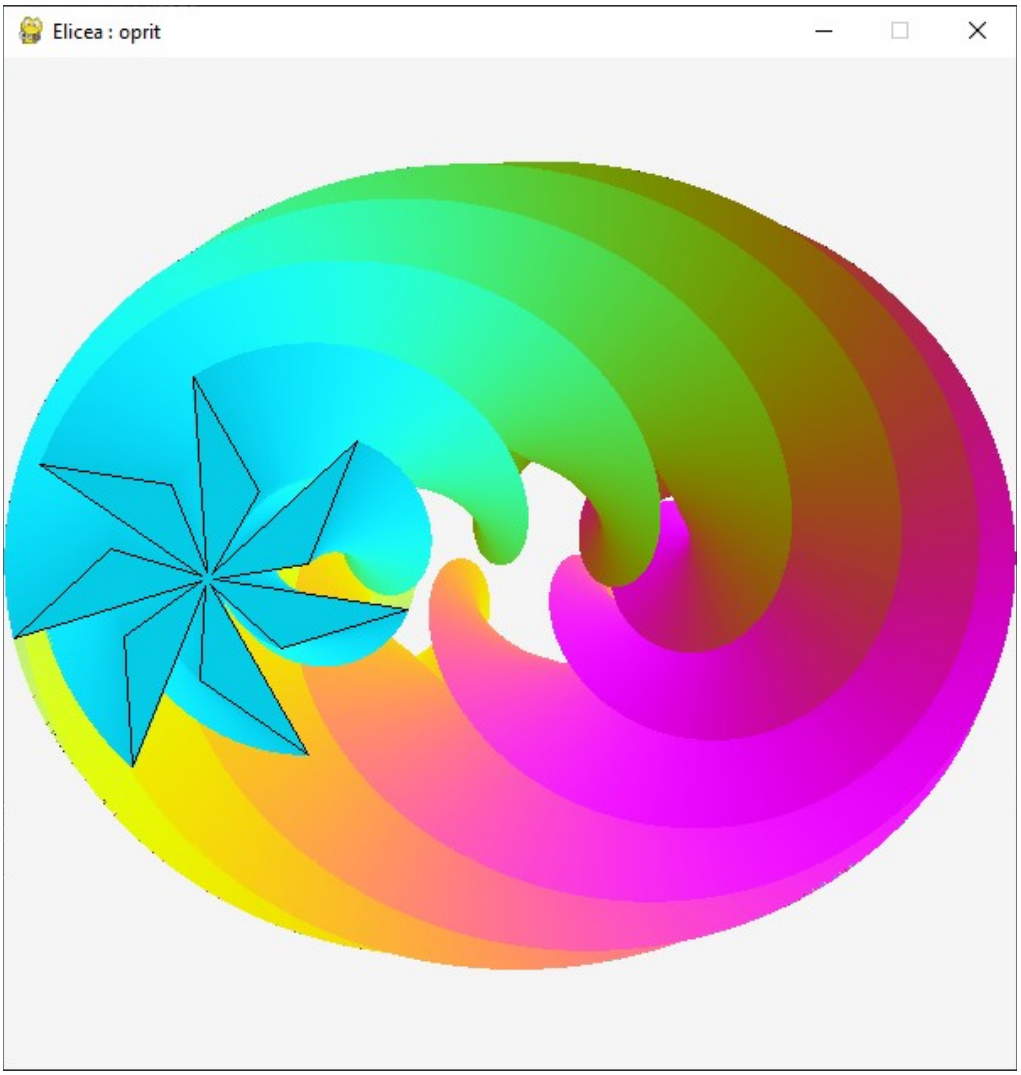
Liste in Python

Vom prezenta aici utilizarea listelor în Python prin rezolvarea unor exerciții.

Exemplul 1. *Realizați o animație care redă mișcarea unei elice care se rotește în timp ce centrul său se mișcă pe o elipsă fixă.*



Rezolvare.

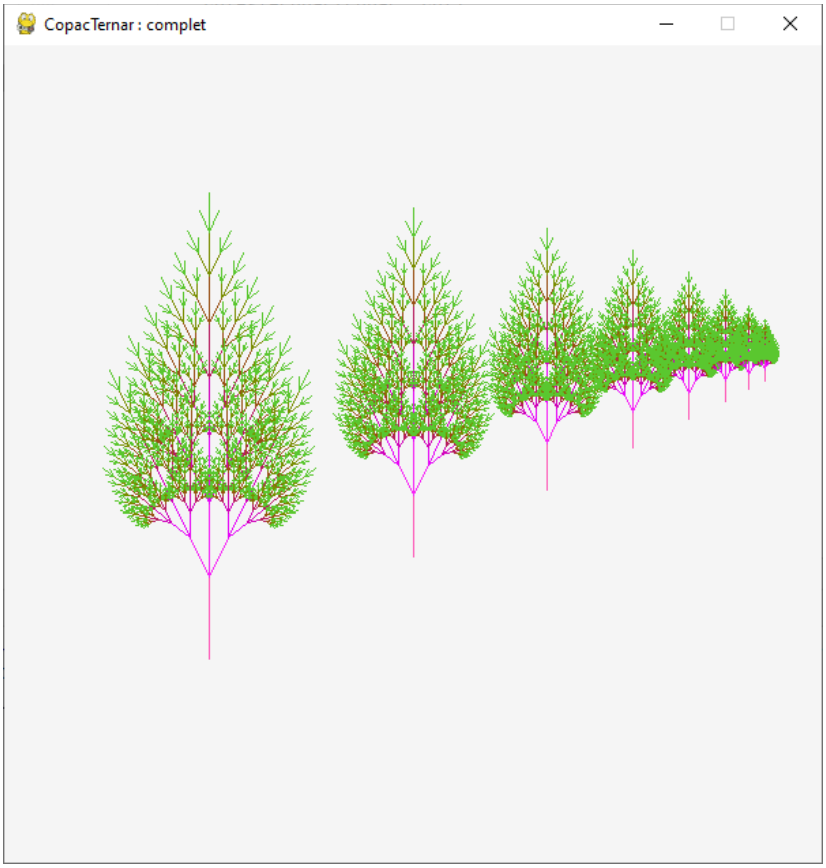
```
import ComplexPygame as C
import Color
import math
def Elicea():
    def miscaElicea(t, li):
        """returnam imaginea elicei"""
        q = complex(3 * math.cos(t), 2 * math.sin(t))
        omega = C.fromRhoTheta(1, 2 * t)
        return [q + omega * z for z in li]

    def formeazaElicea(r, N):
        rez = []
        alfa = 2 * math.pi / N
        for k in range(N):
            z1 = C.fromRhoTheta(r, k * alfa)
            z2 = C.fromRhoTheta(2 * r, (k + 2 / 3) * alfa)
            z3 = z2 / 30
            rez.append(z1)
            rez.append(z2)
            rez.append(z3)
        return rez

    C.setXminXmaxYminYmax(-5, 5, -5, 5)
    elice = formeazaElicea(1, 7)
    for k in range(100000):
        t = k / 1000
        #C.fillScreen()
        img = miscaElicea(t, elice)
        C.drawNgon(img, Color.Black)
        if C.mustClose():
            break
        C.fillNgon(img, Color.Index(k // 10))
        C.drawNgon(img, Color.Index(k // 10))

if __name__ == '__main__':
    C.initPygame()
    C.run(Elicea)
```

Studiem codul. Observăm că elicea din imagine este trasată ca o linie poligonală cu vârfurile stocate într-o listă de numere complexe, cea returnată de funcția `formeazaElicea()`. Deoarece linia poligonală este trasată cu metoda `C.drawNgon()`, nu este necesar ca lista vârfurilor să fie circulară, mai precis nu este necesar ca primul element să coincidă cu ultimul.

Elicea memorată în lista `elice` este fixă și centrată în origine, numai imaginea ei se mișcă, imagine obținută în funcția `miscaElicea()` printr-o după o rotație de rotor `omega` și o translație de vector `q`.

Contiunuăm cu utilizarea listelor, exersând acum pe următorul exemplu:

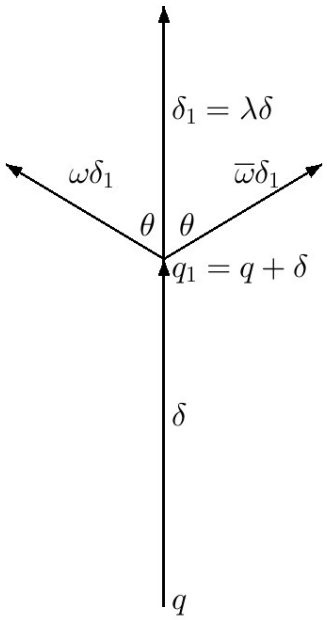
Exemplul 2. *Presupunem că un copac creşte după următoarea schemă simplificată: mugurii de creştere se află numai în vârfurile rămurelelor de un an, câte trei în fiecare vârf, şi din fiecare mugur creşte în fiecare an câte o nouă rămurică. Să se deseneze un astfel de copac colorând diferit fiecare generaţie de ramuri şi apoi imaginea obţinută să fie mutată şi micşorată de mai multe ori, ca în figura următoare*:



Rezolvare. Simplificăm şi mai mult modelul: toate creşterile sunt în linie dreaptă, mugurul din mijloc este în prelungirea ramurii iar ceilalţi doi fac acelaşi unghi θ cu acesta. Un copac va fi, prin urmare, o mulţime de segmente colorate, pe care le vom memora în liste. Un astfel de segment îl vom numi *ram*, indiferent dacă este un segment internoduri sau o rămurică terminală, şi îl vom implementa ca un tuplu `(q, delta)`, în unde `q` este afixul punctului de plecare al ramului, iar `delta` este vectorul care dă direcţia şi lungimea ramului, astfel că vârful ramului are afixul `q+delta`.

Vom memora fiecare etaj anual într-o listă de tupluri `(q, delta)`, iar întregul copac va fi o listă de etaje.

Să prezentăm modelul matematic necesar metodei de generare a creşterilor anuale. Fie q şi $q1 = q + \delta$ baza, respectiv vârful unui ram din ultima generaţie, din care vor creşte trei segmente noi.



Cel din mijloc va fi coliniar cu $q\,q_1$, prin urmare va avea vectorul director de forma $\delta_1 = \lambda\delta$, cu scalar subunitar. Condiţia $\lambda \in (0,1)$ asigură mărginirea copacului (de ce?), şi îl vom fixa la valoarea `Lambda = 0.9,` aceeaşi pentru toţi mugurii centrali.

Fie δ_2 vectorul director al ramului nou din stânga. Definim $\omega = \frac{\delta_2}{\delta_1}$ şi avem

$$\omega = \rho(\cos\theta + i\ \sin\theta)$$

unde $\theta \in \left(0, \frac{\pi}{2}\right)$ este unghiul dintre un ramul lateral şi cel din mijloc iar ρ este raportul lungimilor celor două ramuri, raport care a fost fixat subunitar:

```
rho = 0.6
theta = math.pi / 7
omega = C.fromRhoTheta(rho, theta)
```

În programul următor formarea noii generaţii este dată de funcţia `genereaza()`, folosind precizările de mai sus. Copacul este generat etaj cu etaj în funcţia `formeazaCopac()` şi este apoi este transformat în mod repetat de funcţia `mutaCopac()` printr-o translaţie un vector u şi o scalare cu un factor subunitar σ.

```
import ComplexPygame as C
import Color
import math

def CopacTernar():
    Lambda = 0.9
    rho = 0.6
```

```
theta = math.pi / 7
omega = C.fromRhoTheta(rho, theta)

def genereaza(li):
    """li este o lista de tupluri (q,delta)"""
    rez = []
    for (q, delta) in li:
        # (q, delta) este ramura veche, formam 3 ramuri noi
        q1 = q + delta
        delta1 = Lambda * delta
        rez.append((q1, omega * delta1))
        rez.append((q1, delta1))
        rez.append((q1, omega.conjugate() * delta1))
    return rez

def formeazaCopac(tr0, nivMax=7):
    """tr0 este tuplul (q0,delta0) initial"""
    etaj = [tr0]
    copac = [etaj]  # lista de liste
    for k in range(nivMax):
        etaj = genereaza(etaj)
        copac.append(etaj)
        C.wait(10)
        if C.mustClose():
            break
    return copac

def mutaCopac(copac, u, sigma):
    """copac este o lista de liste de tupluri (q,delta)"""
    (q0, _) = copac[0][0]
    rez = []
    for etaj in copac:
        etajrez = []
        for (q, delta) in etaj:
            etajrez.append((u + q0 + sigma * (q - q0), delta * sigma))
        rez.append(etajrez)
    return rez

def arataCopac(copac):
    """copac este o lista de liste de tupluri (q,delta)"""
    for k in range(len(copac)):
        for (q, delta) in copac[k]:
            C.drawLine(q, q + delta, Color.Index(350 + 40 * k))

C.setXminXmaxYminYmax(-1, 3, -1, 3)
C.fillScreen()
trunchi = (0, 0.4j)
copac = formeazaCopac(trunchi)
arataCopac(copac)
```

```
    u = 1 + 0.5j
    sigma = 0.75
    for k in range(7):
        copac = mutaCopac(copac, u, sigma)
        arataCopac(copac)
        C.refreshScreen()
        C.wait(10)
        u *= 0.65

if __name__ == '__main__':
    C.initPygame()
    C.run(CopacTernar)
```

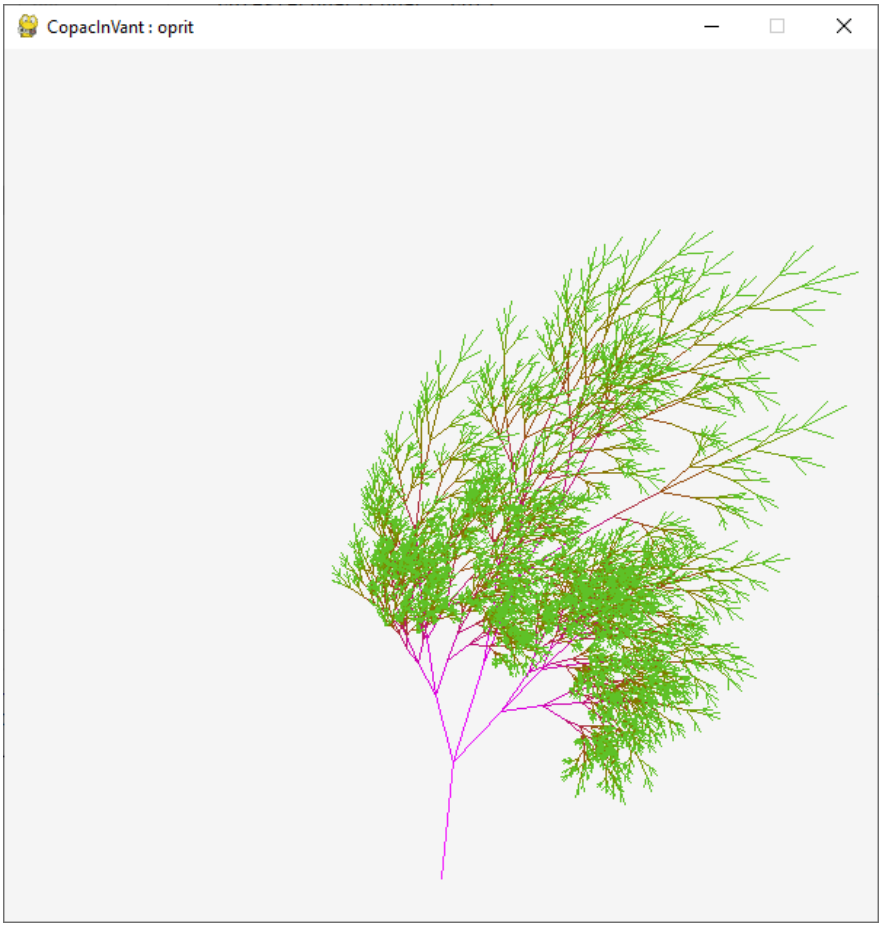
Copacii desenaţi aici au un aspect supărător de regulat. Pentru a obţine un aspect cât mai natural, trebuie să introducem un anumit grad de dezordine în regulile de generare, de exemplu: unghiurile dintre ramurile noi să varieze aleator în jurul valorii θ, ramul nou central să fie şi el înclinat aleator faţa de cel vechi, mugurii laterali să germineze doar cu o anumită probabilitate. In acest scop se poate importa modulul `random` și apoi se poate folosi metoda

```
def zar():
    return random.uniform(-0.1, 0.1)
```

care returnează un număr aleator, uniform distribuit, în intervalul $[-0.1, +0.1]$. Desenul următor a fost obţinut mutând numai punctul iniţial al copacului şi generând la fiecare mutare un copac aleator cu trunchiul iniţial micşorat corespunzător. Încercaţi şi voi.

Acum, după ce am reuşit să plantăm copaci într-o livadă, vrem să-i facem să se legene în vânt.

Exemplul 3. *Să se realizeze o animaţie care să simuleze mişcarea oscilatorie a unui arbore sub acţiunea vântului.*



Rezolvare. Dacă rotim înainte şi înapoi tot copacul în jurul punctului de bază q_0 obţinem o mişcare rigidă, nefirească. Pentru a simula unduirile ramurilor unui copac trebuie să rotim câte puţin, înainte şi înapoi, fiecare ram în jurul propriului punct de bază. Rotirea rămurelelor terminale nu ridică probleme, dar rotirea unui segment internoduri desprinde vârful său de baza celor trei ramuri crescute din el, şi astfel distrugem copacul. Suntem nevoiţi ca pentru fiecare cadru al animaţiei să generăm în memorie din nou întregul copac cu ramurile rotite puţin faţă de propria lor bază şi apoi să-l afişăm.

În cazul unui copac artificial dat de clasa `CopacTernar` modificările sunt puţine şi uşor de implementat, prin urmare sunt lăsate în seama cititorului, noi ne propunem să mişcăm un copac cu aspect natural, unul cu ramurile dispuse aleator. Dificultatea suplimentară este următoarea: ca să reuşim să generăm din nou acelaşi copac dar cu ramurile rotite puţin, va trebui să memorăm cumva înclinaţiile ω alese în mod aleator la prima generare a copacului, şi apoi să le folosim, modificate corespunzător.

Pentru a păstra aceste informaţii folosim obiecte din clasa

```
class Ram:
    def __init__(self, q, delta, muguri):
        self.q = q
        self.delta = delta
        self.muguri = muguri

    def show(self, color):
        C.drawLine(self.q, self.q + self.delta, color)
```

în care `muguri` este o listă de unul, două sau trei numere complexe ω corespunzătoare celor trei ramuri noi (atenţie: în programul `CopacTernar` s-au folosit numai două, `omega` pentru ramul stâng şi `omega.conjugate()` pentru ramul drept, aici primul `omega` din lista `muguri` va determina ramul nou central, care nu mai este coliniar cu ramul vechi) .

Ramurile copacului inițial vor avea mugurii generați aleator de funcția

```
def rndRam(q, delta):  # returneaza un  ram cu muguri aleatori
    def zar():
        return random.uniform(-0.125, 0.125)

    muguri = [C.fromRhoTheta(lambda0 + zar(), zar())]  # omega
    if zar() < 0.08:
        muguri.append(C.fromRhoTheta(rho0 + zar(), theta0 + zar()))  # omegaStg
    if zar() < 0.08:
        muguri.append(C.fromRhoTheta(rho0 + zar(), -theta0 + zar()))  # omegaDrp
    return Ram(q, delta, muguri)
```

care va fi utilizată la generarea, etaj cu etaj, a copacului inițial:

```
def genereazaEtaj(etaj):
    rez = []
    for ram in etaj:
        # ram  este pe etajul vechi, formam 1,2 sau 3 ramuri noi
        q1 = ram.q + ram.delta
        for omg in ram.muguri:
            dt1 = omg * ram.delta
            rez.append(rndRam(q1, dt1))
    return rez
```

Iată întregul text sursă al programului:

```
import ComplexPygame as C
import Color
import math, random
def CopacInVant():
    class Ram:
        def __init__(self, q, delta, muguri):
            self.q = q
            self.delta = delta
            self.muguri = muguri

        def show(self, color):
            C.drawLine(self.q, self.q + self.delta, color)
```

```
lambda0 = 0.9
rho0 = 0.6
theta0 = math.pi / 7

def rndRam(q, delta):  # returneaza un  ram cu muguri aleatori
    def zar():
        return random.uniform(-0.125, 0.125)

    muguri = [C.fromRhoTheta(lambda0 + zar(), zar())]  # omega
    if zar() < 0.08:
        muguri.append(C.fromRhoTheta(rho0 + zar(), theta0 + zar()))  # omegaStg
    if zar() < 0.08:
        muguri.append(C.fromRhoTheta(rho0 + zar(), -theta0 + zar()))  # omegaDrp
    return Ram(q, delta, muguri)

def formeazaCopacEtajat(q, delta, nivMax=9):
    def genereazaEtaj(etaj):
        rez = []
        for ram in etaj:
            # ram  este pe etajul vechi, formam 1,2 sau 3 ramuri noi
            q1 = ram.q + ram.delta
            for omg in ram.muguri:
                dt1 = omg * ram.delta
                rez.append(rndRam(q1, dt1))
        return rez

    omg = C.fromRhoTheta(lambda0, 0)
    omgstg = C.fromRhoTheta(rho0, theta0)
    omgdrp = C.fromRhoTheta(rho0, -theta0)
    etaj = [Ram(q, delta, [omgstg, omg, omgdrp])]
    copacEtajat = [etaj]
    for _ in range(nivMax):
        etaj = genereazaEtaj(etaj)
        copacEtajat.append(etaj)
    return copacEtajat

def rotesteCopac(copacEtajat, rotor):
    # modificam copacul rotind ramurile
    # etaj cu etaj
    etaj = copacEtajat[0]
    etaj[0].delta = rotor * delta0
    for etajNou in copacEtajat[1:]:
        k = 0
        for ram in etaj:
            q1 = ram.q + ram.delta
            for omg in ram.muguri:
                dt1 = rotor * omg * ram.delta
                if ram.muguri.index(omg) > 0:
                    dt1 *= rotor
                etajNou[k].q = q1
                etajNou[k].delta = dt1
                k += 1
        etaj = etajNou

def traseazaCopac(copac):
    C.fillScreen()
```

```
        for k in range(len(copac)):
            for ram in copac[k]:
                ram.show(Color.Index(400 + 25 * k))
        C.refreshScreen()

    C.setXminXmaxYminYmax(-3, 3, 0, 6)
    q0 = 0.3j
    delta0 = 0.8j
    copac = formeazaCopacEtajat(q0, delta0)
    traseazaCopac(copac)
    C.wait(1000)
    t = 0
    while t < 10000:
        rot = C.fromRhoTheta(1, math.sin(t) / 10)
        rotesteCopac(copac, rot)
        traseazaCopac(copac)
        if C.mustClose(): return
        C.wait(10)
        t += 0.05

if __name__ == '__main__':
    C.initPygame()
    C.run(CopacInVant)
```