Una Zuppa di Python

QUALCHE RICETTA PER PROGRAMMARE CON GUSTO

Alessandro Re

Chief Officier of Symmetrical Operations

TOGLIAMOCI 'STO SASSOLINO

DISCLAIMER

Non so come sia, ma ho usato Python per scopi migliori.

INGREDIENTI

- Pitone di piccole dimensioni
- Verdure (1 patata, 1 carota, fagiolini, mais)
- Spezie (curcuma, coriandolo, cumino macinato, pepe, zenzero)

PREPARAZIONE

Tenendolo saldamente fermo, tagliare la testa al pitone. Scottarlo in acqua bollente per pochi istanti per facilitare la rimozione della pelle. Pulire la carne sotto acqua corrente rimuovendo le interiora. Tagliare il pitone a pezzetti e farlo cuocere lentamente in poca acqua per 5-6 ore per ottenere una carne molto tenera. Quando la carne è cotta, aggiungere le verdure e proseguire la cottura. Aggiungere le spezie, mescolare e completare la cottura per altri 10 minuti.

PICCOLA NOTA

- Non serve fare foto alle slide
- Condividerò le slide (ripulite) su slack
  - slack.pycon.it
  - che verranno caricate su github.com/PythonMilano

COSA ASPETTARVI DAL TALK

- Per neofiti non estranei alla programmazione
  - Ma possiamo parlarne...
- Mostra il modo "Pythonico" di fare certe cose
- Se non capite una cosa, prendete appunti
  - Poi sperimentate e leggete la docs
- Se sapete già, spero troviate nuovi spunti
- Linux, MacOSX o Windows Subsystem for Linux
- $ → sistema, >>> → Python

Problema 1

Dato un file con un nome per riga, trovarne i duplicati.

SOLUZIONE 1/2: LETTURA

SEGUENDO L'INTUITO

```
# Apro il file
file_dati = open('ilfile.txt')

# In un ciclo che continua fino alla fine
while True:
    riga = file_dati.readline()  # Leggo una riga
    if riga == '':
        break  # Riga vuota = file finito
    else:
        print(line)

# Chiudo il file
file_dati.close()
```

Va bene così o ci sono problemi?

UNA SOLUZIONE MIGLIORE

```
# Uso un context manager!
with open('ilfile.txt') as file_dati:
    # Iterazione sul file!!
    for riga in file_dati:
        print(line)  # (strip)
```

1. Apertura e chiusura sono gestite assieme
2. La lettura delle righe è in automatico
3. Nessuna condizione esplicita di terminazione

```
with entratore_uscitore [as nome]:
    codice_in_contesto
```

SOLUZIONE 2/2: DUPLICATI

Uso un dizionario:

```
righe_viste = dict()
num_riga = 0
for riga in file_dati:
    num_riga += 1  # Parte da 1
    if riga in righe_viste:  # È una chiave?
        print('Duplicato alla riga', num_riga)
    else:
        righe_viste[riga] = True
```

Ci piace così?

MIGLIORIE

1. Dizionario con vero/falso → Insieme
2. Tener traccia di qualcosa già visto → Insieme
3. `enumerate()` conta elementi di una sequenza

```
righe_viste = set()
for num_riga, riga in enumerate(file_dati, 1):
    if riga in righe_viste:
        print('Duplicato alla riga', num_riga)
    else:
        righe_viste.add(riga)
```

Insiemi: si possono unire, intersecare, sottrarre, etc

Problema 2

Dato un file con un nome per riga, contare quante volte appare ogni nome

SOLUZIONE

Simile a prima, ma uso un dizionario per contare

```
n_righe = dict()
for riga in file_dati:
    if riga not in n_righe:
        n_righe[riga] = 1
    else:
        n_righe[riga] += 1
```

Ci piace?

MEGLIO, COSÌ?

```
n_righe = dict()
for riga in file_dati:
    n_righe[riga] = n_righe.get(riga, 0) + 1
```

`dict.get(key, default)`

se key non c'é, usa il default (che ha None come default)

MEGLIO QUESTO!

```
from collections import defaultdict

n_righe = defaultdict(int)
for riga in file_dati:
    n_righe[riga] += 1  # esegue n_righe[riga] = int()
                        # se necessario
```

`defaultdict(func)`

quando un elemento manca, lo costruisce usando func prima di restituirlo

TOP PROPRIO

```
from collections import Counter

n_righe = Counter()
for riga in file_dati:
    n_righe[riga] += 1
```

...anzi...

```
n_righe = Counter(file_dati)  # Itera sul file

print(n_righe.most_common())  # Comode :)
```

E potete anche sommarli e sottrarli!

```
print(Counter('ciao') - Counter('cosa')) # {'i': 1}
```

Problema 3

Dati due file, nomi e telefoni, creare un dizionario nome → numeri

Una persona può avere più numeri!

SOLUZIONE

Leggiamo tutti i nomi e i numeri

```
with open('filenomi.txt') as file_nomi:
    with open('filenumeri.txt') as file_numeri:
        nomi = list(file_nomi)
        numeri = list(file_numeri)
```

Mettiamoli in un dizionario

```
rubrica = dict()
for i in range(min(len(nomi), len(numeri))):  # len(lista)
    nome, numero = nomi[i], numeri[i]  # N.B.
    rubrica[nome] = rubrica.get(nome, []) + [numero]
    # [a, b, c] + [d] == [a, b, c, d]
```

Ok, non è male, no?

SISTEMIAMO IL DIZIONARIO

`defaultdict` sarebbe stato meglio, ma in questo caso usiamo `dict.setdefault`:

```
for i in range(min(len(nomi), len(numeri))):
    nome, numero = nomi[i], numeri[i]
    rubrica.setdefault(nome, []).append(numero)
```

Ora è tutto a posto!

FACILITARE LA LETTURA

Ok, ammetto che

```
for i in range(min(len(nomi), len(numeri))):
    nome, numero = nomi[i], numeri[i]
```

è abbastanza brutto e illeggibile. Possiamo fare di meglio?

SISTEMIAMO IN UNA LAMPO!

Per fortuna, in Python possiamo fare di meglio

```
for nome, numero in zip(nomi, numeri):
    pass
```

`zip(*iteratori)`

prende un qualunque numero di sequenze e ritorna gli i-esmini elementi in una tupla, finendo quando finisce il primo iteratore.

DOPPIO VANTAGGIO

Cosa succede se abbiamo un mucchio di dati?

```
with open('filenomi.txt') as file_nomi:
    with open('filenumeri.txt') as file_numeri:
        nomi = list(file_nomi)
        numeri = list(file_numeri)

rubrica = dict()
for nome, numero in zip(nomi, numeri):
    rubrica.setdefault(nome, []).append(numero)
```

BENVENUTI ITERATORI

La lettura dei file può avvenire una riga alla volta, possiamo fare così;

```
rubrica = dict()
with open('filenomi.txt') as file_nomi:
    with open('filenumeri.txt') as file_numeri:
        for nome, numero in zip(file_nomi, file_numeri):
            rubrica.setdefault(nome, []).append(numero)
```

Problema 4

Hai un file da 1TB con 1 documento per riga da 1GB.

Si vogliono filtrare le parole in ogni riga in base ad un vocabolario prestabilito, ritornandole in ordine alfabetico.

ESEMPIO

dato il seguente

```
riga = "Perché dimentico tutte le lezioni dell'università"
       "ma ricordo il 100% delle battute dei Simpsons?"
```

viene trasformato in

```
['battuta', 'dimenticare', 'lezione', 'ricordare',
 'Simpsons', 'università']
```

SEPARARE LE PAROLE

Usiamo `str.split()` per separare una stringa in base agli spazi e mettiamole in una lista.

```
filtrate = []
for parola in riga.split():
    if parola in vocabolario:
        filtrate.append(parola)
```

Direi che è Ok!

SEPARARE LE PAROLE

TENENDO CONTO DELLA PUNTEGGIATURA

Decido di non usare una regex per risolvere il problema[1].

Decido quindi di iterare su ogni singola lettera e considerare "parole" solo quelle composte da caratteri alfabetici.

[1]) Anche se `import re` mi suona davvero elegante e familiare, per qualche motivo.

UNA LISTA DI PAROLE

```
def parole(stringa):
    lista_parole = []
    parola = ''
    for l in stringa:
        if l.isalpha():
            parola = parola + l
        elif parola:
            lista_parole.append(parola)
            parola = ''
    if parola:
        lista_parole.append(parola)
    return lista_parole
```

Funziona, c'ho i test!!1!

Ma è un po' bruttino...

IN MENO RIGHE

Dovendo iterare su delle sequenze, itertools ci può essere utile.

```
from itertools import groupby

def parole(stringa):
    """Ritorna una lista di parole nella stringa."""
    lista_parole = []
    for is_alf, gruppo in groupby(stringa, key=str.isalpha):
        if is_alf:
            lista_parole.append(''.join(gruppo))
    return lista_parole
```

Chiaro, no?

ITERATORI IN BREVE

`groupby` non funziona solo su stringhe, ma come fa?

```
>>> s = 'sequenza'
>>> i = iter(s)
>>> i
<str_iterator object at 0x1337133713337>
>>> next(i)
's'
>>> next(i)
'e'
...
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

FOR LOOP IN BREVE

```
for c in 'sequenza':
    print(c)
```

è sostanzialmente equivalente a questo:

```
i = iter('sequenza')  # Iteratore
while True:
    try:
        c = next(i)  # Prossimo elemento
        print(c)
    except StopIteration:
        break  # Esci dal ciclo
```

IL RUOLO DEGLI ITERATORI

Permettono di scorrere tutti gli elementi di una struttura dati indipendentemente da com'è fatta.

```
iter(stringa)  # Itera i caratteri
iter(lista)  # Itera gli elementi
iter(insieme)  # Itera gli elementi
iter(dizionari)  # Itera le chiavi
iter(tua_struttura_dati)  # Non è magia: Iterator Protocol
```

Molte funzioni ricevono un iteratore o un iterabile:

```
list('sequenza')  # ['s', 'e', 'q', 'u', 'e', 'n', 'z', 'a']
list(iter('sequenza'))  # Come sopra
```

Problema 4.1

Ok, abbiamo una funzione che restituisce delle parole...

Ma le stringhe sono di 1GB l'una, quindi avremo davvero tante parole!!

GENERATORS TO THE RESCUE

Simile ad un iteratore in questo senso:

1. Lo crei
2. Lo chiami per ottenere il prossimo dato
3. Quando finisce alza un'eccezione

Differenze:

DEFINIZIONE

Un generatore (funzione generatrice) è una funzione che usa la keyword `yield`:

```
def fibonacci(n):
    n1, n2 = 0, 1
    for i in range(n):
        yield n1
        n1, n2 = n2, n1+n2
```

E si può usare così:

```
for n in fibonacci(3):
    print(n)
```

ESPLORIAMOLO UN PO'...

Il generatore non ritona una sequenza, ma un iterator

```
>>> g = fibonacci(3)
>>> g
<generator object fibonacci at 0x3133780023>
>>> next(g)
0
>>> next(g)
1
...
StopIteration
```

Si "consuma" come gli iteratori!

YIELD SOSPENDE L'ESECUZIONE

È un "return con memoria"

```
def quadrati(n):
    for i in range(1, n+1):
        print('Produco', i**2)
        yield i**2

>>> g = quadrati(3)
>>> _ = next(g)
Produco 1
>>> _ = next(g)
Produco 4
>>> _ = next(g)
Produco 9
```

GENERATORE DI PAROLE

Sistemiamo la funzione che ritorna le parole per generane una alla volta

```
from itertools import groupby

def parole(stringa):
    """Genera una lista di parole nella stringa."""
    for is_alf, gruppo in groupby(stringa, key=str.isalpha):
        if is_alf:
            yield ''.join(gruppo)

sorted(parole(stringa))  # Ritorna una lista
```

Adesso il problema è solo `stringa`...

''.JOIN(GRUPPO) -VS- STR(GRUPPO)

Rappresentazione in stringa di un gruppo (di groupby)

```
<itertools._grouper object at 0xf00b00f00b00>
```

È necessario iterare il gruppo per ottenere le singole lettere ed unirle in una stringa.

```
assert 'XX'.join(['a', 'b', 'c']) == 'aXXbXXc'
assert ''.join(['a', 'b', 'c']) == 'abc'
```

Problema 5

Dato un file con un elenco di spese, vogliamo calcolare il totale e dividere per il numero di rate che andremo a pagare.

FACILISSIMO!!!1!

```
totale = 0
num_rate = 3
with open('spese.txt') as file_spese:
    for riga in file_spese:
        totale += float(riga.strip())
print('Ogni rata sarà di', totale / num_rate, 'euro')
```

Piece of cake.

CIÒ CHE MI PERPLIME

Non fate affidamento su questo codice per molte transazioni.

- Problemi di overflow (int vs float)
- Problemi di precisione

DECIMAL

Rappresentazione precisa di un numero:

```
>>> from decimal import Decimal as D
>>> D('3.14')
Decimal('3.14')
>>> D(3.14)
Decimal('3.140000000000000124344978758017532527446746825')
```

Può quasi sostituire un normale numero float

```
>>> D('3.14') + 1
Decimal('4.14')
```

FRACTION

lo preferisco usare le frazioni:

```
>>> from fractions import Fraction as F
>>> F('12/5')
Fraction(12, 5)
>>> F('3.14')
Fraction(157, 50)
>>> F(3.14)
Fraction(7070651414971679, 2251799813685248)
>>> float(F(1, 3))
0.3333333333333333
```

A volte è più conveniente di Decimal (e.g. probabilità)

ALTRETTANTO FACILE, MA PIÙ SICURO

```
from fractions import Fraction
totale = Fraction(0)
num_rate = 3
with open('spese.txt') as file_spese:
    for riga in file_spese:
        totale += Fraction(riga.strip())
print('Ogni rata sarà di', totale / num_rate, 'euro')
```

UN TRUCCHETTO

A volte potrebbe capitarvi di dover dividere una quantità e di tenere il resto.

Python ha due opzioni molto utili in casi simili:

```
>>> 7 % 3  # Resto (modulo)
1
>>> 7 // 3  # floor division
2
>>> divmod(7, 3) # Divisione intera e resto (modulo)
2, 1
```

Problema N

Scrivo il codice → trovo un bug → sistemo il bug.

Cambio il codice → ritrovo il bug.

UNA STRATEGIA

La cosa migliore da fare in ~~questi casi~~ *generale* è

scrivere dei test, ovvero

> *codice che verifica che altro codice faccia quello che ci si aspetta che faccia.*

(Segue: una micro-introduzione al testing)

IN PRATICA

Quando scrivete un codice che fa qualcosa

```
def calcola(a, op, b):
    if op == '+':
        return a + b
    if op == '-':
        return a - b
```

scrivete anche del codice di test

```
def test_calcola():
    assert calcola(1, '+', 2) == 3  # Conoscete assert?
    assert calcola(1, '-', 2) == -1
    assert calcola(0, '*', 5) == 0  # TODO
```

I test definiscono il comportamento desiderato.

TEST AUTOMATIZZATI

Ovviamente, quel codice di test bisogna eseguirlo in qualche modo.

Esistono strumenti per farlo in modo automatico.

Ad esempio, pytest (pytest.org) è comodissimo:

```
$ pytest
...
   def test_calcola():
       assert calcola(1, '+', 2) == 3
       assert calcola(1, '-', 2) == -1
>      assert calcola(0, '*', 5) == 0
E      assert None == 0
E       + where None = calcola(0, '*', 5)
calcolatrice.py:11: AssertionError
=============== 1 failed in 0.01 seconds ================
```

SCRIVI TEST PER IL TUO CODICE

In generale, volete testare solo le cose che scrivete, non quelle che usate.

```
import statistics
def calcola(a, op, b):
    if op == '+':
        return a + b
    if op == '-':
        return a - b
    if op == 'media':
        return statistics.mean([a, b])
```

Usate dei mock (oggetti fantoccio) per simulare gli elementi esterni.

Scrivere i test non è un esercizio di stile: fatelo bene!

Spezie varie

<Inserire qui battuta sull'erba che aggiunge sapore alla vita>

F-STRINGS

Una buona ragione per passare a Python 3.6

```
n = 100
s1 = str(n-1) + ' scimmie saltavano sul letto'  # Brutto!
s2 = '{} scimmie saltavano sul letto'.format(n-2)  # Meh...
s3 = f'{n-3} scimmie saltavano sul letto'  # Da Python 3.6
```

ASSEGNAMENTO

Sapevate che si possono assegnare più cose assieme?

```
a, b = 1, 2
a, b = b, a  # Swap

a, b, c = (1, 2, 3)

def prova():
    return [1, 2, 3]

x, y, z = prova()

i, *rest = range(40) # i = 0, rest = [1, ... 39]
```

COMPREHENSIONS

Forme compatte per creare "one liners" di insiemi e liste

```
parola = 'indipendenza'

insieme = set()
for l in parola:
    insieme.add(l.upper())
insieme = {l.upper() for l in parola}

lista = list()
for l in insieme:
    lista.append(l.lower())
lista = [l.lower() for l in insieme]
```

COMPREHENSIONS

... Ma anche di dizionari e generatori:

```
diz = dict()
for l in lista:
    diz[l] = parola.count(l)
diz = {l: parola.count(l) for l in lista}

def generatore(diz):
    for l, n in diz.items():  # Items, non chiavi
        yield (l.upper(), n+1)
generatore = ((l.upper(), n+1) for l, n in diz.items())
```

Generatore come argomento di funzione:

```
sum(parola.count(l) for l in insieme)  # Numero lettere
n_righe = Counter(riga for riga in file_dati)
```

ITER() PUÒ TORNARE UTILE

`iter(funzione, sentinella)` può tornarvi utile in alcune occasioni:

```
from random import randint

def lancia():
    """Lancia un dado a 6 facce."""
    return randint(1, 6)

# Lancia un dado finché non esce 6 (escluso)
sequenza = list(iter(lancia, 6))
```

COMPARARE OGGETTI

In python ci sono `is` e ==, ma che differenza c'é?

```
if x is None:  # da preferire per None
    pass
assert None == None and None is None

assert 'ciao' is 'ciao'
bufo = b'ciao'.decode()
assert bufo == 'ciao'
assert bufo is not 'ciao'

assert 2**8 == 2**8
assert 2**8 is 2**8

assert 2**9 == 2**9
assert 2**9 is not 2**9  # In CPython
```

OPERATORI DI CONFRONTO

Sapevate che in Python si può fare questo?

```
assert 1 < 2 < 3
```

oppure

```
assert 1 != 2 == 2
```

OPERATORI DI ALLUCINAZIONI?

E sapevate che quindi si può fare questo?

```
assert 1 == 1 is not True
```

Oppure questo?

```
assert False == False in [False]
```

PARAMETRI VARIABILI

```
def moltiplica(*numeri):
    num = 1
    for n in numeri:
        num *= n
    return num
moltiplica(1, 2, 3)

def applica(*numeri, **opzioni):
    if opzioni.get('sum', False):
        return sum(numeri)
    else:
        return moltiplica(numeri)
applica(3, 2, 1, sum=True)
```

ESPANSIONE DI ARGOMENTI

```
def somma(a, b, c):
    return a + b + c

numeri = [1, 3, 5]
somma(numeri[0], numeri[1], numeri[2])
somma(*numeri)

def sottrai(a=1, b=2, c=3):
    return a - b - c

numeri = {'a': 3, 'b': 5}
sottrai(**numeri)
```

LA SIMMETRIA DI ZIP E *

Zip è stato sviscerato, discusso e analizzato di già.
Ma un suo risvolto vi stupirà, di me che son reietto
è il tema prediletto:

```
>>> x = list(zip([1, 2, 3], [9, 8, 7]))
>>> x
[(1, 9), (2, 8), (3, 7)]
>>> list(zip(*x))
[(1, 2, 3), (9, 8, 7)]
```

UN NOME PER OGNI COSA

Nell'esempio precedente, ho usato una convenzione, tipo

```
dato = (32, 'Ale', 'Milano', 32)  # WAT
```

Anziché affidarsi alle convenzioni, usiamo i nomi

```
from collections import namedtuple

utente = namedtuple('utente', 'age name city days_to_birthday'
dato = utente(32, 'Ale', 'Milano', 32)
print(dato.age, dato.days_to_birthday)
```

COMPARARE STRINGHE

```
assert 'Gauss'.lower() == 'gauss'  # Di solito si fa così...
assert 'Gauß'.lower() != 'gauss'  # Non dovrebbe...
assert 'Gauß'.casefold() == 'gauss'  # Giusto :)
```

Troppa roba?

NON RIUSCITE A DIGERIRE
QUESTA ZUPPA?

VI CONSIGLIO DELLO ZEN-ZERO

```
>>> import this
The Zen of Python, by Tim Peters
```

Fine!

NOTE

- Slide state fatte con RevealJS, è la prima volta che lo uso, scusate.
- Ho tenuto conto solo di CPython, se siete amanti di PyPy o altre implementazioni scusatemi.
- Scusatemi anche se ho fatto errori (ditemelo, per favore!)
- Ma soprattutto, scusatemi per l'intera presentazione.

Problema Zero

Uh, sul mio computer funzionava...

DOCKER

- Non c'entra direttamente con Python (è in Go).
  - Molto comodo per sviluppare
  - Vi fornisce una specie di computer virtuale
- "Eseguire" un'immagine docker è...
  - Più o meno come avviare un nuovo PC
  - Pulito, controllabile, riproducibile
- Usare una versione specifica di Python

ESEMPIO

Lanciare un interprete Python 3.6

```
$ docker run -it python:3.6
```

Lanciare un interprete Python 2.7

```
$ docker run -it python:2.7
```

Accesso ai dati della dir corrente (volume)

```
$ docker run -i -t --rm -v "$PWD:/zuppa" python:3.6
```

AMBIENTI VIRTUALI

- Un computer → Vari progetti
- Un progetto → Vari pacchetti (librerie)
- Un pacchetto → Varie versioni

Come tenere tutto pulito?

COME USARLI

Create un ambiente ed attivatelo

```
$ python3 -m venv mio-ambiente
$ source mio-ambiente/bin/activate
```

Usate python normalmente

```
(mio-ambiente)$ python3 programma.py
```

Disattivatelo quando avete finito

```
(mio-ambiente)$ deactivate
```

GESTIONE DEI PACCHETTI

Ambiente virtuale o meno, i pacchetti vanno gestiti!

PIP - Python Package Index

```
$ python3 -m pip install nome-pacchetto
$ python3 -m pip install -U pacchetto-da-aggiornare
$ python3 -m pip uninstall pacchetto-da-rimuovere
```

(Magari avete `pip` o `pip3` già sul sistema)

ESEMPIO: FACCIAMO LA SCIENZIA

```
          $ python3 -m venv scienzia
          $ source scienzia/bin/activate
(scienzia)$ python3 -m pip install numpy
(scienzia)$ python3
        >>> import numpy as np
        >>> np.sum(np.arange(1, 50))
        1225
        >>> exit()
(scienzia)$ deactivate
          $ rm -rf scienzia
```

MOCKING

```
from unittest.mock import MagicMock, patch

def test_calcola_media():
    with patch('calcolatrice.statistics.mean') as mean:
        mean.return_value = 5  # Valore di ritorno
        assert calcola(4, 'media', 6) == 5

        # Oppure una funzione che ne determina gli effetti
        mean.side_effect = lambda l: 0.5 * (l[0] + l[1])
        assert calcola(4, 'media', 6) == 5
        assert calcola(1, 'media', 5) == 3
        assert calcola(0, 'media', 0) == 0
```

Gli oggetti `Mock` e `MagicMock` sono entità molto curiose, tanto semplici quanto stupefacenti. Il mio consiglio è di usarle e giocarci.