Lezione 05: Closures, Early vs Late Binding, Modificatori di scope global e nonglobal, Generatori

Closures (Chiusure)

Tecnica che permette di *racchiudere* (*catturare*) un valore all'interno di una funzione a tempo di definizione *(non sempre vero, come vedremo)*, accedendo ad uno **scope esterno**. Questo valore sarà poi utilizzabile all'interno della funzione, anche se lo scope esterno non è più attivo.

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n # cattura il valore di n
    return multiplier

double = make_multiplier_of(2)
triple = make_multiplier_of(3)
double(5)   # 10
double(11)  # 22
triple(5)   # 15
triple(11)  # 33
```

Early vs Late Binding

In realtà, non sempre il valore viene catturato al momento della definizione della closure. Esistono due modalità di binding:

- *early binding*: il valore viene catturato al **momento della definizione** della closure. Non cambia se lo scope esterno viene modificato successivamente. Viene utilizzato quando il valore è un parametro della funzione che genera la closure.

  ```
  def make(n):
      def f(x):
          return n * x
      return f

  n = 1
  f = make(n)
  n = 2
  print(f(5)) # 5
  ```

- *late binding*: al momento della definizione, viene catturata solo una **reference al valore**, che viene risolta solo al **momento dell'esecuzione** della closure. Se lo scope esterno cambia, la closure riflette questo cambiamento. Avviene quando il valore è global o nonlocal, oppure se è definito localmente alla funzione che genera la closure.

  ```
  def make():
      def f(x):
          return n * x
      return f

  n = 1
  f = make()
  n = 2
  print(f(5)) # 10

  def make():
      n = 1
      def f(x):
          return n * x
      n = 2
      return f
  ```

```
    f = make()
    print(f(5)) # 10
```

Modificatori di scope: `global` e `nonlocal`

Le variabili globali di scope esterni sono *accessibili* anche internamente alle funzioni, ma la modifica **non** si riflette all'esterno (a meno l'oggetto esterno non sia mutabile ed esponga metodi mutazionali, come una lista con `append`).

```
def f():
    x = 20      # non influisce su x globale
    y.append(4) # modifica y globale

x = 10
y = [1,2,3]
f()
print(x) # 10
print(y) # [1, 2, 3, 4]
```

Per andare a modificare una variabile globale all'interno di una funzione, si deve utilizzare la parola chiave `global`:

```
def f():
    global x  # vogliamo usare la variabile globale x
    x = 20    # modifica la variabile globale x

x = 10
f()
print(x) # 20
```

Allo stesso modo, per riferirsi ad una variabile di uno scopo esterno (ma non globale), si può utilizzare `nonlocal`:

```
def outer():
    x = 10

    def inner():
        nonlocal x  # vogliamo usare la variabile di scope esterno x
        x = 20      # modifica la variabile di scope esterno x

    inner()
    print(x) # 20

x = 5
outer()
print(x) # 5
```

Generatori

Funzioni che generano una sequenza di valori, **uno alla volta** (lazy). Vengono definiti come normali funzioni, ma utilizzano la parola chiave `yield` per restituire un valore e *sospendere* l'esecuzione.

I generatori possono venir *consumati* automaticamente dai cicli, dalla parola chiave `in`, trasformati a lista, oppure manualmente con il metodo `next()`.

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
        yield result
```

```python
gen = factorial(7)
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 6

# solo i valori non ancora generati vengono calcolati
print(list(gen))  # [24, 120, 720, 5040]

for f in factorial(5): # definito nuovo generatore
    print(f)  # 1, 2, 6, 24, 120
```

Per fare ciò, sottobanco i generatori sono degli **iteratori**.

Più avanti, vedremo come definire degli iteratori *personalizzati*, in modo da poter implementare ottimizzazioni come il *caching* dei valori già calcolati (utile ad esempio quando si sta leggendo un file esponendo un generatore).