Python

; in C, : in Python

In C, il punto e virgola `;` è usato per terminare le istruzioni. In Python i due punti `:` sono utilizzati per indicare l'inizio di un blocco di codice all'interno di strutture come funzioni, istruzioni condizionali e loop.

```
if __name__ == "__main__":
  print("hello world")
```

id(): mutabili e non

La funzione `id()` restituisce l'identificatore univoco di un oggetto, ossia un intero che rappresenta la posizione in memoria. Per oggetti immutabili (come interi, stringhe, tuple) l'id rimane lo stesso durante l'intera vita dell'oggetto. Per gli oggetti mutabili (come liste, dizionari, insiemi) l'id può cambiare se l'oggetto viene modificato in modo tale da richiedere una nuova allocazione di memoria.

```
x = [0]
print(id(x))
x.append(1)
print(id(x))
x = x + [2]
print(id(x))
```

Curiosità:

```
x = 256
y = x + 1
y = y - 1
print(x, id(x), id(y), x is y)
x = 257
y = x + 1
y = y - 1
print(x, id(x), id(y), x is y)
```

`x="ciao"` in C e `x="ciao"` in Python

ovvero scatola contro targhetta

In C, una variabile stringa può essere vista come una "scatola" che contiene direttamente il puntatore alla sequenza di caratteri. In Python, una variabile stringa è come una "targhetta" attaccata a un oggetto stringa, ovvero un riferimento ad un oggetto (come in Java).

None
non esistono funzioni che non ritornano nulla, ma funzioni che restituiscono nulla
`None` è un tipo di dato predefinito che rappresenta l'assenza di un valore o un valore nullo. Può essere usato in diversi casi:
- restituito da funzioni
- per inizializzare variabili e/o parametri
- per verificare l'esistenza di un dato

Numeri int float complex
int a precisione illimitata, float IEEE double
Gli interi `int` rappresentano numeri interi senza parte decimale. Possono essere positivi o negativi e non hanno limiti di grandezza. I numeri in virgola mobile `float` rappresentano numeri reali con una parte frazionaria. I numeri complessi `complex` sono rappresentati da una parte reale e una parte immaginaria. Vengono espressi come a + bj, dove a è la parte reale e b è la parte immaginaria, e j è l'unità immaginaria (√-1).

Tipi iterabili: string tuple set list dict
I tipi iterabili sono oggetti che possono essere iterati elemento per elemento, anche senza usare gli indici (tramite il for). Quelli più comuni sono:

- `String` : sequenze immutabili di caratteri. Non possono essere modificate dopo la creazione. Si può accedere ai caratteri tramite indici e ottenere sottostringhe con slicing.
  "ciao" = stringa, "" = stringa vuota.
- `Tuple` : sequenze ordinate e immutabili di elementi. Non possono essere modificate dopo la creazione. Si può accedere agli elementi tramite indici e ottenere sottotuple con slicing. Possono contenere elementi di diversi tipi. Il simbolo che identifica una tupla è la virgola, non la parentesi tonda.
  (1,2,3) = tupla, (1,) = tupla di un elemento, () = tupla vuota.
- `Set` : collezioni non ordinate di elementi unici. Ogni elemento può apparire solo una volta. Gli elementi non hanno un ordine definito. Supportano operazioni matematiche come unione (|), intersezione (&), differenza (-), differenza simmetrica (^). Sono identificati da parentesi graffe.
  {1,2,3} = insieme, set() = insieme vuoto.
- `List` : sequenze ordinate e mutabili di elementi. Possono essere modificate dopo la creazione. Si può accedere agli elementi tramite indici e ottenere sottoliste con slicing. Possono contenere elementi di tipi diversi. Sono identificate da parentesi quadre (non sono array).
  [1,2,3] = lista, [] = lista vuota.
- `Dictionary` : collezioni non ordinate di coppie chiave-valore. Ogni chiave deve essere unica e immutabile (no list, set o dict). Gli elementi possono essere rapidamente recuperati tramite le chiavi. Possono contenere elementi di diversi tipi. Sono identificati da parentesi graffe + due punti.
  {1:10, 2:20, 3:30} = dizionario, {} = dizionario vuoto.

Assegnamenti tra iterabili
Permettono di scomporre una sequenza di valori e assegnare a variabili multiple in un'unica operazione.
L'assegnamento base implica l'assegnazione di ogni elemento dell'iterabile a una variabile corrispondente. `a, b, c = (1,2,3)`
L'unpacking permette di assegnare i valori di un iterabile a variabili in modo più flessibile.
`a, b, *resto = [1,2,3,4,5]`
Si può usare l'underscore come variabile per indicare che un valore non interessa.
`a, _, c = (1,2,3)`
È importante notare che il numero di variabili sul lato sinistro dell'assegnamento deve corrispondere al numero di elementi nell'iterabile, a meno che non si stia utilizzando `*` per catturare valori multipli.

Loop: for, while (con caso else)
Ciclo `while` funziona come negli altri linguaggi, ciclo `for` si usa per iterare su una sequenza (lista, tupla ...). All'interno del corpo si può usare `break` per terminare il ciclo oppure `continue` per passare all'iterazione successiva. Entrambi i costrutti possono avere un caso `else` associato, che viene eseguito se il ciclo termina non a causa di un break.

```
for var in seq:
  # corpo del ciclo
  ...
else:
  # clausola else

while condizione:
  # corpo del ciclo
  ...
else:
  # clausola else
```

pass
Istruzione che non fa nulla, usata come segnaposto. L'uso più costruttivo è nel python ad oggetti per distinguere una sottoclasse dalla sua classe padre anche se identiche (nel corpo si mette `pass`).

with
Gestisce il contesto di esecuzione di un blocco di codice, dove gli assegnamenti sono temporanei al corpo del `with`. Usandolo con i file, quando viene eliminato l'assegnamento che apre il file, esso viene automaticamente chiuso.

```
with open(filename, "r") as f:
  x = f.read()
print(x)
```

Espressione if

Operatore `if` classico:

```
if condizione:
  # condizione è vera
elif altra_condizione:
  # altra_condizione è vera
else:
  # nessuna condizione precedente è vera
```

Operatore `if inline`. Equivalente al C: condition ? then : else

```
def veritas(x):
  return 'vero' if x else 'falso'
```

Argomenti di funzione

- Posizionali: binding classico, i parametri sono associati secondo l'ordine in cui sono definiti.
- Con valore di default: nella dichiarazione della funzione si specifica un valore default per il parametro, nel caso non venisse passato alla chiamata. `def f(param = default, ...)`
- Espressi per nome: al momento della chiamata si specifica a quale parametro assegnare il valore. `f(param = val, ...)`
- Funzioni variadiche con * e **: per dichiarare una funzione con un numero variabile di parametri. Con * si può accedere ai diversi parametri (tupla), con ** si accede ad un dizionario {nome_param : valore}. `def f(*param_var, **param_dict)`

Iteratori

Un iteratore è un oggetto che permette di iterare su una sequenza di elementi, eventualmente eseguendo operazioni.
- Iterable: un oggetto che può restituire un iteratore.
- Iterator: un oggetto con il metodo __next__(), che restituisce gli elementi uno alla volta.

```
class itenumerate:
  def __init__(self, seq):
    self.seq = seq
  def __iter__(self):
    self.i = 0
```

```
    return self
  def __next__(self):
    if self.i >= len(self.seq):
      raise StopIteration
    this = self.i
    self.i += 1
    return this, self.seq[this]

for num, val in itenumerate(['a','b','c']):
  print(num,val)
```

Funzioni con __ hanno un significato particolare. es: __init__ è il costruttore.

Generatori

Un modo per creare iteratori usando una funzione che utilizza l'istruzione `yield` anziché `return`. Sono utili per generare una sequenza di valori uno alla volta, consentendo una gestione efficiente della memoria.

Ogni chiamata costruisce e ritorna un iteratore (funzione con metodo next utile per i loop for). Se la funzione termina emette 'StopIteration'.

```
def myenumerate(seq):   # la enumerate è built-in
  n = 0
  for item in seq:
    yield n, item
    n += 1
for num, val in myenumerate(['a','b','c']):
  print(num,val)
```

```
def fibonacci():
  i = j = 1
  while True:
    r, i, j = i, j, i + j
    yield r
for rabbits in fibonacci():
  print(rabbits, '', end='')
  if rabbits > 100: break
print()
```

Decoratori

Una funzionalità che permette di modificare il comportamento delle funzioni o delle classi, senza

intaccare il codice. Utilizzano la sintassi `@decoratore` prima della definizione di una funzione o classe.

```
@<decorator>
def <name> etc, etc
```

è come

```
def <name> etc, etc
<name> = <decorator>(<name>)
```

trace.py:

```
def trace(f):
  f.indent = 0
  def strtuple(x):
    return "("+str(x[0])+")" if len(x)==1 else str(x)
  def g(*x):
    print('| ' * f.indent + '/-- ', f.__name__, strtuple(x), sep='')
    f.indent += 1
    value = f(*x)
    f.indent -= 1
    print('| ' * f.indent + '\-- ', 'return', repr(value))
    return value
  return g

def memoize(f):
  cache = {}
  def g(*x):
    if x not in cache:
      cache[x] = f(*x)
    return cache[x]
  return g
```

main.py:

```
#!/usr/bin/env python3
import sys
import trace

# try to uncomment the following statements:
#@trace.trace
#@trace.memoize
def fib(i):
  return 0 if i<=0 else 1 if i==1 else fib(i-1)+fib(i-2)

if __name__=="__main__":
  for i in range(int(sys.argv[1])):
    print(i,fib(i))
```

Overload degli operatori

Permette di definire o modificare il comportamento degli operatori standard, implementando metodi speciali con nomi predefiniti (iniziano e finiscono con il doppio underscore).

```
__new__ __init__ __del__    # init/final.
__repr__ __str__ __int__    # conversions
__lt__ __gt__ __eq__ ...    # comparisons
__add__ __sub__ __mul__ ... # arithmetic
__call__ __hash__ __nonzero_ ...
__getattr__ __setattr__ __delattr__
__getitem__ __setitem__ __delitem__
__len__ __iter__ __contains__
```

Scrivere librerie: moduli e package

Esempio1: modulo, file come libreria

m.py:

```
#!/usr/bin/env python3
import lib

if __name__ == "__main__":
  lib.hw()
```

lib.py:

```
#!/usr/bin/env python3
def hw():
  print("hello world")

if __name__ == "__main__":
  print("test code")
```

Esempio2: package

m.py:

```
#!/usr/bin/env python3
import lib

if __name__ == "__main__":
  lib.hw()
```

lib/__init__.py:

```
#!/usr/bin/env python3
from .hw import hw
x=42
```

lib/hw.py:

```
#!/usr/bin/env python3
def hw():
  print("hello world")

if __name__ == "__main__":
  print("test code")
```