Laboratorio
Setup e primi passi con Python

Programmazione di Applicazioni Data Intensive

Laurea in Ingegneria e Scienze Informatiche
DISI – Università di Bologna, Cesena

Proff. Gianluca Moro, Roberto Pasolini
nome.*cognome*@unibo.it

Outline

- Introduzione e versioni di Python
- Sintassi ed istruzioni di base
- Moduli
- Package
- Uso della libreria standard Python
- Installazione di librerie esterne
- Ambienti virtuali
- Struttura di un progetto Python

Python

- Linguaggio interpretato cross-platform
  - disponibile per i principali SO (Linux, Mac, Windows, …)
  - un'implementazione di riferimento (*CPython*) più altre alternative
  - integrabile in altri linguaggi (C, C++, Java, …)
- Creato alla fine degli anni '80, divenuto popolare nei 2000
- Multi-paradigma
  - imperativo, object-oriented, funzionale
  - sintassi facilmente estendibile ad altri paradigmi
- Enfasi sulla facilità di lettura e scrittura del codice
  - "there should be one—and preferably only one—obvious way to do it"

Python 2 e Python 3

- Sono diffuse due diverse versioni *major* di Python
- Su **Python 2** si basa molto software tutt'ora in uso
  - utilizzato di default in molte distribuzioni Linux e in Mac OS X
  - l'ultima versione *minor* prevista è la 2.7, rilasciata nel 2010
  - il termine del supporto è previsto nel 2020
- **Python 3** introduce novità incompatibili con Python 2
  - prima release nel 2008, ultima versione minor 3.6 del 2016
- Molte librerie di uso comune sono state (ri)scritte per funzionare con entrambe le versioni

Nel corso useremo Python 3

Installazione di Python

- Per Windows e Mac, scaricare la distribuzione più recente di Python dal sito ufficiale `python.org` e seguire le istruzioni
- Su Linux, Python 3 è solitamente già installato oppure disponibile tramite gestore di pacchetti
  - es. su Debian/Ubuntu/Mint: `apt-get install python3`
- Per installare librerie esterne, sono da includere anche il modulo `venv` e l'utility `pip`
  - su Windows e Mac sono inclusi nell'installazione (eventualmente vanno selezionati)
  - su Linux sono solitamente pacchetti separati
    `apt-get install python3-venv python3-pip`

Uso di Python

- È possibile eseguire uno script Python contenuto in un file *.py*
  - con un doppio clic sul file corrispondente (se abilitato nel SO)
  - da linea di comando scrivendo:
    `python3 nomeFile.py`
- In più, Python può essere eseguito in modalità interattiva
  - l'utente digita istruzioni una ad una, l'interprete le valuta e stampa il valore che viene restituito
  - nota in generale come *REPL* (*Read-Eval-Print Loop*)
- Per avviare l'interprete in questa modalità, lanciare il comando `python3` senza specificare un file
- Per uscire dall'interprete interattivo digitare `exit()`
  - oppure (Linux/Mac) Ctrl+D oppure (Windows) Ctrl+Z e Invio

Sintassi di base

- Un'istruzione Python è contenuta di default in una riga

```
print("Hello, world")
```

- Si possono però scrivere più istruzioni in riga separate con ";"

```
print("Hello"); print("world")
```

- I commenti sono introdotti da "#" e finiscono a fine riga

```
# Questo è un commento
print("Hello, world")    # altro commento
```

- Si può far continuare un'istruzione in una riga successiva
  - *esplicitamente* se la riga termina in "\"
  - *implicitamente* se ci sono parentesi non chiuse (più comune)

```
print("Hello, "
   + "world")
```

Sintassi di base: blocchi di codice indentati

- In altri linguaggi i blocchi di codice (usati in costrutti if, for, …) sono delimitati da simboli specifici (spesso "{" e "}")
  - l'indentazione è usata convenzionalmente per migliore leggibilità
- Python usa l'indentazione come sintassi per i blocchi
  - ogni riga che introduce un blocco (es. if) termina in ":"
  - le righe a pari livello sono indentate con pari numero di spazi
  - per indicare un blocco vuoto si usa la parola chiave "pass"

```
// esempio in Java
nums = getNumbers();
for (int x: nums) {
> if (x < 0) {
> > System.out.println(x);
> }
}
System.out.println("end");
```

```
# esempio in Python
nums = get_numbers()
for x in nums:
> if x < 0:
> > println(x)
println("end")
```

Uso dell'interprete interattivo

- Si può usare l'interprete interattivo per valutare espressioni
  - sono supportate le tipiche operazioni tra numeri `+ - * /`

```
>>> 12345679 * 8
98765432
```

- Si possono assegnare i valori a variabili per essere riutilizzati

```
>>> a = 123
>>> b = a + 321
>>> b
444
```

- Si possono utilizzare alcune funzioni predefinite in Python tramite la tipica sintassi *nome*(*argomenti*)

```
>>> abs(-123)
123
```

Definizione di funzioni

- Si possono definire funzioni usando un blocco `def`
  - se una riga inizia un blocco (segnalato da "`:`" alla fine), l'interprete non la esegue e attende invece il contenuto del blocco
  - le righe successive del blocco vanno indentate, tutte con lo stesso numero di spazi (almeno uno, spesso 2 o 4)
  - una riga vuota segnala la fine del blocco, che viene interpretato

```
>>> def double(x):
...     y = 2*x
...     return y
...
```

- Una volta definita, la funzione può essere utilizzata

```
>>> double(10)
20
```

Concetti di base: *namespace*

- Un ***namespace***, usato internamente da Python, è una collezione di elementi associati a nomi univoci
- Ogni volta che si dichiara ad es. una variabile o una funzione, questa è inserita in un namespace
- Esempi di namespace includono
  - l'insieme di variabili e funzioni dichiarate in modalità interattiva
  - l'insieme delle variabili dichiarate in una funzione
- Usando diversi namespace, uno stesso nome può essere utilizzato in contesti diversi senza conflitti
  - possiamo ad es. definire variabili con lo stesso nome in funzioni diverse senza che queste interferiscano

Concetti di base: *scope*

- Uno ***scope*** è un contesto in cui è eseguita un'istruzione
- Ad esempio, una chiamata a funzione crea uno scope in cui sono eseguite le istruzioni della funzione stessa
- Ad ogni scope è associato un namespace *locale* a cui di default ci si riferisce nelle istruzioni
  - dichiarando una variabile in una funzione, il suo valore viene inserito nel namespace creato per la chiamata alla funzione
- Ci si può però anche riferire direttamente a oggetti nei namespace degli scope "esterni" a quello corrente
- Il namespace più esterno è sempre quello delle funzioni e degli altri oggetti definiti da Python (*builtin*)
  - per questo è sempre possibile chiamare funzioni predefinite come `abs`, se non "nascoste" da variabili locali con lo stesso nome

Moduli

- I ***moduli*** costituiscono il meccanismo usato in Python per l'utilizzo di codice definito esternamente
- Ogni modulo definisce un proprio namespace *globale* in cui definire variabili, funzioni, ecc., isolato dagli altri moduli
  - diversi moduli possono usare gli stessi nomi senza interferenze
- All'avvio dell'interprete Python, viene creato un modulo principale in cui sono inseriti gli oggetti dichiarati
  - in modalità interattiva, questo contiene gli oggetti creati nelle istruzioni eseguite man mano dall'utente
  - eseguendo un file, contiene gli oggetti dichiarati nel file stesso
- Tramite l'*importazione*, è possibile da un modulo richiamarne altri ed utilizzarne le funzionalità

Creare un file modulo

- Un file `.py`, oltre ad essere eseguito direttamente, può essere usato come modulo da un altro script
- Si prenda ad esempio un file `mymodule.py` in cui sono definite delle funzioni:

```
def factorial(n):
    return n*factorial(n-1) if n>1 else 1
def fibonacci(n):
    return (fibonacci(n-1) + fibonacci(n-2)
            if n>1 else 1)
```

- Eseguendo direttamente questo file, non accade nulla
  - viene dichiarata una funzione, ma non viene usata in alcun modo

Importare un modulo

- Lanciando un interprete Python dalla stessa directory in cui è salvato il file, è possibile *importarlo* come modulo

```
>>> import mymodule
```

Eseguendo questa istruzione

1. viene creato in memoria un nuovo modulo "`mymodule`" con un namespace inizialmente vuoto
2. il file `mymodule.py` viene interpretato e gli oggetti dichiarati in esso sono salvati nel nuovo namespace
3. nel namespace corrente (il modulo principale) viene inserito con nome "`mymodule`" il riferimento al modulo caricato

Usare un modulo importato

- Una volta importato, è possibile accedere agli oggetti di un modulo con la sintassi "*`modulo.oggetto`*"
- Si può quindi usare la funzione `factorial` così:

```
>>> mymodule.factorial(4)
24
```

- La funzione `dir` fornisce la lista di nomi definiti in un modulo
  - sono inclusi alcuni nomi speciali, ad es. "`__name__`" è un attributo il cui valore è il nome del modulo stesso

```
>>> dir(mymodule)
```

Uso di import

- Un modulo viene sempre importato nel namespace locale
  - usando `import` al di fuori delle funzioni (caso più comune), il modulo è importato nel namespace globale del modulo
  - si può però usare `import` anche dentro ad una funzione per importare il modulo solo in essa
  - lo stesso modulo può essere importato più volte in diversi namespace, ma è inizializzato solo la prima volta
- Con la clausola `as` si può cambiare il nome usato per riferirsi al modulo importato nel namespace corrente
  - **non** cambia il nome interno del modulo

```
>>> import mymodule as m
>>> m.factorial(5)
120
```

from … import

- Con la forma `from … import` si può importare direttamente un oggetto da un modulo nel namespace corrente

```
>>> from mymodule import factorial
>>> factorial(6)
720
```

- Si possono importare più oggetti dallo stesso modulo

```
>>> from mymodule import factorial, fibonacci
```

- Usando "*" si possono importare nel namespace corrente tutti gli oggetti di un modulo

```
>>> from mymodule import *
```

  - si può usare per rapidità in modalità interattiva, ma è **sconsigliato** altrimenti (c'è rischio di importare oggetti con stesso nome di altri)

Eseguire un modulo

- Normalmente, il comando `python3` esegue uno script o modulo dato il nome (o percorso completo) del suo file
- Usando l'opzione `-m`, è possibile eseguire un modulo dato invece il suo nome
- L'opzione `-m` è utile soprattutto per eseguire codice definito in librerie esterne, piuttosto che in un file scritto dall'utente
- Aggiungendo altri argomenti, questi sono passati come parametri al modulo
  - i parametri accettati dipendono dal modulo specifico
  - è di solito possibile ottenere un elenco dei parametri accettati passando "`-h`" o "`--help`" come argomento

```
$ python3 -m mymodule arg1 arg2
```

Package

- I ***package*** consentono di organizzare i moduli in una gerarchia
- Ogni package può contenere moduli e/o altri package
  - è comune che ogni libreria esterna sia contenuta in un package "principale" che contiene una propria gerarchia di package e moduli
- Un package è costituito da una directory contenente un file `__init__.py`, che contiene il codice per inizializzarlo
  - il file **deve** esistere, ma può essere vuoto
- La directory può contenere moduli (file `.py`) e altri package (directory con file `__init__.py`)
- Ogni package e modulo ha un nome completo costituito dalla sequenza dei nomi dei package, separati da "`.`"

Esempio di package strutturato

- La seguente gerarchia di file e directory definisce un package "sound" contenente una gerarchia di package e moduli

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file formats
            __init__.py
            wav.py
            aiff.py
            ...
      filters/                  Subpackage for filters
            __init__.py
            equalizer.py
            vocoder.py
            ...
```

Importare moduli dai package

- Si può importare un modulo da un package col suo nome completo e riferirsi ad esso sempre col nome completo

```
>>> import sound.wav
>>> audio = sound.wav.load("audio.wav")
```

- …oppure si può importare il modulo nel namespace locale

```
>>> from sound import wav
>>> audio = wav.load("audio.wav")
```

- Si può comunque importare con un nome a scelta

```
>>> import sound.wav as w
>>> audio = w.load("audio.wav")
```

Libreria standard di Python

- La libreria standard fornisce un insieme di moduli con funzionalità di uso comune
- Questi moduli possono essere importati con l'istruzione `import` e usati allo stesso modo di quelli creati dall'utente
- La documentazione di Python include informazioni approfondite su tutti i moduli della libreria standard

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

Esempio d'uso della libreria standard

- Il modulo `math` della libreria standard contiene diverse funzioni matematiche comuni
  - `sqrt` (radice quadrata), `log2` (logaritmo in base 2), …
- Possiamo importare il modulo intero ed accedere alle funzioni al suo interno…

```
>>> import math
>>> math.sqrt(25)
5.0
```

- …così come possiamo importare le singole funzioni a cui siamo interessati e invocarle direttamente

```
>>> from math import log2
>>> log2(256)
8.0
```

Esempi di moduli della libreria standard

- `collections`: strutture dati aggiuntive
- `re`: espressioni regolari
  - trova/sostituisci pattern in stringhe
- `datetime`: rappresentazione di date e orari
- `random`: generazione di numeri casuali
- `math`: funzioni matematiche
- `csv`: lettura/scrittura tabelle CSV (*Comma Separated Values*)
- `threading`: avvio di thread di esecuzione paralleli
- `argparse`: interpretazione di opzioni da linea di comando

Librerie esterne

- Le funzionalità di Python possono essere estese oltre quelle della libreria standard tramite librerie di terze parti
- Ogni libreria fornisce nuovi moduli e package utilizzabili all'interno dei propri programmi
- Ogni libreria può a sua volta dipendere da altre librerie
- Il *Python Package Index* (PyPI) è un database online di oltre 100.000 librerie disponibili per Python

https://pypi.python.org/

pip

- L'utility `pip` inclusa in Python può essere usata da linea di comando per installare package da PyPI
- Ad es., per installare la libreria NumPy usare dal terminale (**non** da Python) il comando:

```
$ pip install numpy
```

  - si può indicare una versione specifica (es. "`numpy==1.14`")
  - è possibile specificare più package insieme
- Se una libreria dipende da altre, queste sono installate automaticamente
- Una libreria può essere installata
  - a livello di sistema (servono diritti di amministratore)
  - solo per l'utente corrente (con l'opzione "`--user`")

Ambienti virtuali

- Ogni progetto Python può richiedere librerie diverse e anche versioni diverse della stessa libreria
- Un *ambiente virtuale* rappresenta una collezione di librerie installate indipendente da quelle installate nel sistema
  - gli ambienti virtuali condividono solo la libreria standard
- Creando un ambiente virtuale specifico per ciascun progetto, si evitano conflitti tra versioni dei package da utilizzare
- Un ambiente virtuale è contenuto in una directory, in cui sono organizzati i file eseguibili e le librerie installate in esso

Creare un ambiente virtuale

- In Python 3 è stato introdotto il modulo `venv` per la creazione di ambienti virtuali
  - in Python 2 si usa un comando `virtualenv` separato
- Eseguendo il modulo `venv`, si inizializza un nuovo ambiente in una directory specificata (che viene creata se necessario)

```
$ python3 -m venv myvenv
```

- Nell'ambiente virtuale sono inserite nuove copie dell'interprete Python e dei programmi collegati (es. `pip`)
  - nella sottodirectory `bin` in Linux e Mac OS X
  - nella sottodirectory `Scripts` in Windows
- Questa copia di Python ha accesso alla libreria standard, ma inizialmente a nessuna libreria esterna

Utilizzare un ambiente virtuale

- Usando la copia di `pip` nella directory dell'ambiente, si possono installare librerie in esso invece che nel sistema

```
$ myvenv/bin/pip install numpy          (Linux / Mac)
> myvenv\Scripts\pip install numpy      (windows)
```

- Lanciando l'interprete Python presente nella stessa directory, si potranno usare le librerie installate
- L'ambiente virtuale fornisce anche uno script `activate` per rendere default l'uso degli eseguibili in esso

```
$ source myvenv/bin/activate            (Linux / Mac)
> myvenv\Scripts\activate.bat           (windows)
```

- È così possibile avviare `python` e `pip` omettendone il percorso

```
(myvenv) $ pip install numpy
```

Struttura di un progetto Python

- Per essere distribuite ad altri, applicazioni e librerie Python create dagli utenti seguono una struttura tipica
  - questa struttura non serve per semplici script autocontenuti in un file
- Tutti i moduli della libreria sono tipicamente contenuti in un package apposito (eventualmente strutturato in sottopackage)
- Un file `requirements.txt` elenca le librerie esterne richieste dal progetto, che possono essere installate con `pip`:

```
$ pip install -r requirements.txt
```

- Altri file presenti comunemente includono:
  - `README[.*]`: informazioni generali sul progetto
  - `setup.py`: script d'installazione (necessario per progetti su PyPI)

```
myapp/
  myapp/
    __init__.py
    gui.py
    (altri moduli…)
  requirements.txt
  README
```