CURS 6
Fişiere text

Un *fişier text* este o secvenţă de caractere organizată pe linii şi stocată în memoria externă (SSD, HDD, DVD, CD etc.). Practic, fişierele text reprezintă o modalitate foarte simplă prin care se poate asigura *persistenţa datelor*.

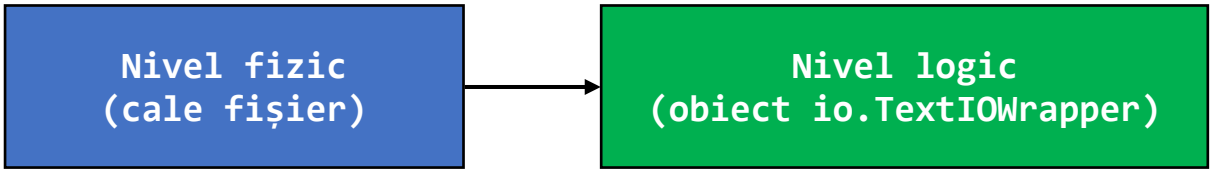
Liniile dintr-un fişier text sunt demarcate folosind caracterele **CR** (*carriage return* sau `'\r'` sau `chr(13)`) şi **LF** (*line feed* sau `'\n'` sau `chr(10)`), astfel:

- în sistemele de operare *Windows* şi *MS-DOS* se utilizează combinaţia **CR+LF**;
- în sistemele de operare de tip *Unix/Linux* şi începând cu versiunea *Mac OS X* se utilizează doar caracterul **LF**;
- în sistemele de operare până la versiunea *Mac OS X* se utilizează doar caracterul **CR**.

Indiferent de modalitatea utilizată pentru demarcarea liniilor într-un anumit sistem de operare, în momentul în care se citesc linii dintr-un fişier text, caracterele utilizate pentru demarcare vor fi transformate automat într-un singur caracter `LF ('\n')`!

Deschiderea unui fişier text

Operaţia de deschidere a unui fişier (text sau binar!) presupune asocierea fişierului, considerat la nivel fizic (i.e., identificat prin calea sa), cu o variabilă/un obiect dintr-un program, ceea ce va permite manipularea sa la nivel logic (i.e., prin intermediul unor funcţii sau metode dedicate):



Practic, după deschiderea unui fişier, obiectul asociat fişierului va conţine mai multe informaţii despre starea fişierul respectiv (dimensiunea în octeţi, modalitatea de codificare utilizată, poziţia curentă a pointerului de fişier etc.), iar programatorul va putea acţiona asupra fişierului într-un mod transparent, folosind metodele puse la dispoziţie de obiectul respectiv.

Deschiderea unui fişier se realizează folosind următoarea funcţie (reamintim faptul că parantezele drepte indică faptul că parametrul este opţional):

```
open("cale fişier", ["mod de deschidere"])
```

Primul parametru al funcţiei reprezintă calea fişierului, sub forma unui şir de caractere, iar în cazul în care fişierul se află în directorul curent se poate indica doar numele său.
Modul de deschidere este utilizat pentru a preciza cum va fi prelucrat fişierul text şi se indică prin următoarele litere:

- **`"r"`** pentru *citire din fișier* (`read`)
  - fișierul va putea fi utilizat doar pentru a citi date din el;
  - este modul implicit de deschidere a unui fișier text, i.e. se va folosi dacă se omite parametrul `"mod de deschidere"` în apelul funcției `open`;
  - dacă fișierul indicat prin calea respectivă nu există, atunci se va genera eroarea `FileNotFoundError`;

- **`"w"`** pentru *scriere în fișier* (`write`)
  - fișierul va putea fi utilizat doar pentru a scrie date în el;
  - dacă fișierul indicat prin calea respectivă nu există, atunci se va crea unul nou, altfel fișierul existent va fi șters automat și va fi înlocuit cu unul nou;

- **`"x"`** pentru *scriere în fișier creat în mod exclusiv* (`exclusive write`)
  - fișierul va putea fi utilizat doar pentru a scrie date în el;
  - dacă fișierul indicat prin calea respectivă nu există, atunci se va crea unul nou, altfel se va genera eroarea `FileExistsError`;

- **`"a"`** pentru *adăugare în fișier* (`append`)
  - fișierul va putea fi utilizat doar pentru a scrie informații în el;
  - dacă fișierul există, atunci noile datele se vor scrie imediat după ultimul său caracter, altfel se va crea un fișier nou și datele se vor scrie de la începutul său.

Pentru a trata excepțiile care pot să apară în momentul deschiderii unui fișier, se va folosi o instrucțiune de tipul `try...except`, așa cum se poate observa în exemplele următoare:

```
try:
    f = open("exemplu.txt")
    print("Fișier deschis cu succes!")
except FileNotFoundError:
    print("Fișier inexistent!")
```

```
try:
    f = open("C:\Test Python\exemplu.txt", "x")
except FileNotFoundError:
    print("Calea fișierului este greșită!")
except FileExistsError:
    print("Fișierul deja există!!")
```

Din cel de-al doilea exemplu se poate observa faptul că eroarea `FileNotFoundError` va fi generată și în cazul modurilor de deschidere pentru scriere (i.e., modurile `"w"`, `"x"` și `"a"`) în cazul în care calea fișierului este greșită!

Citirea datelor dintr-un fişier text

În limbajul Python, datele citite dintr-un fişier text vor fi furnizate întotdeauna sub forma unor şiruri de caractere!

Pentru citirea datelor dintr-un fişier text sunt definite următoarele metode:

- **`read()`**: furnizează tot conţinutul fişierului text într-un singur şir de caractere

  Exemplu:

Program	Fişier de intrare	Ecran

`f = open("exemplu.txt")`
`s = f.read()`
`print(s)`
`f.close()`
	`Ana are multe pere,`
	
	`mere rosii,`
	`si mai multe prune!!!`
		`Ana are multe pere,`
		
		`mere rosii,`
		`si mai multe prune!!!`

  Observaţi faptul că şirul `s` conţine şi caracterele LF (`'\n'`) folosite pentru delimitarea liniilor!

- **`readline()`**: furnizează conţinutul liniei curente din fişierul text într-un şir de caractere sau un şir vid când se ajunge la sfârşitul fişierului

  Exemplu:

Program	Fişier de intrare	Ecran

`f = open("exemplu.txt")`
`s = f.readline()`
`while s != "":`
`  print(s, end="")`
`  s = f.readline()`
`f.close()`
	`Ana are multe pere,`
	
	`mere rosii,`
	`si mai multe prune!!!`
		`Ana are multe pere,`
		
		`mere rosii,`
		`si mai multe prune!!!`

  Observaţi faptul că, de fiecare dată, şirul `s` conţine şi caracterul LF (`'\n'`) folosit pentru delimitarea liniilor, mai puţin în cazul ultimei linii!

  O modalitate echivalentă de parcurgere a unui fişier text linie cu linie constă în iterarea directă a obiectului `f` asociat:

  Exemplu:

Program	Fişier de intrare	Ecran

`f = open("exemplu.txt")`
`for linie in f:`
`  print(linie, end="")`
`f.close()`
	`Ana are multe pere,`
	
	`mere rosii,`
	`si mai multe prune!!!`
		`Ana are multe pere,`
		
		`mere rosii,`
		`si mai multe prune!!!`

- **`readlines()`**: furnizează toate liniile din fişierul text într-o listă de şiruri de caractere

Exemplu:

Program	Fişier de intrare	Ecran

`f = open("exemplu.txt")`
`s = f.readlines()`
`print(s)`
`f.close()`
	`Ana are multe pere,`
	
	`mere rosii,`
	`si mai multe prune!!!`
		`['Ana are multe pere,\n', '\n', 'mere rosii,\n', 'si mai multe prune!!!']`

Observaţi faptul că fiecare şir de caractere din listă, corespunzător unei linii din fişier, conţine şi caracterul LF (`'\n'`) folosit pentru delimitarea liniilor, mai puţin în cazul ultimei linii!

Cele 3 modalităţi de citire a datelor dintr-un fişier text sunt echivalente, dar, în cazul în care dimensiunea fişierului este mare, se preferă citirea linie cu linie a conţinutului său, deoarece necesită mai puţină memorie.

Deoarece toate metodele folosite pentru citirea datelor dintr-un fişier text furnizează şiruri de caractere, dacă vrem să citim datele respective sub forma unor numere trebuie să utilizăm funcţii pentru manipularea şirurilor de caractere.

Exemplu: Fişierul text *numere.txt* conţine, pe mai multe linii, numere întregi separate între ele prin spaţii. Scrieţi un program care afişează pe ecran suma numerelor din fişier.

Soluţia 1 (folosind metoda `read`):
Construim o listă `numere` care să conţină numerele din fişier, împărţind şirul corespunzător întregului conţinut al fişierului în subşiruri cu metoda `split` şi transformând fiecare subşir într-un număr cu metoda `str`:

```
f = open("numere.txt")
numere = [int(x) for x in f.read().split()]
print("Suma numerelor din fisier:", sum(numere))
f.close()
```

Soluţia 2 (folosind metoda `readline`):
Construim tot o listă care să conţină toate numerele din fişier, dar împărţind în subşiruri doar şirul corespunzător liniei curente:

```
f = open("numere.txt")
numere = []
linie = f.readline()
while linie != "":
    numere.extend([int(x) for x in linie.split()])
    linie = f.readline()
print("Suma numerelor din fisier:", sum(numere))
f.close()
```

Evident, o soluție mai eficientă din punct de vedere al memoriei utilizate constă în calculul sumei numerelor de pe fiecare linie și adunarea sa la suma tuturor numerelor din fișier:

```
f = open("numere.txt")
total = 0
linie = f.readline()
while linie != "":
    numere = [int(x) for x in linie.split()]
    total = total + sum(numere)
    linie = f.readline()
print("Suma numerelor din fisier:", total)
f.close()
```

Soluția 3 (folosind metoda `readlines`):
Vom proceda într-un mod asemănător cu soluția 2, dar parcurgând lista cu liniile fișierului furnizată de metoda `readlines`:

```
f = open("numere.txt")
total = 0
for linie in f.readlines():
    numere = [int(x) for x in linie.split()]
    total = total + sum(numere)
print("Suma numerelor din fisier:", total)
f.close()
```

Atenție, în oricare dintre soluțiile prezentate, metoda split trebuie apelată fără parametri, pentru a împărți șirul respectiv în subșiruri considerând toate spațiile albe (care includ caracterele '\n' și '\r')! Altfel, dacă am apela metoda split doar cu parametrul " " (un spațiu) ar fi generată eroarea `ValueError` în momentul în care am încerca să transformăm în număr ultimul subșir de pe o linie (mai puțin ultima din fișier), deoarece acesta ar conține și caracterul '\n'!

Scrierea datelor într-un fișier text

În limbajul Python, într-un fișier text se pot scrie doar șiruri de caractere. Pentru scrierea datelor într-un fișier text sunt definite următoarele metode:

- **`write(șir)`**: scrie șirul respectiv în fișier, fără a adăuga automat o linie nouă

Exemplu:

Program	Fișier de ieșire

`f = open("exemplu.txt", "w")`
`cuvinte = ["Ana", "are", "mere!"]`
`for cuv in cuvinte:`
`  f.write(cuv)`
`f.close()`
	`Anaaremere!`

Dacă dorim să scriem fiecare cuvânt pe o linie nouă, atunci trebuie să modificăm `f.write(cuv)`în `f.write(cuv + "\n")`.

- **`writelines(colecție de șiruri)`**: scrie toate șirurile din colecție în fișier, fără a adăuga automat linii noi între ele

Exemplu:

Program	Fișier de ieșire

`f = open("exemplu.txt", "w")`
`cuvinte = ["Ana", "are", "mere!"]`
`f.writelines(cuvinte)`
`f.close()`
	`Anaaremere!`

Dacă dorim să scriem fiecare cuvânt pe o linie nouă, atunci fie trebuie să adăugăm câte un caracter `"\n"` la sfârșitul fiecărui cuvânt din listă (de exemplu, prin instrucțiunea `cuvinte = [cuv + "\n" for cuv in cuvinte]`), fie să modificăm `f.writelines(cuvinte)` în `f.writelines("\n".join(cuvinte))`, deci să concatenăm toate șirurile din listă într-unul singur, folosind caracterul `"\n"` ca separator (evident, în acest caz putem utiliza și metoda `write`).

Închiderea unui fișier text

Indiferent de modalitatea în care a fost deschis un fișier și, implicit, accesat conținutul său, acesta trebuie închis după terminarea prelucrării sale, folosind metoda `close()`. Închiderea unui fișier constă în golirea eventualei zone tampon (buffer) alocate fișierului și ștergerea din memorie a obiectului asociat cu el. Dacă un fișier deschis într-unul dintre modurile de scriere nu este închis, atunci există riscul ca ultimele informații scrise în fișier să nu fie salvate! În mod implicit, un fișier este închis dacă referința obiectului asociat este atribuită unui alt fișier (i.e., se utilizează, din nou, funcția `open`).

Putem elimina necesitatea închiderii explicite a unui fișier putem folosi o instrucțiune `with...as`, astfel:

```
with open("exemplu.txt") as f:
    s = f.readlines()
    print(f.closed)     #False

print(f.closed)         #True
print(s)
```

În acest caz, fișierul va fi închis automat imediat după ce se va termina prelucrarea sa, așa cum se poate observa din exemplul de mai sus (data membră `closed` permite testarea stării curente a unui fișier, respectiv dacă acesta a fost închis sau nu). Mai mult, închiderea fișierului respectiv se va efectua corect chiar și în cazul apariției unei erori în timpul prelucrării sale. Atenție, chiar dacă se utilizează o instrucțiune

`with...as`, erorile care pot să apară în momentul deschiderii unui fişier trebuie să fie tratate explicit!

În încheiere, vom prezenta un exemplu în care vom utiliza toate cele 3 moduri de deschidere ale unui fişier text.

Mai întâi, vom implementa un program care să genereze un fişier text care va conţine pe prima linie o sumă formată din n numere naturale aleatorii (de exemplu, `12 + 300 + 9 + 1234`):

```
import random

numef = input("Numele fisierului: ")
n = int(input("Numarul de valori aleatorii: "))

with open(numef, "w") as f:
    # initializam generatorul de numere aleatorii
    random.seed()
    for k in range(n - 1):
        # generam un  numar aleatoriu
        # cuprins intre 1 si 1000
        x = random.randint(1, 1000)
        f.write(str(x) + " + ")
    f.write(str(random.randint(1, 1000)))
```

În continuare, vom implementa un program care să calculeze suma numerelor din fişier şi apoi să o scrie la sfârşitul primei linii, după un semn "=" (de exemplu, `12 + 300 + 9 + 1234 = 1555`):

```
numef = input("Numele fisierului: ")

# deschidem fisierul pentru citire si
# calculam suma numerelor de pe prima linie
f = open(numef, "r")
nr = [int(x) for x in f.readline().split("+")]
s = sum(nr)

# deschidem fisierul pentru adaugare
# si scriem suma la sfarsitul primei linii

# fisierul deschis anterior pentru citire
# va fi inchis implicit
f = open(numef, "a")
f.write(" = " + str(s))
# inchidem explicit fisierul
f.close()
```

Funcţii

O *funcţie* este un set de instrucţiuni, grupate sub un nume unic, care efectuează o prelucrare specifică a unor date în momentul în care este apelată.

Funcţiile sunt utilizate pentru a modulariza codul dintr-un program, prin împărţirea sa în subprograme (i.e., funcţii) care execută fiecare o singură prelucrare bine definită. În plus, funcţiile permit o reutilizare simplă şi sigură a codului, prin organizarea lor în module şi pachete.

În limbajul Python sunt predefinite mai multe funcţii care sunt des utilizate în programe, cum ar fi funcţii de conversie (`int`, `str`, `bool` etc.), funcţii matematice (`min`, `max`, `sum` etc.), funcţii pentru crearea unei colecţii (`list`, `dict`, `set` etc.) ş.a.m.d. O listă completă a tuturor funcţiilor predefinite din limbajul Python poate fi consultată aici: https://docs.python.org/3/library/functions.html.

O funcţie poate fi definită de către un utilizator folosind cuvântul cheie `def`, astfel:

```
def nume_funcție(parametrii formali)
      instrucțiuni
```

Parametrii formali sunt utilizaţi în antetul unei funcţii pentru a preciza datele sale de intrare într-un mod abstract (formal). *Parametrii actuali* ai unei funcţii sunt expresii ale căror valori sunt asociate parametrilor formali în momentul apelării unei funcţii.

Parametrii formali ai unei funcţii pot fi de mai multe tipuri în limbajul Python:

- *parametri simpli* au, de obicei, un caracter poziţional, respectiv în momentul apelării unei funcţiei trebuie să fie înlocuiţi cu acelaşi număr de parametrii efectivi, compatibili ca tipuri de date:

```
def suma(x, y):
    return x + y


# apelare pozițională
s = suma(3, 7)
print("s =", s)

# apelare prin nume
s = suma(y=7, x=3)
print("s =", s)
```

  În acest exemplu, parametrii formali ai funcţiei `suma` sunt `x` şi `y`, iar constantele `3` şi `7` sunt parametrii efectivi. Deoarece în limbajul Python nu se precizează explicit tipurile de date ale parametrilor formali, compatibilitatea dintre ei şi parametrii efectivi poate

fi, în unele cazuri, o noțiune destul de vagă. Astfel, în acest exemplu, parametrii efectivi ai funcției pot fi și două șiruri de caractere sau două liste, dar nu pot fi, de exemplu, un număr și un șir de caractere! De asemenea, se observă faptul că parametrii simpli permit și apelarea prin nume.

- *parametri cu valori implicite* sunt utilizați dacă se dorește inițializarea lor cu niște valori implicite, care vor fi utilizate în momentul apelării funcției cu un număr de parametrii efectivi mai mic decât numărul parametrilor formali:

```
def suma(x=0, y=0):
    return x + y


s = suma()              # x = 0 și y = 0
print("s =", s)         # s = 0

s = suma(7)             # x = 7 și y = 0
print("s =", s)         # s = 7

s = suma(y=19)          # x = 0 și y = 19
print("s =", s)         # s = 19

s = suma(7, 10)         # x = 7 și y = 10
print("s =", s)         # s = 17
```

Atenție, dacă antetul unei funcții conține și parametri simpli și parametri cu valori implicite, parametrii poziționali trebuie să fie precizați primii, pentru a evita ambiguitățile care pot să apară în momentul apelării funcției! De exemplu, dacă funcția se mai sus ar avea antetul `suma(x=0, y)`, atunci apelul `s = suma(7)` ar putea fi interpretat fie ca `s = suma(7, y)` și ar fi incorect (parametrului `y` nu i-ar fi asociat niciun parametru efectiv), fie ca `s = suma(0, 7)` și ar fi corect. În schimb, antetul `suma(x, y=0)` nu va mai genera nicio ambiguitate în cazul apelului `s = suma(7)`, putând fi interpretat doar ca `s = suma(7, 0)`.

În limbajul Python, se pot defini foarte simplu și funcții cu *număr variabil de parametri*, care sunt utile când nu se poate preciza numărul exact de parametrii ai unei funcții. De exemplu, în același program putem să avem nevoie de o funcție care să calculeze suma a două numere și de o funcție care să calculeze suma a trei numere. Deoarece în limbajul Python nu se pot defini două funcții cu același nume și număr diferit de parametri (de fapt, se pot defini, dar compilatorul o va lua în considerare doar pe ultima definită și orice apelare a primei funcții definite va fi semnalată ca eroare), înseamnă că ar trebui să definim două funcții cu nume diferite, dar care sunt foarte asemănătoare din punct de vedere al prelucrărilor efectuate.

Definirea unei funcții cu număr variabil de parametri se realizează adăugând simbolul * înaintea unui parametru, ceea ce indică faptul că parametrul respectiv va conține, sub forma unui tuplu, un număr oarecare de parametri efectivi. De exemplu, o funcție cu număr variabil de parametri care calculează suma acestora se poate defini și apela astfel:

```
def suma(*args):
    sa = 0
    for x in args:
        sa = sa + x
    return sa
s = suma()
print("s =", s)          #s = 0

s = suma(7)              #s = 7
print("s =", s)

s = suma(1, 2, 3, 4)     #s = 10
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista)
print("s =", s)          #s = 55
```

Evident, o funcție poate să aibă un singur parametru variabil, iar eventualii parametrii existenți după el trebuie să fie precizați explicit, prin numele lor, în momentul apelării funcției (deoarece ei nu mai pot fi accesați pozițional, fiind precedați de un număr necunoscut de parametrii). De exemplu, următoarea funcție calculează suma parametrilor variabili care sunt cel puțini egali cu valoarea `minim`:

```
def suma(*valori, minim):
    sv = 0
    for x in valori:
        if x >= minim:
            sv = sv + x
    return sv


s = suma(7, minim=5)             #s = 7
print("s =", s)

s = suma(1, 2, 3, minim=10)      #s = 0
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista, minim=8)
print("s =", s)                  #s = 27
```

Dacă în exemplul de mai sus nu vom preciza explicit valoarea parametrului `minim` (de exemplu, scriind `s = suma(1, 2, 3, 10)`), atunci se va semnala o eroare de tipul `TypeError` (pentru exemplul considerat aceasta va fi `TypeError: suma() missing 1 required keyword-only argument: 'minim'`).

Bineînțeles, dacă o funcție cu număr variabil de parametri are și parametri simpli și parametri cu valori implicite, atunci trebuie respectată și regula ca parametrii simpli să

fie scrişi înaintea celor cu valori implicite! De exemplu, următoarea funcţie `suma` calculează suma parametrilor variabili cuprinşi între valorile `minim` şi `maxim` care sunt şi multiplii ai numărului `t`:

```
def suma(t, *valori, minim=0, maxim=100):
    sv = 0
    for x in valori:
        if minim <= x <= maxim and x % t == 0:
            sv = sv + x
    return sv


s = suma(7, 14, 19, 21, -56)                          #s = 35
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100)              #s = -21
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100, maxim=0)     #s = -56
print("s =", s)
```

Datorită operaţiei implicite de împachetare a mai multor valori sub forma unui tuplu, o funcţie poate să furnizeze mai multe valori. Astfel, o instrucţiune de forma `return a, b` este echivalentă cu `return (a, b)`, aşa cum se poate observa din următorul exemplu:

```
def suma_prod(x, y):
    return x + y, x * y


s, p = suma_prod(3, 7)
print("s =", s, "\tp =", p)

(s, p) = suma_prod(3, 7)
print("s =", s, "\tp =", p)

t = suma_prod(3, 7)
print("s =", t[0], "\tp =", t[1])
print("Suma si produsul:", *t)
```

Observaţi faptul că la apelarea funcţiei se poate utiliza operaţia de despachetare a unui tuplu, complementară celei de împachetare utilizată în instrucţiunea `return`!

Modalităţi de transmitere a parametrilor

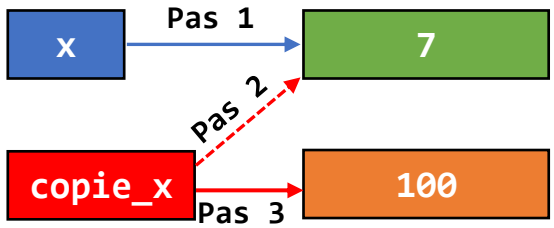
În limbajele C/C++ transmiterea unui parametru efectiv către o funcţie se poate realiza în două moduri:

- *transmitere prin valoare*: se transmite o copie a valorii parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcţiei NU se reflectă şi în exteriorul său;
- *transmitere prin adresă/referinţă*: se transmite adresa parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcţiei se reflectă şi în exteriorul său.

În limbajul Python, orice parametru efectiv al unei funcţii este o referinţă către un obiect (i.e., nu se transmite obiectul în sine, ci o referinţă spre el) care se transmite implicit prin valoare (i.e., se transmite o copie a referinţei respective), deci modificarea referinţei respective în interiorul funcţiei nu se va reflecta în exteriorul său. Din acest motiv, mecanismul de transmitere a parametrilor către o funcţie în limbajul Python se numeşte *transmitere prin referinţă la obiect* (*call by object reference*).

Exemplul 1:

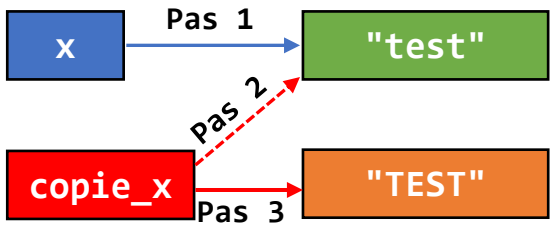
Funcţia	Executarea programului	Rezultat

`def functie(t):`
`  t = 100  # Pas 3`
 	
`x = 7  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x = 7`

Explicaţie: După pasul 2, variabilele `x` şi `copie_x` vor conţine aceeaşi referinţă (spre obiectul 7), dar după pasul 3 doar `copie_x` se va modifica (va conţine referinţa obiectului 100), deoarece `x` este o referinţă transmisă prin valoare.

Exemplul 2:

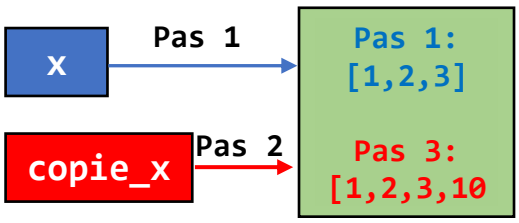
Funcţia	Executarea programului	Rezultat

`def functie(t):`
`  t = t.upper() # Pas 3`
 	
`x = "test"  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x = test`

Explicaţie: vezi exemplul anterior!

Exemplul 3:

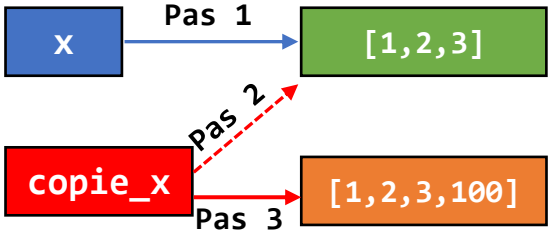
Funcția	Executarea programului	Rezultat

`def functie(t):`
`  t.append(100) # Pas 3`
 	
`x = [1, 2, 3]  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x =`
		`[1,2,3,100]`

Explicație: După pasul 2, variabilele x și `copie_x` vor conține aceeași referință, iar după pasul 3 conținutul listei se va modifica prin intermediul referinței din `copie_x` (însă fără a modifica referința listei!), deci variabila x va putea accesa lista modificată.

Exemplul 4:

Funcția	Executarea programului	Rezultat

`def functie(t):`
`  t = t + [100] # Pas 3`
 	
`x = [1, 2, 3]  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x =`
		`[1,2,3]`

Explicație: După pasul 2, variabilele x și `copie_x` vor conține aceeași referință (spre lista [1,2,3]), dar la pasul 3 operatorul de concatenare (+) va crea o nouă listă [1,2,3,100], deci după pasul 3 doar `copie_x` va conține referința noii liste. Practic, deși o listă este mutabilă, elementul 100 a fost adăugat într-o manieră specifică obiectelor imutabile!

În concluzie, mecanismul de *transmitere prin referință la obiect* a parametrilor efectivi către o funcție în limbajul Python utilizează transmiterea prin valoare a referințelor parametrilor efectivi (copii ale referințelor lor, deci modificarea acestora nu va fi vizibilă în exteriorul funcției) și acționează astfel:

- dacă obiectul asociat referinței este *imutabil*, atunci modificarea parametrului respectiv în interiorul funcției nu se va reflecta în exteriorul funcției deoarece conținutului obiectului asociat referinței nu poate fi modificat (din cauza imutabilității), iar crearea unei referințe noi nu se va reflecta în exteriorul funcției (din cauza transmiterii prin valoare a referinței);
- dacă obiectul asociat referinței este *mutabil*, atunci modificarea conținutului parametrului respectiv în interiorul funcției se va reflecta în exteriorul funcției deoarece nu se va modifica referința obiectului.

Atenție la exemplul 4 de mai sus, deoarece acolo nu se modifică direct conținutul obiectului mutabil de tip listă, ci se creează un nou obiect (tot o listă mutabilă) care conține noua listă și se atribuie referința sa copiei referinței parametrului!