UNIVERSITÀ di VERONA

Linguaggio di Programmazione Python

11/04/2022

Giulio Mazzi

giulio.mazzi@univr.it

Dizionari (1/3)

- Le liste sono strutture dato utili, ma poco flessibili:
  - Dobbiamo usare numeri per indicizzare
  - Gli indici partono sempre da zero e sono contigui
- Per superare queste limitazioni usiamo un **dizionario:**
  - Mappa ogni chiave a un valore
  - Può usare qualsiasi tipo immutabile come chiave
  - Efficiente (hash table)
  - Mutabile (come le liste)

```
ita2eng = dict()
ita2eng['ciao'] = 'hello'
ita2eng['gatto'] = 'cat'

print(ita2eng)
# stampa {'ciao': 'hello', 'gatto': 'cat'}
```

Dizionari (2/3)

- Posso creare un dizionario vuoto usando `dict()` oppure `{}`
- Gli elementi dentro un dizionario vengono chiamate **coppie chiave-valore**
- Posso accedere a un valore usando la chiave tra parentesi quadre

```
print(ita2eng['gatto'])
# stampa cat
```

- Le **chiavi** possono avere **tipo eterogeneo** e sono **univoche**
- La **chiave deve essere immutabile, il valore no**

```
l = [1, 2, 3]
ita2eng['uno, due, tre'] = l
# ok, da stringa a lista

ita2eng[l] = 'one, two, three'
# errore! lista non è 'hashable'
```

```
Traceback (most recent call last):
  File "prova.py", line 16, in <module>
    ita2eng[l] = 'one, two, three'
TypeError: unhashable type: 'list'
```

Dizionari (3/3)

- I dizionari forniscono un interfaccia simile a list (e.g, `in`, `len()`, `clear()`, `copy()`,…)
- Non possiamo modificare direttamente l'ordine degli elementi (`insert(val, pos)`, `sort()`). L'ordine è gestito dalla funzione hash (una funzione che facilita la ricerca delle chiavi nel dizionario).
- Possiamo usare cicli for per iterare tutti i *valori*. Se vogliamo iterare solo le chiavi, o le coppie chiave valore, usiamo rispettivamente I metodi `keys()` e `items()`.

```
for eng in ita2eng:
    # stampa parole inglesi
    print(eng)

for ita, eng in ita2eng.items():
    # stampa parola italiana e inglese
    print(ita, eng)
```

Documentazione completa: https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

Dizionari – varianti

Il modulo `collections` fornisce alcune varianti a dizionari e liste. In particolare:

- `Counter` è un dizionario che mappa degli oggetti a un numero. Permette di contare con facilità le occorrenze di alcuni elementi
- `defaultdict` è un dizionario che inizializza gli elementi che non esistono a un valore predefinito specificato dall'utente
- `OrderedDict` crea dizionari che rispettano l'ordine d'inserimento degli elementi

Per usarli devo importare il modulo `collections`

```
import collections

text = 'testo da analizzare'
conta_lettere = collections.Counter(text)
# stampa le 3 lettere più frequenti
print(conta_lettere.most_common(3))
```

Input – linea di comando

Fino a ora, abbiamo usato solo dati inseriti manualmente nel codice, ma spesso non è un buon modo di operare.

Possiamo acquisire dati da diverse fonti: linea di comando, file, internet, …

Per acquisire direttamente un valore, usiamo la funzione `input()`

```
>>> n = input('inserisci nome: ')
inserisci nome: Giulio
>>> print(n)
Giulio
>>> type(n)
<class 'str'>
```

Il valore è **sempre letto come stringa**, se voglio altro, devo convertirlo (`int()`, `float()`, `bool()`, ….)

Input – file

- Posso aprire file con `open(<nome>, <lettura>)`
  - 'r' apre in sola lettura (valore di default se ometto <lettura>)
  - 'w' apre in scrittura (cancella il contenuto, se il file esiste)
  - 'a' appende contenuto al file esistente
  - 'r+' lettura e scrittura
- Quando abbiamo finito di lavorare con il file, dobbiamo chiuderlo (`f.close()`)
- Il metodo `f.read()` ci ritorna il file come unica stringa, `f.readline()` legge una riga, `f.readlines()` torna una lista di stringhe (una stringa per riga). In alternativa, possiamo usare il for
- Possiamo usare `write()` e `writeline()` per scrivere

```
f = open('example.txt', 'r')
l = []
for line in f:
    if line != '\n':
        l.append(int(line[:-1]))

print(l)
f.close()
```

Input – with

Aprire un file e chiuderlo quando non serve più è un operazione comune, ma prona a errori.

Python ci mette a disposizione un costrutto comodo per automatizzare questa operazione, il `with`

```
with open('example.txt', 'r') as f:
    l = []
    for line in f:
        if line != '\n':
            l.append(int(line[:-1]))

print(l)
```

Attenzione: le variabili dichiarate dentro il with sopravvivono allo scopo del with stesso

Exceptions

Gli errori che avvengono durante l'esecuzione del codice vengono chiamati **eccezioni.**

In molte situazioni, per esempio quando gestiamo l'input, non possiamo essere certi che tutti i dati che leggiamo siano al 100% corretti e coerenti con le nostre specifiche.

Possiamo gestire esplicitamente questi errori con l'istruzione `try... except`

Il codice prova a eseguire le istruzioni presenti nel blocco try, se qualcosa fallisce, eseguiamo il blocco except

```
with open('example.txt', 'r') as f:
    l = []
    for line in f:
        try:
            l.append(int(line[:-1]))
        except:
            print('impossibile convertire: "' + line +'"')
            continue

print(l)
```

Exceptions - opzioni

Possiamo adattare il try... except alle nostre necessità:

- Possiamo aggiungere un blocco `finally:` alla fine. Questo viene eseguito in ogni caso (sia che ci sia un eccezione, sia che non ci sia niente)
- Possiamo aggiungere un blocco `else:`, viene eseguito solo se non ci sono eccezioni
- Possiamo specificare che errori vogliamo trattare (per gestirne di diversi, uso più di un except)

```
with open('example.txt', 'r') as f:
    l = []
    for line in f:
        try:
            n = int(line[:-1])
        except ValueError as exc:
            # gestisco solo errori di tipo ValueError
            print(exc.args)
            n = 0
        finally:
            l.append(n)
```

Sollevare un eccezione

Le eccezioni vengono generate (o sollevate, da *raise*) quando avviene un errore.

È possibile creare manualmente delle eccezioni, usando l'istruzione `raise`.

Per farlo, dobbiamo specificare il tipo di eccezione, e il contenuto del messaggio.

```
raise ValueError('impossibile convertire')
```

```
Traceback (most recent call last):
  File "prova_file.py", line 14, in <module>
    raise ValueError('impossibile convertire')
ValueError: impossibile convertire
```

Python include già molte eccezioni, potete trovare una lista completa qui:

https://docs.python.org/3/library/exceptions.html

Funzioni – parametri opzionali

Abbiamo già visto come dichiarare una funzione che richiede dei parametri (detti *argomenti*).È possibile assegnare un valore di default ad un argomento, se la funzione viene chiamata senza assegnare niente usiamo il valore di default.

Attenzione: I valori di default vengono valutati una sola volta, meglio evitare oggetti mutabili (come le liste). In questi casi, buona norma mettere None come valore di default e inizializzare il valore in seguito

```
def somma(lista, base = 0):
    for i in lista:
        base += i
    return base

# conta partendo da 0
somma([10, 20, 30])
# conta partendo da 10
somma([10, 20, 30], 10)
```

Funzioni –keyword arguments

Quando chiamiamo una funzione, è possibile esplicitare direttamente i nomi degli argomenti. In questo caso, l'ordine non conta. È molto utile per interagire con funzioni di libreria che hanno molti parametri.

```
def somma(lista, base = 0):
    for i in lista:
        base += i
    return base

# passaggio esplicito degli argomenti
somma(base=5, lista=[0, 1, 2, -2])
```

Funzioni – tipo del return

In python, il tipo di una variabile non viene specificato, ma viene dedotto dal contesto. Lo stesso vale per il risultato di una funzione: finch la funzione non esegue, non possiamo sapere quale sarà il tipo del risultato.

Il tipo di ritorno può essere eterogeneo, input diversi dello stesso tipo possono darmi risultati di tipo diverso! (si pensi alla radice quadrata, che può restituire numeri reali o complessi).

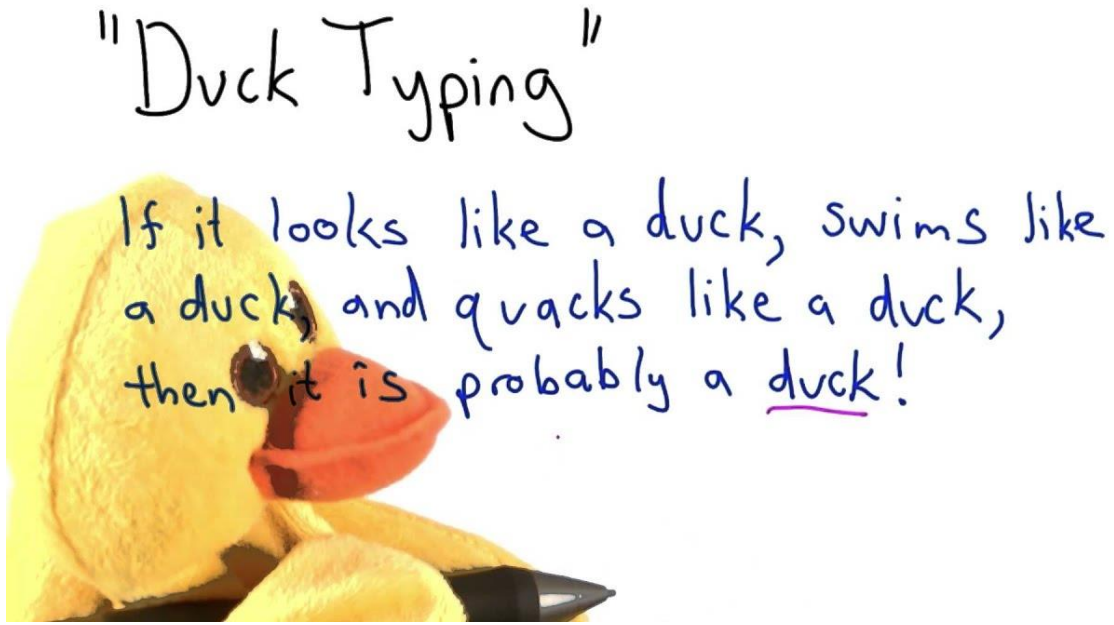
```
def my_find(lista, valore):
    # se il find fallisce, ritorna None
    try:
        return lista.find(valore)
    except AttributeError:
        return None

x = my_find([1, 2, 3, 4], 5)
```

Questa è una feature che può generare più problemi che benefici, va usata con parsimonia. Un caso d'uso valido e comune è, però, quello di ritornare None se una funzione fallisce e non riesce a costruire un risultato

Funzioni – duck typing

In luce dei nuovi elementi che abbiamo appena visto, vale la pena ricordare che python predilige il *duck typing*. In particolare, non è buona pratica controllare esplicitamente i tipi, meglio usare gli argomenti come desidero e gestire le eccezioni se le cose non vanno a buon fine!




```python
def somma(lista, base = 0):
    if not isinstance(base, int):
        raise ValueError("accetto solo interi")
    for i in lista:
        if not isinstance(i, int):
            raise ValueError("accetto solo interi")
    # ...
```

Funzioni - annotazioni

Indicare il tipo che una funzione si aspetta, tuttavia, è molto importante. Questo spesso è parte della documentazione.

Da python 3.5, possiamo fare di meglio. È possibile *annotare* argomenti e valore di ritorno di una funzione. Questo non modifica il codice in nessun modo, è solo un suggerimento, ma può essere molto utile per documentare il nostro codice. Inoltre, tool esterni, come linter e IDE, possono sfruttare queste informazioni in maniera sistematica.

```
from typing import List

def somma(lista : List[int], base : int = 0) -> int:
    for i in lista:
        base += i
    return base
```