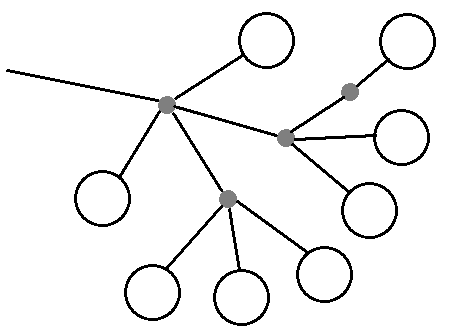
STRUTTURE DATI ASTRATTE E LORO IMPLEMENTAZIONI



Leonardo Vanni

versione 0 del 5 gennaio 2018

Sommario

[Premessa 2](#_Toc534540542)

[Le strutture dati astratte 2](#_Toc534540543)

[Le strutture dati primitive 3](#_Toc534540544)

[Concetto generale 3](#_Toc534540545)

[Le principali strutture primitive 5](#_Toc534540546)

[Strutture Booleane 5](#_Toc534540547)

[Strutture Byte 7](#_Toc534540548)

[Strutture Numeri Interi 10](#_Toc534540549)

[Le strutture dati composite 11](#_Toc534540550)

[Strutture Concatenate 11](#_Toc534540551)

[Strutture Articolate 16](#_Toc534540552)

[Articolazioni Sequenza 23](#_Toc534540553)

[Articolazioni Sequenza Stringa 26](#_Toc534540554)

[Articolazioni Insieme 28](#_Toc534540555)

[Rappresentazione canonica dei tipi di dato Python 29](#_Toc534540556)

# Premessa

Questo testo nasceva per chiarirmi su quali fossero i tipi di dato predefiniti nel linguaggio di programmazione Python. Poi nello scriverlo, il discorso si è generalizzato in una riflessione sui principali dati, che un linguaggio di programmazione è in grado di gestire.

Credo che ci sia qualche elemento di originalità nella definizione di cosa sia un dato elementare per un calcolatore o su cosa sia una struttura articolata.

Il testo è scritto in Italiano, perché pensato in Italiano. Non è un manuale tecnico di istruzioni da eseguire, ma piuttosto cerca di individuare le finalità che hanno spinto alla costruzione di determinate astrazioni. Resta comunque valido un approccio pratico, che affianca alla lettura dei tentativi pratici da eseguire sull’interprete di Python.

# Le strutture dati astratte

Una **struttura dati astratta** è una descrizione di un dato, di un’informazione, utile affinché il dato possa essere scritto, letto, conservato e manipolato su un qualunque supporto informatico, attraverso l’uso di un qualunque software applicativo (APP).

Il concetto di struttura dati astratta si contrappone a quello di **tipo di dato**, che è invece la concreta realizzazione in uno specifico linguaggio di programmazione della struttura dati astratta.

Le strutture dati astratte possono essere suddivise in due grosse categorie: le strutture **dati primitive** e quelle **composite**.

Una struttura dati composita può sempre essere scomposta in un gruppo di strutture primitive. Le strutture dati composite sono perciò anche dette strutture container.

# Le strutture dati primitive

## Concetto generale

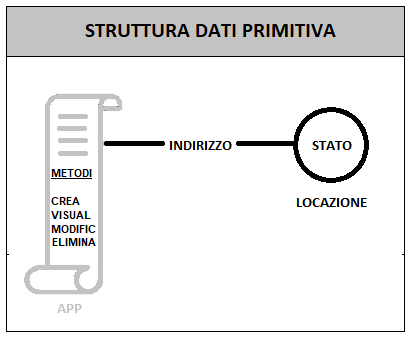


figura 1: rappresentazione schematica di una struttura dati primitiva.

La figura 1 è una rappresentazione schematica della struttura dati più semplice possibile: la struttura dati primitiva. Lo schema è finalizzato a descrivere come l’utente dell’app e più specificatamente del linguaggio di programmazione possa interagire con una struttura dati primitiva.

La struttura dati primitiva è costituita essenzialmente da una coppia inseparabile indirizzo-locazione e dai suoi metodi. La locazione è un supporto fisico, che può assumere diversi stati di funzionamento, mentre l’indirizzo è uno strumento che l’utente dell’app può usare per identificare e manipolare lo stato della locazione stessa. I metodi invece sono gli strumenti associati alla struttura affinché l’app possa crearla, visualizzarla e modificarla.

L’**indirizzo** è una correlazione, un’associazione diretta tra i due spazi posti alle sue estremità: l’app e la locazione. L’utente può agire direttamente solo nello spazio dell’app, dove potrebbe ad esempio premere un pulsante in una finestra o scrivere un comando. L’indirizzo è costituito da una sequenza ordinata di bit non modificabili durante l’esecuzione dell’app. Essendo l’indirizzo costituito da una sequenza ordinata di bit è sempre rappresentabile in modo univoco come un numero naturale, infatti la notazione binaria di un numero naturale è proprio una sequenza ordinata di bit. Ovviamente in qualunque macchina reale il numero degli indirizzi non può che essere un numero finito.

L’importanza dell’indirizzo è data dal fatto che l’app non potrebbe mai conoscere lo stato della locazione se non avesse l’indirizzo a cui riferirsi.

Sul lato dell’app l’indirizzo coincide con un “nome della variabile” o “nome di riferimento” o appellativi simili.

Anche la **locazione** è costituita da una sequenza ordinata di bit, ma a differenza dell’indirizzo, l’app li può modificare. I bit della locazione sono infatti costituiti da apparati in grado di mantenere stabilmente due stati di funzionamento alternativi, ma che possono essere modificati da un circuito elettronico esterno. Tipicamente ogni bit è realizzato da un circuito con un terminale di uscita, che fornisce una tensione elettrica di 5 V o di 1 V rispetto a terra.

Ogni particolare stato di funzionamento dei bit di una locazione costituisce uno dei possibili stati di funzionamento della locazione o detto più semplicemente uno stato della locazione. In modo analogo agli indirizzi il numero di stati di una locazione in una macchina reale non può che essere un numero finito, rappresentabile con un insieme finito di numeri naturali. Il numero totale degli stati assumibili da una locazione dipende dal numero di bit che costituiscono la locazione stessa. Ad esempio una locazione con 4 bit può assumere 16 stati, mentre una con 8 bit ne può avere 256. Questa semplice messa in relazione diretta tra gli stati della locazione e insiemi finiti di numeri naturali porta talvolta a chiamare le strutture dati primitive, come strutture dati numeriche, anche se la denominazione è per certi versi fuorviante.

Sul lato app (linguaggio di programmazione) la rappresentazione di uno stato di una locazione potrebbe avvenire usando insiemi di numeri naturali in notazione binaria, ma avviene più spesso facendo ricorso a specifiche sequenze di caratteri alfanumerici facilmente riconoscibili dall’utente.

Si può notare infine come in ultima analisi gli stati assunti dalla locazione siano il software custodito dall’hardware locazione.

**I METODI**. Per una struttura dati primitiva i metodi astratti fondamentali sono:

CREA

VISUALIZZA

MODIFICA

ELIMINA

I metodi CREA generano la struttura dati, impostando o meno il suo stato a partire da una rappresentazione dell’app o derivandolo da altre strutture dati esistenti.

I metodi VISUALIZZA sono metodi in grado di rendere noto lo stato di una locazione all’utente.

I metodi MODIFICA cambiano lo stato di una locazione esistente, impiegando o meno altre strutture dati dello stesso tipo.

I metodi ELIMINA liberano le risorse hardware occupate dalla struttura dati precedentemente creata dall’app.

## Le principali strutture primitive

Le strutture primitive si distinguono tra loro per il numero di stati assumibili dalla locazione e per i loro metodi propri.

Le strutture primitive più usate sono:

booleano

byte

intero

## Strutture Booleane

Le strutture dati booleane hanno locazioni in grado di assumere soltanto due stati.

Gli stati di una struttura booleana sono facilmente descrivibili con le notazioni e i metodi dell’algebra di Bool, Si fa riferimento spesso a descrizioni quali: Vero/Falso, On/Off, 1/0.

Dall’algebra di Bool derivano direttamente 4 metodi di creazione interni, quali:

AND OR XOR NOT

rappresentazione degli STATI della locazione:

in Python,

True, False

in C++,

true, false

1, 0 // alternativa equivalente

Metodi CREA:

in Python,

a = bool() # a è il nome arbitrario assegnato alla struttura

a = bool(True) # mentre *bool()* è una funzione predefinita creatrice;

a = True # le due ultime notazioni sono equivalenti.

c = bool.\_\_and\_\_(a,b) # creazione quale risultato del metodo AND

c = a.\_\_and\_\_(b) # in tutte e quattro le notazioni il metodo produce

c = a & b # una nuova struttura booleana nominata c

c = a and b # Operazioni analoghe sugli altri operatori booleani

Nota che in Python lo stesso metodo può essere richiamato ricorrendo a 4 notazioni equivalenti e alternative:

nome\_tipo\_struttura.nome\_del\_metodo(nome\_struttura\_1, nome\_struttura\_2)

nome\_struttura\_1.nome\_del\_metodo(nome\_struttura\_2)

nome\_struttura\_1 simbolo\_metodo nome\_struttura\_2

nome\_struttura\_1 parola\_chiave nome\_struttura\_2

in C++,

bool a; // a è il nome arbitrario assegnato alla struttura

bool a = true; // mentre *bool* è un comando di assegnazione di tipo

c = a && b; // la struttura c deve essere stata precedentemente creata

c = a and b;

***Metodi VISUALIZZA****:*

I metodi per visualizzare lo stato di una locazione nei linguaggi di programmazione più vecchi non potevano essere altro che una stampa (PRINT), mentre per le app più moderne che usano un’interfaccia a riga di comando o grafica interattiva con l’utente, i metodi di visualizzazione sono più vari, ad esempio un immagine che cambia colore o un messaggio che compare.

Se si usa Python in modalità interattiva la visualizzazione dello stato della locazione è immediata, basta digitare il nome della variabile e premere invio.

>>> a

True

>>>

Se invece la nostra app è eseguita con interfaccia a riga di comando non interattiva, la modalità più semplice di visualizzazione è la stampa sulla riga al momento dell’esecuzione.

in Python,

print(a) # farà comparire una rappresentazione di a quando eseguito

in C++,

cout << a; // richiede prima il richiamo della libreria

// di C++ per gestire il terminale, #include <iostream>

***Metodi MODIFICA****:*

Basta assegnare al nome di una struttura esistente un metodo CREA.

***Metodi ELIMINA****:*

In Python,

del a # elimina la struttura dati, a.

del(a)

## Strutture Byte

Il byte è una struttura dati primitiva con 256 stati assumibili dalla locazione. La sua importanza risiede nel fatto che in molte macchine una locazione di memoria in grado di funzionare da byte è anche la locazione più piccola a cui può far riferimento il processore.

Il byte è anche detto dato grezzo (raw data), perché per esso si prescinde da un suo uso di qualunque sorta (numero, lettera, nota musicale, ecc.).

Resta tuttavia l’esigenza di poterlo rappresentare.

Come per tutti gli altri dati primitivi, la rappresentazione più semplice è quella di associare a ciascuno dei suoi 256 stati un numero naturale nell’insieme che va da 0 a 255. Spesso in informatica tale numero viene espresso usando la notazione esadecimale e più raramente quella binaria. Così ad esempio gli stati 77 e 255 possono essere descritti come:

(77)dec = (4D)hex = (1001101)bin

(255)dec = (FF)hex = (11111111)bin

Una ulteriore rappresentazione del byte utile per descrivere l’hardware in modo concettuale e quella della figura sottostante. La locazione del byte è rappresentata come qui come un insieme ordinato di 8 bit, mentre lo stato dei singoli byte con una colorazione: nero bit in stato basso (spento), bianco bit in stato alto (acceso). Allo stato alto potrebbe ad esempio corrispondere un circuito elettronico, che fornisce sul terminale di uscita (filo di rame) una tensione di 5 V, mentre allo stato basso una tensione di 1 V. Durante il funzionamento della macchina la parte indirizzo della struttura primitiva rimane immutato, mentre la locazione assume i diversi stati di funzionamento, che gli sono ordinati dal processore.

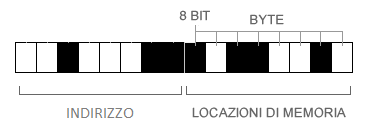
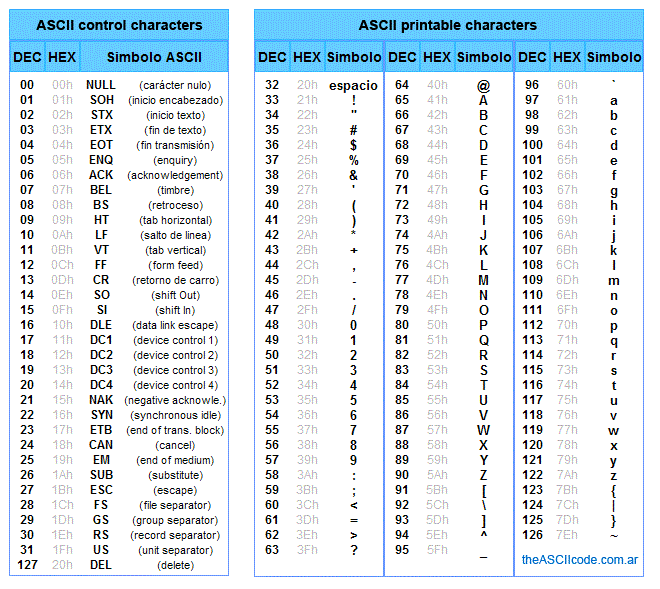


figura 2: rappresentazione fisica di una struttura byte.

Un’ulteriore rappresentazione storica del byte è quella dello standard ASCII.

ASCII (American Standard Code for Information Interchange) fu promulgato da ANSI nel 1968 con l’intento di usare il byte per i caratteri delle telescriventi, dove figuravano le lettere latine, i numeri ed alcuni comandi utili al funzionamento della telescrivente.

Tabella 1: codifica alfanumerica ASCII.



Stante la sua natura di dato grezzo al byte non è associato alcun metodo proprio.

In Python la struttura primitiva byte viene implementata con un tipo di dato predefinito, il ***bytes***. Il ***bytes*** Python non è una struttura primitiva, perché costituita da una serie ordinata di byte. Nei singoli elementi del tipo *bytes* si può però riconoscere l’implementazione della struttura dati astratta primitiva byte.

In C++ il byte viene implementato col tipo predefinito *unsigned char*, il cui nome fa evidente riferimento alla sua rappresentazione con lo standard ASCII.

**rappresentazione canonica degli STATI della locazione**:

in Python,

b’L’ # dove al posto della lettera L

# può esserci qualunque altro carattere ASCII

b’\x4C’ # dove 4C è un numero naturale in notazione esadecimale

# compreso tra 0 e 255

in C++,

‘L’; // oppure qualunque altro carattere ASCII

**Metodi CREA**:

in Python,

a = bytes()

a = b’L’ # dove al posto della lettera L

# può esserci qualunque altro carattere ASCII

a = b’\x4C’ # dove 4C è un numero naturale in notazione esadecimale

# compreso tra 0 e 255

in C++,

unsigned char b; // dove al posto della lettera L

unsigned char b = ‘L’; // può esserci qualunque altro carattere ASCII

## Strutture Numeri Interi

Come facilmente intuibile dal nome, le strutture per numeri interi sono quelle in cui gli stati della locazione sono posti in correlazione biunivoca con un insieme finito di numeri interi (relativi). Su quanto grande debba essere questo insieme non c’è accordo, però nei limiti di una macchina si può arrivare a locazioni con circa 4 miliardi di stati della locazione, per locazioni realizzate con 32 bit (4 byte).

Gli operatori interni della struttura sono SOMMA, SOTTRAZIONE, MOLTIPLICAZIONE, DIVISIONE INTERA. Da notare che la divisione matematica di numeri interi non fa parte dei metodi di struttura, perché può produrre numeri con virgola periodici non rappresentabili con una struttura primitiva.

**rappresentazione degli STATI della locazione**:

in Python e in C++, tutte le notazioni numeriche decimali da -1000000000 a 1000000000

Si possono usare anche notazioni non decimali,

**0o**414 # ottale, dove il prefisso 0o viene seguito

# dalle cifre ottali

**0x**10C # esadecimale, dove il prefisso 0x viene

# seguito dalle cifre esadecimali

**0b**100001100 **#** binario, dove il prefisso 0b viene seguito # dalle cifre binarie

268 # decimale

**Metodi CREA**:

in Python,

a = int()

b = int(‘1256’)

b = 1256

c = int.\_\_add\_\_(a,b) # creazione quale risultato del metodo somma

c = a.\_\_add\_\_(b) # lo stesso per gli altri metodi

c = a + b

in C++,

int a;

a = 2;

int b = 1256;

c = a + b;

# Le strutture dati composite

Le strutture composite sono composizioni di strutture primitive. I metodi fondamentali di una struttura composita sono gli stessi di una struttura primitiva, a cui si aggiunge il metodo decomposizione per riottenere le strutture primitive originarie.

Le operazioni di composizione delle strutture primitive possono riguardare o le loro locazioni o i loro indirizzi. Si ottengono così due diverse tipologie fondamentali di strutture composite:

strutture per concatenazione delle locazioni, code e pile

strutture per articolazione degli indirizzi, alberi

## Strutture Concatenate

Una struttura concatenata è costituita da più locazioni poste in sequenza ordinata, delle quali solo una locazione terminale è dotata di indirizzo. La locazione terminale dotata di indirizzo è detta testa, mentre la terminazione opposta è detta fondo della concatenazione.

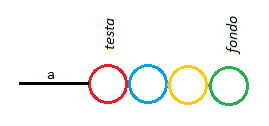


figura 3: struttura concatenata

La composizione di una concatenazione a partire da due strutture primitive può avvenire con due modalità alternative. Detti a e b gli indirizzi delle due strutture primitive si può concatenare la locazione di b alla locazione di a (concatenazione di fondo), oppure si può concatenare la locazione di b interponendola tra l’indirizzo a e la sua locazione originaria (concatenazione di testa). Quando l’inserimento di un nuovo elemento nella coda avviene per concatenazione di fondo si parla di strutture a coda (o lista), quando invece l’inserimento avviene per concatenazione di testa si parla di strutture a pila.

In figura 4 è mostrata in alto l’operazione di composizione di una coda denominata c, a partire da due strutture primitive a e b, mentre la stessa operazione è mostrata in basso nel caso in cui a sia già una coda.

In figura 5 è mostrata in alto l’operazione di composizione di una pila denominata c, a partire da due strutture primitive a e b, mentre la stessa operazione è mostrata in basso nel caso in cui a sia già una coda.

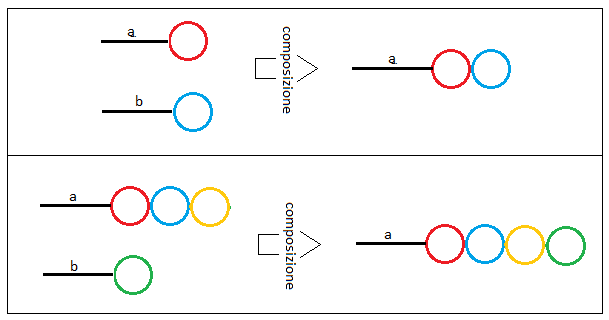


figura 4: operazione di composizione di una nuova **coda** da due strutture primitive (in alto) e composizione di una coda esistente con un dato primitivo (in basso).

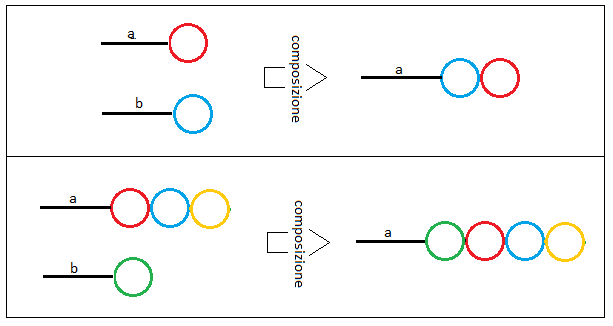


figura 5 operazione di composizione di una nuova **pila** da due strutture primitive (in alto) e composizione di una pila esistente con un dato primitivo (in basso).

In termini algebrici, detta CCF la concatenazione di fondo, essa può essere descritta come:

a = a CCF b

In modo del tutto analogo, detta CCT la concatenazione di testa, essa può essere descritta come:

a = a CCT b

Un’app può poi decomporre una struttura composita, usando il suo indirizzo. La struttura concatenata mostra però all’app un unico indirizzo collegato alla locazione di testa, perciò la decomposizione può procedere solo estraendo la prima locazione, a cui viene dato un nuovo indirizzo, e spostando l’indirizzo della struttura concatenata sulla locazione, che era collegata a quella di testa (vedi figura 6).

La modalità di decomposizione giustificano il nome di code e pile, infatti nelle code accade che il primo elemento inserito è anche il primo ad essere estratto (First In First Out, FIFO) e viceversa nella pile, l’ultimo arrivato è il primo ad essere estratto (Last In First Out, LIFO).

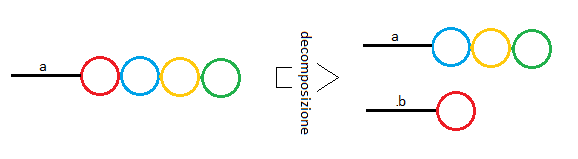


figura 6: operazione di decomposizione di una struttura concatenata.

Si può notare come code e pile differiscano tra loro per la modalità di composizione, ma non per quella di decomposizione.

In termini algebrici, detta DCT la decomposizione di testa, l’operazione di figura 6, può essere descritta come:

( a, b ) = a DCT

L’operazione ha un doppio effetto modifica a e produce la struttura b.

La decomposizione di testa è l’operazione inversa alla composizione di testa, mentre

l’asimmetria della DCT rispetto alla CCF fa sì che non ne sia l’operazione inversa.

Python dispone di un’implementazione delle strutture concatenate nella libreria standard denominata queue. Per creare una struttura concatenata a coda basta importare dalla libreria queue il tipo Queue, mentre per creare una pila, il tipo LifoQueue.

from queue import Queue # importa dalla libreria queue la classe Queue

coda1 = Queue() # creazione di una coda vuota

coda1.put('primo elemento') # inserimento degli elementi in coda

# esegue l’operazione di composizione della coda

# coda1 aggiungendo la stringa ‘primo elemento’

# vale a dire: a CCF (‘primo elemento’)

estratto = coda1.get() # esegue l’operazione di decomposizione della coda # coda1, vale a dire:

# ( coda1, estratto ) = coda1 DCT

from queue import LifoQueue # importa dalla libreria queue la classe LifoQueue

# Lifo sta per Last in first out

pila1 = LifoQueue() # creazione di una pila vuota

pila1.put('primo elemento') # inserimento degli elementi in pila

# esegue l’operazione di composizione della pila

# pila1 aggiungendo la stringa ‘primo elemento’

# vale a dire: a CCT (‘primo elemento’)

# da notare che Python usa lo stesso identificativo

# di metodo, put, ma assume comportamento

# diverso su tipi di strutture diverse

estratto = pila1.get() # esegue l’operazione di decomposizione della coda # pila1, vale a dire:

# ( pila1, estratto ) = pila1 DCT

Le strutture a coda sono spesso impiegate nella scambio di dati tra app, che funzionano simultaneamente. Se ne parla anche come di comunicazione tra processi (Inter Process Communication, IPC). L’esempio più comune è quello delle code usate come struttura dati intermedia fra app e Sistema Operativo (SO) per il salvataggio dei dati nei file registrati sulle memorie permanenti. Il SO è più lento dell’app nel gestire i dati provenienti dalle memorie permanenti ed è quindi utile creare una struttura dati intermedia (data buffer) in cui l’app possa completamente scrivere i propri dati, senza aspettare che il SO li abbia salvati su una periferica più lenta.

I file sono le strutture dati, che i sistemi operativi impiegano per immagazzinare i dati forniti dalle app e che poi restituiscono per il loro uso.

Pur essendoci differenze tra sistemi operativi, generalmente il sistema operativo fornisce all’app (linguaggio di programmazione) una struttura a coda, i cui dati primitivi costituenti sono i byte custoditi dal sistema operativo. Questa struttura a coda fornita dal S.O. duplica nella RAM i dati contenuti nelle memorie permanenti.

Questa costruzione della coda nella RAM è eseguita in Python dal comando predefinito, open().

Ad esempio in Python:

a = open(file = ‘Pippo’, mode = ‘rb’)

# crea nella RAM una coda di byte di nome **a** partendo

# dai dati del file Pippo

*b = a.read(1)* # esegue l’operazione di decomposizione della coda

# a, vale a dire: ( a, b ) = a DCT

# da ricordare che il metodo read(1) elimina da a la

# locazione di testa

a.close() # il file non subisce variazioni, mentre la coda a

# viene eliminata

a = open(file = ‘Pippo’, mode = ‘wb’)

# crea nella RAM una nuova coda vuota di byte

# di nome **a**

a.write(b’\x9F’) # esegue l’operazione di composizione della coda

# a aggiungendo il byte b’x/9F’

# vale a dire: a CCF (b’x/9F’)

a.close() # il file Pippo viene creato dal S.O. con i dati

# contenuti dalla coda a e poi questa viene eliminata

# (con mode=’ab’ avrebbe appeso su un file esistente)

Da notare che in Python le code create con la funzione open() non realizzano un’implementazione completa della struttura dati astratta coda, perché in mode = ‘rb’ è possibile solo creare la coda dal file e poi decomporla, senza poter aggiungere altri elementi, ed analogamente in mode = ‘wb’ si può comporre la coda da strutture primitive, ma non la si può decomporre, se non inviandola al S.O. ed eliminandola.

## Strutture Articolate

Le strutture articolate si differenziano nettamente da quelle concatenate, perché possono restituire qualunque dato primitivo componente, attraverso una gerarchia di indirizzi discendente da un unico indirizzo, detto radice. Le strutture articolate di indirizzi assomigliano a quelle che comunemente sono dette strutture ad albero.

Al termine di ciascuna gerarchia di indirizzi si trovano gli indirizzi associati alle strutture primitive componenti (vedi figura 7). Mantenendo il parallelismo botanico, le locazioni poste agli estremi terminali prendono anche il nome di foglie.

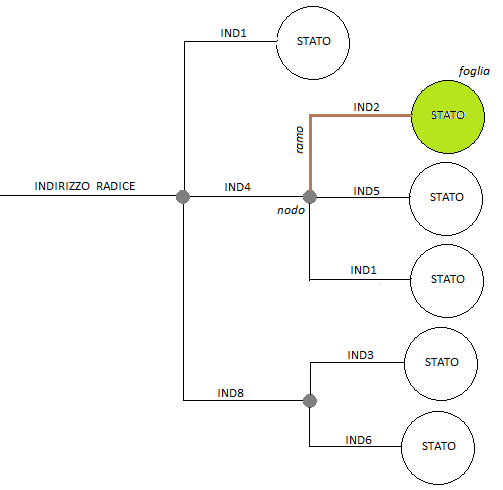


figura 7: struttura dati articolata.

L’elemento nevralgico che consente la costruzione di un albero sono delle strutture intermedie chiamate equivalentemente puntatori, articolatori o nodi.

I puntatori sono strutture, in cui gli stati della locazione rappresentano indirizzi di altre locazioni. Il nome puntatore è giustificato dal fatto che la loro locazione è impiegata per indicare (puntare) altri indirizzi.

Rispetto alla rappresentazione grafica delle locazioni primitive è conveniente darne una rappresentazione modificata, sostituendo al cerchio della locazione un punto, così come mostrato in figura 8.



figura 8: rappresentazione schematica di un puntatore

L’indirizzo di un puntatore viene anche detto chiave del puntatore.

Sempre con riferimento al puntatore della figura 8 , posto ad esempio che la locazione contenga l’indirizzo, b, allora l’app potrà gestire la locazione di b in termini algebrici con:

a.b

ovvero

(indirizzo puntatore).(indirizzo contenuto in locazione)

Ulteriore esempio: in figura 7 la locazione colorata in verde potrà essere raggiunta grazie ai puntatori con questo indirizzo articolato:

indirizzo radice.IND4.IND2

Restringendo la libertà di scelta sui nomi degli indirizzi contenuti dai nodi si ottengono i due principali sottotipi di strutture articolate: sequenze ed insiemi.

**\_\_\_\_\_\_\_\_\_ SEQUENZA**

**|**

**ARTICOLATE \_\_\_\_\_|**

**|**

**|\_\_\_\_\_\_\_\_\_ INSIEME**

Nel caso degli alberi sequenza le chiavi (indirizzi dei nodi) possono essere soltanto dei numeri naturali da 0 a N, mentre negli insiemi esiste un solo nodo e le chiavi solo uguali allo stato della locazione.

In figura 9 è rappresentato una struttura articolata sequenza. In questo caso la struttura primitiva con la locazione verde viene individuata dall’indirizzo articolato:

INDIRIZZO RADICE.1.0

In figura 10 è invece rappresentata una struttura articolata insieme. Nella struttura articolata insieme i singoli dati primitivi componenti sono unici (non è possibile ripeterli). In sostanza una struttura articolata insieme consente di collezionare i dati e verificare, tramite l’indirizzo radice se un dato appartiene o meno a quell’insieme.

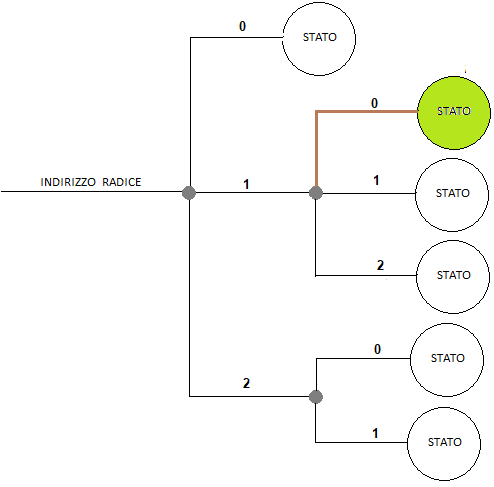


figura 9: struttura articolata sequenza.

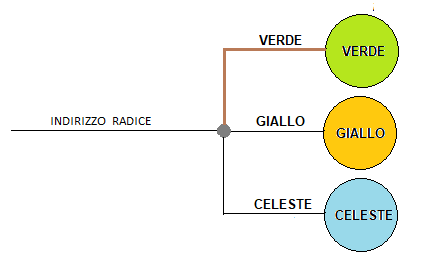


figura 10: struttura articolata insieme.

In Python una struttura articolata ad albero è realizzata nella sua forma più generale mediante la strutture predefinita “dictionary” (dizionario). Nei dizionari Python gli indirizzi sono chiamati “keys” (chiavi), mentre le foglie sono dette “values” (valori).

Ad esempio la struttura articolata di figura 11 raccoglie tre strutture primitive di tipo diverso: un booleano, un numero intero ed un byte.

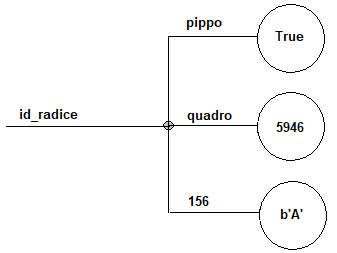


figura 11: esempio di semplice struttura articolata.

In Python la costruzione di una struttura articolata per rappresentazione diretta è con coppie indirizzo-locazione, separate da virgola e racchiuse in parentesi graffe. Così la struttura di figura 11 diventa:

id\_radice = {'pippo': True, 'quadro': 5946, 156: b'A'}

# o equivalentemente, disposto in colonne

id\_radice = {

'pippo' : True,

'quadro' : 5946,

156 : b'A'

}

Oppure si può creare una struttura articolata vuota ed aggiungere uno ad uno gli elementi primitivi costituenti:

id\_radice = dict() # crea un dizionario vuoto

id\_radice['pippo'] = True # aggiunge la chiave (indirizzo) 'pippo' col valore # booleano True

id\_radice['quadro'] = 5946

id\_radice[156] = b'A'

Operazione inversa dell’inserimento degli elementi primari è la loro estrazione. Per estrarre un indirizzo-locazione si può far riferimento al metodo pop() appartenente alla classe dictionary:

estratto = id\_radice.pop('quadro') # la variabile estratto raccoglie il valore

# associato alla chiave ‘quadro’

In modo simile il metodo get() copia il valore, però senza eliminarlo dal dizionario. La stessa operazione di copiatura può essere fatta per riferimento diretto:

copiato = id\_radice.get('quadro')

copiato = id\_radice['quadro'] # modo equivalente diretto

Python consente di creare strutture articolate con un numero arbitrario di nodi.

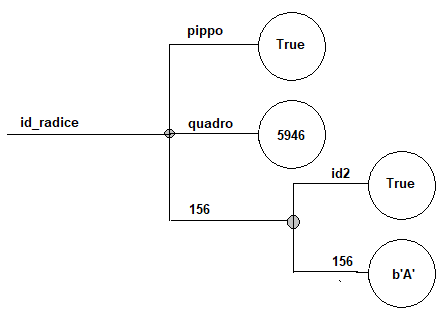


figura 12: struttura articolata con due nodi.

Ad esempio la struttura articolata di figura 12, può essere costruita come:

id\_radice = {'pippo': True, 'quadro': 5946, 156: {156: b'A', 'id2': True}}

# oppure

id\_radice = {

'pippo' : True,

'quadro' : 5946,

156 : {

156 : b'A',

'id2' : True

}

}

Tutti gli elementi di un dizionario sono semplicemente modificabile per assegnazione diretta; ad esempio:

id\_radice[‘pippo’] = 7456 # abbiamo cambiato non solo valore,

#ma anche tipo di dato

Python offre la possibilità di creare strutture articolate, oltreché con il tipo predefinito dictionary, anche mediante la procedura di creazione di un oggetto generico, **class**. In questo caso si definisce prima un modello della struttura con i suoi indirizzi e poi si possono creare specifiche strutture con i propri valori. L’unica limitazione di questa procedura è che non si possono usare indirizzi numerici. Ad esempio la struttura articolata di figura 11 diventa:

>>> class strut\_fig\_11: # creazione del modello di struttura

pippo = ''

quadro = ''

n156 = ''

>>> id\_radice = strut\_fig\_11() # creazione della struttura

>>> id\_radice.pippo = True # inserimento dei valori nelle foglie

>>> id\_radice.quadro = 5946

>>> id\_radice.n156 = b'A'

Come si vede l’identificativo numerico, 156, è stato sostituito da un identificativo valido per le variabili Python.

Nel caso della figura 12 la costruzione della struttura avviene invece mediante “annidamento”:

>>> class annidata:

id2 = ''

n156 = ''

>>> class strut\_fig\_12:

pippo = ''

quadro = ''

n156 = annidata()

>>> id\_radice = strut\_fig\_12()

>>> id\_radice.pippo = True

>>> id\_radice.quadro = 5946

>>> id\_radice.n156.id2 = True

>>> id\_radice.n156.n156 = b'A'

La definizioni delle **class** Python, oltreché nel app interprete, può avvenire in file di testo separati, il cui nome deve avere estensione “.py”. In questo modo il modello di struttura può essere importato in qualsiasi app Python, tramite il comando **import**. Ad esempio se abbiamo salvato il modello di struttura nel file, “esempio.py”, essa può essere semplicemente portata nella nostra app come:

import esempio

id\_radice = esempio.strut\_fig\_12()

Il file “esempio.py” è detto modulo o libreria. I moduli consentono di immagazzinare modelli di strutture, ma anche variabili e funzioni. Una volta importati i moduli, se ne può usare gli elementi contenuti con il comando:

nome\_modulo.nome\_elemento

Come si vede anche i moduli impiegano una struttura articolata degli indirizzi per gestire gli elementi che vi sono contenuti.

Le strutture articolate sono usate anche dai sistemi operativi per gestire i file. In questo caso le foglie sono i file, mentre i nodi sono le directory. Ad esempio la figura 13, mostra come il file manager di windows10 archivia il file con indirizzo “NOTICE” nel nodo (directory) “i386”, a sua volta puntato da “Drivers”, a sua volta puntato dal nodo radice “C:”.

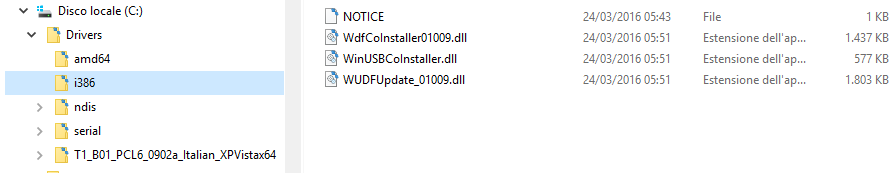


figura 13: esempio di file manager di windows10.

Windows10 consente altresì di gestire lo stesso file usando una notazione algebrica tramite l’app con riga di comando, PowerShell. In questo caso si avrebbe su riga di comando:

PS C:\Drivers\i386> ls

Directory: C:\Drivers\i386

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 24/03/2016 05:43 239 NOTICE

-a---- 24/03/2016 05:51 1470736 WdfCoInstaller01009.dll

-a---- 24/03/2016 05:51 589936 WinUSBCoInstaller.dll

-a---- 24/03/2016 05:51 1846040 WUDFUpdate\_01009.dll

La riga di comando costantemente mostra un nodo dell’intera struttura articolata che contiene i file (filesystem). Attraverso dei comandi è poi possibile gestire i file puntati dal nodo. In questo caso il comando **ls** mostra gli indirizzi dei file del nodo.

### Articolazioni Sequenza

Le strutture articolate sequenza possono essere pensate come un caso particolare di una struttura articolata ad albero, in cui gli indirizzi interni (tutti tranne la radice) sono dati da numeri naturali da 0 a N. Queste strutture sono quelle più facilmente realizzabili nelle memorie dei calcolatori e perciò sono presenti loro implementazioni in tutti i linguaggi di programmazione con vari nomi (vettori, array).

Python implementa le articolazioni indicizzate con il tipo predefinito **list** (scelta di nome infelice, perché con Lista si indica una struttura dati astratta diversa).

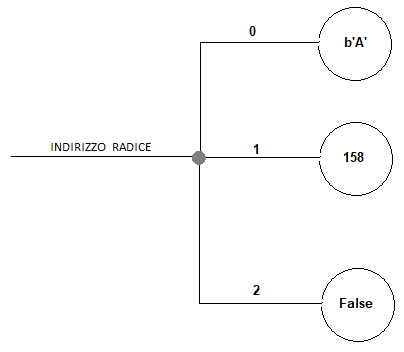


figura 14: esempio di struttura articolata sequenza.

La **creazione di un’articolazione** come quella in figura 14 può avvenire come al solito in Python o per rappresentazione diretta o con metodo costruttore.

indirizzo\_radice = [ b'A', 158, False] # creazione per assegnazione diretta canonica

indirizzo\_radice = [ #creazione per assegnazione diretta verticale

b'A',

158,

False

]

indirizzo\_radice = ['']\*3 # creazione di una sequenza con 3 indirizzi

indirizzo\_radice[1] = 158 # assegnazione diretta all’indirizzo [1]

indirizzo\_radice[0] = b'A'

indirizzo\_radice[2] = False

indirizzo\_radice = list() # creazione di una sequenza vuota

indirizzo\_radice.append(b'A') # aggiunta in fondo alla sequenza di un elemento

indirizzo\_radice.append(159)

indirizzo\_radice.append(False)

La lettura di un elemento presente nella sequenza è fatto per accesso diretto:

elemento\_letto = indirizzo\_radice[1] # legge e salva l’elemento n.1

Mentre l’**estrazione** di un elemento avviene con il metodo di struttura **pop()**:

elemento\_estratto = indirizzo\_radice.pop(1) # estrae l’elemento numero 1 e

# modifica così la sequenza

Le strutture sequenza con più nodi, come quella in figura\_ 15, vengono realizzate annidando nella locazione (nodo) della radice indirizzi che puntano ad ulteriori nodi.

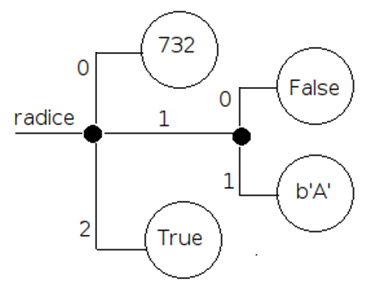


figura 15: sequenza con 2 nodi.

L'esempio di figura 15 può essere implementato come:

radice = [ 732, [False, b'A'], True] # creazione per assegnazione diretta canonica

# oppure in modo equivalente

radice = ['']\*3 # creazione di una sequenza con 3 indirizzi

radice[0] = 732 # assegnazione diretta all’indirizzo [0]

radice[1] = [False, b'A']

radice[2] = True

La lettura avviene sempre per accesso diretto:

elemento\_letto = radice[1] # legge e salva l’elemento n.1

Anche l’estrazione di un elemento avviene sempre con **pop()**:

elemento\_estratto = radice[1].pop(1) # estrae l’elemento numero 1 e

# modifica così la sequenza

In Python le articolazioni sequenza possono essere implementate anche con il tipo predefinito **tuple**. In questo caso però la struttura ha valori costanti, perciò viene chiamata **sequenza immutabile**. Una volta creata la struttura con i suoi valori, questi non possono essere ulteriormente modificati.

La struttura di figura 14 implementata come tuple si realizza con:

# creazione per assegnazione diretta in forma canonica

indirizzo\_radice = ( b'A', 158, False )

# equivalente senza parentesi

indirizzo\_radice = b'A', 158, False

#creazione per assegnazione diretta verticale

indirizzo\_radice = (

b'A',

158,

False

)

indirizzo\_radice = tuple([b'A', 158, False]) # creazione con metodo creatore

Una sequenza immutabile consente solo di leggere gli elementi contenuti per accesso diretto:

elemento\_letto = indirizzo\_radice[1] # legge e salva l’elemento n.1

# la scrittura è identica a quella delle list

### Articolazioni Sequenza Stringa

Le articolazioni sequenza stringa, o più brevemente stringhe, sono delle sequenze immutabili con un solo nodo e i cui dati primitivi possono essere soltanto caratteri alfanumerici. La loro importanza è talmente alta che sono presenti implementazioni in tutti i linguaggi di programmazione, anche i più vecchi.

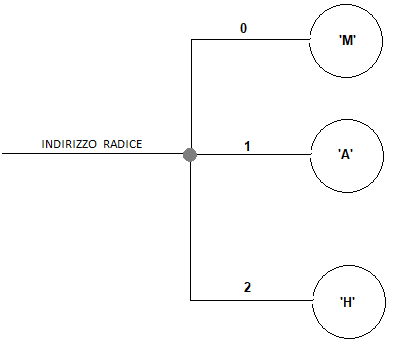


figura 16: stringa di 3 caratteri.

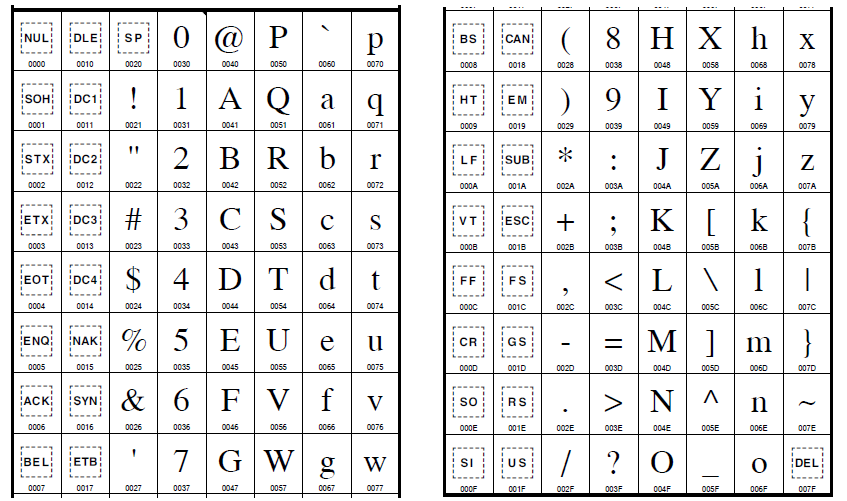
Fin dall’inizio i calcolatori oltreché gestire numeri hanno gestito parole e frasi intere. Le stringhe costituiscono l’esempio più chiaro dell’importanza delle strutture dati, che evidenzia come il valore dell’informazione appartenga all’intera struttura e non ai singoli elementi componenti. Infatti banalmente la parola ‘CIAO’ porta un significato, che nessuna delle lettere componenti possiede.

Il passaggio dalle lettere ai numeri binari contenuti nelle locazioni ha richiesto la scelta di una convenzione, di un codice per passare da lettera a numero binario (codifica) e viceversa (decodifica), che potesse essere assunto dalle locazioni del calcolatore.

Ad oggi lo standard, che cerca di rappresentare il maggior numero di lettere e vari simboli da tutto il mondo, è quello Unicode, che codifica oltre 1 milione di simboli. Per compattare la scrittura del numero, viene rappresentato spesso in base 16. Così l’emoticon “faccina che ride”, ☺ , corrisponde al numero 9786 (in formato esadecimale 263A), mentre la lettera ‘A’ è il numero 65 (esadecimale 41). Lo standard Unicode è a sua volta una specifica astratta, perché non dice quale debba essere il numero binario realmente presente in locazione. Al contrario la vecchia codifica “ASCII-estesa” associava i propri numeri ad 1 singolo byte. Il metodo era molto semplice, ma limitava i caratteri a 256.

Lo standard di codifica di Unicode universalmente impiegata è UTF-8 (Unicode Trasformation Format – 8 bit), che impiega un byte per la codifica dei numeri unicode più bassi e 2 byte per quelli più alti. A questo si aggiunge che la codifica dei numeri più bassi è identica a quella ASCII-estesa cosicché ne è diventata un’ulteriore estensione.

Tabella 2: primi numeri (esadecimali) della codifica astratta Unicode.



In Python le stringhe sono implementate con il tipo **str** e la creazione di una struttura stringa può avvenire come sempre per rappresentazione diretta o con metodo costruttore. Ad esempio la struttura di figura 16 diventa:

# creazione con notazione canonica avviene equivalentemente con 1, 2 o 3 virgolette

radice = 'MAH'

radice = "MAH"

radice = '''MAH'''

# creazione con metodo creatore

radice = str('MAH')

Essendo un le stringhe Python immutabili si possono soltanto visualizzare i singoli elementi componenti, ma non modificare la struttura.

valore\_letto = radice[1] # leggo il valore 'A', salvandolo in valore\_letto

Da notare in Python, che per tutte le strutture dati si può ottenere la rappresentazione canonica testuale in forma di stringa mediante la funzione **repr()**.

Da notare anche che Python mette a disposizione il metodo **encode()** per trasformare una stringa in una sequenza lineare di byte, che in Python corrisponde al tipo **bytes**.

### Articolazioni Insieme

Le articolazioni insieme corrispondono al concetto matematico di insieme. I suoi elementi sono unici ed i suoi metodi propri sono quelli dell’insiemistica: verifica se un elemento fa parte dell’insieme, intersezione, unione e differenza tra insiemi.

In Python l’implementazione degli insiemi è fatta mediante il tipo predefinito **set**. I dati contenuti devono essere di tipo immutabile.

La creazione di un insieme avviene nelle due modalità diretta e con metodo creazione. Ad esempio per la struttura in *figura* 10:

radice = {'verde', 'celeste', 'giallo'} # imputazione diretta forma canonica

radice = set() # creazione insieme vuoto

radice.add('verde') # aggiunta degli elementi

radice.add('celeste')

radice.add('giallo')

Per gli insiemi è fondamentale verificare se un elemento ne faccia o meno parte. In Python tale verifica viene fatta con l’operatore in, che restituisce un booleano:

test\_A = 'verde' in radice # il risultato sarà True

test\_B = 'rosso' in radice # il risultato sarà False

Per copiare i singoli elementi di un insieme si può ricorrere ad un costrutto di comandi Python come il ciclo **for ... in ...** :

ListaCopia = list() # creo una lista vuota,

# che ospiterà gli elementi dell’insieme

# il ciclo ***for*** scorrerà tutti gli elementi di radice

for elemento in radice:

ListaCopia.append(elemento)

Per eleminare un elemento dall’insieme:

radice.discard('giallo')

Metodi fondamentali tra insiemi:

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7}

# A unione B , A ∪ B

C = A.union(B) # C è {1, 2, 3, 4, 5, 6, 7}

C = A | B

# A intersezione B , A ∩ B

D = A.intersection(B) # D è {4, 5}

D = A & B

# A differenza B , A - B

E = A.difference(B) # E è {1, 2, 3}

E = A - B

## Rappresentazione canonica dei tipi di dato Python

|  |  |  |  |
| --- | --- | --- | --- |
| **ADT** | **Primitivo**  **Composto, Mutabile**  **Immutabile** | **Tipo** | **esempio di rappresentazione canonica** |
| Booleani | P, I | Bool | True | False |
| Byte | P, I | ----- | b’L’ | b’\x4C’ |
| Interi | P, I | Int | 268 | 0o414 | 0x10C | 0b100001100 |
| Code | C, M | Queue | --------------------------- |
| Pile | C, M | LifoQueue | --------------------------- |
| Articolate | C, M | Dict | {'pippo': True, 'quadro': 5946, 156: b'A'} |
| Sequenze | C, M | List | [ b'A', 158, False ] |
| Sequenze | C, I | Tuple | ( b'A', 158, False ) |
| Stringhe | C, I | Str | ‘qualunque carattere Unicode’ |
| Bytes | C, I | Bytes | b’qualunque carattere ASCII’ | b’\x4C’ |
| Insiemi | C, M | Set | {True, 5946, b'A'} |