Programmazione avanzata a.a. 2023-24

A. De Bonis

Introduzione a Python (III parte)

23

Parametri keyword

- Sono argomenti di una funzione preceduti da un identificatore oppure passati come dizionario (**dict**) preceduto da **
- Un argomento keyword può essere specificato anche assegnando esplicitamente, attraverso il nome, un parametro attuale ad un parametro formale
- Nella definizione di una funzione i parametri keyword possono essere rappresentati dall'ultimo parametro della funzione preceduto da **
  - Il parametro è considerato un dizionario (**dict**)

24

L'operatore **

- L'operatore ** è il mapping unpacking operator e può essere applicato ai tipi mapping (collezione di coppie chiave-valore) quali i dizionari per produrre una lista di coppie chiave-valore adatta ad essere passata come argomento ad una funzione.

25

Esempio

Qui **cmd** è un dizionario

```
def esempio_kw(arg1, arg2, arg3, **cmd):
   if cmd.get('operatore') == '+':
      print('La somma degli argomenti è: ', arg1 + arg2 + arg3)
   elif cmd.get('operatore') == '*':
      print('Il prodotto degli argomenti è: ', arg1 * arg2 * arg3)
   else:
      print('operatore non supportato')

   if cmd.get('azione') == "stampa":
      print('arg1 =', arg1, 'arg2 =', arg2, 'arg3 =', arg3)
```


26

Esempio

```
esempio_kw(2, 3, 4, operatore='+')
```
La somma degli argomenti è: 9

```
esempio_kw(2, 3, 4, operatore='*')
```
Il prodotto degli argomenti è: 24

```
esempio_kw(2, 3, 4, operatore='/')
```
operatore non supportato

```
esempio_kw(2, 3, 4, operatore='+', azione='stampa')
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

```
esempio_kw(2, 3, 4, **{'operatore':'+', 'azione':'stampa'})
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

```
diz= {'operatore':'+', 'azione':'stampa'}
esempio_kw(2, 3, 4, **diz)
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

27

Esempio

Parametri variabili

Parametro keyword

```
def concat(*args, sep="/"):
    return sep.join(args)

print(concat('ciao','a','tutti', sep='/'))

print(concat('ciao','a','tutti', sep='.'))
```

→

```
ciao/a/tutti

ciao.a.tutti
```


28

Riassumendo

- Ci sono due modi per assegnare valori ai parametri formali di una funzione
- Secondo la posizione Parametri *tradizionali* Parametri *di default*
  - Gli argomenti posizionali non hanno keyword e devono essere assegnati per primi
  - La posizione è importante
- Secondo la keyword
  - Gli argomenti keyword hanno keyword e sono assegnati in seguito, dopo i parametri posizionali
  - La posizione non è importante
    - def f(x, a, b): ...
    - f('casa', a=3, b=7) è la stessa cosa di f('casa', b=7, a=3)

29

Riassumendo

- Una funzione può anche essere definita con tutti e tre i tipi di parametri
  - Parametri posizionali
    - Non inizializzati e di default
  - Parametro variabile
  - Parametri keyword

```
def tutti(arg1, arg2=222, *args, **kwargs):
    #Corpo della funzione
```


30

Esempio

```
def tutti(arg1, arg2=222, *args, **kwargs):
    print('arg1        =', arg1)
    print('arg2        =', arg2)
    print('*args       =', args)
    print('**kwargs =', kwargs)
```

```
tutti('prova', 999, 'uno', 2, 'tre', a=1, b='sette')
```

→

```
arg1       = prova
arg2       = 999
*args      = ('uno', 2, 'tre')
**kwargs = {'a': 1, 'b': 'sette'}
```

```
tutti('seconda prova')
```

→

```
arg1       = seconda prova
arg2       = 222
*args      = ()
**kwargs = {}
```


31

Annotazioni

- Le annotazioni sono dei metadati associati alle funzioni definite dal programmatore
- Sono memorizzate come un dizionario nell'attributo __annotation__ della funzione
- Non hanno nessun effetto sulla funzione
- Servono ad indicare il tipo dei parametri e del valore eventualmente restituito

32

Annotazioni

- L'annotazione di parametri è definita da **:** dopo il nome del parametro seguito da un'espressione che, una volta valutata, indica il tipo del valore dell'annotazione
- Le annotazioni di ritorno sono definite da **->** seguita da un'espressione e sono poste tra la lista dei parametri e i due punti che indicano la fine dell'istruzione def

33

Esempio

```
def saluta(nome: str, età: int = 23) -> str:
  print('Ciao ', nome, 'hai ', età, ' anni')
  return nome + ' ' + str(età)


s=saluta('mario')
print(s)
s=saluta('luisa', 21)
print(s)
```

```
Ciao  mario hai  23  anni
mario 23
Ciao  luisa hai  21  anni
luisa 21
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': <class 'str'>, 'return': <class 'str'>}
```


34

A cosa servono?

- Potrebbero essere utilizzate come help della funzione

```
def saluta(nome: 'rappresenta il nome dell\'utente ', età: int = 23) -> str:
  print('Ciao ', nome, 'hai ', età, ' anni')
  return nome + ' ' + str(età)
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': "rappresenta il nome dell'utente ", 'return': <class 'str'>}
```


35

Funzioni come parametro di funzioni

- È possibile passare l'identificatore di una funzione a come parametro di un'altra funzione b
  - Si passa il riferimento alla funzione a
- Nel corpo della funzione b, si può invocare a
  - Come nome della funzione si usa il parametro formale specificato nella definizione della funzione b

36

Esempio

```
def insertion_sort(a):
    for i in range(1,len(a)):
        val=a[i]
        j=i-1
        while (j>=0 and a[j]>val):
            a[j+1]=a[j]
            j=j-1
        a[j+1]=val
    return a
```

riferimento a funzione

```
def ordina(lista, metodo, copia=True):
    if copia == True:
        #si ordina una copia della lista
        return metodo(lista[:])
    else:
        return metodo(lista)
```

```
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = ordina(a, insertion_sort)
print('a =', a)
print('b =', b)
print('-----')
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = ordina(a, bubble_sort, copia=False)
print('a =', a)
print('b =', b)
```

```
a = [5, 3, 1, 7, 8, 2]

a = [5, 3, 1, 7, 8, 2]
b = [1, 2, 3, 5, 7, 8]
-----

a = [5, 3, 1, 7, 8, 2]

a = [1, 2, 3, 5, 7, 8]
b = [1, 2, 3, 5, 7, 8]
```


37

Espressioni **lambda**

- Funzioni anonime create usando la keyword **lambda**
- **lambda** a,b,c **:** a + b + c
  - Restituiscono la valutazione dell'espressione presente dopo i due punti
    - Può essere presente solo un'istruzione
  - Possono far riferimento a variabili presenti nello scope (ambiente) in cui sono definite
  - Possono essere restituite da funzioni
    - Una funzione che restituisce una funzione
  - Possono essere assegnate ad un identificatore
- Maggiori dettagli in seguito

38

Esempi

```
def f(x): return x**2
g = lambda x: x**2

print(g(3))
print(f(3))
```

→

```
9
9
```

f e g sono equivalenti, nel senso che producono lo stesso risultato

Ordina la lista dati considerando il valore assoluto degli elementi

```
dati = [1, -4, 2, 7, -10, -3]
print(dati)

dati.sort(key=lambda x: abs(x))
print(dati)
```

→

```
[1, -4, 2, 7, -10, -3]


[1, 2, -3, -4, 7, -10]
```


39

Funzioni Python built-in

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

40

Output: funzione print

- Riceve un numero variabile di parametri da stampare e due parametri keyword (end e sep)
- Aggiunge automaticamente \n alla fine dell'output
- Parametri keyword (opzionali)
  - sep - stringa di separazione dell'output (default spazio)
  - end - stringa finale dell'output (default \n)
- Gli argomenti ricevuti sono convertiti in stringhe, separati da sep e seguiti da end

41

Esempi

```
dati = [1, -4, 2, 7, -10, -3]
a = 1
b = 'a'
c = 'casa'
```

```
print(a, b, c, dati)
print(a, b, c)
print(dati)
print(a, b, c, dati, sep=':')
```

→

```
1 a casa [1, -4, 2, 7, -10, -3]
1 a casa
[1, -4, 2, 7, -10, -3]
1:a:casa:[1, -4, 2, 7, -10, -3]
```

```
for v in dati:
    print(v)
```

→

```
-4
2
7
-10
-3
```

```
for v in dati:
    print(v, end=' ')
```

→ `1 -4 2 7 -10 -3 1`

42

Output formattato

```
print('{} {}'.format('primo', 'secondo'))
print('{0} {1}'.format('primo', 'secondo'))
print('{1} {0}'.format('primo', 'secondo'))
print('{2} {0}'.format('primo', 'secondo', 'terzo'))
```

→

```
primo secondo
primo secondo
secondo primo
terzo primo
```


43

Output formattato

- Esempio di uso di format con parametri keywords

```
>>> d={"parola1":"ciao", "parola2":"?"}
>>> s="{parola1} Laura, come va {parola2}".format(**d)
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola1="ciao", parola2="?")
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola2="?", parola1="ciao")
>>> s
'ciao Laura, come va ?'
```


44

Output formattato

- Consultare
  - https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
- Oppure consultate il tutorial più immediato presso
  - https://pyformat.info/

45

Input: funzione input

- Riceve input da tastiera
- Può mostrare un cursore opzionale specificato come stringa
- Quello che viene letto è considerato stringa
  - Potrebbe dover essere convertito al tipo richiesto
- L'input termina con la pressione di invio (\n) che non viene inserito nella stringa letta

46

Esempi

```
a = input('Inserisci un valore: ')
print(a, type(a))
```

→

```
Inserisci un valore: e
e <class 'str'>
```

```
a = input('Inserisci un valore: ')
print(a, type(a))
```

→

```
Inserisci un valore: 12
12 <class 'str'>
```

```
a = int(input('Inserisci un valore: '))
print(a, type(a))
```

→

```
Inserisci un valore: 14
14 <class 'int'>
```


47

Lettura e scrittura di file

- La funzione built-in open() restituisce un file object che ci permette di agire sui file
- Comunemente open() è invocato con due argomenti:
  - open(filename,mode)
  - Esempio: p=open("file.txt","w")
- Il primo argomento filename è la stringa contenente il nome del file
- Il secondo argomento mode è una piccola stringa che indica in quale modalità deve essere aperto il file
  - 'r' : modalità di sola lettura
  - 'w' : modalità di sola scrittura; se il file non esiste lo crea; se il file già esiste il suo contenuto viene cancellato
  - 'a' : modalità di append; se il file non esiste lo crea; se il file già esiste il suo contenuto viene non cancellato
  - 'r+' : modalità di lettura e scrittura; il contenuto del file non viene cancellato
  - Se il secondo argomento non è specificato viene utilizzato il valore di default che è 'r'

48

Lettura e scrittura di file

Esempio: file.txt inizialmente vuoto

```
>>> fp=open("file.txt",'r+')
>>> fp.write("cominciamo a scrivere nel file")
30
>>> fp.write("\nvado al prossimo rigo")
22
```


49

Lettura e scrittura di file

- Possiamo usare close() per chiudere il file e liberare immediatamente qualsiasi risorsa di sistema usata per tenerlo aperto.
- Se il file non venisse chiuso esplicitamente, il garbage collector di Python ad un certo punto distruggerebbe il file object e chiuderebbe il file.
  - Ciò potrebbe avvenire però dopo molto tempo.
    - Dipende dall'implementazione di Python che stiamo utilizzando
- Dopo aver chiuso il file non è possible accedere in lettura o scrittura al file

50

Lettura e scrittura di file

Esempio (stesso file di prima)

```
>>> fp.close()
>>> fp.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```


51

Funzioni sui file

Calling Syntax	Description
fp.read( )	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next k bytes of a readable file as a string.
fp.readline( )	Return (remainder of) the current line of a readable file as a string.
fp.readlines( )	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell( )	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

52

Lettura e scrittura di file

Esempio:

```
>>> f=open("newfile",'w')
>>> f.write("prima linea\n")
12
>>> f.write("seconda linea\n")
14
>>> f.write("terza linea\n")
12
>>> f.write("quarta linea\n")
13
>>> f.close()
>>> f=open('newfile','r')
>>> for line in f:
...     print(line)
...
prima linea

seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```


Lettura e scrittura di file

Esempio: continua dalla slide precedente

```
>>> f.seek(0)
0
>>> f.readline()
'prima linea\n'
>>> for linea in f:
...         print(linea)
...
seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```

Gestione dei file

- Maggiori dettagli in
  - https://docs.python.org/3/library/filesys.html

55

Namespace

- Quando si utilizza un identificativo si attiva un processo chiamato risoluzione del nome (*name resolution*) per determinare il valore associato all'identificativo
- Quando si associa un valore ad un identificativo tale associazione è fatta all'interno di uno scope
- Il **namespace** (spazio dei nomi) gestisce tutti i nomi definiti in uno scope (ambito)

56

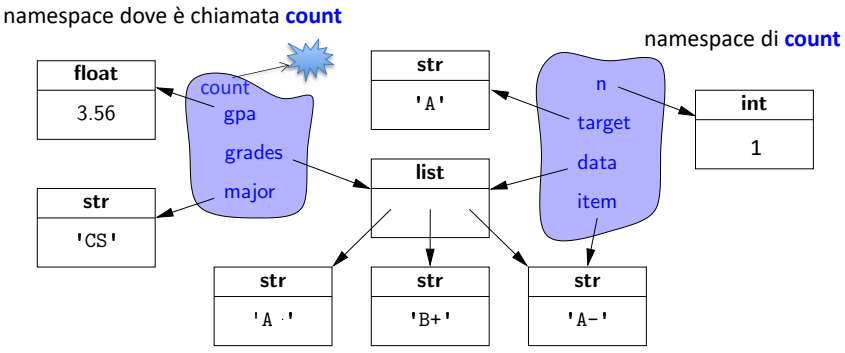
Namespace

- Python implementa il namespace tramite un dizionario che mappa ogni identificativo al suo valore
- Uno scope può contenere al suo interno altri scope
- Non c'è nessuna relazione tra due identificatori che hanno lo stesso nome in due namespace differenti
- Tramite le funzioni dir() e vars() si può conoscere il contenuto del namespace dove sono invocate
  - dir elenca gli identificatori nel namespace
  - vars visualizza tutto il dizionario

Esempio

```
grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'
count(grades, 'A')
```

```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n
```



Esempio

```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n

grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'

count(grades, 'A')

print(dir())
```

```
[
'__annotations__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__',
'__spec__', 'count', 'gpa', 'grades', 'major'
]
```

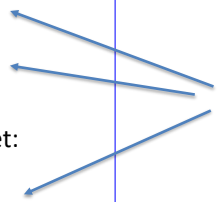

59

Esempio

```
def count(data, target):
    print(vars())
    n=0
    print(vars())
    for item in data:
        if item == target:
            n += 1
    print(vars())
    return n

grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'

count(grades, 'A')
```



aggiungiamo print(vars())) in count

```
{'data': ['A', 'B+', 'A-'], 'target': 'A'}
{'data': ['A', 'B+', 'A-'], 'target': 'A', 'n': 0}
{'data': ['A', 'B+', 'A-'], 'target': 'A', 'n': 1, 'item': 'A-'}
```


60

I moduli in Python

- Un modulo è un particolare script Python
  - È uno script che può essere utilizzato in un altro script
  - Uno script incluso in un altro script è chiamato modulo
- Sono utili per decomporre un programma di grande dimensione in più file, oppure per riutilizzare codice scritto precedentemente
  - Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) attraverso il comando import
  - Il nome di un modulo è il nome del file script (esclusa l'estensione '.py')
  - All'interno di un modulo si può accedere al suo nome tramite la variabile globale __name__

61

Moduli esistenti

- Esistono vari moduli già disponibili in Python
  - Alcuni utili moduli sono i seguenti

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

62

Utilizzare i moduli

- All'interno di un modulo/script si può accedere al nome del modulo/script tramite l'identificatore __name__
- Per utilizzare un modulo deve essere incluso tramite l'istruzione **import**
  - **import** math
- Per far riferimento ad una funzione del modulo importato bisogna far riferimento tramite il nome qualificato completamente
  - math.gcd(7,21)

63

Utilizzare i moduli

- Con l'istruzione from si possono importare singole funzioni a cui possiamo far riferimento direttamente con il loro nome
  - **from** math **import** sqrt
  - **from** math **import** sqrt, floor

```
import math
print(math.gcd(7,21))

from math import sqrt
print(sqrt(3))
```

→

```
7


1.7320508075688772
```

from math **import** * tutte le funzioni di math sono importate

64

Caricamento moduli

- Ogni volta che un modulo è caricato in uno script è eseguito
- Il modulo può contenere funzioni e codice *libero*
- Le funzioni sono *interpretate,* il codice libero è eseguito
- Lo script che importa (eventualmente) altri moduli ed è eseguito per primo è chiamato dall'interprete Python __main__
- Per evitare che del codice *libero* in un modulo sia eseguito quando il modulo è importato dobbiamo inserire un controllo nel modulo sul nome del modulo stesso. Se il nome del modulo è __main__ allora il codice libero è eseguito; altrimenti il codice non viene eseguito.

65

La variabile __name__

- Ogni volta che un modulo è importato, Python crea una variabile per il modulo chiamata __name__ e salva il nome del modulo in questa variabile.
- Il nome di un modulo è il nome del suo file .py senza l'estensione .py.
- Supponiamo di importare il modulo contenuto nel file test.py. La variabile __name__ per il modulo importato test ha valore "test".
- Supponiamo che il modulo test.py contenga del codice libero. Se prima di questo codice inseriamo il controllo **if** __name__ == **'__main__'**: allora il codice libero viene eseguito se e solo se __name__ ha valore __main__ . Di conseguenza, se importiamo il modulo test allora il suddetto codice libero non è eseguito.
- Ogni volta che un file .py è eseguito Python crea una variabile per il programma chiamata __name__ e pone il suo valore uguale a "__main__" . Di conseguenza se eseguiamo test.py come se fosse un programma allora il valore della sua variabile __name__ è __main__ e il codice libero dopo l'if viene eseguito.

66

Esempio

testNoIfMain.py

```
def modifica(lista):
    lista.append('nuovo')


lst = [1, 'due']
print('lista =', lst)
modifica(lst)
print('lista =', lst)
```

esecuzione testNoIfMain.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```

Stesso comportamento se eseguiti entrambi come programmi

test.py

```
def modifica(lista):
    lista.append('nuovo')

if __name__ == '__main__':
    lst = [1, 'due']
    print('lista =', lst)
    modifica(lst)
    print('lista =', lst)
```

esecuzione test.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```


67

Esempio

importUNO.py

```
import test
lista = [3,9]
print(lista)
test.modifica(lista)
print(lista)
```

esecuzione importUNO.py

```
[3, 9]
[3, 9, 'nuovo']
```

In questo caso l'if presente in test.py evita che vengano eseguite le linee di codice libero presenti in test.py

importDUE.py

```
import testNoIfMain
lista = [3,9]
print(lista)
testNoMain.modifica(lista)
print(lista)
```

esecuzione importDUE.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
[3, 9]
[3, 9, 'nuovo']
```

In questo caso vengono eseguite anche le linee di codice libero di testNoIfMain.py perché non sono precedute dall'if

68