

Introducció a la Programació
Grau en Estadística
Universitat Autònoma de
Barcelona

Sessió 5:
Tuples, diccionaris, conjunts,
Arrays n-Dimensionals i
Pas de Paràmetres a
Funcions

Vicenç Soler

Índex

1.	Introducció	1
2.	Tipus d'arrays	1
3.	Arrays n-dimensionals (matrius, etc.)	3
4.	Continuem amb les funcions.....	4
5.	Pas per de paràmetres per valor i per referència	9
6.	Solucions als exercicis proposats	16
7.	Apèndix.....	17

1. Introducció

En aquesta sessió es veuran altres tipus d'arrays, a part de les llistes, com són les tuples, diccionaris i conjunts. També es veuran, molt resumidament, com fer arrays de més d'una dimensió i els tipus de pas de paràmetres a funcions: per valor i per referència.

2. Tipus d'arrays

2.1. Llistes

Fins ara, hem estat treballant amb llistes. Les llistes permeten emmagatzemar un conjunt de dades, indexades per un nombre que va de 0 al tamany-1 .

Per a declarar-les i treballar amb elles, fem servir els **corchetes** “[...]” .

Exemple:

```
1. v=[10,20,30,40,50]    #declarem la llista amb “[...]”
2. v[0]=3                #modifiquem el primer valor
3. print(v)              #Mostraria [3,20,30,40,50]
4. print(v[:3])          #recordem que es pot demanar un subarray. Mostraria: [3,20,30]
```

2.2. Tuples

És un tipus de llista que **no és modificable**. És a dir, una vegada definida, no es pot canviar res, ni elements ni tamany.

La diferència sintàctica respecte les llistes és que la declaració es fa fent servir els **parèntesi** “(...)” en lloc dels corchetes. En canvi, per a agafar valors i treballar amb ells, també es fan servir els corchetes i els índexs també comencen per 0, com a les llistes.

Exemple:

```
1. t=(10,20,30,40,50)    #declarem la tupla amb “(...)”
2. t[0]=3              #no es pot fer, no podem canviar valors a una tupla
3. print(t[0])           #mostraria 10, indexem amb [...]
4. print(t)              #mostraria (10,20,30,40,50)
5. print(t[:3])          #recordem que es pot demanar un subarray. Mostraria: (10,20,30)
```

2.3. Diccionaris

És un tipus de llista que **permet indexar per qualsevol tipus de valor**, sense haver de ser enters que comencin al 0 com les llistes i les tuples, **incloent strings (texts)**.

Per exemple, imaginem que volem guardar el nombre d'estudiants >40 anys de cada facultat. En aquest cas, un diccionari ens ajudaria tenir clar l'índex de cada dada, podent assignar-li el nom de les facultats. A continuació veurem que es fa servir el símbol de les **claus** “{...}” per a definir el diccionari i també els corchetes “[...]” per a poder-hi indexar:

```

1. d={                                #Declarem el diccionari amb {...}
2.   "ciències":14,                  #El símbol ":" separa l'índex ("ciències") del valor (14), i els separem amb coma.
3.   "lletres":20
4. }
5. print(d["ciències"])              #mostraria 14, indexem amb [...]
6. d["lletres"]=21                   #canviem el valor de "lletres" de 20 a 21.
7. d["biociències"]=5               #Per a afegir nous elements, ho fem afegint un valor a un índex nou.
8. d.pop("lletres")                  #Per a eliminar elements, fem servir la funció "pop" (eliminem "lletres").
9. print(d)                          #Mostrarà {'ciències': 14, 'biociències': 5}
10. print(d["medicina"])              #Donarà ERROR, degut a què "medicina" no és un índex existent.
11. if "medicina" in d:               #L'error anterior es podria haver evitat si preguntéssim si és un índex.
12.     print(d["medicina"])
13. for i in d:                       #Quan "d" és un diccionari, la variable "i" conté els índexs: "ciències", "biociències",...
14.     print("index=",i," valor=",d[i])

```

2.4. Conjunts

Els conjunts són una simple **col·lecció de valors, sense cap índex**. Es fan servir per a guardar una sèrie de valors, on l'únic que vols fer és veure si un valor està en un conjunt o simplement passar-los per un 'for'.

Els conjunts poden variar de tamany (podem afegir i treure valors) i poden tenir valors repetits. També es fa servir el símbol de les **claus "{...}"** per a definir el diccionari i no farem servir els corchetes "[...]" ja no podem indexar-los:

Exemple:

```

1. c={10,20,30}
2. print(c)                          #mostra {10,20,30}
3. for x in c:
4.     print(x)                      #mostra els valors 10, 20 i 30
5. c.add(40)                          #afegeix el 40 al conjunt
6. c.remove(30)                       #treu el 30 del conjunt
7. print(c)                          #mostra {10,20,40}

```

2.5. Resum

Taula resum:

	Definició	Índex	Modificable	Afegir elements	Treure elements
Llista	[...]	0..tamany-1	Sí	append(), insert()	pop(), remove()
Tupla	(...)	0..tamany-1	NO	NO	NO
Diccionari	{...}	Qualsevol	Sí	v[índex]=valor	pop()
Conjunt	{...}	NO	Sí	add()	remove()

3. Arrays n-dimensionals (matrius, etc.)

Fins ara, hem vist com definir una llista (array) d'una sola dimensió:

`v=[None]*5`

Defineix un "array" (llista) de 5 posicions buides.

Si volem definir una matriu ("array" de 2 dimensions), haurem de fer que cada element d'una llista sigui una llista. Per exemple:

`m=[[1,2] , [3,4]]` equival a la matriu $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

on cada element de la llista principal és una fila i les 2 llistes internes defineixen les columnes de cada fila. Si volguéssim fer-la general, seria:

`m=[[None]*3]*5`

defineix una matriu de 5 **files** x 3 **columnes** amb valors buits.

Com en les llistes, els índexs van de 0 a tamany-1 . Així, per a guardar valors faríem:

`m[0][0]=3` #guarda el nombre 3 a la primera cel·la de la matriu m

`m[4][2]=-53` #guarda el nombre -53 a la darrera cel·la de la matriu

`m[2][1] =1` #guarda el nombre 1 a la 3ª fila, 2ª columna de la matriu

El resultat seria:

	0	1	2
0	3		
1			
2		1	
3			
4			-53

Si volguéssim declarar un array de 3 dimensions seria:

`matriu3D=[[[0]*2]*3]*5`

I així successivament.

Per a **afegir** i **treure** elements a una matriu, ho hem de fer com una llista normal. Per exemple:

`m.append ([10,20,])`

afegeix una fila nova al final de la matriu, amb els valors 10 i 20.

Nota: Si definiu una matriu 5x3 amb `[[0]*3]*5`, tenim un problema. Descubriu quin?

4. Continuem amb les funcions

4.1. Àmbit de les variables

Tant el programa principal com les funcions d'un programa contenen les seves pròpies variables. Anomenarem:

- **Variables locals:** a les variables pròpies d'una funció, inclosos els paràmetres que rep.
- **Variables globals:** a les variables del programa principal.

En aquest sentit, cada variable pertany al seu àmbit i no és visible des d'un altre àmbit. Així, les variables d'una funció només són visibles i accessibles des de la pròpia funció, i no des d'una altra, ni tampoc des del programa principal que, a la seva vegada, també té el seu propi àmbit de variables.

Recordem que a les funcions se li pot o no passar paràmetres i també poden o no retornar algun valor.

Exemples de variables locals:

```
1.  def funcioDeProva (p) :  
2.      a=p  
3.      b=2  
4.      c=(a+b)/2  
5.      return c  
6.  
7.  #programa principal  
8.  k=3  
9.  h=funcioDeProva(k)
```

La funció **funcioDeProva** té 3 variables locals:

- El paràmetre 'p': És una variable local (quan s'acaba la funció, es destrueix), i la diferència entre aquesta i les variables 'a' i 'b' és que el seu valor inicial el donem des del lloc on es crida (li passem el valor 3 des de la línia 9).
- Les variables 'a' i 'b': són variables locals de la funció. Quan acaba d'executar la funció, aquestes variables també es destrueixen.

Des del programa principal, no podem accedir a les variables 'p', 'a' i 'b'; i des de la funció no podem accedir a les variables del programa principal: 'k' i 'h'.

4.2. Treball amb funcions. Exemples d'àmbit de les variables

Exemple 1: Fer una funció que calculi l'àrea d'un rectangle passant-li els 2 costats.

Primer, el que hem de fer és definir la capçalera de la funció. Com hem de passar dos nombres amb decimals i n'hem de tornar un altre:

def areaRectangle (costat1, costat2) :

Recordeu que la funció comença amb "def" i les 2 variables entre parèntesi serveixen per a rebre els valors dels paràmetres que rep la funció.

Quan es crida la funció "areaRectangle", s'han de passar 2 valors i aquesta en retornarà 1 (p.ex., si cridéssiu la funció "log" per a calcular el logarisme, li passeu un valor i en retorna un altre),

i el seu codi seria:

```
1. def areaRectangle (costat1, costat2) :  
2.     area = costat1 * costat2  
3.     return area  
4.
```

Amb la instrucció "return" indiquem el valor que colem retornar. En aquest cas, el càlcul de l'àrea.

Nota: Pel tema que pertoca, que és l'explicació de l'àmbit de les variables, en el codi que hi ha just a sota s'ha afegit la variable local "factor" de la funció, que sempre val 1, i que no influeix en el càlcul final. Normalment, com el codi que hi ha just a dalt, no hi hauria de ser, però l'hem inclòs per a millorar l'explicació.

A continuació es mostra el codi, amb aquesta variable "factor" de més:

```
1. def areaRectangle (costat1, costat2) :  
2.     factor = 1          #Afegida per a millorar l'explicació de l'àmbit de les variables  
3.     area = costat1 * costat2 * factor    #Sempre factor=1 => no influeix en el càlcul  
4.     return area  
5.  
6. #Programa principal, que mostra la meitat de l'àrea d'un rectangle  
7. factor = 0.5           #només la fem servir per a calcular la meitat, al final  
8. base=float(input("Entra la base del rectangle:"))#entrem els 2 costats  
9. altura=float(input("Entra l'altura del rectangle:"))  
10.  
11. area = areaRectangle(base,altura)  
12. area = area * factor  
13. print("La meitat de l'àrea del rectangle és: ",area)
```

Recordem que la 1ª línia que el programa executa és la 7, demanant que entrem la base, ja que l'execució sempre comença pel programa principal.

Després, a la 11, cridem la funció amb els 2 valors entrats que ens retorna el resultat en la variable "area".

També recordem que cada funció té les seves variables, que s'anomenen variables locals. Les **variables locals** d'una funció es creen quan la cridem i es destrueixen quan sortim de la funció

(si arribem a un “return” quan es retorna un valor o bé si arribem al final del a funció quan no es retorna), i no són visibles entre les funcions. És a dir, una funció no pot veure les variables locals d’una altra.

En aquest cas, doncs, són diferents les variables “area” de “programa principal” i de “areaRectangle”. Veiem com s’executaria el programa:

Execució	variables	
1. El sistema inicia executant el programa principal i hi crea 3 variables locals (factor , base i altura), assignant valors entrant per teclat amb “input”.	<u>principal</u> factor=0.5 base= altura=	
2. S’entren els 2 valors. Imaginem que entren: base=3.2 i altura =2	<u>Principal</u> factor=0.5 base=3.2 altura=2	
3. Quan el programa arriba a la <u>línia 11</u> , crida la funció “ areaRectangle ”, li passa el contingut de les 2 variables: 3.2 i 2. De la <u>línia 11</u> , l’execució passa a la <u>línia 1</u> , on comencem l’execució de la funció “ areaRectangle ”.	<u>principal</u> factor=0.5 base=3.2 altura=2	<u>areaRec</u> costat1=3.2 costat2=2
4. Ara comença l’execució de la funció que, com el programa principal , comença creant les variables locals. En aquest cas en tindrem 3: 2 venen de paràmetres (costat1 i costat2) i una altra és la variable ‘ factor ’. Per tant, ara el sistema ha de mantenir 6 variables: <ul style="list-style-type: none"> - Programa principal: base, altura i factor. - areaRectangle: costat1, costat2 i factor. <p>Com cada funció té les seves variables locals, no hi ha cap problema en què 2 variables es diguin exactament igual. En aquest cas, tenim que 2 variables es diuen “factor” i ambdues guarden valors independents. A dins de “areaRectangle” la variable “factor” té valor 1 i a principal de 0.5 . Per tant, les 2 variables són diferents encara que tinguin el mateix nom.</p>	<u>principal</u> factor=0.5 base=3.2 altura=2	<u>areaRec</u> factor=1 costat1=3.2 costat2=2
5. A la <u>línia 3</u> creem la variable “ area ” de “ areaRectangle ” amb el producte d’ambdós costats i el factor ($3.2 \cdot 2 \cdot 1 = 6.4$) i el retornem.	<u>principal</u> factor=0.5 base=3.2 altura=2	<u>areaRec</u> factor=1 costat1=3.2 costat2=2 area=6.4
6. A l’executar el “ return ” de la <u>línia 4</u> , les 4 variables locals de la funció “ areaRectangle ” es destrueixen i continuem l’execució per la <u>línia 11</u> , d’on se l’havia cridat, per a guardar el valor retornat en la variable “ area ” del “programa principal”.	<u>principal</u> factor=0.5 base=3.2 altura=2 area=6.4	
7. Finalment mostra el valor resultat per pantalla, multiplicat pel factor 0.5.	<u>principal</u> factor=0.5 base=3.2 altura=2 area=6.4	

Nota: És important recalcar que , en Python, les funcions es declaren a l'inici del programa, ja que es necessiten tenir-les declarades abans de cridar-les des del programa principal.

Exemple 2: Un altre exemple de pas de paràmetres, calculant la mitja d'una llista.

En aquest exemple es cridaran a 2 funcions successivament i es veurà un exemple més clar de com funcionen les variables locals (l'àmbit de les variables). Veiem el següent exemple:

```
1. def suma (v) :  
2.     s = 0  
3.     for x in v:  
4.         s += x  
5.     return s  
6.  
7. def mitja(v) :  
8.     tamany = len(v)  
9.     s = suma(v)    #cridem a suma, a dalt  
10.    return s / tamany  
11.  
12. #Programa principal  
13. v=[10,20,30]  
14.  
15. m = mitja (v)  
16. print("La mitja aritmètica és: ", m)
```

Si ens fixem, el programa principal crida a '**mitja**' (línia 15) i, després, '**mitja**' crida a '**suma**' a la línia 9. Quan el programa principal crida a '**mitja**', passa la variable '**v**' que, a la seva vegada, '**mitja**' passa a '**suma**'.

Veiem el procés d'execució pas a pas:

Execució	Variables de cada àmbit		
1. El programa principal inicia l'execució per la <u>línia 13</u> , creant la llista ' v '.	<u>Principal</u> v=[10,20,30]		
2. A continuació crida a ' mitja ' a la <u>línia 15</u> .			
3. Continua l'execució per la <u>línia 7</u> , on crea l'espai de variables de la funció mitja i crea la 1ª variable de la funció, ' v ', que conté la llista que li passem des del programa principal.	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30]	
4. La <u>línia 8</u> crea una nova variable local de la funció, tamany , que és el resultat de la funció len definida a Python.	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30] tamany=3	
5. A la <u>línia 9</u> cridem a la funció suma , passant-li la llista, i l'execució continua per la <u>línia 1</u> , creant la variable local ' v ' de suma i assignant-li el valor que enviem des de mitja.	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30] tamany=3	<u>suma()</u> v=[10,20,30]
6. La <u>línia 2</u> crea i inicialitza la variable s=0. Important: Si aquí volguéssim fer servir la variable tamany de la funció mitja , donaria ERROR , ja que tamany només és visible dins la funció mitja , i no pas a la funció suma , que és on estem ara.	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30] tamany=3	<u>suma()</u> v=[10,20,30] s=0
7. Durant les línies 3 i 4 el ' for ' crea una nova variable ' x ' que rebrà els valors 10, 20 i 30	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30] tamany=3	<u>suma()</u> v=[10,20,30] s=0..10..30..60 x= 10..20..30
8. A l'executar el " return " de la <u>línia 5</u> , les variables locals de la funció suma es destrueixen i continuem l'execució per la <u>línia 9</u> , d'on se l'havia cridat, per a poder calcular la divisió i retornar la mitja.	<u>Principal</u> v=[10,20,30]	<u>mitja()</u> v=[10,20,30] tamany=3 s=60	
9. A l'executar el " return " de la <u>línia 10</u> , les variables locals de la funció mitja es destrueixen i continuem l'execució per la <u>línia 15</u> , d'on se l'havia cridat, per a guardar el valor retornat en la variable m del programa principal .	<u>Principal</u> v=[10,20,30] m=20		
10. Finalment mostra el valor resultat per pantalla: "La mitja aritmètica és: 20"	<u>Principal</u> v=[10,20,30] m=20		

Nota: Més endavant en aquest document, en el capítol de Pas per referència, es veurà que el pas de paràmetre d'una llista a una funció no funciona exactament com hem vist en aquest exemple. De totes maneres, l'exemple vist serveix per a il·lustrar i explicar el funcionament de les crides a funció i de l'àmbit de les variables, i el resultat del programa no té cap variació respecte a la realitat.

5. Pas per de paràmetres per valor i per referència

Quan passem paràmetres a una funció, ho podem fer de 2 maneres:

- Per valor
- Per referència

Nota: En els següents exemples s'ha escollit el mateix nom de variable al programa principal i a la funció, per a què es vegi que el nom de la variable no influeix en res, és a dir, encara que les 2 variables que es mostraran tenen el mateix nom, són 2 variables diferents, emmagatzemades en llocs de memòria diferents, ja que pertanyen a àmbits diferents: programa principal i funció.

5.1. Pas de paràmetres per valor

Aquest tipus és el que hem vist fins ara. Quan passem una dada per paràmetre a una funció, es passa el valor. Veiem un exemple senzill:

```

1. def funcio (n) :
2.     n=4
3.
4. #Programa principal
5. n=3           #El programa inicia la seva execució en aquesta línia
6. funcio(n)
7. print(n)
```

Què mostrarà el programa en la línia 7? 3 o 4?

Mostrarà el valor 3, ja que la '**n**' del **programa principal** i la '**n**' de la **funció** són variables diferents, ja que estan en àmbits diferents (zones de memòria diferents): la '**n**' de la línia 5 és una variable del programa principal i la '**n**' de la línia 2 és una variable local de la funció.

Veiem la seva execució:

Execució	Variables de cada àmbit	
1. El programa principal inicia l'execució per la <u>línia 5</u> , creant la variable ' n '.	Principal n=3	
2. Continua l'execució per la <u>línia 6</u> on crida a la funció que hem anomenat funcio		
3. Continua l'execució per la <u>línia 1</u> , on crea l'espai de variables de la funció funcio i crea la 1ª variable de la funció, ' n ', que conté el valor que li passem des del programa principal (3).	Principal n=3	<u>funcio()</u> n=3
4. La <u>línia 2</u> canvia el valor de la variable local de la funció. Veiem que aquest canvi no afecta a la variable ' n ' del programa principal, que continua valent 3.	Principal n=3	<u>funcio()</u> n=4

5. Després, acaba l'execució de la funció i, també, s'eliminen les seves variables locals i el fil de l'execució torna a la línia 6.	<u>Principal</u> n=3	
6. Finalment, s'executa la línia 7, que mostra el valor 3 per pantalla.	<u>Principal</u> n=3	

Com saber si un pas de paràmetre és per valor? En Python el pas de paràmetre per a variables simples (tipus numero enter i real, string, booleà, complex, etc.) són **SEMPRE PER VALOR**.

5.2.Pas de paràmetres per referència

Acabem de veure que en el pas de paràmetres per valor, si es canvia el valor d'una variable de la funció, no es veu afectada la variable d'origen (des d'on es crida la funció).

En canvi, en el pas per referència sí. Un exemple d'això és el pas d'un "array" (llista, diccionari, conjunt; a tupla no afecta ja que no es pot modificar). **Els "arrays" que es passen com a paràmetre a una funció són sempre passats per referència**, és a dir, si modifiquem una posició de l'array, l'array es veu modificat a l'origen. Veiem un exemple senzill:

Aquest tipus és el que hem vist fins ara. Quan passem una dada per paràmetre a una funció, es passa el valor. Veiem un exemple senzill:

```

1. def funcio2 (v) :
2.     v[0]=4
3.
4. #Programa principal
5. v=[1,2,3]           #El programa inicia la seva execució en aquesta línia
6. funcio2(v)
7. print(v)
```

Què mostrarà el programa en la línia 7? La llista de la línia 5 o la que conté un 4 a la 1a posició provocada per l'execució de la línia 2?

Mostrarà la llista [4,2,3], ja que la llista és passada per referència a "**funcio2**", implicant que qualsevol canvi en el contingut de la llista (valors, tamany, etc.) es veurà reflectit a la variable d'origen.

Per què canvia el valor en aquest cas? Doncs perquè la variable '**v**', quan s'envia a la funció, no s'envia la llista completa, sinó que s'envia només la referència, és a dir, l'adreça de memòria on està ubicada.

Aleshores, la '**v**' del programa principal i la '**v**' de la funcio2 són diferents ja que pertanyen a àmbits diferents? Sí, cadascuna ocupa un espai de memòria diferent, com en l'exemple anterior del pas per valor, però el que guarden aquestes variables no són els valors, sinó l'adreça de memòria on comença l'array.

Veiem la seva execució pas a pas:

Execució	Variables de cada àmbit	
1. El programa principal inicia l'execució per la <u>línia 5</u> , creant la variable ' v '.	Principal v=[1,2,3]	
2. Continua l'execució per la <u>línia 6</u> on crida a la funció que hem anomenat funcio2		
3. Continua l'execució per la <u>línia 1</u> , on crea l'espai de variables de la funció funcio2 i crea la 1ª variable de la funció ' v ', que conté l'adreça de memòria on està ubicada la llista.	Principal v=[1,2,3]	<u>funcio2()</u> v=v(ppal)
4. La <u>línia 2</u> canvia el primer valor de la llista a 4. Veiem que aquest canvi SÍ afecta a la llista que havíem creat al programa principal, ja que el valor del primer element canvia a 4.	Principal v=[4,2,3]	<u>funcio2()</u> v=v(ppal)
5. Després acaba l'execució de la funció i, també, s'eliminen les seves variables locals (en aquest cas la ' v ' que guardava l'adreça de memòria de la llista), i el fil de l'execució torna a la <u>línia 6</u> .	Principal v=[4,2,3]	
6. Finalment, s'executa la <u>línia 7</u> , que mostra la llista [4,2,3] per pantalla.	Principal v=[4,2,3]	

Mentalment¹, hem d'entendre un array (llista, en aquest cas) con una sèrie de valors que estan en posicions de memòria contigües. Així, quan enviem una llista a una funció, no cal enviar-li totes les dades, ja que només necessitem l'adreça de memòria on comença, ja que a partir d'aquesta podem accedir a tots els valors, pel fet que són posicions de memòria consecutives. Això fa que:

- **Estalviem temps i memòria.** Suposem que passem una llista gegantina amb 1 milió de valors. Si només passem l'adreça on comença és molt ràpid i eficient (només hem de passar 1 dada), però si passem tota la llista sencera, a la funció de destí s'hauria de crear una altra vegada tota la llista i copiar-hi el milió de valors, cosa que faria gastar molt de temps i espai de memòria (tornar a reservar 1 milió de cel·les de memòria i copiar-hi les dades no és immediat ni l'espai de memòria per a treballar és il·limitat), i ser del tot ineficient.
- **Ens permet modificar valors** a dins de les funcions i que els canvis es vegin reflectits en origen. Aquesta funcionalitat permetria passar un array a una funció i que aquesta l'omplís de valors.

¹ Nota important: Les explicacions que es faran de com està organitzada la memòria (adreces, llistes, etc.) no és exactament així, i depèn de cada llenguatge de programació, però l'explicació plantejada aquí serveix per a entendre com funciona aquest mecanisme en qualsevol llenguatge de programació que us trobeu en el futur.

Podem veure la distribució de les variables a memòria en els següents gràfics:

Pas per valor	Pas per referència
<pre> 1. def funcio (n) : 2. n=4 3. 4. #Programa principal 5. n=3 6. funcio(n) 7. print(n </pre>	<pre> 1. def funcio2 (v) : 2. v[0]=4 3. 4. #Programa principal 5. v=[1,2,3] 6. funcio2(v) 7. print(v) </pre>
<p>Exemple d'organització a memòria :</p>	<p>Exemple d'organització a memòria:</p>

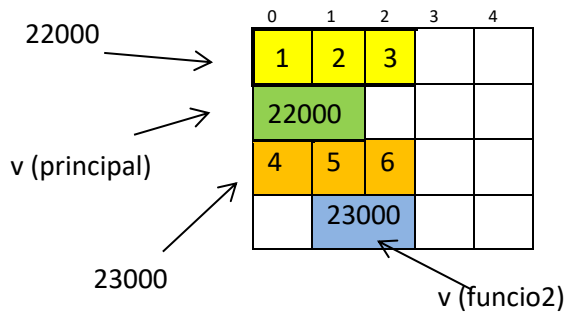
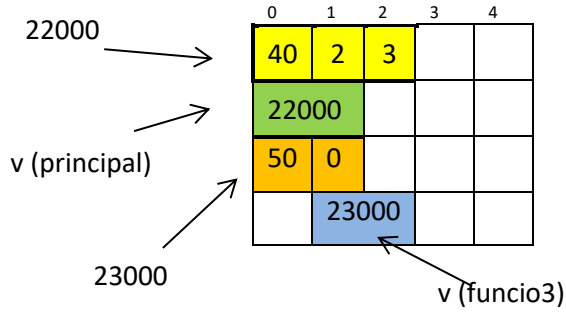
En el cas de **pas per valor**, les 2 variables 'n' es guarden en 2 caselles de memòria diferents.

En el cas de **pas per referència**, les 2 variables 'v' també es guarden en 2 caselles de memòria diferents. Suposem que l'adreça on es guarda la llista és la **22000** (com podria ser qualsevol altra). Aleshores, la variable 'v' del programa principal guardarà l'adreça **22000**, que és on està la llista. Quan es cridi a la funció **funcio2**, passarà com a paràmetre el contingut de la variable 'v' (l'adreça **22000**), com abans hem passat el contingut de la variable 'n' (és a dir, 3).

Com saber si un pas de paràmetres és per referència? En Python, el pas de paràmetre per a variables amb continguts de més d'un valor: tipus arrays (l·listes, tuples, diccionaris, conjunts) i objectes (els veurem més endavant) són **SEMPRE PER REFERÈNCIA**.

5.3.Consideracions sobre el pas per referència.

Hem d'anar amb compte, perquè el pas per referència funciona només si mantenim l'adreça de memòria d'origen. En els següents exemples, el resultat obtingut no és l'esperat, ja que canviem l'adreça de memòria d'origen:

<pre> 1. def funcio2 (v) : 2. v=[4,5,6] 3. 4. #Programa principal 5. v=[1,2,3] 6. funcio2(v) 7. print(v) 8. </pre>	<p>En aquest cas mostrarà per pantalla [1,2,3].</p> <p>Suposem que l'adreça de memòria de [1,2,3] del programa principal és la 22000. Quan es crida a funcio2 passem l'adreça 22000, però la <u>línia 2</u>, el que fa, és crear una altra llista nova amb [4,5,6] en una altra adreça de memòria (suposem la 23000) i emmagatzemar-la a la 'v' local de la funció, que ha canviat de valor (del 22000 al 23000). En canvi, la 'v' del programa principal continua valent 22000.</p> <p>Quan funcio2 acaba, mostra per pantalla el contingut de la llista que està a la 22000, ja que és l'adreça que encara guarda la variable 'v' del programa principal.</p> <p>Exemple d'organització a memòria:</p>  <p>A l'acabar la funcio2, les caselles blaves i taronja es destrueixen.</p>
<pre> 1. def funcio3 (v) : 2. v[0]=40 3. v=[0]*2 4. v[0]=50 5. 6. #Programa principal 7. v=[1,2,3] 8. funcio3(v) 9. print(v) </pre>	<p>En aquest cas mostrarà per pantalla [40,2,3].</p> <p>Suposem, també, que l'adreça de memòria de [1,2,3] del programa principal és la 22000. Quan es crida a funcio3 passem l'adreça 22000.</p> <p><u>Línia 2</u>: 'v' de funcio3 encara val 22000 i la 1ª posició canvia a 40.</p> <p><u>Línia 3</u>: Es crea una nova llista (anem a suposar que a la 23000) de tamany 2, que omple amb 0's. Ara 'v' de la funcio3 ha canviat el valor de la 22000 a la 23000.</p> <p><u>Línia 4</u>: Modifiquem la posició 0 de la nova llista a 50. Ara la nova llista (que està a la 23000) té valor [50,0].</p> <p>Quan torna de la funció funcio3, mostra la 'v', que està a la 22000, i que conté [40,2,3].</p> <p>Exemple d'organització a memòria:</p>  <p>A l'acabar la funcio3, les caselles blaves i taronja es destrueixen</p>

```
1. def funcio4 (v) :  
2.     v=v[:2] #subllista  
3.     v[0]=27  
4.  
5. #Programa principal  
6. v=[1,2,3]  
7. funcio4(v)  
8. print(v)
```

En aquest cas mostrarà per pantalla **[1,2,3]**.

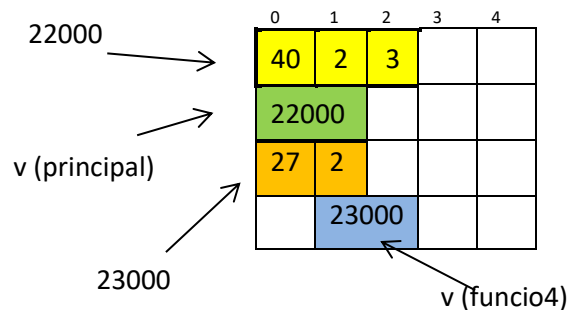
Suposem, també, que l'adreça de memòria de [1,2,3] del programa principal és la 22000. Quan es crida a **funcio4** passem l'adreça 22000.

Línia 2: Quan es demana una subllista, es crea una nova llista amb el tamany especificat i els valors corresponents. Com abans, anem a suposar que la nova llista es crea a l'adreça 23000 i, en aquest cas, tindrà tamany 2 i valors [1,2]. Ara '**v**' de la **funcio4** ha canviat el valor de la 22000 a la 23000.

Línia 3: Modifiquem la posició 0 de la nova llista a 27. Ara, la nova llista (que està a la 23000) té valor [27,2].

Quan torna de la funció **funcio4**, mostra la '**v**', que està a la 22000, i que conté el valor original [1,2,3], ja que no l'hem modificat.

Exemple d'organització a memòria:



A l'acabar la **funcio4**, les caselles blaves i taronja es destrueixen

Per què els índexs de les llistes comencen per un 0 i no per un 1?

Quan executem:

- $v[2]=34$, el que fa el Python (i tots els llenguatges de programació) és anar a la casella "adreça que guarda v " + 2 i guardar-hi el 34 (en aquest cas seria a la 22002).
- Si fem $v[0]=4$, guardarà un 4 a l' "adreça que guarda v " + 0 (l'adreça 22000).

Ara comparem què es necessitaria per a canviar el valor de 3 a 56 de la 3^a casella de l'array de l'exemple anterior, de color groc (faríem $v[2]=56$):

- Com els índexs comencen per 0, per a calcular l'adreça de memòria o guardarà el 56 farà: "adreça que guarda v " + 2 (l'adreça 22002).
- Si els índexs comencessin per 1, executaríem $v[3]=56$ (ja que és la 3^a posició de la llista) i, per a calcular l'adreça de memòria, faríem: "adreça que guarda v " + 3 - 1 (la 22002 també).

Com veiem, si sempre comencessin per 1, qualsevol feina que féssim amb llistes hauríem d'estar restant sempre 1 al càlcul, feina extra que és ineficient.

Per això en molts llenguatges de programació els índexs dels arrays comencen per 0.

Podem escollir quan volem passar per valor o per referència?

En Python no. Depèn del tipus de variable que es passi per paràmetre, aquesta passarà per valor o per referència. Recordem-ho:

Per valor	Per referència
Enters	Llistes
Float	Tuples
Complex	Conjunts
Bool : True i False	Diccionaris
String: Cadenes de text	Objectes

Exercici proposat 1: Repàs. Fer una funció per a invertir les posicions d'un array. Per exemple, si un array de 5 posicions té els valors 5,3,6,8,1 la funció els hauria de col·locar en ordre invers: 1,8,6,3,5 .

6. Solucions als exercicis proposats

Exercici proposat 1: Repàs. Fer una funció per a invertir les posicions d'un array. Per exemple, si un array de 5 posicions té els valors 5,3,6,8,1 la funció els hauria de col·locar en ordre invers: 1,8,6,3,5 .

```
1. def invertirLlista (llista) :  
2.     cant_elem=len(llista)  
3.     for i in range(cant_elem//2) :  
4.         aux=llista[i]  
5.         llista [i]= llista [cant_elem-1-i]  
6.         llista [cant_elem-1-i]=aux  
7.
```

Apunts:

- Com el pas de l'array és per referència, no cal retornar res, ja que la pròpia llista que es passa, contindrà la llista invertida resultant.
- En aquest cas hem decidit guardar el tamany en la variable "cant_elem". Crec que així el codi queda una mica més net respecta a cridar massa vegades a len(..), i també serà més ràpid ja que així no hem de calcular len(llista) a cada pas del for.
- Fixeu-vos com ho fem: intercanviem les posicions 0 i darrera, després 1 i darrera-1, etc. (si "cant_elem"=5, intercanviem 0 amb 4, 1 amb 3 i 2 amb 2, que és no fer res). Per a poder intercanviar el valor de 2 variables necessitem una tercera, "temp", anomenada així per què el se ús és temporal, només com a auxiliar de l'operació que fem. Primer ens guardem el valor de la posició "i" a temp. Una vegada el tenim, ja podem canviar el valor de la posició "i" i posar la desitjada ("cant_elem-1-i"). Una vegada ja ho he posat, ja podem recuperar el valor que havíem guardat a "temp" i posar-lo a la posició intercanviada.

Proveu l'execució pas a pas, en paper, posant vosaltres mateixos el valor de cada variable en cada pas i veureu que realment funciona.

- Finalment, només ens queda discernir que realment l'algorisme funciona per a llistes de tamany senar (ho acabem de comprovar) i parell. Proveu-la amb una llista de tamany parell.

7. Apèndix

Altres exemples de pas de paràmetres a funcions.

Pas de paràmetres per valor

Posem que tenim una funció on volem calcular l'àrea d'un rectangle i com la cridem:

```

1. def areaRectangle (costat1, costat2) :
2.     area = costat1 * costat2
3.     return area
4.
5. #Programa principal
6. base=4
7. altura=5
8. area = areaRectangle(base,altura)
9. print(area)

```

Recordem, ara, com treballa internament l'ordinador, i les variables que fa anar:

Execució	Variables existents	
1. Comença l'execució per la <u>línia 5</u> .		
2. A les línies 5 i 6, crea espai a memòria per a guardar 2 variables (locals del programa principal), i posa valor a 2 d'elles per a poder calcular l'àrea més fàcilment.	principal: base=4 altura=5	
3. A la <u>línia 7</u> crida la funció areaRectangle, passant-li els valors de les variables base i altura.	principal: base=4 altura=5	
4. L'execució es trasllada a la <u>línia 1</u> . El sistema crea 2 variables locals (costat1, costat2) agafant els valors passats en la <u>línia 7</u> .	principal: base=4 altura=5	areaRect: costat1=4 costat2=5
5. La <u>línia 2</u> crea una altra variable local, i li assigna el valor del producte dels 2 costats.	principal: base=4 altura=5	areaRect: costat1=4 costat2=5 area=20
6. Finalment, es retornaria el valor de l'àrea i s'assignaria a la variable "area" del programa principal, i, en sortir de la funció "areaRectangle", es destrueixen totes les seves variables locals.	principal: base=4 altura=5 area=20	

En aquest cas, el pas de paràmetres ha estat per valor, ja que hem traslladat el valor quan hem fet el pas de paràmetres.

Si féssim un canvi a la funció "areaRectangle" i modifiquéssim el valor d'un dels costats:

```

1. def areaRectangle (costat1, costat2):
2.     costat1 =35
3.     area = costat1 * costat2
4.     return area
5.
6. #Programa principal
7. base=4
8. altura=5
9. area = areaRectangle(base,altura)
10. print(area)

```

Afectaria al resultat, ja que no importa el que el programa principal passi com a primer paràmetre, ja que sempre considerarà costat1 amb valor 35. Però com el pas de paràmetre és per valor, el fet de canviar de valor la variable “costat1” de “areaRectangle”, no canvia el valor de la variable origen, que era “base” del programa principal.

En aquest segon cas, l'execució seria:

Execució	Variables existents	
1. Comença l'execució per la <u>línia 5</u> .		
2. A les línies 7 i 8, crea espai a memòria per a guardar 2 variables (locals del programa principal), i posa valor a 2 d'elles per a poder calcular l'àrea més fàcilment.	principal: base=4 altura=5	
3. A la <u>línia 9</u> crida la funció areaRectangle, passant-li els valors de les variables base i altura.	principal: base=4 altura=5	
4. L'execució es trasllada a la <u>línia 1</u> . El sistema crea 2 variables locals (costat1, costat2) agafant els valors passats en la <u>línia 7</u> .	principal: base=4 altura=5	areaRect: costat1=4 costat2=5
5. La <u>línia 2</u> assigna el valor 35 a costat1, però no canvia la variable “base” del programa principal, ja que són 2 variables diferents en espais de memòria diferents	principal: base=4 altura=5	areaRect: costat1=35 costat2=5
6. La <u>línia 3</u> crea una altra variable local, i li assigna el valor del producte dels 2 costats.	principal: base=4 altura=5	areaRect: costat1=35 costat2=5 area=175
7. Finalment, es retornaria el valor de l'àrea i s'assignaria a la variable “area” del programa principal, i, en sortir de la funció “areaRectangle”, es destrueixen totes les seves variables locals.	principal: base=4 altura=5 area=175	

Pas de paràmetres per referència

Acabem de veure que en el pas de paràmetres per valor, si es canvia el valor d'una variable de la funció, no es veu afectada la variable d'origen (des d'on es crida la funció).

En canvi, en el pas per referència sí. Un exemple d'això, és el pas d'un "array". Els "arrays" que es passen com a paràmetre a una funció són sempre passats per referència, és a dir, si modifiquem una posició de l'array, l'array es veu modificat a l'origen. Veiem un exemple senzill:

```
1. def f(arr):
2.     arr[5] = 35
3.
4.
5. #programa principal
6. v=[None]*10
7.
8. #omplim l'array amb el valor 20 a totes les posicions
9. for i in range(len(v)):
10.     v[i] = 20
11.
12. #cridem la funció "f" que modifica un dels valors a 35
13. f(v)
14.
15. #mostrem l'array final
16. print(v)
17.
```

En aquest exemple, el programa simplement omple amb el valor 20 les 10 posicions de l'array "v", després crida la funció "f" que modifica el valor de l'índex 5 a 35. Finalment, veiem que l'array d'origen "v" es veu modificat en el seu contingut, i ho comprovem mostrant el seu contingut per pantalla. Això no passaria si, en lloc de passar per paràmetre un "array" fos, per exemple, una variable "int".