Università di Roma Tor Vergata

Tutoraggio

Corso di ingegneria degli algoritmi
A.A. 2017/2018

Tutoraggio IA 2017/18

- **Giacomo Marciani**: giacomo.marciani@alumni.uniroma2.eu
- **Laura Trivelloni**: laura.trivelloni@alumni.uniroma2.eu
- **Emanuele Vannacci**: emanuele.vannacci@alumni.uniroma2.eu

Inserire oggetto [IA17]

Sito Web del Corso: https://utv-teaching.github.io/algorithms-engineering-2017/

- Orario Lezione:
  - Ogni lunedì dalle 14.00 alle 15.45 in Aula B2

Modalità esame

La prova pratica influirà sul **25%** del voto finale.

- Due prove in itinere
  - Prima prova (bassa complessità)
    - Da svolgersi individualmente
    - Disponibile da metà Novembre
  - Seconda prova (media difficoltà)
    - Da svolgersi individualmente o in un team (max 3 persone)
    - Disponibile da metà Dicembre

Oppure

- Prova finale (alta complessità)
  - Da svolgersi individualmente o in team (max 3 persone)

Modalità esame

La prova pratica influirà sul **25%** del voto finale.

- Due prove in itinere
  - Prima prova (bassa complessità)
    - Da svolgersi individualmente
    - Disponibile da metà Novembre
  - Seconda prova (media difficoltà)
    - Da svolgersi individualmente o in un team (max 3 persone)
    - Disponibile da metà Dicembre

Oppure

- Prova finale (alta complessità)
  - Da svolgersi individualmente o in team (max 3 persone)

È **caldamente consigliata** la prima opzione!

Development

- Interprete **Python**, la PVM e i relativi moduli di base.
  - http://www.python.org (oppure su http://www.python.it) nella sezione download.
  - I sistemi Unix-based dovrebbero avere già installato Python, ma potrebbero aver bisogno di aggiornare la versione di Python preinstallata nel sistema.
  - python --version

- Potete utilizzare l'IDE che più preferite.
  - Noi consigliamo: **PyCharm**: https://www.jetbrains.com/pycharm/
  - Licenza disponibile registrandosi con l'e-mail istituzionale nome.cognome@students.uniroma2.eu

Codice

Il codice esaminato nelle prossime lezioni sarà raggiungibile dal sito del corso.

I link portano a un repository su **github**. Potrete:

- Scaricare il codice

Oppure

- Utilizzare **git** per creare un repository locale.
  - git clone <repository>

Git

- https://git-scm.com/
- git book: https://git-scm.com/book/en/v2

In particolare: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository

Perchè Python

- Grande utilizzo in ambito aziendale
- Vasta gamma di librerie
- Permette di concentrarsi sulla logica di programmazione
  - Sul cosa fare e non sul come
- Performance
  - Compilato in Bytecode ottimizzato
  - Molti moduli sono scritti in C
- Vasta community
  - https://www.python.org/dev/peps/
- La NASA usa python!
  - https://www.python.org/about/success/usa/

Come scrivere il codice

- Commento introduttivo in cui includere almeno il nome dell'autore e una breve descrizione del programma
- Import dei moduli richiesti dal programma subito dopo il commento introduttivo
- Inizializzazione di eventuali variabili di modulo
- Definizione delle funzioni, tra cui la funzione main
- Docstring per ogni funzione definita nel programma.
- Uso di nomi significativi.

Attenzione alle convenzioni:

- Nomi di funzioni, metodi e di variabili iniziano sempre con la lettera minuscola.
- Nomi di classi con la lettera maiuscola.
- Usare in entrambi i casi la notazione CamelCase.
- Nel caso di costanti scrivere il nome tutto in maiuscolo

I moduli in Python

- Un modulo è un particolare script Python.
- Sono utili per decomporre un programma di grande dimensione in più files, oppure per riutilizzare codice scritto precedentemente.
- Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) attraverso il comando **import**.
- Il nome di un modulo è il nome del file script (esclusa l'estensione «.py»).
- All'interno di un modulo si può accedere al suo nome tramite la variabile globale **__name__**

L’istruzione import

- **import math :** per importare tutto il modulo math.
  - es. math.sqrt(<numero>).
- **from math import pi, sqrt** : permette di importare solo alcune risorse da un modulo.
  - In questo modo si può evitare l’uso del qualificatore math.
  - es. sqrt(<numero>)
- Per importare i propri moduli: **import <percorso relativo del modulo>**
- Una volta effettuato l’import, lo script importato verrà eseguito.

L’istruzione import: esempio

- **import math :** per importare tutto il modulo math.
  - es. math.sqrt(<numero>).
- **from math import pi, sqrt** : permette di importare solo alcune risorse da un modulo.
  - In questo modo si può evitare l’uso del qualificatore math.
  - es. sqrt(<numero>)
- Per importare i propri moduli: **import <percorso relativo del modulo>**
- Una volta effettuato l’import, lo script importato verrà eseguito.

Come evitare che ad ogni import venga eseguito il codice del modulo corrispondente?

__name__

- Un modulo dovrebbe eseguire il proprio codice solo quando viene lanciato direttamente dall'utente, non quando ne viene effettuato l'import
- Se un modulo viene eseguito direttamente la variabile globale **__name__** sarà uguale alla stringa **"__main__"**.

```
def main():
    <istruzioni>

if __name__ == “__main__”:
    main()
```

Accedere agli argomenti passati

- Eseguire uno script: **python main.py <prog> arg1 .. argN**
- Importare il modulo **sys**
- All'interno dello script si può accedere alla lista degli argomenti tramite **sys.argv**
  - L'argomento in posizione 0 è il nome del file che si sta eseguendo.
- Si controlla se il numero degli argomenti passato è proprio quello che ci aspettavamo
  - Ad esempio **if len(sys.argv) == 4:**
- Si accede ad ogni singolo elemento attraverso **sys.argv[<indice>]**
- Altrimenti: **getopt** C style parser for command line options
  - https://docs.python.org/2/library/getopt.html#module-getopt

Tipizzazione dinamica

- Python è **dinamicamente tipizzato**
- Una volta che una variabile viene inizializzata, Python ne desume il giusto tipo e lo assegna alla variabile in questione.
- Le variabili hanno tipi che possono cambiare durante l'esecuzione del programma

- Alcuni tipi gestiti nativamente da Python:
  - Variabili numeriche: interi (int e long), virgola mobile (float)
  - Boolean: può assumere una delle due costanti True o False
  - Strutture dati: liste, insiemi, dizionari, ...

Istruzioni di controllo

Ciclo for:

for <variabile> in <sequenza>:
<operazione>

La sequenza può essere espressa da:

- [1, 2, 3] una sequenza di numeri
- Una stringa che viene vista come una sequenza di caratteri
- Un file che viene visto come una sequenza di righe di testo.
- Con la funzione range
  - range(10) indica la sequenza da 0 a 9
  - range(1, 10) indica la sequenza da 1 a 9
  - range(10, 0, -1) indica la sequenza decrescente da 10 a 1

Istruzioni di controllo

Il costrutto **if-elif-else** permette di scegliere tra diverse alternative:

```
if <condizione1>:
        <istruzioni 1>
elif <condizioneN>:
        <istruzioni N>
else:
        <struzioni di default>
```

Il ciclo while permette di creare un ciclo controllato da contatore

```
while <condizione>:
        <sequenza di istruzioni>
```

Per uscire da un ciclo si può usare anche l'istruzione **break**.

Input e output da standard I\O

- Scrivere sullo standard output: **print()**
- Leggere dallo standard input: **raw_input()**
- In alternativa: **sys.stdin** e **sys.stdout**
- Per aprire un file:
  - f = open(‘pathname’, ‘modalità’)
  - Apre il file che si trova nel path indicato e restituisce un oggetto file.
- Il pathname può essere relativo o assoluto. Se il file definito
- dal pathname non esiste, viene creato.
-
- La modalità può essere:
  - ‘r’ (read only), ‘w’ (write only), ‘a’ (append, only write), ‘r+’ (read and write), w+’ (write and read), a+’ (append, read and write)
  - Esempio: f = open(‘myFile.txt’, ‘r’)

Gestione dei file

- **f.write(<stringa>)** per scrivere una stringa in un file
  - Restituisce il numero di caratteri che sono stati scritti.
  - Se non si scrive a partire dalla fine del file, si sovrascrive il file.
- **f.read()** legge tutto il contenuto del file e lo restituisce in una singola stringa.
  - Restituisce “” (la stringa vuota) se raggiunge la fine del file.
- **f.readline()** legge una singola riga di testo dal file e la restituisce compresa di carattere di accapo.
  - Restituisce “” se raggiunge la fine del file.
- **f.close()** Chiude un file.
- **f.seek(<offset>, <position>)** Per modificare la pos. corrente.
  - <position> può essere 0 (inizio), 1(pos.corrente) e 2 (fine).
- N.B. Se si apre un file con modalità r o r+, se il file non esiste non verrà creato in automatico, bensì verrà lanciato l’errore **FileNotFoundError**

Liste

Struttura dati mutabile che rappresenta la sequenza di una serie di dati chiamati elementi e caratterizzati da un indice unico che ne specifica la posizione.
Possono contenere collezioni di oggetti di tipo diverso

- Possono contenere collezioni di oggetti di tipo diverso
- Ottenere delle sottoliste da l = [0, [1, 2, 3], 4, 5, [6, 7, 8]] :
  - l[1:2] #restituisce [[1, 2, 3]] l[:2] #restituisce [0, [1, 2, 3]]
  - l[2:] #restituisce [4, 5, [6, 7, 8]]
  - l[-2:] #restituisce [5, [6, 7, 8]]
  - l[:-2] #restituisce[0, [1, 2, 3], 4]
- Funzioni e operatori applicabili alle liste:
  - len(<lista>)
  - <lista-1> =+ <lista> #concatenamento di liste

Liste

Metodi da applicare alle liste:

- **L.append(<elem>)** inserisce l'elemento in coda a L.
- **L.extend(<lista>)** aggiunge gli elementi di <lista> in coda a L
- **L.insert(<indice>,<elem>)**
- **L.pop()** restituisce ed elimina l'elemento che si trova alla fine di L
- **L.pop(<indice>)** restituisce ed elimina l'elemento che si trova in posizione indice.
- **L.sort()** ordina gli elementi della lista (se sono tutti interi o stringhe, non ordina liste miste)
- **L.index(<index>)** restituisce la posizione occupata dall'elemento che si sta cercando.

Alcuni concetti fondamentali

- **Aliasing**
  - Più variabili possono fare riferimento allo stesso oggetto.
  - Modificando il contenuto di una variabile, la modifica sarà osservata anche dalle altre variabili.
- **Identità degli oggetti**
  - Quando due variabili fanno riferimento allo stesso oggetto.
  - listaA is listaB : L'operatore **is** ci permette di capire se due variabili fanno riferimento allo stesso oggetto.
- **Equivalenza strutturale**
  - quando due variabili fanno riferimento a due oggetti diversi che però sono strutturalmente identici.
  - In questo secondo caso, la modifica di una variabile non implica la modifica dell'altra variabile.
  - L'operatore == ci dice se due variabili fanno riferimento a oggetti strutturalmente equivalenti.

Dizionari

- Le liste (ma anche stringhe e tuple) associano un dato alla posizione occupata da quel dato.
  - Un dato è indicizzato ad una chiave, che è il numero intero che rappresenta la posizione occupata.
- Il dizionario è una struttura che organizza i dati per associazione e non per posizione.
  - Un dato è indicizzato ad una chiave, che può essere qualsiasi tipo di dato immutabile.

Sintassi del dizionario

```
studenti = {‘’012345’’:’’Carlo’’ , ‘’098765’’:’’Alessia’’}
```

In questo esempio si è costruito il dizionario studenti indicizzato sul numero di matricola.

Dizionari

- Per accedere ai valori si può fare:
  - **<dict>[key]** : in tal modo viene restituito il valore associato alla chiave, se la chiave esiste, altrimenti viene generato un errore.
- Per aggiungere o sostituire valori all'interno di un dizionario:
  - **<dict>[key] = value**
  - Se la chiave non esiste si aggiunge la coppia key-value al dizionario
  - Se la chiave esiste si effettua la sostituzione del vecchio valore associato a con il nuovo

Dizionari

Metodi da applicare ai dizionari

- **<dict>.get(<key>,<default>)** : restituisce il valore associato alla chiave cercata, altrimenti il valore di default
- **<dict>.pop(<key>,<default>)** : elimina la voce identificata da <key>
- **<dict>.items()** : restituisce l'insieme di tuple
- **<dict>.values()** : restituisce l'insieme dei valori
- **<dict>.keys()** : restituisce l'insieme delle chiavi .
- **<dict>.clear()** : elimina tutte le chiavi del dizionario
- **len()** : restituisce il numero di voci

Definizione di Funzioni

- Una funzione si definisce utilizzando la parola chiave **def**

```
nome_funzione(<args>, <key-1>=<value-1 … ,
        <key-n>=<value-n> ):
    """docstring"""
    codice funzione
```

- Il corpo della funzione deve essere indentato.
- Subito dopo la definizione della funzione può esserci una stringa che documenta la stringa (la docstring).
- Se la funzione viene chiamata senza argomenti opzionali, questi assumono i valori di default.

Definizione di Classi

- La classe è un meccanismo di astrazione per poter definire entità del mondo esterno.
- Un oggetto è un'istanza di una classe.
  - Un oggetto è in relazione "istance of" con la propria classe
  - La classe ne definisce il tipo.
- Una classe possiede: Attributi e Operazioni.
  - Un metodo è l'implementazione di un'operazione
  - Gli attributi definiscono lo stato
  - Le operazioni definiscono il comportamento

Definizione di Classi

- Nome della classe solitamente scritto con l'iniziale in maiuscolo. a
- La classe **object** è la classe "radice" che generalizza tutte le altre.
- Tutti i metodi di una classe hanno come primo parametro **self**
- **__init__** rappresenta il costruttore della classe.
  - Viene chiamato quando l'oggetto è istanziato
- **__str__** è un metodo che restituisce una stringa.
  - Quando viene eseguita la print dell'oggetto, sarà stampata a schermo la stringa ritornata da questo metodo.

Definizione di Classi

Gli oggetti «nascono» quando vengono istanziati, ma quando «muoiono»?

- Un oggetto diventa candidato all'obitorio quando quando non ci sono più riferimenti all'oggetto.
- L'operazione di liberazione della memoria occupata dall'oggetto viene eseguito dal **garbage collector** (letteralmente raccolta dei rifiuti).
- In generale il tempo di inizio esecuzione del garbage collector non è predicibile
    - La macchina python gestisce in modo trasparente il meccanismo

Gestione delle eccezioni

- Gli errori di sintassi (errori di parsing) vengono sollevati dall'analizzatore sintattico (parser)
- Anche se un'istruzione è sintatticamente corretta, può generare degli errori a tempo di runtime
  - es: integer division or modulo by zero (ZeroDivisionError)
  - es: cannot concatenate 'str' and 'int' object (TypeError)
- Esiste un elenco completo delle eccezioni built-in in python
  - http://docs.python.it/html/lib/module-exceptions.html

Si possono definire delle eccezioni non presenti nativamente.

Gestione delle eccezioni

- Gli errori di sintassi (errori di parsing) vengono sollevati dall'analizzatore sintattico (parser)
- Anche se un'istruzione è sintatticamente corretta, può generare degli errori a tempo di runtime
  - es: integer division or modulo by zero (ZeroDivisionError)
  - es: cannot concatenate 'str' and 'int' object (TypeError)
- Esiste un elenco completo delle eccezioni built-in in python
  - http://docs.python.it/html/lib/module-exceptions.html

Si possono definire delle eccezioni non presenti nativamente.

Come?

Gestione delle eccezioni

- Un’eccezione è un oggetto
- Si può estendere per aggiungere significato e sollevare attraverso il meccanismo **raise**

```
>>class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)

>> raise MyError(2)
```

Gestione delle eccezioni

```
try:
    <istruzioni>
except <tipo di eccezioni>:
    <istruzioni>
finally:
    <istruzioni>
```

- Se una un'istruzione all'interno del blocco **try** solleva un'eccezione, verranno saltate le righe successive ed eventualmente verrà eseguito il codice del giusto blocco except.
- Un blocco **except** cattura e dunque gestisce solo le eccezioni del tipo specificato.
- Le istruzioni nel blocco **finally** vengono eseguite sempre in ogni caso

Gestione delle eccezioni

Sintassi completa del blocco try/except :

```
try:
    <istruzioni>
except <eccezione> :
    <istruzioni>
else:
    <istruzioni>
finally:
    <istruzioni>
```

Code Profiling

Se siamo interessati ad avere codice efficiente abbiamo bisogno di:

- Conoscere buone tecniche algoritmiche
- Saperle implementare in modo efficiente
- Valutare l'efficienza del proprio codice

Per poter misurare le prestazioni del nostro codice, potremmo valutare il tempo di esecuzione globale del codice

- Importare il modulo time e usare la funzione ***time()*** che restituisce l'equivalente del timestamp in secondi, attraverso un valore float.

Code Profiling

Esempio:

```
import time
inizio = time.time()
funzioneDaTestare()
tempoTrascorso = time.time() – inizio
```

Sebbene soggette ad errori, questo tipo di misurazioni è molto utile per capire l’efficienza delle nostre implementazioni.

Code Profiling

Ma se il nostro scopo fosse quello di individuare le porzioni di codice da ottimizzare o di avere una comprensione migliore di come viene ripartito il tempo in diversi task nel codice che eseguiamo?

In questo caso si deve fare profiling del codice, ossia monitorare e riportare informazioni temporali riguardanti l'esecuzione del codice.

In particolare siamo interessati a conoscere il tempo speso nell'esecuzione di funzioni e sotto-funzioni. Per rispondere a questa esigenza Python ci viene in aiuto con i moduli **profile, cprofile e pstat**.

Code Profiling

- **cProfile** e **profile** sono due moduli che espongono le stesse interfacce.
- **cProfile** è una versione più efficiente di profile, perché fatta in C e introduce poco overhead.

```
import cProfile
cProfile.run('funzioneDaTestare()', 'fileOutput')
```

Le istruzioni precedenti eseguono e fanno il profiling del codice passato come primo argomento (funzioneDaTestare) e salvano le informazioni nel file fileOutput.

Code Profiling

Per caricare le informazioni salvate, ci viene in aiuto il modulo **pstats**:

```
import pStats
p = pstats.Stats('fileOutput')
```

Le informazioni recuperate sono molteplici e potrebbero essere di difficile visualizzazione. Un modo standard per visualizzare le informazioni utili è il seguente:

```
p.strip_dirs().sort_stats(''time'').print_stats()
```

Il primo metodo rimuove path estranei a tutti i nomi dei moduli Il secondo ordina la tabella restituito rispetto al tempo speso per ogni funzione Il terzo metodo stampa le statistiche risultanti

Code Profiling

L’output che si potrà visualizzare sarà una tabella le cui colonne saranno (in ordine):

- **ncalls**: numero di chiamate eseguita per la data funzione
- **tottime**: tempo totale speso per tutte le chiamate ad una funzione, escluso il tempo speso per le sotto-funzioni chiamate
- **percall**: Tempo medio speso per ogni chiamata della funzione, escluse le sue sotto-funzioni.
- **cumtime**: tempo speso nell’esecuzione di tutte le chiamate della funzione, comprese le sue sotto-funzioni.
- **percall**: Tempo medio speso per ogni chiamata della funzione, comprese le sue sotto-funzioni.
- **filename:lineno(function)**: informazioni relative alla funzione.