Informatica per Statistica
Riassunto della lezione del 18/10/2013

Igor Melatti

Le Funzioni in Python

- Nella scorsa lezione si sono introdotte alcune delle istruzioni del Python
- È possibile anche *creare* nuove istruzioni a piacimento, definendo e chiamando delle *funzioni*
  - ad esempio, il programma in Figura 2 è equivalente a quello di Figura 1 ma definisce ed usa (chiamandola) una funzione, `effettua_operazione`
- Una funzione è concettualmente definita dal suo input, dal suo output, e dalla sequenza di operazioni (codifica di un algoritmo) che trasformano l'input nell'output
  - attenzione: l'output di una funzione *non è* costituito da eventuali `print` contenute al suo interno
  - e l'input non è costituito da eventuali `input` o `raw_input` contenute al suo interno

```
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: Semplice programma Python senza funzioni

```
1  def effettua_operazione(op1, op, op2):
2    if op == "+":
3      return op1 + op2
4    else:
5      return op1 - op2
6
7  operazione = raw_input("Immettere '+' per addizione, qualsiasi " +
8                         " altro carattere per la sottrazione: ")
9  print "Immettere il primo operando: "
10 primo_operando = input()
11 secondo_operando = input("Immettere il secondo operando: ")
12 print str(primo_operando) + " " + operazione + " " + \
13       str(secondo_operando)  + " = " + \
14       str(effettua_operazione(primo_operando, operazione, \
15       secondo_operando))
```

Figure 2: Una variante del programma Python di Figura 1

- Una *definizione di funzione* in Python rispecchia tutto ciò, ed è composta come segue:

  `def <nome_funzione> (<argomenti>):`
  `<blocco_istruzioni>`

  - <nome_funzione> deve essere un *identificatore* (come per le variabili)
  - `<argomenti>` è la lista degli input della funzione
    - dev'essere a sua volta così formata: `<nome_input_1>, ..., <nome_input_`n`>`
    - in realtà, la lista degli argomenti può anche essere scritta diversamente, ma per ora non verranno trattate altre possibilità
    - n può anche essere zero (funzione senza argomenti o senza input); occorre comunque aprire e chiudere le parentesi tonde, senza metterci nulla in mezzo
    - se $n = 1$, allora la virgola ovviamente non va messa
    - i nomi degli input devono essere a loro volta degli identificatori
    - nell'esempio di Figura 2, gli argomenti sono 3 (parte finale della riga 1)
  - il blocco delle istruzioni è tipicamente chiamato *corpo* di una funzione
  - nell'esempio di Figura 2, il corpo va da riga 2 a riga 5
  - il nome della funzione e lista degli argomenti, invece, sono spesso chiamati *intestazione* di una (definizione di) funzione
  - il corpo di una funzione, ovvero la categoria sintattica `<blocco_istruzioni>` di cui sopra, è ancora una volta una sequenza di operazioni (come dentro l'`if`)

- la semantica di `def` $N(a_1, \ldots, a_n)$: B è la seguente: **ricorda che all'identificatore N è associata una funzione con n argomenti $a_1, \ldots, a_n$ e corpo B; questa informazione viene mantenuta in un'opportuna zona di RAM**
- in realtà è una cosa molto simile all'assegnamento, solo che la zona di RAM, anziché contenere la codifica binaria di un valore, contiene la codifica binaria della funzione N
- in effetti, fare un successivo assegnamento su N "distrugge" la funzione, esattamente come succede con assegnamenti successivi sulla stessa variabile
- una definizione di funzione è quindi a tutti gli effetti un'istruzione del Python
- questo implica che è possibile scrivere un programma Python così:

  B_1
  $D_{1,1}$
  $\vdots$
  D_{1,k_1}
  B_2
  $D_{2,1}$
  $\vdots$
  D_{2,k_2}
  $\vdots$ B_{n-1}
  $D_{n-1,1}$
  $\vdots$
  $D_{n-1,k_{n-1}}$
  B_n

  dove i B_i sono blocchi di istruzioni anche vuoti (cioè, istruzioni consecutive scritte senza indentazione, verranno nel seguito chiamate *istruzioni principali*) e i D_j sono definizioni di funzioni (anche vuote)
- i B_i vanno eseguiti *comunque* (a meno di errori)
- i corpi delle D_j vanno eseguiti solo se chiamati (direttamente o indirettamente) da qualche istruzione di qualche B_i
  - per "indirettamente" si intende che un'istruzione potrebbe non chiamare direttamente una funzione f, ma potrebbe chiamare una funzione g che poi chiama f
- inoltre, questo implica che una definizione di funzione può contenere altre definizioni di funzioni
- in quest'ultimo caso, le funzioni definite dentro una funzione possono essere usate solo da dentro quella funzione; su questo argomento si ritornerà più avanti

- La *chiamata di funzione* ha la seguente sintassi: `<nome_funzione>(<lista_espressioni>)`
  - `<lista_espressioni>` è del tipo `<espressione1>, ..., <espressionen >`, dove n è lo stesso della definizione di funzione (sono possibili eccezioni, ma per ora non le trattiamo)
  - notare come la sintassi sia diversa dalla definizione di funzione (niente `def` con i `:` alla fine, niente corpo della funzione)
  - per quanto riguarda la *semantica* dell'istruzione di chiamata a funzione $N(E_1, \ldots, E_n)$, effettuata all'interno di una istruzione I (detta *istruzione chiamante*), con N definita precedentemente come `def` $N(a_1, \ldots, a_n)$`:` B (se la definizione non è stata data in precedenza, verrà ritornato un errore), vale quanto segue:
  - **come prima cosa, si valutano le** n **espressioni** $E_1, \ldots, E_n$**; siano** $v_1, \ldots, v_n$ **i corrispondenti valori** (questo passo nasconde alcune difficoltà)
  - **dopodiché, si sospende l'esecuzione di** I**, si esegue** B **e poi si ritorna il controllo all'istruzione** I**; nell'eseguire** B**, prima di ogni cosa, si inizializzano (cioè, si assegnano) le variabili di input** $a_1, \ldots, a_n$ **a** $v_1, \ldots, v_n$ **(nell'ordine)**
  - **l'esecuzione di** B **finisce o quando si esegue normalmente l'ultima istruzione di** B**, o quando si esegue una** `return`
    - l'istruzione I potrebbe essere:
      1. un assegnamento del tipo `var =` E dove E contiene la chiamata a funzione
      2. un `if` del tipo
         `if` E`:`
         B
         dove E contiene la chiamata a funzione
      3. un `if .. else` del tipo
         `if` E`:`
         B_1
         `else:`
         B_2
         dove E contiene la chiamata a funzione
      4. una `print` del tipo `print` E, dove E contiene la chiamata a funzione
      5. una istruzione I, dove I è costituita da una chiamata a funzione
      6. ...
    - in tutti i casi di cui sopra, escluso il caso 5, la chiamata a funzione è all'interno di una espressione, quindi occorre sapere quale sia il *valore* della funzione

· per esempio, nel caso della `print`, se si scrive `print effettua_operazione(34, '-', 45)*2`, occorrerà sapere qual è l'effettivo valore di `effettua_operazione(34, '-', 45)*2`, e quindi di `effettua_operazione(34, '-', 45)`: Fa 13? "+"? -22?

· per saperlo, si chiama la funzione `effettua_operazione`, assegnando 34 ad `op1`, '+' a `op` e 45 ad `op2`, si esegue l'`if` all'interno del corpo di `effettua_operazione` e si esegue la `return op1 - op2`

· dato che `op1 - op2` in questo caso fa -11, l'intera chiamata `effettua_operazione(34, '-', 45)` viene valutata come -11

· quindi la `print` scriverà -22

· pertanto, *il valore di una funzione (o meglio, il valore ritornato da una funzione) è quello della prima (e unica)* `return` *che esegue* (vedere anche più avanti)

∗ per "ritornare il controllo" si intende che, alla fine dell'esecuzione della funzione chiamata, si riprende ad eseguire I

· nell'esempio di sopra, dopo aver eseguito il blocco di istruzioni all'interno di `effettua_operazione`, si ritorna ad eseguire la `print`, la cui esecuzione era stata sospesa

∗ una funzione chiamata potrebbe anche non effettuare alcuna `return`: in questo caso, funziona semplicemente come una *nuova istruzione*, senza bisogno di trovarsi in un'espressione di un assegnamento (caso 5 di cui sopra)

∗ in realtà, il caso 5 di cui sopra va bene anche se la funzione fa una `return`: il valore ritornato viene ignorato nel caso di esecuzione di un programma, mentre viene stampato nel caso dell'interprete interattivo

∗ è da notare che I potrebbe a sua volta essere l'istruzione di un blocco di istruzioni di un'altra funzione (si dice allora che questa seconda funzione è la *funzione chiamante*)

∗ ovvero, una funzione può chiamarne un'altra (cosa ovvia, visto che la poteva anche definire)

– ad esempio, la riga 14 contiene una chiamata alla funzione `effettua_operazione`

∗ in questo caso, da dentro un assegnamento

- A questo punto è il caso di dare la sintassi e la semantica generali dell'istruzione `return`

– sintassi: `return` <`espressione`>

  - semantica di `return` E (effettuata da dentro una funzione N): **l'effetto è quello di valutare E; ne sia v il valore. Dopodiché, si termina l'esecuzione della funzione nella quale ci si trova e si ritorna all'istruzione chiamante, dove il valore della chiamata ad N sarà v**
  - se la `return` viene eseguita senza che si sia all'interno di una funzione, viene dato errore

- Altri effetti delle chiamate a funzione: le variabili che usano esistono solo al loro interno (sono cioè *locali*): è lo *scoping*
  - all'interno della *definizione* di una funzione, saltano i controlli sul fatto che una variabile sia definita o no
  - per esempio, se non ci si trova all'interno di alcuna definizione di funzione (quindi ci si trova nelle istruzioni principali, che il programma esegue comunque) e l'interprete incontra un'espressione `p + 2` senza che a `p` sia stato assegnato prima alcunché, l'interprete dà errore e si ferma
  - invece, nel definire il corpo di una funzione questa cosa viene tollerata
  - è quando la si chiama, che l'interprete effettua il controllo: vedere Figura 3 (nota: tutto quello che si trova dopo un cancelletto (`#`) viene ignorato dall'interprete Python, ovvero è un *commento*)
  - questo quando, all'interno del corpo di una funzione, si usa (senza tentare di modificarla con un assegnamento) una variabile esterna (*globale*)
  - cosa succede invece alle variabili definite (ovvero assegnate) all'interno di un corpo di funzione?
  - quelle variabili (e funzioni, se ne vengono definite altre) hanno vita solo dentro la funzione; dopo la `return`, o dopo la fine del corpo della funzione (se la `return` non c'è o non viene eseguita) cessano di esistere
  - se ci sono variabili globali con lo stesso nome, esse vengono momentaneamente oscurate durante l'esecuzione (chiamata) della funzione, per poi tornare ad avere il loro valore originario
  - ad esempio, in Figura 4 c'è una variabile `p` globale definita a riga 1, che però viene ridefinita (riassegnata) localmente a riga 4
  - quando viene chiamata `f` in riga 10 (e quindi `g` a riga 7) `p` cambia il suo valore, ma una volta che la chiamata finisce e si ritorna alle istruzioni principali `p` riprende il suo valore
  - sarà necessario tornare su questo argomento, per trattare di (apparenti) eccezioni e spiegare come fa l'interprete a non perdersi i pezzi
  - è comunque evidente che ci saranno due zone di RAM diverse per la `p` globale e per quella locale

- Questo comportamento è alterabile usando l'istruzione `global`
  - confrontare le Figure 4 e 5
  - sintassi: `global <nome_var>`
  - semantica di `global` N (supponendo che sia dentro il corpo di una funzione, altrimenti è inutile): **se N è stata definita con scoping globale in precedenza (tramite assegnamento), l'effetto è quello di far sì che, quando la funzione viene chiamata, da questo punto in poi e per tutta la durata della funzione si faccia riferimento alla N globale**
    - quindi, anziché dare luogo ad una variabile locale che oscura quella globale fintantoché si sia all'interno della funzione, si fa riferimento proprio alla variabile globale
    - in Figura 4, esiste una `p` globale, ma all'interno della funzione `g` (e solo lì dentro) viene oscurata dalla nuova `p` locale: ovvero, l'interprete Python non considera la `p` globale, e lavora con `p` come se fosse una variabile nuova, che però sparisce alla fine della chiamata a `g`
    - in Figura 5, esiste una `p` globale, e all'interno della funzione `g` vi si fa direttamente riferimento (a causa dell'istruzione `global p`)
    - N non deve essere necessariamente definita nelle istruzioni principali: può anche essere stata definita come `global` in un'altra funzione già chiamata in precedenza (o anche in questa stessa, che potrebbe essere chiamata più volte)
  - **altrimenti, da questo punto in poi N viene promossa a scoping globale, e si farà riferimento alla N globale per tutta la durata della funzione**

```
1 p = p + 4 #errore, p non esiste
2 def f():
3   return p + 4 #nessun problema
4 f() #errore, p non esiste
5 p = 3
6 f() #adesso si', torna 7
```

Figure 3: Un esempio per lo scoping

- Le funzioni non esistono solo per dare la possibilità al programmatore di creare nuove istruzioni
- Servono anche a chiamare funzioni già definite da altri

```
1  p = 4
2  def f():
3    def g(): #definizione di funzione annidata
4      p = 2
5      return p
6    q = 3
7    return g() + q
8  print q #errore, q esiste solo dentro f
9  print g() #errore, g esiste solo dentro f
10 print f() #torna 5
11 print p #vale ancora 4, non 2
```

Figure 4: Un altro esempio per lo scoping (variabili nascoste)

```
1  p = 4
2  def f():
3    def g():
4      global p
5      p = 2 #modifica la variabile di riga 1
6      return p
7    global q
8    q = 3
9    return g() + q
10 print q #errore, q non esiste (non esisteva prima di f)
11 print g() #errore, g esiste solo dentro f
12 print f() #torna 5
13 print p #questa volta scrive 2
14 print q #adesso (dopo la chiamata ad f che la definisce) esiste, e scrive 3
```

Figure 5: Esempio di Figura 4 rifatto con `global`

- alcune funzioni sono definite di default: nelle lezioni precedenti sono state introdotte `input()`, `raw_input`, `int()`, `str()`, `float()` (le ultime 3 sono un po' particolari, ma per il momento non verranno dati altri dettagli)
- per sapere cosa fa una funzione (*semantica* della funzione) si può scrivere `help(<nome_funzione>)` sull'interprete interattivo; a sua volta, `help` è una funzione (anch'essa particolare)
- ovviamente, la risposta è in inglese
- esistono collezioni di funzioni importanti, racchiuse in *moduli* (che non sono altro che altri programmi Python già fatti e pronti per essere usati)
- ad esempio: se si vuole conoscere il logaritmo di un numero, o la tangente trigonometrica di un angolo, si può:
  - scrivere una sequenza di istruzioni che li calcoli (ad esempio implementando lo sviluppo di Taylor...), e ripetere questa sequenza di istruzioni ogni volta che occorre
    - decisamente poco furbo: e se poi ci si accorge che si è sbagliato qualcosa nello scrivere questa sequenza di istruzioni? bisogna ritrovare tutti questi blocchi e cambiarli...
    - inoltre potrebbero trovarsi a diversi livelli di indentazione, occorre cambiarli...
  - scrivere una funzione che calcoli quanto necessario e chiamarla (già meglio)
  - usare la funzione già fatta (meglio ancora)
- per usare una funzione già fatta occorre sapere che, per l'appunto, è già stata fatta; Internet può aiutare in tal senso
- nell'esempio del logaritmo e della tangente, basta *importare* il modulo `math` e poi chiamare `math.log` e `math.tan`
- la sintassi dell'importazione di moduli è la seguente: `import <nome_modulo>`, ed è una istruzione a tutti gli effetti
- la semantica di `import` M è la seguente: **esegui il programma** M`.py`
- in pratica, dato che i moduli predefiniti hanno come istruzioni principali solo definizioni di funzioni, vuol dire "definisci tutte le funzioni in M"
- è possibile vedere tutte le funzioni di un modulo M con `help(`M`)`
- per chiamare una funzione N in un modulo M del quale sia stato fatto `import`, occorre scrivere M`.`N`(`a_1`, ..., `a_n`)`

- **Esercizio**: fare gli stessi esercizi di lezione 7, ma modificando il programma in Figura 2.