Appunti di Programmazione Avanzata

Caccaro Sebastiano
A.A.2019/2020

Contents

7 Ottobre 2019

1 Informazioni sul corso

- **Login:**pa
- **Password:**PA+#2009#
- **Sito del corso:** `cazzola.di.unimi.it/pa.html`

Libri e cose del genere sono presenti nelle slide. L'esame è scritto e si svolge al computer.

2 Python in breve

È un linguaggio di scripting multi-paradigma, quindi imperativo, object-oriented, e funzionale. È interpretato, **object-based** (=ogni cosa è un oggetto) e tipizzato dinamicamente.
Nel corso si userà **Python 3**.

2.1 Osservazioni su humanize.py

- Le variabili non hanno un tipo statico. Il tipo dinamico è attaccato all'oggetto che contiene la variabile.
- Supporta alcune strutture dati base di default. Ad esempio i dizionari (HashMap), liste ecc.
- Si può indentare con i tab o con gli spazi, molto meglio con gli spazi.
- Dopo la prima linea della funzione se commento con ‘‘‘ `commento` ‘‘‘, posso fare un commento tipo javadoc.
- Come in javascript, tutto è un oggetto, anche le funzioni. Anche i primitivi sono degli oggetti. Come in JS, posso passare le funzioni come parametri.
- `__name__` assume il nome del file se viene importato come libreria, e assume il valore `__main__`

- Ci sono le **format string** che sono un po' come le template string di Javascript e PHP. Dalla 3.5 in poi non serve nemmeno il .format.
- Solo i tipi primitivi sono passati per valore, i tipi "complessi" sono passati per valore, quindi il contenuto è passato per riferimento.
- Qualsiasi riga di codice non protetta da if o dentro funzioni verrà eseguita. Non esiste un main.

2.2 Altre Caratteristiche

- La keyword import mi permette di importare da altri moduli. Se voglio accedere a una funzione o campo dati specifico, faccio modulo.funzione. Posso accedere alla doc di un oggetto con `oggetto.__doc__`.

2.3 Eccezioni

Per lanciare un'eccezione si usa la keyworw `raise`. Si usano dei blocchi `try - except`.

7 Ottobre 2019

2.4 Tipi di dato in Python

I tipi di Python funzionano come in javascript, dove il tipo è associato all'oggetto. Ci sono vari tipi numerici. Anche i primitivi sono oggetti, in quanto istanze di classi.

2.4.1 Numeri

Si usa, ad esempio la classe `int`. I primi numeri di Python, circa i primi 256, sono implementati tramite **singleton**.
Alcuni comandi:

- `type()` mi torna il tipo di una variabile
- `isinstance()` controlla il tipo di una variabile.

Posso rappresentare qualsiasi intero fino a infinito. I float sono precisi fino a 15 cifre decimali. Ci sono vari operatori, DIP pg57.

2.4.2 Liste

Sono trattate come degli array. Come javascript, posso avere più tipi di dato all'interno della stessa lista. Posso anche indicizzare al contrario. Esempio: `[-2]` accede al penultimo elemento della lista.
L'operatore di **slicing** è il seguente `[x:y]` dove `x` è mantenuto e `y` escluso. Posso sempre usare gli indici negativi. Se ometto uno dei due indici parto dall'inizio/fine. Creo sempre un nuovo oggetto.
Operatori: DIP pg64

- `+`: Somma fra liste, creo un nuovo oggetto.
- `append`: appende un elemento alla fine della lista
- `extend:` aggiunge una lista ad un'altra lista
- `insert:` inserisce un elemento in una certa posizione nella lista
- `in`: del tipo `if x in list`, torna true e false
- `count` conta le istanze, in caso non ci sono, lancio eccezione
- `del`: `del list[2]` rimuovo per posizione
- `value`: `list.remove(3.13)` rimuove per valore

2.4.3 Tuple

Sono delle liste immutabili. Si scrivono con le tonde, non con le graffe.
Siccome non possono cambiare:

- Sono più efficienti
- Posso usarle come chiave in un dizionario

Posso usare più o meno le stesse operazioni sulle liste, apparte quelle che apportano modifiche. Posso usare per assegnazioni multiple a variabili.

2.4.4 Sets (insiemi)

Struttura dati non ordinata con elementi univoci. Le creo come una lista ma con le graffe. Si crea un set vuoto con `set()`. Posso anche crearli da una lista. Operatori:

- `set.add` se possibile aggiunge un elemento al set

- `set1.update(set2)` unisce due set nel primo set
- `move discard`
- `union, difference ecc.` Tutte le operazioni matematiche sugli insiemi sono implementati. Potrebbe essere necessario ridefinire l'uguaglianza.

2.4.5 Dizionari

Insieme di coppie chiave-valore. La sintassi è praticamente quella del JSON. Sono praticamente delle HashMap. Il dizionario vuoto si scrive con . Sono disponibili i classici metodi che mi aspetterei.

2.4.6 Stringe

Si comportano come le liste. Uso len per la lunghezza. Posso delimitare le stringe con ", ' e ‘‘‘.
Stringhe di formattazione (DIP pg114), meglio che guardo la doc. È simile a js e php, ma tipo printf del C. Esiste anche il comando Template. Ci sono un'infinità di operazioni.

2.5 Ricorsione

La ricorsione in Python non è ricorsiva in coda. Quindi insomma lol, potrebbe dare qualche problema. L'iterazione è più efficiente. In Python ho massimo 1000 ricorsioni. Posso aggirare il problema, ma la ricorsione in Python **fa cagare**.

14 Ottobre 2019

2.6 Comprehension

Tramite una **comprehension** diamo una descrizione comprensiva di una lista di dati. È come dare una formula matematica che descrive un insieme e una proprietà. È una sorta di map, se vogliamo:

```
>>> {elem:elem**2 for elem in range(1,10)}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Posso usarlo per creare liste, dizionari, tuple ecc. È da preferire rispetto ad usare un map, perchè è molto efficiente.
Posso utilizzare la comprehension anche per fare cose tipo un filter:

Listing 1: Espressione che crea tutti i quadrati perfetti da uno a cento

```
>>> [elem for elem in range(1,100) if (int(elem**.5))**2 == elem]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Posso usarlo anche per mergiare set diversi, e fare varie figate. Ad esempio posso fare un prodotto cartesiano (doppio for) in modo figo.

```
>>> {(x,y) for x in range(3) for y in range(5)}
{(0,1),(1,2),(0,0),(2,2),(1,1),(1,4),(0,2),(2,0),(1,3),
(2,3),(2,1),(0,4),(2,4),(0,3),(1,0)}
```

Lo svantaggio/problematica di questo metodo è che non mi da modi (facili) per interrompere l'esecuzione di questo finto for.

21 Ottobre 2019

3 Programmazione funzionale

È un modo di programmare che scoraggia l'uso dello stato, e dei side effect. L'oggetto primario è la funzione, e il flusso è regolato tramite la **ricorsione**.

3.1 Motivi

Dovrebbe essere più simile al tipo di definizione matematiche a cui dovremmo essere abituati. Il vero motivo è che programmando funzionalmente dovrei commettere **meno errori**. Questo perchè avere uno stato, e in particolar modo l'assegnazione di variabili, è una delle principali fonti di errori. Oltre a ciò, in un linguaggio di programmazione funzionale è più facile individuare gli errori commessi.

3.2 Istruzioni a supporto

Le istruzioni cardine della programmazione funzionale sono il `map()`, il `filter()` e il `reduce()`. Il map e il filter non ritorano direttamente una lista, ma un iterabile. Se dopo voglio una lista, devo mettere questo iterabile dentro il costruttore della lista. Il reduce ha peculiarità che il primo elemento della lista di partenza viene usato come accumulatore. Volendo c'è anche un parametro opzionale per l'accumulatore.

3.3 Condizionali

Nella programmazione funzionale pura non è possibile usare il construtto `if`. Uso quindi lo **short-circuiting**.

```
return (x==1 and 'one') or (x==2 and 'two') or 'other'
```

Figure 1: Esempio di short circuiting

Lo short circuiting funziona praticamente come in javascript.
Vado bene a usarlo con le lambda.

```
cond = \
 lambda x: \
  (x==1 and block("one")) or (x==2 and block("two")) or (block("other"
```

Figure 2: Esempio di lambda

3.4 Codice sequenziale

Nella programmazione funzionale, scrivere il codice in modo sequenziale (aka imperativo) non è accettabile.
Posso fare una cosa del genere:

```
# let's create an execution utility function
do_it = lambda f: f()

# let f1, f2, f3 (etc) be functions that perform actions
map(do_it, [f1,f2,f3])
# map()-based action sequence
```

3.5 Funzioni monadiche

Sono funzioni che causano un side effect ma che comunque ritornano comunque qualcosa.

```
# Funzione Imperativa
def echo_IMP():
        while True:
                x = input("FP -- ")
                if x == 'quit': break
                else: print(x)

if __name__ == "__main__": echo_IMP()
```

```
#Funzione Monadica Funzionale
echo_FP = \
lambda: monadic_print(input("FP -- "))=='quit' or echo_FP()
if __name__ == "__main__": echo_FP()
```

22 Ottobre 2019

4 Clousures

È solo sostanzialmente un modo come un altro per dire che posso passare le funzioni come parametro e fare alcune figate.

5 Generatori

Un generatore è una funzione che può tornare più volte un valore, tramite il costrutto `Yield`, e si avanza la funzione tramite la keyword `next`. Posso usare i generatori anche nei for, e in tal caso non serve che chiamo il next.

```
def make_counter(x):
        print('entering make_counter')
        while True:
                yield x
                print('incrementing x')
                x = x + 1
...
print(next(make_counter))
```

Figure 3: Esempio di generatore che funge da contatore

6 Tipizzazione Dinamica

Python funziona un po'come Javascript: Quando creo una variabile, essa non ha tipo, ma è un semplice riferimento ad un oggetto. È l'oggetto che contiene il tipo di una variabile. Posso quindi pensare alla variabili come a dei semplici puntatori a oggetti. Come Java e Javascript, python è **garbage collected**.
Per controllare l'uguaglianza fra due variabili esistono due operatori:

- `==` controlla l'uguaglianza di valore di due oggetti. Ex `A == B`
- `is` controlla l'uguaglianza di reference fra due oggetti. Ex `A is B`. Attenzione che siccome interi e altri piccoli oggetti vengono cachati, `is` in questi casi restituisce sempre `True`

In python il passaggio di parametri a funzione avviene sempre **per riferimento**. Fanno eccezione alcuni tipi immutabili, come interni, stringhe, e tuple, che vengono passate per valore.