Librerie, condizioni e cicli

Corso Programmazione Python 2024

Modulo 2

Luca Di Pietro Martinelli

Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)

Il concetto di funzione

La **funzione** è uno degli elementi chiave della programmazione in generale, e questo si rispecchia completamente anche in Python.

Una funzione può essere definita come un blocco di codice riutilizzabile che esegue un insieme specifico di istruzioni quando viene richiamato. Nel caso in cui una funzione si riferisca a un oggetto di una certa tipologia questa è chiamata **metodo**.

Abbiamo già utilizzato una funzione senza saperlo:

```
print("Hello World")
```

L'utilizzo delle funzioni all'interno di un programma porta diversi vantaggi:

- maggiore organizzazione del codice
- alta leggibilità del codice
- forte riduzione della duplicazione del codice.

Vedremo poi più avanti come definire da noi nuove funzioni.

Il concetto di funzione: parametri

La stringa di testo `"Hello World"` inserita tra le parentesi è detta **parametro** (o argomento). In questo caso dunque abbiamo usato la funzione `print()` con un solo parametro in input per poter eseguire la stampa in console, ma esistono funzioni che richiedono anche più di un parametro (come la `print` stessa).

Considerando una funzione generica definita nel seguente modo:

```
def function_name(a, b, c):
    print(a + b + c)
```

il passaggio di parametri alla funzione può avvenire secondo due modalità:

- *Posizionale:* i parametri passati devono necessariamente rispettare l'ordine specificato nella definizione della funzione
  Esempio: `function_name("ciao ", "come ", "stai") # Stampa: ciao come stai`

- *Chiave-valore:* specificando i nomi dei parametri è possibile specificare anche un ordine differente
  Esempio: `function_name(b="come ", c="stai", a="ciao ") # Stampa: ciao come stai`

Le funzioni built-in: input()

Le **funzioni built-in** di Python ci permettono di utilizzare alcune funzionalità già "pronte all'uso".
L'insieme di più funzioni built-in compone la cosiddetta **libreria**.

La funzione built-in `input()` può essere utilizzata in due forme:

```
valore = input()
```
in cui l'interprete resta in attesa che l'utente inserisca nella shell, attraverso la tastiera, una sequenza di caratteri che dovrà essere conclusa dal tasto <INVIO>. Questa sequenza di caratteri è poi inserita in una stringa che viene restituita come valore.

```
dato = input(messaggio)
```
in cui l'interprete stampa `messaggio`, che deve essere una stringa, nella shell, poi procede come indicato sopra. La stringa `messaggio` viene di norma usata per indicare all'utente che il programma è in attesa di ricevere un particolare dato in ingresso.

Le funzioni built-in: print()

Sintassi:
`print(espressione)`

Regole:
- non devono esserci spazi prima di `print`
- `espressione` deve essere un'espressione valida del linguaggio Python

Il risultato è che nella console viene mostrato il valore di `espressione`.
È anche possibile mostrare con una sola istruzione `print` i valori di un numero qualsiasi di espressioni, con la seguente sintassi:

```
print(espressione1, espressione2, ...)
```

In questo caso, i valori delle espressioni vengono mostrati su una stessa riga, separati da una spaziatura.

Parametri opzionali:
`end`: specifica una stringa che viene stampata dopo l'output della funzione
`sep`: specifica una stringa che separa i parametri passati alla funzione

Le funzioni built-in: type()

Python ha una funzione built-in denominata `type`, che può essere usata per ottenere il tipo di ogni variabile o valore costante.

Esempi:

- `type(11)`
  - `Console: <class 'int'>`
- `type(5.2)`
  - `Console: <class 'float'>`
- `type("stringa di prova")`
  - `Console: <class 'str'>`
- `type(True)`
  - `Console: <class 'bool'>`

Altre funzioni built-in

`len(stringa)`
restituisce il numero di caratteri di una stringa

`abs(numero)`
restituisce il valore assoluto di un numero

`str(espressione)`
restituisce una stringa composta dalla sequenza di caratteri corrispondenti alla rappresentazione del valore di espressione (che può essere di un qualsiasi tipo: numero, stringa, valore logico, ecc.)

Altre funzioni built-in

È possibile "forzare" il tipo di una variabile modificandolo attraverso la cosiddetta operazione di **casting esplicito**, che si differenzia da quella **implicita** che abbiamo già visto essere eseguito automaticamente da Python ad esempio nelle operazioni tra numeri interi e decimali.
Di seguito alcune funzioni che realizzano il casting esplicito:

`int(numero)`
Restituisce la parte intera di `numero`

`float(numero)`
Restituisce il valore `numero` come numero frazionario (floating point)

`int(stringa)`
Se `stringa` contiene la rappresentazione di un numero intero, restituisce il numero corrispondente a tale valore, in caso contrario produce un errore

`float(stringa)`
Se `stringa` contiene la rappresentazione di un numero qualsiasi (sia intero che frazionario), restituisce il suo valore espresso come numero frazionario, in caso contrario produce un errore

Chiamata di funzione

Sintassi:

```
nome_funzione(par1, par2, ..., parn)
```

- `par1`, `par2`, …, `parn` sono espressioni Python i cui valori costituiranno gli argomenti della funzione
- il numero degli argomenti e il tipo di ciascuno di essi (per es., numeri interi, numeri frazionari, stringhe, valori logici) dipende dalla specifica funzione; se il tipo di un argomento non è tra quelli previsti, si otterrà un messaggio d'errore
- come tutte le espressioni, anche la chiamata di una funzione produce un valore: questo coincide con il valore restituito dalla funzione

Funzioni built-in: librerie

L'insieme delle funzioni built-in, dette anche predefinite, viene detto libreria.

Esempi di librerie:

- funzioni matematiche (libreria `math`)
- funzioni per la generazione di numeri pseudo-casuali (libreria `random`)

Importare le librerie

Per poter chiamare una funzione di librerie come `math` e `random` è necessario utilizzare la combinazione `from import`.

Sintassi:

```
from nome_libreria import nome_funzione
```

- `nome_libreria` è il nome simbolico di una libreria
- `nome_funzione` può essere:
  - il nome di una specifica funzione di tale libreria (questo consentirà di usare solo tale funzione)
  - il simbolo * indicante tutte le funzioni di tale libreria

Se la combinazione `from import` non viene usata correttamente la chiamata di funzione produrrà un errore.

La libreria math

Di seguito sono mostrate alcune delle funzioni messe a disposizione all'interno della libreria `math`:

Funzione	Descrizione
`exp(x)`	e^x
`log(x)`	$\ln x$
`pow(x, y)`	x^y
`sqrt(x)`	$\sqrt{x}$

La libreria random

Di seguito sono mostrate alcune delle funzioni messe a disposizione all'interno della libreria `random`:

Funzione	Descrizione
`random()`	Genera un numero reale nell'intervallo [0, 1), da una distribuzione di probabilità uniforme (cioè, ogni valore di tale intervallo ha la stessa probabilità di essere "estratto")
`uniform(a, b)`	Come sopra, nell'intervallo [*a*, *b*) (gli argomenti sono numeri qualsiasi)
`randint(a, b)`	Genera un numero intero nell'insieme {a, …,b} da una distribuzione di probabilità uniforme (gli argomenti devono essere numeri interi)

Ogni chiamata di tali funzioni produce un numero pseudo-casuale, indipendente (in teoria) dai valori prodotti dalle chiamate precedenti.

Metodi delle stringhe

Le funzioni applicabili a stringhe di testo, definiti **metodi** poiché relativi all'oggetto di tipo stringa, sono molteplici e permettono di sfruttare funzionalità molto comode da poter riutilizzare quando si vuole all'interno del proprio codice.
La sintassi di utilizzo è la seguente:

```
stringa.nome_metodo(parametri)
```

Di seguito i metodi delle stringhe più utilizzati:

`startswith(insieme_di_caratteri)`
Indica se una data stringa inizia con determinato insieme di caratteri

`endswith(insieme_di_caratteri)`
Indica se una data stringa finisce con determinato insieme di caratteri

`upper()`
Restituisce una versione della stringa con tutti caratteri maiuscoli

`lower()`
Restituisce una versione della stringa con tutti caratteri minuscoli

Metodi delle stringhe

Di seguito i metodi delle stringhe più utilizzati:

`isalnum()`
Indica se una data stringa contiene solo caratteri alfanumerici

`isalpha()`
Indica se una data stringa contiene solo caratteri dell'alfabeto

`isdecimal()`
Indica se una data stringa contiene solo caratteri numerici

`isspace()`
Indica se una data stringa contiene solo spazi

Espressioni condizionali

Sintassi:

```
espressione1 operatore espressione2
```

- `espressione1` e `espressione2` sono due espressioni Python qualsiasi, definite come visto in precedenza (possono quindi contenere nomi di variabili, purché a tali variabili sia stato già assegnato un valore).
- `operatore` è un simbolo che indica il tipo di confronto tra le due espressioni.

Operatori di confronto

Nel linguaggio Python sono disponibili i seguenti operatori di confronto, che possono essere usati sia su espressioni aritmetiche che su espressioni composte da stringhe:

Simbolo	Operatore
==	" Uguale a "
!=	" Diverso da "
<	" Minore di "
<=	" Minore o uguale a "
>	" Maggiore di "
>=	" Maggiore o uguale a "

Espressioni condizionali composte

Sintassi:

- `espr_cond_1 and espr_cond_2`
- `espr_cond_1 or espr_cond_2`
- `not espr_cond`

`espr_cond_1`, `espr_cond_2` e `espr_cond` sono espressioni condizionali qualsiasi (quindi, possono essere a loro volta espressioni composte).

Gli operatori `and`, `or` e `not` sono detti operatori logici e corrispondono rispettivamente ai connettivi logici del linguaggio naturale “e”, “oppure” e “non”.

È possibile usare le parentesi tonde per definire l'ordine degli operatori logici.

Istruzione condizionale: if-else

Sintassi:

```
if espr_cond:
    sequenza_di_istruzioni_1
else:
    sequenza_di_istruzioni_2
```

- le keyword `if` e `else` devono avere lo stesso rientro
- espr_cond è un'espressione condizionale
- `sequenza_di_istruzioni_1` e `sequenza_di_istruzioni_2` sono due sequenze di una o più istruzioni qualsiasi
- ciascuna istruzione deve essere scritta in una riga distinta, con un rientro di almeno un carattere
- il rientro deve essere identico per tutte le istruzioni

Istruzione condizionale: varianti

Variante if:

```
if espr_cond:
    sequenza_di_istruzioni
```

Variante elif:

```
if espr_cond_1:
    sequenza_di_istruzioni_1
elif espr_cond_2:
    sequenza_di_istruzioni_2
else:
    sequenza_di_istruzioni_3
```

Istruzione condizionale: rientri

I rientri sono l'unico elemento sintattico che indica quali istruzioni fanno parte di un'istruzione condizionale. Le istruzioni che seguono un'istruzione condizionale (senza farne parte) devono quindi essere scritte senza rientri.

Come esempio, si considerino le due sequenze di istruzioni:

```
if x > 0:
    print("A")
    print("B")
```

```
if x > 0:
    print("A")
print("B")
```

Nella sequenza a sinistra le due istruzioni `print` sono scritte con un rientro rispetto a `if`: questo significa che fanno entrambe parte dell'istruzione condizionale e quindi verranno eseguite solo se la condizione `x > 0` sarà vera.

Nella sequenza a destra solo la prima istruzione `print` dopo `if` è scritta con un rientro e quindi solo essa fa parte dell'istruzione condizionale, mentre la seconda istruzione `print` verrà eseguita dopo l'istruzione condizionale, indipendentemente dal valore di verità della condizione `x > 0`.

Istruzioni condizionali annidate

Una istruzione condizionale può contenere al suo interno istruzioni qualsiasi, quindi anche altre istruzioni condizionali. Si parla, in questo caso, di istruzioni annidate.

L'uso di istruzioni condizionali annidate consente di esprimere la scelta tra più di due sequenze di istruzioni alternative.

Una istruzione condizionale annidata all'interno di un'altra andrà scritta tenendo conto che:

- le keyword `if`, `elif` ed `else` (se presenti) devono essere scritte con un rientro rispetto a quelle dell'istruzione condizionale che le contiene
- le sequenze di istruzioni che seguono `if` , `elif` ed `else` devono essere scritte con un ulteriore rientro.

Istruzioni condizionali annidate

Esempio:

```
if x > 0:
    print("A")
    if y == 1:
        print("B")
    else:
        print("C")
        print ("D")
else:
    print("E")
```

Istruzione while

L'istruzione `while` permette di eseguire un blocco di codice finché una determinata condizione è e resta `True`.

Sintassi:

```
while espr_cond:
    sequenza_di_istruzioni
```

- la keyword `while` deve essere scritta senza rientri
- `espr_cond` è una espressione condizionale qualsiasi
- `sequenza_di_istruzioni` consiste in una o più istruzioni qualsiasi. Ciascuna di tali istruzioni deve essere scritta in una riga distinta, con un rientro di almeno un carattere. Il rientro deve essere identico per tutte le istruzioni della sequenza

Istruzione while: break

L'istruzione `break` serve per terminare un ciclo `while` prematuramente: non appena quest'espressione viene letta e processata all'interno del ciclo, Python blocca il loop istantaneamente.

Esempio:

```
run = True
counter = 0
stop = 10

while run == True:
    print(counter)
    counter += 1
    if counter > stop:
        print("Sto uscendo dal loop...")
        break
```

Istruzione while: continue

L'istruzione `continue` invece è simile al `break`, ma invece di interrompere il ciclo fa saltare tutto il codice dopo l'istruzione e fa ripartire Python dalla prima riga del ciclo.

Esempio:

```
run = True
skip = 5
counter = 0

while counter < 10:
    counter += 1
    if counter == skip:
        print("Saltato")
        continue
    print(counter)
```

Istruzione while: variante else

Tramite l'istruzione `else` è possibile aggiungere al termine dell'istruzione `while` un blocco di codice che vogliamo venga eseguito una volta che la condizione non è più verificata.

Esempio:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i non è più < 6")
```