esecuzione della funzione nel decoratore non vengono passati i parametri di cui necessita la funzione da decorare. Sostanzialmente, si risolve così:

```

test.py x
1  def myDecorator(originalFunction):
2      def wrapper(*args, **kwargs):
3          print("I'm a decorated function!")
4          return originalFunction(*args, **kwargs)
5
6      return wrapper
7
8
9  @myDecorator
10 def myFunction(first, second):
11     print("I'm a function!", first, second)
12
13
14 myFunction("Pippo", "Pluto")

```

```

I'm a decorated function!
I'm a function! Pippo Pluto

```

I parametri che verranno utilizzati dalla funzione da decorare devono essere passati al wrapper. Ciò ci fa intuire un'altra cosa: è possibile, a questo punto, operare sugli argomenti del wrapper? La risposta è sì:

```
test.py x
1  def myDecorator(originalFunction):
2      def wrapper(*args, **kwargs):
3          print("I'm a decorated function!")
4          a, b = args
5          a += b
6          args = a, b
7          return originalFunction(*args, **kwargs)
8
9      return wrapper
10
11
12  @myDecorator
13  def myFunction(first, second):
14      print("I'm a function!", first, second)
15
16
17  myFunction(10, 20)
```

```
I'm a decorated function!
I'm a function! 30 20
```

È possibile, quindi, effettuare qualsiasi operazione all'interno del wrapper: è fondamentale, però, ritornare sempre l'esecuzione della funzione da decorare.

Approfondimento: closures

Le chiusure (o meglio, closures) non sono un argomento trattato nella guida, ma può essere utile capire cosa siano e come funzionino. Essendo un argomento la cui spiegazione non è affatto semplice, partiamo direttamente con un semplice esempio che si avvicinerà sempre di più al concetto di closure:

```
test.py x
1 def outerFunc():
2     message = "Hello"
3
4     def innerFunc():
5         print(message)
6
7     return innerFunc()
8
9
10 outerFunc()
```

Hello

Abbiamo una funzione esterna `outerFunc` la quale non fa altro che assegnare alla variabile `message` una stringa e ritornare l'esecuzione della funzione `innerFunc`; la funzione `innerFunc` non fa altro che stampare il contenuto della variabile `message`. Notiamo che `innerFunc` non ha parametri e che stampa una variabile al di fuori di essa: `message` viene detta "free variable" siccome non è definita nella `innerFunc`, ma è comunque raggiungibile da essa. Se eseguiamo tale codice, `outerFunc` non farà altro che richiamare `innerFunc` che stamperà `message`.

Vediamo ora qualcosa di leggermente diverso:

```
test.py x
1 def outerFunc():
2     message = "Hello"
3
4     def innerFunc():
5         print(message)
6
7     return innerFunc
8
9
10 myFunc = outerFunc()
11 myFunc()
```

Hello

Stavolta l'esecuzione di `outerFunc` non ritorna l'esecuzione di `innerFunc`, bensì ritorna la funzione stessa, la quale viene conservata in `myFunc`. Ebbene sì, le funzioni sono oggetti e gestibili come tali. Ovviamente l'esecuzione di `myFunc` corrisponderà all'esecuzione di `innerFunc`, proprio perché `myFunc` contiene `innerFunc`.

In poche parole, abbiamo una variabile `myFunc` contenente `innerFunc`, la quale può essere invocata in modo da eseguire l'invocazione di `innerFunc`. Ciò vuol dire che, in parole semplici, `myFunc` ed `innerFunc` sono in un certo senso la stessa cosa, difatti l'attributo `__name__` di

myFuncnt conterrà proprio innerFuncnt, quindi lo stesso contenuto dell'attributo `__name__` di innerFuncnt:

```
test.py x
1 def outerFuncnt():
2     message = "Hello"
3
4     def innerFuncnt():
5         print(message)
6
7     return innerFuncnt
8
9
10 myFuncnt = outerFuncnt()
11 print(myFuncnt.__name__) innerFuncnt
```

Ciò, però, è decisamente interessante: se innerFuncnt è salvata in una variabile ed è, quindi, dissociata da outerFuncnt (difatti myFuncnt non viene eseguita dentro outerFuncnt), come può la sua esecuzione stampare il contenuto di message, la quale è una variabile locale della funzione outerFuncnt? È proprio qui che subentra il concetto di closure.

In altre parole, la closure è una inner function che ricorda ed ha accesso alle variabili poste nello scope in cui è stata creata, nonostante sia in qualche modo "dissociata" dalla outer function o nonostante la outer function abbia concluso la sua esecuzione.

La situazione diventa ancora più interessante se passiamo dei parametri alla funzione esterna:

```
test.py x
1 def outerFuncnt(msg):
2     message = msg
3
4     def innerFuncnt():
5         print(message)
6
7     return innerFuncnt
8
9
10 hiFuncnt = outerFuncnt("Hi")
11 byeFuncnt = outerFuncnt("Bye")
12 hiFuncnt()
13 byeFuncnt() Hi
Bye
```

Come possiamo ben vedere, ogni funzione ricorda addirittura la stringa della variabile msg associata.

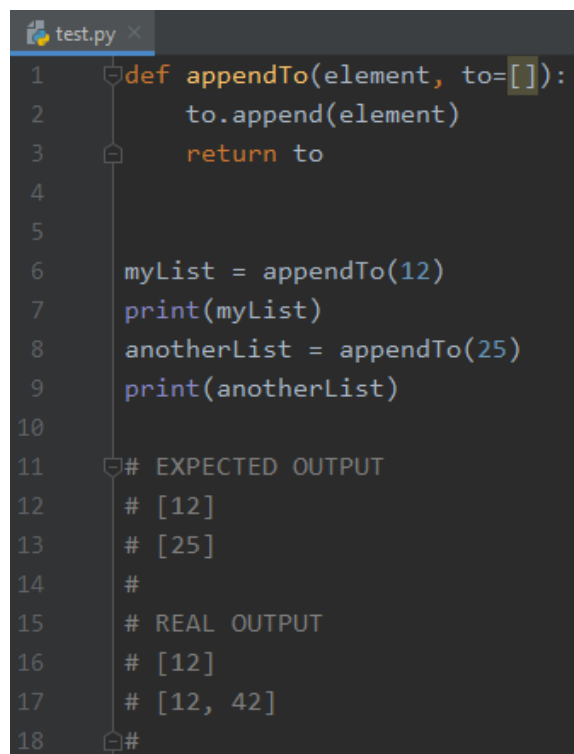
Approfondimento: common gotchas

Python tende ad essere un linguaggio pulito e consistente e cerca di evitare brutte sorprese al programmatore. Purtroppo, ci sono alcuni casi in cui i programmatori dovrebbero aspettarsi qualche sorpresa, le quali potrebbero sembrare davvero strane ed ambigue.

Suddividiamo tale capitolo in molteplici piccoli paragrafi.

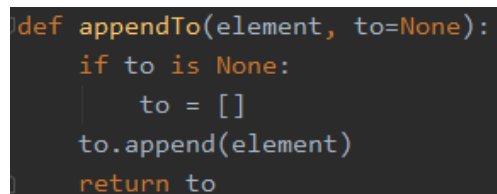
Argomenti default mutabili

Analizziamo il seguente codice, con tanto di output:



```
1 def appendTo(element, to=[]):
2     to.append(element)
3     return to
4
5
6 myList = appendTo(12)
7 print(myList)
8 anotherList = appendTo(25)
9 print(anotherList)
10
11 # EXPECTED OUTPUT
12 # [12]
13 # [25]
14 #
15 # REAL OUTPUT
16 # [12]
17 # [12, 25]
18 #
```

Ci saremmo aspettati, come output, due liste con un unico valore diverso: sbagliato. Siccome gli argomenti di default, in Python, vengono valutati solo quando la funzione viene definita e non ad ogni uso, verrà riutilizzata sempre la stessa lista ad ogni chiamata. Ciò significa che nel caso si utilizzi un oggetto mutabile come argomento di default, ad ogni chiamata verrà gestito sempre lo stesso oggetto mutabile. Piuttosto, vediamo cosa sarebbe stato corretto fare:



```
def appendTo(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

In tal modo, forziamo la creazione di una nuova lista ogni volta in cui il parametro to è None.

Chiusure con late binding

Prima di continuare con tale paragrafo, è altamente consigliabile consultare l'approfondimento sulle chiusure (closures). Analizziamo il seguente codice, con tanto di output:

```

test.py x
1  def create_multipliers():
2      multipliers = []
3      for i in range(5):
4          def multiplier(x):
5              return i * x
6
7          multipliers.append(multiplier)
8      return multipliers
9
10
11  for mult in create_multipliers():
12      print(mult(2))
13
14  # EXPECTED OUTPUT
15  # 0, 2, 4, 6, 8
16  #
17  # REAL OUTPUT
18  # 8, 8, 8, 8, 8
19  #

```

Perché vengono stampati 5 valori uguali? In `create_multipliers` vengono create 5 funzioni, in cui in ognuna la `x` viene moltiplicata per 4: in tal caso, le chiusure (o closures) sono late binding, nel senso che i valori delle variabili utilizzate dalle closures sono “bloccati” nel momento in cui la inner function viene chiamata.

Il codice appena visto non fa altro che ritornare una lista composta da 5 oggetti callable, quali sono 5 istanze della funzione `multiplier`. La funzione `multiplier` viene definita in un ciclo `for` e inserita nella lista ad ogni ciclo. Il problema è che `multiplier` eseguirà sempre $4 * x$, dove `x` è l’argomento: il valore di `i` viene controllato nel momento in cui viene effettuata la chiamata a funzione. Nell’esempio, `multiplier` viene eseguita dopo lo svolgimento della funzione `create_multipliers`, quindi in un momento in cui `i` avrà raggiunto il suo valore massimo, cioè 4.

Una soluzione è la seguente: il valore della variabile `i` viene esplicitamente salvato tramite assegnamento in una variabile interna `internal_i`. L’assegnamento viene eseguito ad ogni ciclo, in modo da salvare ogni volta il reale valore della `i`.

Il codice della soluzione è mostrato nella pagina successiva.

```
test.py x
1  def create_multipliers():
2      multipliers = []
3      for i in range(5):
4          def multiplier(x, internal_i=i):
5              return internal_i * x
6
7          multipliers.append(multiplier)
8      return multipliers
9
10
11  for mult in create_multipliers():
12      print(mult(2))
```

Approfondimento: iteratori, iterabili e generatori

In questo capitolo approfondiremo gli iteratori, gli iterabili e i generatori, in modo da poter scrivere codice più efficiente e in modo da trovare soluzioni a determinati problemi. Iniziamo col capire cosa significa che un qualcosa sia iterabile: in parole povere, è un qualcosa sul quale è possibile effettuare loop, come una lista. Vediamo un esempio concreto di loop su iterabile:

```
test.py x
1  nums = [1, 2, 3]
2
3  for num in nums:
4      print(num)
```

È possibile effettuare loop su molteplici tipi, come tuple, dizionari, stringhe, files, generators, ecc.

Ma sorge spontanea una domanda: come capiamo se un qualcosa è iterabile? Ebbene, è fondamentale sapere che qualsiasi tipo iterabile possiede un metodo chiamato `__iter__()`: per sapere se l'oggetto in questione possiede tale metodo, è necessario utilizzare la funzione `dir`:

```
test.py x
1  nums = [1, 2, 3]
2  print(dir(nums))
```

```
init_subclass_', '__iter__', '__le__', '__len__'
```

Quindi siamo tutti d'accordo sul fatto che una lista sia iterabile, ma è necessario sapere che non è un iteratore, il quale è un oggetto possedente uno stato che indichi lo stato dell'iterazione. Può sembrare strana come definizione, ma sarà tutto molto più chiaro con qualche esempio; per ora ci basta sapere che gli iteratori possiedono il metodo `__next__()`.

Nel caso ve lo stiate chiedendo, la lista non possiede il metodo `__next__()`:

```
test.py x
1  nums = [1, 2, 3]
2  print(nums.__next__())
```

```
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 2, in <module>
    print(nums.__next__())
AttributeError: 'list' object has no attribute '__next__'
```

Meglio ancora:

```
test.py x
1  nums = [1, 2, 3]
2  print(next(nums))
```

```
Traceback (most recent call last):
  File "D:/Local Workspace/Python-Workspace/test.py", line 2, in <module>
    print(next(nums))
TypeError: 'list' object is not an iterator
```

Giusto per chiarezza, l'esecuzione del metodo `next(...)` implica l'esecuzione in background del metodo `__next__()`.

In ogni caso, è possibile ottenere un oggetto iteratore da un oggetto iterabile tramite la funzione vista prima, cioè `__iter__()`:

```
test.py x
1  nums = [1, 2, 3]
2  iteratorNums = nums.__iter__()
3  print(iteratorNums)
```

```
<list_iterator object at 0x0384AE50>
```

Ovviamente, siccome i metodi magici vengono chiamati in background dai normali metodi, è possibile usare la seguente ed equivalente sintassi:

```
test.py x
1  nums = [1, 2, 3]
2  iteratorNums = iter(nums)
3  print(iteratorNums)
```

La cosa curiosa è che nel caso controllassimo i metodi del tipo iteratore, ritroviamo proprio il metodo `__iter__()`, nonostante avessimo ricavato il tipo iteratore proprio grazie a quest'ultimo metodo. Perché? Ebbene, i tipi iteratori sono anche iterabili e l'esecuzione di `__iter__()` sul tipo iteratore restituirebbe lo stesso e identico oggetto, quindi `self`.

Mettiamo alla prova il tipo iteratore:

```
test.py x
1  nums = [1, 2, 3]
2  iteratorNums = iter(nums)
3  print(next(iteratorNums))
```

Il metodo `next` restituisce il prossimo elemento della lista, in questo caso il primo elemento della lista. Come già detto, l'iteratore è un oggetto possedente uno stato, il quale non fa altro che ricordare in quale posizione si è arrivati durante l'iterazione: nel caso si riesegua il metodo `next`, con lo stato attuale, si otterrebbe il secondo elemento della lista, siccome l'iteratore ha salvato lo stato corrente dell'iterazione.

```
test.py x
1  nums = [1, 2, 3]
2  iteratorNums = iter(nums)
3  next(iteratorNums)
4  print(next(iteratorNums))  2
```

Ciò che abbiamo visto lo fa in maniera molto semplificata il `for` loop, cioè crea un iteratore sull'oggetto originale e lo itera ad ogni ciclo, finché non viene lanciata la `StopIteration` exception, automaticamente gestita dal `for` loop. Possiamo anche noi ottenere la `StopIteration` exception, iterando oltre il limite dell'iteratore, gestibile con un semplice `try except`.

Vediamo ora un esempio pratico ricreando il metodo `range` come classe che simulerà un iteratore: la classe `MyRange` possiede due attributi, quali sono la value attuale e la value massima, indicate con `value` ed `end`. Il costruttore di `MyRange` prende come argomenti due valori `start` ed `end`, i quali indicano il valore da cui partirà l'iteratore ed il valore massimo che potrà raggiungere. Sappiamo che un iteratore possiede sia il metodo `__iter__()` che il metodo `__next__()`, quindi li scriviamo simulando il funzionamento che avrebbero con un reale tipo iteratore: il metodo `__iter__()` restituirà `self`, siccome l'istanza di `MyRange` sarà l'iteratore; il metodo `__next__()` eseguirà due compiti: controllerà se la value attuale sarà maggiore del limite raggiungibile e, nel caso, lancerà l'eccezione `StopIteration`; nel caso l'eccezione non venga lanciata, viene ritornato il valore corrente ed incrementata la value per la prossima iterazione, salvando, quindi, lo stato alla prossima iterazione.

```
test.py x
1 class MyRange:
2     def __init__(self, start, end):
3         self.value = start
4         self.end = end
5
6     def __iter__(self):
7         return self
8
9     def __next__(self):
10        if self.value >= self.end:
11            raise StopIteration
12        currentValue = self.value
13        self.value += 1
14        return currentValue
15
16
17 nums = MyRange(5, 10)
18 for num in nums:
19     print(num)
```

5
6
7
8
9

Come spieghiamo, quindi, il funzionamento del ciclo for? Come già detto, il for non fa altro che richiamare il metodo next sull'oggetto da iterare finchè non si presenta la StopIteration, la quale gestirà nascondendola al programmatore. La classe che abbiamo creato, quindi, è sia un tipo iterabile che iteratore, siccome è possibile utilizzarla in un ciclo for e possiede il metodo `__next__()`.

Introduciamo ora i generatori, i quali sono estremamente utili riguardo la creazione di oggetti praticamente identici agli iteratori: si creano tramite normali funzioni le quali invece di ritornare qualcosa non fanno altro che "conservare" valori.

Sostanzialmente, i generatori non sono altro che iteratori i cui metodi magici `__iter__()` e `__next__()` vengono creati automaticamente, quindi si evita la loro creazione ed il procedimento seguito nell'esempio visto poco fa.

Vediamo un esempio pratico riguardo il come la classe scritta prima possa esser scritta come funzione generatore: sostanzialmente, si stabilisce una funzione `myRangeGen` la cui esecuzione non fa altro che "salvare" in un oggetto generatore (che ricordiamo essere praticamente identico ad un iteratore) dei valori tramite lo `yield` per poi restituirlo. L'oggetto che la funzione restituirà sarà di tipo generatore. Nell'esempio sottostante a cui stiamo facendo riferimento tornerà un oggetto generatore che itererà fino ad un certo valore `end`, quindi salverà nel generatore qualsiasi valore minore di `end`.

Non è necessario vedere tale funzione come un qualcosa di ostile: basta sapere che ad ogni invocazione dello `yield` viene salvato in un generatore il valore che accompagna l'istruzione `yield`.

```

test.py x
1 def myRangeGen(start, end):
2     current = start
3     while current < end:
4         yield current
5         current += 1
6
7
8     nums = myRangeGen(1, 5)
9     for num in nums:
10        print(num)

```

```

1
2
3
4

```

Per dimostrare il fatto che tale funzione possa esser trattata come qualsiasi altra funzione senza alcuna limitazione, potremmo inserire in un generatore solamente i valori pari:

```

test.py x
1 def myRangeGen(start, end):
2     current = start
3     while current < end:
4         if current % 2 == 0:
5             yield current
6         current += 1
7
8
9     nums = myRangeGen(1, 10)
10    for num in nums:
11        print(num)

```

```

2
4
6
8

```

Ovviamente, essendo i generatori eseguibili in un for, possiamo dire siano tipi iterabili; inoltre sappiamo scrivono in background anche i metodi `__iter__()` e `__next__()`, quindi possiamo affermare siano anche tipi iteratori.

Ultimo appunto riguardo gli iteratori è il fatto che non necessariamente debbano avere un valore limite, quindi la funzione `__next__()` funzionerà all'infinito. Riprendiamo l'esempio di `myRangeGen` (useremo tale esempio per semplicità, siccome il generatore è anche un iteratore) ed evitiamo di impostare un valore limite:

```

test.py x
1 def myRangeGen(start):
2     current = start
3     while True:
4         yield current
5         current += 1
6
7
8     nums = myRangeGen(1)
9     for num in nums:
10        print(num)

```

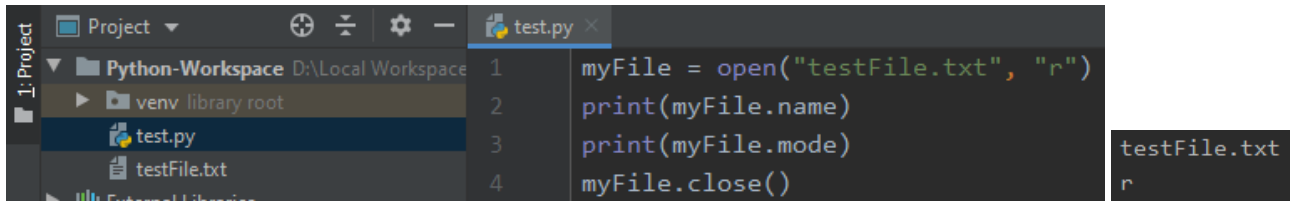
```

5789211
5789212
5789213
5789214
5789215
5789216
5789217
5789218
5789219
5789220
Process finished with exit code -1

```

Approfondimento: lettura e scrittura su files

In questo capitolo approfondiremo tramite l'utilizzo di molteplici esempi l'utilizzo dei files. Iniziamo con qualche esempio che utilizzi un file, situato nella stessa cartella in cui si trova il sorgente:



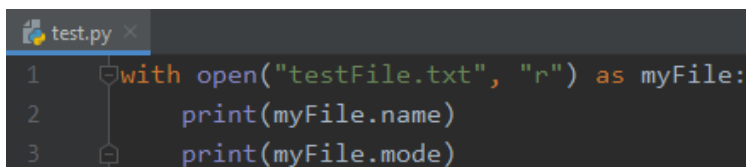
```
1 myFile = open("testFile.txt", "r")
2 print(myFile.name)
3 print(myFile.mode)
4 myFile.close()
```

Il file viene aperto tramite la funzione `open`, specificando il suo nome con estensione e la modalità di apertura, la quale viene scelta tra le seguenti:

- “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.

La `open` ritorna un oggetto che rappresenta in sé e per sé il file, con associati metodi che ne permettono la sua gestione: difatti, nell'esempio stampiamo il nome del file e la modalità con cui è stato aperto.

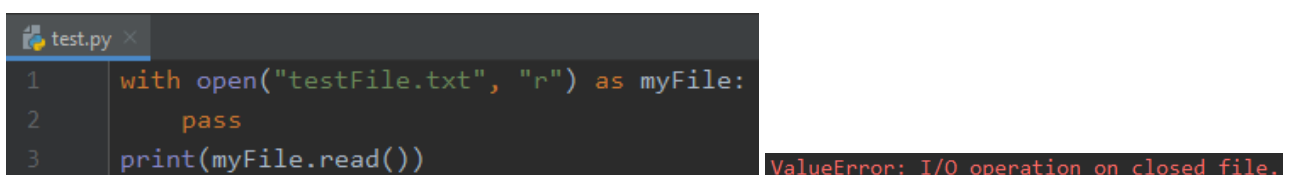
È importante ricordare di chiudere sempre il file, onde non riscontrare leaks nel funzionamento dei nostri programmi. La gestione della chiusura del file per i programmatori non è obbligatoria, difatti esiste un modo per evitarne la chiusura manuale: i context managers.



```
1 with open("testFile.txt", "r") as myFile:
2     print(myFile.name)
3     print(myFile.mode)
```

Sostanzialmente, tale sintassi ci permette di stabilire un blocco di codice dedicato interamente al file su cui si sta lavorando. All'uscita da tale blocco, l'oggetto file viene automaticamente chiuso, automatizzando, quindi, la chiusura e facendo evitare la gestione di essa al programmatore.

Ovviamente, nel caso si provi ad accedere al file dall'esterno del context manager, verrà lanciata un'eccezione:



```
1 with open("testFile.txt", "r") as myFile:
2     pass
3     print(myFile.read())
```

ValueError: I/O operation on closed file.

Sia chiaro: l'accesso al file al di fuori del context manager è vietato siccome il file è stato chiuso, ma l'accesso all'oggetto file no. Infatti, è possibile accedere all'oggetto file senza alcun problema siccome è stato chiuso solamente lo stream del file vero e proprio.

```
test.py x
1 with open("testFile.txt", "r") as myFile:
2     pass
3     print(myFile.name)
4     print(myFile.read())
```

ValueError: I/O operation on closed file.
testFile.txt

Per leggere da un file utilizzeremo i seguenti metodi:

- `read()` legge l'intero file;
- `readlines()` legge l'intero file ponendolo su un'unica riga e mostrando i caratteri speciali, come il carattere di newline;
- `readline()` legge la prossima riga del file;

L'utilizzo della `read` è sconsigliato per files di grandi dimensioni, siccome la lettura dell'intero file potrebbe occupare l'intera memoria. Nel caso, però, sia necessario ottenere lo stesso e identico risultato della `read` senza eventuali problemi con la memoria, è possibile scansionare il file con un semplice ciclo `for`, siccome l'oggetto file è iterabile:

```
test.py x
1 with open("testFile.txt", "r") as myFile:
2     for line in myFile:
3         print(line, end="")
```

1. First line
2. Second line
3. Third line

Nel caso si voglia, invece, leggere un determinato numero di caratteri, è necessario utilizzare la `read` con un suo parametro che specifica il numero di caratteri da leggere:

```
test.py x
1 with open("testFile.txt", "r") as myFile:
2     print(myFile.read(35))
```

1. First line
2. Second line
3. Thi

Bisogna sapere, inoltre, che l'utilizzo della `read` porta avanti un indicatore, il quale terrà conto dell'attuale posizione nel file. Ciò ci fa intuire che nel caso si utilizzino molteplici `read` con specificati caratteri da leggere, a seguire dalla seconda invocazione in poi si continuerà a leggere da dove l'indicatore si sarà fermato con la precedente lettura.

```
test.py x
1 with open("testFile.txt", "r") as myFile:
2     print(myFile.read(5))
3     print(myFile.read(15))
```

1. Fi
rst line
2. Sec

Sapendo ora che la `read` può leggere un determinato numero di caratteri alla volta, potremmo valutare di effettuare un ciclo in cui si legga l'intero file "a spezzoni" utilizzando la `read`.

Ricordiamo che la `read` restituisce ciò che si legge, mentre nel caso non ci sia nulla da leggere ritorna una stringa vuota. Potremmo usare la lunghezza della stringa a nostro favore, come nel seguente esempio:

```

test.py x
1 with open("testFile.txt", "r") as myFile:
2     sizeToRead = 10
3     readString = myFile.read(sizeToRead)
4     while len(readString) > 0:
5         print(readString, end="")
6         readString = myFile.read(sizeToRead)

```

```

1. First line
2. Second line
3. Third line

```

In tal modo, ad ogni ciclo vengono letti 10 caratteri e stampati. Nel caso non ci sia più nulla da leggere e venga restituita una stringa vuota, si uscirà dal ciclo e non verrà stampato più nulla.

Per rendere più chiaro dove viene interrotta ogni read, all'attributo end della print associato una star. Il risultato è il seguente:

```

1. First l*ine
2. Sec*ond line
3*. Third li*ne*

```

A questo punto sorge spontanea la domanda: se utilizzando la read l'indicatore va sempre avanti, è possibile riportarlo all'inizio del file e ricominciare la lettura del file? La risposta è sì: applicando alcune funzioni di gestione dell'indicatore, sarà possibile modificare la sua posizione. La funzione tell ritorna la posizione attuale dell'indicatore; la funzione seek cambia la posizione dell'indicatore.

```

test.py x
1 with open("testFile.txt", "r") as myFile:
2     sizeToRead = 10
3     readString = myFile.read(sizeToRead)
4     print(readString, end="")
5     readString = myFile.read(sizeToRead)
6     print(readString, end="")
7     if myFile.tell() > 15:
8         myFile.seek(0)
9     readString = myFile.read(sizeToRead)
10    print(readString, end="")

```

```

1. First line
2. Sec1. First l

```

Nell'esempio sopra mostrato, con le due read si porta l'indicatore del file alla posizione che segue ciò che si è letto (si son letti 20 caratteri, quindi l'indicatore sarà in posizione 21); nell'IF si controlla se l'indicatore è posto in una posizione maggiore a 15 (in tal caso è banalmente vero): si procede, quindi, con la seek(0), quindi si riporta l'indicatore sulla posizione iniziale; la prossima lettura comincerà dall'inizio.

Riguardo la scrittura su file, invece, iniziamo con l'aprire un file in modalità scrittura:

```

test.py x
1 with open("testFile.txt", "w") as myFile:
2     pass

```

Aperto il file in modalità scritta, ci scriviamo immediatamente dentro tramite la funzione write:

```

test.py x testFile.txt x
1 with open("testFile.txt", "w") as myFile:
2     myFile.write("Hello World")

```

```

test.py x testFile.txt x
1 Hello World

```

Come potrete bene intuire, la write utilizza l'indicatore del file: sostanzialmente, la write scriverà a partire dalla posizione dell'indicatore e, nel caso sia necessario sovrascrivere, sovrascriverà solo dove ce ne sarà bisogno. Può sembrare lecita la domanda: "quando mi capiterà mai di dover sovrascrivere in un file?". Ebbene, eccovi la risposta:

```

test.py x testFile.txt x
1 with open("testFile.txt", "w") as myFile:
2     myFile.write("Hello World")
3     myFile.seek(3)
4     myFile.write("Hello")

```

```

test.py x testFile.txt x
1 HelHelloWorld

```

Come possiamo ben vedere, viene sovrascritto solo il necessario e non viene cancellato ciò che è presente dopo.

Come esempio riepilogativo vedremo la copia di un file esistente in un altro file:

```

test.py x testFile.txt x testFile_copy.txt x
1 with open("testFile.txt", "r") as source:
2     with open("testFile_copy.txt", "w") as destination:
3         for line in source:
4             destination.write(line)

```

```

test.py x testFile.txt x testFile_copy.txt x
1 Hello there! That's a file!
2 I'm full of contents :D
3 Do you like it?

```

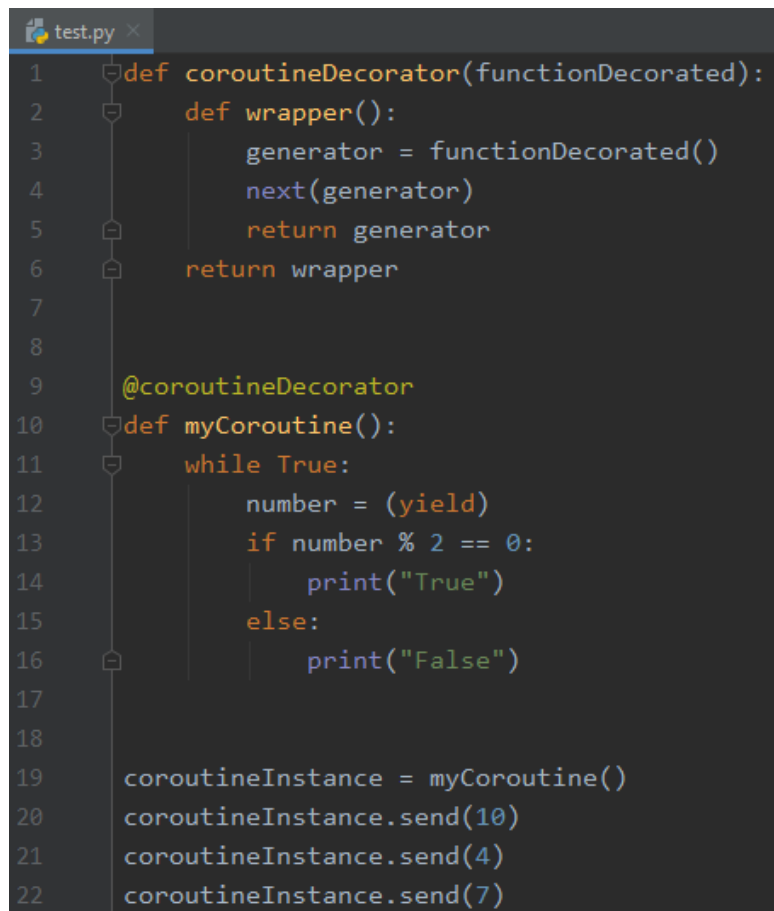
```

test.py x testFile.txt x testFile_copy.txt x
1 Hello there! That's a file!
2 I'm full of contents :D
3 Do you like it?

```

Approfondimento: coroutines

Una coroutine non è altro che una semplice funzione la quale può fermarsi e riprendere l'esecuzione in qualsiasi momento nel suo ciclo di vita, precisamente facendo uso della keyword `yield`. Come ben sappiamo, `yield` viene usata nei generatori per ottenere un "insieme" di valori su cui poter iterare; nelle coroutines, `yield` blocca la funzione finché essa non riceve un valore tramite il metodo `send(value)`: l'esecuzione del metodo `send` fa ricevere all'istruzione `yield` un valore, il quale verrà conservato in una variabile e gestito, come nel seguente esempio:



```
1 def coroutineDecorator(functionDecorated):
2     def wrapper():
3         generator = functionDecorated()
4         next(generator)
5         return generator
6     return wrapper
7
8
9 @coroutineDecorator
10 def myCoroutine():
11     while True:
12         number = (yield)
13         if number % 2 == 0:
14             print("True")
15         else:
16             print("False")
17
18
19 coroutineInstance = myCoroutine()
20 coroutineInstance.send(10)
21 coroutineInstance.send(4)
22 coroutineInstance.send(7)
```

Vediamo passo per passo il procedimento, tralasciando un momento il decoratore: con l'esecuzione della funzione `myCoroutine` ritorniamo una coroutine vera e propria, sulla quale potremo operare. Prima di eseguire le `send`, è bene sapere che la coroutine è bloccata sulla `yield`, in attesa di un valore: proprio grazie alle successive `send`, la coroutine riceve il valore passato come parametro e lo elabora. Prendiamo la riga 20: sulla coroutine viene invocato il metodo `send(10)`, quindi lo `yield` della coroutine restituisce 10, il quale viene conservato in `number` ed elaborato. Essendoci un `while True`, finita l'elaborazione ricomincerà il `while`, bloccandosi di nuovo sullo `yield`, in attesa di un nuovo valore.

Ora è lecito sapere a cosa serve il `coroutineDecorator`: tale decoratore non fa altro che far avanzare l'esecuzione della funzione fino al primo `yield`, in modo tale che da quel momento in poi si possano accettare valori tramite chiamata a funzione `send(value)`. Difatti, la coroutine deve ricevere le `send` solo quando si trova bloccata sulla `yield`, quindi effettuare una chiamata a funzione `next` è necessario. Nel caso ve lo fosse chiesti, non è obbligatorio utilizzare tale

decoratore, ma ciò non toglie sia comodo. Vediamo un esempio nel quale il decoratore non viene utilizzato:

```
test.py x
1 def myCoroutine():
2     while True:
3         number = (yield)
4         if number % 2 == 0:
5             print("True")
6         else:
7             print("False")
8
9
10 coroutineInstance = myCoroutine()
11 next(coroutineInstance)
12 coroutineInstance.send(10)
13 coroutineInstance.send(4)
14 coroutineInstance.send(7)
```

Il ragionamento è semplice: il decoratore non fa altro che far avanzare l'esecuzione della funzione fino al primo yield, operazione fattibile anche senza quest'ultimo invocando semplicemente la next sul generatore restituito dall'esecuzione di myCoroutine.

Vediamo un esempio riepilogativo:

```
test.py x
1 def bookCoroutine():
2     books = ["FirstBook", "SecondBook", "ThirdBook"]
3     while True:
4         inputBook = (yield)
5         if inputBook in books:
6             print("The book {} is in the list!".format(inputBook))
7         else:
8             print("The book {} does not exists".format(inputBook))
9
10
11 bookSearcher = bookCoroutine()
12 next(bookSearcher)
13 bookSearcher.send("FirstBook")
14 bookSearcher.send("AnotherBook")
15 bookSearcher.send("SecondBook")
```

```
The book FirstBook is in the list!
The book AnotherBook does not exists
The book SecondBook is in the list!
```

Concludiamo, infine, vedendo un tipo di coroutine che richiede l'utilizzo dei metodi next e send concatenati. Tale tipologia di coroutine richiede l'utilizzo di due yield: uno yield otterrà un valore, un altro yield conserverà il valore nel generatore e sarà disposto anche a ritornarlo, in un certo

senso. Per essere più precisi, quando con la send invieremo un valore, uno yield conserverà il valore mentre un altro yield calcolerà secondo le nostre istruzioni un altro valore e lo ritornerà.

Vediamo un esempio:

```
test.py x
1  def coroutine(myFunction):
2      def wrapper(*args, **kwargs):
3          generator = myFunction(*args, **kwargs)
4          next(generator)
5          return generator
6      return wrapper
7
8
9  @coroutine
10 def doDouble():
11     value = 0
12     while True:
13         value = (yield)
14         value += 1
15         yield value * 2
16
17
18 myGen = doDouble()
19 print(myGen.send(5))
20 next(myGen)
21 print(myGen.send(15))
22 next(myGen)
23 print(myGen.send(10))
24 next(myGen)
```

12
32
22

Come possiamo ben vedere, la prima send invia il 5 alla prima yield, la quale conserverà tale valore in value; la seconda yield, invece, ritorna value, eventualmente modificato tramite operazioni, funzioni o metodi (in tal caso si incrementa value di uno e poi lo si raddoppia).

Sostanzialmente, quindi, la send manda un valore alla prima yield e ne ottiene un altro dalla seconda yield.

Vediamo, però, viene eseguita l'istruzione next subito dopo ogni send: questo perché nel caso venga effettuata solamente la send, il generatore si bloccherebbe dopo il ritorno del valore tramite yield. La next serve a "mandare avanti" il generatore, in questo caso a fargli ricominciare il ciclo while.

Approfondimento: la magia di __slots__

La variabile `__slots__` permette di risparmiare il consumo di RAM: essendo il dict davvero pesante, è possibile sostituirlo, in determinati casi, con `__slots__`, il quale occupa circa il 40-50% di spazio in meno in memoria RAM. Sostanzialmente, Python usa un dict per conservare gli attributi di un'istanza, allocando memoria dinamica per ogni dict. Python non può allocare un quantitativo statico di memoria alla creazione dell'oggetto per poter conservare gli attributi, quindi è possibile utilizzare `__slots__` per allocare spazio solo per quel determinato set di attributi.

Vediamo un esempio senza l'utilizzo di `__slots__`:

```
test.py x
1 class MyClass:
2     def __init__(self, name, identifier):
3         self.name = name
4         self.identifier = identifier
```

Ora, invece, vediamo un esempio con l'utilizzo di `__slots__`:

```
test.py x
1 class MyClass:
2     __slots__ = ["name", "identifier"]
3
4     def __init__(self, name, identifier):
5         self.name = name
6         self.identifier = identifier
```

È bene precisare che l'utilizzo di `__slots__` esclude l'utilizzo della variabile dict contenente le variabili di istanza: `__slots__` conterrà le variabili di istanza allocando staticamente memoria. Ciò significa che, dopo l'inizializzazione, non sarà possibile aggiungere nuove variabili di istanza. Nel caso fosse permesso, si tratterebbe di allocazione dinamica della memoria, la quale è decisamente dispendiosa.

Approfondimento: Process e ProcessPoolExecutor

In questo capitolo vediamo come eseguire codice in parallelo utilizzando i processi. L'uso di molteplici processi è dato dal fatto che si voglia un aumento nelle prestazioni del sistema, eseguendo tasks in parallelo. Inizieremo con esempi basici per poi passare ai più complessi.

Iniziamo col vedere un piccolo script che consiste nell'attendere un secondo per poi misurare l'intera durata dell'esecuzione:

```
test.py x
1  import time
2
3  start = time.perf_counter()
4
5
6  def operation():
7      print("Sleeping 1 second...")
8      time.sleep(1)
9      print("Done sleeping!")
10
11
12  operation()
13  finish = time.perf_counter()
14  print("Finished in {} seconds".format(round(finish - start, 5)))
```

Sleeping 1 second...
Done sleeping!
Finished in 1.00196 seconds

Ovviamente nel caso accodassimo più volte l'uso di operation, si avrebbe un tempo di esecuzione maggiore: ciò perché le istruzioni vengono eseguite linearmente, quindi si accoda il tempo da attendere. Cosa succede, però, se le operation venissero svolte contemporaneamente? Ebbene, si attenderebbe un secondo circa in totale: è questo il risparmio che vogliamo!

Si vuole, quindi, eseguire la operation in parallelo: iniziamo importando il modulo multiprocessing per poi utilizzare la classe Process, la quale ci permette di eseguire funzioni in parallelo.

È importante specificare il fatto che Windows sia limitato riguardo la gestione dei processi, difatti durante la creazione del seguente esempio ci son stati non pochi problemi. Un paio di consigli da adottare è il fatto di dover utilizzare sempre il main ed il fatto di dover dichiarare al di fuori del main le funzioni, onde evitare problemi con la creazione dei processi. Fatto ciò, dovrebbe andare tutto per il meglio.

L'esempio è mostrato nella pagina seguente.


```
test.py x
1  import time
2  import multiprocessing
3
4
5  def operation():
6      print("Sleeping 1 second...")
7      time.sleep(1)
8      print("Done sleeping!")
9
10
11  if __name__ == '__main__':
12      start = time.perf_counter()
13      p1 = multiprocessing.Process(target=operation)
14      p2 = multiprocessing.Process(target=operation)
15      p3 = multiprocessing.Process(target=operation)
16      processList = [p1, p2, p3]
17      for p in processList:
18          p.start()
19      for p in processList:
20          p.join()
21      finish = time.perf_counter()
22      print("Finished in {} seconds".format(round(finish - start, 2)))

Sleeping 1 second...
Sleeping 1 second...
Sleeping 1 second...
Done sleeping!Done sleeping!Done sleeping!

Finished in 1.28 seconds
```

Notiamo, quindi, l'utilizzo della classe `Process`, la quale prende come parametro un `target`, quale non è altro che la funzione che il processo dovrà eseguire una volta partito tramite il metodo `start`: la creazione del processo non implica il suo avvio, difatti è sempre necessario l'utilizzo della `start` per avviarlo. Fatti partire tutti i processi, è necessario che il processo principale (il `main`) attenda i processi figli: ciò è possibile tramite il metodo `join`, il quale blocca l'esecuzione del processo attuale (in questo caso, appunto, il `main`) finché il processo su cui è stata invocata la `join` non finirà l'esecuzione. Ovviamente nel caso non venga fatta alcuna `join`, il processo principale si concluderebbe prima della fine dell'esecuzione del processo figlio.

È possibile, però, che la funzione `target` possa assumere dei parametri per svolgere il proprio lavoro. Come visto finora non sarebbe possibile specificarli, ma un modo c'è ed è tramite il costruttore di `Process`, il quale accetterà come argomento una lista di argomenti denotata come `args`. È importante che, appunto, `args` sia una lista, siccome è possibile passare molteplici argomenti alla funzione `target`.

L'esempio è mostrato nella pagina successiva.

```
test.py x
1  import time
2  import multiprocessing
3
4
5  def operation(seconds):
6      print("Sleeping {} second...".format(seconds))
7      time.sleep(seconds)
8      print("Done sleeping!")
9
10
11  if __name__ == '__main__':
12      start = time.perf_counter()
13      p1 = multiprocessing.Process(target=operation, args=[1.5])
14      p2 = multiprocessing.Process(target=operation, args=[2])
15      p3 = multiprocessing.Process(target=operation, args=[1])
16      processList = [p1, p2, p3]
17      for p in processList:
18          p.start()
19      for p in processList:
20          p.join()
21      finish = time.perf_counter()
22      print("Finished in {} seconds".format(round(finish - start, 2)))

Sleeping 1.5 second...
Sleeping 2 second...
Sleeping 1 second...
Done sleeping!
Done sleeping!
Done sleeping!
Finished in 2.38 seconds
```

Python ha successivamente introdotto il process pull executor, il quale è un utile strumento per eseguire processi in maniera più semplice ed efficiente. Ne vediamo il suo utilizzo nel prossimo esempio, importandolo nello script e gestendolo come context manager. Ottenuto un oggetto `ProcessPoolExecutor` dal context manager, dichiarato come executor, sarà possibile effettuare operazioni sui processi: per creare un processo, passargli la funzione da eseguire ed avviarlo, basta un unico metodo, quale è `executor.submit`, il quale accetta come parametri la funzione da eseguire ed i suoi argomenti. Il metodo `submit` ritorna un oggetto `future`, sul quale è possibile effettuare operazioni riguardanti il processo, come l'ottenere il valore di ritorno dell'esecuzione della funzione passata al processo: ciò viene fatto tramite l'esecuzione del metodo `result`.

Nell'esempio seguente abbiamo due liste: una per i futures, quali non sono che i processi del pool, ed una per i results, la quale conterrà i risultati delle esecuzioni dei processi. Ad ogni iterazione viene creato un processo tramite la `submit` e viene accodato alla lista dei futures; viene poi salvato, per ogni future, il proprio valore di ritorno accodandolo nella lista dei results; si conclude mostrando tutti gli output.

```

test.py x
1  import time
2  import concurrent.futures
3
4
5  def operation(seconds):
6      print("Sleeping {} second...".format(seconds))
7      time.sleep(seconds)
8      return "Done sleeping!"
9
10
11  if __name__ == '__main__':
12      start = time.perf_counter()
13      with concurrent.futures.ProcessPoolExecutor() as executor:
14          listOfFutures = []
15          listOfResults = []
16          for i in range(10):
17              listOfFutures.append(executor.submit(operation, i+1))
18          for i in range(10):
19              listOfResults.append(listOfFutures[i].result())
20          for i in range(10):
21              print(listOfResults[i])
22      finish = time.perf_counter()
23      print("Finished in {} seconds".format(round(finish - start, 2)))

```

Tornando alla gestione di Process, come abbiamo già visto, è necessario eseguire il metodo start per avviare l'esecuzione del processo generato. È possibile generare ed avviare i processi in una maniera decisamente più efficiente rispetto a quella mostrata precedentemente. Come mostrato, venivano prima creati tutti i processi e dopo eseguiti tutti i processi: ne passa di tempo durante la creazione di tutti i processi, lo si potrebbe sfruttare per eseguire, nel frattempo, delle operazioni, no? Ebbene, risulta efficiente, come è giusto che sia, eseguire il processo subito dopo la creazione. Se prima si creavano tutti i processi e poi eseguiti, ora possiamo creare un processo, eseguirlo, passare alla creazione del prossimo e la sua esecuzione e così via.

Con il metodo visto precedentemente, dati i processi A, B, C:

A, B, C vengono creati e poi eseguiti, mettendoci tempo $a+b+c$.

Con il metodo appena visto, dati i processi A, B, C:

A viene creato ed eseguito, B viene creato ed eseguito, C viene creato ed eseguito. In tal modo, mentre B viene creato, A sta già eseguendo. Lo stesso discorso vale per A con C e B con C. Il tempo di esecuzione sarà sicuramente minore ad $a+b+c$.

Lo pseudocodice di quest'ultimo metodo è:

```

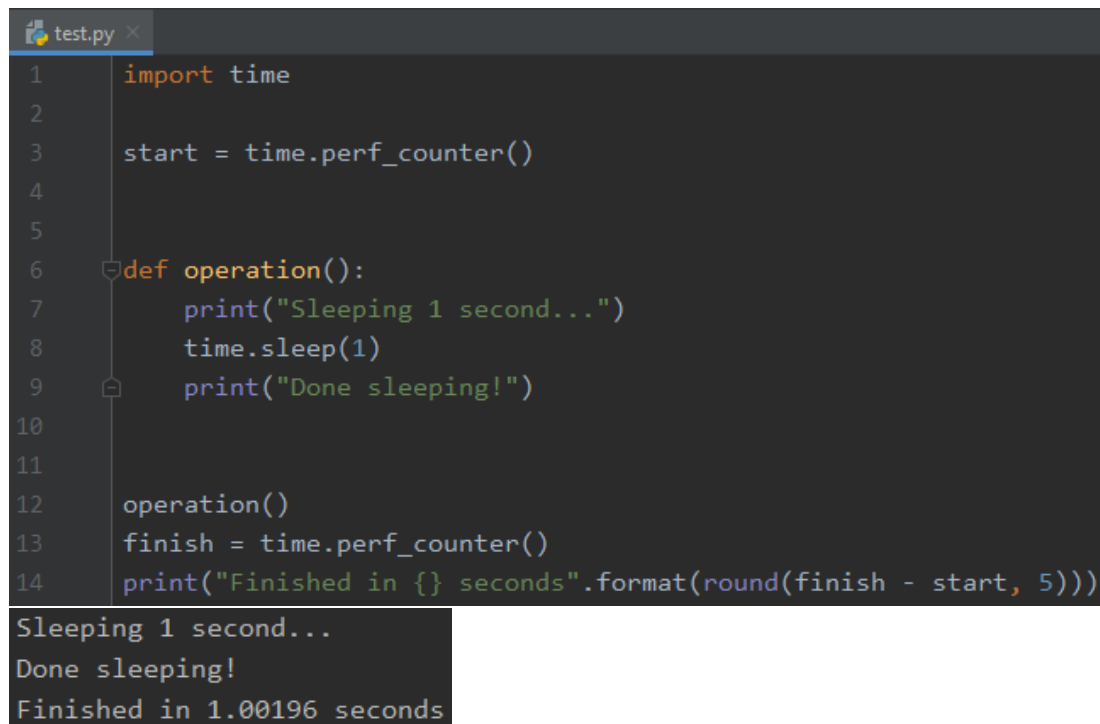
A = Process(target=funct); A.start(); B = Process(target=funct); B.start(); C = Process(target=funct);
C.start(); A.join(); B.join(); C.join();

```

Approfondimento: Thread e ThreadPoolExecutor

In questo capitolo vedremo come eseguire codice concorrente utilizzando i threads. Vedremo una serie di esempi riferiti a quelli del capitolo precedente, trattante i processi. Come i processi, anche i thread permettono di ottimizzare l'esecuzione dello script: la tecnica migliore dipende dal tipo di operazione da eseguire in concorrenza (ad esempio, l'utilizzo dei processi è consigliato per le operazioni CPU-bound, mentre l'utilizzo dei thread è consigliato per le operazioni I/O-bound).

Prendiamo il seguente esempio:



```
test.py x
1  import time
2
3  start = time.perf_counter()
4
5
6  def operation():
7      print("Sleeping 1 second...")
8      time.sleep(1)
9      print("Done sleeping!")
10
11
12  operation()
13  finish = time.perf_counter()
14  print("Finished in {} seconds".format(round(finish - start, 5)))

Sleeping 1 second...
Done sleeping!
Finished in 1.00196 seconds
```

Ne evitiamo la spiegazione siccome è stata già trattata all'inizio del capitolo precedente.

Il principio che si vuole seguire è lo stesso mostrato con i processi: si vuole eseguire codice in concorrenza, in tal caso la funzione operation molteplici volte contemporaneamente.

Iniziamo, quindi, importando il modulo threading ed utilizzando la classe Thread per istanziare, appunto, i threads: proprio come già visto con i processi, basta istanziare un oggetto Thread passandogli la funzione da eseguire come argomento, per poi avviarlo tramite il metodo start ed attenderlo tramite il metodo join, evitando che il thread principale (in tal caso il main) si concluda prima dei thread creati. Ovviamente è possibile passare argomenti alla funzione da eseguire proprio come se si stessero gestendo i processi, quindi tramite una lista args.

Nota da fare per i thread è il fatto che non hanno problemi con Windows, quindi non bisogna agire come fatto con i processi creando il main e definendo le funzioni fuori da esso.

Gli esempi sono mostrati nella pagina successiva.

```

test.py x
1  import threading
2  import time
3
4      start = time.perf_counter()
5
6
7  def operation():
8      print("Sleeping 1 second...")
9      time.sleep(1)
10     print("Done sleeping!")
11
12
13     t1 = threading.Thread(target=operation)
14     t2 = threading.Thread(target=operation)
15     t1.start()
16     t2.start()
17     t1.join()
18     t2.join()
19     finish = time.perf_counter()
20     print("Finished in {} seconds".format(round(finish - start, 2)))

```

```

test.py x
1  import threading
2  import time
3
4      start = time.perf_counter()
5
6
7  def operation(seconds):
8      print("Sleeping {} second...".format(seconds))
9      time.sleep(seconds)
10     print("Done sleeping!")
11
12
13     listOfThreads = []
14     for i in range(10):
15         t = threading.Thread(target=operation, args=[i])
16         t.start()
17         listOfThreads.append(t)
18     for i in range(10):
19         listOfThreads[i].join()
20
21     finish = time.perf_counter()
22     print("Finished in {} seconds".format(round(finish - start, 2)))

```

Come con i processi, anche con i thread esiste un pool che automatizzi l'uso dei threads.

Tale pool è detto `ThreadPoolExecutor` e restituisce un oggetto che interpretiamo come `executor` nel context manager. Ottenuto un oggetto `ThreadPoolExecutor` dal context manager, dichiarato come `executor`, sarà possibile effettuare operazioni sui threads: per creare un thread, passargli la funzione da eseguire ed avviarlo, basta un unico metodo, quale è `executor.submit`, il quale accetta come parametri la funzione da eseguire ed i suoi argomenti. Il metodo `submit` ritorna un oggetto `future`, sul quale è possibile effettuare operazioni riguardanti il thread, come l'ottenere il valore di ritorno dell'esecuzione della funzione passata al thread: ciò viene fatto tramite l'esecuzione del metodo `result`. Per utilizzare tale pool è necessario importare il modulo `concurrent.futures`.

```
test.py x
1  import concurrent.futures
2  import time
3
4  start = time.perf_counter()
5
6
7  def operation(seconds):
8      print("Sleeping {} second...".format(seconds))
9      time.sleep(seconds)
10     return "Done sleeping!"
11
12
13  with concurrent.futures.ThreadPoolExecutor() as executor:
14      listOfSeconds = [1, 2, 3, 4, 5]
15      listOfThreads = []
16      listOfResults = []
17      for s in listOfSeconds:
18         listOfThreads.append(executor.submit(operation, s))
19      for i in range(len(listOfSeconds)):
20         listOfResults.append(listOfThreads[i].result())
21      for i in range(len(listOfSeconds)):
22         print(listOfResults[i])
23  finish = time.perf_counter()
24  print("Finished in {} seconds".format(round(finish - start, 2)))
```