



Pattern Recognition
and Applications Lab

Le basi della Programmazione con il linguaggio Python

Docente: Ambra Demontis

Anno Accademico: 2023 - 2024

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica



University of Cagliari,
Italy

Department of Electrical and
Electronic Engineering



Le Basi di Python

In queste slide vedremo:

- Le strutture dati: tupla, lista, dizionario
- Il ciclo for
- Definizione di funzioni

Memorizzare Sequenze di Dati

In Python, un insieme di dati può essere **memorizzato sequenzialmente** utilizzando la struttura dati **tupla** o la struttura dati **lista**.

Si può creare una tupla scrivendo gli elementi da memorizzare separati da una virgola e racchiudendoli tra **parentesi tonde**.

Esempio: `voti = (27, 28, 26, 30)`

Questa istruzione crea la tupla `voti` e la assegna alla variabile `voti`.

Per creare una lista invece, gli elementi vengono racchiusi tra parentesi quadre.

Esempio: `voti = [27, 28, 26, 30]`

Memorizzare Sequenze di Dati

NB: Gli elementi non devono necessariamente essere tutti dello stesso tipo!

Ad esempio, entrambe queste strutture dati possono contenere sia dati di tipo intero che dati di tipo frazionario.

Esempi:

voti = (27, 28.5, 26, 30)

voti = [27, 28.5, 26, 30]

Tuple e Liste Vuote

Una tupla vuota viene rappresentata con le parentesi tonde vuote.

Esempio:

```
voti = ()
```

Una lista vuota viene rappresentata con le parentesi quadre vuote.

```
voti = []
```

Le Strutture Dati Tupla e Lista - l'Operatore +

Ad entrambe può essere applicato l'operatore di **concatenazione** + per concatenarle ad una struttura dati dello **stesso tipo**.

Esempi:

voti = (27, 28.5, 26, 30) + (25, 24)

voti = [27, 28.5, 26, 30] + [25, 24]

Verrà creata una nuova tupla o una nuova lista che contiene tutti gli elementi presenti nella prima e nella seconda struttura dati.

Le Strutture Dati Tupla e Lista - l'Operatore +

NB: L'operatore di concatenazione **non può concatenare** una struttura dati di un tipo ad una struttura dati di **tipo differente**.

Esempio:

voti = (27, 28.5, 26, 30) + [25, 24]

Solleverà un'eccezione.

Le Strutture Dati Tupla e Lista - la Funzione Print

La funzione `print` può essere utilizzata per stampare a schermo entrambe.

Esempi:

```
voti = (27, 28.5, 26, 30) + (25, 24)
```

```
print(voti)
```

Stamperà a schermo: (27, 28.5, 26, 30, 25, 24)

```
voti = [27, 28.5, 26, 30] + [25, 24]
```

```
print(voti)
```

Stamperà a schermo: [27, 28.5, 26, 30, 25, 24]

Le Strutture Dati Tupla e Lista - la Funzione Len

La funzione `len` restituisce il **numero di elementi** che compongono una tupla o una lista.

Esempi:

```
voti = [28, 29, 22, 25]  
print(len(voti))
```

```
voti = (28, 29, 22, 25)  
print(len(voti))
```

In entrambi i casi verrà stampato a schermo: 4

Le Strutture Dati Tupla e Lista - Selezione di Elementi

Si può **selezionare un singolo elemento** della **tupla** o della **lista**, indicando l'**indice dell'elemento tra parentesi quadre**.

In Python **l'indice del primo elemento è zero**.

Esempio

Elementi: 27, 28.5, 26, 30

Indici: 0, 1, 2, 3

Le Strutture Dati Tupla e Lista - Selezione di Elementi

Esempi:

```
voti = (27, 28.5, 26, 30)
```

```
print(voti[1])
```

Stamperà a schermo: 28.5

```
voti = [27, 28.5, 26, 30]
```

```
print(voti[1])
```

Stamperà a schermo: 28.5

Le Strutture Dati Tupla e Lista - Selezione di Elementi

Si può anche selezionare elementi utilizzando **indici** calcolati prendendo come **riferimento la fine della lista (indici negativi)**.

Elementi: 27, 28.5, 26, 30

Indici: -4, -3, -2, -1

Esempi:

voti = (27, 28.5, 26, 30)

`print(voti[-1])`

Stamperà a schermo: 30

Le Strutture Dati Tupla e Lista - Selezione di Elementi

Si può **selezionare più elementi** di una **tupla o di una lista** utilizzando l'**operatore di slicing** (di taglio), il cui simbolo sono i due punti : .

La sintassi è:

<lista /tupla> [idx 1° numero da selezionare : idx 1° numero da non selezionare]

Esempio:

voti = (27, 28.5, 26, 30)

print(voti[1:3])

Stamperà a schermo: (28.5, 26)

Le Strutture Dati Tupla e Lista - Selezione di Elementi

La sintassi è:

<lista /tupla> [idx 1° numero da selezionare : idx 1° numero da non selezionare]

Se il primo numero che si vuole selezionare è il **primo elemento della lista** si può **omettere**.

Esempio:

voti = (27, 28.5, 26, 30)

print(voti[:3])

Stamperà a schermo: (27, 28.5, 26)

Le Strutture Dati Tupla e Lista - Selezione di Elementi

La sintassi è:

<lista /tupla> [idx 1° numero da selezionare : idx 1° numero da non selezionare]

Se l'ultimo elemento che si vuole selezionare è l'**ultimo elemento della tupla/lista** il numero dopo i due punti si può **omettere**.

Esempio:

voti = (27, 28.5, 26, 30)

print(voti[1:])

Stamperà a schermo: [28.5, 26, 30]

Le Strutture Dati Tupla e Lista - Selezione di Elementi

Anche con l'operatore di slicing si possono utilizzare indici negativi.

Esempio:

```
voti = (27, 28.5, 26, 30)
```

```
print(voti[-2:])
```

Stamperà la lista dall'elemento con indice -2 in poi, quindi: (26, 30)

Esempio:

```
voti = (27, 28.5, 26, 30)
```

```
print(voti[:-2])
```

Stamperà la lista fino all'elemento con indice -2, quindi: (27, 28.5)

Differenza tra Tuple e Liste

La differenza tra tuple e liste è che **gli elementi delle liste possono essere modificati, mentre quelli delle tuple no.**

Per questo le liste vengono dette **mutabili**, mentre le tuple **immutabili**.

Esempi:

```
voti = [27, 28.5, 26, 30]
```

```
voti[0] = 28
```

```
print(voti)
```

Stamperà a schermo: [28, 28.5, 26, 30]

Mentre se si cerca di modificare un elemento di una tupla l'interprete solleverà un'eccezione.

Differenza tra Tuple e Liste

NB. Esiste una differenza nel caso in cui vi sia un solo elemento.

Una lista composta da un solo elemento si rappresenta con l'elemento racchiuso tra quadre.

voti = [3]

Una tupla composta da un singolo elemento si rappresenta con l'elemento **seguito da una virgola** racchiuso tra tonde.

voti = (3,)

Differenza tra Tuple e Liste

Cosa succede **se dimentichiamo la virgola?**

Python non vede quell'elemento come una tupla.

(Ignora le parentesi tonde se la virgola non è presente).

Esempio:

```
voti = (3,)
```

```
print(voti)
```

Stampa: (3,)

```
voti = (3)
```

```
print(voti)
```

Stampa: 3 alla variabile voti è stato assegnato l'intero 3 **e non una tupla.**

Esercizi

Esercizio sull'Uso delle Liste

Create un programma che:

- (1) Assegna ad una variabile chiamata numeri la seguente lista: [-3, 2, 7, -5]
- (2) Trasforma tutti in numeri negativi presenti nella lista in positivi.
- (3) Stampa a schermo la lista trasformata.

Esercizio sull'Uso delle Liste

Soluzione:

```
numeri = [-3, 2, 7, -5 ]  
num_numeri = len(numeri)  
i = 0  
while i < num_numeri:  
    if numeri[i] < 0:  
        numeri[i] = numeri[i] * -1  
    i = i + 1  
  
print(numeri)
```

Esercizi sull'Uso delle Tuple

Scrivete un programma che acquisisca in input una sequenza di numeri e li memorizzi in una tupla. (I numeri dovranno essere acquisiti uno alla volta).

Esercizio sull'Uso delle Tuple

Soluzione:

```
n = int(input("Quanti numeri vuoi inserire? "))
tupla= ()
k = 0
while k < n:
    elemento = float(input("Inserisci un valore: "))
    tupla = tupla + (elemento,)
    print(tupla)
    k = k + 1

print("La tupla creata è:", tupla)
```


Esercizi sull'Uso delle Liste

Scrivere un programma che:

(1) Acquisisce una sequenza di numeri e li memorizza in una lista.

(2) Verifica se la lista è ordinata in modo crescente.

Es. [1, 3, 5, 7] è ordinata in modo crescente, [1, 8, 5, 7] no.

(3) Stampa a schermo un messaggio che comunica all'utente se la lista è o no ordinata in modo crescente.

Esercizio sull'Uso delle Liste

Soluzione parte 1 (acquisizione lista):

```
n = int(input("Quanti numeri vuoi inserire? "))
```

```
lista = []
```

```
k = 0
```

```
while k < n:
```

```
    elemento = float(input("Inserisci un valore: "))
```

```
    lista = lista + [elemento]
```

```
    k = k + 1
```

```
...
```

Esercizio sull'Uso delle Liste

Soluzione parte 2 (controllo ordinamento):

```
ordinata_in_modo_crescente = True
```

```
i = 0
```

```
while (i < len(lista) - 1) and ordinata_in_modo_crescente :
```

```
    if lista[i] > lista[i+1] :
```

```
        ordinata_in_modo_crescente = False
```

```
    i = i + 1
```

```
if ordinata_in_modo_crescente:
```

```
    print("la lista è ordinata in modo crescente")
```

```
else:
```

```
    print("la lista non è ordinata in modo crescente")
```

Note sulle Liste

Se **assegniamo** ad una variabile **una porzione di una lista**, alla variabile viene assegnata una **copia** di quella porzione della lista.

Esempio:

```
voti = [27, 29, 30, 29]
```

```
primi_due_voti = voti[:2]
```

voti

27
29
30
29

primi_due_voti

27
29

Note sulle Liste

Se **assegniamo ad una variabile una porzione di una lista**, alla variabile viene assegnata una **copia** di quella porzione della lista.

Esempio:

```
voti = [27, 29, 30, 29]
```

```
primi_due_voti = voti[:2]
```

```
primi_due_voti[0] = 30 se modifichiamo un elemento della copia  
print(voti)
```

Stamperà [27, 29, 30, 29]

La lista originale (voti) non viene modificata.

Note sulle Liste

Se **assegniamo** ad una variabile **una porzione di una lista**, alla variabile viene assegnata una **copia** di quella porzione della lista.

Esempio:

```
voti = [27, 29, 30, 29]
```

```
primi_due_voti = voti[:2]
```

Se modifichiamo un elemento della copia la lista originale non viene modificata!

voti

27
29
30
29

primi_due_voti

30
29

Note sulle Liste

Se **assegniamo ad una variabile una lista intera** (e non solo una porzione di essa), alla variabile viene assegnato un riferimento alla lista originale.

Esempio:

```
voti = [27, 29, 30, 29]
```

```
voti_reference = voti
```

voti_reference

voti

27
29
30
29

Note sulle Liste

Esempio:

```
voti = [27, 29, 30, 29]  
voti_reference = voti  
voti_reference[0] = 30  
print(voti)
```

Stamperà [30, 29, 30, 29]

La lista originale (voti) è stata modificata.

Note sulle Liste

Se **assegniamo ad una variabile una lista intera** (e non solo una porzione di essa), alla variabile viene assegnato un riferimento alla lista originale.

Esempio:

```
voti = [27, 29, 30, 29]
```

```
voti_reference = voti
```

Se modifichiamo un elemento del riferimento alla lista la lista originale viene modificata!

voti_reference

voti

30

29

30

29

Note sulle Liste

Esempio:

```
voti = [27, 29, 30, 29]  
voti_reference = voti  
voti_reference[0] = 30  
print(voti)
```

Stamperà [30, 29, 30, 29]

La lista originale (voti) è stata modificata.

Note sulle Stringhe

Python vede le stringhe come **sequenze ordinate di caratteri**.

E' possibile selezionare una parte di una stringa utilizzare l'**operatore di selezione**.

Esempio:

```
nome_e_cognome = "Albert Einstein"
```

```
nome = nome_e_cognome[:7]
```

```
print(nome)
```

```
cognome = nome_e_cognome[-8:]
```

```
print(cognome)
```

Note sulle Stringhe

Come le tuple, le stringhe come tuple sono **immutabili** (non possiamo modificare parte di una stringa)

Esempio:

```
nome_e_cognome = "Stringa di esempio"
```

```
nome_e_cognome[-2] = "p"
```

L'interprete solleverà un' eccezione.

La Struttura Dati Dizionario

Si utilizzano per memorizzare un insieme **non omogeneo** e **non sequenziale** di dati.

Sono formati da coppie **chiave - valore** racchiuse tra parentesi graffe. La chiave indica cosa rappresenta il valore ad essa associato.

Sintassi:

```
{ <nome_chiave1>: <nome_valore1>,  
  <nome_chiave2>: <nome_valore2>,  
  ...  
}
```

La Struttura Dati Dizionario

Si utilizzano per memorizzare un insieme **non omogeneo** e **non sequenziale** di dati.

Sono formati da coppie **chiave - valore** racchiuse tra parentesi graffe.
La chiave indica cosa rappresenta il valore ad essa associato.

Esempio:

```
data = {  
    'giorno' : 10,  
    'mese' : 3,  
    'anno' : 2020  
}
```

La Struttura Dati Dizionario

Per **ottenere il valore corrispondente ad una chiave**, la sintassi è la seguente:

`<dizionario>[<nome_chiave>]`

Esempio:

```
data = {  
    'giorno' : 10,  
    'mese' : 3,  
    'anno' : 2020  
}  
print(data['giorno'])
```

Stamperà a schermo 10

La Struttura Dati Dizionario

I dizionari sono oggetti **mutabili**.

E' quindi possibile creare dizionari vuoti e **aggiungervi i dati** uno alla volta.

Un **dizionario vuoto** viene rappresentato con le sole parentesi graffe: {}

Per aggiungere una coppia chiave valore ad un dizionario, la sintassi è la seguente.

Sintassi:

<dizionario_esistente> [<nuova_chiave>] = <nuovo_valore>

La Struttura Dati Dizionario

E' possibile creare dizionari vuoti e **aggiungervi i dati** uno alla volta.

Esempio:

```
data = {}
```

```
data['giorno'] = 10
```

```
data['mese'] = 3
```

```
data['anno'] = 2020
```

```
print(data)
```

Stamperà {'giorno': 10, 'mese': 3, 'anno': 2020}

Esercizio sull'Uso dei Dizionari

Creare un programma che permetta ad uno studente di inserire i suoi dati anagrafici quali:

- nome
- cognome
- matricola
- anno di nascita
- anno di immatricolazione

e li memorizzi in un dizionario.

Il programma dovrà poi stampare a schermo il dizionario creato.

Esercizio sull'Uso dei Dizionari

Soluzione:

```
dati_anagrafici = {}
```

```
dati_anagrafici['nome'] = input("inserisci il tuo nome ")
```

```
dati_anagrafici['cognome'] = input("inserisci il tuo cognome ")
```

```
dati_anagrafici['matricola'] = input("inserisci la tua matricola ")
```

```
dati_anagrafici['anno_nascita'] = int(input("inserisci il tuo anno di nascita " ))
```

```
dati_anagrafici['anno_immatricolazione'] = int(input("inserisci il tuo anno di "  
"immatricolazione "))
```

```
print(dati_anagrafici)
```

Istruzioni Iterative - Ciclo For

In Python, l'istruzione iterativa **for** può essere usata per scorrere, uno ad uno, gli elementi di una struttura dati “**iterabile**”.

Le strutture dati iterabili che abbiamo visto sono: tuple, liste, dizionari e stringhe.

La sintassi è la seguente:

```
for <variabile> in <iterabile>:
```

```
...
```

Ad ogni iterazioni a <variabile> viene assegnato un elemento di <iterabile>.

Istruzioni Iterative - Ciclo For

Nel caso iterabile memorizzi i dati in maniera **sequenziale** (come fanno tuple, liste e stringhe), **la variabile assumerà il valore degli elementi di iterabile secondo l'ordine in cui sono memorizzati nella struttura dati.**

```
voti = [27, 25, 28]  
for voto in voti:  
    print(voto)
```

Stamperà:

27

25

28

Istruzioni Iterative - Ciclo For

Esempio:

```
nome = "Anna"
```

```
for lettera in nome:  
    print(lettera)
```

Stamperà:

A

n

n

a

Istruzioni Iterative - Ciclo For

Nel caso iterabile **non** memorizzi i dati in maniera **sequenziale** (come fanno i dizionari), **la variabile potrebbe assumere il valore degli elementi di iterabile in ordine casuale o nell'ordine di inserimento.**

Quando **iterabile è un dizionario**, la variabile assume il valore delle **chiavi** del dizionario.

Istruzioni Iterative - Ciclo For

In altri linguaggi il ciclo for si utilizza al posto del while per evitare di dover **definire e aggiornare un contatore**. In Python viene spesso fatto lo stesso.

L'istruzione iterativa for ha **necessità di ricevere un iterabile**.

Per crearlo si utilizza la funzione **range**.

Alla funzione range dobbiamo passare come argomento il **numero di iterazioni che vogliamo eseguire**.

Essa creerà un iterabile che ad ogni iterazione del ciclo restituisce un elemento che va da zero al numero di iterazioni scelto.

Istruzioni Iterative - Ciclo For

Esempio:

```
for numero in range(3):  
    print(numero)
```

Stamperà a schermo:

0

1

2

Esercizio sull'Uso dell'Istruzione Iterativa For

Dato il seguente dizionario contenente i dati di uno studente, utilizzare il ciclo for per stampare a schermo, per ogni coppia chiave-valore:

<chiave del dizionario> : <valore>

Dizionario dello studente:

```
studente = { "nome": "Sara", "cognome": "Bianchi", "anno_nascita": 1989 }
```

Esercizio sull'Uso dell'Istruzione Iterativa For

Soluzione:

```
studente = {"nome": "Sara", "cognome": "Bianchi", "anno_nascita": 1989}
```

```
for chiave in studente:
```

```
    print(chiave + ":" + str(studente[chiave]))
```

Definizione di Funzioni

La sintassi della definizione di una funzione in Python è la seguente:

```
def <nome funzione> (<nome parametro 1>, .. <nome_parametro n>):  
  
    <istruzione 1>  
    ...  
    <istruzione n>  
  
    return <valore ritorno>
```

Dove le istruzioni sono quelle che verranno eseguite quando la funzione viene chiamata e *<valore ritorno>* è il valore restituito dalla funzione.

Definizione di Funzioni

La sintassi della definizione di una funzione in Python è la seguente:

```
def <nome funzione> (<nome parametro 1>, .. <nome_parametro n>):
```

```
    <istruzione 1>
```

```
    ...
```

```
    <istruzione n>
```

```
    return <valore ritorno>
```

Notate che anche nel caso delle funzioni l'interprete capisce quali istruzioni fanno parte della definizione della funzione perchè sono scritte **rientrate** rispetto all'istruzione def.

Dove le istruzioni sono quelle che verranno eseguite quando la funzione viene chiamata e *<valore ritorno>* è il valore restituito in output dalla funzione.

Definizione di Funzioni

Definiamo una funzione che riceve una lista e restituisce il primo elemento.

Esempio:

```
def calcola_primo_elemento_lista(lista):  
    primo = lista[0]  
    return primo
```

Definizione di Funzioni

Definiamo una funzione che riceve una lista e restituisce il primo elemento.

Esempio:

```
def calcola_primo_elemento_lista(lista):  
    primo = lista[0]  
    return primo
```

Per verificarne il funzionamento proviamo a chiamarla:

```
primo_elem = calcola_primo_elemento_lista([3, 5, 2, 9, 8])  
print(primo_elem)
```

Stamperà a schermo: 3

Definizione di Funzioni

Se una funzione ha più parametri gli argomenti verranno associati ai parametri in base all'ordine nel quale vengono specificati nella chiamata.

Esempio:

```
def minore_di(num_1, num_2):  
    return num_1 < num_2
```

```
risultato = minore_di(2, 3)
```

(l'argomento 2 verrà assegnato al parametro num_1 e l'argomento 3 a num_2).

```
print(risultato)
```

Stamperà a schermo: True

Definizione di Funzioni - Valori di Default

In Python è possibile assegnare un **valore di default** ad alcuni parametri.

Sintassi:

```
def <nome funzione> (<nome param1>, <nome param 2> = <valore di default>):  
    ...
```

Esempio:

```
def somma(a, b=1):  
    return a + b
```

Definizione di Funzioni - Valori di Default

Esempio:

```
def somma(num_1, num_2=1):  
    return num_1 + num_2
```

Possiamo usare il valore di default non passando un argomento per quel parametro.

```
risultato = somma(3) passiamo solo l'argomento 3 che verrà assegnato a num_1  
print(risultato)  
Stamperà 4.
```

Definizione di Funzioni - Valori di Default

Esempio:

```
def somma(num_1, num_2=1):  
    return num_1 + num_2
```

Oppure, possiamo scegliere di passare un argomento anche per quel parametro.

```
risultato = somma(3, 2)  
print(risultato)  
Stamperà 5.
```

Definizione di Funzioni - Valori di Default

Non possono esservi parametri senza valore di default dopo parametri con un valore di default.

Ad esempio scrivendo:

```
def somma(num_1=1, num_2):  
    return num_1 + num_2
```

Verrà segnalato un errore di sintassi e il programma non potrà essere eseguito.

Definizione di Funzioni - Valori di Default

Nella chiamata di una funzione **i parametri con e senza valori di default possono essere specificati in disordine.**

Ad esempio creiamo una funzione con parametri di default che restituisce True se il primo numero è minore del secondo.

```
def minore_di(num_1=1, num_2=1):  
    return num_1 < num_2
```

```
risultato = minore_di(num_2=3, num_1=2)  
print(risultato)  
Stamperà True
```

Definizione di Funzioni - Valori di Default

Anche i parametri senza valori di default possono essere specificati in disordine ma nella chiamata dobbiamo specificare il nome del parametro al quale vogliamo associare l'argomento.

```
def minore_di(num_1, num_2):  
    return num_1 < num_2
```

```
risultato = minore_di(num_2=3, num_1=2)  
print(risultato)  
Stamperà True
```

Definizione di Funzioni - Valori di Default

Perchè è utile?

Perchè una funzione potrebbe avere tanti parametri con valori di default. Potendo specificarli in disordine, se ad esempio vogliamo usare i valori di default per tutti i parametri tranne l'ultimo, nella chiamata possiamo specificare solo l'ultimo.

Definizione di Funzioni - Valori di Default

Cosa si intende con “è utile specificarli in disordine”?

E' utile specificare gli argomenti seguendo lo stesso ordine dei parametri ma non specificarli tutti.

Si possono anche specificare semplicemente con un ordine differente?

Ni... l'inteprete riuscirà ugualmente ad interpretare il codice.

Tuttavia in questo modo il codice risulterà molto meno leggibile, soprattutto se il numero di parametri è veramente molto elevato.

Definizione di Funzioni

Le funzioni possono anche:

- non avere parametri

In questo caso la sintassi della definizione sarà:

`def <nome funzione> ():`

`...`

- non restituire nessun valore.

In questo caso si può:

- non scrivere l'istruzione `return` (metodo più usato)
- scrivere l'istruzione `return` senza alcun valore accanto:

`return`

Definizione di Funzioni

Un esempio di funzione che **non ha parametri** e **non restituisce alcun valore**.

Definizione:

```
def buongiorno():  
    print("Buongiorno utente!")
```

Quando chiameremo questa funzione ovviamente non le passeremo alcun argomento.

Chiamata:

```
buongiorno()
```

Stamperà a schermo: Buongiorno utente!

Definizione di Funzioni

Una funzione può anche “**apparentemente**” restituire più valori..

Ad esempio, analizziamo la seguente funzione che restituisce il primo e l'ultimo elemento di una lista.

```
def calcola_primo_e_ultimo_elem_lista(lista):  
    primo = lista[0]  
    ultimo = lista[-1]  
    return primo, ultimo
```

Sembrerebbe che restituisca due valori, quello memorizzato nella variabile *primo* e quello memorizzato nella variabile *ultimo*.

Definizione di Funzioni

```
def calcola_primo_e_ultimo_elem_lista(lista):  
    primo = lista[0]  
    ultimo = lista[-1]  
    return primo, ultimo
```

In realtà restituisce un solo valore: una tupla

In Python le tuple possono essere definite anche senza scrivere le parentesi tonde.

scrivere: primo, ultimo

è uguale a scrivere: (primo, ultimo)

Definizione di Funzioni

```
def calcola_primo_e_ultimo_elem_lista(lista):  
    ...  
    return primo, ultimo
```

Quando chiamiamo la funzione potremmo quindi assegnare l'output della funzione ad una singola variabile e poi “dividere” gli elementi della tupla.

Esempio:

```
lista = [1, 4, 3]  
primo_e_ultimo = calcola_primo_e_ultimo_elemento_lista(lista)  
primo = primo_e_ultimo[0]  
ultimo = primo_e_ultimo[1]
```

Definizione di Funzioni

Esempio:

```
lista = [1, 4, 3]
```

```
primo_e_ultimo = calcola_primo_e_ultimo_elemento_lista(lista)
```

```
primo = primo_e_ultimo[0]
```

```
ultimo = primo_e_ultimo[1]
```

Tuttavia, **generalmente, gli elementi della tupla vengono assegnati direttamente a variabili differenti.**

Esempio:

```
primo, ultimo = calcola_primo_e_ultimo_elemento_lista(lista)
```

Definizione di Funzioni

primo, ultimo = calcola_primo_e_ultimo_elemento_lista(lista)

Perchè funziona?

Perchè **in Python si possono assegnare gli elementi di una tupla contenente n elementi ad altrettante variabili** con la seguente sintassi:

<variabile1>, <variabile2> = (<elemento1>, <elemento2>)

Definizione di Funzioni - Variabili Locali

I **parametri e le variabili alle quali viene assegnato un valore all'interno della funzione** sono variabili **locali**, cioè vengono “create” dall'interprete nel momento in cui la funzione viene eseguita (con una chiamata), e vengono “distrutte” quando l'esecuzione della funzione termina.

Definizione di Funzioni - Variabili Locali

Esempio con una variabile alla quale viene assegnato un valore dentro la funzione (e quindi una variabile locale):

```
def incrementa_contatore():  
    contatore = 5 viene definita una variabile locale  
    print("valore contatore dentro la funzione: ", contatore)
```

```
contatore = 0  
incrementa_contatore()  
print(contatore)
```

Stamperà: 0 (la variabile locale viene creata dentro la funzione e poi distrutta)

Definizione di Funzioni - Variabili Locali

Esempio con un parametro (e quindi una variabile locale):

```
def incrementa_contatore(contatore): contatore è una variabile locale  
    contatore = contatore + 1 la variabile locale contatore viene incrementata  
    print("valore contatore dentro la funzione: ", contatore)
```

```
contatore = 0  
incrementa_contatore(contatore)  
print(contatore)
```

Stamperà: 0 (il valore della variabile è stato alterato solo all'interno della funzione.)

Definizione di Funzioni - Variabili Globali

Tutte **le variabili che non sono dei parametri e alle quali non viene assegnato alcun valore all'interno della funzione** sono considerate **globali**.

```
def mostra_contatore():  
    print("valore contatore dentro la funzione: ", contatore) #stamperà 0 e se  
    tentassimo di modificarne il valore verrebbe sollevata un'eccezione
```

```
contatore = 0  
mostra_contatore()  
print(contatore)
```

NB: l'uso di variabili globali in Python è fortemente sconsigliato!

Definizione di Funzioni - Variabili Globali

Tutte **le variabili che non sono dei parametri e alle quali non viene assegnato alcun valore all'interno della funzione** sono considerate **globali**.

```
def mostra_contatore():  
    contatore[0] = contatore[0] + 1
```

```
contatore = [1]  
mostra_contatore()  
print(contatore) # stamperà [2] perchè è stata modificata la variabile globale
```

NB: l'uso di variabili globali in Python è fortemente sconsigliato!

Definizione di Funzioni - Passaggio per Riferimento

In Python, i **tipi mutabili** (e.g., liste e dizionari) **vengono passati per riferimento**. (Quindi se il loro valore viene alterato all'interno della funzione, viene alterato anche all'esterno di essa). **Gli altri, vengono passati per valore**.

```
def altera_primo_valore_lista(l, nuovo_valore):  
    l[0] = nuovo_valore (altera il primo valore della lista)
```

```
lista = [1, 4, 3]  
altera_primo_valore_lista(lista, 0)  
print(lista)
```

Stamperà: [0, 4, 3] (Il primo valore della lista è stato alterato)

Nota sui Valori di Default

E' fortemente **sconsigliato** utilizzare **tipi mutabili** come valori di default.

E' sconsigliato perchè tale valore può essere modificato dalla funzione e quindi l'argomento di default associato a quel parametro non è detto che sia quello specificato nella definizione della funzione.

Nota sui Valori di Default

A lista viene associato un tipo mutabile: una lista.

```
def modifica_primo_elem_lista(new_elem, lista = [0]):  
    """Sostituisce il primo elemento di una lista con l'elemento ricevuto in input.  
    """
```

```
    print(lista)
```

```
    lista[0] = new_elem
```

La lista viene modificata...

```
    return lista
```

```
modifica_primo_elem_lista(3)
```

Viene stampato [0]

```
modifica_primo_elem_lista(4)
```

Viene stampato [3]

In questo esempio, poichè la funzione non fa uso dell'elemento in posizione lista [0] non è un grosso problema ma in funzioni differenti potrebbe esserlo..

Commenti

Si può inserire un **commento su una singola linea** facendo precedere il commento dal carattere cancelletto: #.

Esempio:

```
cont = 0 # inizializzazione della variabile contatore
```


Commenti

Si possono inserire **commenti su più righe** iniziando e terminando il commento con tre doppi apici affiancati """.

Questo tipo di commenti viene generalmente utilizzato per spiegare lo scopo di una funzione e dei suoi parametri.

Commenti

```
def somma_numeri(num_1, num_2):
```

```
    """Sum two numbers.
```

```
Parameters
```

```
-----
```

```
num_1 : int or float
```

```
    First number.
```

```
num_2 : int or float
```

```
    Second number.
```

```
Returns
```

```
-----
```

```
int or float
```

```
    Result of the sum.
```

```
    """
```

```
    return num_1 + num_2
```

Esercizio sulla Definizione di Funzioni

Implementate la funzione *min_max_lista* che riceve in input una lista che contiene almeno un elemento e restituisce in output il minimo e il massimo valore presenti nella lista.

Esercizio sulla Definizione di Funzioni

```
def min_max_lista(lista):  
    lunghezza_lista = len(lista)  
    for i in range(lunghezza_lista):  
        if i == 0:  
            idx_min = 0  
            idx_max = 0  
        else:  
            if lista[i] > lista[idx_max]:  
                idx_max = i  
            if lista[i] < lista[idx_min]:  
                idx_min = i  
    return lista[idx_min], lista[idx_max]
```

Esercizio sulla Definizione di Funzioni

```
def min_max_lista(lista):  
    lunghezza_lista = len(lista)  
    for i in range(lunghezza_lista):  
        if i == 0:  
            idx_min = 0  
            idx_max = 0  
        else:  
            if lista[i] > lista[idx_max]:  
                idx_max = i  
            if lista[i] < lista[idx_min]:  
                idx_min = i  
    return lista[idx_min], lista[idx_max]
```

Per verificarne il funzionamento
possiamo provare a chiamarla

```
lista = [3, 5, 2, 9, 8]  
min, max = min_max_lista(lista)  
print("min ", min)  
print("max ", max)
```

Stamperà:

```
min 2  
max 9
```