Lezione 02: Tipi di dato, Ricorsione, Tail Recursion

Tipi di dato

Ogni *valore* (e quindi ogni variabile) ha un *tipo* (una *classe*), ma non è necessario dichiararlo esplicitamente (viene dedotto automaticamente e salvato internamente).

Si può accedere al tipo di un valore tramite la funzione `type()`, che restituisce un oggetto di tipo `type` (ovvero una classe).

```
type(2) # <class 'int'>
type(2.0) # <class 'float'>
type("ciao") # <class 'str'>
type([1, 2, 3]) # <class 'list'>
```

Si può verificare se un valore è di un certo tipo tramite la funzione `isinstance()`, che restituisce `True` o `False`.

```
isinstance(2, int) # True
isinstance(2.0, float) # True
isinstance(2, float) # False
isinstance(2.0, int) # False
```

Caratteristiche da sottolineare

Molte cose "ovvie" o "note" sono ovviamente omesse.

Booleani `True` e `False` (lettera maiuscola).

Operatori logici "a parola intera": `and`, `or`, `not`.

Operatori bitwise "a simbolo": `&`, `|`, `^`, `~`.

È possibile concatenare le condizioni: `0 < x < 5` (equivalente a `0 < x and x < 5`).

Numeri Gli interi sono "infiniti" (ovvero non hanno un limite di grandezza, ma solo di memoria disponibile).

I float sono precisi a circa 15 cifre decimali e contengono l'infinito `float('inf')`, il negativo infinito `float('-inf')` e il Not a Number, `float('nan')`.

Operatori In Python 2, l'operatore `/` esegue la divisione intera se entrambi gli operandi sono interi, altrimenti esegue la divisione float. In Python 3, l'operatore `//` esegue sempre la divisione intera, mentre l'operatore `/` esegue sempre la divisione float.

Liste Sono eterogenee (possono contenere elementi di tipi diversi).

È possibile accedere agli elementi tramite l'indice 0-based e negativi (partendo dal fondo con `lista[-1]`).

È possibile fare subslicing, anche omettendo il primo (default `0`) o l'ultimo (default `len(lista)`) estremo. Si possono anche usare indici negativi per il subslicing.

Non possono essere hashate (quindi non possono essere usate come chiavi di dizionari).

Tuple Liste immutabili, vale *quasi* tutto quello che valeva per le liste.

Lo slicing di una tupla restituisce una tupla.

Le tuple sono usate sottobanco quando vengono restituiti più valori da una funzione, tramite il *packing* e *unpacking* (assegnamento multiplo, vale anche per le liste).

```
def get_coordinates():
  return 10, 20 # packing

x, y = get_coordinates() # unpacking
```

Sono immutabili, quindi sono thread-safe e più efficienti delle liste.

Possono essere hashate (quindi usate come chiavi di dizionari).

Insiemi (Set) Sono collezioni di elementi unici, non ordinati e non indicizzati.

Non possono contenere elementi mutabili (come le liste).

È sbagliato fare supposizioni sull'ordine degli elementi.

Non possono essere hashati (quindi non possono essere usati come chiavi di dizionari), ma esistono i *frozenset*, che sono set immutabili e possono essere hashati.

Per dichiarare un set vuoto bisogna per forza usare il costruttore `set()`, perché le parentesi graffe `{}` sono riservate per i dizionari e le quadre `[]` per le liste. Ma per dichiarare un set con elementi si possono usare le parentesi graffe `{1, 2, 3}`.

Dizionari (Dict) Lista di coppie chiave valore, la chiave deve essere hashabile (ovvero immutabile, come le stringhe, i numeri e le tuple).

È sbagliato fare supposizioni sull'ordine degli elementi (anche se in realtà l'ordine è l'ordine di inserimento, ma non è garantito in tutte le versioni di Python).

Stringhe Sequenza immutabile di caratteri unicode.

È possibile fare slicing, concatenazione `+` e ripetizione `*`.

Per formattare le stringhe si possono usare le f-string `f"ciao sono {nome}"` o il metodo `"ciao sono {}".format(nome)`.

Bytes Sequenza immutabile di byte (ovvero numeri interi da 0 a 255).

Definiti dal letterale `b''` (`b'abcd\xff\x65'`) o tramite il costruttore `bytes()`.

Per modificare i byte bisogna convertirli in `bytearray`, che è una sequenza di byte mutabile.

Ricorsione

Funzione che chiama se stessa, con un caso base che termina la ricorsione. Può essere:

- *diretta*: la funzione chiama se stessa direttamente
- *indiretta*: la funzione chiama un'altra funzione che a sua volta chiama la prima

Di default il numero di step ricorsivi è limitato a `1000`, ma è possibile aumentarlo con `sys.setrecursionlimit()`.

Funzionamento della ricorsione (stack frame)

Quando viene chiamata una funzione, viene creato un *frame di chiamata* (o *stack frame*) che contiene le informazioni necessarie per eseguire la funzione, come i parametri e il valore di ritorno e viene messo sullo *stack*.

Le informazioni importanti di un frame sono:

- gli *argomenti* vengono messi in cima al frame del *chiamante*
- il *valore di ritorno* viene messo in fondo al frame del *chiamato*

Grazie a questa disposizione, è possibile accedere ai parametri e al valore di ritorno facendo un offset e uscendo dal proprio frame, senza necessità di copiarli.

Tail Recursion

La ricorsione può avere della *computazione locale*, ovvero delle operazioni che per poter essere valutate richiedono il risultato della chiamata ricorsiva. Quando sono presenti delle *locals*, tutti i frame **devono** essere mantenuti sullo stack fino a quando non si è arrivati al caso base, per poter accedere ai risultati delle chiamate precedenti.

```python
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1) # computazione locale (n * ...)
```

Spesso si può evitare di avere computazione locale, *accumulando* il risultato in un parametro della funzione. In questo modo non è più necessario mantenere i frame di tutte le chiamate ricorsive, ma solo un *unico frame*, che viene "aggiornato" sovrascrivendo i parametri ad ogni chiamata ricorsiva.

Le performance migliorano notevolmente, sia in termini di memoria che di velocità, perchè si evita di creare nuovi frame per ogni chiamata ricorsiva. Questa *ottimizzazione* deve essere supportata dal linguaggio, che altrimenti manterrà tutti i frame sullo stack. Python non supporta questa ottimizzazione di default, ma è possibile implementarla manualmente giocando con gli stack frame *(lo vedremo più avanti)*.

```python
def factorial(n, acc=1):
  if n == 0:
    return acc
  return factorial(n - 1, n * acc) # accumulazione del risultato, non ci sono locals
```

Tutte le funzioni tail recursive sono trasformabili in cicli **iterativi**.

```python
def factorial(n):
  acc = 1
  while n != 0:
    acc = n * acc
    n -= 1
  return acc
```

Non sempre è **possibile** trasformare una ricorsione normale in una tail recursion, la trasformazione è possibile solo quando il risultato della funzione può essere accumulato nei parametri. In alcuni casi, come per funzioni che devono elaborare i risultati delle chiamate ricorsive (ad esempio algoritmi che richiedono backtracking), la tail recursion non è applicabile senza cambiare radicalmente la logica della funzione.