Callables

- I tipi "*chiamabili*" hanno delle istanze che supportano operatori di chiamata a funzione
- Le funzioni sono chiamabili (ovviamente!)
- Anche i tipi sono chiamabili (ad esempio: dict, list, int…)
- Gli oggetti di classe sono chiamabili
- I metodi sono chiamabili
- Le istanze le cui classi corrispondenti forniscono i metodi __call__ sono chiamabili

Boolean

- Prima di Python 2.3 non esisteva un tipo esplicito *booleano*
- Tuttavia, ogni dato può essere considerato come un valore logico: *vero* o *falso*
- Ad esempio lo 0 numerico, le liste/tuple/dizionari/stringhe vuote corrispondono al valore false
- In Python 2.3 *bool* diventa un tipo (sottoclasse di int) che assume i valori False e True (prima si usavano i numeri 0 e 1)

Variabili e altre referenze

- Un programma Python accede ai dati tramite le *referenze*
- Una referenza è un nome che si riferisce ad una specifica locazione di memoria in cui è presente un certo valore (oggetto)
- Una variabile o referenza non ha tipo intrinseco ma lo ha l'oggetto cui è riferita
- Una qualunque referenza può essere associata ad oggetti di diverso tipo durante l'esecuzione del programma

Variabili e altre referenze

- In Python NON esistono dichiarazioni
- L’esistenza di una variabile dipende da un istruzione che associa tale variabile ad un valore
- La dipendenza può essere eliminata tramite l’istruzione *del*
- Associare un oggetto diverso ad una referenza esistente ha come effetto di distruggere l’oggetto, nel caso in cui niente si riferisca più ad esso (operazione compiuta automaticamente dal *garbage collector*)

Variabili e altre referenze

- NON è possibile accedere a referenze inesistenti!
- Nel caso di compilazione del codice, vengono segnalati solo gli errori di sintassi
- Errori “semantici”, come l’accesso a referenze non inizializzate, non vengono segnalati e provocano errori in run-time (generalmente viene lanciata un’*eccezione*)

Assegnamento

- Le istruzioni di assegnamento possono essere *semplici (plain)* o *aumentate (augmented)*
- Gli assegnamenti semplici su una variabile (*var=obj)* creano una variabile nuova o ne assegnano una esistente ad un nuovo oggetto
- Gli assegnamenti semplici sull'attributo di un oggetto *(obj.attr=value)* creano o ricollegano l'attributo ad un nuovo valore
- Gli assegnamenti semplici su un contenitore (obj[key]=value) creano l'oggetto corrispondente ad una chiave o ne cambiano il valore

Assegnamento

- L’assegnamento semplice ha come forma più semplice la seguente:
  destinazione = *espressione*
- Viene prima valutata l’*espressione* a destra e poi viene collegato il suo risultato con l’etichetta *destinazione* a sinistra
- Il collegamento (*binding*) non dipende dal tipo del valore associato
- E’ quindi possibile assegnare ad una variabile anche funzioni, tipi e metodi

Assegnamento

- I “dettagli” del collegamento dipendono comunque dal tipo della destinazione
- La destinazione può infatti essere:
  - un identificatore
  - un attributo di un oggetto
  - un’indicizzazione
  - una “slicing”

Assegnamento

- Un *identificatore* è un nome di variabile: un'espressione è associata ad un nome
- Il riferimento ad un attributo ha la sintassi *obj.name*, dove *obj* è un'espressione che denota un oggetto e *name* è un identificatore
- Un'*indicizzazione* ha la sintassi *obj[expr]*, dove *obj* e *expr* possono essere oggetti di qualunque tipo (*obj* è un contenitore e *expr* rappresenta una chiave/indice d'accesso)

Assegnamento

- Una *slicing* ha la sintassi *obj[start:stop]* oppure *obj[start:stop:stride]*
- start, stop e stride possono essere indici di vario tipo e sono opzionali (è possibile ad esempio scrivere obj[:stop:])
- L'assegnamento ad una slicing richiede al contenitore *obj* di collegare soltanto alcuni dei suoi elementi

Assegnamento

- Assegnare un’espressione ad un identificatore è un’operazione che va sempre a buon fine
- Gli altri casi di assegnamento possono generare eccezioni, nel caso in cui non sia possibile creare un collegamento tra attributi, parte di valori, etc.
- E’ possibile effettuare assegnamenti multipli:
  a = b = c = 0

  L’espressione a destra è valutata una sola volta e viene creato un collegamento con tutte le variabili sulla sinistra

Assegnamento

- L’espressione
  a, b, c = x

  richiede invece che *x* sia una sequenza di 3 elementi (né più né meno!) e collega *a* al primo elemento, *b* al secondo e *c* al terzo
- Questo tipo di assegnamento viene definito di *scompattazione* (*unpacking*) e richiede alla destra una sequenza con un numero esatto di elementi
- Può essere usato anche per scambiare referenze:
  a, b = b, a

Assegnamento aumentato

- Gli assegnamenti *aumentati* differiscono dagli assegnamenti semplici per il fatto che usano tra la destinazione e la espressione un *operatore aumentato*: un operatore binario seguito da ‘=‘
- Gli assegnamenti aumentati non possono creare nuove referenze
- Gli assegnamenti aumentati possono cambiare il collegamento di una variabile
- Non supportano destinazioni multiple

Assegnamento aumentato

- Gli operatori aumentati sono:
  `+=  -=  *=  /=  //=  %=  **=  |=`
  `>>=  <<=  &=  ^=`
- In un assegnamento aumentato viene valutata l'espressione a destra e, se a sinistra si ha un riferimento ad un oggetto che contiene un metodo speciale, questo viene richiamato; altrimenti viene applicato il corrispondente operatore binario a sinistra e a destra e la referenza viene collegata al risultato

Assegnamento aumentato

- Esempio:
  x += y

  equivale a:

  x = x.__iadd__(y) #se esiste il metodo __iadd__

  x = x + y #se non esiste __iadd__

Assegnamento aumentato

- L’assegnamento aumentato non crea mai la referenza a sinistra: essa deve essere stata già definita prima dell’assegnamento
- Può invece ricollegare la referenza ad un nuovo oggetto o modificare quello originale cui puntava la referenza
- L’assegnamento semplice può sia creare che ricollegare la referenza alla sinistra dell’uguale ma non modifica l’oggetto cui eventualmente puntava la referenza

Assegnamento aumentato

- Attenzione alla differenza tra:
  x = x + y
  e
  x += y

- Nel primo caso l'oggetto cui puntava x non viene modificato (x ora indicherà un nuovo oggetto)
- Nel secondo caso viene modificato l'oggetto puntato da x nel caso in cui esista il metodo __iadd__ (altrimenti è uguale al primo caso)

Istruzione ‘del’

- Nonostante il nome, l’istruzione non distrugge oggetti, ma “scollega” dei riferimenti
- La *cancellazione* di un oggetto può diventare una conseguenza, per effetto del garbage collector (un oggetto è distrutto quando nessuna etichetta è più riferita ad esso)

Espressioni ed Operatori

- Un'*espressione* è una parte di codice che può essere valutata per produrre un valore
- Le espressioni più semplici sono i *letterali* e gli *identificatori*
- Espressioni più complesse possono essere create collegando sottoespressioni con *operatori* e/o *delimitatori*
- Esiste un ordine di precedenza tra gli operatori, tipicamente riassunto in una tabella

Precedenza tra gli operatori

- Conversione a stringa, creazione di Dizionario, Lista e Tupla
- Chiamata di Funzione
- Slicing, indicizzazione, accesso ad attributo
- Elevamento a potenza
- Not binario
- Operatori + e – unari
- Moltiplicazione, divisione, troncamento, resto
- Addizione, sottrazione
- Shift sinistro e destro (<< e >>)
- And, xor e or bit a bit
- Confronto, test di uguaglianza
- Not booleano
- And, or booleani
- Funzioni anonime con lambda

Espressioni ed Operatori

- E’ possibile concatenare tra loro i confronti, sottintendendo un *and* logico:

  a < b <= c < d

  è equivalente (ma più leggibile!) a:

  a < b **and** b <= c **and** c < d

Espressioni ed Operatori

- L’espressione “x **and** y” è così interpretata:

  prima si valuta x e se è falso il risultato è x; altrimenti il risultato è y

- L’espressione “x **or** y” è così interpretata:

  prima si valuta x e se è vero il risultato è x; altrimenti il risultato è y

Espressioni ed Operatori

- In altre parole, *and* e *or* non forzano il risultato ad un *booleano* (True o False) ma rendono uno dei loro operandi
- Questo consente l'utilizzo di tali operatori in un contesto più esteso di quello booleano

Operazioni Numeriche

- Tutti i numeri sono oggetti immutabili: un'operazione numerica produce sempre un nuovo oggetto
- Da notare che il segno anteposto ad un numero non fa parte della sua sintassi, ma viene trattato come un operatore unario
- L'espressione -2**2 dà infatti come risultato -4 perché l'elevamento a potenza ha priorità superiore rispetto al meno unario!

Operazioni Numeriche

- E’ possibile effettuare un’operazione numerica e un confronto tra qualunque tipo numerico: se gli operandi sono differenti, viene applicata la *coercion*
- Python dunque converte automaticamente gli operandi di tipo più “piccolo” in operandi di tipo più “grande” (es. interi in floating point)
- E’ possibile comunque effettuare una conversione esplicita mediante le funzioni interne *int*, *long*, *float* e *complex*

Operazioni Numeriche

- Ciascun tipo predefinito può ricevere come argomento una stringa contenente la sintassi numerica corrispondente, con due piccole estensioni:
  - La stringa può cominciare con un segno
  - Nel caso di complessi è possibile sommare o sottrarre parte reale e immaginaria
- *int* e *long* possono accettare due argomenti: il secondo è la base numerica nella quale si considera espresso il numero dentro la stringa

Operazioni Numeriche

- Esempio:

```
int (‘101’, 2)    # risultato: 5 corrispondente
                  # al binario 101

int (‘ff’,16)     # risultato: 255 corrispondente
                  # all’esadecimale FF
```

Operazioni Numeriche

- Se l'operatore a destra di '/','//' o '%' è zero, viene lanciata un eccezione
- L'operatore '//' effettua la divisione "*troncata*": viene restituito come risultato un numero intero (convertito poi nello stesso tipo dell'operando più grande)
- Se entrambi gli operandi sono interi, '/' si comporta come '//' (per evitare questo comportamento, occorre convertire in floating point almeno uno degli operandi)

Operazioni Numeriche

- La funzione *divmod* prende due argomenti e restituisce una coppia i cui elementi sono il quoziente e il resto:

  divmod (3,5) # risultato: tupla (0, 3)

- L'operazione a**b lancia un'eccezione se *a* è negativo e *b* è un floating point con parte frazionale non nulla
- ***pow*** *(a,b,c)* equivale a (a**b)%c ma è più veloce

Operazioni Numeriche

- E’ possibile effettuare operazioni di *confronto* tra intero, sia come uguaglianza (==), sia come disuguaglianza (!=)
- Le operazioni di confronto che implicano *ordinamento* sono ammesse tra tutti i numeri, eccezion fatta per i complessi (verrebbe lanciata un’eccezione)
- Tutti questi operatori rendono un valore *booleano* (False o True)

Operazioni sulle Sequenze

- Le *sequenze* sono contenitori con gli elementi accessibili tramite *indicizzazione* o *slicing*
- La funzione *len* prende un contenitore come argomento e rende il numero di elementi nel contenitore
- Le funzioni *min* e *max*, prendono come argomento una sequenza, i cui elementi sono confrontabili e rendono l'elemento più piccolo o più grande della sequenza
- *min* e *max* possono ricevere anche più argomenti

Coercion/Conversion

- Non vi è conversione implicita tra sequenze differenti eccetto l'eventuale conversione in stringhe unicode
- E' possibile utilizzare le funzioni *tuple* e *list* con un argomento (una sequenza o un oggetto iterabile) per ottenere un'istanza del tipo che si sta chiamando, con gli stessi elementi (e nello stesso ordine) del parametro passato

Concatenazione

- E’ possibile *concatenare* sequenze dello stesso tipo mediante l’operatore ‘+’
- E’ possibile inoltre duplicare *n* volte una sequenza moltiplicandola per un intero tramite l’operatore ‘*’
- Una sequenza moltiplicata per 0 dà come risultato una sequenza vuota

```
“a” * 5   # “aaaaa”
“a” * 0   # “”
```

Appartenenza

- E’ possibile verifcare l’*appartenenza* di un elemento ad una sequenza mediante l’operatore:
  x **in** S
  che rende True o False

- Analogamente, l’operatore
  x **not in** S
  verifica la non appartenenza di x a S e equivale a
  not (x **in** S)

Indicizzazione

- E’ possibile accedere agli elementi di una sequenza racchiudendo un *indice* tra parentesi quadre ‘[]’
- Il primo elemento della sequenza è indicizzato dallo 0
- Si accede all’ultimo elemento con l’indice -1, al penultimo con -2, etc. (il massimo consentito è –L, con L la dimensione della sequenza)
- Usare un indice maggiore di L o minore di –L genera un’eccezione

Slicing

- E’ possibile estrarre una sottosequenza di una sequenza mediante *slicing*, usando la sintassi:
  S[i:j]
  dove *i* e *j* sono indici interi

  sono estratti gli elementi della sottosequenza a partire dall’i-esimo (incluso) sino al j-esimo (escluso)

  NOTA: in Python viene sempre incluso l’estremo sinistro di un range ed escluso quello destro

Slicing

- Una *slicing* può produrre una sequenza vuota se *j* è inferiore a *i* oppure se *i* è maggiore o uguale alla dimensione della sequenza (*L*)
- *i* può essere omesso se è uguale a 0
- *j* può essere omesso se maggiore o uguale a *L*
- L'intera sequenza (una copia) può essere ottenuta omettendo entrambi gli indici:

  S[:]

Slicing

- Gli indici possono anche essere negativi e a partire da Python 2.3 si può utilizzare la sintassi *estesa* S[i:j:k]
- *k* è il parametro *stride* della slicing, cioè la distanza tra indici successivi
- Ad esempio, S[::2] è la sottosequenza che ha gli elementi con indice pari di S, mentre S[::-1] è S in ordine inverso

Operazioni sulle liste

- E’ possibile modificare una lista tramite assegnamento ad un indice:

```
x = [1,2,3,4]
x[1] = 42        # x diventa [1,42,3,4]
```

- Un’altra maniera di modificare una lista è usare una slice come destinazione in un’istruzione di assegnamento:

```
x[1:3] = [22,33,44]
```

Operazioni sulle liste

- La sottolista a sinistra e quella a destra dell'assegnamnto possono essere di dimensione qualunque: un assegnamento tramite slicing può quindi aggiungere o rimuovere elementi:

```
x = [1,22,33,44,4]

x[1:4] = [2,3]        # x vale [1,2,3,4]
```

Operazioni sulle liste

- Alcuni casi particolari:
- La lista vuota a destra dello ‘=‘ permette di rimuovere una parte della lista:
  L[i:j] = [] equivale a **del** L[i:j]
- Usare una slice vuota a sinistra dello ‘=‘ permette l’inserimento di elementi in una certa posizione:
  L[i:i] = [‘a’,’b’] inserisce ‘a’ e ‘b’ dopo l’elemento i-esimo di L
- Usare una slice “completa” (L[:]) a sinistra dello ‘=‘ sostituisce la lista originale

Operazioni sulle liste

- Gli oggetti di tipo lista ammettono l'uso degli operatori * e +, anche in versione aumentata (*= e +=)
- L'istruzione L+=L1 aggiunge gli elementi di L1 alla fine di L
- L*=n aggiunge n copie di L alla fine di L
- E' possibile eliminare uno o più elementi da una lista tramite del:
  del x[1]
  del x[1:3]

Operazioni sulle liste

- Esistono diversi metodi applicabili alle liste
- Alcuni (*non-mutating*) non modificano la lista originale, altri (*mutating*) no
- Esempio di metodi *non-mutating*:

```
L.count(x)   #rende il numero di occorrenze di x
L.index(x)   #rende l'indice della prima occorr.
             #di x o lancia un'eccezione se non
             #presente
```

Operazioni sulle liste

- Esempi di metodi *mutating*:

```
L.append(x)  # aggiunge x alla fine di L
L.extend(l)  # aggiunge gli el. di l alla fine di L
L.insert(i,x)  # inserisce x in posiz. i-esima di L
L.remove(x) # rimuove la prima occorr. di x in L
L.pop(i)       # rende il valore in posiz. i-esima
               # (default i=0) e lo rimuove
L.reverse()  # inverte gli elementi di L
L.sort(f)      # ordina gli elementi di L usando la
              # funzione f (default=cmp)
```

Operazioni sulle liste

- Tutti i metodi *mutating* (eccetto *pop*) rendono *None*
- La funzione *f* di *sort* (se presente) prende in ingresso due argomenti e rende -1,0 o 1 a seconda che il primo elemento sia minore, uguale o maggiore del secondo

Operazioni sui dizionari

- La funzione *len* con argomento un dizionario rende il numero di coppie chiave/valore presenti nel dizionario
- L'operatore k **in** D verifica se k è una delle chiavi presenti nel dizionario D
- Analogamente k **not in** D verifica che k non sia presente ed equivale a **not** (k **in** D)
- Il valore corrispondente ad una certa chiave, si ottiene con indicizzazione D[k] e viene lanciata un eccezione se la chiave non è presente

Operazioni sui dizionari

- L’assegnamento con una chiave non ancora presente aggiunge un nuovo elemento nel dizionario:

  D[newkey] = value

- L’istruzione **del** D[k] rimuove dal dizionario l’elemento corrispondente alla chiave k o lancia un’eccezione se non è presente

Operazioni sui dizionari

- Gli oggetti di tipo dizionario hanno vari metodi (mutating e non-mutating)
- Alcuni metodi mutating sono:

```
D.clear()        #rimuove tutti gli elementi di D
D.update(D1) #per ogni k di D1 imposta
                 #D[k] = D1[k]
D.setdefault(k,x)      #rende D[k] se k è presente
                       #in D altrimenti imposta
                       #D(k) = x e rende x
D.popitem()   #rende (rimuovendolo) un
              #elemento a caso di D
```

Operazioni sui dizionari

- Alcuni metodi non-mutating sono:

```
D.copy()        # rende una copia (shallow) di D
D.has_key(k) # rende True se k è una chiave di D
D.items()       # rende una lista con le coppie di D
D.keys()        # rende una lista con le chiavi di D
D.values()      # rende una lista con i valori di D
D.iteritems()  # rende un iteratore sulle coppie
D.iterkeys()   # rende un iteratore sulle chiavi
D.itervalues() # rende un iteratore sui valori
D.get(k,x)      # rende D[k] se k esiste; x altrim.
```

Operazioni sui dizionari

- I metodi *items*, *keys* e *values* creano delle liste con gli elementi in ordine arbitrario
- Gli iteratori consumano meno memoria di una lista
- Non è possibile modificare un dizionario mentre si sta iterando su di esso (cosa invece possibile con le liste)
- Il metodo *popitem* può essere utilizzato come iterazione distruttiva su un dizionario
- *setdefault* ha risultato simile a get, ma se k non è presente in D, D[k] viene collegato ad x

Istruzione print

- L’istruzione *print* è seguita da zero o più espressioni separate da virgola
- Ogni espressione x è stampata come una stringa sulla destinazione corrispondente all’attributo *stdout* del modulo *sys*
- Al posto di ogni virgola viene inserito uno spazio e un ritorno a capo finale (tranne nel caso in cui l’ultimo elemento sia seguito da una virgola)
- *print* è il modo più semplice per ottenere degli output

Controllo di flusso

- Il *controllo di flusso* di un programma è l'ordine con cui il codice viene eseguito
- Un programma Python è regolato da *istruzioni condizionali*, *loop* e *chiamate di funzione*
- Anche le *eccezioni* influenzano il controllo di flusso

L’istruzione if

- E’ un’istruzione composta dalle clausole **if**, **elif** ed **else** ed ha la seguente sintassi:

  if espressione**:**
  istruzione/i
  elif espressione**:**
  istruzione/i
  elif espressione**:**
  …
  else espressione**:**
  istruzione/i

L’istruzione if

- Le clausole **elif** ed **else** sono opzionali
- NON esiste un’istruzione diretta tipo *switch* ma si deve ricondurre alle forme precedenti
- Un esempio…

if x < 0: **print** “x è negativo”
elif x % 2: **print** “x è positivo e dispari”
else: **print** “x è pari e non negativo”

L’istruzione if

- Se una clausola ha istruzioni multiple, queste sono disposte in linee logiche separate e indentate della stessa quantità
- Il blocco termina quando l’indentazione ritorna al livello dell’header della clausola
- Se l’istruzione è singola, è possibile metterla sulla stessa linea della clausola o su una linea logica immediatamente successiva con un livello in più d’indentazione

L’istruzione if

- Ad esempio:

```
if x < 0:
    print “x è negativo”
elif x % 2:
    print “x è positivo e dispari”
else:
    print “x è pari e non negativo”
```

- Quest’ultimo “stile” è considerato generalmente più leggibile

L’istruzione while

- L’istruzione **while** gestisce l’esecuzione ripetuta di un’istruzione od un blocco, ed è controllata da un’espressione condizionale.
- La sintassi è la seguente:

  while espressione**:**
      istruzione/i
- E’ possibile anche includere una clausola **else** e usare le istruzioni **break** e **continue**

L’istruzione while

- Esempio:

```
count = 0
while x > 0:
        x = x // 2
        count += 1
print “risultato =“, count
```

L’istruzione while

- Se la condizione del loop non diventa mai falsa, il ciclo non termina finché non viene lanciata un eccezione o viene incontrata un’istruzione *break*
- Un loop all’interno di una funzione termina anche nel caso in cui si incontra l’istruzione *return* (che fa terminare l’intera funzione)

L’istruzione for

- Supporta l’esecuzione ripetuta di un’istruzione o un blocco, controllata da un’espressione *iterabile*.
- La sintassi è la seguente:

  for destinazione **in** iterabile:
  istruzione/i

- Notare che la keyword **in** fa parte della sintassi dell’istruzione ed ha significato diverso dal controllo di appartenenza di un oggetto ad un contenitore

L’istruzione for

- Anche l’istruzione **for** supporta la clausola **else** e le istruzioni **break** e **continue**
- *iterabile* può essere una qualunque espressione Python accettata dalla funzione **iter**, che restituisce un oggetto iteratore
- *destinazione* è un’identificatore detto “*variabile di controllo del ciclo*” (assumerà, nell’ordine, il valore di ogni elemento dell’iteratore)

L’istruzione for

- È possibile usare più elementi nella destinazione:

  for key, value **in** d.items():
      if not key **or not** value: **del** d[key]

- Se l’iteratore si riferisce ad un oggetto mutabile, tale oggetto non deve essere modificato durante un loop (*for* o *while*) su di esso!
  L’esempio precedente è corretto perché *items* restituisce una copia degli elementi in una lista

L’istruzione for

- La *variabile di controllo* può essere modificata durante l’esecuzione del ciclo ma riacquisterà il valore corretto durante l’iterazione successiva
- Il loop non verrà mai eseguito se l’iteratore non contiene elementi (in questo caso la variabile del ciclo non risulta definita)
- Alla fine del ciclo la variabile di controllo è riutilizzabile e contiene l’ultimo valore assunto dopo le iterazioni

L’istruzione for

- L’istruzione *for* chiama implicitamente la funzione *iter* per ottenere un iteratore
- Infatti, l’istruzione:

for x **in** c:
    istruzione/i

è equivalente a:

```
_temp_iterator = iter(c)
while True:
    try: x = _temp_iterator.next()
    except StopIteration: break
    istruzione/i
```

L’istruzione for

- Gli iteratori sono stati introdotti a partire dal Python 2.2
- Nelle versioni precedenti era richiesta una sequenza indicizzabile
- Per mezzo degli iteratori, il ciclo *for* può ora essere utilizzato anche in strutture più generali delle sequenze (per cui sia stato definito il metodo speciale *__iter__)*

Range

- Per eseguire agevolmente dei loop su una sequenza di interi, il Python mette a disposizione le due funzioni **range** e **xrange**
- Il modo più semplice di ripetere un'istruzione n volte è infatti:

```
for i in xrange(n):
    istruzione/i
```

Range

- **range**(x) rende una lista i cui elementi sono gli interi successivi che vanno da 0 (compreso) a *x* (escluso)
- **range**(x,y) rende una lista avente gli interi tra *x* (compreso) e *y* (escluso); se x >= y si ottiene una lista vuota
- **range**(x,y,step) rende una lista di interi tra *x* (incluso) a *y* (escluso) in modo che la differenza tra due numeri consecutivi sia pari a *step*; se *step* è negativo si ottiene una sequenza decrescente

Xrange

- **range** restituisce una normale lista, **xrange** rende invece un oggetto speciale, utilizzabile per le iterazioni
- **xrange** occupa meno memoria di **range** per questo uso specifico
- al di là di questa differenza, **range** e **xrange** sono perfettamente interscambiabili

List comprehensions

- Un utilizzo tipico del loop basato su **for** è di prelevare gli elementi di una sequenza, eseguire delle operazioni e costruire una lista con i risultati
- Il Python fornisce un'espressione particolare, detta "*list comprehension*" che permette di eseguire in modo conciso le operazioni precedenti

List comprehensions

- La sintassi è la seguente:
  [*espressione* **for** *destinazione* **in** *iterabile* *clausole*]
  destinazione e *iterabile* sono le stesse del ciclo **for**.
- *espressione* deve essere racchiusa tra parentesi se indica una tupla
- *clausole* è una serie di zero o più clausole espresse in due forme possibili:
  - **for** destinazione **in** iterabile
  - **if** espressione

List comprehensions

- Ad esempio:

  res = [x+1 **for** x **in** s]

  equivale a:

```
res = []
for x in s:
    res.append(x+1)
```

List comprehensions

- Una list comprehension che usa l'if:

```
res = [x+1 for x in s if x>24]
```

equivale a:

```
res = []
for x in s:
    if x>24:
        res.append(x+1)
```

List comprehensions

- Si possono usare anche più for:

  res = [x+y **for** x **in** s **for** y **in** t]

  che equivale a:

```
res = []
for x in s:
    for y in t:
        res.append(x+y)
```

Istruzione break

- E’ consentita soltanto dentro il corpo di un loop
- Forza la terminazione del loop
- In caso di loop annidati, viene terminato solo il loop più interno
- Esempio:

```
while True:
    x = get_next()
    if not keep_looping(x):
    process (x)
```

Istruzione continue

- E’ consentita soltanto all’interno del corpo di un loop
- Termina l’iterazione corrente di un loop e l’esecuzione continua con l’iterazione successiva
- Esempio:

```
for x in s:
    if not seems_ok(x): continue
    if final_check(x):
       process(x)
```

Clausola else

- Entrambe le istruzioni **while** e **for** possono terminare con una clausola **else** opzionale
- Le istruzioni dopo l'**else** vengono eseguite quando il loop termina in modo naturale (fine dell'iteratore del **for** o condizione del **while** che diventa False) ma non quando viene interrotto (da **break**, **return** o un'*eccezione*)

Clausola else

- Esempio:

```
for x in s:
    if is_ok(x): break
else:
   print "NON trovato!"
   x = None
```

Istruzione pass

- Il corpo di un istruzione composta non può essere vuoto: deve contenere almeno una istruzione
- L'istruzione **pass**, che non esegue alcuna azione, può essere usata laddove è richiesta sintatticamente un'istruzione. Ad esempio:

```
if cond(x):
    process(x)
elif x > y:
   pass
else: process_default(x)
```

Funzioni

- In un tipico programma Python, la maggior parte delle istruzioni sono organizzate all'interno di *funzioni*
- Una *funzione* è un gruppo di istruzioni che viene eseguito su richiesta
- Il Python fornisce alcune funzioni di base e permette la definizione di nuove
- La richiesta di eseguire una funzione è nota come "*chiamata di funzione*" (*function call*)

Funzioni

- Le funzioni sono oggetti e sono gestite come gli altri oggetti
- E’ dunque possibile passare una funzione come argomento in una chiamata di un’altra funzione
- Una funzione può rendere come risultato un’altra funzione e può essere assegnata ad una variabile
- Le funzioni possono essere gli elementi di una sequenza o anche le chiavi di un dizionario

L’istruzione def

- E’ il modo più comune per definire una funzione ed ha la seguente sintassi:

  def *nome-funzione* (*parametri*):
  istruzione/i

- *nome-funzione* è un identificatore; è una variabile riferita all’oggetto funzione
- *parametri* è una lista opzionale di identificatori, detti “*parametri formali*”, separati da virgole

L’istruzione def

- La sequenza di istruzioni costituenti il corpo della funzione, non viene eseguita all’esecuzione di **def**, ma quando la funzione è chiamata
- Una funzione può avere zero o più istruzioni **return**
- Esempio:

```
def double(x):
    return x*2
```

L’istruzione def

- Se una funzione ha come parametri dei semplici identificatori, questi sono obbligatori: ogni chiamata a tale funzione deve specificare un valore corripondente ad ogni parametro
- E’ possibile specificare dei parametri opzionali (che assumono quindi un valore di *default*, nel caso in cui non vengano passati alla funzione) usando la forma:
  identificatore = *espressione*

L’istruzione def

- Esempio:

```
def f(x, y=[]):
    y.append(x)
    return y

print f(23)      # visualizza [23]
print f(42)      # visualizza [23,42]
```

L’istruzione def

- Se si vuole usare ogni volta una nuova lista vuota, occorre scrivere:

```
def f(x, y=None):
    if y is None: y = []
    y.append(x)
    return y

print f(23)       # visualizza [23]
print f(42)       # ora visualizza solo [42]
```

Parametri * e **

- Esistono due forme speciali nella lista parametri:
  *identificatore1
  e
  **identificatore2
- Se sono entrambe presenti, la forma con i due asterischi deve essere l'ultima
- *identificatore1 permette di specificare argomenti extra detti "*positional arguments*"
- **identificatore2 permette di specificare argomenti extra detti "*named arguments*"

Parametri * e **

- Alla forma *identificatore1 è associata una *tupla*, i cui elementi sono i "*positional arguments*"
- Alla forma **identificatore2 è associato un *dizionario* i cui elementi sono i nomi e i valori dei "*named arguments*"
- Esempio (funzione che somma i parametri):

```
def sum(*numbers):
    result = 0
    for x in numbers:
        result += x
    return result
```

Parametri * e **

- Esempio di una funzione che crea un dizionario:

```
def adict(**kwds):
    return kwds

print adict(a=23, b=42)    #scrive {‘a’:23,’b’:42}
```

Attributi degli oggetti funzione

- Ogni oggetto di tipo funzione ha degli attributi:
- **func_name** (o **__name__**) è read-only e contiene l'*identificatore* usato per la funzione nella **def**
- **func_defaults** contiene la *tupla* dei valori di default per i parametri opzionali
- **func_doc** (o **__doc__**) è una *stringa* contenente la documentazione della funzione (disponibile anche durante l'esecuzione del programma); La *documentazione* si specifica anche con una stringa dopo la riga **def** (tipicamente tra tripli apici)

Istruzione return

- E’ permessa solo all’interno del corpo di una funzione e ha lo scopo di *terminarla* restituendo eventualmente un’espressione
- Una funzione rende **None** se termina senza aver incontrato un’istruzione **return**
- E’ preferibile, per questioni di stile, usare sempre **return None** in luogo di un semplice **return**

Chiamata di funzione

- E’ un’espressione con la seguente sintassi:

  oggetto-funzione (*argomenti*)

- *oggetto-funzione* può essere un qualunque riferimento a un oggetto funzione (spesso è il suo nome)
- *argomenti* è una serie di zero o più espressioni separate da virgola, che danno i valori ai corrispondenti parametri formali della funzione

Chiamata di funzione

- In Python gli argomenti sono passati per valore: se l'argomento è una variabile la funzione riceve l'oggetto (valore) cui la variabile si riferisce e non la variabile in sé!
- Tuttavia, se è passato come argomento un oggetto mutabile, la funzione riceve l'oggetto stesso e non una sua copia (ricollegare una variabile e modificare un oggetto sono concetti differenti)

Chiamata di funzione

- Esempio:

```
def f (x,y):
    x = 23
    y.append(42)

a=77
b=[99]
f (a,b)
print a,b        # risultato: 77 [99, 42]
```

Tipi di argomenti

- Gli argomenti che sono soltanto espressioni sono definiti “*argomenti posizionali*”
- Ciascun *argomento posizionale* fornisce il valore al *parametro formale* che corrisponde alla stessa posizione nella definizione della funzione

Tipi di argomenti

- In una chiamata di funzione zero o più "*positional arguments*" possono essere seguiti da zero o più "*named arguments*" con la sintassi:

  identificatore = *espressione*

- *identificatore* deve essere uno dei parametri formali della funzione
- *espressione* attribuisce il valore al parametro formale con quel nome

Tipi di argomenti

- Una chiamata di funzione deve fornire o mediante *parametro posizionale* o per *nome*, esattamente un valore per ogni parametro obbligatorio e zero o più valori per ogni parametro opzionale
- Esempio:

```
def divide (divisor, dividend):
    return dividend // divisor
print divide (12, 94)
print divide (dividend=94, divisor=12)
```

Tipi di argomenti

- Esempio:
  def f (middle, begin=‘init’, end=‘finis’):
  return begin+middle+end
  print f(‘tini’, end=‘’)

- Grazie ai “*named arguments*” la funzione può essere chiamata specificando il valore obbligatorio *middle* più uno dei parametri opzionali, lasciando al secondo parametro il valore di default