Laboratorul 2

Ce știm deja? În laboratorul anterior am stabilit ce presupune programarea funcțională și am învățat să definim funcții simple în Python.

Obiectiv: Săptămâna aceasta vom aprofunda tema funcțiilor și vom argumenta de ce limbajul Python suportă paradigma programării funcționale.

Python tratează funcțiile ca orice valoare (int, float, ...) pe care ați întâlnit-o până acum. Puteți de exemplu să asignați unei variabile o funcție:

```
def suma(a, b):
    return a + b

suma_numere = suma
print(suma_numere(2, 3))
```

Dacă vom vrea să vedem tipul variabilei definite mai sus putem scrie:

```
a = -3
print(type(a))
print(type(suma_numere))  #functia din exemplul anterior
```

Remarcați tipul pe care îl are variabila suma_numere.

Python și programarea funcțională

Pentru a facilita scrierea de programe conform paradigmei programării funcționale un limbaj de programare ar trebui să pună la dispoziție următoarele două mecanisme:

- să poată primi o funcție drept parametru
- să poată returna o funcție (către entitatea care a apelat-o)

Deoarece am văzut mai sus modul în care Python tratează funcțiile, cele două cerințe de mai sus sunt asigurate cu succes de către Python.

Funcții ca parametru

O funcție poate la randul său să fie trimisă ca parametru altei funcții. De exemplu, ne putem defini funcția multiply_5, care ia ca parametru o funcție și un număr. Ne definim și o funcție simplă numită increment_1, care ia un parametru și îl returnează incrementat cu 1. Acum putem apela funcția multiply_5 cu parametrii incrementare și un număr, de exemplu 5.

```
def increment_1(x):
    return x + 1
print(increment_1(5))

def multiply_5(f, x):
    return f(x) * 5

print(multiply_5(increment_1, 5))
```

Observatie: funcția multiply_5 poate fi apelată cu orice parametru pentru f, și x. Totuși în cazul în care am apela-o cu două valori întregi, am avea o eroare la rulare. De aceea e important să știm ce parametrii așteaptă funcția pe care o apelăm.

Funcții anonime

Notația `lambda` *argument* `:` *expresie* definește în Python o funcție anonimă (funcție lambda). Aceasta este o *expresie* de tip funcție și poate fi deci folosită în alte expresii. Putem evalua direct fără a fi nevoie să dăm întâi un nume funcției. Acest exemplu simplu ilustrează că în limbajul Python, o funcție (aici `lambda x : x + 3`) poate fi folosită la fel de simplu ca și orice altă valoare.

```
>>> (lambda x : x + 3)(2)
    5
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
```

Revenind la notația `def`, este echivalent să definim:

```
>>> def f(x):
...     return x + 3
```

sau

```
>>> def f(x):
...     return (lambda x : x + 3)(x)
```

Acest mod de a defini o funcție nu prea area sens, deoarece în cadrul funcției f ne definim o funcție anonimă pe care o apelăm imediat ce o definim. În schimb, funcțiile anonime pot fi

folositoare dacă dorim să returnăm o funcție. De exemplu, putem defini o funcție care ne returnează o funcție de incrementare cu 1 dacă parametrul primit este True sau o funcție de decrementare cu 1 dacă parametrul primit este False. În acest caz funcțiile anonime sunt folositoare.

```
def return_functie_incrementare(x):
    if x:
        return lambda x: x + 1
    else:
        return lambda x: x - 1


inc = return_functie_incrementare(True)
dec = return_functie_incrementare(False)

print(inc(5))
print(dec(5))
```

Se poate observa că rezultatul apelării funcției return_functie_incrementare este salvat în variabilele `inc`, respectiv `dec`. Aceste două variabile sunt la rândul lor funcții, deci pot fi apelate.

După cum am văzut în subcapitolul anterior, funcțiile pot fi și ele transmise ca parametru. În loc de o funcție anterior definită, putem trimite o funcție anonimă ca parametru. Astfel, exemplul cu multiply_5 devine:

```
def multiply_5(f, x):
    return f(x) * 5
print(multiply_5((lambda x: x + 1), 5))
```

Compunerea funcțiilor

Una din cele mai simple și larg folosite operații cu funcții este *compunerea* lor, în acest fel putem obține ușor funcții (prelucrări) complexe pornind de la funcții simple. În matematică, dacă $f : A \to B$ și $g : C \to A$, compunerea $f \circ g : C \to B$ e definită prin relația $(f \circ g)(x) = f(g(x))$.
Deci, pornind de la o valoare $x \in C$ se obține o valoare $g(x) \in A$, și apoi prin aplicarea lui f valoarea $f(g(x)) \in B$.
în Python putem defini o funcție de compoziție cu doi parametrii, care returnează o altă funcție, reprezentând funcția f compusă cu funcția g.

```
def comp(f,g):
    return lambda x : f(g(x))

def f1(x):
    return x * 2
```

```
def f2(x):
    return x + 1

f = comp(f1, f2)
print(f(6))
f = comp(f2, f1)
print(f(6))
```

Apelarea funcției comp cu parametrii f1 și f2, care sunt la rândul lor tot funcții, ne dă o altă funcție. Bineînțeles, ordinea în care dăm parametrii contează, astfel că rezultatul primei compuneri este o funcție diferită de rezultatul celeilalte compuneri. Acest lucru se poate vedea și prin faptul că rezultatele apelării loc cu același parametru.

Aplicarea repetată (compunerea unei funcții cu ea însăși)

În particular, dacă o funcție are același domeniu de definiție și de valori, poate fi compusă cu ea însăși: $f \circ f$, unde $f : A \to A$. Pornind de la funcția de compunere `comp` putem defini o funcție de ordin superior care compune o funcție (dată ca parametru) cu ea însăși.

```
def appl2(f):
    return comp(f, f)

f = appl2(f1)
print(f(6))
```

În același fel, putem compune funcția `appl2` cu ea însăși, obținând o funcție care aplică de 4 ori funcția primită ca parametru:

```
appl4 = comp(appl2, appl2)

f = appl4(f1)
print(f(6))
```

Putem defini însă și altă funcție:

```
def appl4(f):
    return appl2(appl2(f))

f = appl4(f1)
print(f(6))
```

În acest caz, primul `appl2` produce aplicarea de două ori a funcției dată ca parametru, care e tot `appl2` . Încercând cu mai multe funcții și argumente întregi, obținem valori egale, sugerând că în cele două variante am definit de fapt aceeași funcție.

Operatorii sunt funcții

Pentru a avea acces la operatori trebuie sa importam modulul *operator. Apoi putem folosi un operator in felul urmator:*

```
import operator
>>> operator.add(2,3)
    5
>>> 2 + 3
    5
```

Pentru mai multe detalii consultați link-ul.

Ne putem defini o funcție generică de operații, care ia ca parametrii o funcție care reprezintă operația pe care vrem să o efectueze și două valori numerice. Funcția se definește și se apelează astfel:

```
def operatie_generica(op, x, y):
    return op(x, y)

print(operatie_generica(operator.add, 3, 4))
```

Funcții cu parametrii impliciți (default)

Python ne premite să declarăm funcții cu parametrii care au o valoare implicită. Astfel de funcții pot fi apelate și normal, dar și fără a da o valoare parametrului cu valoare implicită. În acest caz, parametrul va fi egal cu valoarea implicită. Mai jos se poate vedea un exemplu de funcție care are un parametru implicit.

```
def increment(x = 0):
    return x + 1

print(increment(4))
print(increment())
```

În cazul în care avem o funcție cu mai mulți parametrii, dintre care unii impliciți, este important ca parametrii impliciți să se afle la finalul listei de parametrii. De exemplu, nu putem avea o funcție cu primul parametru implicit și al doilea explicit. Mai jos se poate vedea un exemplu cu mai mulți parametrii. Este indicat să îl rulați pentru a înțelege cum funcționează parametrii expliciți în funcții cu mai mulți parametrii.

```
def print_params(a, x = "a", y = "b", z = "c"):
    print(a, x, y, z)
```

```
print_params("aaa")
print_params("aaa", "x")
print_params("aaa", "y")
print_params("aaa", y = "x")
print_params("1", "2", "3", "4")
```

Acest cod printează următoarele linii:

```
aaa a b c
aaa x b c
aaa y b c
aaa a x c
1 2 3 4
```