

Le strutture dati in Python

Classe III

I crediti fotografici delle immagini presenti in questo PowerPoint sono riportati all'interno del volume
informatica per progetti – Programmare in Python, di M. Fiore, A. Zanga, ISBN 9788808899477

Tipi di dati semplici

Tipo	Codice	Valore	Esempio
Valore booleano	bool	{False, True}	>>> soleggiato = True
Numero intero	int	Insieme Z	>>> prodotti = 42
Numero in virgola mobile	float	Insieme R	>>> sconto = 0.31
Numero complesso	complex	Insieme C	>>> fase = 1.23 + 3.50j

Tipi di dati complessi

Esistono tipi di dati che vengono definiti **complessi** perché raggruppano una struttura di dati in un'unica variabile: questi dati possono a loro volta essere semplici o complessi, dando origine a strutture anche molto articolate.

Un **tipo di dato complesso** è una struttura dati che può contenere più elementi, alcuni anche di tipo diverso.

In Python i tipi di dati complessi sono molto importanti e molto utilizzati e includono **liste, tuple, insiemi e dizionari**. Anche il tipo di dato **stringa** è un tipo complesso, ma è un po' particolare.

Stringhe

In Python non esiste il tipo di dato semplice “carattere”, ma esiste un tipo di dato complesso che si chiama **stringa** e che è molto utilizzato.

Una **stringa** è una sequenza di caratteri **immutabile**, che può contenere lettere, numeri, simboli e spazi.

Si possono definire utilizzando singoli apici (') o doppi apici (").

```
>>> parola = "computer"
```

```
>>> frase = "questa è una stringa"
```

Operatore []

Una stringa è una sequenza di caratteri ed è possibile accedere al singolo carattere attraverso l'operatore parentesi quadre [].

```
>>> lettera_singola = parola[1]
>>> lettera_singola
'o'
```

Le posizioni dei caratteri nelle stringhe partono da zero e non da uno. Se rappresentiamo la stringa *parola* indicando le posizioni dei suoi caratteri, abbiamo:

parola	c	o	m	p	u	t	e	r
posizione	0	1	2	3	4	5	6	7

L'operatore [] è molto utile perché ci permette di accedere al singolo carattere della stringa conoscendo la sua posizione.

Indice

La posizione del carattere all'interno della stringa è indicata dall'**indice**, che è un numero intero che parte da zero e arriva fino alla *lunghezza della stringa - 1*.

L'**indice** è un numero intero che indica la posizione di un carattere all'interno della sequenza.

L'indice può essere anche *una variabile o un'espressione*:

```
>>> i = 2
>>> parola[i]
'm'
>>> parola[i + 1]
'p'
```

L'indice può essere anche un *numero negativo* e in questo caso indica la corrispondente posizione partendo dalla fine della stringa:

```
>>> parola[-1]
'r'
>>> parola[-2]
'e'
```

Lunghezza di una stringa

La funzione `len()` è una funzione predefinita che ci restituisce la lunghezza di una stringa, cioè il numero di caratteri che la stringa contiene:

```
>>> frase = "questa è una stringa"
```

```
>>> len(frase)
20
```

Utilizzando in modo combinato l'indice e la funzione `len` siamo in grado di estrarre tutti i caratteri che compongono la stringa con un semplice ciclo di `for`.

```
1 parola = "computer"
2 for i in range(len(parola)):
3     print(parola[i])
```

Per scorrere tutti i caratteri della stringa possiamo usare anche un'altra forma del ciclo `for`, che usa l'operatore `in`:

```
1 for lettera in parola:
2     print(lettera, end = " ")
```

Slicing

Uno **slice** è una porzione di una stringa, o sottostringa, contenuta tra due indici. È possibile selezionare porzioni di stringhe con un'operazione definita slicing.

```
>>> stringa = "Linguaggio Python"
>>> stringa[0:6]
'Lingua'
```

L'istruzione `stringa[0:6]` ci permette di estrarre i caratteri che vanno dall'indice 0 (compreso) all'indice 6 (escluso)
Se vogliamo estrarre la sottostringa finale:

```
>>> stringa[11:17]
'Python'
```

Casi d'uso slicing

stringa	L	i	n	g	u	a	g	g	i	o		P	y	t	h	o	n
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Slicing	Sottostringa	Spiegazione
<code>>>> stringa[0:6]</code>	'Lingua '	porzione di stringa dall'indice 0 incluso all'indice 6 escluso
<code>>>> stringa[:6]</code>	'Lingua '	porzione di stringa dall'inizio all'indice 6 escluso
<code>>>> stringa[11:17]</code>	'Python '	porzione di stringa dall'indice 11 incluso all'indice 17 escluso
<code>>>> stringa[11:]</code>	'Python '	porzione di stringa dall'indice 11 incluso al termine della stringa
<code>>>> stringa[4:10]</code>	'aggio'	porzione di stringa dall'indice 4 incluso all'indice 10 escluso
<code>>>> stringa[4:4]</code>	''	stringa vuota, quando il primo indice è maggiore o uguale al secondo
<code>>>> stringa[6:3]</code>	''	stringa vuota, quando il primo indice è maggiore o uguale al secondo
<code>>>> stringa[:]</code>	'Linguaggio Python '	tutta la stringa

Concatenazione di stringhe

Una stringa è **immutabile**: ciò vuol dire che non è possibile cambiare una stringa già dichiarata e usata.

Se vogliamo modificare una stringa la cosa migliore è crearne una nuova, per esempio come **concatenazione** di due stringhe.

La concatenazione è l'operazione di unione di stringhe, si esprime tramite il simbolo + e fornisce come risultato una stringa.

```
>>> nome = 'Mario'
>>> cognome = 'Rossi'
>>> user = nome + cognome
>>> user
'MarioRossi'
```

Metodi e funzioni per le stringhe - 1

Operatore/ Funzione/Metodo	Utilizzo	Esempio
<code>s1 + s2</code>	concatenazione	<pre>>>> s1 = 'Oggi' >>> s2 = 'piove' >>> s1 + s2 'Oggipiove'</pre>
<code>s1 * n</code>	ripetizione della stringa s1 per <i>n</i> volte	<pre>>>> s1 = 'Oggi' >>> s1 * 2 'OggiOggi'</pre>
<code>s1 == s2</code>	confronta s1 con s2: se sono uguali restituisce True, altrimenti restituisce False	<pre>>>> s1 = 'Oggi' >>> s2 = 'piove' >>> s1 == s2 False</pre>
<code>s1 < s2</code> <code>s1 > s2</code>	confronta s1 con s2: se s1 viene prima (o dopo) di s2 in ordine alfabetico restituisce True, altrimenti False.	<pre>>>> s1 = 'Oggi' >>> s2 = 'piove' >>> s1 < s2 True</pre>

Metodi e funzioni per le stringhe - 2

<code>s in s1</code>	controlla se <code>s</code> è presente nella stringa <code>s1</code> : in caso positivo restituisce <code>True</code> , altrimenti restituisce <code>False</code>	<pre>>>> s1 = 'Oggi' >>> 'g' in s1 True >>> 'M' in s1 False</pre>
<code>s1.upper()</code>	mette <code>s1</code> in maiuscolo	<pre>>>> s1 = 'Oggi' >>> s1.upper() 'OGGI'</pre>
<code>s1.lower()</code>	mette <code>s1</code> in minuscolo	<pre>>>> s1 = 'Oggi' >>> s1.lower() 'oggi'</pre>
<code>s2.capitalize()</code>	mette l'iniziale di <code>s2</code> in maiuscolo	<pre>>>> s2 = 'piove' >>> s2.capitalize() 'Piove'</pre>
<code>s.strip()</code> <code>s.strip(sottostr)</code>	elimina i caratteri di spaziatura inseriti all'inizio e alla fine della stringa (o della sottostringa)	<pre>>>> nome = ' Mario ' >>> nome.strip() 'Mario'</pre>
<code>s.count(sottostr)</code>	conta quante volte la sottostringa si ripete nella stringa	<pre>>>> frase = 'Amo la mamma' >>> frase.count('ma') 2</pre>

Vettori o array

In Python **non** esiste un tipo di dato **vettore**.

Un **vettore**, o **array**, è una struttura dati di dimensione fissa, che permette di memorizzare dati tutti dello stesso tipo semplice, identificati da un indice.

Le **caratteristiche** del vettore sono:

1. lunghezza fissa;
2. elementi tutti dello stesso tipo semplice;
3. indice per identificare la posizione dell'elemento nel vettore.

```
>>> array_interi = [4, 18, 23, 2, 145]
>>> array_interi
[4, 18, 23, 2, 145]
```

La **dichiarazione** di un vettore è piuttosto semplice:

1. il vettore è racchiuso tra parentesi quadre [];
2. gli elementi sono separati da virgole “,”;
3. gli elementi sono tutti dello stesso tipo.

Indice del vettore

Ogni elemento del vettore ha una posizione che è indicata dall'**indice**, che come per le stringhe parte dal valore 0.

array_interi	4	18	23	2	145
indice	0	1	2	3	4

E' possibile usare la funzione `len()` per trovare la lunghezza del vettore, che è fissata nel momento della sua dichiarazione.

```
>>> len(array_interi)
5
>>> array_interi[1]
18
```

Utilizzando in modo combinato l'indice e la funzione `len()` è possibile estrarre tutti gli elementi del vettore con un ciclo `for`.

```
1 array_interi = [4, 18, 23, 2, 145]
2 for i in range(len(array_interi)):
3     print(array_interi[i], end = " ")
4 print()
5 for intero in array_interi:
6     print(intero, end = " ")
```

Creare un vettore da valori in input

```
1  #Carica un array di 5 elementi interi
2  #dichiaro il vettore di 5 elementi tutti 0
3  array = [0, 0, 0, 0, 0]
4  for i in range(len(array)):
5      #controllo che l'input sia intero
6      err_intero = False
7      while not err_intero:
8          try:
9              array[i] = int(input("Inserisci un numero intero: "))
10             err_intero = True
11         except ValueError:
12             print("Inserire un numero intero!")
13 print("Array: ", array)
```

Creare un vettore di dimensione variabile

```
1  #Definisci un array di dimensione chiesta in input e caricalo con interi
2  #chiedo la dimensione e controllo sia un numero intero
3  err_int = False
4  while not err_int:
5      try:
6          dim = int(input("Inserisci la dimensione del vettore: "))
7          err_int = True
8      except ValueError:
9          print("Inserire un numero intero!")
10 #creo il vettore contenente tutti 0
11 array = [0] * dim
12 #carico i valori nel vettore
13 for i in range(len(array)):
14     #controllo che l'input sia intero
15     err_intero = False
16     while not err_intero:
17         try:
18             array[i] = int(input("Inserisci un numero intero: "))
19             err_intero = True
20         except ValueError:
21             print("Inserire un numero intero!")
22 print("Array: ", array)
```

Ricerca di un valore in un vettore

Per cercare un elemento in un vettore dobbiamo utilizzare un algoritmo che determina se l'elemento appartiene al vettore e restituisce il suo indice per poter identificare la sua posizione.

L'algoritmo più semplice che svolge questa funzione è l'algoritmo di *ricerca lineare*, che scorre gli elementi del vettore nel loro ordine e si ferma quando incontra l'elemento cercato.

Nel caso peggiore, cioè quello in cui l'elemento cercato si trova in fondo al vettore, l'algoritmo deve scorrere tutto il vettore.

L'algoritmo di ricerca lineare si chiama così perché è *lineare* il tempo di esecuzione per svolgere la sua ricerca.

```
1  #ricerca lineare in un vettore
2  def ricerca_array(array, elemento):
3      trovato = False
4      indice = -1
5      for i in range(len(array)):
6          if array[i] == elemento:
7              trovato = True
8              indice = i
9              break
10     return indice, trovato
11
```

Ordinamento di un vettore

La ricerca lineare di un elemento in un vettore scorre tutto il vettore per cercare l'elemento che cerchiamo: se il vettore fosse ordinato, la ricerca potrebbe fermarsi non appena abbiamo superato nell'ordine l'elemento da cercare.

Il tempo di esecuzione del nostro algoritmo di ricerca potrebbe diminuire (*potrebbe* diminuire, perché se l'elemento che cerchiamo si trova in fondo, l'algoritmo deve comunque scorrere tutto il vettore).

Un **algoritmo di ordinamento di un vettore** viene utilizzato per posizionare gli elementi di un insieme secondo una sequenza stabilita, in modo che ogni elemento sia minore o maggiore di quello che lo segue.

Esistono diversi algoritmi di ordinamento dei vettori. I due più semplici sono:

1. Selection Sort;
2. Bubble Sort.

Selection Sort

- L'algoritmo **Selection Sort** confronta il primo elemento del vettore con il secondo, e se è minore (o maggiore, a seconda di come vogliamo ordinare il vettore) lo scambia di posto.
- L'algoritmo continua in questo modo fino alla fine del vettore, poi passa a considerare il secondo elemento, che verrà confrontato con il terzo, e così via.
- Al termine di queste operazioni di confronto e di eventuale scambio, il vettore sarà ordinato.
- Si dice che l'algoritmo effettua gli scambi *in place*, poiché gli scambi degli elementi vengono fatti direttamente sul vettore, per cui al termine delle operazioni il vettore iniziale non ci sarà più, ma avremo il vettore con gli elementi ordinati.
- A quel punto potremo salvare il vettore ordinato e restituito dalla funzione in una nuova variabile, come *vettore_ordinato*.

```
1  #Ordinamento array con Selection Sort
2  def selection_sort(array):
3      for i in range(len(array)):
4          for j in range(i+1, len(array)):
5              if array[j] < array[i]:
6                  temp = array[i]
7                  array[i] = array[j]
8                  array[j] = temp
9      return array
```

Bubble Sort

- L'algoritmo Bubble Sort confronta gli elementi a coppie: il primo elemento del vettore viene confrontato con il secondo e, se è minore (o maggiore, a seconda di come si vuole ordinare il vettore), lo scambia di posto.
- L'algoritmo continua in questo modo, confrontando il secondo con il terzo e scambiandoli se necessario, e così via fino alla fine del vettore.
- Al termine di queste operazioni di confronto e degli eventuali scambi, il vettore sarà ordinato.
- Anche l'algoritmo Bubble Sort effettua gli scambi *in place* per cui al termine delle operazioni il vettore iniziale non ci sarà più ma avremo il vettore con gli elementi ordinati.
- A quel punto potremo salvare il vettore ordinato e restituito dalla funzione in una nuova variabile, come *vettore_ordinato*.

Bubble Sort

```
1  #Ordinamento array con Bubble Sort
2  def bubble_sort(array):
3      #flag indica quando fare gli scambi
4      flag = 1
5      #stop indica la penultima posizione dell'array
6      stop = len(array) - 1
7      while flag == 1:
8          #flag = 0 consente di uscire dal ciclo while
9          flag = 0
10         for i in range(stop):
11             if array[i] > array[i+1]:
12                 temp = array[i]
13                 array[i] = array[i+1]
14                 array[i+1] = temp
15                 #continuo a fare scambi
16                 flag = 1
17         #a ogni passaggio escludo l'ultimo elemento dell'array
18         stop = stop - 1
19     return array
```

Liste

In Python il tipo di dato vettore non esiste e il modo in cui è implementato nel linguaggio è attraverso la struttura dati complessa della **lista**.

Una **lista** in Python è una struttura di dati complessa, costituita da una sequenza di valori (anche di tipo diverso e a loro volta complessi), che non ha una lunghezza fissa e i cui elementi sono caratterizzati dall'indice che ne indica la posizione all'interno della sequenza.

Una lista può essere dichiarata in Python in diversi modi, usando sempre l'operatore parentesi quadre `[]` e separando gli elementi con la virgola.

```
>>> lista_interi = [4, 18, 23, 2, 145]
>>> lista_interi
[4, 18, 23, 2, 145]
```

Può anche essere dichiarata una lista vuota

```
>>> lista_vuota = []
>>> lista_vuota
[]
```

Caratteristiche delle liste - 1

Per scorrere gli elementi di una lista possiamo utilizzare l'indice che indica la posizione dell'elemento nella lista; il primo elemento ha indice zero, e l'ultimo ha indice pari a `len(lista)`

In Python, la lista è *l'implementazione della struttura vettore*.

Ma la lista, in Python, ha anche altre caratteristiche che i vettori non hanno:

1. una dimensione che può variare dinamicamente;
2. la possibilità di contenere al suo interno elementi di tipo diverso, anche di tipo complesso.

```
>>> lista_prodotti = ["latte", 1.30, "pane", 3.2, "gelato", 1.5]
>>> type(lista_prodotti)
<class 'list'>
```

Caratteristiche delle liste - 2

Le liste, al contrario delle stringhe, sono mutabili: utilizzando l'operatore `[]` possiamo accedere all'elemento della lista e fare un'assegnazione.

```
>>> lista_interi = [4, 18, 23, 2, 145]
>>> lista_interi[2] = 12
>>> lista_interi
[4, 18, 12, 2, 145]
```

Le operazioni di concatenazione, iterazione e di slicing valgono anche per le liste, così come i metodi per scorrere una lista (for con l'uso dell'operatore in).

Dimensione dinamica – aggiungere elementi

La dimensione di una lista può variare dinamicamente: è possibile creare una lista partendo dalla lista vuota e «appendendo» uno alla volta i nuovi elementi, che verranno inseriti nella lista dopo l'ultimo elemento presente.

```
>>> lista_vuota = []
>>> lista_vuota.append(18)
>>> lista_vuota
[18]
>>> lista_vuota.append(5)
>>> lista_vuota
[18, 5]
>>> lista_vuota.append(14)
>>> lista_vuota
[18, 5, 14]
```

Il metodo `append()` permette di aggiungere un elemento alla fine della lista.

Dimensione dinamica – eliminare elementi

Primo metodo: usare `pop(indice)`. In questo caso abbiamo bisogno di conoscere l'indice dell'elemento che vogliamo eliminare, a meno che non vogliamo eliminare l'ultimo elemento e in questo caso è sufficiente scrivere `pop()`. Il metodo `pop()` restituisce l'elemento rimosso dalla lista.

```
>>> lista_interi = [4, 18, 23, 2, 145]
>>> lista_interi.pop(1)
18
>>> lista_interi
[4, 23, 2, 145]
```

Secondo metodo: usare `remove(elemento)`. Questo metodo si può usare quando conosciamo qual è l'elemento da eliminare, ma per esempio non il suo indice. Il metodo `remove()` restituisce `None`.

```
>>> lista_interi = [4, 18, 23, 2, 145]
>>> lista_interi.remove(18)
>>> lista_interi
[4, 23, 2, 145]
```

Terzo metodo: usare l'operatore `del`. In questo caso bisogna indicare con precisione quale elemento eliminare, usando l'operatore `[]`. L'operatore `del` non restituisce l'elemento rimosso.

```
>>> lista_interi = [4, 18, 23, 2, 145]
>>> del lista_interi[1]
>>> lista_interi
[4, 23, 2, 145]
```

Liste parallele

Quando abbiamo più informazioni che si riferiscono allo stesso oggetto, possiamo usare le liste parallele.

Ad esempio se abbiamo più prodotti con la loro descrizione e il loro prezzo possiamo usare 2 liste parallele:

`lista_prodotti`

elemento	scarpe	calze	maglia
indice	0	1	2

`lista_prezzi`

elemento	25.90	2.5	15
indice	0	1	2

Allo stesso indice nelle due liste corrisponde logicamente lo stesso dato. Il prezzo del prodotto “scarpe” è 25.90 poiché entrambi gli elementi si trovano allo stesso indice 0 delle due diverse liste parallele.

Metodi e funzioni per le liste

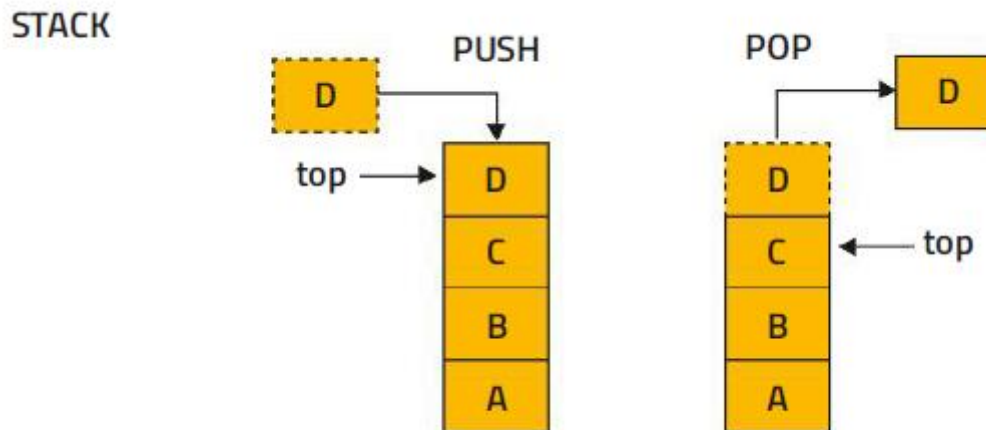
Operatore/Funzione/Metodo	Utilizzo
<code>l1 + l2</code>	concatenazione di liste
<code>l1 * n</code>	ripetizione della lista l1 per <i>n</i> volte
<code>l1 == l2</code>	confronta la lista l1 con la lista l2: se sono uguali restituisce True, altrimenti restituisce False
<code>l1 = l2</code>	l1 e l2 sono la stessa lista, quindi le operazioni effettuate su l1 sono effettuate anche su l2
<code>sum(l1)</code>	restituisce la somma degli elementi della lista l1
<code>max(l1) ; min(l1)</code>	restituiscono rispettivamente il valore massimo e il valore minimo della lista l1
<code>l1.sort()</code>	ordina la lista l1 <i>in place</i>
<code>l2 = sorted(l1)</code>	restituisce la nuova lista l2 che è la lista l1 ordinata; l1 non si modifica
<code>l1.append(elem)</code>	inserisce elem alla fine della lista l1
<code>l1.pop()</code>	elimina l'ultimo elemento della lista l1
<code>l1.pop(ind)</code>	elimina l'elemento di indice ind della lista l1
<code>l1.index(elem)</code>	restituisce l'indice dell'elemento elem della lista l1
<code>l1.insert(indice, elem)</code>	inserisce elem in posizione indice
<code>l1.insert(0,elem)</code>	inserisce elem all'inizio della lista l1

Stack

Uno **stack** (o **pila**) è un tipo di dato astratto di genere LIFO (*Last In First Out*), che permette l'inserimento e l'estrazione di valori solo dalla cima; per questo viene chiamata pila (come una pila di piatti).

Consente solo due tipi di operazioni:

1. push: per inserire un elemento in cima alla pila;
2. pop: per estrarre un elemento dalla cima della pila.



Stack e lista

Lo stack viene implementato in Python usando una lista.

E' possibile definire le due operazioni `push` e `pop` facendo inserimenti ed estrazioni solo dal fondo della lista:

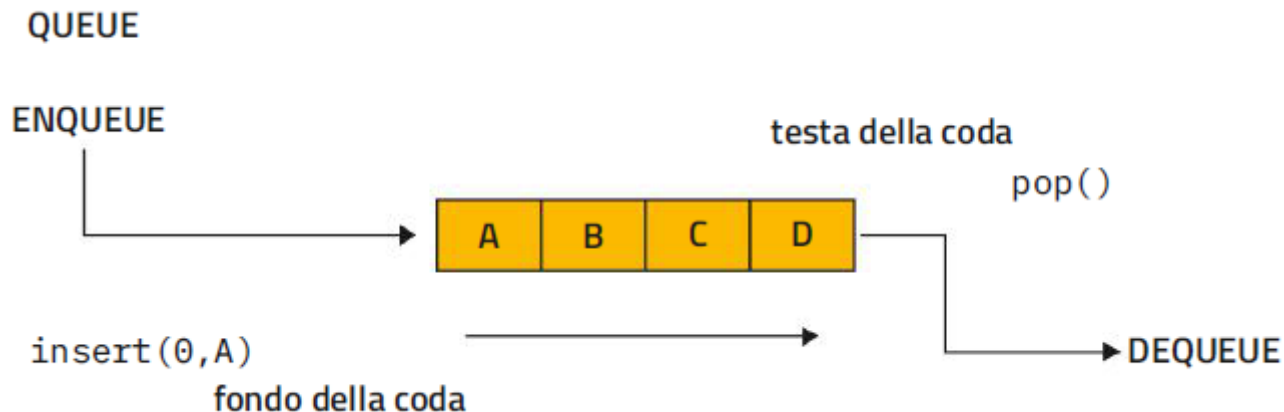
- usando il metodo `append()` per inserire un elemento alla fine
- usando il metodo `pop()` che permette di estrarre l'ultimo elemento della lista

Queue

Una **queue (o coda)** è un tipo di dato astratto di genere FIFO (*First In First Out*), che permette la gestione di una sequenza di valori aggiungendo entità a un estremo e rimuovendole dall'altro estremo, come in una coda di persone in attesa di fare il biglietto per assistere a un concerto.

Le operazioni possibili per una coda sono solo due:

1. enqueue, per inserire un elemento in fondo alla coda;
2. dequeue, per estrarre l'elemento in testa alla coda.



Queue e lista

Anche la queue viene implementata in Python utilizzando le liste.

Le due uniche operazioni possibili, `enqueue` e `dequeue`, vengono realizzate attraverso:

- il metodo `insert(0, elemento)`, che consente l'inserimento in posizione di indice 0 (quindi all'inizio della lista),
- il metodo `pop()`, che consente di estrarre l'ultimo elemento della lista

Tuple

Una **tupla** in Python è una sequenza di valori, anche di tipo diverso, che non ha una lunghezza fissa e i cui elementi sono caratterizzati dall'indice che ne indica la posizione nella sequenza.

La tupla rappresenta una sequenza di valori spesso racchiusa tra parentesi tonde () e separati dalla virgola.

La tupla è **immutabile** (come la stringa) quindi, una volta che una tupla è stata già dichiarata, non è possibile fare un'assegnazione a uno dei suoi elementi.

Per questa sua caratteristica, in genere si utilizza la tupla per rappresentare dati che non cambiano, come ad esempio i nomi dei giorni della settimana.

```
>>> giorni_feriali = ('lunedì', 'martedì', 'mercoledì', 'giovedì', 'venerdì', 'sabato')
>>> type(giorni_feriali)
<class 'tuple'>
```

Uso delle tuple

Per creare una tupla con un solo elemento è necessario aggiungere la virgola dopo l'elemento, altrimenti la tupla non viene riconosciuta come tale, ma come tipo dell'elemento inserito.

Se vogliamo scambiare tra loro i valori di due variabili `var1` e `var2`, possiamo usare le tuple. A sinistra dell'uguale abbiamo una tupla di variabili e a destra una tupla di espressioni. Per effettuare correttamente questa assegnazione è necessario che il numero delle variabili a sinistra sia uguale al numero delle espressioni a destra dell'assegnazione.

```
>>> tupla_stringa = ('sole')
>>> type(tupla_stringa)
<class 'str'>
>>> tupla_stringa = ('sole',)
>>> type(tupla_stringa)
<class 'tuple'>
>>> tupla_int = (8)
>>> type(tupla_int)
<class 'int'>
>>> tupla_int = (8,)
>>> type(tupla_int)
<class 'tuple'>
```

```
>>> var1, var2 = var2, var1
```

Insiemi

Vettori, liste e tuple sono collezioni *sequenziali*, in cui ogni elemento ha una precisa posizione all'interno della collezione, al contrario di quanto succede per gli insiemi.

Un **insieme** è una struttura di dati non sequenziale che contiene elementi non ripetuti.

Gli elementi di un insieme non hanno un ordine e la struttura è mutabile, permettendo di aggiungere e togliere elementi in base alla necessità.

In Python possiamo costruire un insieme con la funzione `set`.

```
>>> A = ["banane", "mele", "arance", "banane"] #Lista
>>> A = set(A) #Costruzione dell'insieme da una lista
>>> A
{"arance", "banane", "mele"}
```

Costruzione di insiemi

Possiamo costruire un insieme usando una coppia di parentesi graffe { } nello stesso modo con cui usiamo le parentesi quadre [] per la costruzione di una lista.

Una volta costruito l'insieme possiamo conoscerne la lunghezza con la funzione `len`.

```
>>> A = {"banane", "mele", "arance", "banane"} #Costruzione di un insieme
>>> len(A)                                     #Numero di elementi
3
>>> len(A) > 0                                #L'insieme contiene elementi?
True
>>> bool(A)                                    #Equivalente al precedente
True
>>> if A:                                       #Funziona anche con l'if
>>>     print("L'insieme non è vuoto")
>>> else:
>>>     print("L'insieme è vuoto")
L'insieme non è vuoto
```

Accedere agli elementi di un insieme

Dato che un insieme non è una collezione sequenziale, non possiamo accedere agli elementi tramite un indice, ma possiamo controllare la loro esistenza con l'operatore `in`:

```
>>> A = {"banane", "mele", "arance", "banane"} #Costruzione di un insieme
>>> "mele" in A
True
```

Oppure scorrere la collezione con un ciclo `for`:

```
>>> A = {"banane", "mele", "arance", "banane"}
>>> for a in A:
>>>     print(a)
mele
arance
banane
```

Metodi per gli insiemi

- Aggiungere un elemento:
- Aggiungere più elementi:
- Rimuovere un elemento (l'elemento deve essere contenuto, altrimenti viene sollevata una eccezione che deve essere gestita con try...except...):
- Rimuovere un elemento (può anche non essere contenuto):

```
>>> A = {1, 2, 3}
>>> A.add(4)
>>> A
{1, 2, 3, 4}
```

```
>>> A = {1, 2, 3}
>>> A.update([4, 5, 6])
>>> A
{1, 2, 3, 4, 5, 6}
```

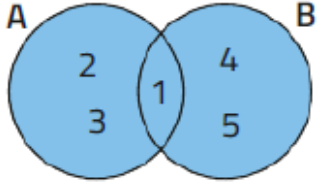
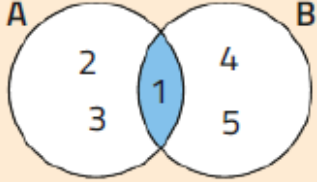
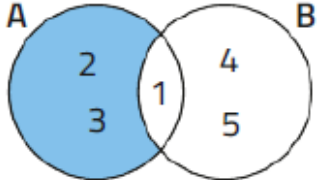
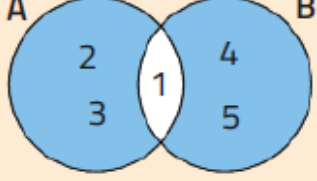
```
>>> A = {1, 2, 3}
>>> A.remove(2)
>>> A
{1, 3}
```

```
>>> A = {1, 2, 3}
>>> A.discard(2)
>>> A
{1, 3}

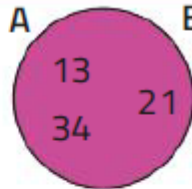
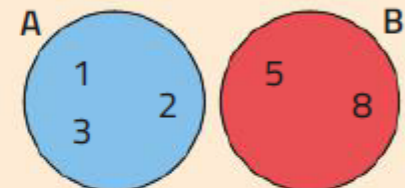
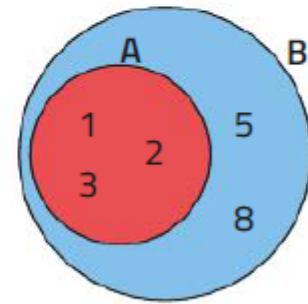
>>> A.discard(5)
>>> A
{1, 3}
```

Operazioni tra insiemi

Dati due insiemi $A = \{1, 2, 3\}$ e $B = \{1, 4, 5\}$ possiamo eseguire diverse operazioni di comparazione degli elementi:

Operazione	Codice	Diagramma di Eulero-Venn
Unione di A e B	<pre>>>> A.union(B) #Metodo {1, 2, 3, 4, 5} >>> A B #Operatore {1, 2, 3, 4, 5}</pre>	
Intersezione di A e B	<pre>>>> A.intersection(B) {1} >>> A & B {1}</pre>	
Differenza di A e B	<pre>>>> A.difference(B) {2, 3} >>> A - B {2, 3}</pre>	
Differenza simmetrica di A e B	<pre>>>> A.symmetric_difference(B) {2, 3, 4, 5} >>> A ^ B {2, 3, 4, 5}</pre>	

Relazioni tra insiemi

Operazione	Codice	Diagramma di Eulero-Venn
Coincidenza	<pre> >>> A = {13, 21, 34} >>> B = {13, 21, 34} >>> A == B True </pre>	
Disgiunzione	<pre> >>> A = {1, 2, 3} >>> B = {5, 8} >>> A.isdisjoint(B) True </pre>	
Sottoinsieme e sovrainsieme	<pre> >>> A = {1, 2, 3} >>> B = {1, 2, 3, 5, 8} >>> A.issubset(B) #Metodo True >>> A <= B #Operatore True >>> A.issuperset(B) #Metodo False >>> A >= B #Operatore False </pre>	

Dizionari

Un **dizionario** è una struttura dati formata da coppie chiave-valore in cui ogni chiave identifica univocamente il valore associato.

Possiamo costruire un dizionario con la funzione `dict`.

```
>>> sconto = dict([                                #Sconto per ogni categoria
>>>     ("magliette", 0.20),                        #Coppia chiave - valore,
>>>     ("scarpe", 0.10)                           #come una tupla di due elementi
>>> ])
>>> sconto
{'magliette': 0.2, 'scarpe': 0.1}
```

La rappresentazione più efficace di un dizionario è quella di una tabella a due colonne:

l'interprete Python cerca nella prima colonna la chiave e restituisce il valore della riga corrispondente.

Chiave	Valore
magliette	0.20
scarpe	0.10

Costruzione di un dizionario

Per la costruzione di un dizionario utilizziamo le parentesi graffe, come per gli insiemi, ma separiamo chiave e valore con i due punti.

```
>>> sconto = {                                     #Sconto per ogni categoria.
>>>     "magliette": 0.20,                         #Coppia chiave - valore,
>>>     "scarpe": 0.10                             #separate dai due punti.
>>> }
>>> sconto
{'magliette': 0.2, 'scarpe': 0.1}
```

Bisogna fare attenzione al caso del **dizionario vuoto**, in cui la costruzione di un dizionario può sembrare un insieme vuoto (che si crea invece con **set**):

```
>>> { }                                             #Costruzione di un insieme
>>> { }                                             #Insieme vuoto?
>>> type({ })                                     #Controlliamo con type ...
dict                                              #No! Dizionario vuoto!
```

Accedere agli elementi del dizionario

Per accedere a un elemento è sufficiente utilizzare la sua chiave (se la chiave non è presente all'interno del dizionario, allora viene sollevata un'eccezione):

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> sconto["magliette"]
0.20
```

L'operatore `in` ci permette di controllare se la chiave è presente nel dizionario prima di tentare l'accesso:

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> chiave = "cappelli"
>>> if chiave in sconto:
>>>     print(sconto[chave])
>>> else:
>>>     print(f"Errore! la chiave '{chave}' non esiste!")
Errore! La chiave 'cappelli' non esiste!
```

Aggiungere/rimuovere elementi al dizionario

Se una chiave non esiste, è possibile aggiungerla al dizionario e assegnarle un valore:

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> sconto["cappelli"] = 0.15
>>> sconto
{'magliette': 0.2, 'scarpe': 0.1, 'cappelli': 0.15}
```

è anche possibile aggiornare il valore di una chiave già esistente:

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> sconto["magliette"] = 0.15
>>> sconto
{'magliette': 0.15, 'scarpe': 0.1}
```

la rimozione di un elemento da un dizionario utilizza la keyword `del`, come per le liste:

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> del sconto["magliette"]
>>> sconto
{'scarpe': 0.1}
```

Iterazioni sui dizionari

Iterazione per...	Codice
... chiave-valore	<pre>>>> for (chiave, valore) in sconto.items() >>> print(f"{chiave} -> {valore}") magliette -> 0.2 scarpe -> 0.1</pre>
... chiavi	<pre>>>> for chiave in sconto.keys() #Il metodo è opzionale >>> print(chiave) magliette scarpe</pre>
... valori	<pre>>>> for valore in sconto.values() >>> print(valore) 0.2 0.1</pre>

Metodi per i dizionari

- Accedere ai valori di un dizionario: metodo `get`

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> sconto.get("cappelli")           #Se la chiave non esiste, non restituisce nulla
```

- Aggiornare un dizionario con un altro, sovrascrivendo i valori delle chiavi esistenti o aggiungendo quelle mancanti: metodo `update`

```
>>> sconto = {"magliette": 0.20, "scarpe": 0.10}
>>> sconto.update({"collane": 0.15, "magliette": 0.60})
>>> sconto
{'magliette': 0.6, 'scarpe': 0.1, 'collane': 0.15}
```

- Rimuovere elementi dal dizionario:
 - Metodo `pop`: occorre specificare le chiavi dell'elemento da rimuovere
 - Metodo `clear`: rimuove tutti gli elementi da un dizionario

Archivi o file

Un **file**, o **archivio**, è una collezione di informazioni salvata in modo persistente in un computer.

Un file è identificato univocamente da una stringa, detta *percorso*, che è composta da tre parti:

1. la cartella che contiene il file;
2. il nome del file;
3. l'estensione del file.

C:\Users\nome_utente\nome_file.txt

Estensione	Descrizione
.txt	File che contiene testo libero e stringhe, senza una struttura ben definita.
.csv	<i>Comma-Separated Values</i> , file di testo che contiene informazioni strutturate, simili a tabelle, in cui le celle sono separate da virgole.

Operare con i file

Dato che un file è salvato su un tipo di memoria diversa da quella su cui eseguiamo il programma, **è necessario trasferire i dati dal file al programma e viceversa.**

Per fare questo ci serviremo di alcuni punti di riferimento che ci indicano lo stato del trasferimento.

Riferimento	Descrizione
Puntatore	Rappresenta la posizione corrente nel file.
Fine della riga - End Of Line (EOL)	Segnala la fine della riga corrente, rappresentata dalla stringa <code>\n</code> , dove «n» sta per <i>newline</i> , «nuova riga».
Fine del file - End Of File (EOF)	Segnala la fine del file, rappresentata dalla stringa vuota <code>""</code> .

I file hanno un inizio e una fine, e quando apriamo un file per leggere dei dati e caricarli nel programma, Python restituisce un *puntatore* all'inizio del file, che può essere utilizzato per scorrerlo fino alla fine. Se il file contiene del testo, allora ogni riga di testo è segnata da un *fine riga*, che può essere particolarmente utile se vogliamo caricare le informazioni riga per riga.

Operazioni sui file

Oltre a **leggere** un file, possiamo anche **scrivere** dei dati su un file. Python supporta due modalità di scrittura:

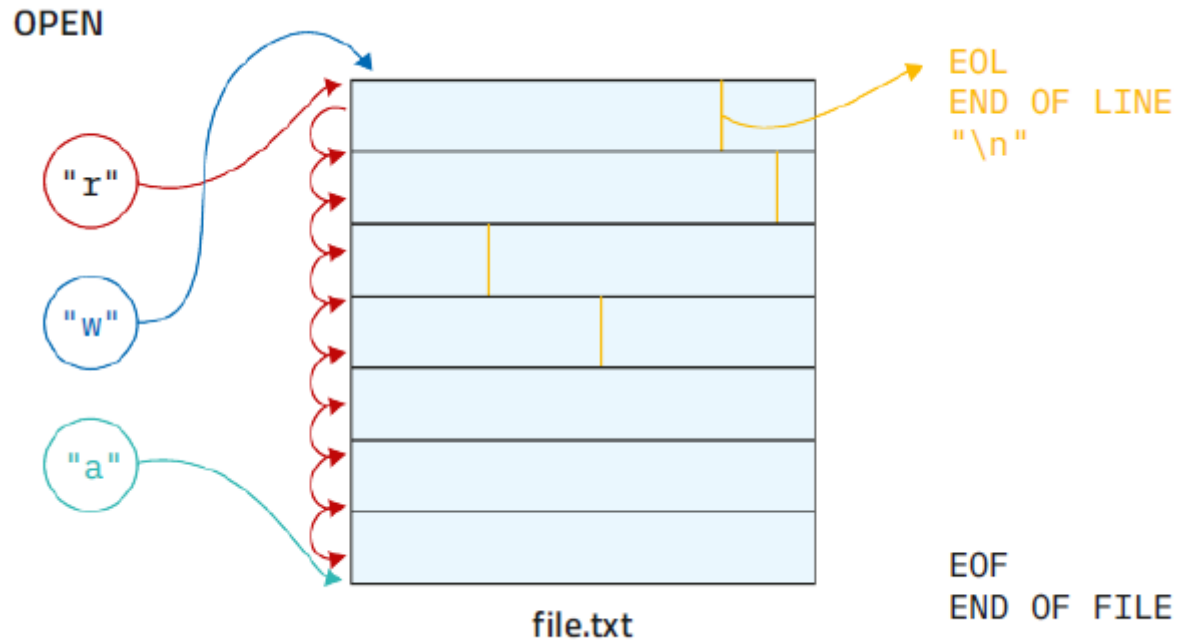
- *aggiunta*, con cui possiamo inserire i dati alla fine del file;
- *scrittura*, con cui sovrascriviamo un file esistente con dei nuovi dati.

Modalità di apertura	Descrizione
"r" (read)	Apre in sola lettura. Se il file non esiste, solleva un'eccezione.
"a" (append)	Apre in «aggiunta». Se il file non esiste, lo crea.
"w" (write)	Apre in scrittura. Se il file non esiste, lo crea. Se il file esiste, lo sovrascrive.

Python fornisce la funzione **open** che, dati il percorso del file e la modalità di accesso, restituisce il puntatore al file con cui effettuare le operazioni di lettura e/o scrittura. Una volta terminate le operazioni, possiamo chiudere il file invocando il metodo **close** sul puntatore.

```
1 | p = open("file.txt", "r")      #Apertura di un file in modalità lettura
2 | ...                            #Operazioni sui file
3 | p.close()                     #Chiusura del file
```

Modalità di accesso a un file



Lettura di un file una riga alla volta

Per leggere un file possiamo caricare una riga alla volta con il ciclo `while` e il metodo `readline`:

```
1 p = open("file.txt", "r")      # Apertura del file in lettura
2 riga = p.readline()            # Lettura della prima riga
3 while riga != "":              # Riga vuota? Fine del file?
4     print(riga)                 # Visualizzazione della riga
5     riga = p.readline()         # Lettura riga successiva
6 p.close()                       # Chiusura del file
```

Mentre il ciclo `while` ci dà il controllo sulla lettura del file, il ciclo `for` automatizza il processo di lettura e controllo della fine della riga.

```
1 p = open("file.txt", "r")      #Apertura del file in lettura
2 for riga in p:                 #Iterazione sulle righe
3     print(riga)                 #Visualizzazione della riga
4 p.close()                       #Chiusura del file
```

Lettura di un file più righe insieme

È possibile leggere un file e salvarne il contenuto in una lista d'appoggio con il metodo `readlines`:

```
1 p = open("file.txt", "r")      #Apertura del file in lettura
2 testo = p.readlines()          #Lettura dell'intero file
3 print(testo)                   #Visualizzazione del testo
4 p.close()                      #Chiusura del file
```

Scrittura di un file

Possiamo anche scrivere il contenuto di una lista in un file

```
1  #Testo da scrivere, con EOL per ogni riga
2  testo = [
3      "Nel mezzo del cammin di nostra vita\n",
4      "mi ritrovai per una selva oscura,\n",
5      "ché la diritta via era smarrita.\n"
6  ]
7  p = open("inferno.txt", "w")      #Apertura del file in scrittura
8  for riga in testo:                #Iterazione sulle righe del testo
9      p.write(riga)                  #Scrittura delle righe
10 p.close()                          #Chiusura del file
```

Aprire un file in scrittura ha due effetti collaterali che dobbiamo sempre tenere presenti:

- Se il file non esiste, allora verrà creato un nuovo file vuoto; lo stesso succede se apriamo il file in aggiunta.
- Se il file esiste e contiene dei dati, allora verranno sovrascritti con i nuovi dati.

File strutturati - record

I file CSV (*Comma-Separated Values*) sono file di testo strutturati in modo che ogni riga sia composta da un record.

Un record rappresenta una sequenza di dati (per esempio nome, descrizione, prezzo) separati da una virgola.

Per **leggere** i record contenuti in un file possiamo procedere:

```
1      p = open("magazzino.csv", "r")      #Apertura del file in lettura
2      for riga in p:                        #Iterazione sui record
3          riga = riga.strip("\n")          #Rimozione EOL
4          riga = riga.split(",")           #Separazione dei campi
5          print(riga)                      #Visualizzazione campi separati
6      p.close()                            #Chiusura del file

['nome', 'descrizione', 'prezzo']
['maglietta', 't-shirt di colore blu', '8.50€']
['cappello', 'cappellino di lana', '7.00€']
['jeans', 'jeans slim fit', '11.95€']
```

File strutturati - record

Per **scrivere** i record in un file possiamo procedere:

```
1      #Rappresentazione dei prodotti da aggiungere
2      prodotti = [
3          ["scarpe", "mocassini marroni", "68.50€"],
4          ["cintura", "cintura di pelle", "37.60€"]
5      ]
6      p = open("magazzino.csv", "a")      #Apertura magazzino in aggiunta
7      for prodotto in prodotti:          #Iterazione sui prodotti
8          prodotto = ",".join(prodotto)   #Unione delle stringhe con la virgola
9          prodotto = prodotto + "\n"      #Aggiunta EOL
10         p.write(prodotto)               #Scrittura su file
11     p.close()                           #Chiusura del file
```

