Alcuni lucidi sono tratti da una presentazione di Marco Barisione

Introduzione al linguaggio Python

Gli operatori matematici

- Esistono gli stessi operatori del C, le parentesi possono essere usate per raggruppare:

```
>>> (5 + 3) * 2
16
>>> (6 & 3) / 2
1
```

- Esiste anche l'operatore elevamento "**":

```
>>> 5 ** 2
25
```

- Non esistono "++" e "—"

  Come in C, "3/2" è "1"

  Altrimenti usare 3.0/2 (1.5)

  3.0//2 è la divisione intera

Variabili

- I nomi di variabili sono composti da lettere, numeri e underscore, il primo carattere non può essere un numero (come in C)
  - Sono validi:
    - “x”, “ciao”, “x13”, “x1_y”, “_”, “_ciao12”
  - Non sono validi:
    - “1x”, “x-y”, “$a”, “àñÿô”
- Le variabili non devono essere dichiarate (Tipizzazione dinamica)
- Una variabile non può essere utilizzata prima che le venga assegnato un valore
- Ogni variabile può riferirsi ad un oggetto di qualsiasi tipo

Le variabili

Esempi:

```
x=5
nome="Marco"
```

- Sintetico

```
inizo,fine=2,100
```

- type restituisce il tipo di una variabile

```
x=[ 5  3]
    type (x)
  >>>   <type 'list'>
```

x è di tipo lista (verrà vista più avanti)

Assegnamento (1)

- L'assegnamento avviene attraverso l'operatore "="
- Non è creata una copia dell'oggetto:
  - `x = y # si riferiscono allo stesso oggetto`
- Esempio:

```
>>> x = [0, 1, 2]
>>> y = x
>>> x.append(3)
>>> print y
[0, 1, 2, 3]
```

del

- L'istruzione "del" ha due usi differenti
  - "del x" cancella la variabile "x", cioè non si potrà più usare "x" senza avergli prima assegnato un nuovo valore
    - "del" non distrugge ciò a cui la variabile si riferisce come "free" in C!
  - "del seq[ind]" cancella l'elemento con indice/chiave "ind" da "seq"

```
li = [1, 2, 3]
del li[1]
```
→ li = [1, 3]

```
d = {"a": 1, "b": 6}
del d["a"]
```
→ d = {'b': 6}

```
l2 = [1, 2, 3, 4]
del l2[1:3]
```
→ l2 = [1, 4]

Vero e falso

- In Python esistono due variabili di tipo bool:

  “True” uguale a 1 e “False” uguale a 0
- Ogni singolo tipo o classe può definire quando il suo valore è vero o falso
- Per i tipi predefinti sono considerati falsi:
  - Il numero “0” o “0.0”
  - Una stringa vuota (“”)
  - “{}”, “[]”, “()
- Gli altri valori sono considerati veri

Gli operatori di confronto

- Sono gli stessi del C
  - `1 == 3` → Falso
  - `1 == 2 - 1` → Vero
  - `1 != 2` → Vero
  - `1 < 2` → Vero
  - `1 > 3` → Falso
  - `1 >= 1` → Vero
- Se necessario vengono eseguite le necessarie conversioni intero → virgola mobile
  - `1 == 1.0` → Vero
- Esiste anche l'operatore "<>" equivalente a "!=" ma obsoleto

Altri operatori di confronto

- “in”, vero se il primo operando è contenuto nel secondo
  - `5 in [1, 2, 3]` → Falso
  - `2 in [1, 2, 3]` → Vero
  - `"a" in {"x": 1, "a": 2}` → Vero
  - `"a" in "ciao"` → Vero
- “is”, vero se il primo operando è il secondo (non solo è uguale!)
  - Attualmente implementato come confronto fra le posizioni in memoria degli operandi (provate id (x), da l’indirizzo di memoria)
  - Usato al posto di “==” per il confronto con “None” per motivi di prestazioni

Gli operatori booleani

- “not x” 0 se “x” è vero, “1” se è falso
- “x and y” vero se sia “x” sia “y” sono veri. Ritorna:
  - Se “x” è falso lo ritorna
  - Altrimenti ritorna “y”
    - `1 and 5` → 5 → Vero
    - `[] and 1` → [] → Falso
- “x or y” vero se almeno uno degli argomenti è vero
  - Se “x” è vero lo ritorna
  - Altrimenti ritorna “y”
    - `1 or 0` → 1 → Vero
    - `() or 0` → 0 → Falso
- Sia “and” sia “or” utilizzano la logica del corto circuito
  - Il secondo argomento non viene valutato se il risultato dell’operazione è già noto in base al solo primo argomento

if

- La sintassi di “if” è:

```
if condizione:
    ...
elif condizione_alternativa:
    ...
else:
    ...
```

- Sia la parte “elif” sia la parte “else” sono facoltative
- Può esserci un numero qualsiasi di “elif”
- Non sono necessarie le parentesi intorno all’espresisone booleana
- Non sono possibili assegnamenti all’interno della condizione

L’indentazione

- Il raggruppamento è definito dall’indentazione
  - Non si usano parentesi graffe, coppie “begin”/“end” e simili
  - Obbliga a scrivere codice ordinato
  - Più naturale, evita i tipici problemi del C:

```
if (0)
    printf("Questo non viene eseguito");
    printf("Questo sì");
```

- Si possono usare spazi o tabulazioni
- È *fortemente* consigliato usare 4 spazi
  - Tutti gli editor decenti possono essere configurati per sostituire il TAB con 4 spazi

pass

- Come tradurre il seguente codice in Python?

```
if (a){}        /* Oppure "if (a);" */
b = 12;
```

- In Python dopo un “if” deve esserci un blocco indentato

```
if a:
b = 12          # Dovrebbe essere indentato!
```

- Si usa quindi l’istruzione “pass” che non ha *nessun* effetto

```
if a:
    pass        # Non fa nulla
b = 12
```

while

- La sintassi è:

```
while condizione:
    ...
while i<10:
    ...
```

- Si può uscire dal ciclo usando “break”
- Si può passare all’iterazione successiva usando “continue”
- Esempio:

```
while 1: # significa ciclo infinito
    if condizione1:
        continue
    if condizione2:
        break
print 42
```

for

- La sintassi è:

```
for variabile in iteratore:
    ...
```

- "iteratore" può essere:
  - Una sequenza:
    - Liste
    - Tuple
    - Stringhe
    - Dizionari
    - Classi definite dall'utente
  - Un iteratore (nuovo concetto introdotto in Python 2.2)
- Si possono usare "continue" e "break" come per il "while"

Le stringhe

- Le stringhe sono racchiuse fra apici singoli o doppi e si utilizzano le stesse sequenza di escape del C

```
>>> 'Python'
'Python'
>>> print "Ciao\nmondo"
'Ciao'
'mondo'
```

- Si possono usare alcuni operatori visti per i numeri

```
>>> "ciao " + "mondo"    # concatenazione
'ciao mondo'
>>> "ABC" * 3            # ripetizione
'ABCABCABC'
```

Sottostringhe

c	i	a	o
0	1	2	3
-4	-3	-2	-1

- `"ciao"[1]` `# carattere 1` → “i”
- `"ciao"[1:3]` `# dall'1 al 3 escluso` → “ia”
- `"ciao"[2:]` `# dal 2 alla fine` → “ao”
- `"ciao"[:3]` `# fino al 3 escluso` → “cia”
- `"ciao"[-1]` `# l'ultimo carattere` → “o”
- `"ciao"[:-2]` `# fino al penultimo` → “ci”
- `"ciao"[-1:1]` `# non esiste` → “”
- Le stringhe sono immutabili (come i numeri):
  - `"ciao"[2] = "X"` → **Errore!**

Altri tipi di stringhe

- Le stringhe possono estendersi su più righe, in questo caso sono delimitate da tre apici singoli o doppi

```
"""Questa è una
stringa su più righe"""
```

- For e stringhe

```
nome=”Gianluigi”
for c in nome:
    print c
```

Stringhe, metodi

- “S.split([sep[, max]])” ritorna una lista contenente le parti di “S” divise usando i caratteri di “sep” come separatore. “max” è il numero massimo di divisioni eseguite. Il valore di default di “sep” è ogni spazio bianco
- “S.join(seq)” ritorna una stringa contenente gli elementi di “seq” uniti usando “S” come delimitatore
- “S.lower()” ritorna “S” convertito in minuscolo
- “S.upper()” ritorna “S” convertito in maiuscolo
- “S.find(what[, start[, end]])” ritorna il primo indice di “what” in “S[start, end]”. Se la sottostirnga non è trovata ritorna “-1”
- “S.rfind(what[, start[, end]])” come “find” ma a partire dal fondo
- “S.replace(old, new[, max])” ritorna una copia di “S” con “max” occorrenze di “old” sostituite con “new”. Il valore di default di “max” è tutte.
- “S.strip()” restituisce una copia di “S” senza gli spazi iniziali e finali
- “S.lstrip()”, “S.rstrip()” come “strip” ma eliminano rispettivamente solo gli spazi iniziali e finali

None

- “None” è una variabile molto importante con lo stesso ruolo di “NULL” in C
- In C “NULL” è uguale a “0”
  - Rischio di confusione
- In Python “None” è di un tipo non numerico
  - Non vi è rischio di confusione con altri tipi

Alcune funzioni built-in

- “range([start,] stop[, step])” ritorna una lista contenente gli interi in [“start”, “end”).
  - “step” è l’incremento, il valore di default è “+1”.
  - Il valore di default di “start” è “0”
  - Molto usato con i cicli “for”
    - `for i in range(5): print i`

- “len(seq)” ritorna il numero di elementi in “seq”
  - `len("ciao")` → 4
  - `len("x\0x")` → 3
  - `len([1, 2, 3])` → 3
  - `len({"a": 1, "b": 5})` → 2

Liste

- Contengono elementi anche eterogenei
- Sono implementate usando array e non liste
- Per creare una lista si usano le parentesi quadre, gli elementi sono delimitati da virgole

```
>>> [1, 2, "ciao"]
[1, 2, 'ciao']
```

- Stessi operatori delle stringhe ma sono mutabili
  - `[1] + [3, 6]` → [1, 3, 6]
  - `[1, 0] * 3` → [1, 0, 1, 0, 1, 0]
  - `[2, 3, 7, 8][1:3]` → [3, 7]
  - `[2, 3, 7, 8][:2]` → [2, 3]
  - `[1, 2, 3][0] = 5` → [5, 2, 3]

Liste, metodi

- Ecco i metodi più usati:
  - “L.append(obj)”, aggiunge “obj” fondo
  - “L.extend(list)”, aggiunge in fondo gli elementi di “list”
  - “L.insert(index, obj)”, aggiunge “obj” prima di “index”
  - “L.pop([index])”, rimuove l’elemento in posizione “index” e lo ritorna, il valore di default di “index” è -1
  - “L.remove(value)”, cancella la prima occorrenza di “value”
  - “L.reverse()”, inverte la lista
  - “L.sort()”, ordina la lista
- Tutti “in place”, viene modificata la lista, non viene ritornata una nuova lista
- ATTENZIONE + rispetto ad append crea una nuova lista