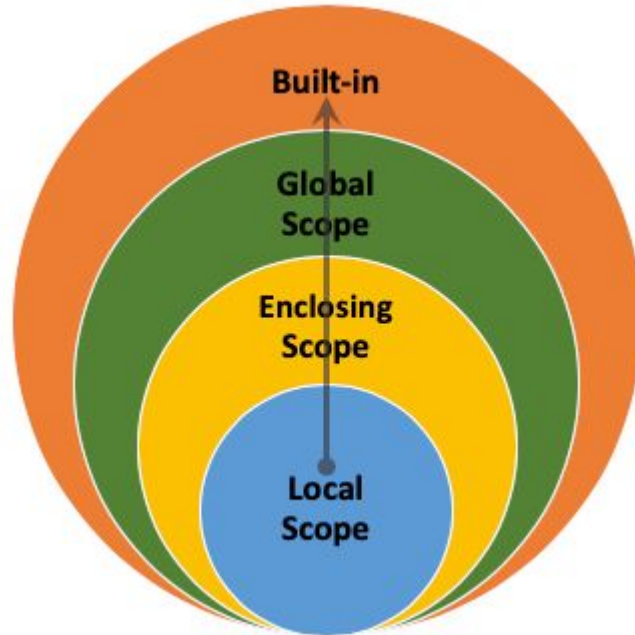
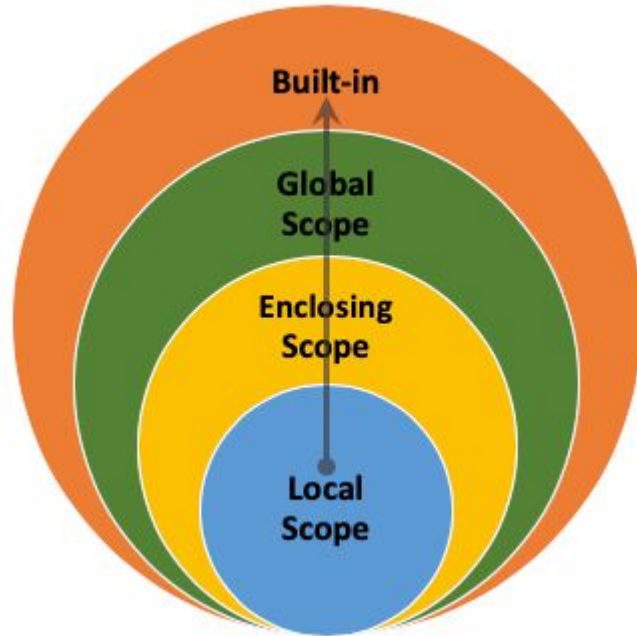


# Scope



# Scope



- Built-in: entità sempre disponibili. Es. print, disponibile non appena apriamo la shell
- Global: entità definite nel codice (non dentro a funzioni o blocchi)
- Enclosing: entità incapsulate per effetto della closure
- Local: entità definite dentro il blocco di codice in cui si è

# Scope

```
print('ciao')
```

- **Built-in: entità sempre disponibili. Es. print, disponibile non appena apriamo la shell**
- Global: entità definite nel codice (non dentro a funzioni o blocchi)
- Enclosing: entità incapsulate per effetto della closure
- Local: entità definite dentro il blocco di codice in cui si è

# Scope

```
print('ciao')
```

```
a = 1
```

- Built-in: entità sempre disponibili. Es. print, disponibile non appena apriamo la shell
- **Global: entità definite nel codice (non dentro a funzioni o blocchi)**
- Enclosing: entità incapsulate per effetto della closure
- Local: entità definite dentro il blocco di codice in cui si è

# Scope

```
print('ciao')
```

```
a = 1
```

```
def encaps():
```

```
    b = 2
```

```
    def local():
```

```
        c = 3 + b
```

- Built-in: entità sempre disponibili. Es. print, disponibile non appena apriamo la shell
- Global: entità definite nel codice (non dentro a funzioni o blocchi)
- **Enclosing: entità incapsulate per effetto della closure**
- Local: entità definite dentro il blocco di codice in cui si è

# Scope

```
print('ciao')
```

```
a = 1
```

```
def encaps():
```

```
    b = 2
```

```
    def local():
```

```
        c = 3 + b
```

- Built-in: entità sempre disponibili. Es. print, disponibile non appena apriamo la shell
- Global: entità definite nel codice (non dentro a funzioni o blocchi)
- Enclosing: entità incapsulate per effetto della closure
- **Local: entità definite dentro il blocco di codice in cui si è**

# Scope

```
print('ciao')
```

```
a = 1
```

```
def encaps():
```

```
    b = 2
```

```
    def local():
```

```
        c = 3 + b
```

```
def another():
```

```
    d = 0
```

In questo punto del codice posso vedere  
print, a, b, c

- print -> scope built-in
- a -> scope globale
- b -> scope enclosed
- c -> scope locale

Non vedo d in quanto, in questo punto non  
è in nessuno scope

# Scope

```
print('ciao')
```

```
a = 1
```

```
def encaps():
```

```
    b = 2
```

```
    def local():
```

```
        c = 3 + b
```

```
def another():
```

```
    d = 0
```

In questo punto del codice posso vedere  
print, a, b, c

- print -> scope built-in
- a -> scope globale
- b -> scope enclosed
- c -> scope locale

Non vedo d in quanto, in questo punto non  
è in nessuno scope



# Scope

```
print('ciao')
```

```
a = 1
```

```
def encaps():
```

```
    b = 2
```

```
    def local():
```

```
        c = 3 + b
```

```
def another():
```

```
    d = 0
```

Meccanismo di risoluzione:

- si parte dal Local
- poi si guarda enclosed
- poi global
- infine built-in

# Scope

```
print('ciao')
```

```
x = 1
```

```
def encaps():
```

```
    x = 2
```

```
    def local():
```

```
        x = 3
```

```
def another():
```

```
    x = 0
```

Si possono chiamare le variabili con lo stesso nome 'nascondendo' quelle degli scope più esterni

- si guarda lo scope più interno
- solo se quella entità non vi è si passa ai successivi

# Scope

```
print('ciao')
```

```
x = 1
```

```
def encaps():
```

```
    x = 2
```

```
    def local():
```

```
        global x
```

```
        x = 3
```

```
def another():
```

```
    x = 0
```

Si può forzare l'utilizzo di un'entità dello scope globale con la parola chiave **global**

- **in questo caso x si riferisce al 'primo' x**

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

Anche gli argomenti delle funzioni entrano nel Local scope (altrimenti non li potremmo usare)

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

Anche gli argomenti delle funzioni entrano nel Local scope (altrimenti non li potremmo usare)

output:

5

ciao

5

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

output:

5

ciao

5

All'interno della funzione viene cambiata la  
x locale

Al di fuori rimane la x di prima

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

E come se la x sia copiata nel parametro della funzione.

Passaggio per valore

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x.append(3)  
    print(x)
```

Output

?

```
x = [1,2]  
fun(x)  
print(x)
```



# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x.append(3)  
    print(x)
```

```
x = [1,2]  
fun(x)  
print(x)
```

Output

[1,2]

[1,2,3]

**[1,2,3]**

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x.append(3)  
    print(x)
```

```
x = [1,2]  
fun(x)  
print(x)
```

Output

[1,2]

[1,2,3]

**[1,2,3]**

Come se fosse passato per riferimento

# Scope - argomenti funzioni

Abbiamo detto che in Python è tutto un oggetto.

Come mai abbiamo due comportamenti differenti?

# Scope - argomenti funzioni

In realtà in Python viene **copiato il riferimento all'oggetto**

Quindi nella x della funzione vi è il riferimento allo stesso oggetto puntato dalla x globale

# Scope - argomenti funzioni

In realtà in Python viene **copiato il riferimento all'oggetto**

Quindi nella x della funzione vi è il riferimento allo stesso oggetto puntato dalla x globale

Se si chiama **append** lo si chiama sullo stesso oggetto poichè tutte e due le x puntano al medesimo oggetto

```
x.append(3)
```

# Scope - argomenti funzioni

In realtà in Python viene **copiato il riferimento all'oggetto**

Quindi nella x della funzione vi è il riferimento allo stesso oggetto puntato dalla x globale


Quando si ri-assegna x si cambia il riferimento della x locale che ora punterà al nuovo oggetto

x = 3

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```



<b>Variabile</b>	<b>Punta a</b>
x globale	5
x locale	non esiste

# Scope - argomenti funzioni

```
def fun(x):  
    print(x) ←  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

Variabile	Punta a
x globale	5
x locale	5 (è stato copiato il riferimento)



# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao' ←  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```

Variabile	Punta a
x globale	5
x locale	'ciao' (è stato cambiato il riferimento)

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```



```
x = 5  
fun(x)  
print(x)
```

Variabile	Punta a
x globale	5
x locale	'ciao'

# Scope - argomenti funzioni

```
def fun(x):  
    print(x)  
    x = 'ciao'  
    print(x)
```

```
x = 5  
fun(x)  
print(x)
```



<b>Variabile</b>	<b>Punta a</b>
x globale	5
x locale	non esiste più

# Gestione delle eccezioni

Come in altri linguaggi, Python gestisce gli errori attraverso il concetto di Eccezione.

Nell'ambiente di runtime (RE), viene definita una classe madre che rappresenta eccezioni software generiche.

Ogni anomalia a runtime è associata a una sottoclasse specifica della classe madre.

Le eccezioni possono essere sollevate in due modi:

1. **Automaticamente:** Queste eccezioni si verificano in seguito a anomalie a runtime causate da istruzioni (ad esempio, divisione per zero).
2. **Manualmente:** I programmatori possono sollevare manualmente eccezioni utilizzando istruzioni come "throw" o "raise".

In caso di eccezione, viene eseguito del codice di gestione. Se non è presente un gestore specifico, verrà utilizzato un gestore predefinito che stamperà lo stack delle chiamate ed uscirà dal programma.

# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

**codice**

**except Classe\_eccezione as identificativo:**

**codice**

**except Altraclasse\_eccezione as identificativo:**

**codice**

**else:**

**codice**

**finally:**

**codice**

# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

```
try:
    codice
except Classe_eccezione as identificativo:
    codice
except Altraclasse_eccezione as identificativo:
    codice
else:
    codice
finally:
    codice
```

← viene eseguito

# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

**codice**

**except Classe\_eccezione as identificativo:**

**codice**



**except Altraclasse\_eccezione as identificativo:**

**codice**

**else:**

**codice**

**finally:**

**codice**

viene eseguito se nel blocco try  
viene sollevata un'eccezione  
del tipo Classe\_eccezione

# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

**codice**

**except Classe\_eccezione as identificativo:**

**codice**

**except Altraclasses\_eccezione as identificativo:**

**codice**



**else:**

**codice**

**finally:**

**codice**

viene eseguito se nel blocco try  
viene sollevata un'eccezione  
del tipo Altraclasses\_eccezione



# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

    codice

**except Classe\_eccezione as identificativo:**

    codice

**except Altraclasse\_eccezione as identificativo:**

    codice

**except:**

    codice

**else:**

    codice

**finally:**

    codice

viene eseguito se nel blocco try  
viene sollevata un'eccezione  
non catturata esplicitamente  
prima



# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

    codice

**except Classe\_eccezione as identificativo:**

    codice

**except Altraclasse\_eccezione as identificativo:**

    codice

**except:**

    codice

**else:**

    codice

**finally:**

    codice



viene eseguito in ogni caso

# Gestione delle eccezioni

In Python la sintassi per catturare un'eccezione è la seguente:

**try:**

codice

**except** `Classe_eccezione as identificativo`:

codice

**except** `Altraclasse_eccezione as identificativo`:

codice

**except:**

codice

**else:**

codice

**finally:**

codice

possono esserci più coppie  
classe identificativo se si vuole  
utilizzare lo stesso codice per  
gestirle

# Gestione delle eccezioni

All'interno di una funzione (o blocco) si sollevano le eccezioni con il comando **raise**

```
raise NameException(arg1,arg2)
```

# Decoratori

Si usano i decoratori per aggiungere funzionalità ad una funzione.

Es.

misurare il tempo di esecuzione di una funzione.

```
def func(x):
```

```
    time.sleep( random.random()*x )
```

```
start = time.time()
```

```
func(5)
```

```
end = time.time()
```

```
print ("ci ha messo ", end - start, " secondi")
```

# Decoratori

Se si voglio cronometrare tante funzioni diventa noioso e poco leggibile.

Possiamo definire una funzione per misure il tempo

```
def time_function(function):
```

```
    def new_function():
```

```
        start = time.time()
```

```
        value = function()
```

```
        end = time.time()
```

```
        print ("ci ha messo ", end - start, " secondi")
```

```
        return value
```

```
return new_function
```

```
f = time_function( func )  
f()
```

# Decoratori

La funzione però potrebbe necessitare di un parametro

```
def time_function(function):
```

```
    def new_function(x):
```

```
        start = time.time()
```

```
        value = function(x)
```

```
        end = time.time()
```

```
        print ("ci ha messo ", end - start, " secondi")
```

```
        return value
```

```
    return new_function
```

```
f = time_function( func )  
f(5)
```

# Decoratori

La funzione però potrebbe necessitare di un parametro... o più parametri

```
def time_function(function):
```

```
    def new_function(args*, kwargs**):
```

```
        start = time.time()
```

```
        value = function(args*, kwargs**)
```

```
        end = time.time()
```

```
        print ("ci ha messo ", end - start, " secondi")
```

```
        return value
```

```
return new_function
```

```
f = time_function( func )  
f(5,2)
```



# Decoratori

Con questa funzione copriamo tutti i casi... I decorator semplificano il suo utilizzo

Si definisce una funzione che prenda come parametro una funzione e restituisca una funzione

```
def time_function(function):
```

```
    def new_function(args*, kwargs**):
```

```
        start = time.time()
```

```
        value = function(args*, kwargs**)
```

```
        end = time.time()
```

```
        print ("ci ha messo ", end - start, " secondi")
```

```
        return value
```

```
return new_function
```

# Decoratori

Con questa funzione copriamo tutti i casi... I decorator semplificano il suo utilizzo

La si applica ad altre funzioni con la sintassi @

**@time\_function**

**def func4(x,y,z):**

**time.sleep( random.random()\*x+(y-z) )**

# Decoratori

A questo punto func4 non sarà la funzione definita dall'utente ma la funzione restituita da time\_function

Sarà una funzione arricchita

**@time\_function**

**def func4(x,y,z):**

**time.sleep( random.random()\*x+(y-z) )**