python

generatori

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

generatori

I generatori sono uno strumento semplice e potente per la creazione di iteratori (qualsiasi oggetto abbia implementati i metodi __iter__ e __next__). Possono essere definiti anche mediante funzioni regolari che usano la keyword yield ogni volta che vogliono restituire un dato.

```
>>> def gen():
...     for n in range(3):
...             yield n
...
>>> for n in gen():
...     print(n)
...
0
1
2
```

I generatori, a dispetto delle funzioni, hanno il vantaggio di poter interrompere e di riprendere l'esecuzione quando viene invocata la funzione next().
Inoltre il loro utilizzo risulta vantaggioso circa l'impiego di memoria in quanto viene generato un oggetto alla volta.

```
>>> def gen():
...     for n in range(5):
...             yield n
...
>>> iter_gen = iter(gen())
>>> next(iter_gen)
0
>>> next(iter_gen)
1
>>> next(iter_gen)
2
>>> next(iter_gen)
3
>>> next(iter_gen)
4
>>> next(iter_gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

generatori - generator expression -

Un metodo sintetico e veloce per creare un generatore è il generator expression. La sintassi deriva dalla syntax comprehension di Python (vedi slide successive)

Il generatore creato attraverso la generator expression ha le stesse caratteristiche di quello definito con una funzione ma risulta più sintetico anche se meno versatile.

```
>>> def gen():
...     for n in range(5):
...             yield n
...
>>> iter_gen = iter(gen())
>>> next(iter_gen)
0
>>> next(iter_gen)
1
>>> next(iter_gen)
2
>>> next(iter_gen)
3
>>> next(iter_gen)
4
>>> next(iter_gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

generatori - generator expression -

Può essere più efficiente usare le generator expression anche in combinazione con funzioni built-in tipiche degli oggetti iterabili (max(), min(), sum(), etc.)

```
>>> sum([x for x in range(5)]) # list comprehension
10
>>> sum(x for x in range(5)) # generator comprehension
10
```

Il valore di ritorno delle due funzioni è il medesimo (sum() → somma tutti gli elementi di un oggetto iterabile: 0+1+2+3+4=10), ma nel primo caso verrà creato l'oggetto lista sul quale verrà utilizzata la funzione sum() e poi verrà cancellato, con la generator expression si creerà una variabile alla quale verrà addizionato l'elemento successivo ad ogni ciclo.

La funzione getsizeof() ritorna le dimensioni di un oggetto in byte. Di seguito si può notare quanto un generatore occupi molta meno memoria dell'omologo lista.

```
>>> from sys import getsizeof
>>> getsizeof([x for x in range(1000)])
9016
>>> getsizeof(x for x in range(1000))
112
```

syntax comprehension

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - syntax comprehension -

La syntax comprehension fornisce un modo conciso per creare qualsiasi tipo di struttura dati. Le applicazioni comuni sono di creare nuovi oggetti iterabili in cui ogni elemento è il risultato di alcune operazioni applicate a ciascun membro di un'altra sequenza o di creare una sottosequenza di quegli elementi che soddisfano una determinata condizione.

```
>>> quadrati = []
>>> for n in range(11):
...     quadrati.append(n ** 2)
...
>>>
>>> quadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Forma tradizionale:
- lista vuota;
- for loop;
- append di ogni elemento alla potenza di 2.

Syntax comprehension:
- elemento della sequenza (opzionale: espressione → n2);
- for loop.

```
>>> quadrati = [n ** 2 for n in range(11)]
>>> quadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - syntax comprehension -

La syntax comprehension può generare qualsiasi tipo di struttura dati e generatori:

Lista

```
>>> l = [n for n in range(5)]
>>> l
[0, 1, 2, 3, 4]
```

Dizionario

```
>>> d = dict((k,v) for k,v in zip(range(5), 'abcde'))
>>> d
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

Tupla

```
>>> t = tuple(n for n in range(5))
>>> t
(0, 1, 2, 3, 4)
```

Set

```
>>> s = {n for n in range(5)}
>>> s
{0, 1, 2, 3, 4}
```

N.B. È necessario usare il costruttore tuple(). Con le sole parentesi tonde si crea un generatore.

Generatore

```
>>> g = (n for n in range(5))
>>> g
<generator object <genexpr> at 0x7f2481f7fb30>
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - syntax comprehension -

È inoltre possibile stabilire delle condizioni (con conditional statement) nella struttura della syntax comprehension.

Condizione x % 2 != 0 per creare una lista di quadrati di numeri dispari

```
>>> l = [x ** 2 for x in range(11) if x % 2 != 0]
>>> l
[1, 9, 25, 49, 81]
```

```
>>> l = [(n, 'pari') if n % 2 == 0 else (n, 'dispari') for n in range(5)]
>>> for el in l:
...     print(el)
...
(0, 'pari')
(1, 'dispari')
(2, 'pari')
(3, 'dispari')
(4, 'pari')
```

- ogni elemento annidato in una tupla;
- condizioni multiple;

stringhe, byte, unicode

python - stringhe, byte, unicode -

Unicode è un sistema progettato per rappresentare ogni simbolo/carattere di ogni lingua. Rappresenta qualsiasi lettera, carattere o ideogramma come un numero a 4 byte (con UTF-32), consentendo quindi di rappresentare tutte le lingue esistenti. Ogni simbolo è rappresentato da un solo numero e ogni numero è rappresenta un solo simbolo.

```
>>> '\u0041' # rappresentazione esadecimale
'A'
>>> chr(65)  # rappresentazione decimale
'A'
```

Es: 'A' corrisponde sempre al punto Unicode U+0041

In Python3 le stringhe sono sequenze di caratteri UNICODE e questo permette di gestire qualsiasi simbolo o carattere a prescindere dalla lingua utilizzata.

python - stringhe, byte, unicode -

Tuttavia appare immediatamente evidente che l'utilizzo di 4 byte a carattere risulta molto dispendioso in termini di memoria quando in realtà si utilizzano tipicamente pochi caratteri in una stringa (normalmente facenti parte di una stessa lingua). E in realtà raramente si utilizzano caratteri che vanno oltre i primi 65535 (i primi 2 byte).

In realtà la maggior parte del testo contiene TANTI caratteri ASCII. Anche una pagina web (scritta per esempio in cinese) contiene comunque spazi, tag HTML, CSS e Javascript. Per questo si è creata la codifica UTF-8, una codifica Unicode a lunghezza variabile ottimizzata per la rappresentazione dei caratteri ASCII.

UTF-8 è la codifica utilizzata di default da Python3 per gestire i file (sia sorgenti, sia gestione file in generale).

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - stringhe, byte, unicode -

Python implementa due funzioni fondamentali per la codifica e decodifica di stringhe e decodifica di byte:

encode() → Restituisce una versione codificata della stringa come oggetto byte.

```
>>> '蛇'.encode('utf-8')
b'\xe8\x9b\x87'
```

decode() → Restituisce una stringa decodificata da un oggetto byte.

```
>>> b'\xe8\x9b\x87'.decode()
'蛇'
```

classi

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - classi -

Come già specificato, tutto in Python è un oggetto. Le classi forniscono un mezzo per raggruppare dati (variabili di classe) e funzionalità (metodi di classe) insieme (attributi di classe). La creazione di una nuova classe crea un nuovo tipo di oggetto consentendo di creare nuove istanze di quell'oggetto. Le istanze di classe possono anche avere metodi (definiti dalla sua classe) per modificare il suo stato.

Per convenzione, il nome della classe ha la prima lettera maiuscola.

```
>>> class Test:
...
...     a = "variabile di classe"
...
...     def p(self):
...             print('metodo di classe')
...
```

#1 Istanza classe Test()

#2 Accesso alla variabile di classe

#3 Accesso al metodo di classe

```
>>> t = Test() #1
>>>
>>> t.a #2
'variabile di classe'
>>>
>>> t.p() #3
metodo di classe
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - classi -

Python offre attributi e metodi speciali relativi alle classi (questi sono riconoscibili perché preceduti e seguiti da due "underscore" detti "dunder"). L'operazione di istanziazione crea un oggetto vuoto. Con il metodo speciale __init__ (simile a un costruttore di C++) è possibile creare oggetti con istanze personalizzate per definire uno stato iniziale specifico.

```
class Persona:

    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def presentazione(self):
        printf(f"Ciao, mi chiamo {self.nome} {self.cognome}")
```

self → il primo parametro di ogni metodo di classe. Parola convenzionale che è un riferimento all'istanza corrente della classe.(non va specificato quando si utilizza un metodo)

Due istanze della stessa classe con argument diversi.

```
In [9]: francesco = Persona('Francesco', 'Faenza')
        francesco.presentazione()

        Ciao, mi chiamo Francesco Faenza

In [10]: gianni = Persona('Gianni', 'Rossi')
         gianni.presentazione()

         Ciao, mi chiamo Gianni Rossi
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - classi -

In Python le classi supportano ereditarietà (inheritance). La classe che eredita si chiama Sub Class, quella che fa ereditare Super Class.

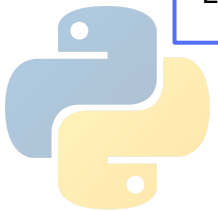
Inheritance:
Prendiamo la classe precedentemente creata Persona" e definiamo una nuova classe "Lavoratore" che eredita da "Persona"

#1 Perché la classe "Lavoratore" eriditi il metodo __init__ dalla classe genitore, è necessario specificarlo nel __init__ della classe che eredita.
#2 Definizione metodo di "Lavoratore".

```
class Lavoratore(Persona):

        def __init__(self, nome, cognome, azienda):
            Persona.__init__(self, nome, cognome) #1
            self.azienda = azienda

        def presentazione_azienda(self): #2
            printf(f"Lavor per {self.azienda}")
```



UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - classi -

#1 Istanza della classe Lavoratore

#2 Lavoratore ha ereditato il metodo __init__ di persona ...

#3 … e i suoi metodi

#4 Variabile e metodo di lavoratore

```
In [23]: cosimo = Lavoratore('Cosimo', 'Giannini', 'Tipo S.r.l') #1
         cosimo
Out[23]: Oggetto Persona <Giannini Cosimo>

In [13]: cosimo.nome #2
Out[13]: 'Cosimo'

In [14]: cosimo.cognome #2
Out[14]: 'Giannini'

In [16]: cosimo.azienda #4
Out[16]: 'Tipo S.r.l'

In [17]: cosimo.presentazione() #3
         Ciao, mi chiamo Cosimo Giannini

In [24]: cosimo.presentazione_azienda() #4
         Lavor per Tipo S.r.l
```

python - classi -

Python supporta anche l'ereditarietà multipla (multiple-inheritance).

Definiamo due classi, ciascuna con un metodo __init__ e una funzione di classe → queste saranno Super Classi di una Sub Classe.

```
class Padre:
    def __init__(self, nome):
        self.nome_padre = nome

    def mostra_padre(self):
        print(f'Padre: {self.nome_padre}')

class Madre:
    def __init__(self, nome):
        self.nome_madre = nome

    def mostra_madre(self):
        print(f'Madre: {self.nome_madre}')
```

python - classi -

Ora definiamo una classe “Figlio” che erediterà da “Padre” e “Madre”

#1 *Figlio* eredita da *Padre* e *Madre*

#2 Metodo __init__ di *Figlio* che chiama __init__ di *Padre* e di *Madre*

#3 Metodo di *Figlio* che mostra i componenti della famiglia chiamando metodi di *Padre* e *Madre*.

```
class Figlio(Padre, Madre): #1
    def __init__(self, nome, nome_padre, nome_madre):
        Padre.__init__(self, nome_padre) #2
        Madre.__init__(self, nome_madre) #2
        self.nome_figlio = nome

    def mostra_famiglia(self): #3
        print(f'Questa è la mia famiglia: ')
        self.mostra_padre() #3
        self.mostra_madre() #3
        print(f'Figlio: {self.nome_figlio}')
```

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

python - classi -

#1 Istanza classe *Figlio* assegnata all'identificatore “luca”

#2 Chiamata di attributi delle Super Classi

#3 Chiamata del metodo mostra_famiglia() definito nella Sub Classe *Figlio* che invoca al suo interno metodi dell Super Classi.

```
In [27]: luca = Figlio('Carlo', 'Gabriele', 'Elisabetta') #1

In [29]: luca.nome_padre
Out[29]: 'Gabriele'

In [30]: luca.nome_madre
Out[30]: 'Elisabetta'

In [31]: luca.mostra_padre() #2
         Padre: Gabriele

In [32]: luca.mostra_madre() #2
         Madre: Elisabetta

In [33]: luca.mostra_famiglia() #3
         Questa è la mia famiglia:
         Padre: Gabriele
         Madre: Elisabetta
         Figlio: Carlo
```