



# Matlab: Funzioni

Informatica B AA 2021/2022

Luca Cassano

[luca.cassano@polimi.it](mailto:luca.cassano@polimi.it)

25 Novembre 2021



# A cosa servono funzioni e script?

## Calcolo del **fattoriale**

Scrivere un programma che legge da tastiera un intero  $x$  e calcola  $fx = \prod_{i=1}^x i$

Se  $fx$  è maggiore di 220, il programma legge da tastiera un intero  $y$  e calcola  $fy = \prod_{i=1}^y i$



## A cosa servono funzioni e script?

```
x = input('inserisci x: ');
```



## A cosa servono funzioni e script?

```
x = input('inserisci x: ');  
fx = 1  
for ii = 1 : x  
    fx = fx * ii;  
end
```



## A cosa servono funzioni e script?

```
x = input('inserisci x: ');
```

```
fx = 1
```

```
for ii = 1 : x
```

```
    fx = fx * ii;
```

```
end
```

```
if (fx > 220)
```

```
end
```



## A cosa servono funzioni e script?

```
x = input('inserisci x: ');  
fx = 1  
for ii = 1 : x  
    fx = fx * ii;  
end  
if (fx > 220)  
    y = input('inserisci y: ');  
  
end
```



## A cosa servono funzioni e script?

```
x = input('inserisci x: ');  
fx = 1  
for ii = 1 : x  
    fx = fx * ii;  
end  
if (fx > 220)  
    y = input('inserisci y: ');  
    fy = 1  
    for ii = 1 : y  
        fy = fy * ii;  
    end  
end
```



# A cosa servono funzioni e script?

```
x = input('inserisci x: ');
```

```
fx = 1  
for ii = 1 : x  
    fx = fx * ii;  
end
```

```
if (fx > 220)
```

```
    y = input('inserisci y: ');
```

```
    fy = 1  
    for ii = 1 : y  
        fy = fy * ii;  
    end
```

```
end
```

Entrambi i  
frammenti di  
codice  
eseguono il  
calcolo del  
fattoriale





# A cosa servono funzioni e script?

## Riusabilità

- Scrivo una sola volta codice utilizzato spesso
- Lo stesso codice viene richiamato in diversi programmi



# A cosa servono funzioni e script?

## Riusabilità

- Scrivo una sola volta codice utilizzato spesso
- Lo stesso codice viene richiamato in diversi programmi

## Leggibilità

- Incapsulo porzioni di codice complesso, il programmatore non deve entrare nei dettagli



# A cosa servono funzioni e script?

## **Flessibilità**

- Posso aggiungere funzionalità non presenti nelle funzioni di libreria



# A cosa servono funzioni e script?

## **Flessibilità**

- Posso aggiungere funzionalità non presenti nelle funzioni di libreria

## **Manutenibilità**

- Modifiche e correzioni sono gestibili facilmente
- E' più difficili commettere errori sistematici



## Usiamo uno script file?

Uno script è un file che può essere usato per incapsulare porzioni di codice riusabili in futuro



## Usiamo uno script file?

Uno script è un file che può essere usato per incapsulare porzioni di codice riusabili in futuro

```
x = input('inserisci x: ');
fx=1
for ii=1:x
    fx = fx*ii
end
if (fx>220)
    y = input('inserisci y: ');
    fy=1
    for ii=1:y
        fy = fy*ii
    end
end
```



## Usiamo uno script file?

Uno script è un file che può essere usato per incapsulare porzioni di codice riusabili in futuro

```
x = input('inserisci x: ');
```

```
fx=1  
for ii=1:x  
    fx = fx*ii  
end
```

```
if (fx>220)
```

```
    y = input('inserisci y: ');
```

```
    fy=1  
    for ii=1:y  
        fy = fy*ii  
    end
```

```
end
```

```
f=1  
for ii=1:n  
    f = f*ii  
end
```

fattoriale.m



## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
f=1
for ii=1:n
    f = f*ii
end
```

fattoriale.m





## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
x = input('inserisci x: ');
```

```
f=1
for ii=1:n
    f = f*ii
end
```

fattoriale.m



## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
x = input('inserisci x: ');
```

```
n=x ← Prepara l'input in n
```

```
f=1
for ii=1:n
    f = f*ii
end
```

fattoriale.m



## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
x = input('inserisci x: ');
```

```
n=x ← Prepara l'input in n
```

```
fattoriale ← chiama lo script
```

```
f=1  
for ii=1:n  
    f = f*ii  
end
```

fattoriale.m



## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
x = input('inserisci x: ');
```

```
n=x ← Prepara l'input in n
```

```
fattoriale ← chiama lo script
```

```
fx=f ← Salva il risultato in fx
```

```
f=1
for ii=1:n
    f = f*ii
end
```

fattoriale.m



## Limiti degli script-files

### Problemi:

- Come fornisco l'input allo script?
- Dove recupero l'output?

Gli script utilizzano le variabili del workspace:

```
x = input('inserisci x: ');  
n=x ← Prepara l'input in n  
fattoriale ← chiama lo script  
fx=f ← Salva il risultato in fx  
if (fx>220)  
    y = input('inserisci y: ');  
    n=y ← Prepara l'input  
    fattoriale ← chiama lo script  
    fy=f ← Salva il risultato in fy  
end
```

```
f=1  
for ii=1:n  
    f = f*ii  
end
```

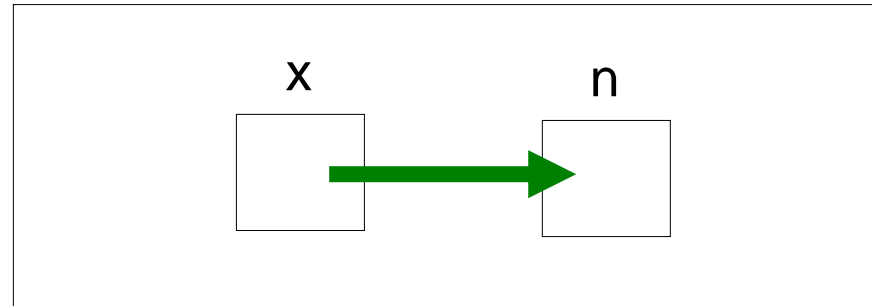
fattoriale.m



## Limiti degli script-files (2)

**n=x**

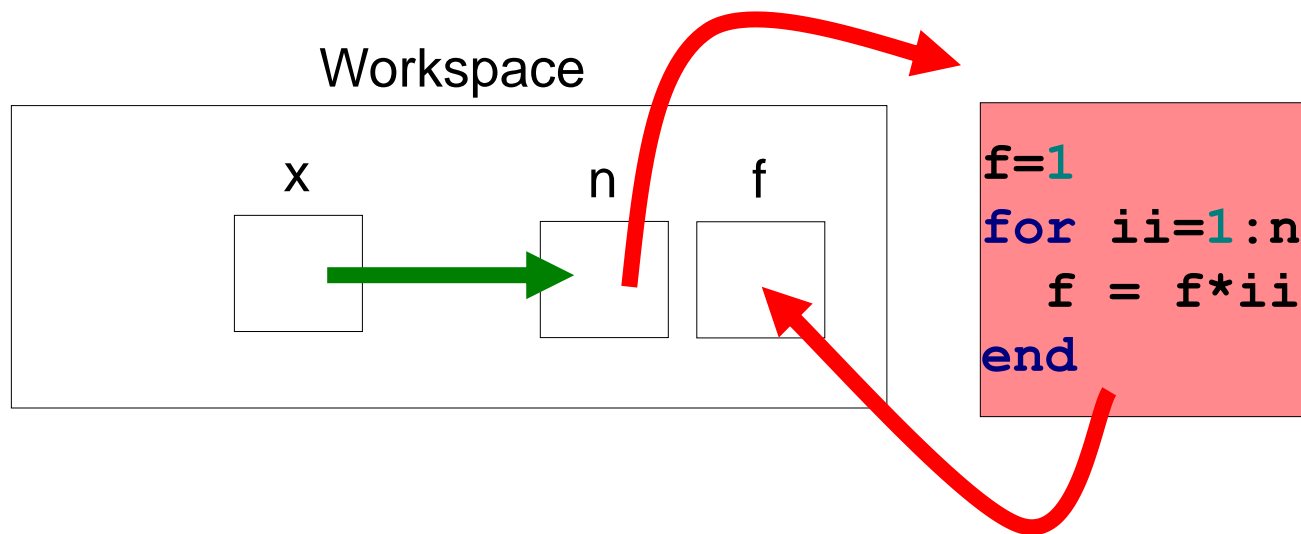
Workspace





## Limiti degli script-files (2)

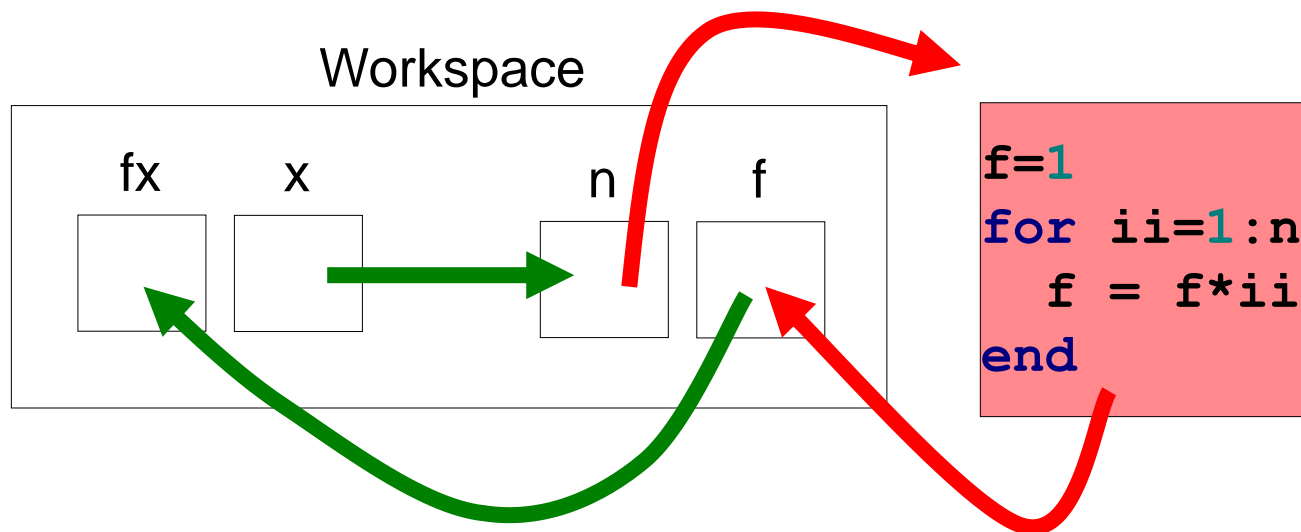
```
n=x  
fattoriale
```





## Limiti degli script-files (2)

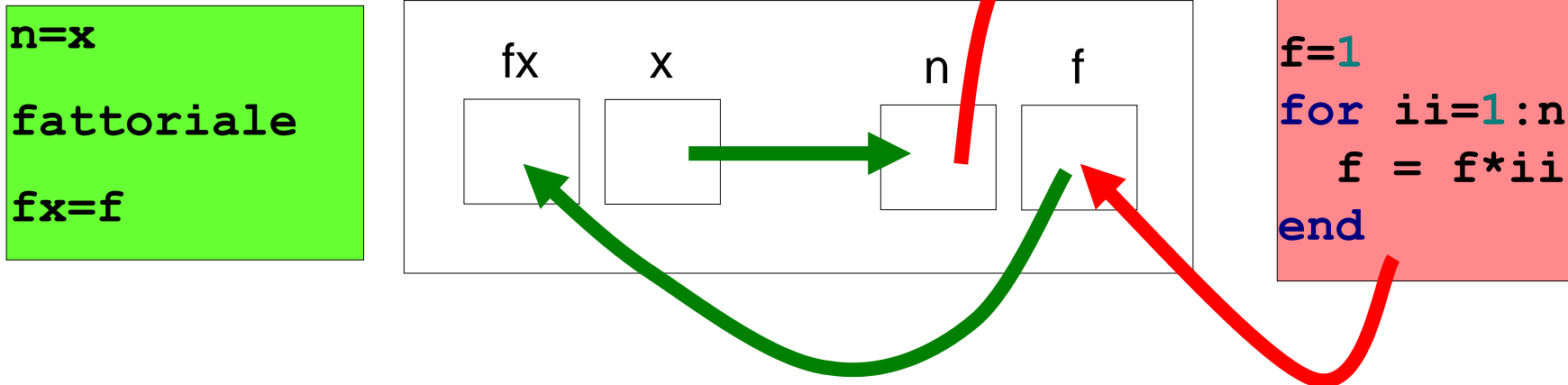
```
n=x  
fattoriale  
fx=f
```







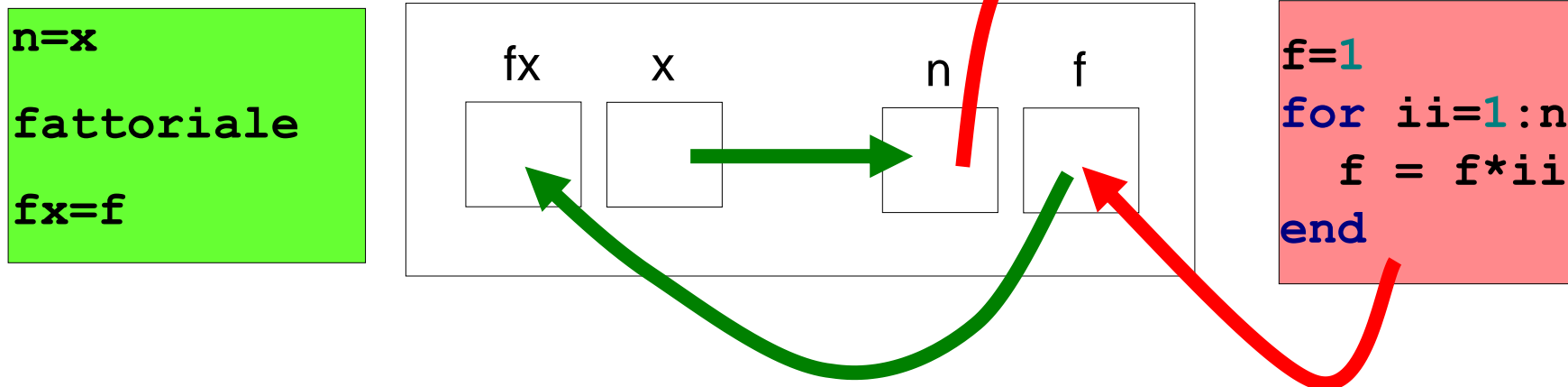
## Limiti degli script-files (2)



- Questo meccanismo ha molti svantaggi:
  - poco leggibile
  - richiede molte istruzioni
  - poco sicuro



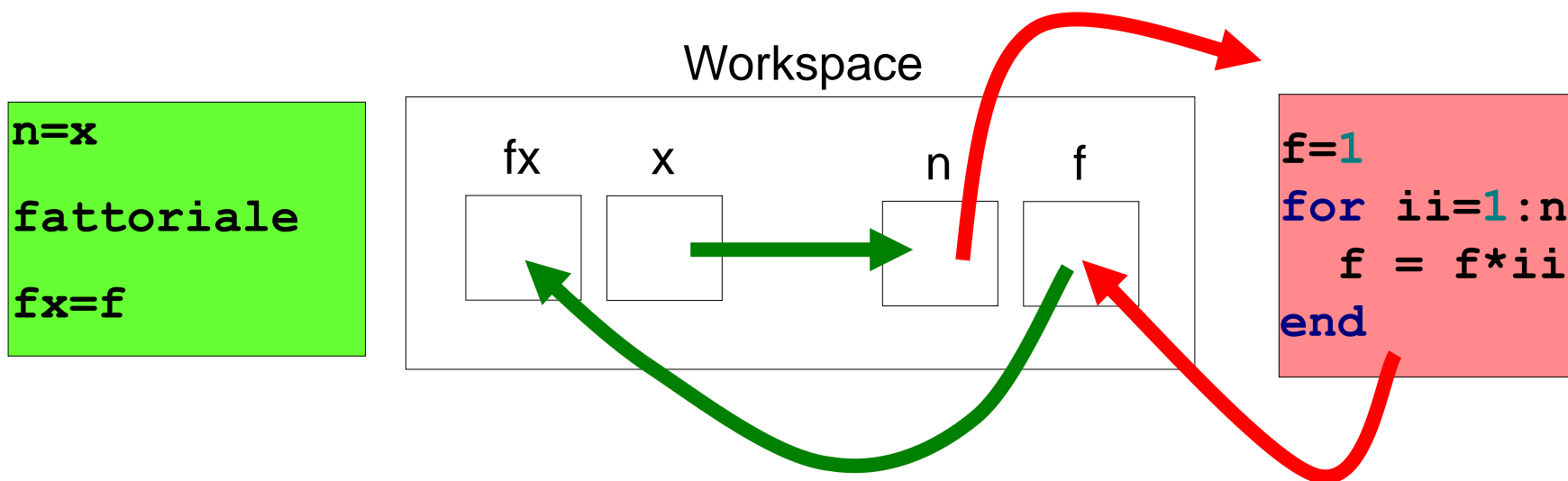
## Limiti degli script-files (2)



- Questo meccanismo ha molti svantaggi:
  - poco leggibile
  - richiede molte istruzioni
  - poco sicuro
- Tutte le variabili sono nello stesso workspace (fattoriale.m può modificare tutte le variabili del workspace)



## Limiti degli script-files (2)



- Questo meccanismo ha molti svantaggi:
  - poco leggibile
  - richiede molte istruzioni
  - poco sicuro
- Tutte le variabili sono nello stesso workspace (fattoriale.m può modificare tutte le variabili del workspace)

**Le funzioni non hanno questi problemi**



## Le funzioni

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```



## Le funzioni

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```

**n** è l'argomento della funzione (serve a fornire l'input)



## Le funzioni

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```

**n** è l'**argomento** della funzione (serve a fornire l'input)

**f** è il **valore di ritorno** della funzione (serve a fornire l'output)



# Le funzioni

```
header
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```

**n** è l'argomento della funzione (serve a fornire l'input)

**f** è il **valore di ritorno** della funzione (serve a fornire l'output)

- La testata (o intestazione o header) inizia con la parola chiave **function** e definisce:
  - nome della funzione
  - argomenti (input)
  - valore di ritorno (output)



## Le funzioni

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```

header

body

**n** è l'argomento della funzione (serve a fornire l'input)

**f** è il **valore di ritorno** della funzione (serve a fornire l'output)

- La testata (o intestazione o header) inizia con la parola chiave `function` e definisce:
  - nome della funzione
  - argomenti (input)
  - valore di ritorno (output)
- Il corpo definisce le istruzioni da eseguire se la funzione viene chiamata
  - Utilizza gli argomenti e assegna il valore di ritorno





## Sintassi per la Definizione di una Funzione

La sintassi per definire l'header di funzione è:

```
function [out1, .., outM] = nomeFunzione(in1, .., inN)  
end
```



## Sintassi per la Definizione di una Funzione

La sintassi per definire l'header di funzione è:

```
function [out1, .., outM] = nomeFunzione(in1, .., inN)  
end
```

**Gli argomenti (parametri in ingresso)  $in1, \dots, inN$**   
vanno elencati tra parentesi tonde e seguono il nome della  
funzione



## Sintassi per la Definizione di una Funzione

La sintassi per definire l'header di funzione è:

```
function [out1, .., outM] = nomeFunzione(in1, .., inN)  
end
```

**Gli argomenti (parametri in ingresso)  $in1, \dots, inN$**   
vanno elencati tra parentesi tonde e seguono il nome della  
funzione

**I valori ritornati (parametri in uscita)  $out1, \dots, outN$**   
vanno elencati tra parentesi quadre e seguono la keyword  
**function**.



## Sintassi per la Definizione di una Funzione

La sintassi per definire l'header di funzione è:

```
function [out1, .., outM] = nomeFunzione(in1, .., inN)
end
```

**NB:** se la funzione non ha parametri in ingresso/uscita le parentesi tonde/quadre rimangono vuote

```
function [] = nomeFunzione(in1, .., inN)
end
```

```
function [out1, .., outM] = nomeFunzione()
end
```



## Esempi

Una funzione può avere più argomenti separati da virgola:

```
function [v1] = f(x,y)
```

Nel caso sia necessario ritornare più valori, definiamo l'header affiancando più variabili in output usando la stessa notazione degli array (**l'output non deve necessariamente essere omogeneo**):

```
function [v1,v2,...] = f(x,y)
```



## Esempi

Una funzione può avere più argomenti separati da virgola:

```
function [v1] = f(x,y)
```

Nel caso sia necessario ritornare più valori, definiamo l'header affiancando più variabili in output usando la stessa notazione degli array (**l'output non deve necessariamente essere omogeneo**):

```
function [v1,v2,...] = f(x,y)
```

Esempio:

```
function [s, p] = sumProd(a, b)
    s = a + b;
    p = a * b;
end
```



## Invocazione di una funzione

Una funzione può essere invocata in un programma attraverso il suo nome, seguito dagli argomenti fra parentesi rotonde

La funzione viene quindi eseguita e il suo valore di ritorno viene calcolato (i suoi valori di ritorno vengono calcolati, nel caso fossero più di uno).

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```



## Invocazione di una funzione

Una funzione può essere invocata in un programma attraverso il suo nome, seguito dagli argomenti fra parentesi rotonde

La funzione viene quindi eseguita e il suo valore di ritorno viene calcolato (i suoi valori di ritorno vengono calcolati, nel caso fossero più di uno).

```
x = input('inserire x: ');
```

```
fx = fattoriale(x);
```



```
if (fx>220)
```

```
    y = input('inserisci y: ');
```

```
    fy = fattoriale(y);
```



```
end
```

```
function f=fattoriale(n)
    f=1
    for ii=1:n
        f = f*ii
    end
```





## Definizioni:

- I **parametri formali** sono le variabili usate come **argomenti** e **valori di ritorno** **nella definizione** della funzione



### Definizioni:

- I **parametri formali** sono le variabili usate come **argomenti** e **valori di ritorno** **nella definizione** della funzione
- I **parametri attuali** sono i valori (o le variabili) usati come **argomenti** e come **valori di ritorno** **nell'invocazione** della funzione



## Definizioni:

```
function f=fattoriale(n)
```

```
    f = 1;
```

```
    for ii=1:n
```

```
        f = f*ii;
```

```
    end
```

f ed n sono parametri formali



## Definizioni:

```
function f=fattoriale(n)
```

```
    f = 1;
```

```
    for ii=1:n
```

```
        f = f*ii;
```

```
    end
```

f ed n sono parametri formali

```
>> fat5 = fattoriale(5) %Invocazione
```

```
fat5 =
```

```
    120
```

fat5 e 5 sono parametri attuali



## I Parametri (2)

**Qualsiasi tipo di parametri** è ammesso (scalari, vettori, matrici, strutture, ecc.)



## I Parametri (2)

**Qualsiasi tipo di parametri** è ammesso (scalari, vettori, matrici, strutture, ecc.)

**I parametri attuali** vengono **associati a quelli formali** in base alla **posizione**:

Esempio

```
>> [x,y]=sumProd(4,5)
```

```
function [s,p]=sumProd(a,b)  
    s=a+b;  
    p=a*b;
```



## I Parametri (2)

**Qualsiasi tipo di parametri** è ammesso (scalari, vettori, matrici, strutture, ecc.)

**I parametri attuali** vengono **associati a quelli formali** in **base alla posizione**:

- il primo parametro attuale viene associato al primo formale

Esempio

```
>> [x,y]=sumProd(4,5)
```

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```



## I Parametri (2)

**Qualsiasi tipo di parametri** è ammesso (scalari, vettori, matrici, strutture, ecc.)

**I parametri attuali** vengono **associati a quelli formali** in **base alla posizione**:

- il primo parametro attuale viene associato al primo formale
- il secondo parametro attuale al secondo parametro formale

Esempio

```
>> [x,y]=sumProd(4,5)
```

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```





## I Parametri (2)

**Qualsiasi tipo di parametri** è ammesso (scalari, vettori, matrici, strutture, ecc.)

**I parametri attuali** vengono **associati a quelli formali** in **base alla posizione**:

- il primo parametro attuale viene associato al primo formale
- il secondo parametro attuale al secondo parametro formale

Esempio

>> [x,y]=sumProd(4,5)

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```



## Esecuzione di una funzione

Quando una funzione viene invocata, viene creato un **workspace “locale”** in cui vengono memorizzate tutte le variabili usate nella funzioni **inclusi i parametri formali**.





## Esecuzione di una funzione

Quando una funzione viene invocata, viene creato un **workspace “locale”** in cui vengono memorizzate tutte le variabili usate nella funzioni **inclusi i parametri formali**.

- All'interno delle funzioni **non si può accedere al workspace “principale”** (nessun conflitto di nomi)





## Esecuzione di una funzione

Quando una funzione viene invocata, viene creato un **workspace “locale”** in cui vengono memorizzate tutte le variabili usate nella funzioni **inclusi i parametri formali**.

- All'interno delle funzioni **non si può accedere al workspace “principale”** (nessun conflitto di nomi)
- Al termine dell'esecuzione della funzione, **il workspace “locale” viene distrutto!**

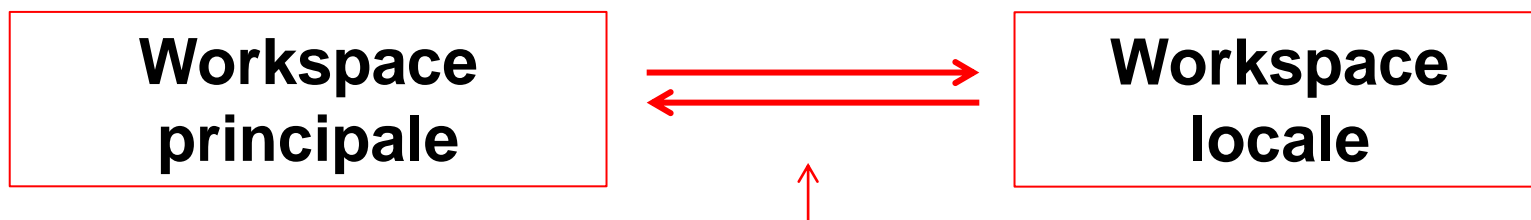




## Esecuzione di una funzione

Quando una funzione viene invocata, viene creato un **workspace “locale”** in cui vengono memorizzate tutte le variabili usate nella funzioni **inclusi i parametri formali**.

- All'interno delle funzioni **non si può accedere al workspace “principale”** (nessun conflitto di nomi)
- Al termine dell'esecuzione della funzione, **il workspace “locale” viene distrutto!**



Le comunicazioni tra i workspace avvengono solamente mediante copia dei valori dei parametri in ingresso ed in uscita



## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso



## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso
2. Viene **creato un workspace “locale”** per la funzione



## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso
2. Viene **creato un workspace “locale”** per la funzione
3. I **valori** dei **parametri attuali** di ingresso vengono **copiati** nei **parametri formali** all'interno del **workspace “locale”**
  - Il workspace locale ora contiene solamente i parametri formali con assegnati i valori dei parametri attuali





## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso
2. Viene **creato un workspace “locale”** per la funzione
3. I **valori** dei **parametri attuali** di ingresso vengono **copiati** nei **parametri formali** all'interno del **workspace “locale”**
  - Il workspace locale ora contiene solamente i parametri formali con assegnati i valori dei parametri attuali
4. Viene **eseguito il corpo** della **funzione**



## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso
2. Viene **creato un workspace “locale”** per la funzione
3. I **valori** dei **parametri attuali** di ingresso vengono **copiati** nei **parametri formali** all'interno del **workspace “locale”**
  - Il workspace locale ora contiene solamente i parametri formali con assegnati i valori dei parametri attuali
4. Viene **eseguito il corpo** della **funzione**
5. Vengono **copiati i valori di ritorno dai parametri formali nel workspace “locale” al workspace “principale”** nei corrispondenti parametri attuali



## Riepilogando: Esecuzione di una funzione (2)

Quando viene invocata una funzione:

1. Vengono **calcolati** i valori dei **parametri attuali** di ingresso
2. Viene **creato un workspace “locale”** per la funzione
3. I **valori** dei **parametri attuali** di ingresso vengono **copiati** nei **parametri formali** all'interno del **workspace “locale”**
  - Il workspace locale ora contiene solamente i parametri formali con assegnati i valori dei parametri attuali
4. Viene **eseguito il corpo** della **funzione**
5. Vengono **copiati i valori di ritorno dai parametri formali nel workspace “locale” al workspace “principale”** nei corrispondenti parametri attuali
6. Il workspace “locale” viene **distrutto**



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
end
```

W “principale”





## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
end
```

W “principale” dopo (1)

```
x=3
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
end
```

W “principale” dopo (2)

```
x=3  
w=2
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
end
```

W “locale” prima (1')

```
x=4
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
end
```

W “principale” dopo (2)

```
x=3  
w=2
```

W “locale” dopo (1')

```
x=4  
y=8
```





## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
end
```

W “principale” dopo (2)

```
x=3  
w=2
```

W “locale” dopo (2')

```
x=0  
y=8
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
end
```

W “principale” dopo (2)

```
x=3  
w=2
```

W “locale” dopo (3')

```
x=0  
y=8  
z=4
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
end
```

W “principale” dopo (3)

```
x=3  
w=2  
r=8
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale”

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
    x = w - 1;    %(4')  
end
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (1)

```
x=3
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;        %(2')  
    z = 4;        %(3')  
    x = w - 1;   %(4')  
end
```

W “locale”





## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```

W “locale” prima (1')

```
x=4
```





## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```

W “locale” dopo (1')

```
x=4  
y=8
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```

W “locale” dopo (2')

```
x=0  
y=8
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;  %(4')  
end
```

W “locale” dopo (3')

```
x=0  
y=8  
z=4
```



## Esecuzione di una funzione: esempio

```
(1) >> x=3;  
(2) >> w=2;  
(3) >> r = funz(4);
```

W “principale” dopo (2)

```
x=3  
w=2
```

```
function y = funz(x)  
    y = 2*x;      %(1')  
    x = 0;       %(2')  
    z = 4;       %(3')  
    x = w - 1;   %(4')  
end
```

W “locale” dopo (4')

```
x=0  
y=8  
z=4  
w=? → errore
```



## I Parametri (3)

**Il numero di parametri di ingresso attuali all'invocazione della funzione deve essere identico al numero di **parametri di ingresso formali della funzione****



## I Parametri (3)

**Il numero di parametri di ingresso attuali** all'invocazione della funzione deve essere identico al numero di **parametri di ingresso formali della funzione**

**Il numero di parametri di uscita attuali** all'invocazione della funzione può essere minore del numero di **parametri di uscita formali della funzione**

- Ad esempio `s = sommaProd(5, 2)` il valore della somma viene assegnato a `s` ma non il valore del prodotto (anche se la funzione lo calcola)



## Note sui Parametri in Uscita

**I parametri formali** dei valori **di ritorno** devono essere **sempre definiti** (eventualmente possono essere vuoti)



## Note sui Parametri in Uscita

**I parametri formali dei valori di ritorno devono essere sempre definiti** (eventualmente possono essere vuoti)

```
function [positivi, media] = mediaPositivi(vett)
    somma = 0; cnt = 0;
    positivi = [];
    for ii = 1 : length(vett)
        if vett(ii) > 0
            positivi = [positivi, vett(ii)];
            somma = somma + vett(ii);
            cnt = cnt + 1;
        end
    end
    if cnt > 0
        media = somma / cnt;
    end
end
```

```
>> [a,b] = mediaPositivi(-[1 : 10])
Error in mediaPositivi (line 2)
positivi = vett(vett >0);
```

Output argument "media" (and maybe others) not assigned during call to mediaPositivi





## Note sui Parametri in Uscita

**I parametri formali dei valori di ritorno devono essere sempre definiti (eventualmente possono essere vuoti)**

```
function [positivi, media] = mediaPositivi(vett)
    somma = 0; cnt = 0;
    positivi = [];
    for ii = 1 : length(vett)
        if vett(ii) > 0
            positivi = [positivi, vett(ii)];
            somma = somma + vett(ii);
            cnt = cnt + 1;
        end
    end
    if cnt > 0
        media = somma / cnt;
    else
        media = [];
    end
end
```



## Note sull'output

>> [x,y]=sumProd(4,5)

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```



## Note sull'output

>> [x,y]=sumProd(4,5)

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```

È però possibile invocare la funzione senza specificare due parametri in uscita,

- `x = sumProd(4,5)`. In tal caso solamente il primo output viene assegnato ad `x`



## Note sull'output

>> [x,y]=sumProd(4,5)

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```

È però possibile invocare la funzione senza specificare due parametri in uscita,

- `x = sumProd(4,5)`. In tal caso solamente il primo output viene assegnato ad `x`

L'invocazione associa alla variabile `ans` il secondo argomento restituito da `sumProd`



## Note sull'output

>> [x,y]=sumProd(4,5)

```
function [s,p]=sumProd(a,b)
    s=a+b;
    p=a*b;
```

Per ricevere solo il secondo output uso ~ come se fosse una variabile da non considerare [~,y] = sumProd(4,5)



## File Funzione

Come nel caso degli script, anche le funzioni possono essere scritti in file di testo sorgenti

- Devono avere estensione .m
- Devono avere lo stesso nome della funzione
- La prima riga del file deve contenere l'header della funzione e di fatto iniziare con la parola chiave **function**



## File Funzione

Come nel caso degli script, anche le funzioni possono essere scritti in file di testo sorgenti

- Devono avere estensione .m
- Devono avere lo stesso nome della funzione
- La prima riga del file deve contenere l'header della funzione e di fatto iniziare con la parola chiave **function**

Attenzione a non “ridefinire” funzioni esistenti

- `exist('nomeFunzione')` → 0 se la funzione non esiste



## File Funzione

Come nel caso degli script, anche le funzioni possono essere scritte in file di testo sorgenti

- Devono avere estensione `.m`
- Devono avere lo stesso nome della funzione
- La prima riga del file deve contenere l'header della funzione e di fatto iniziare con la parola chiave **function**

Attenzione a non “ridefinire” funzioni esistenti

- `exist('nomeFunzione')` → 0 se la funzione non esiste

Se le prime righe della funzione (dopo l'header) contengono commenti, questi rappresentano l'help della funzione e vengono visualizzati quando si scrive:

```
>> help nomeFunzione
```





## Esercizio

Scrivere una funzione che prende in ingresso due coefficienti  $m, q$  ed un vettore di punti  $xx$  e restituisce il vettore  $yy$  dei punti che stanno sulla retta  $y = mx + q$  in corrispondenza a  $xx$



## Esercizio

Scrivere una funzione che prende in ingresso due coefficienti  $m, q$  ed un vettore di punti  $xx$  e restituisce il vettore  $yy$  dei punti che stanno sulla retta  $y = mx + q$  in corrispondenza a  $xx$

```
function yy = retta(m, q, xx)
    % function yy = retta(m, q, xx)
    % m, q sono i coefficienti e xx un vettore di punti
    % la funzione restituisce il vettore yy dei punti
    % che stanno sulla
    % retta  $y = mx + q$  in corrispondenza a xx

    yy = m * xx + q;

end
```



## Esercizio

Implementare la funzione trasposizione per le matrici



## Esercizio

Implementare la funzione trasposizione per le matrici

```
function [t]=trasposta(m)
    [r,c]=size(m);
    for ii=1:r
        for jj=1:c
            t(jj,ii)=m(ii,jj);
        end
    end
end
```

```
>> m =[1,2,3,4
        5,6,7,8
        9,10,11,12]

m =

     1     2     3     4
     5     6     7     8
     9    10    11    12

>> trasposta(m)

ans =

     1     5     9
     2     6    10
     3     7    11
     4     8    12
```




## Esercizio

```
%% scrivere un programma che chiede all'utente di inserire un
% numero positivo (nel caso in cui il numero non è positivo
ripetere l'inserimento)
%
% verificare se il numero è perfetto
%
% in caso contrario dice se è abbondante o difettivo.
% Dopo di che richiede un altro numero e controlla se
% i due numeri sono amici
%
% un numero è perfetto se corrisponde alla somma
% dei suoi divisori, escluso se stesso
% abbondante se è > della somma dei suoi divisori
% altrimenti difettivo
%
% a,b sono amici se la somma dei divisori di a= b e viceversa
```



## Implemento diverse funzioni che richiamo

```
function n = inserisciInteroPositivo()  
% function n = inserisciInteroPositivo()  
%  
% richiede all'utente di inserire un intero positivo  
% e lo restituisce  
  
function somma = calcolaSommaDivisori(n)  
%function somma = calcolaSommaDivisori(n)  
%  
% calcola la somma di tutti i divisori di n escluso n  
  
function [res, abb] = controllaSePerfetto(n)  
% function [res, abb] = controllaSePerfetto(n)  
%  
% res = true se n è perfetto (uguale alla somma dei suoi  
divisori escluso se stesso)  
% se res = false e abb = true/false se è abbondante o  
difettivo  
  
function res = controllaSeAmici(a,b)  
% function res = controllaSeAmici(a,b)  
%  
% res = 1 se a è amico di b, 0 altrimenti
```



```
function n = inserisciInteroPositivo()
    %
    % function n = inserisciInteroPositivo()
    %
    % richiede all'utente di inserire un intero
    positivo
    % e lo restituisce

    isPositivo = 0

    while(isPositivo == 0)
        n = input('Inserire intero positivo: ')

        isPositivo = (n > 0 && n == round(n));

    end
end
```




```
function somma = calcolaSommaDivisori(n)
%
%function somma = calcolaSommaDivisori(n)
%
% calcola la somma di tutti i divisori di n escluso
n

somma = 0;
for ii = 1 : n / 2 % inutile procedere oltre a n/2.
    if (mod(n, ii) == 0)
        somma = somma + ii;
    end
end
end
```





```
function [res, abb] = controllaSePerfetto(n)
    % function [res, abb] = controllaSePerfetto(n)
    %
    % res = true se n è perfetto
    %
    % se res = false e abb = true/false n è
    abbondante/difettivo
    s = calcolaSommaDivisori(n);
    abb = [];
    if (n == s)
        res = true;
    else
        res = false;
        if n > s
            abb = true;
        else
            abb = false;
        end
    end
end
end
```



```
function res = controllaSeAmici(a,b)
%
% function res = controllaSeAmici(a,b)
%
% res = 1 se a è amico di b, 0 altrimenti

if b == calcolaSommaDivisori(a) && a ==
calcolaSommaDivisori(b)
    res = true;
else
    res = false;
end
end
```



## Script per l'utilizzo delle funzioni definite

```
n = inserisciInteroPositivo();
[perf, abbond] = controllaSePerfetto(n);
if(perf == true)
    disp([num2str(n), ' è perfetto']);
else
    disp([num2str(n), ' NON è perfetto']);
    if(abbond == true)
        disp([num2str(n), ' è abbondante']);
    else
        disp([num2str(n), ' è difettivo']);
    end
end
m = inserisciInteroPositivo();
amici = controllaSeAmici(n,m);
if(amici)
    disp([num2str(n), ' e ', num2str(m), ' sono amici']);
else
    disp([num2str(n), ' e ', num2str(m), ' NON sono amici']);
end
end
```