Python

Ing. Ovidiu Daniel Barba - Università degli Studi di Roma Tor Vergata

Corso di Ingegneria degli Algoritmi (Parte Pratica)

A.A. 2018/2019

Contatti

- **Ovidiu Daniel Barba** : ovidiudaniel.barba@alumni.uniroma2.eu
- **Luca Pepè Sciarria** : luca.pepesciarria@alumni.uniroma2.eu
  - Inserire [IA18] nell'oggetto

Sito Web del Corso (solo per la parte pratica):

- https://uniroma2-algorithms.github.io/ingegneria-algoritmi-2018/
- Qui troverete le slide e il codice delle lezioni

Orario Lezione:

- Lunedì dalle 14:00 alle 15:45 in Aula 4 (edificio Didattica)

Modalità esame
(Parte Pratica)

La prova pratica influirà sul 25% del voto finale di Ingegneria degli Algoritmi.

- Due prova in itinere
  - Di media complessità

OPPURE

- Una prova finale
  - Di elevata complessità

Le prove possono essere svolte individualmente o in gruppo (max 3 persone).

E' *consigliato* lo svolgimento delle prova in itinere.

Perchè Python?

- Le grandi compagnie lo usano
  - Google, Facebook, Netflix...
- Sintassi chiara e leggibile
- Vasta gamma di librerie
  - Calcolo scientifico, Web, Machine Learning...
- Permette di concentrarsi sulla logica di programmazione
  - Sul cosa fare e non sul come
- Rende più semplice la traduzione della logica di un algoritmo
- Supporta diversi paradigmi di programmazione
  - Imperativo, Procedurale, Object-Oriented e Funzionale

Tool di sviluppo

- **Python** versione 3.7
  - Il vostro sistema potrebbe già avere Python 2.x installato (*python --version*)
  - Verificare l'esito dell'installazione con il comando *python3 --version*
  - Python 2.x e 3.x possono coesistere sullo stesso sistema, basta chiamare i corrispondenti programmi con la cifra *2* o *3* finale (*python2* / *python3*)
- L' **IDE** che preferite
  - Consigliato **PyCharm**
  - Potete ottenere la licenza gratuita di tutti i prodotti JetBrains (incluso Pycharm) con la vostra e-mail istituzionale *<nome>.<cognome>@students.uniroma2.eu*
  - Va bene anche l' IDE integrato con Python

Documentazione

- Per rivedere tutte le funzionalità di base di Python, seguire il tutorial ufficiale
- La documentazione ufficiale di Python (disponibile anche in altre lingue)
- Documentazione nella shell
  - Funzione ***help()*** per visualizzare le informazioni di un particolare modulo o funzione
  - Esempio: ***help(math)*** oppure ***help(math.sqrt)***

Come scrivere il codice

- **Import** dei moduli richiesti dal programma all'inizio
- Inizializzazione di eventuali variabili di modulo
- *Docstring* per le funzioni principali
- Uso di nomi significativi per variabili e funzioni

Convenzione:

- Nome di funzioni, metodi e variabili iniziano con la lettera minuscola

  (variabili costanti scrivere tutto il node in maiuscolo)
- Nomi di classi con lettera maiuscola
- Usare in entrambi i casi **CamelCase**
  - nomevariabile => **nomeVariabile**
  - nomeclasse => **NomeClasse**

1.

Introduzione a Python

Script, moduli e controllo di flusso

I moduli Python

- Un modulo è un particolare script Python
- Sono utili per decomporre un programma di grande dimensioni in più parti oppure per riutilizzare codice scritto precedentemente
- Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) con 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__**
- Qui trovate tutti i moduli base di Python

Comando import

- **import <modulo>** : per importare il modulo specifico
  - >> import math
  - >> math.sqrt(4) # radice quadrata di 4
- **from <modulo> import <risorsa1>, <risorsa2>, ... :** permette di importare solo alcune risorse specifiche da un modulo
  - >> from math import sqrt
  - >> sqrt(4) # non c'è bisogno di specificare in che modulo si trova
- **from <modulo> import * :** importa tutto del modulo
- per importare i propri moduli: **import <path 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():
    pass   # does nothing, null operation
if __name__ == "__main__":
    main()
```

- Questo permette di impostare diversi comportamenti dello script nel caso in cui esso venga importato come modulo oppure eseguito direttamente

Formattazione Stringhe

- Definite le variabili:
  - >>> corso = “Ingegneria Algoritmi”
  - >>> cfu = 9
- **%-Formatting**:
  - >>> “Seguo il corso %s da %d cfu” % (corso, cfu)
- **str.format()**:
  - >>> “Seguo il corso { } da { } cfu”.format(corso, cfu)
  - >>> “Seguo il corso {n} da {c} cfu”.format(n=corso, c=cfu)
- **f-Strings** (da Python 3.6 in poi):
  - >>> f“Seguo il corso {corso} da {cfu} cfu”
  - >>> f“Seguo il corso {corso.lower()} da {cfu + 3} cfu”
  - Output: Seguo il corso ingegneria algoritmi da 12 cfu
  - Stringa sempre preceduta da ‘f’ e dentro le graffe qualsiasi espressione o chiamata a funzione

Accedere agli argomenti passati - 1

- Eseguire uno script con **python3 <file.py> arg1 arg2 arg3** …
- Accedere agli argomenti passati usando il modulo **sys**:
  - >>> import sys
  - >>> print(sys.argv) # all arguments
  - >>> print(sys.argv[0]) # ALWAYS file name
  - >>> print(sys.argv[1]) # arg1
  - >>> print(sys.argv[2]) # arg2
  - *len(sys.argv)* può essere usato per controllare il numero di argomenti passati
  - Gli argomenti passati sono tutti stringhe. Si possono convertire in altri formati
  - >>> num = int(argv[2]) # be sure argument is a number !

Accedere agli argomenti passati - 2

- Eseguire uno script con **python3 <file.py> arg1 arg2 arg3 … [--oarg1 value1 --oarg2 value2 …]**
- --oarg1 sono argomenti opzionali
- Usare modulo **argparse:**
  - >>> import argparse
  - >>> parser = argparse.ArgumentParser()
  - >>> parser.addArgument(“arg1”, help=”arg description”
  - >>> parser.addArgument(“-o”, “--oarg1”, type=int, help=”Description”
  - >>> args = parser.parse_args()
  - >>> if args.oarg1:   # check if optional is set!
  - >>>   # do something with it

Controllo di flusso

- L'istruzione **for** compie un'iterazione sugli elementi di qualsiasi sequenza nell'ordine in cui appaiono nella sequenza.
  - **for <variabile> in <sequenza>:**

    <fai qualcosa>
- Per fermare il ciclo, usare l'istruzione **break**
- Per continuare l'iterazione con il prossimo elemento della sequenza e non eseguire il resto del codice, usare **continue**
- for i in range(20):

  if i > 10:

    break

  if i % 2 == 1:

    continue

  # stampa i numeri pari da 0 a 10
- Istruzione while:
  - **while <condizione>:**

    <fai qualcosa>
- **if <condizione1>:**

  <istruzioni 1>

  ….

  elif <condizioneN>:

  <istruzioniN>

  else:

  <istruzioni default>

2.
Strutture Dati

Liste, Tuple e Dizionari

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
  - >>> l = [] # empty list
  - >>> l.append(1) # add integer
  - >>> l.append(“hello”) # can add different types
  - >>> l.insert(1, 100) # [1, 100, “hello”]
  - >>> l[0] # 1
  - >>> l[1:3] # [100, “hello”]
  - >>> l.pop(0) # removes element at index
  - >>> len(l) # list length
  - >>> l.sort()
  - >>> l.clear() # empties list
  - >>> l3 = l1 + l2 # concatena 2 liste
- Se oltre al valore dell’elemento si vuole anche l’indice durante un’iterazione:
  - **for index, value in enumerate(l):**

    print(f”At position {index} value is {value}”)

List Comprehension

- Modo più conciso per scrivere un ciclo for per creare nuove liste da sequenze già esistenti
- **lc = [<espressione> for <elemento> in <lista> if <condizione>]**
- Esempio: creare una lista con i tutti numeri pari minori di 100
  - l = []
  - for i in range(100):

    if i % 2 == 0:

    l.append(i)
- Invece con list comprehension si ha:
  - ***l = [i for i in range(100) if i % 2 == 0]***

Tuple

- In pratica è una lista immutabile e ordinata di elementi.
- Una volta creata, i suoi valori non possono essere modificati
- >>> t = (1, 2, 3)
- >>> t[0] = 10 # error
- >>> print(t[0]) # 1
- >>> len(t) # 3
- >>> for el in t:

  print(el)

Dizionari

- Struttura che organizza i dati per associazione e non per posizione
  - Un dato è indicizzato da una chiave, che può essere qualsiasi tipo di dato *immutabile*
- Inizializzazione: *d = {}* (vuoto) oppure *d = {1: "Hello", "second": "World"}*
- Per accedere ai valori per una determinata chiave:
  - *<dict>[key]*
- Per aggiungere o sostituire valori:
  - *<dict>[key] = val*
  - Se la chiave non esiste, viene aggiunta altrimenti aggiornato il valore
- Altre operazioni:
  - <dict>.get(<key>, <default>) # se valore di key non esiste, ritorna default
  - <dict>.pop(<key>,<default>)
  - <dict>.values() # restituisce tutti i valori
  - <dict>.keys() # restituisce tutte le chiavi
  - <dict>.items() # restituisce tutte le tuple (chiave, valore)

3.
Programmazione Funzionale

Definizione funzioni, funzioni anonime(lambda), map e filter

Definizione Funzioni

- Una funzione si definisce con la parola chiave def:
- **def funzione(<args>, <key1>=<value1>,..., <keyN>=<valueN>):**

  ”””docstring”””

  codice funzione indentato
- <args> sono i parametri obbligatori
- <key1>=<value1>,..., <keyN>=<valueN> sono i parametri opzionali
  - Se non specificato quando chiamata la funzione, assumono il valore di default <valueN>

```
def f(n, p=1, c=10):
    print(n, p, c)
f(100)                  #  100 1  10
f(100, p=20)            #  100 20  10
f(100, p=20, c=30)      #  100 20  30
```

Funzioni lambda

- **lambda <argomenti> : <espressione>**
- L'istruzione *lambda* crea una funzione anonima senza usare il costrutto *def*
- Deve essere definita su una sola riga e può contenere solo un'espressione
- Numero variabile di argomenti
- Può essere assegnato ad una variabile
- >>> f = lambda x : x ** 2 # eleva x alla seconda
- >>> f(3) # 9
- Le funzioni lambda possono anche essere ritornare da un'altra funzione
    - def potenza(exp):
    - return lambda x : x ** exp
    - f = potenza(3)
    - f(2) # 2 ** 3 = 8
    - potenza(3)(2) # stesso risultato di prima

Map e Filter

- Map e Filter sono funzioni di più alto livello che prendono come argomenti una funzione e una o più sequenze e applica la funzione a tutti gli elementi delle sequenze
- Ritornano una lista
- **map(<funzione>, <seq1>,..<seqN>)**
  - m = map(lambda x : x + 1, [0, 1, 2])
  - l = list(m) # convertire in lista
  - print(l) # [1, 2, 3]

- **filter(<funzione>, <seq>)**
  - <funzione> deve ritornare un booleano
  - <funzione> viene applicata a ogni elemento di <seq>
  - mantiene tutti quei elementi per i quali <funzione> ritorna True
  - f = filter(lambda x: x % 2 == 0, [1,2,3,4])
  - print(list(f)) # [2, 4]

3.

Programmazione a Oggetti ed Eccezioni

Definizione classe, metodi, attributi
Gestione eccezioni

Classi

- La classe è un meccanismo di astrazione per poter definire entità del mondo esterno
  - **class <nomeClasse>(<classeGenitore>):**

    <attributi>

    <definizione metodi>
- Un oggetto è un'istanza di una classe
- Gli attributi descrivono lo stato dell'oggetto
- I metodi sono le operazioni che è possibile eseguire sulle istanze degli oggetti.
- La classe **object** è la classe radice che generalizza tutte le altre
- Tutti i metodi hanno come primo parametro **self**
- I metodi sono definiti come normali funzioni (costrutto def)
- **__init__** è il costruttore e viene chiamato quando un oggetto è istanziato
- **__str__** è un metodo che restituisce una stringa e viene chiamato quando si fa la stampa di un oggetto

Eccezioni

- Gli errori di sintassi vengono sollevati dall'analizzatore sintattico (parser)
- Anche se un'istruzione è sintatticamente corretta, può generare degli errori a runtime (mentre il programma è in esecuzione)
  - Es: divisione per zero (ZeroDivisionError)
  - Es: concatenare tipi di dato diversi (TypeError)
  - Una lista di tutte le eccezioni di Python si può trovare qui

- Esiste la classe Exception
- Si possono definire delle eccezioni non presenti nativamente creando una classe che estende Exception
  - **class MyError(Exception):**

    def __init__(self, mess): # si possono aggiungere altri parametri

    super().__init__(mess)

    # codice custom

>>> error = MyError("Messaggio di errore")

Gestione eccezioni

- Le eccezioni vengono sollevate con l'istruzione **raise**
- Gestire le eccezioni
  - **try:**

    <istruzioni che possono generare eccezioni>

    except <tipo eccezione> as <nome variabile>:

    <istruzioni in caso di eccezione>

    finally:

    <istruzioni eseguite sia in caso di eccezioni che non>

Esempio eccezioni

```
def f(num):
    if num > 10:
        raise MyError("Numero > 10")

try:
    f(20)
except MyError as e:
    print(e)
finally:
    print("Finally")
```

4.
File I/O

Aprire, scrivere e leggere un file

Gestione File

- Per aprire un file f = open(<path>, <modalità>)
  - La path può essere assoluta o relativa
  - Modalità: ‘r’ (read-only), ‘w’ (write-only), ‘r+’ (read and write), ‘w+’(write and read), ‘a+’ (append read and write)
- Operazioni su un file f:
  - f.write(<stringa>): scrive sul file e ritorna il numero dei caratteri scritti
  - f.read(): legge e ne ritorna il contenuto in una stringa
  - f.readline(): legge una riga e la restituisce
  - f.close(): chiude la sessione con il file
- E’ buona norma chiudere la sessione con il file dopo aver completato le operazioni su di esso
- Oppure usare il costrutto **with** (chiama f.close() dopo che le operazioni sono terminate)

with open(<path>, <modalità>) as <nome variabile file>:

<operazioni su file>

5.
Code Profiling

Valutare efficienza del codice

Profiling - Approccio semplice

- Per 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
  - >>> from time import time
  - >>> inizio = time()
  - >>> funzioneDaTestare()
  - >>> tempoTrascorso = time() - inizio
- Approccio semplice ma non molto preciso

Profiling - cProfile e pstats

- Per avere una misurazione più accurata e a grana fine del nostro codice (misure per funzioni e sotto-funzioni) usiamo il modulo cProfile
  - >>> import cProfile
  - >>> **cProfile.run('funzioneDaTestare()', 'fileOutput')**
  - Viene eseguito il codice specificato e le misurazioni vengono scritte su sul file 'fileOutput' (in un formato proprio non leggibile)
- Per recuperare le statistiche dal file usiamo il modulo pstats
  - >>> import pStats
  - >>> **p = pstats.Stats('fileOutput')**
  - >>> **p.strip_dirs().sort_stats("time").print_stats()**
- Maggiori informazioni sul profiling qui