LINGUAGGIO PYTHON

TIPI STRUTTURATI

SISTEMI INFORMATIVI E DBMS

CORSO DI LAUREA MAGISTRALE IN
MANAGEMENT E MONITORAGGIO DEL TURISMO SOSTENIBILE

PROF. ANDREA PINNA

AA 2020/2021

TIPI DI DATO STRUTTURATI

I tipi di dato strutturati sono tipi di dato che organizzano un insieme di dati in una certa modalità. I dati strutturati possono essere scomposti in dati semplici o primitivi (int, float, bool e str). Ciascun tipo di dato strutturato è caratterizzato dalla sua modalità di accesso e di modifica dei singoli valori registrati.

I più importanti tipi strutturati di Python sono:

- Liste
- Tuple
- Dizionari

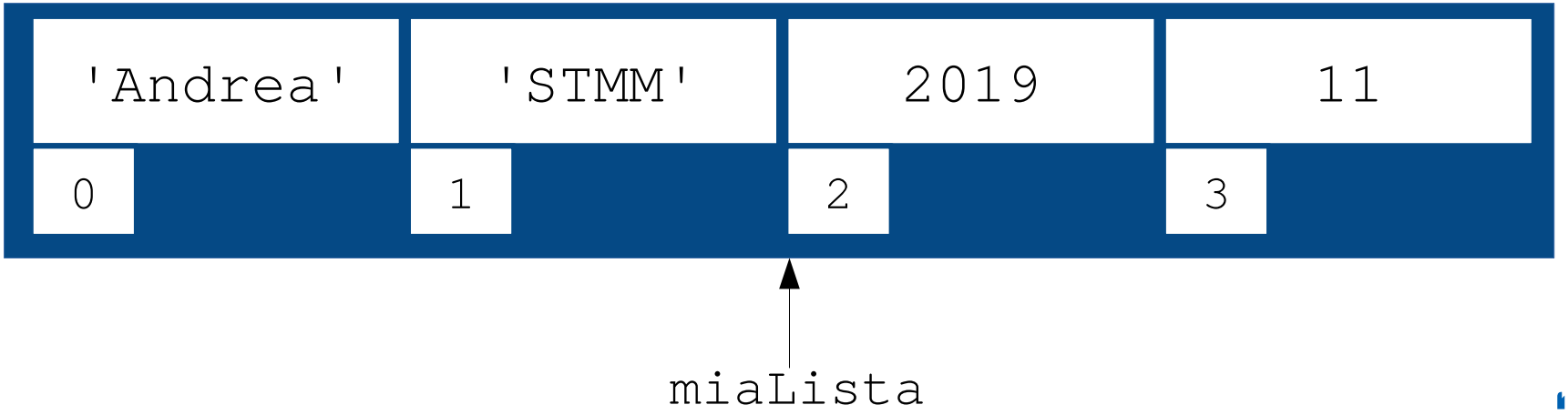
Vedremo anche un tipo di dato aggiuntivo

- Numeri complessi

LISTE

Una `list` è una sequenza di valori di qualsiasi tipo separati da virgola e racchiusi tra parentesi quadre. Ogni valore è collocato in una casella di memoria alla quale è associato un indice. Il primo elemento ha indice zero.

```
miaLista = ['Andrea', 'STMM', 2019, 10+1]
```



LISTE

Dichiarazione e assegnamento: avviene per riferimento

```
#assegna a miaLista il riferimento alla seguente
lista di valori:

miaLista = ['Andrea', 'MMTS', 2019, 11]
```

Dichiarazione lista vuota:

```
>>> miaLista2 = []
>>> type(miaLista2)
<class 'list'>
```

LISTE

Accesso al singolo valore: per ottenere uno dei valori devo specificare l'indice tra parentesi quadre dopo il nome della variabile.

```
#dammi il valore di miaLista di indice 2.
anno =  miaLista[2]
```

Oppure posso specificare la posizione partendo dalla fine della lista utilizzando l'indice negativo. L'indice negativo parte da -1.

```
numero = miaLista[-1] #Ultimo elemento della lista
```

LISTE

Modifica del valore di uno degli elementi: devo specificare l'indice.

```
#assegna 2020 alla casella di indice 2 di
miaLista
miaLista[2]=2020
```

Attenzione: per effettuare la lettura o la modifica di un elemento della lista in base ad un indice, deve esistere un elemento con quel indice. Se l'indice non esiste, l'interprete emette un errore di indice.

LISTE

Sotto Lista : Posso ottenere una porzione degli elementi della lista specificando l'indice di partenza e l'indice di arrivo sommato di uno.

```
#sotto lista dall'indice 2 all'indice 3
sottoLista = miaLista[2:4]
print(sottoLista)
[2019, 11]
```

Nota: `miaLista[2:]` parte da due e arriva alla fine.

LISTE

Eliminazione di un elemento dalla lista: per eliminare un elemento dalla lista posso usare la parola chiave **del**.

```
#elimina l'elemento di indice 2
del miaLista[2]
print(miaLista)
['Andrea', 'MMTS', 11]
```

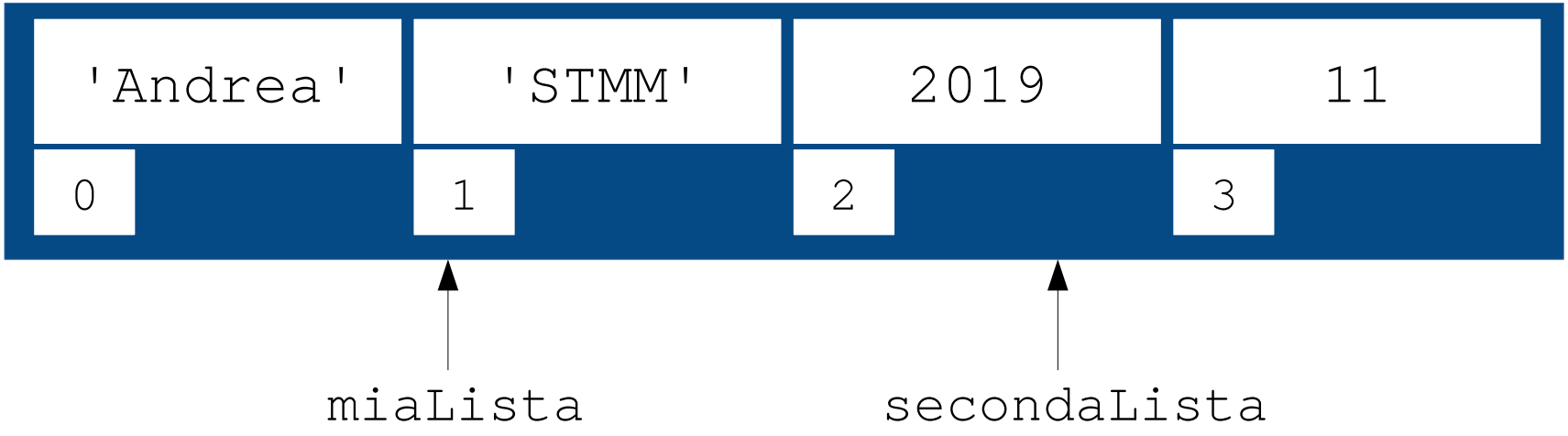
Gli elementi con indice superiore a 2 vengono spostati indietro di una posizione

RIFERIMENTO A LISTE

Una `list` è collocata in memoria. Posso creare più riferimenti alla stessa lista. Basta assegnare ad una variabile una lista già collocata. Esempio:

```
secondaLista = miaLista
```

Ora i due nomi riferiscono alla stessa lista.



RIFERIMENTO A LISTE

Le modifiche fatte agli elementi della lista eseguiti usando uno dei nomi modificano la casella di memoria. Quindi tutti i riferimenti vedranno la lista modificata.
Esempio:

```
>>> lista1 = [1,4]
>>> lista2 = lista1
>>> print(lista2)
[1, 4]
>>> lista1[1]=0  #Modifico l'elemento 1 usando lista1
>>> print(lista2)  #La modifica la vede anche lista2
[1, 0]
```

RIFERIMENTO A LISTE

Il riferimento si perde se eseguo un nuovo assegnamento

```
>>> lista1 = [1,4]
>>> lista2 = lista1
>>> lista1 = ['nuova lista',10]
>>> print(lista2)
[1,4]
>>> print(lista1)
['nuova lista',10]
```

L'assegnamento non modifica i singoli elementi ma crea una nuova lista.

OPERAZIONI CON LISTE

Le liste possono essere parte di espressioni. Tra le operazioni più comuni si hanno:

		Restituisce:
Calcolo lunghezza:	`len([2, 12, 3])`	`3 #int`
Appartenenza:	`3 in [1, 2, 3]`	`True #bool`
Concatenazione:	`[1, 4] + [4, 5]`	`[1, 4, 4, 5]`
Ripetizione:	`['Hi!'] * 2`	`['Hi!','Hi!']`

OPERAZIONI CON LISTE

Conversione a lista:
La funzione di built-in `list()` converte un iterabile in una lista, separando i suoi elementi.

```
inLista = list('ciao')
 ['c', 'i', 'a', 'o']

inLista = list(range(0,3))
 [0,1,2]

inLista = list(fileApertoInLettura)
```

OPERAZIONI CON LISTE

Esempio di codice:

```
miaLista=[3,4,'ciao']
if 3 in miaLista:
   miaLista = miaLista+['contiene 3']

print(len(miaLista))

for x in miaLista:
   print(x)
```

ESERCIZIO LISTE E FILE

Aprire il file Leo.txt in modalità lettura, convertire il contenuto in una lista e stampare a video il numero di righe.

Nella lista, sostituire la riga di indice 5 con la frase "Questa riga è stata rimossa"

Stampare a video il contenuto della lista.
Scrivere sul file "censura.txt" il contenuto della lista.

TUPLE

Le “tuple” sono un tipo di dato strutturato indicizzato e immutabile. Una tupla è una sequenza di dati separati da virgola.

Esempio:

```
>>> miaTupla='esempio',2,3.14159,True
>>> miaTupla
('esempio', 2, 3.14159, True)
>>> type(miaTupla)
<class 'tuple'>
>>>
```

TUPLE

È anche ammesso l'uso di parentesi tonde.

Esempio:

```
>>> miaTupla=('esempio',2,3.14159,True)
>>> miaTupla
('esempio', 2, 3.14159, True)
```

TUPLE

Si accede agli elementi delle tuple allo stesso modo delle liste,
Esempio:

```
unValore = miaTupla[1]

primi2Elementi = miaTupla[:2]

ultimi2Elementi = miaTupla[-2:]
```

TUPLE

La tupla è “immutable”. Ciò significa che non posso assegnare un nuovo valore ad un elemento della tupla.

```
>>> miaTupla[1] = 239
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

TUPLE

Se si ha necessità di manipolare i dati della tupla, posso convertire la tupla in una lista e viceversa.

```
>>> miaLista = list(miaTupla)
>>> miaLista+=['aggiunta']
>>> miaTupla = tuple(miaLista)
>>> print(miaTupla)
('esempio', 2, 3.1415, True, 'aggiunta')
```

TUPLE E ASSEGNAMENTO

Le tuple e le liste permettono l'assegnamento multiplo.
Posso assegnare una lista di valori a un elenco di variabili.

Sintassi:

```
miaTupla = ('dato1', 'dato2')
var1, var2 = miaTupla
```

Equivale a:

```
var1= miaTupla[0]
var2= miaTupla[1]
```

Il numero di elementi della tupla o della lista deve essere uguale al numero di variabili.

DIZIONARI

Il tipo dictionary `(dict)` è un tipo strutturato che permette di conservare un insieme di coppie di dati secondo lo schema "chiave-valore". I valori possono essere di qualsiasi tipo. Le chiavi non possono essere tipi mutable (quindi le liste sono vietate)

Sintassi: L'insieme delle coppie è racchiuso tra parentesi graffe. Ogni coppia è definita con due dati separati dal simbolo "**:**". Il dato a sinistra dei due punti è detto chiave. Il dato a destra è detto valore.

Esempio:

```
aDict = {'key1':'val1', 12:3.17, (1,3):'ciao'}
```

DIZIONARI

Posso conoscere tutte le chavi del dizionario utilizzando il metodo `keys`. Posso conoscere tutte i valori usando il metodo `values`.

Esempio:

```
>>> aDict = {'key1':'val1', 12:3.17, (1,3):'ciao'}
>>> aDict.keys()
dict_keys(['key1', 12, (1, 3)])
>>> aDict.values()
dict_values(['val1', 3.17, 'ciao'])
```

Nota: il risultato di questi metodi non è una lista ma si può convertite con il casting.

DIZIONARI

Esiste inoltre il metodo `items` che permette di ottenere l'insieme di coppie, sotto forma di un elenco di tuple.

```
>>> aDict.items()
dict_items([('key1', 'val1'), (12, 3.17), ((1, 3), 'ciao')])
>>> listaDaDizionario=list(aDict.items())
>>> listaDaDizionario
[('key1', 'val1'), (12, 3.17), ((1, 3), 'ciao')]
>>> type(listaDaDizionario[0])
<class 'tuple'>
```

DIZIONARI: ACCESSO AI DATI

Per accedere ai dati del dizionario si utilizza una sintassi molto simile a quella delle tuple e delle liste. Ma tra parentesi quadre al posto dell'indice si inserisce la chiave.

```
>>> aDict[12]
3.17
>>> aDict['key1']
'val1'
>>> aDict[(1,3)]
'ciao'
>>> aVar = aDict[(1,3)]
```

DIZIONARI: MODIFICA

Si può modificare il dato associato ad una chiave tramite un assegnamento in cui si utilizza la chiave come indice.

```
>>> aDict[12] = 0
```

Posso in ogni momento aggiungere una coppia chiave-valore al dizionario tramite un assegnamento chiave-valore:

```
>>> aDict['keyNew'] = [1, 4, 5]
```

DIZIONARI: USO TIPICO

Abbiamo visto che non esistono regole sull'uso delle chiavi e dei valori. È però preferibile che un dizionario sia definito da chiavi dello stesso tipo e da valori tra essi coerenti.

Ad esempio:

```
cognomiAnno = {'Da Vinci':1452, 'Raffaello':1483, 'Della Francesca':1416}

matricolaEsami = {3001:['SIDBMS', 'SST', 'ING'], 3012:['STDA', 'SST']}
```

IN - TUPLE LIST E DICT

La parola chiave **in** permette di sapere se un dato è presente in una tupla, in una lista o tra le chiavi di un dizionario.

Esempio:

```
>>> listaEsempio = [41, 5.0, 9.9]
>>> 5 in listaEsempio
True
>>> 10 in listaEsempio
False
```

LEN - TUPLE LIST STRING E DICT

La funzione di built-in `len` prende come argomento un tipo di dato strutturato e restituisce il numero di elementi. Se come argomento ha una lista restituisce il numero di caratteri.

Esempio:

```
>>> listaEsempio = [41, 5.0, 9.9]
>>> len(listaEsempio)
3
>>> len("ciao")
4
```

NUMERI COMPLESSI

Questo tipo di dato memorizza una sola coppia di valori chiamati `real` (parte reale) e `imag` (parte immaginaria).

```
>>> c = complex(1,2.0) #definizione, metodo 1
>>> c.real  #accesso alla parte real (sola lettura)
1
>>> c.imag  #accesso alla parte imag (sola lettura)
2.0
>>> print(c)
(1+2j)
>>> d = (3+4j) #definizione, metodo 2
>>> c + d #consente le operazioni
(4+6j)
```