Capitolul 5. Functii, exceptii, lucrul cu fisiere

5.1 Functii in Python

5.1.1 Necesitatea utilizarii functiilor

Ati observat ca am putut realiza programe simple, compuse din cateva linii de cod, fara a utiliza functii definite de utilizator (am utilizat totusi functii predefinite cum sunt print, del, dir, len, etc.).

Pentru a scrie un program complex, cu multe linii de cod, avem nevoie de functii, cu ajutorul carora vom evita scrierea repetata a aceluiasi cod, apeland doar functiile care contin codul de care avem nevoie in mod repetat (reutilizare). Asadar, unul din avantajele functiilor este reducerea codului duplicat. Acest lucru are si un alt avantaj, putem gestiona mai usor codul, orice modificare fiind facuta intr-un singur loc, in functia respectiva.

Unul din principiile programarii este descompunerea unei probleme complexe in probleme mai mici, care sunt mai usor de rezolvat si prin asamblarea acestora rezolvam problema principala (modularizare). Functiile ne ajuta in aceasta constructie.

Imbunatatirea calitatii si claritatii codului este un alt beneficiu adus de functii.

Incapsularea, este o metoda care permite utilizatorului sa se concentreze pe ceea ce face un instrument, in cazul nostru functia si sa nu-l interezeze ce are in interior, cum este construita. Asupra acestui subiect vom reveni in capitolul urmator cand vom discuta despre clase.

5.1.2 Definitie, sintaxa, docstring

Definitie: functia este o portiune de cod stocata, care are un nume, poate primi date de intrare si poate furniza date de iesire.

Sintaxa:

```
>>> def nume_functie( [parametri] ) :
        """Docstring"""
        bloc de instructiuni
        [return expresie]
```

Cuvantul cheie def este cel cu care incepe definirea unei functii, urmat de nume, eventualii parametri, intre paranteze rotunde si caracterul "**:**". Acestea se constituie in prima linie a unei functii (header).

Docstringul este optional. Acesta va fi cuprins intre ghilimele triple si va fi ignorat de catre interpretorul Python. Docstringul ofera informatii despre ceea ce face functia. Doctringul unei functii poate fi vizualizat cu ajutorul instructiunii **nume_functie.__doc__.**

Blocul de instructiuni poate fi compus dintr-o singura instructiune sau mai multe instructiuni. Putem pune **pass** in loc de blocul de instructiuni, caz in care functia nu va rula nimic. Acest procedeu poate fi util cand schitam componentele principale ale programului, pana la scrierea codului, permitandu-ne sa testam celelate functionalitati.

Return, urmat de o expresie, face parte din blocul de instructiuni si este o modalitate de a oferi date de iesire, specifica functiilor. O functie fara return va putea genera un rezultat temporar, care insa nu va putea fi stocat intr-o variabila si nu va putea fi prelucrat.

Indentare : atat docstringul, cat si blocul de instructiuni vor fi indentate fata de prima linie a functiei.

Sintaxa utilizata la definierea functiei trebuie respectata, in caz contrar va fi generata o eroare.

Toate aceste elemente **definesc functia, fara sa o si ruleze**.

Definirea functiilor se va face, de regula, la inceputul programului, imediat dupa variabile. Exista cel putin doua argumente pentru acest lucru:

- O functie trebuie definita inainte de utilizare, altfel va genera o eroare. Fiind definita la inceputul programului ne asiguram ca nu vom fi in aceasta situatie;
- Este mult mai usor sa identificam si se depanam eventualele erori aparute la functii stiind ca sunt intr-un singur loc, decat sa le cautam prin tot codul.

Exemplu, definim o functie simpla, care printeaza un mesaj:

```
>>> def end_page( ) :
        """marcheaza sfarsitul paginii"""
        return "#-----#----#----#----#----#----#".center(80)
>>>print(end_page( ))                # apelarea functiei produce efecte

                #-----#----#----#----#----#----#
```

5.1.3 Denumirea functiilor

Numele functiilor respecta regulile de denumire ale variabilelor.

Nu pot fi utilizate pentru denumirea functiilor cuvintele rezervate sau numele functiilor predefinite. Numele vor fi cat mai sugestive pentru ceea ce au de facut.

La crearea unei functii cu acelasi nume functia precedenta va fi stearsa (rescriere).

Numele unei functii nu trebuie sa fie identic cu cel al unei variabile globale, deoarece variabila va suprascrie functia si reciproc. Putem in schimb sa dam acelasi nume cu al unei variabile locale, cu care nu intra in conflict.

Putem avea o functie cu mai multe nume. Mai mult, daca stergem functia creata initial, cea redenumita functioneaza in continuare.

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

adunare = aduna
aduna( 5, 8 )
13
adunare( 7, 8 )
15
del (aduna)
aduna( 3, 9 )
Traceback (most recent call last):
File "<input>", line 1, in <module>
NameError: name 'aduna' is not defined
adunare( 3, 9 )
12
```

5.1.4 Parametrii si apelarea functiilor

Asa cum am mentionat anterior, definirea functiei se face fara rularea acesteia. Apelarea functiei are ca rezultat rularea ei.

Apelarea functiei se face cu numele acesteia urmat de paranteze rotunde. Daca functia necesita parametrii acestia vor fi pusi in paranteze.

```
>>> print ( end_page() )
'                    #-----#----#----#----#----#----#                    '
```

Parametrii unei functii pot fi siruri de caractere, numere sau Boolean. Acestia pot fi, de asemenea, variabile. De regula functia nu schimba valoarea variabilelor primite ca parametri.

Parametrii functiei sunt optionali la definirea functiei. In functia definita mai sus nu avem parametri.

Parametrii trebuie sa fie adecvati pentru instructiunile ce compun blocul de instructiuni (tip de date si numarul lor). Ordinea parametrilor conteaza. Daca am definit mai multi parametri, la apelarea functiei vor fi mentionati in aceeasi ordine in care sunt definiti. In caz contrar vor aparea erori la apelare.

```
>>>def test_parametri(nume, varsta):
        if type ( nume ) is str and type ( varsta ) is int:
            print ( nume, 'are', varsta, 'ani!' )
        else:
            print ( 'Parametri incorecti!' )

>>>test_parametri ( 'Maria', 34 )      # apelare corecta a functie
Maria are 34 ani!

>>>test_parametri ( 34, 'Maria' )      # parametrii inversati, tipuri de date diferite
Parametri incorecti!

>>>test_parametri ( )                  # numar incorect de parametrii
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: test_parametri() missing 2 required positional arguments: 'nume' and
'varsta'
```

Putem atribui si valori default parametrilor, caz in care ordinea nu mai conteaza. In aceasta situatie, in lipsa parametrilor la apelare vor fi preluate valorile default si functia va fi rulata cu acesti parametri.

```
>>>personal = dict ( )

>>>def angajat(nume='Nume', anul_ang=2016):
        """Completeaza datele angajatilor in 'personal' """
        personal[nume] = anul_ang

>>>angajat ( 'Mitica', 2001 )          # corect

>>>angajat ( 'Corina' )                # corect, va prelua anul default

>>>angajat ( 2005 )          # incorect, numele va fi considerat 2005, anul default 2016

>>>angajat ( nume = 'Micsunica', anul_ang = 2009 )          # corect0

>>>angajat ( anul_ang = 2010 )         # corect, numele va fi default 'Nume'

>>>angajat ( nume='Mihai' )            # corect, anul va fi default 2016

>>>angajat ( anul_ang = 2002, nume = 'Constanta' )    # corect, nu conteaza ordinea


>>>print ( personal )
{'Constanta': 2002, 'Mihai': 2016, 2005: 2016, 'Nume': 2010, 'Micsunica': 2009,
'Mitica': 2001, 'Corina': 2016}
```

Parametrii functiei NU sunt vizibili in exteriorul functiei (incapsulare);

```
>>>def calculeaza(a, b, c):
        d = (a + b) * c          # d este variabila locala, nu este vizibila in afara functiei
        return d                 # la fel si parametrii a, b, c

>>>calculeaza ( 1, 2, 3 )
9

>>>print ( d )                          # NU sunt vizibili in afara functiei
Traceback (most recent call last):
```

```
  File "<input>", line 5, in <module>
NameError: name 'd' is not defined
```

5.1.5 Domeniul de vizibilitate (namespace)

Namespace face referire la locul unde sunt declarate variabilele sau create functiile. Locul in care sunt create determina domeniul de vizibilitate al acestora si este in stransa legatura cu durata de viata (scope). Exista un namespace numit ***built in***, aferent functiilor predefinite, cu vizibilitate in tot codul, un ***namespace global*** in care sunt definite variabile si functiile, care au vizibilitate, de asemenea, in tot codul si un ***namespace local*** in care sunt definite variabilele locale (de exemplu in interiorul unei functii). Acestea din urma au vizibilitate doar la rularea functiei in cauza.

Existenta acestor ***namaspace diferite*** ajuta la gestionarea optima a memoriei. Eliberarea memoriei de variabilele locale duce la cresterea rapiditatii incarcarii si rularii programelor.

Variabilele globale pot fi incapsulate (functii, clase). In mod normal acestea nu-si pot modifica valoarea in timpul apelarii (rularii) functiei. Daca dorim sa le modificam valoarea, in interiorul acestora, va trebui sa le declaram globale (**global nume_variabila**).

```
>>>def test_global(numar):
        global x                          # declaram variabila globala x
        x = numar + 1
        return x

>>>test_global ( 100 )
101
>>>print ( x )
101
```

In Python au prioritate variabilele definite in namespace-ul cel mai apropiat. In acest sens, intr-o functie, o variabila locala, care are acelasi nume ca al unei variabile globale, va avea prioritate.

```
>>>glo = 100              # definim o variabila globala

>>>def test_loc():        # scriem o functie cu o var locala cu nume identic cu al celei globale
       glo = 5            # variabila locala are prioritate fata de cea globala
      print ( glo )

>>>test_loc ( )
5
>>>print ( glo )          # valoarea variabilei globale ramane neschimbata
100
```

In concluzie, o functie poate utiliza atat variabilele din namespace-ul propriu (local) cat si pe cele din namespace-ul global. In timpul rularii functiei poate fi modificata valoarea variabilelor locale si, in mod exceptional valoarea variabilelor globale, doar daca sunt declarate global.

5.2 Gestionarea erorilor. Try, except, else

5.2.1 Try, except, else

Python are un un system de returnare a erorilor foarte bine pus la punct. O eroare va intrerupe rularea programului

Daca dorim ca aceste erori sa fie suprimate, in special cand utilizatorul introduce anumite date, putem s-o facem indicand o cale de urmat, fara intreruperea programului. Pentru aceasta folosim ***try / except***;

Sintaxa este de forma:

```
try:                                    # Incearca operatiunea data...
    bloc de instructiuni

except[(tip_exceptie1, tip_exceptie 2,...)][var] :
                                        # Daca nu reuseste are alternativa
```

bloc de instructiuni

else: *# Daca reuseste executa si instructiunile else*
bloc de instructiuni

finally: *# Ruleaza in orice situatie*
bloc de instructiuni

Blocul de instructiuni de sub except va rula **doar daca exista o eroare** in rularea blocului de instructiuni aferente lui try.

Daca avem si else, instructiunile de sub else ruleaza **doar daca nu apar erori** (impreuna cu cele aferente try).

Daca nu mentionam tipul de eroare (erori), orice eroare va fi suprimata. Tipurile de erori posibile sunt prezentate in tabelul de mai jos:

Tipul exceptiei	**Descriere**
IOError	Daca incercam sa deschidem un fisier inexistent
IndexError	Daca indexam cu un numar inexistent
KeyError	Daca nu exista cheia in dictionary
NameError	Nume de functie sau variabila inexistent
SyntaxError	Eroare de sintaxa
TypeError	Functie aplicata la un type neadecvat
ValueError	Functie aplicata cu o valoare gresita
ZeroDivisionError	Impartirea la zero nu este posibila

In cazul in care dorim ca doar anumite tipuri de erori sa fie suprimate le vom mentiona explicit intre paranteze rotunde, ca argumente ale except. Optional putem mentiona si o variabila in care sa capturam erorile aparute.

De asemenea, putem folosi mai multe ramuri except, pentru tipuri diferite de eroare, fiecare cu propriul bloc de instructiuni aplicabile.

In toate situatiile, daca apare alt tip de eroare, decat cele definite in cadrul except, nu va fi exceptata.

5.3 Functia prescurtata lambda

5.3.1 Functia prescurtata lambda

In python putem scrie si o asa-zisa functie prescurtata. Aceasta functie simpla este scrisa pe un singur rand.

Sintaxa functiei lambda este urmatoarea:

nume_functie = lambda [parametrii] : instructiuni

Parametri vor fi mentionati imediat dupa cuvantul cheie lambda, despartiti de virgula in cazul in care sunt mai multi. Urmeaza caracterul "**:**" si instructiunile de executat.

Functia se apeleaza cu propriul nume si parametrii in paranteze, ca si in cazul functiilor obisnuite.

Functia lambda NU are return.

```
>>>aria_tri = lambda baza, inaltimea: baza * inaltimea * .5

>>>celsius = lambda fahr: (5.0 / 9) * (fahr - 32)

>>>aria_tri ( 10, 5 )
25
>>>celsius ( 100 )
37.77777777777778
```

5.4 Lucrul cu fisiere text

In Python este foarte usor, ca de altfel si in alte limbaje, sa fie accesate si prelucrate fisiere de tip text. Un fisier text nu trebuie neaparat sa aiba extensia .txt.

Fisierele de tip text au avantajul ca pot fi accesate independent de sistemul de operare.

Un fisiere text este compus din mai multe linii. Daca fisierul va contine date organizate tabelar fiecare linie va avea acelasi numar de campuri, delimitate de un caracter (cele mai uzuale caractere delimitatoare sunt virgula, punct si virgula, tab). Exista, de asemenea, un delimitator de linie (in Linux '\n', in Windows '\r\n'). Python recunoaste automat delimitatorul de linie, indiferent de sistemul de operare.

5.4.1 Deschiderea fisierelor

Pentru a deschide un fisier text vom crea un obiect (o variabila), cu ajutorul functiei **open().** Aceasta va primi doua argumente: calea catre fisier si modul in care il deschidem. Calea catre fisier poate fi absoluta sau relativa si trebuie sa contina inclusiv denumirea fisierului si extensia acestuia. Calea va fi scrisa sub forma de sir de caractere, cu delimitatoarele specifice. Modul in care il deschidem poate fi unul dintre urmatoarele:

- 'r' - exclusiv pentru citirea de fisiere. Fisierul trebuie sa existe
- 'w' - exclusiv pentru scrierea in fisiere. Daca fisierul nu exista va fi creat. Daca exista va fi rescris, deci veti pierde vechiul continut;
- 'a' - exclusiv pentru scrierea in fisiere. Daca fisierul nu exista va fi creat. Daca exista va fi completat la final, fiind pastrat vechiul continut;
- 'r+' - pentru citirea si scrierea de fisiere. Fisierul trebuie sa existe. Daca scriem vechiul continut va fi sters

Modul in care deschidem fisierul poate sa lipseasca. In acest caz va fi preluat automat modul 'r', exclusiv scriere.

```
>>> text = open ( "c:/wt/guta.txt", "r" )
```

Aceasta instructiune va crea variabila text, care va deschide fisierul guta.txt, aflat pe partitia C a HDD, in directorul WT. Fisierul este deschis exclusiv pentru scriere.

```
>>> text = open ( "c:/wt/guta.txt", "a+" )
```

Aceasta instructiune va crea variabila text, care va deschide fisierul guta.txt, aflat pe partitia C a HDD, in directorul WT. Fisierul este deschis pentru scriere si citire cu scriere la finalul fisierului existent. Daca nu exista fisierul va fi creat.

5.4.2 Citirea fisierelor

Citirea fisierelor se poate face in mai multe moduri astfel:

Citeste intregul continut al fisierului text, cu metoda ***read()***:

```
>>> text = open ( "c:/wt/guta.txt", "r" )

>>> x = text.read ( )
>>> print ( x )

>>> text.close ( )                    # fisierul trebuie inchis dupa utilizare
```

Citeste linie cu linie continutul fisierului text, cu metoda ***readline()***:

```
>>> text = open ( "c:/wt/guta.txt", "r" )

>>> print (text.readline ( ))          # Prima linie
>>> print (text.readline ( ))          # A doua linie
```

Citeste toate liniile fisierului text si adauga continutul fiecarei linii ca element intr-o lista, cu metoda ***readlines()***:

```
>>> text = open ( "c:/wt/guta.txt", "r" )

>>> linii = text.readlines ( )

>>> print (linii)
```

5.4.3 Scrierea fisierelor

Scrierea in fisiere se face cu ajutorul ***write()***, care va primi ca argument textul pe care dorim sa-l scriem:

```
>>> text = open ( "c:/wt/scrie.txt", "w" )
```

```
>>> text.write ("Astazi va fi soare!\n")    # \n semnifica inceierea liniei. Este obligatoriu,
                                            # altfel continua pe aceeasi linie
>>> text.close ( )
```

5.4.4 Utilizarea buclei for in fisiere

O alta metoda de citire a liniilor unui fisiere este printr-o ***bucla for***:

```
>>> text = open ( "c:/wt/guta.txt", "r" )
>>> for line in text:
        print (line)                    # printeaza tot continutul linie cu linie
```

In exemplul urmator cream un dictionar cu toate cuvintele din text si numarul aparitiilor.

```
>>> text = open ( "c:/wt/guta.txt", "r" )
>>> counts = dict ( )
>>> for line in text:
        words = line.split ( )                  # text linii split in cuvinte
        for word in words:
            if word not in counts:              # daca nu exista in dictionar il creaza
                counts[word] = 1
            else:
                counts[word] += 1               # daca exista in dictionar aduna 1
>>> print ( counts )
```

5.5 Recapitulare

5.5.1 Recapitulare