Python vs C, C#, Java: alcune riflessioni

Alberto Montresor

Il dibattito su quale linguaggio utilizzare per introdurre la programmazione va avanti da sempre, a livello universitario ma non solo. Se trent'anni fa Pascal faceva da padrone, attorno agli anni 2000 è stato spodestato da Java. A partire dal 2014, Python è diventato il linguaggio più popolare nei corsi introduttivi delle università americane: il 70% dei migliori Dipartimenti di Informatica USA utilizzano Python nei propri corsi introduttivi alla programmazione. [Guo14]

Nella scelta del linguaggio, è necessario considerare due categorie distinte:

- studenti che non hanno (ancora) scelto una carriera informatica - studenti delle scuole superiori, studenti universitari in ambiti diversi dall'informatica, ma anche studenti universitari americani all'inizio del bachelor, prima di scegliere il major (l'area principale di studio)
- studenti che hanno scelto esplicitamente una carriera informatica, come ad esempio gli studenti delle lauree in informatica e ingegneria informatica in Italia.

Per il primo gruppo (chiamiamolo NonCS), molti articoli nel campo della didattica dell'Informatica puntano nella direzione di Python. Molti di questi lavori sono rapporti quasi aneddotici di passaggi da linguaggi come C e Java verso Python, quasi tutti coronati da successo (vengono chiamati "articoli Marco Polo": *been there, seen that*); esempio interessanti studi più seri, con un'analisi quantitativa di approcci differenti, sono gli articoli di Koulori, Lauria e Macredie [KLM15] e di Wainer e Xavier [WX18], che mostrano chiari miglioramenti nella capacità di risolvere problemi utilizzando Python, con una riduzione significativa di errori sintattici e di compilazione.
Esistono anche alcuni lavori dedicati esplicitamente alla programmazione nelle scuole superiori; si vedano ad esempio i lavori dell'Università di Turku [GPBS06], [MPS06] e un lavoro in Italiano di Maurizio Boscaini [Bos09], tutti a favore di Python.

Per quanto riguarda informatici e ingegneri informatici (CS), il dibattito è ancora aperto. Molte università continuano ad utilizzare C e Java come linguaggi iniziali, forti anche del fatto che questi linguaggi vengono presentati in corsi molto più vasti ed intensivi (ad es., 96 ore per la sola programmazione nel primo semestre presso UniTN, in C - corrispondenti a circa tre anni di informatica presso un Liceo Scientifico). Esistono quindi anche articoli che non consigliano l'uso di Python a livello universitario, quale ad esempio un articolo di Hunt che critica l'approccio a livello troppo "alto" di Python rispetto alle caratteristiche fisiche della macchina [Hun15].

La mia opinione personale è che Python ben si adatta alle scuole superiori, in modo particolare nei licei scientifici opzione scienze applicate, perché permette di concentrarsi sui concetti fondamentali della programmazione (sequenza di comandi, ripetizioni, istruzioni condizionali, modularità, ricorsione) e di porre l'accento sul problem solving piuttosto che sui dettagli della sintassi. La possibilità di integrarsi con un insieme amplissimo di librerie permette poi di estendere il suo uso a casi applicativi realistici, fornendo così agli studenti un senso di "rilevanza" che l'uso di linguaggi di più basso livello non danno, e questo lo rende interessante anche per istituti tecnici.

Nel seguito di questo documento riporto alcune considerazioni ulteriori derivanti dalla lettura della bibliografia riguardo alla scelta del primo linguaggio di programmazione e relativamente a Python.

Linguaggi di programmazione: come scegliere

Mannila e de Raadt [MdR06] individuano in Python uno dei candidati più forti come primo linguaggio di programmazione, stilando un elenco di caratteristiche che un linguaggio di programmazione introduttivo dovrebbe avere e confrontandosi con articoli simili.
Fra i criteri di scelta più interessanti, sono citati la semplicità della sintassi, meccanismi semplici per realizzare input/output, la capacità di fornire feedback immediato tramite meccanismi di scripting e interfaccia a linea di comando, un ecosistema di librerie facili da installare ed utilizzare.

Una rassegna sistematica della letteratura su questo argomento è presentata da Moone e Mooney [MM17], i quali confrontano linguaggi visuali (tipo Scratch) e linguaggi testuali, e si interrogano su quale sia l'approccio più adatto. Vengono poste due domande:

- ci sono benefici nell'imparare a programmare tramite un linguaggio visuale rispetto ad un linguaggio testuale?
- la scelta del primo linguaggio ha effetti sulla capacità di imparare a programmare? Quale linguaggio scegliere?

Alcune conclusioni tratte dall'articolo:

- L'approccio visuale funziona bene nei primi anni di istruzione;
- Studenti che sono stati esposti ad un approccio visuale nei primi anni di istruzione sembrano trarne vantaggio quando passano ad un approccio testuale;
- Gli studenti delle superiori sono più interessati ad un approccio testuale; introdurre approcci visuali a questo livello non sembra avere benefici, anche se paiono interessanti alcune sperimentazioni con linguaggi ibridi, che permettono di utilizzare sia forme visuali che testuali;
- Per i corsi universitari di livello base, la scelta di Python sembra la più indicata, ma risultati interessanti si ottengono anche con altri linguaggi (C, C++, Java)

Python: la possibilità di "contestualizzare" la programmazione dando un senso di "rilevanza"

Nel corso dei loro studi, gli studenti CS studiano diversi linguaggi di programmazione (presso UniTN: C, C++, Java, ML, Python); applicheranno poi le loro conoscenze in un gran numero di progetti applicativi, scegliendo di volta in volta il linguaggio più adatto.

Gran parte degli studenti NonCS studiano un solo linguaggio; se possibile, questo linguaggio deve dare loro la possibilità di esprimersi in maniera rilevante per i propri interessi ed essere immediatamente applicabile a problemi realistici.

In questo senso, un'esperienza interessante è quella svolta presso Georgiatech, la principale università della Georgia, che ha deciso, fin dal 1999, che tutti gli studenti (CS e NonCS) devono seguire un corso di programmazione [Guz09]. Nel periodo 1999-2003, questo corso era lo stesso per tutti, ed aveva un problema di voti bassi e scarso interesse. Gli studenti descrivevano il corso come "irrilevante".

A partire dal 2003, i corsi sono diventati tre: uno per informatici (Scheme), uno per gli studenti di altre discipline ingegneristiche (Matlab), uno per gli studenti dei college di Liberal Arts, Management e Architecture (Python). Particolarmente interessante è il corso per le discipline più umanistiche; non solo era basato su Python, ma si concentrava su Media Computation: in altre parole, partendo dai dati e dalle immagini, mostrava come utilizzare Python per trattare immagini, audio, video e testo. Rispetto alla versione precedente, il corso ha suscitato un enorme interesse tra gli studenti, risolvendo al contempo il problema relativo ai voti.

L'enorme insieme di librerie di Python permette di applicare la programmazione ad un numero enorme di aree applicative; personalmente, ritengo che la possibilità di applicare il linguaggio a problemi pratici non sia da sottovalutare.

Python: una sintassi molto compatta

Python ha una sintassi compatta e molto simile al linguaggio naturale, che permette di concentrarsi sui concetti di programmazione (sequenza, ripetizione, istruzioni condizionali, modularità, ricorsione) piuttosto che perdere tempo nella descrizione di sintassi complesse.

Si confronti nella tabella sotto la scrittura di Hello World in C, C++, Java, Python. Ovviamente è un esempio banale, ma mostra come Python richieda di scrivere molto meno di altri linguaggi. Inoltre, non richiede di anticipare e/o nascondere concetti più avanzati quali l'inclusione di moduli, il concetto di classe, i modificatori di visibilità, etc.

`#include <stdio.h>`
 	
`int main()`
`{`
`  printf("Hello   World!");`
`}`
	`#include <iostream.h>`
	
	`int main()`
	`{`
	`  cout << "Hello World!";`
	`}`

`class Hello {`
`  public static void main(String args[])`
`  {`
`  System.out.println("Hello World!");`
`  }`
`}`
	`print("Hello World")`

Python è molto più vicino al linguaggio naturale e al linguaggio matematico. Per esempio, si consideri un costrutto come il seguente, che stampa tutti i colori di di un lista:

```
colors = ["red", "green", "blue"]
for color in colors:
```

```
  print(color)
```

La versione C è molto più complessa e richiede di descrivere un elevato numero di concetti:

```
# include <stdio.h>

int main(int argc, char *argv[])
{
  char *colors[] = { "red", "green", "blue", };
    for (int i = 0; i < 3; ++i) {
      printf("%s\n", colors[i]);
    }
}
```

Python: alcuni svantaggi

Fra le incomprensioni più comuni per chi inizia a programmare [KPEH10], due sono potenzialmente aggravate da Python e vanno trattate con attenzione.

Dynamic typing

Python è dynamically typed, ovvero il tipo delle variabili non viene specificato ma viene determinato a runtime. Per un programmatore novizio, questo è potenzialmente uno svantaggio. Si consideri un classico esempio di funzione per il calcolo del fattoriale di un numero.

```
def factorial(n):
    tot = 1
    for i in range(2,n+1):
        tot = tot * i
    return tot
```

Se chiamato con un valore float (`factorial(5.0)`), il codice restituisce errore a runtime.
Il problema si aggrava qualora si riescano ad introdurre strutture dati più complesse ma con simile sintassi, quali liste e dizionari; ad esempio, `A["name"]` può avere senso se `A` è un dizionario, ma genera un errore a runtime se `A` è una lista.

Alcuni studi sembrano indicare che come linguaggio introduttivo, questo non comporta particolari problemi durante l'apprendimento del linguaggio. Questo è confermato anche da alcuni studi che hanno analizzato il passaggio da un corso CS1 basato su Python ad un corso CS2 basato su C++ [EPM09].

Personalmente, ritengo che debba essere comunque fatta particolare attenzione nel descrivere il concetto di tipo; suggerisco di sfruttare le funzioni predefinite `type()` e **isistance()** per rendere i tipi "visibili" e "tangibili".

Evoluzioni più recenti di Python (dalle 3.6 in poi) permettono di definire il tipo di variabili e parametri; essendo un linguaggio interpretato, tuttavia, queste definizioni hanno effetto solo in

combinazione con tool "linter", ovvero tool integrati nelle IDE che cercando di individuare problemi sintattici, di tipi, etc. prima dell'esecuzione.

```
def factorial(n: int) -> int:
    tot: int = 1
    for i in range(2, n+1):
        tot = tot * i
    return tot
```

Personalmente, ritengo che sia opportuno attendere che i meccanismi utilizzati per specificare tipi diventino più stabili (mypy, il linter associato, è ancora in beta).

Cicli troppo "semplici"

Spesso e volentieri, i cicli espressi con il costrutto **`for`** di Python possono essere fin troppo semplici: spesso e volentieri nascondono l'uso degli indici nelle liste e si corre il rischio di non sapere cosa "sta sotto il cofano". Si considerino questi esempi di codice per sommare il contenuto di una lista **`L`**:

`tot = sum(L)`	`tot = 0`
	`for v in L:`
	`  tot = tot + v`
		`tot = 0`
		`for i in range(len(L)):`
		`  tot = tot + L[i]`
			`tot = 0`
			`i=0`
			`while i < len(L):`
			`  tot = tot + L[i]`
			`  i = i +1`

Da sinistra a destra, i metodi vanno dal più "pitonico" al meno "pitonico". Introdurre il metodo meno pitonico per primo permette di capire cosa sta succedendo, per poi apprezzare la semplicità con cui vengono espressi i cicli in Python e la ricchezza della libreria standard.

Un motivo ulteriore per introdurre il ciclo **`while`** è che l'utilizzo troppo spinto delle caratteristiche di Python nasconde soluzioni computazionalmente inefficienti. Il codice seguente salva in **`common`** gli elementi comuni a due liste **`A`** e **`B`**:

```
A = [1,2,3,4,5,6]
B = [1,5,6,7]
common = []
for a in A:
  for b in B:
    if a==b:
     common.append(a)
```

può essere semplificato ancora di più utilizzando *list comprehension*:

```
common = [a for a in A for b in B if a==b]
```

La complessità dei due codici proposti è *O(mn)*, dove *m* e *n* sono le lunghezze delle liste. Se sappiamo che le due liste sono ordinate, questo codice non è computazionalmente efficiente,

perché lo stesso problema può essere risolto in tempo *O(m+n)*. Per fare questo, è però necessario tornare ad utilizzare cicli while e indici.

```
indexA = 0
indexB = 0
common = []
while indexA < len(A) and indexB < len(B):
  if A[indexA] < B[indexB]:
    indexA = indexA+1
  elif A[indexA] > B[indexB]:
    indexB = indexB+1
  else:
    common.append(A[indexA])
    indexA = indexA+1
    indexB = indexB+1
```

Ovviamente, esiste una versione pitonica "efficiente" (ma che utilizza una grande quantità di memoria):

```
common = list(set(A)|set(B))
```

che trasforma le due liste in insiemi, in tempo O(m+n). Poiché gli insiemi sono memorizzati come hashtable, è possibile calcolare l'intersezione in tempo O(max(m),max(n)). La trasformazione in lista prende di nuovo O(m+n), e quindi il costo è limitato superiormente da O(m+n). Si noti che le due liste vengono replicate in memoria sotto forma di insiemi.

Bibliografia

[Bos09] Maurizio Boscaini. *L'informatica e Python al Liceo: conoscere, pensare, progettare, realizzare*. Didamatica 2009. [PDF]

[EPM09] Richard J. Enbody, William F. Punch, and Mark McCullen. *Python CS1 as preparation for C++ CS2*. SIGCSE Bull. 41(1): 116-120. 2009. [PDF]

[GPBS06] Linda Grandell, Mia Peltomäki, Ralph-Johan Back and Tapio Salakoski. *Why Complicate Things? Introducing Programming in High School Using Python*. In Proc. of the 8th Australasian Conference on Computing Education (ACE'06). 52:71-80, 2006 [PDF]

[Guo14] Philip Guo. *Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities*. Communication of the ACM, 2014(7). [Link]

[Guz09] Mark Guzdial. *Teaching computing to everyone*. Commun. ACM 52(5):31-33, May 2009. [PDF]

[Hun15] John M. Hunt. *Python in CS1 - Not*. In J. Comput. Sci. Coll. 31(2):172-179. 2015. [Link]

[KLM15] Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. 2014. *Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches*. Trans. Comput. Educ. 14(4):26, 2015. [PDF]

[KPEH10] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. *Identifying student misconceptions of programming*. In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). 2010. [PDF]

[MdR06] Linda Mannila, Micahel de Raadt. *An objective comparison of languages for teaching introductory programming*. In Proceedings of the 6th Baltic Sea Conference on Computing education research. 2006. [PDF]

[MPS06] Linda Mannila, Mia Peltomäki and Tapio Salakoski. *What about a simple language? Analyzing the difficulties in learning to program*. In Computer Science Education, 16(3):211-227, 2006 [PDF]

[NM18] Mark Noone, Aidan Mooney. *Visual and Textual Programming Languages: A Systematic Review of the Literature*. J. Comput. Educ. 5: 149, 2018. [PDF]

[WX18] Jacques Wainer and Eduardo C. Xavier. *A Controlled Experiment on Python vs C for an Introductory Programming Course: Students' Outcomes*. ACM Trans. Comput. Educ. 18(3):12. [PDF]