Informatica per Statistica
Riassunto della lezione del 16/10/2013

Igor Melatti

Introduzione alla sintassi del linguaggio Python

- Caratteristiche fondamentali della *sintassi* del linguaggio Python
  - cioè come si fa a scrivere un programma Python di modo che poi l'interprete Python lo accetti senza dare errori (di sintassi, appunto)
- Innanzitutto, ci sono 2 modi per usare l'interprete Python: interattivamente e facendogli eseguire un programma
  - tutto quello che si può scrivere su un programma Python può essere scritto, istruzione per istruzione, nell'interprete
  - in più, l'interprete può valutare e stampare i valori delle espressioni (vedere più avanti)
  - per lanciare l'interprete interattivo, basta scrivere `python` su un prompt di comandi
  - per lanciare l'interprete su un programma scritto a parte e salvato su un file `programma.py`, basta scrivere `python programma.py` su un prompt di comandi
    - in questo modo, l'intero programma.py viene eseguito tutto in una volta
    - in alternativa, si può avviare l'interprete interattivo e copiarvi dentro ciascuna istruzione del programma, per vederlo eseguito passo passo (una istruzione per volta)
- Un programma (scritto in linguaggio) Python è costituito da una successione di *istruzioni* (vedere esempio in Figura 1)
  - in Figura 1 sono presenti 8 istruzioni
- In generale, le istruzioni possono essere di uno dei seguenti tipi (prima parte, l'argomento verrà ripreso in lezioni seguenti):

  istruzione vuota ovvero `None`; non fa nulla, ma in alcuni casi è utile

```
1 operazione = raw_input("Immettere '+' per addizione, qualsiasi altro
   carattere per la sottrazione: ")
2 print "Immettere il primo operando: "
3 primo_operando = input()
4 secondo_operando = input("Immettere il secondo operando: ")
5 if (operazione == '+'):
6   print str(primo_operando) + " + " + str(secondo_operando) + " = " + str
     (primo_operando + secondo_operando);
7 else:
8   print str(primo_operando) + " - " + str(secondo_operando) + " = " + str
     (primo_operando - secondo_operando);
```

Figure 1: Un semplice programma Python

- **Attenzione**: il Python è *case-sensitive*, ovvero due parole scritte con le stesse lettere ma con diversi *case* (maiuscola/minuscola) sono parole diverse
- quindi scrivere `none` (o `nOne`) è *diverso* da scrivere `None`

assegnamento è del tipo <`nome_var`> `=` <`espressione`>

- tutto ciò che viene scritto tra < e > può essere sostituito con un elemento della relativa *categoria sintattica*
- tutto il resto (parentesi tonde, virgole, punti, uguaglianze etc) va messo esattamente com'è
- nota: tra una categoria sintattica e l'altra ci possono essere tutti gli spazi che si vuole
  - per "spazi" si intende lo spazio vero e proprio (tasto centrale in basso di ogni tastiera) o il tab (tasto di tabulazione, tipicamente a sinistra della Q sulla tastiera)
  - ad esempio, tra `operazione` e `=` ci potrebbero anche essere 2 spazi, una tabulazione, un altro spazio, e il programma sarebbe corretto ugualmente
  - non è invece corretto inserire spazi all'interno delle categorie sintattiche, ad esempio è sbagliato scrivere `opera zione`
  - fa eccezione l'indentazione, ovvero gli spazi a sinistra (vedere più sotto)
- <`espressione`> può essere complicata a piacere, contenere sia operazioni (somma, prodotto, ...) che chiamate a funzioni (vedere più avanti), coinvolgere sia costanti che variabili (anche la stessa variabile che si trova a sinistra dell'uguale)
- nei casi più semplici, può essere solo una costante o solo una variabile
- ad esempio, le righe 1, 3 e 4 di Figura 1 sono degli assegnamenti: rispettivamente, <`nome_var`> è `operazione`,`primo_operando` e

`secondo_operando`, mentre <`espressione`> consiste in tre chiamate a funzione: rispettivamente, a `raw_input` e ad `input` (2 volte)

- è importante a questo punto distinguere tra *variabili* e *costanti*
- una ***variabile*** è una stringa (sequenza di caratteri) che sia un *identificatore*
- cioè, qualcosa che identifichi univocamente quella variabile, dandogli un nome
- la regola per scrivere un identificatore è semplice: dev'essere una sequenza di simboli, ciascuno dei quali può essere una lettera (non accentata: dev'essere ASCII...), una cifra o il carattere _ (*underscore*)
- ma non può cominciare con una cifra
  - identificatori validi: `ciao3`, `ciao`, `c3i1a245o`, `_c3i1a2_45o`
  - identificatori non validi: `3ciao3`, `3_ciao`, `c3i1+a245o`, `_c3i1a2-45o`
- qualsiasi nome di variabile può essere usato a sinistra di un assegnamento (invece, non si possono usare costanti od espressioni *a sinistra* di un assegnamento)
- una ***costante*** è un qualsiasi numero (con o senza segno, con o senza virgola, se con virgola in virgola mobile o fissa) o una sequenza qualsiasi di caratteri racchiusi tra “” o ” (*stringa*)
  - questo in realtà non esaurisce tutte le possibili costanti: ci si ritornerà
  - `2` è un numero intero, `2.0` è un numero reale in virgola fissa, `22.0e+1` è un numero reale in virgola mobile (uguale a $22 \times 10^1 = 220$)
  - `"ciao mondo!"` è una stringa, e lo è anche `'ciao mondo!'` (queste due stringhe costanti sono effettivamente uguali)
  - alcuni caratteri speciali delle stringhe sono indicati con il carattere backslash, ovvero \
  - ad esempio \`n` (*newline*, andata a capo), \`t` (*tabulation*, tabulazione), \\ (*backslash*, se si vuole scrivere il backslash stesso): provare ad eseguire `print "ciao\nvado a capo\tquesto lo scrivo dopo un tab"`
  - i backslash servono anche a mettere dentro una stringa i caratteri `"` e `'`; ad esempio `"questa stringa contiene sia \" che '"` che è uguale a `'questa stringa contiene sia " che \''`
- è a questo punto necessario definire meglio cos'è un *tipo*
  - per i nostri scopi, un tipo è un insieme di valori più un certo numero di operazioni su tali valori

* quasi tutti i tipi fanno riferimento ad insiemi concettualmente infiniti, ma praticamente finiti (anche se enormi)
* per ora, verranno considerati i seguenti tipi di Python: gli interi, i reali (o meglio, razionali) e le stringhe
* gli interi sono un sottinsieme di $\mathbb{Z}$ (un sottoinsieme molto grande, limitato solo da quanta memoria c'è sul computer, disco compreso)
* i razionali (o *reali a precisione finita*) sono sottinsiemi di $\mathbb{Q}$
* le stringhe sono sottoinsiemi di A^+, dove A contiene tutti i simboli stampabili: $A = \{$`a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`, `,` , `.`, `(`, `)`, `+`, ...$\}$. Da notare che A è finito (e anche non molto esteso, ha un centinaio di elementi), mentre A^+, che è l'insieme di tutte le sequenze finite di simboli in A, è infinito
  - quanto siano grandi questi sottoinsiemi, dipende dalla quantità di memoria (anche su disco...) della macchina
  - si considerino gli interi: questo vuol dire che si tratta di numeri talmente grandi da essere quasi uguali a tutto $\mathbb{Z}$. Infatti, considerando anche solamente il numero di bytes contenuti in RAM (lo si chiami B), il sottoinsieme rappresentabile di $\mathbb{Z}$ risulta $[-2^{8B-1}, 2^{8B-1} - 1]$. Se B è nell'ordine dei GB, come spesso succede, si ha $B > 2^{30}$, e quindi $2^{8B-1} - 1 > 2^{2^{30}} = 2^{2^{10} \times 2^{10} \times 2^{10}} > 2^{10^3 \times 10^3 \times 10^3} = 2^{10^9} = 2^{100 \times 10^7} = 2^{20 \times 5 \times 10^7} > 10^{6 \times 5 \times 10^7} = 10^{3 \times 10^8}$, e l'ultimo è un numero fatto da circa un miliardo di cifre decimali
  - giusto per avere un'idea: un googol è solo 10^{100}, ed è già maggiore della stima per il numero di atomi osservabili nell'universo
  - con un ragionamento simile, si può far vedere che sono rappresentabili quasi tutti i razionali (e sicuramente tutti quelli che interessano nelle applicazioni pratiche)
  - tuttavia, usare gli interi enormi è facile, mentre usare i razionali precisi richiede qualche accorgimento in più, dato che quelli definiti per default sono invece molto limitati: per motivi storici, non vanno oltre 10^{308}, né sotto 10^{-325} (per approfondimenti, vedere IEEE 754)
  - per usare razionali a precisione virtualmente illimitata occorre importare un opportuno modulo (`decimal`), se ne riparlerà
  - niente da fare per i reali "veri": per rappresentare π od e o $\sqrt{2}$ servirebbero infinite cifre decimali, ma i computer sono finiti... però si hanno ottime approssimazioni
* le operazioni definite sui tipi numerici sono quelle aritmetiche usuali

    * è però da notare che la divisione ha significato diverso a seconda che sia applicata su interi o su razionali: nel primo caso si parla della *divisione intera*, ovvero dell'operazione di *quoziente*, nel secondo della divisione frazionaria
      · `5/2` dà come risultato 2, mentre `5.0/2.0` (o anche `5/2.0`, o `5.0/2`) dà come risultato 2.5
      · si possono anche fare *cast* espliciti: `float(`v`)` "trasforma" v da intero a razionale (a virgola mobile, ovvero *float(ing point)*); `int(`v`)` fa la trasformazione inversa (restituendo $\lfloor v \rfloor$, quindi niente arrotondamento)
      · quindi, `float(5)/2` fa ancora 2.5; quanto fa `float(5/2)`? e `int(5.0)/2`?
    * si possono fare facilmente anche elevamenti a potenza, usando `**`
    * il significato delle operazioni cambia al cambiare dei tipi: mentre `+` è la normale somma per tutti i tipi numerici, se applicata a stringhe fa la *concatenazione*: `"ciao" + "a dopo"` fa `"ciaoa dopo"`
    * analogamente si può moltiplicare una stringa per un intero: `'ciao'*2` fa `'ciaociao'`
    * il cast esplicito per le stringhe è `str`: cosa succede a fare `str(5) + str(10)`? e `"3" + "4"`? e `int("3") + int('4')`? e `int("3") + '4'`?
    * `str` è particolarmente utile se si vuole scrivere un risultato con una spiegazione (vedere più avanti per la `print`): `print "Sommando 3 a 4 si ha " + str(3 + 4)`
    * come detto, per ogni tipo possono essere usate non solo delle variabili, ma anche delle *costanti*
      · le costanti, per l'appunto, non cambiano di valore, quindi non hanno riservata nessuna zona di RAM durante l'esecuzione di un programma
    * il modo di esprimere le costanti cambia a seconda del loro tipo, come detto sopra
      · di che tipo sono le seguenti espressioni? `str(3)`, `str("3")`, `3 + 25*int("4")`, `"ciao"*3`
      · controllare la risposta sull'interprete interattivo, usando l'operatore `type()`
- le espressioni saranno quindi, per ora, di tipo numerico o di tipo stringa, a seconda dei tipi di variabili e costanti al loro interno
- per quanto riguarda la *semantica* dell'istruzione di assegnamento, vale quanto segue:
- si supponga che la generica istruzione di assegnamento sia $N = E$

- **l'effetto è il seguente: come prima cosa, viene valutato il valore E; sia v tale valore e sia b_v il numero di byte necessari a rappresentare v**
  - ad esempio, se l'assegnamento da eseguire è `p = 2*p + 4`, allora per prima cosa viene valutata l'intera espressione `2*p + 4`
  - se il valore precedente della variabile `p` era 20, si avrà $v = 44$ e, dato che occorrono $\lfloor \log_2 44 \rfloor + 1 = 6$ bit per rappresentare 44, 1 byte è sufficient a rappresentare v
  - in pratica, per questioni di efficienza, se l'espressione ha valore intero, vengono usati almeno 24 bytes, quindi in questo esempio avremo $b_v = 24$
- **se N non è mai stato usato in precendenza, allora viene riservata una zona di RAM grande a sufficienza per memorizzare v, e in tale zona viene memorizzato v**
- **se N è già stato usato in precendenza, ma la corrispondente zona di RAM era grande $b' < b_v$ bytes, allora viene riservata una nuova zona di RAM con b_v bytes; infine, in tale zona viene memorizzato v**
  - questa descrizione nasconde alcune difficoltà (per approfondire: vedere *garbage collection*)
- **altrimenti, l'effetto è quello di memorizzare v nei byte di RAM precedentemente dedicati alla variabile N, con ciò cancellando il precedente valore di N**
- concretamente, le variabili sono dunque zone di RAM all'interno dei quali sono memorizzati dei valori
- nel momento in cui si voglia riusare un valore memorizzato in una zona di RAM, basta usare la corrispondente variabile
- molto simile a quello che si fa con le variabili in matematica

`print` è del tipo `print <espressione>` o, in modo totalmente equivalente, `print(<espressione>)`

- semantica di `print` E: **l'effetto è quello ci valutare il valore v dell'espressione E (come nel caso dell'assegnamento); dopodiché di scrivere v su schermo**
- ad esempio `print 1` scrive 1
- `print ab` scrive il contenuto della variabile `ab`, se tale variabile esiste; altrimenti, dà errore
- `print "ab"` scrive la stringa "ab"
- in generale, qualsiasi cosa non sia evidentemente una costante viene trattato come se fosse variabile
- è la stessa cosa scrivere `print(1)`, `print(ab)` e `print("ab")`, rispettivamente

- questa facoltà di omettere le parentesi tonde vale solo per l'istruzione `print`: si vedrà che con le chiamate a funzione le parentesi saranno obbligatorie
- l'istruzione `print` può essere usata anche in modo più sofisticato, ci si ritornerà

controllo di flusso ricadono in questa categoria le seguenti istruzioni:

- `if`; `if ... else`; `while`; `for`; `continue`; `break`; `return`
- queste istruzioni si chiamano così perché alterano la normale sequenza di esecuzione delle istruzioni (non più l'una dopo l'altra, ma ci possono essere dei salti)
- sintassi dell'`if`:
  `if <condizione>:`
  `<blocco_istruzioni>`
  - <blocco_istruzioni> è una successione di 1 o più istruzioni
  - se <blocco_istruzioni> è costituito da una sola istruzione, allora si può anche scrivere senza andare a capo, ovvero
    `if <condizione>: <istruzione>`
  - se <blocco_istruzioni> è costituito da più di una istruzione, allora tutte le istruzioni al suo interno vanno *indentate* alla stessa maniera; quindi così va bene:
    `if <condizione>:`
    `<istruzione1>`
    ⋮
    `<istruzione`n `>`

    e, ad esempio, così no:
    `if <condizione>:`
    `<istruzione1>`
    ⋮
    `<istruzione`n `>`

  - questo varrà per tutti i <blocco_istruzioni> di cui si tratterà in seguito
  - <condizione> dev'essere un'espressione valutabile come un valore *booleano* (vero o falso)
  - per il momento, si assumerà che le espressioni booleane siano quelle ottenibili tramite confronti tra variabili e/o costanti dello stesso tipo, tramite quindi gli operatori `==` (uguaglianza, notare la differenza con l'assegnamento), `<=`, `>=`, `<`, `>`, `!=` (diverso)
  - il confronto può essere sottinteso, ovvero la condizione può essere:

· un numero, e allora l'intera condizione vale falso solo se il numero vale 0, e vero in ogni altro caso

· una stringa, e allora l'intera condizione vale falso solo è la stringa vuota (`''` oppure `""`), e vero in ogni altro caso (compreso quindi `" "`)

· altri casi verranno trattati nelle lezioni a seguire

* per quanto riguarda la *semantica* della generica istruzione `if` C`:` B, vale quanto segue:
* **l'effetto è quello di valutare C: se è vero, viene eseguito B, altrimenti non fa nulla**

– sintassi dell'`if ... else`:

```
if <condizione>:
            <blocco_istruzioni1>
else:
            <blocco_istruzioni2>
```

* vale quanto detto per l'`if`
* mentre all'interno di <`blocco_istruzioni1`> e di <`blocco_istruzioni2`> occorre mantenere la stessa indentazione, i due blocchi possono anche essere indentati diversamente tra loro, dato che sono comunque blocchi distinti
* tuttavia, non è consigliabile farlo, in quanto peggiora la leggibilità
* nell'esempio in Figura 1 c'è un `if ... else` che inizia alla riga 5 e finisce a riga 8; qui <`condizione`> è `operazione == '+'`, mentre sia <`blocco_istruzioni1`> che <`blocco_istruzioni2`> sono costituiti da un'unica istruzione, ovvero una `print`
* per quanto riguarda la *semantica* della generica istruzione `if` C`:` B_1 `else` B_2, vale quanto segue:
* **l'effetto è quello di valutare C: se è vero, viene eseguito B_1, altrimenti viene eseguito B_2**

sequenza le istruzioni vanno separate le une dalle altre per mezzo di andate a capo (e l'indentazione dev'essere la stessa)

- **Esercizio**: modificare il programma di Figura 1 in modo tale che faccia la sottrazione se viene immesso `'-'` da tastiera, mentre con un qualsiasi altro carattere viene fatta la somma

- **Esercizio**: modificare il programma di Figura 1 in modo tale che faccia la divisione (intera, ovvero il quoziente) se viene immesso `'/'` da tastiera, mentre con un qualsiasi altro carattere viene fatto il prodotto

- **Esercizio**: modificare il programma di Figura 1 in modo tale che venga fatta la somma se `operazione` vale 1, la sottrazione se `operazione` vale 2, il quoziente se `operazione` vale 3, e il prodotto con qualsiasi altro valore

  - suggerimento: <blocco_istruzioni> può contenere qualsiasi categoria di istruzioni; quindi anche un altro `if`...

- **Esercizio**: modificare il programma fatto al punto precedente in modo tale che venga fatta la somma se `operazione` vale `'+'`, la sottrazione se `operazione` vale `'-'`, il quoziente se `operazione` vale `'/'`, e il prodotto con qualsiasi altro valore

- **Esercizio**: eseguire i programmi di cui sopra, con diversi possibili input, sia interattivamente che passando l'intero programma all'interprete