Jacopo De Angelis

Manuale per Python 3

Integrato con l’uso della libreria Pandas per l’analisi dei dati

Sommario

[1 Elementi preliminari 4](#_Toc13737278)

[1.1 Operatori aritmetici 4](#_Toc13737279)

[1.2 Valori e tipi 4](#_Toc13737280)

[1.3 Linguaggi formali e linguaggi naturali 4](#_Toc13737281)

[2 Variabili, espressioni ed istruzioni 6](#_Toc13737282)

[2.1 Istruzione di assegnazione 6](#_Toc13737283)

[2.2 Nomi delle variabili 6](#_Toc13737284)

[2.3 Espressioni ed istruzioni 6](#_Toc13737285)

[2.4 Modalità script 6](#_Toc13737286)

[2.5 Ordine delle operazioni 7](#_Toc13737287)

[2.6 Operazioni sulle stringhe 7](#_Toc13737288)

[2.7 Commenti 7](#_Toc13737289)

[2.8 Debug 7](#_Toc13737290)

[2.8.1 Errori di sintassi 7](#_Toc13737291)

[2.8.2 Errori in esecuzione 7](#_Toc13737292)

[2.8.3 Errori di semantica 7](#_Toc13737293)

[2.9 Differenze di scrittura rispetto ad altri linguaggi 7](#_Toc13737294)

[3 Funzioni 9](#_Toc13737295)

[3.1 Chiamate di funzione 9](#_Toc13737296)

[3.2 Funzioni matematiche 9](#_Toc13737297)

[3.3 Aggiungere nuove funzioni 9](#_Toc13737298)

[3.4 Diagrammi di stack 9](#_Toc13737299)

[3.5 Perché usare le funzioni? 10](#_Toc13737300)

[3.6 Funzioni built-in 10](#_Toc13737301)

[4 Progettazione dell’interfaccia (turtle) 13](#_Toc13737302)

[4.1 Il modulo turtle 13](#_Toc13737303)

[4.2 Incapsulamento 13](#_Toc13737304)

[4.3 Generalizzazione 14](#_Toc13737305)

[4.4 Progettare un’interfaccia 14](#_Toc13737306)

[4.5 Tecnica di sviluppo 14](#_Toc13737307)

[4.6 Stringa di documentazione 14](#_Toc13737308)

[5 Istruzioni condizionali e ricorsione 16](#_Toc13737309)

[5.1 if 16](#_Toc13737310)

[5.2 Input da tastiera 16](#_Toc13737311)

[6 Iterazione 17](#_Toc13737312)

[6.1 while 17](#_Toc13737313)

[6.2 break 17](#_Toc13737314)

[6.3 for 17](#_Toc13737315)

[6.4 Pass 17](#_Toc13737316)

[7 Stringhe 18](#_Toc13737317)

[7.1 Attraversamento 18](#_Toc13737318)

[7.2 Slicing 18](#_Toc13737319)

[7.3 in 18](#_Toc13737320)

[7.4 Confronto di stringhe 18](#_Toc13737321)

[7.5 Funzioni sulle stringhe 18](#_Toc13737322)

[8 Liste 21](#_Toc13737323)

[8.1 Una lista è una sequenza 21](#_Toc13737324)

[8.2 Le liste sono mutabili 21](#_Toc13737325)

[8.3 Operazioni sulle liste 21](#_Toc13737326)

[8.4 Slicing delle liste 21](#_Toc13737327)

[8.5 Lista e stringhe 21](#_Toc13737328)

[8.6 Oggetti e valori 22](#_Toc13737329)

[8.7 Liste come argomenti 22](#_Toc13737330)

[8.8 Funzioni sulle liste 22](#_Toc13737331)

[9 Dizionari 24](#_Toc13737332)

[9.1 Lookup inverso 24](#_Toc13737333)

[9.2 Memoizzazione 25](#_Toc13737334)

[9.3 Variabili globali 25](#_Toc13737335)

[9.4 Metodi sui dizionari 25](#_Toc13737336)

[10 Tuple 27](#_Toc13737337)

[10.1 Le tuple sono immutabili 27](#_Toc13737338)

[10.2 Assegnazione di tupla 27](#_Toc13737339)

[10.3 Tuple come valori di ritorno 27](#_Toc13737340)

[10.4 Tuple di argomenti a lunghezza variabile 27](#_Toc13737341)

[10.5 Liste e tuple 28](#_Toc13737342)

[10.6 Dizionari e tuple 29](#_Toc13737343)

[10.7 Sequenze di sequenze 29](#_Toc13737344)

[10.8 Metodi sulle tuple 29](#_Toc13737345)

[11 File 30](#_Toc13737346)

[11.1 Persistenza 30](#_Toc13737347)

[11.2 Lettura e scrittura 30](#_Toc13737348)

[11.3 L’operatore di formato 30](#_Toc13737349)

[11.4 Nomi di file e percorsi 31](#_Toc13737350)

[11.5 Gestire le eccezioni 31](#_Toc13737351)

[11.6 Database 31](#_Toc13737352)

[11.7 Pickling 32](#_Toc13737353)

[11.8 Pipe 33](#_Toc13737354)

[11.9 Scrivere moduli 33](#_Toc13737355)

[12 Classi e oggetti 34](#_Toc13737356)

[12.1 Tipi personalizzati 34](#_Toc13737357)

[12.2 Attributi 34](#_Toc13737358)

[12.3 Istanze come valori di ritorno 34](#_Toc13737359)

[12.4 Copia 34](#_Toc13737360)

[13 Classi e funzioni 36](#_Toc13737361)

[13.1 Funzioni pure 36](#_Toc13737362)

[13.2 Modificatori 36](#_Toc13737363)

[13.3 Sviluppo prototipale e sviluppo pianificato 36](#_Toc13737364)

[14 Classi e metodi 37](#_Toc13737365)

[14.1 Funzionalità orientate agli oggetti 37](#_Toc13737366)

[14.2 Stampa di oggetti 37](#_Toc13737367)

# Elementi preliminari

Sito per utilizzare Python senza installarlo: PythonAnywhere (<https://www.pythonanywhere.com/>).

Utilizzeremo Python 3.

## Operatori aritmetici

* Addizione: +, +=
* Sottrazione: -, -=
* Moltiplicazione: \*, \*=
* Divisione: /
* Divisione intera: //
* Elevamento a potenza: \*\* o ^ (operazione bitwise[[1]](#footnote-1))
* Resto: %

## Valori e tipi

Uno dei vantaggi di Python è la non necessità di esplicitare il tipo di variabile durante la sua dichiarazione. Per creare una variabile intera, ad esempio, basta scrivere “var = 42”.

I tipi di base sono:

* Intero
* Virgola mobile
* Stringa
* Carattere
* Booleano

Per conoscere il tipo di una variabile si può utilizzare la funzione type con sintassi “type(variabile)”

## Linguaggi formali e linguaggi naturali

I linguaggi naturali sono le lingue parlate dagli esseri umani.

I linguaggi formali sono linguaggi appositamente creati per specifiche applicazioni. I linguaggi di programmazione sono linguaggi formali progettati per esprimere dei calcoli.

Le regole sintattiche hanno due aspetti, che riguardano i simboli e la struttura:

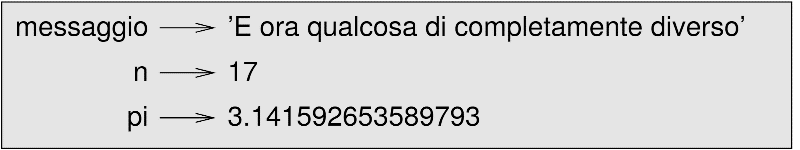
* I simboli (token) sono gli elementi di base del linguaggio, come le parole e i numeri
* La struttura è come vengono disposti i simboli

Quando si legge una frase o un’espressione bisogna analizzare la struttura della frase. Questo processo di analisi è chiamato parsing.

|  |  |  |
| --- | --- | --- |
| Diversità | | |
|  | **Linguaggi naturali** | **Linguaggi formali** |
| Ambiguità | Ne sono pieni | Sono progettati per limitare le ambiguità, ovvero una singola istruzione non può essere interpretata in modi differenti |
| Ridondanza | Per risolvere l’ambiguità sono pieni di ridondanze | Meno ridondanti e più concisi |
| Letteralità | Contengono metafore e frasi il cui significato e significante non coincidono | Un’istruzione ha un’interpretazione unica ed evidente |

# Variabili, espressioni ed istruzioni

## Istruzione di assegnazione

Un’istruzione di assegnazione serve a creare una nuova variabile specificandone il nome e assegnandole un valore.

Un modo frequente di raffigurare le variabili è quello di scriverne il nome e disegnare una freccia che punta al loro valore. Questa illustrazione è chiamata diagramma di stato.

Figura 1 - Diagramma di stato

## Nomi delle variabili

I nomi possono contenere sia numeri che lettere ma non possono iniziare con un numero.

Per i nomi di variabile è convenzione utilizzare solo lettere minuscole con le parole intervallate da \_ (ad esempio “camel\_case”).

Un altro limite è quello delle parole chiave riservate, ovvero un elenco di parole che non possono essere usate come nomi di varabile.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parole chiave | | | | |
| False | class | finally | is | return |
| None | continue | for | lamda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |

Le parole chiave vengono evidenziate con un colore diverso, quindi non è necessario ricordarle a memoria.

## Espressioni ed istruzioni

Un’espressione è una combinazione di valori, variabili e operatori.

Quando viene scritta **un’espressione** al prompt dei comandi, l’interprete la valuta, cioè trova il valore dell’espressione.

Quando si scrive **un’istruzione**, l’interprete la esegue, cioè fa quello che l’istruzione dice di fare. In linea generale, le istruzioni non contengono valori.

## Modalità script

Fino ad ora abbiamo avviato Python in maniera interattiva, ovvero a riga di comando, cioè comunicando direttamente con l’interprete.

In alternativa alla riga di comando, si può scrivere e salvare un programma in un file di testo semplice, chiamato script, ed usare poi l’interprete in modalità script per eseguirlo. I file contenenti programmi Python hanno nomi che terminano con l’estensione “.py”.

## Ordine delle operazioni

Quando un’espressione contiene più operatori, la successione con cui viene eseguito il calcolo dipende dall’ordine delle operazioni. Per quelle matematiche, Python segue le classiche convenzioni matematiche.

## Operazioni sulle stringhe

Si possono eseguire due operazioni:

* Concatenamento: +
* Ripetizione: \*

## Commenti

Per scrivere un commento su di una sola riga basta anteporre # prima del commento.

Per fare un commento multilinea bisogna scrivere anteporre e posporre “””.

|  |  |
| --- | --- |
| Commento linea singola | Commento multilinea |
| # Questo è  # un commento | “””  Questo è  un commento  “”” |

## Debug

In un programma si possono verificare tre tipi di errori:

* Errori di sintassi
* Errori in esecuzione
* Errori di semantica

### Errori di sintassi

Il termine sintassi si riferisce alla struttura di un programma e alle regole che la governano.

Se c’è un errore di sintassi in qualche parte del programma, questo non verrà eseguito.

“2 + 8)” è ad esempio un errore di sintassi per la mancanza della prima parentesi.

### Errori in esecuzione

Questo tipo di errore non compare fino a quando il programma non viene eseguito. Questi errori sono anche detti eccezioni e devono essere gestiti direttamente dal programmatore tramite l’uso delle eccezioni.

### Errori di semantica

È correlato al significato del programma. Un errore di semantica non ferma l’esecuzione, anzi, probabilmente terminerà dando un esito sbagliato. Non sono semplici da identificare e si risolvono modificando il codice.

## Differenze di scrittura rispetto ad altri linguaggi

Python non utilizza ; alla fine della riga di codice per terminare l’istruzione. ; viene utilizzato solo se su di una riga ci sono più istruzioni.

Python non usa {} per creare blocchi di codice, bensì utilizza un sistema di indentazione.

|  |  |
| --- | --- |
| Java | Python |
| if (3 < 2){  System.out.println(3);  System.out.println(2);  } | if 3 < 2:  print(3)  print(2) |

# Funzioni

Una funzione è una serie di istruzioni che esegue un calcolo, alla quale viene assegnato un nome.

## Chiamate di funzione

L’invocazione di una funzione è semplice, basta scrivere “nomeFunzione(argomenti)”

Python contiene una raccolta di funzioni per convertire i valori da un tipo a un altro:

* int()
* float()
* str()

## Funzioni matematiche

Python è provvisto di un modulo matematico (math) da importare.

Per farlo basta anteporre l’istruzione di importazione al nome della libreria, “import math”. Per convenzione le librerie sono importate all’inizio del file.

Per accedere ad una funzione del modulo bisogna specificare prima il nome del modulo poi quello della funzione separati da un punto, ad esempio ”math.log10(argomento)”.

All’interno di un modulo possono essere contenute anche variabili, come ad esempio math.pi che contiene l’approssimazione di pi greco a circa 15 cifre.

## Aggiungere nuove funzioni

Una definizione di funzione specifica il nome di una nuova funzione e la serie di istruzioni che viene seguita quando la funzione viene chiamata.

“def” è una parola chiave che indica la definizione di una nuova funzione. Le regole per i nomi delle funzioni sono le stesse che per le variabili. Tra le parentesi vengono inseriti i nomi che assumeranno le variabili passate alla funzione. Di base le variabili vengono passate per valore.

Figura 2 - Esempio di funzione

def somma\_di\_esempio(arg1, arg2):

var = arg1 + arg2

return var

Se la funzione termina con “return” si dice che è “produttiva”, ovvero restituisce qualcosa alla funzione chiamante, altrimenti si dice che è “vuota” (da void, parola usata in altri linguaggi per definire la mancanza di un tipo di ritorno).

## Diagrammi di stack

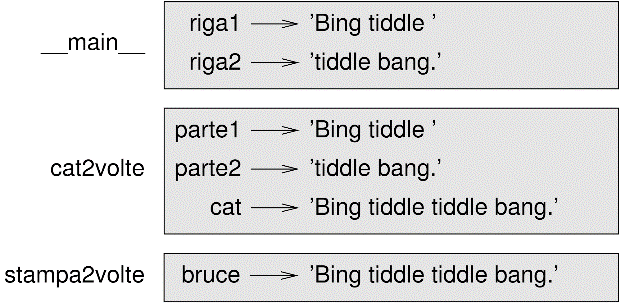
Per tenere traccia di quali variabili possono essere usate e dove, a volte può risultare utile disegnare un diagramma di stack, ovvero un insieme di diagrammi di stato dove viene specificato anche a quale funzione appartengono le variabili.

Figura 3 - Esempio di diagramma di stack

## Perché usare le funzioni?

Ci sono svariati motivi, tra i quali:

* Creare una nuova funzione vi dà modo di dare un nome a un gruppo di istruzioni, rendendo il programma più facile da leggere e da correggere.
* Le funzioni possono rendere un programma più breve, eliminando il codice ripetitivo. Se in un secondo tempo dovete fare una modifica, basterà farla in un posto solo.
* Dividere un programma lungo in funzioni vi permette di correggere le parti una per una, per poi assemblarle in un complesso funzionante.
* Funzioni ben fatte sono spesso utili per più programmi. Quando ne avete scritta e corretta una, la potete riutilizzare tale e quale.

## Funzioni built-in

|  |  |
| --- | --- |
| Metodo | Descrizione |
| abs() | Restituisce il valore assoluto del valore |
| all() | Restituisce True se tutti gli oggetti di un oggetto iterabile sono True |
| any() | Restituisce True se almeno uno degli oggetti di un oggetto iterabile sono True |
| ascii() | Restituisce una versione leggibile di un oggetto. Sostituisce tutti i caratteri non-ascii con caratteri di escape |
| bin() | Restituisce la versione binaria di un numero |
| bool() | Restituisce il valore booleano di un oggetto specifico |
| bytearray(x, encoding, error) | Restituisce un array di byte. Se x è un intero l’array avrà quella dimensione. Se è una stringa bisogna specificare l’encoding. Error dice cosa fare in caso di errore. |
| bytes(x, encoding, error) | Restituisce un oggetto byte immutabile. Se x è un intero l’array avrà quella dimensione. Se è una stringa bisogna specificare l’encoding. Error dice cosa fare in caso di errore. |
| callable() | Restituisce True se l’oggetto è chiamabile, altrimenti False |
| chr() | Restituisce il carattere legato al codice unicode specificato |
| classmethod() | Converte un metodo in un metodo di classe |
| compile() | Ritorna la fonte specificata come un oggetto eseguibile |
| complex() | Ritorna un numero complesso |
| delattr() | Cancella un attributo specifico dall’oggetto specificato |
| dict() | Restituisce un dizionario |
| dir() | Restituisce una lista delle proprietà e dei metodi dell’oggetto specificato |
| divmod() | Restituisce il quoziente e il resto della divisione tra due elementi |
| enumerate() | Prende in ingresso una collezione e restituisce un oggetto enumerabile |
| eval() | Valuta ed esegue un’espressione |
| exec() | Esegue un codice specificato o un oggetto |
| filter() | Una funzione filtro per escludere certi oggetti da un oggetto iterabile |
| float() | Restituisce un float |
| format(valore, codifica) | Restituisce un valore in ogni formato nel modo specificato.  **'<'** – allinea a sinistra  **'>'** – allinea a destra  **'^'** – allinea al centro  **'='** – mette il segno nella posizione più a sinistra  **'+'** – usa un segno per indicare se il valore è positivo o negativo  **'-'** – una un segno solo per i valori negativi  **' '** – usa uno spazio per i numeri positivi  **','** – usa una virgola per separare le migliaia  **'\_'** – usa un underscore per separare le migliaia  **'b'** – formato binario  **'c'** – converte il valore nel corrispondete carattere unicode  **'d'** – formato decimale  **'e'** – formato scientifico, con una e minuscola  **'E'** - formato scientifico, con una E maiuscola  **'f'** – formato in virgola fissa  **'F'** – formato in virgola fissa, maiuscolo  **'g'** – formato generico  **'G'** – formato generico (con E per la notazione scientifica)  **'o'** – formato ottale  **'x'** – formato esadecimale, minuscole  **'X'** - formato esadecimale, maiuscole  **'n'** – formato numerico  **'%'** – formato percentuale |
| frozenset() | Restituisce un set di oggi immutabile |
| getattr() | Restituisce il valore di uno specifico atrributo |
| globals() | Restituisce un dizionario con i valori globali |
| hasattr() | Restituisce True se l’oggetto ha un attributo specificato |
| hash() | Restituisce il valore hash di uno specifico oggetto |
| help() | Esegue il sistema di aiuto del sistema |
| hex() | Converte un numero in un esadecimale |
| id() | Restituisce l’id di un oggetto |
| input() | Permette l’input da parte dell’utente |
| int() | Restituisce un intero |
| isinstance() | Restituisce True se lo specifico oggetto è un’istanza di un oggetto specificato |
| issubclass() | Restituisce True se lo specifico oggetto è una sottoclasse di un oggetto specificato |
| iter() | Restituisce True se l’oggetto è iterabile |
| len() | Restituisce la lunghezza di un oggetto |
| list() | Restituisce una lista |
| locals() | Restituisce un dizionario con la tabella delle variabili locali |
| map() | Restituisce l’oggetto iterabile con una funzione applicata ad ogni suo elemento |
| max() | Restituisce il massimo in un oggetto iterabile |
| memoryview() | Restituisce una lista di oggetti accessibili con il loro valore in memoria |
| min() | Restituisce il minimo in un oggetto iterabile |
| next() | Restituisce l’oggetto successivo in un oggetto iterabile |
| object() | Restituisce un nuovo oggetto |
| oct() | Converte un numero in un ottale |
| open() | Apre un file e restituisce un oggetto file |
| ord() | Converte un intero che rappresenta l’unicode di un carattere specificato |
| pow() | Restituisce la potenza di x alla y |
| print() | Stampa sullo standard output |
| property() | Ottiene, imposta, cancella una proprietà |
| range(x, y, z) | Restituisce una sequenza di numeri, da x a y con incrementi z (1 di default) |
| repr() | Restituisce una versione leggibile di un oggetto |
| reversed() | Inverte un oggetto iterabile |
| round() | Arrotonda un numero |
| set() | Restituisce un nuovo set |
| setattr() | Imposta un attributo di un oggetto |
| slice(inizio, fine) | Restituisce un oggetto slice, utilizzabile per segnalare un range |
| sorted() | Restituisce una lista ordinata |
| @staticmethod() | Converte un metodo in un metodo statico |
| str() | Restituisce una stringa |
| sum() | Somma tutti gli oggetti di un oggetto iterabile |
| type() | Restituisce il tipo di un oggetto |
| vars() | Restituisce la proprietà \_\_dict\_\_ di un oggetto |
| zip() | Restituisce un oggetto iterabile composto da tuple generate da due o più oggetti iterabili |

# Progettazione dell’interfaccia (turtle)

Viene proposta un’esercitazione che dimostra una procedura per progettare delle funzioni che collaborano tra loro.

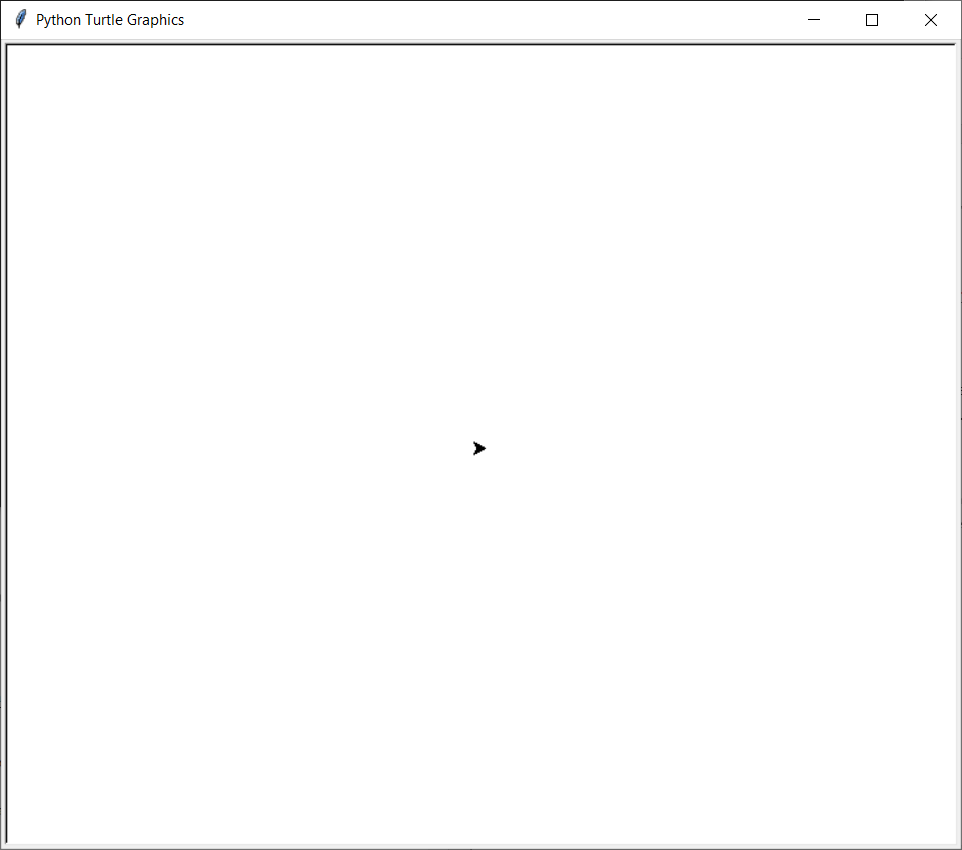
Viene illustrato il modulo grafico turtle che permette di creare immagini utilizzando turtle graphics, modulo già compreso nella maggior parte delle installazioni di Python.

## Il modulo turtle

Per controllare se turtle è installato aprire l’interfaccia Python e scrivere l’istruzione riportata di seguito. Eseguendo questo codice dovrebbe comparire una nuova finestra con un cursore a forma di freccetta.

import turtle

bob = turtle.Turtle()



Craere un file di nome miopoligono.py e scriveteci il seguente codice:

import turtle

bob = turtle.Turtle()

turtle.mainloop()

bob.fd(100)

bob.lt(90)

bob.fd(100)

ora cerchiamo di capire cosa faccia questo codice:

Figura 4 - Controllo presenza turtle

* import turtle: importa il modulo turtle
* bob = turtle.Turtle(): la funzione Turtle crea un oggetto Turtle e viene assegnato ad una variabile di nome bob
* turtle.mainloop(): mainloop dice alla finestra di attendere che l’utente faccia qualcosa
* bob.fd(100): il metodo fd (“forward”), complementare al metodo bk (“backward”), è associato all’oggetto che abbiamo chiamato bob. Viene spostato di 100 pixel nella direzione indicata dalla freccia
* bob.lt(90): il metodo lt (“left turn”), complementare al metodo rt (“right turn”), fa girare di 90 gradi a sinistra la freccia
* bob.fd(100): come prima sposta di 100 pixel la freccia

Inoltre, ogni tartaruga regge una penna, che può essere appoggiata o sollevata; se la penna è appoggiata, la tartaruga lascia un segno dove passa. I metodi pu e pd stanno per “penna su (up)” e “penna giù (down)”.

## Incapsulamento

L’inglobare un pezzo di codice in una funzione è chiamato incapsulamento. Uno dei benefici dell’incapsulamento è che appiccica un nome al codice, il che può servire come una sorta di documentazione. Un altro vantaggio è il riuso del codice: è più conciso chiamare una funzione due volte che copiare e incollare il corpo!

## Generalizzazione

L’aggiunta di un parametro a una funzione è chiamata generalizzazione poiché rende la funzione più generale.

Quando in una chiamata di funzione avete più di qualche argomento numerico, è facile dimenticare a cosa si riferiscono o in quale ordine vanno disposti. In questi casi, è bene includere i nomi dei parametri nell’elenco degli argomenti: “poligono(bob, n=7, lunghezza=70)”. Questi sono detti argomenti con nome perché includono il nome del parametro a cui vengono passati, quale “parola chiave” (da non confondere con le parole chiave riservate come while e def).

## Progettare un’interfaccia

L’interfaccia è un riassunto di come è usata la funzione: quali sono i pratametri? Che cosa fa la funzione? Qual è il valore restituito? Un’interfaccia è considerata “pulita” se permette al chiamante di fare ciò che deve, senza avere a che fare con dettagli non necessari.

## Tecnica di sviluppo

Una tecnica di sviluppo è una procedura di scrittura dei programmi. Quello che abbiamo usato in questa esercitazione si chiama “incapsulamento e generalizzazione”. I passi della procedura sono:

1. Iniziare scrivendo un piccolo programma senza definire funzioni.
2. Una volta ottenuto un programma funzionante, identificare una sua porzione che sia in sé coerente e autonoma, incapsularla in una funzione e dargli un nome.
3. Generalizzare la funzione aggiungendo i parametri appropriati.
4. Ripetere i passi da 1 a 3 fino ad avere un insieme di funzioni. Copiate e incollate il codice funzionante per evitare di riscriverlo (e ricorreggerlo).
5. Cercare le occasioni per migliorare il programma con il refactoring. Ad esempio, se avete del codice simile in più punti, valutate di semplificare rielaborandolo in una funzione più generale.

Questa procedura ha alcuni inconvenienti—vedremo più avanti alcune alternative—ma può essere di aiuto se in principio non sapete bene come suddividere il vostro programma in funzioni. È un approccio che vi permette di progettare man mano che andate avanti.

## Stringa di documentazione

Una stringa di documentazione, o docstring, è una stringa posta all’inizio di una funzione che ne illustra l’interfaccia.

def polilinea(t, n, lunghezza, angolo):

**""" Disegna n segmenti di data lunghezza e**

**Angolo (in gradi) tra di loro. t e' una**

**tartaruga.**

**"""**

for i in range(n):

t.fd(lunghezza)

t.lt(angolo)

È breve, ma contiene le informazioni essenziali di cui qualcuno potrebbe aver bisogno per usare la funzione. Spiega in modo conciso cosa fa la funzione (senza entrare nei dettagli di

come lo fa). Spiega che effetti ha ciascun parametro sul comportamento della funzione e di che tipo devono essere i parametri stessi (se non è ovvio).

# Istruzioni condizionali e ricorsione

Operatori booleani:

* And
* Or
* <, <=, >, >=, ==, !=

## if

l’istruzione if per Python è molto semplice

if x < y:

print('x è minore di y')

elif x > y:

print('x è maggiore di y')

else:

print('x e y sono uguali')

## Input da tastiera

Python 3 ha una semplice funzione per l’inserimento dell’input da parte dell’utente: input(). Input accetta anche un argomento che sarà il messaggio mostrato dal prompt prima della richiesta di inserimento.

nome = input('Come...ti chiami?\n')

Come...ti chiami?

Artù, Re dei Bretoni!

>>> nome

'Artù, Re dei Bretoni!'

# Iterazione

## while

l’istruzione while è semplice.

while condizione:

corpo

## break

Vi può capitare di poter stabilire il momento in cui è necessario terminare un ciclo solo mentre il flusso di esecuzione si trova nel bel mezzo del corpo. In questi casi potete usare l’istruzione break per interrompere il ciclo e saltarne fuori.

## for

il for viene utilizzato quando si ha un blocco che si vuole ripetere un numero determinato di volte. Per usare loop dobbiamo conoscere anche la funzione range.

Figura 5 - For fisso e for su oggetto

for x in range(0, 3):

print "We're on time %d" % (x)

string = "Hello World"

for x in string:

print x

Range(inizio, fine, [passo]), dove passo è opzionale e indica di quanto deve aumentare ad ogni step. Ad esempio range(1, 10, 2) sono tutti i numeri dispari tra 1 e 10.

Quindi ci sono due modi per utilizzare for:

* iterazione per un numero fissato di volte: si usa un range per determinare il numero di volte
* iterazione su di una lista, un dizionario, una stringa o simili: si usa segnando una variabile temporanea che ogni volta assume come valore l’elemento selezionato durante il ciclo

## Pass

Pass viene usato quando il programma non deve fare niente. Generalmente viene usato quando serve un placeholder prima di implementare una funzione e si vuole sviluppare successivamente.

def initlog(\*args):

pass # Remember to implement

this!

# Stringhe

Una stringa è una sequenza di caratteri. Potete accedere ai singoli caratteri usando gli operatori parentesi quadre. Gli indici tra parentesi quadre possono essere positivi (dal primo all’ultimo carattere) o negativi (dall’ultimo al primo carattere).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Indici positivi | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | S | T | R | I | N | G | A |
| Indici negativi | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

Per avere la lunghezza della stringa si può usare la funzione len().

Le stringhe sono immutabili.

## Attraversamento

Una stringa può essere attraversata mediante l’utilizzo di un ciclo for o di un ciclo while.

## Slicing

Un segmento o una porzione di stringa è detto slice. Per isolare uno slice vengono usate le parentesi quadre nella forma “stringa[inizio(incluso):fine(esclusa)]”. I due indici possono essere esclusi, se viene escluso l’inizio si inizierà dal primo carattere, se viene omessa la fine si arriva in fondo alla stringa.

## in

La parola in è un operatore booleano che confronta due stringhe e restituisce True se la prima è una sottostringa della seconda.

'a' in 'banana'

True

## Confronto di stringhe

Gli operatori di confronto funzionano anche sulle stringhe.

* Stringa1 == stringa2: controlla se le stringhe sono uguali
* Stringa1 < stringa2: controlla se la stringa1 viene prima della seconda
* Stringa1 < stringa2: controlla se la stringa1 viene dopo la seconda

Attenzione che Python non gestisce le lettere maiuscole e minuscole come siamo abituati: in un confronto, le lettere maiuscole vengono sempre prima di tutte le minuscole. Questo problema si risolve facilmente convertendo le stringhe in un formato standard, ad esempio tutte lettere minuscole, prima di effettuare il confronto.

## Funzioni sulle stringhe

Nota: tutti i metodi sulle stringhe ritornano un valore. Non cambian)o la stringa originale.

|  |  |
| --- | --- |
| Metodo | Descrizione |
| stringa.capitalize() | Converte il primo carattere in maiuscola |
| stringa.casefold() | Converte la stringa in minuscole |
| stringa.center() | Ritorna una stirnga centrata |
| stringa.count() | Ritorna il numero di volte che un valore specifico è presente in una stringa |
| stringa.encode() | Ritorna il valore codificato della stringa |
| stringa.endswith() | Ritorna True se la stringa termina con un valore specificato |
| stringa.expandtabs() | Imposta la dimensione del rientro della stringa |
| stringa.find(valore, inizio, fine) | Cerca un valore specifico nella stringa e ritorna la prima occorrenza |
| stringa.format() | Sostituisce il valore tra parentesi al posto dei placeholder posti all’interno della stringa, segnati con {} |
| stringa.format\_map() | Sostituisce i valori appartenenti ad un dizionario tra parentesi al posto dei placeholder posti all’interno della stringa, segnati con {} |
| stringa.index() | Cerca un valore specifico nella stringa e ritorna la prima occorrenza |
| stringa.isalnum() | Ritorna True se tutti i caratteri della stringa sono alfanumerici |
| stringa.isalpha() | Ritorna True se tutti i caratteri della stringa sono nell’alfabeto |
| stringa.isdecimal() | Ritorna True se tutti i caratteri della stringa sono decimali (0-9), questo metodo è usato con gli oggetti in unicode |
| stringa.isdigit() | Ritorna True se tutti i caratteri della stringa sono numeri |
| stringa.isidentifier() | Ritorna True se la stringa è un identificatore |
| stringa.islower() | Ritorna True se tutti i caratteri della stringa sono minuscole |
| stringa.isnumeric() | Ritorna True se tutti i caratteri della stringa sono numerici (0-9) |
| stringa.isprintable() | Ritorna True se tutti i caratteri della stringa sono stampabili. Esempi di caratteri non stampabili sono /n, /r |
| stringa.isspace() | Ritorna True se tutti i caratteri della stringa sono spazi |
| stringa.istititle() | Ritorna True se tutti i caratteri della stringa seguono le regole per un titolo (ogni parola inizia con una maiuscola e il resto delle lettere sono minuscole) |
| stringa.isupper() | Ritorna True se tutti i caratteri della stringa sono maiuscole |
| stringa.join() | Unisce gli elementi di un oggetto iterabile alla fine della stringa |
| stringa.ljust() | Restituisce una versione giustificata a sinistra della stringa |
| stringa.lower() | Converte la stringa in minuscole |
| stringa.lstrip() | Restituisce una versione tagliata a sinistra della stringa |
| stringa.maketrans() | Restituisce una tabella di traduzione per usarla nelle traduzioni |
| stringa.partition() | Resituisce delle tuple dove la stringa è divisa in tre parti |
| stringa.replace() | Ritorna una stringa dove un valore è sostituito con un altro |
| stringa.rfind() | Cerca un valore nella stringa e restituisce l’ultima occorrenza |
| stringa.rindex( | Cerca un valore nella stringa e restituisce l’ultima occorrenza |
| stringa.rjust() | Ritorna una versione giustificata a destra della stringa |
| stringa.rpartition() | Ritorna una tupla dove la stringa è divisa in tre parti |
| stringa.rsplit() | Divide la stringa con un separatore specificato e ritorna una lista. Lo spazio è il separatore di default |
| stringa.splitlines() | Divide la stringa ad ogni carattere di fine linea e ritorna una stringa |
| stringa.startswith() | Ritorna True se la stringa inizia con un valore specifico |
| stringa.strip() | Ritorna una versione tagliata della stringa |
| stringa.swapcase() | Rende le minuscole maiuscole e viceversa |
| stringa.title() | Converte il primo carattere di ogni parola in una maiuscola |
| stringa.translate() | Ritorna una versione tradotta della stringa utilizzando un dizionario |
| stringa.upper() | Converte la stringa in maiuscole |
| stringa.zfill() | Riempie la stringa con un numero specifico di 0 all’inizio |

# Liste

## Una lista è una sequenza

Come una stringa, una lista è una sequenza di valori. Mentre in una stringa i valori sono dei caratteri, in una lista possono essere di qualsiasi tipo. I valori che fanno parte della lista sono chiamati elementi. Ci sono vari modi per creare una nuova lista; quello più semplice consiste nel racchiudere i suoi elementi tra parentesi quadrate ([ e ]).

['spam', 2.0, 5, [10, 20]]

## Le liste sono mutabili

La sintassi per accedere agli elementi di una lista è la stessa usata per i caratteri di una stringa: le parentesi quadre, con un’espressione tra parentesi che specifica l’indice dell’elemento.

>>> numeri = [42, 123]

>>> numeri[1] = 5

>>> numeri

[42, 5]

Ma a differenza delle stringhe, le liste sono mutabili. Quando l’operatore parentesi quadre compare sul lato sinistro di un’assegnazione, identifica l’elemento della lista che sarà riassegnato.

## Operazioni sulle liste

* +: concatenazione di liste
* \*: ripete una lista un numero di volte

## Slicing delle liste

Funziona come con le stringhe.

## Lista e stringhe

Per convertire una stringa in una lista di caratteri, si può usare list(stringa).

>>> s = 'spam'

>>> t = list(s)

>>> t

['s', 'p', 'a', 'm']

La funzione list separa la stringa in singole lettere. Se invece si vuole spezzare la stringa in parole si può usare stringa.split().

Un argomento opzionale, chiamato delimitatore, specifica quale carattere va considerato come separatore delle parole. Di default viene preso lo spazio.

>>> s = 'profonda-nostalgia-dei-fiordi'

>>> t = s.split(‘-‘)

>>> t

['profonda', 'nostalgia', 'dei', 'fiordi']

>>> t = ['profonda', 'nostalgia', 'dei', 'fiordi']

>>> delimita = ' '

>>> s = delimita.join(t)

'profonda nostalgia dei fiordi'

join è l’inverso di split: prende una lista di stringhe e concatena gli elementi. join è un metodo delle stringhe, quindi si deve invocare per mezzo del delimitatore e passare la lista come parametro.

## Oggetti e valori

Se eseguiamo queste istruzioni di assegnazione:

a = 'banana'

b = 'banana'

Sappiamo che a e b si riferiscono a una stringa, ma non sappiamo se si riferiscono alla stessa stringa. Ci sono due possibili stati, in un caso, a e b si riferiscono a due oggetti diversi che hanno lo stesso valore. Nell’altro, si riferiscono allo stesso oggetto. Per controllare se due variabili si riferiscono allo stesso oggetto, potete usare l’operatore is.

## Liste come argomenti

Quando si passa una lista a una funzione, questa riceve un riferimento alla lista. Se la funzione modifica la lista, il chiamante vede la modifica.

È importante distinguere tra operazioni che modificano le liste e operazioni che creano nuove liste. Per esempio, il metodo append modifica una lista, ma l’operatore + ne crea una nuova.

## Funzioni sulle liste

|  |  |
| --- | --- |
| Metodo | Descrizione |
| lista.append() | Aggiunge un elemento alla fine della lista |
| lista.clear() | Rimuove tutti gli elementi dalla lista |
| lista.copy() | Ritorna una copia della lista |
| lista.count() | Ritorna il numero di elementi con un valore specificato |
| lista.extend() | Aggiunge gli elementi di una lista (o di un oggetto iterabile) alla fine della lista |
| lista.insert() | Aggiunge un elemento nella posizione specificata |
| lista.pop(indice) | Rimuove l’elemento alla posizione inidicata e lo restituisce. Se non viene specificata la posizione è l’ultimo |
| lista.remove() | Rimuove il primo oggetto con il valore specificato |
| lista.reverse() | Inverte l’ordine della lista |
| lista.sort() | Ordina la lista |
| del lista[indice] | Elimina il valore all’indice specificato. Accetta lo slicing |
| sum(lista) | Restituisce la somma dei valori della lista |

# Dizionari

>>> eng2it = {'one': 'uno', 'two': 'due'}

>>>eng2it[‘three’] = ‘tre’

>>> eng2it

{'one': 'uno', 'three': 'tre', 'two': 'due'}

>>> eng2it['two']

'due'

>>> vals = eng2it.values()

>>> 'uno' in vals

True

Un dizionario è simile ad una lista, ma è più generico. Infatti, mentre in una lista gli indici devono essere numeri interi, in un dizionario possono essere (quasi) di ogni tipo.

Un dizionario contiene una raccolta di indici, chiamati chiavi, e una raccolta di valori. Ciascuna chiave è associata ad un unico valore. L’associazione tra una chiave e un valore è detta coppia chiave-valore o anche elemento.

La funzione dict() crea un nuovo dizionario privo di elementi.

Le parentesi graffe, {}, rappresentano un dizionario vuoto. Per aggiungere elementi al dizionario, si usano le parentesi quadre.

Si può usare la notazione “chiave in dizionario” per sapere se la chiave esista.

Per controllare invece se qualcosa compare come valore, si può usare il metodo values, che restituisce una raccolta dei valori, e quindi usare l’operatore in.

L’operatore in utilizza algoritmi diversi per liste e dizionari:

* Per le prime, ne ricerca gli elementi in base all’ordine. Se la lista si allunga, anche il tempo di ricerca si allunga in proporzione.
* Per i secondi, Python usa un algoritmo chiamato tabella hash che ha notevoli proprietà: l’operatore in impiega sempre circa lo stesso tempo, indipendentemente da quanti elementi contiene il dizionario

## Lookup inverso

Dato un dizionario d e una chiave k, è facile trovare il valore corrispondente alla chiave: v = d[k]. Questa operazione è chiamata lookup.

def inverso\_lookup(d, v):

for k in d:

if d[k] == v:

return k

raise LookupError(‘valore non presente’)

Ma se invece volete trovare la chiave k conoscendo il valore v? Ci sono due problemi:

* primo, ci possono essere più chiavi che corrispondono al valore v. A seconda dell’applicazione, si può riuscire a trovarne uno, oppure può essere necessario ricavare una lista che li contenga tutti.
* Secondo, non c’è una sintassi semplice per fare un lookup inverso; bisogna impostare una ricerca.

Questa funzione è un altro esempio di schema di ricerca, ma usa una nuova istruzione, raise. L’istruzione raise solleva un’eccezione; in questo caso genera un errore LookupError, che è un’eccezione predefinita usata per indicare che un’operazione di lookup è fallita. Se si arriva a fine ciclo, significa che v non compare nel dizionario come valore, per cui solleviamo un’eccezione. L’istruzione raise può ricevere come parametro opzionale un messaggio di errore dettagliato.

Un lookup inverso è molto più lento di un lookup; se dovete farlo spesso, o se il dizionario diventa molto grande, le prestazioni del programma potrebbero risentirne.

## Memoizzazione

Una soluzione per rendere più veloci delle funzioni con chiamate esponenziali è tenere da parte i valori che sono già stati calcolati, conservandoli in un dizionario. La tecnica di conservare per un uso successivo un valore già calcolato, così da non lo si deve ricalcolare ogni volta, viene detta memoizzazione.

Un esempio è la sequenza di fibonacci.

|  |  |
| --- | --- |
| Senza memoizzazione | Con memoizzazione |
| def fibonacci(n):  if n == 0:  return 0  elif n == 1:  return 1  else:  return fibonacci(n-1) + fibonacci(n-2) | memo = {0:0, 1:1}  def fibonacci(n):  if n in memo:  return memo[n]  res = fibonacci(n-1) + fibonacci(n-2)  memo[n] = res  return res |

Ogni volta che fibonacci viene chiamata, controlla innanzitutto memo. Se quest’ultimo contiene già il risultato, ritorna immediatamente. Altrimenti deve calcolare il nuovo valore, lo aggiunge al dizionario e lo restituisce.

## Variabili globali

Nell’esempio precedente, memo viene creato esternamente alla funzione, pertanto appartiene al frame speciale chiamato \_\_main\_\_. Le variabili di \_\_main\_\_ sono dette anche globali perché ad esse possono accedere tutte le funzioni. A differenza delle variabili locali, che sono distrutte una volta terminata l’esecuzione della loro funzione, quelle globali persistono tra una chiamata di funzione e l’altra.

Di frequente le variabili globali vengono usate come controlli o flag; vale a dire, variabili booleane che indicano quando una certa condizione è soddisfatta (True).

Per riassegnare una variabile globale dall’interno di una funzione, dovete dichiarare la variabile globale prima di usarla.

|  |  |
| --- | --- |
| Sbagliato | Corretto |
| stata\_chiamata = False  def esempio():  stata\_chiamata = True | stata\_chiamata = False  def esempio():  global stata\_chiamata  stata\_chiamata = True |

## Metodi sui dizionari

|  |  |
| --- | --- |
| Metodo | Descrizione |
| dizionario.clear() | Rimuove tutti gli elementi |
| dizionario.copy() | Crea una copia del dizionario |
| dizionario.fromkeys() | Ritorna un dizionario con le chiavi specificate |
| dizionario.get() | Ritorna il valore legato alla chiave specificata |
| dizionario.items() | Ritorna una lista contenente le tuple rappresentanti tutte le coppie chiave-valore |
| dizionario.keys() | Ritorna una lista con tutte le chiavi |
| dizionario.pop() | Rimuove l’elemento con la chiave specificata |
| dizionario.popitem() | Rimuove l’ultima coppia chiave-valore inserita |
| dizionario.setdefault() | Ritorna il valore legato alla chiave specificata. Se la chiave non esiste: inserisce la chiave con il valore specificato, None se non specificato |
| dizionario.update() | Aggiorna il dizionario con le coppie chiave-valore specificate |
| dizionario.values() | Ritorna la lista con tutti i valori del dizionario |

# Tuple

## Le tuple sono immutabili

Una tupla è una sequenza di valori. I valori possono essere di qualsiasi tipo, sono indicizzati tramite numeri interi, e in questo somigliano moltissimo alle liste. La differenza fondamentale è che le tuple sono immutabili.

Sintatticamente, la tupla è un elenco di valori separati da virgole:

>>> t = 'a', 'b', 'c', 'd', 'e'

Sebbene non sia necessario, è convenzione racchiudere le tuple tra parentesi tonde:

>>> t = ('a', 'b', 'c', 'd', 'e')

Per creare una tupla con un singolo elemento, occorre aggiungere una virgola dopo l’elemento.

Un altro modo di creare una tupla è usare la funzione predefinita tuple().

Se l’argomento è una sequenza (stringa, lista o tupla), il risultato è una tupla con gli elementi della sequenza.

La maggior parte degli operatori delle liste funzionano anche con le tuple. L’operatore parentesi quadre indicizza un elemento della tupla. Anche lo slicing funziona.

## Assegnazione di tupla

Durante lo scambio di due valori possiamo non usare una variabile temporanea e sostituire questa con un’assegnazione di tupla.

>>> a, b = b, a

>>> indirizzo = 'monty@python.org'

>>> nome, dominio = indirizzo.split('@')

>>> nome

'monty'

>>> dominio

'python.org'

Sul lato sinistro abbiamo una tupla di variabili; su quello destro, una tupla di espressioni. Ciascun valore viene assegnato alla rispettiva variabile. Tutte le espressioni sul lato destro vengono valutate prima di ogni assegnazione.

## Tuple come valori di ritorno

In senso stretto, una funzione può restituire un solo valore di ritorno, ma se il valore è una tupla, l’effetto pratico è quello di restituire valori molteplici.

## Tuple di argomenti a lunghezza variabile

Le funzioni possono ricevere un numero variabile di argomenti. Un nome di parametro che comincia con \*, raccoglie gli argomenti in una tupla. Il parametro di raccolta può avere qualunque nome, ma per convenzione si usa args.

Il contrario della raccolta è lo spacchettamento. Se si ha una sequenza di valori e la si vuole passare a una funzione come argomenti multipli, si può usare ancora l’operatore \*.

def stampatutti(\*args):

print(args)

>>> stampatutti(1, 2.0, '3')

(1, 2.0, '3')

>>> t = (7, 3)

>>> divmod(\*t)

(2, 1)

## Liste e tuple

zip è una funzione predefinita che riceve due o più sequenze e restituisce una lista di tuple, dove ciascuna tupla contiene un elemento di ciascuna sequenza. Il risultato è un oggetto zip capace di iterare attraverso le coppie. L’uso più frequente di zip è in un ciclo for.

>>> s = 'abc'

>>> t = [0, 1, 2]

>>> zip(s, t)

>>> for coppia in zip(s, t):

... print(coppia)

('a', 0)

('b', 1)

('c', 2)

>>> list(zip(s, t))

[('a', 0), ('b', 1), ('c', 2)]

def corrispondenza(t1, t2):

for x, y in zip(t1, t2):

if x == y:

return True

return False

Un oggetto zip è un tipo di iteratore, che è un qualsiasi oggetto in grado di iterare attraverso una sequenza.

Se si desidera usare operatori e metodi delle liste, si può crearne una utilizzando un oggetto zip.

Se le sequenze non sono della stessa lunghezza, il risultato ha la lunghezza di quella più corta.

Combinando zip, for e assegnazione di tupla, si ottiene un utile costrutto per attraversare due o più sequenze contemporaneamente.

Se si vuole attraversare gli elementi di una sequenza e i loro indici, si può usare la funzione predefinita enumerate.

Il risultato di enumerate è un oggetto enumerate, che itera una sequenza di coppie; ogni coppia contiene un indice (a partire da 0) e un elemento della sequenza data.

for indice, elemento in enumerate('abc'):

print(indice, elemento)

0 a

1 b

2 c

## Dizionari e tuple

I dizionari supportano un metodo di nome items che restituisce una sequenza di tuple, dove ogni tupla è una delle coppie chiave-valore.

>>> d = {'a':0, 'b':1, 'c':2}

>>> t = d.items()

>>> t

dict\_items([('c', 2), ('a', 0), ('b', 1)])

>>> d = dict(zip('abc', range(3)))

>>> d

{'a': 0, 'c': 2, 'b': 1}

Il risultato è un oggetto dict\_items, un iteratore che itera le coppie chiave-valore.

La combinazione di dict e zip produce un modo conciso di creare un dizionario.

Anche il metodo dei dizionari update prende una lista di tuple e le aggiunge, come coppie chiave-valore, a un dizionario esistente.

## Sequenze di sequenze

|  |  |  |
| --- | --- | --- |
|  | Mutabilità | Elementi |
| Stringhe | Immutabili | Caratteri |
| Liste | Mutabili | Qualsiasi |
| Dizionari | Mutabili | Qualsiasi in coppie  chiave : elemento |
| Tuple | Immutabili | qualsiasi |

Ci siamo concentrati finora sulle liste di tuple, ma quasi tutti gli esempi di questo capitolo funzionano anche con liste di liste, tuple di tuple, e tuple di liste. Per evitare di elencare tutte le possibili combinazioni, è più semplice usare il termine sequenze di sequenze.

## Metodi sulle tuple

|  |  |
| --- | --- |
| Metodo | Descrizione |
| tupla.count() | Ritorna il numero di volte che un elemento compare nella tupla |
| tupla.index() | Restituisce l’indice di uno specifico valore |

# File

## Persistenza

Per i programmi, uno dei modi più semplici di mantenere i loro dati è di leggerli e scriverli su file di testo. Abbiamo già visto qualche programma che legge dei file di testo; in questo capitolo ne vedremo alcuni che li scrivono.

Un’alternativa è conservare la situazione del programma in un database. In questo capitolo vedremo un semplice database e un modulo, pickle, che rende agevole l’archiviazione dei dati.

## Lettura e scrittura

Per scrivere un file, lo dovete aprire indicando la modalità 'w' come secondo parametro, Se il file esiste già, l’apertura in modalità scrittura lo ripulisce dai vecchi dati e riparte da zero, altrimenti ne viene creato uno nuovo.

>>> fout = open('output.txt', 'w')

>>> riga1 = "E questa qui è l'acacia,\n"

>>> fout.write(riga1)

>>> fout.close()

File.open(nome, modalità) restituisce un oggetto file che fornisce i metodi per lavorare con il file.

Il metodo file.write(stringa) inserisce i dati nel file.

Quando avete finito di scrivere, è opportuno chiudere il file con la funzione file.close(). Se non chiudete il file, viene comunque chiuso automaticamente al termine del programma.

## L’operatore di formato

L’argomento di write deve essere una stringa, e se volessimo inserire valori di tipo diverso in un file dovremmo prima convertirli in stringhe. Il metodo più semplice per farlo è usare str().

>>> cammelli = 42

>>> '%d' % cammelli

'42'

Un’alternativa è utilizzare l’operatore di formato, %. Quando viene applicato agli interi, % rappresenta l’operatore modulo. Ma se il primo operando è una stringa, % diventa l’operatore di formato.

Il primo operando è detto stringa di formato, che contiene una o più sequenze di formato, che specificano il formato del secondo operando. Il risultato è una stringa.

|  |  |
| --- | --- |
| Conversione | Significato |
| %d | Decimale |
| %i | Decimale |
| %o | Ottale |
| %u | Identico a d, ormai obsoleto |
| %x | Esadecimale (minuscole) |
| %X | Esadecimale (maiuscole) |
| %f | Virgola mobile |
| %F | Virgola mobile |
| %g | Notazione esponenziale (e minuscola) se l’esponente è minore di -4, altrimenti formato decimale |
| %G | Notazione esponenziale (e maiuscola) se l’esponente è minore di -4, altrimenti formato decimale |
| %c | Singolo carattere (accetta un intero o una stringa di un unico carattere) |
| %r | Stringa (converte qualsiasi oggetto tramite il metodo repr()) |
| %s | Stringa (converte qualsiasi oggetto tramite il metodo str()) |
| %a | Stringa (converte qualsiasi oggetto tramite il metodo ascii()) |
|  | Nessun argomento è convertito, in un carattere |

## Nomi di file e percorsi

Il modulo os fornisce delle funzioni per lavorare con file e directory (“os” sta per “sistema operativo”). os.getcwd restituisce il nome della directory corrente.

>>> import os

>>> cwd = os.getcwd()

>>> cwd

'/home/dinsdale'

>>> os.path.abspath('memo.txt')

'/home/dinsdale/memo.txt'

I percorsi visti finora sono semplici nomi di file, quindi sono percorsi relativi alla directory corrente. Per avere invece il percorso assoluto, potete usare os.path.abspath.

os.path fornisce altre funzioni per lavorare con nomi di file e percorsi. Per esempio, os.path.exists controlla se un file o una cartella esistono.

Se esiste, os.path.isdir controlla se è una directory, Similmente, os.path.isfile controlla se è un file.

os.path.join prende il nome di una directory e il nome di un file e li unisce a formare un percorso completo.

## Gestire le eccezioni

Parecchie cose possono andare storte quando si cerca di leggere e scrivere file. Se si prova ad aprire un file che non esiste, si verifica un IOError. Se non si ha permesso di accedere ad un file si ha un PermissionError e infine cercando di aprire una directory in lettura si ha un IsADirectoryError.

try:

fin = open('file\_corrotto')

except:

print('Qualcosa non funziona.')

Per evitare questi errori, si possono usare funzioni come os.path.exists e os.path.isfile, ma ci vorrebbe molto tempo e molto codice per controllare tutte le possibilità. È meglio allora andare avanti e provare—e affrontare i problemi quando si presentano, che è proprio quello che fa l’istruzione try. La sintassi è simile a un’istruzione if...else, ovvero è if…except.

## Database

Un database è un file che è progettato per archiviare dati. Molti database sono organizzati come un dizionario, nel senso che fanno una mappatura da chiavi in valori.

Il modulo dbm fornisce un’interfaccia per creare e aggiornare file di database.

>>> import dbm

>>> db = dbm.open('didascalie', 'c')

>>> db['cleese.png'] = 'Foto di John Cleese.'

>>> db['cleese.png']

b'Foto di John Cleese.'

>>> db['cleese.png'] = 'Foto di John Cleese che cammina in modo ridicolo.'

>>> db['cleese.png']

b'Foto di John Cleese che cammina in modo ridicolo.'

Un database si apre in modo simile agli altri file tramite la funzione dbm.open(nome, modalità). La modalità 'c' significa che il database deve essere creato se non esiste già. Il risultato è un oggetto database che può essere utilizzato (per la maggior parte delle operazioni) come un dizionario.

Se si crea un nuovo elemento, dbm aggiorna il file di database.

Quando si accede a uno degli elementi, dbm legge il file: Il risultato è un oggetto bytes, ed è per questo che comincia per b. Un oggetto bytes è per molti aspetti simile ad una stringa[[2]](#footnote-2).

Se si fa una nuova assegnazione a una chiave esistente, dbm sostituisce il vecchio valore.

Certi metodi dei dizionari, come keys e items, non funzionano con gli oggetti database, ma funziona l’iterazione con un ciclo for.

Come con gli altri file, si deve chiudere il database tramite database.close().

## Pickling

Un limite di dbm è che le chiavi e i valori deovno essere delle stinghe, oppure bytes. Se si prova ad utilizzare qualsiasi altro tipo, si verifica un errore.

Il modulo pickle può essere di aiuto: trasforma quasi ogni tipo di oggetto in una stringa, adatta per essere inserita in un database, e quindi ritrasforma la stringa in oggetto.

pickle.dumps accetta un oggetto come parametro e ne restituisce una serializzazione, ovvero una rappresentazione sotto forma di una stringa.

>>> import pickle

>>> t = [1, 2, 3]

>>> pickle.dumps(t)

b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> t1 = [1, 2, 3]

>>> s = pickle.dumps(t1)

>>> t2 = pickle.loads(s)

>>> t2

[1, 2, 3]

Il formato non è immediatamente leggibile: è progettato per essere facile da interpretare da parte di pickle. In seguito, pickle.loads ricostruisce l’oggetto.

Sebbene il nuovo oggetto abbia lo stesso valore di quello vecchio, non è in genere lo stesso oggetto. In altre parole, fare una serializzazione con pickle e poi l’operazione inversa, ha lo stesso effetto di copiare l’oggetto.

Si può usare pickle per archiviare in un database tutto ciò che non è una stringa. In effetti, questa combinazione è tanto frequente da essere stata incapsulata in un modulo chiamato shelve.

La differenza tra un oggetto shelf (derivante da shelve) e un oggetto dbm è che i valori (non le chiavi) della shelf possono essere oggetti di qualsiasi tipo, qualunque cosa che pickle possa gestire.

Gli oggetti shelf supportano tutti i metodi dei dizionari e due metodi ulteriori:

* Shelf.sync(): scrive tutti i valori nella cache se la shlef è stata aperta con “writeback = True”. Svuota anche la cache e sincronizza il dizionario, se possibile. Questo metodo è invocato automaticamente da close()
* Shelf.close(): sincronizza e chiude l’oggetto dict.

## Pipe

Qualsiasi programma lanciabile dalla shell può essere lanciato anche da Python usando un oggetto pipe, che rappresenta un programma in esecuzione.

Ad esempio, il comando Unix ls -l di norma mostra il contenuto della cartella attuale (in formato esteso). Potete lanciare ls anche con os.popen() che accetta in ingresso una stringa contenente il comando. Il valore di ritorno è un oggetto che si comporta come un file aperto. Si può leggere l’output del processo ls una riga per volta con readline, oppure ottenere tutto in una volta con read.

## Scrivere moduli

Qualunque file che contenga codice Python può essere importato come modulo. Semplicemente si deve importare come qualsiasi libreria. Supponiamo di avere un file chiamato “wc.py”, ci basterà scrivere “import wc.py”.

# Classi e oggetti

## Tipi personalizzati

Come esempio, creeremo un tipo che chiameremo Punto, che rappresenta un punto in un piano cartesiano bidimensionale.

Class [nome classe]:

dettagli classe

Class Punto:

x

y

nuovo = Punto()

Ci sono alcuni modi per rappresentare i punti in Python:

* Memorizzare le coordinate in due variabili separate, x e y.
* Memorizzare le coordinate come elementi di una lista o di una tupla.
* Creare un nuovo tipo che rappresenti i punti come degli oggetti.

Un tipo personalizzato, definito dal programmatore, è chiamato anche classe.

## Attributi

Si possono assegnare dei valori ad un’istanza usando la notazione a punto. Questa sintassi è simile a quella usata per la selezione di una variabile appartenente ad un modulo.

nuovo.x = 3

nuovo.y = 4

## Istanze come valori di ritorno

Le funzioni possono restituire istanze.

def centro (punto):

p = (punto.x + punto.y)/2

**return p**

## Copia

La copia di un oggetto è spesso una comoda alternativa all’alias. Il modulo copy contiene una funzione, anch’essa di nome copy, che permette di duplicare qualsiasi oggetto.

Import copy

p2 = copy.copy(p1)

p1 is p2

>>> False

p1 == p2

>>> False

L’operatore is indica che p1 e p2 non sono lo stesso oggetto, come volevasi dimostrare. Ma attenzione, nel caso di istanze, il comportamento predefinito dell’operatore == è lo stesso dell’operatore is: controlla l’identità dell’oggetto e non l’equivalenza. Questo perché, per i tipi personalizzati, Python non sa cosa debba essere considerato equivalente.

Questa operazione è chiamata copia shallow (o copia superficiale) perché copia l’oggetto ed ogni riferimento che contiene, ma non gli oggetti contenuti.

il modulo copy è dotato anche di un altro metodo chiamato deepcopy che non solo copia l’oggetto, ma anche gli oggetti a cui si riferisce, e gli oggetti a cui questi ultimi a loro volta si riferiscono, e così via.

box2 = copy.**copy**(box)

box2 is box

>>> False

box2.angolo is box.angolo

>>> True

box3 = copy.**deepcopy**(box)

box3 is box

>>> False

box3.angolo is box.angolo

>>> False

# Classi e funzioni

## Funzioni pure

Una funzione è pura quando non modifica alcuno degli oggetti che le vengono passati come argomento e, oltre a restituire un valore, non ha effetti visibili come visualizzare valori o chiedere input all’utente.

## Modificatori

Ci sono casi in cui è utile che una funzione possa modificare gli oggetti che assume come parametri. I cambiamenti risulteranno visibili anche al chiamante. Funzioni che si comportano in questo modo sono dette modificatori.

## Sviluppo prototipale e sviluppo pianificato

Una possibile tecnica di sviluppo è “prototipo ed evoluzioni”: per ogni funzione, si inizia scrivendo una versione grezza (prototipo) che effettui solo i calcoli fondamentali, provandola e via via migliorandola e correggendo gli errori.

Sebbene questo approccio possa essere abbastanza efficace, specie se non si ha un’adeguata conoscenza del problema, può condurre a scrivere del codice inutilmente complesso (perché deve affrontare molti casi particolari) e poco affidabile (dato che è difficile essere certi che tutti gli errori siano stati rimossi).

Un’alternativa è lo sviluppo pianificato, nel quale una conoscenza approfondita degli aspetti del problema da affrontare rende la programmazione molto più semplice.

# Classi e metodi

## Funzionalità orientate agli oggetti

Python è un linguaggio di programmazione orientato agli oggetti, in altre parole contiene delle funzionalità a supporto della programmazione orientata agli oggetti, che ha le seguenti caratteristiche distintive:

* I programmi includono definizioni di classi e metodi.
* Buona parte dell’elaborazione è espressa in termini di operazioni sugli oggetti.
* Gli oggetti corrispondono spesso ad un oggetto o concetto del mondo reale, mentre i metodi che operano sugli oggetti corrispondono spesso al modo in cui gli oggetti interagiscono tra loro nella realtà quotidiana.

Da un punto di vista logico, i metodi sono la stessa cosa delle funzioni, ma con due differenze sintattiche:

* I metodi sono definiti all’interno di una definizione di classe, per rendere esplicita la relazione tra la classe stessa ed il metodo.
* La sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione.

## Stampa di oggetti

Ci sono due modi di chiamare la funzione, quello classico oppure la scrittura suffissa mettendo oggetto.metodo(), questo perché Python identifica automaticamente la classe di appartenenza dell’oggetto e quindi ricerca i metodi della sua classe; una volta trovato, passa automaticamente l’oggetto al metodo.

|  |  |
| --- | --- |
| class Tempo:  """Rappresenta un'ora del giorno."""  def stampa\_tempo(self):  print('%.2d:%.2d:%.2d' % (self.ora, self.minuto, self.secondo)) | |
| Metodo classico | **Dot notation** |
| Tempo.stampa\_tempo(oggetto) | Oggetto.stampa\_tempo() |

Per convenzione, il primo parametro di un metodo viene chiamato self.

La ragione di questa convenzione è una metafora implicita:

* La sintassi di una chiamata di funzione, stampa\_tempo(inizio), suggerisce che la funzione è la parte attiva, che dice qualcosa del tipo: “Ehi, stampa\_tempo! Ti passo un oggetto da stampare!”
* Nella programmazione orientata agli oggetti, la parte attiva sono gli oggetti. L’invocazione di un metodo come inizio.stampa\_tempo() dice: “Ehi, inizio! Stampa te stesso!”

def viene\_dopo(self, other):

return self.tempo\_in\_int() > other.tempo\_in\_int()

>>> fine.viene\_dopo(inizio)

True

## Il metodo speciale init

1. Non trattano il dato nella sua forma rappresentata ma nella sua forma binaria in complemento a due. [↑](#footnote-ref-1)
2. La differenza verrà vista nelle fasi avanzate [↑](#footnote-ref-2)