Tratti dai lucidi di Marco Barisione

Classi in Python

OOP

- OOP = Programmazione Orientata agli Oggetti
- Incapsulamento
  - Incapsulamento di dati e codice in un unico contenitore
  - Nasconde la complessità
  - Migliora l'indipendenza
- Ereditarietà
  - Definisce le specializzazioni delle classi
  - Permette di "riciclare" il codice specializzandolo
- Polimorfismo

Classi (1)

- Una classe è simile ad una struttura del C
  - Può però contenere anche funzioni al suo interno
- La sintassi per definire una nuova classe è:

```
class NomeClasse:
    "Documentazione della classe"
    ...
```

- Una classe può contenere dati (variabili locali alla classe) e metodi (funzioni specifiche della classe)
- Per essere usate le classi vengono istanziate, viene cioè creata un'istanza della classe
  - Simile al concetto di allocazione di una struttura in C

Classi (2)

```
class C:                                (1)
    cl = 42                             (2)
    def __init__(self):                 (3)
        self.ist = 12                   (4)
```

1. Viene creata una classe “C”
2. “cl” è una variabile condivisa da tutte le istanze
3. “__init__” è un metodo della classe “C”
   - “__init__” è un metodo speciale (chiamato costruttore), richiamato al momento della creazione dell’istanza
   - “self” è un argomento speciale che si riferisce all’istanza sulla quale è richiamato il metodo
4. “ist” è una variabile locale all’istanza

Classi (3)

```
i1 = C()                                    (1)
i2 = C()                                    (2)
print i1.cl,i2.cl,i1.ist,i2.ist   → 42 42 12 12
C.cl = 0                                    (3)
i1.ist = 3                                  (4)
print i1.cl,i2.cl,i1.ist,i2.ist   → 0 0 3 12
```

1. Viene creata un'istanza di "C"
2. Viene creata un'altra istanza di "C"
3. Viene assegnato "0" alla variabile di classe.
   - Il cambiamento avviene per ogni istanza!
4. Viene assegnato "3" alla variabile di istanza "i1.ist"
   - Il cambiamento avviene solo per "i1"!

self

- Ogni metodo accetta come primo argomento “self” che si riferisce all’istanza sulla quale il metodo è stato richiamato.

```
class C:
    def foo(self): print self, self.var
i1 = C()
i2 = C()
i1.var = 12
i2.var = 23
```

`i1.foo()` → <__main__.C instance at 0x00931750> 12

`i2.foo()` → <__main__.C instance at 0x0092BF18> 23

__init__

- Il costruttore viene richiamato ogni volta che viene creata un'istanza
- Il suo compito è porre la classe in una condizione iniziale utilizzabile
- Accetta "self" più eventuali altri argomenti

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def stampa(self): print self.x, self.y
Point().stampa()                    → 0 0
Point(1).stampa()                   → 1 0
Point(y=3).stampa()                 → 0 3
```

Metodi speciali

- Oltre ad __init__ esistono altri metodi speciali, tutti nella forma “__nomemetodo__”
- “__del__()” richiamato quando l’oggetto viene distrutto
- “__str__()” deve ritornare una stringa. Viene richiamato dalla funzione “str”
- “__repr__()” come il precedente ma chiamato da “repr”

Metodi speciali, operatori binari

- Quando l'interprete incontra "x + y" chiama "x.__add__(y)" se il metodo non esiste chiama "y.__radd__(x)"
- Il valore ritornato è il risultato dell'operazione
- Se l'operazione è commutativa allora normalmente si avrà

```
class C:
    def __add__(self, other): return ...
    __radd__ = __add__
```

- Tutti gli operatori binari accettano l'argomento "self" ed un altro argomento (generalmente chiamato "other") che è l'altro operando
- Nel caso vi sia un "+=" viene invece richiamato il metodo "__iadd__(other)"

Metodi speciali, operatori binari (2)

L’operatore * utilizza i metodi __mul__ e __rmul__.

print ”abc" * 3 -> ‘abcabcabc’
print 3 * ”abc” -> ‘abcabcabc’

NB “abc” è di tipo stringa
3 è di tipo int

Variabili e metodi privati

- Per una buona progettazione del software è necessario che i dettagli implementativi di una classe siano nascosti
- In Python i nomi di variabili e metodo che iniziano per "__" (eccetto quelli che finiscono anche per "__") non sono accessibili dall'esterno

```
class C: __x = 2
C().__x
```

→ **Errore!**

- Può, però, essere sufficiente lasciarli accessibili ma "segnarli" come pericolosi
- Per fare ciò basta far iniziare i nomi con un singolo "_"

Le eccezioni

- Quando una funzione incontra un errore lancia un eccezione
- L'errore viene porpagato nel codice (uscendo dalla funzione corrente se necessario)
- Se l'eccezione non viene catturata in nessuna parte di codice il programma termina
- Se l'eccezione viene catturata viene eseguito del codice particolare e l'esecuzione continua da quel punto

Esempio di propagazione delle eccezioni

- Esaminiamo il seguente codice:

```
def foo(): int("X")
def bar(): foo()
bar()
```

- Ecco cosa succede:
  - Viene chiamata “bar”, “bar” chiama “foo”, “foo” chiama “int”
  - “int” incontra un errore e lancia un’eccezione
  - “foo” non gestisce l’eccezione che si propaga al chiamante
  - “bar” non gestisce l’eccezione che si propaga al chiamante
  - Il codice esterno alle funzioni non gestisce l’eccezione che si propaga all’interprete
  - L’interprete termina il programma stampando un messaggio di errore

Il messaggio di errore

- Il messaggi di errore stampato contiene diverse utili informazioni.
- Messaggio prodotto dall'esempio precedente (supponiamo si trovi nel file "f.py")

```
Traceback (most recent call last):
  File "f.py", line 3, in ?
    bar()
  File "f.py", line 2, in bar
    def bar(): foo()
  File "f.py", line 1, in foo
    def foo(): int("X")
ValueError: invalid literal for int(): X
```

Catturare le eccezioni (1)

- Vediamo un caso tipico di gestione per gli errori:

```
try: n = int("X")          # errore
except ValueError: n = 0 # valore di default
```

- Il codice all'interno del blocco "try" viene eseguito
- Se non vi sono errori il blocco "except" è saltato
- Se vi è un errore il resto del blocco "try" è saltato
  - Se il tipo dell'errore coincide con il tipo dopo la keyword "except" viene eseguito il blocco successivo
  - Se il tipo non coincide l'errore viene propagato
- Vi possono essere più blocchi "except"
  - Viene eseguito il primo il cui tipo coincide

Catturare le eccezioni (2)

- Un singolo blocco “except” può catturare eccezioni di più tipi

```
try: ...
except (NameError, TypeError, ValueError):...
```

- È possibile memorizzare l’eccezione in una variabile

```
try: ...
except (NameError, ValueError)as e: print e
```

- Un blocco except senza tipo cattura tutte le eccezioni

```
try: ...
except: pass
```

  – Potrebbe catturare eccezioni che non si volevano bloccare

Catturare le eccezioni (3)

- È possibile risollevare la stessa eccezione con "raise"

```
try: ...
except NameError:
    print "Qualche informazione di debug"
    raise
```

- "else" definisce un blocco di codice eseguito se non vi sono stati errori

```
try: ...
except NameError: ...
else: ...
```

  - È meglio usare "else" che inserire il codice nel blocco "try"
    - Potrebbe generare altre eccezioni catturate per sbaglio

Lanciare le eccezioni

- La sintassi per lanciare le eccezioni è:

```
raise Tipo[, arg1, arg2, ...]
```

- Il primo argomento a “raise” è il tipo dell’eccezione
- Gli altri argomenti successivi sono gli argomenti al costruttore di “Tipo”

```
>>> try: raise ValueError, "Valore sbagliato"
>>> except ValueError as e: print e
Valore sbagliato
```

- Le eccezioni sono derivate normalmente da “Exception”
  - Si trova nel modulo “exceptions”
  - Il costruttore di “Exception” accetta un argomento stringa che viene restituito utilizzando “str” sull’istanza dell’oggetto

Operazioni di pulizia

- "try" ha un'altra istruzione opzionale "finally"

```
try: ... # codice che può generare eccezioni
finally: ... # codice di clean-up
```

- "finally" viene eseguito sempre
  - Se vengono lanciate eccezioni nel blocco "try"
  - Se si esce dal blocco "try" normalmente
  - Se si esce dal blocco "try" per una "return", "continue", "break"
- "finally" serve per rilasciare risorse esterne (file aperti, socket aperti ecc.)
- Un'istruzione try può avere solo uno (o più) "except" *o* un "finally"

Classi e funzioni

1) Scrivere una serie di metodi per lavorare su matrici . I metodi devono leggere la matrice da file, stamparla, trovare la media, il massimo e il minimo, ecc…
2) Scrivere un metodo per verificare l'operazione di Kaprekar.

L'algoritmo di Kaprekar itera I seguenti punti:

a) Prendi un qualsiasi numero di quattro cifre, usandone almeno due diverse. (Si possono inserire degli zero anche all'inizio.)
b) Sistema le cifre in ordine decrescente e poi in ordine crescente così da ottenere due numeri di quattro cifre, aggiungendo degli zero iniziali (o finali) se necessario.
c) Sottrai il numero più piccolo da quello più grande.
d) Ripeti il processo partendo dal punto b.

Questo processo, conosciuto come operazione di Kaprekar, andrà sempre incontro al suo punto fisso, o kernel, il 6174. Es.

8082 -> 8820 - 288 = 8532

8532 -> 8532 - 2358 = 6174

Classi

5) Costruire un dizionario di studenti in cui la chiave è la matricola ed il valore è una lista di tuple del tipo (nomesame, voto).

Creare una classe che accetta il dizionario precedente come parametro ed implementi I metodi:

- Lettura da file
- Stampa dei voti di uno studente
- Calcolo della media di uno studente
- Restituire una lista di tutti gli studenti che hanno sostenuto un esame
- Restituire la lista degli studenti con più di 5 esami

Classi

Scrivere una classe che data una lista o un file di
parole (vedi file word.txt) stampa gli anagrammi ordinati per numero decrescente.

Es.

…

(5, ['acronimo', 'armonico', 'ormonica', 'romancio', 'romanico'])

(4, ['stivare', 'svitare', 'verista', 'vistare'])

…

Suggerimento: scrivere I metodi

-calcola_anagrammi_lista(lista di parole)

-calcola_anagrammi(file)

-stampa_anagrammi()

-stampa_anagrammi_in_ordine()

Esercizi proposti

1) Scrivere una classe che gestisca un oggetto Data con gli attributi giorno, mese e anno. Scrivere una funzione chiamata incrementa_data che prende un oggetto Data, la data e un'intero n, e restituisce un nuovo oggetto Data che rappresenta il giorno dopo la data. Provate a farlo funzionare anche con gli anni bisestili? Vedi http://it.wikipedia.org/wiki/Anno_bisestile
2) Scrivere un programma che accetta come input un compleanno e stampi l'età dell'utente ed il numero di giorni, ore, minuti e secondi fino alloro prossimo compleanno.
   Chi vuole può utilizzare la classe date-time (http://docs.python.org/library/datetime.html) o in italiano (http://docs.python.it/html/lib/module-datetime.html)