Generatori

Generatori

Supponiamo di voler iterare su una sequenza crescente di interi

In [1]:
```
for i in [0, 1, 2, 3]:
    print(i)
```

```
0
1
2
3
```

- Per poter iterare sulla sequenza, siamo prima costretti a costruirla
- ...Questo richiede tempo e (soprattuto) memoria

È inefficiente: sarebbe meglio "generare" i numeri uno alla volta, quando servono

In Python, si può fare utilizzando un generatore (generator expression)

...Purché la sequenza su cui iterare rientri in alcuni casi particolari

- Generatori e collezioni a volte vengono chiamati "iterabili"


Generatore `range`

Il generatore `range` permette di iterare su una sequenza di interi

In [2]:
```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

Se invocato con `range(n)`, il generatore:

- Restituisce progressivamente i numeri da `0` a `n-1`

Generatore `range`

Il generatore `range` permette di iterare su una sequenza di interi

In [3]:
```
for i in range(1, 4):
    print(i)
```

```
1
2
3
```

Se invocato con `range(n1, n2)`, il generatore:

- Restituisce progressivamente i numeri da `n1` a `n2-1`

Generatore `range`

Il generatore `range` permette di iterare su una sequenza di interi

In [4]:
```
for i in range(0, 4, 2):
    print(i)
```

```
0
2
```

Se invocato con `range(n1, n2, step)`, il generatore:

- Restituisce progressivamente i numeri da `n1` a `n2-1`
- ...Con passo `step` (un po' come per gli slice)


Una Parentesi: Help in Linea

`range` è un esempio di funzione che può essere chiamata in più modi

In Python ce ne sono diverse!

- Se avete dubbi su come usarle potete cercare su Google
- ...Oppure usare il sistema di aiuto di Jupyter

Potete attivarlo in (almeno) due modi:

- Aggiungere `??` dopo il nome della funzione, quindi "eseguirla"

```
In [5]: range??
```

- Usare la funzione `help()`:

```
In [6]: help('range')
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
```

Generatore `enumerate`

Il generatore `enumerate` in Python permette di iterare su una collezione

...E disporre contemporaneamente dell'indice dell'elemento corrente

In [7]:
```
nomi = ['Marco', 'Lucia', 'Gianni']
voti = [28, 30, 25]
for i, nome in enumerate(nomi):
    print(f'Voto di {nome}: {voti[i]}')
```

```
Voto di Marco: 28
Voto di Lucia: 30
Voto di Gianni: 25
```

Invocato con `enumerate(<iterabile>)`, il generatore:

- Restituisce progressivamente delle tuple
- Il primo di ogni elemento di ogni tuple è l'indice, il secondo il valore

Di solito si usa tuple unpacking per separarli


Generatore `zip`

Il generatore `zip` in Python

...permette di iterare su due collezioni contemporaneamente

In [8]:
```
nomi = ['Marco', 'Lucia', 'Gianni']
voti = [28, 30, 25]
for nome, voto in zip(nomi, voti):
    print(f'Voto di {nome}: {voto}')
```

```
Voto di Marco: 28
Voto di Lucia: 30
Voto di Gianni: 25
```

Invocato con `zip(<iterabile>, <iterabile>, ...)`, il generatore:

- Restituisce progressivamente delle tuple
- Il primo di elemento viene della prima collezione, il secondo dalla seconda, etc.

Di solito si usa tuple unpacking per separarli


Comprehension

List Comprehension

Python fornisce una notazione compatta

...Per ottenere una collezione a partire da un altro iterabile

- La notazione si chiama comprehension e ve ne sono diversi tipi
- In particolare: list/tuple/set/dictionary comprehension

Per le list comprehension, la sintassi base è:

```
[<espr.> for <var> in <iterabile>]
```

- Costruisce una lista facendo iterare `<var>` su `<iterabile>`
- ...E valutando `<espr.>` ad ogni passo


List Comprehension

Vediamo qualche esempio

Lista delle potenze di 2, da 2^0 a 2^{10}:

```
In [9]: [2**n for n in range(11)]

Out[9]: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

- La variabile `n` itera su `range(11)`
- Ad ogni iterazione, nella lista risulato si inserisce `2**n`

Un altro esempio: lista dei quadrati dei naturali da 1 a 9:

```
In [10]: [n*n for n in range(1, 10)]

Out[10]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List Comprehension

Si può specificare una condizione perché l'espressione sia valutata

La sintassi in questo caso è:

```
[<espr.> for <var> in <iterabile> if <condizione>]
```

Vediamo un paio di esempi

Quadrati nei numeri pari fino a 10:

```
In [11]: [n*n for n in range(11) if n % 2 == 0]

Out[11]: [0, 4, 16, 36, 64, 100]
```

Lista dei valori negativi da una lista preesistente:

```
In [12]: l = [-2, 0, 4, -7, -1, 10, 21]
         [v for v in l if v < 0]

Out[12]: [-2, -7, -1]
```

Set e Dictionary Comprehension

Si usa lo stesso approccio per gli insiemi

...Semplicemente si usano le parentesi graffe:

```
In [13]: {2*v for v in range(10)}

Out[13]: {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
```

Nel caso dei dizionari, si devono specificare sia la chiave che il valore

Nella sintassi, si sostituisce `<expr>` con `<chiave>:<expr>`:

```
In [14]: {n: n**2 for n in range(6)}

Out[14]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

- Si usano le parentesi graffe, come per gli insiemi
- Ma si riconoscono per la presenza sia della chiave che del valore

Tuple Comprehension

Il caso delle tuple comprehension è un filo più complesso

La sintassi:

```
(<espr.> for <var> in <iterabile> if <condizione>])
```

...Non definisce una tuple comprehension, ma un generatore:

```
In [15]: (2*n for n in range(11))

Out[15]: <generator object <genexpr> at 0x111888040>
```

- In questo modo si ottengono gli stessi benefici di una comprehension
- ...Ma con un minore utilizzo di memoria


Tuple Comprehension

Se è necessario ottenere una collezione

...Si può passare il generatore come argomento al costruttore di tuple:

```
In [16]: tuple((2*n for n in range(11)))

Out[16]: (0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

- In generale, se una funzione può accettare come argomento una collezione
- ...Si può passare come argomento anche un generatore

In questo caso le parentesi interne possono essere omesse:

```
In [17]: tuple(2*n for n in range(11))

Out[17]: (0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```