**MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**RAPORT**

Lucrare de laborator nr. 2

la cursul ***„Medii Integrate de Proiectare Interactiva”***

**Varianta 13**

**Tema:** *OOP în Python. Clase. Module în Python. Module numerice. Prelucrarea și vizualizarea datelor.*

**A efectuat :**  **St. gr. CR-221FR Serba Cristina**

**A verificat:**  **Victoria Alexei**

**Chișinău 2024**

# Crearea unei simple clase

## A. Alegeți varianta clasei din tabelul nr.1



Când este implementată, clasa trebuie să conțină:

* metode speciale:
* ***\_\_init\_\_(self, ...)*** - inițializare cu parametrii necesari;
* ***\_\_str\_\_(self)*** - reprezentarea obiectului într-o formă prietenoasă omului;
* metode speciale pentru posibilitatea de adunare, diferență și altele operațiuni pe care clasa trebuie să le suporte;
* metode de clasă:
  + ***from\_string(cls, str\_value)*** - creează un obiect pe baza siruri de caractere ***str\_value*** ;
* câmpuri, metode, proprietăți:
  + câmpuri obligatorii pentru clasa selectată;
  + metoda ***save(self, filename)*** - salvează obiectul în fișier ***filename***;
  + metoda ***load(self, filename)*** - încarcă un obiect de fișier in ***filename***;
  + alte metode (cel puțin 3) și proprietăți identificate în etapa de studio clasă.

Implementați clasa și testați-i toate capacitățile.

## B. Creați o clasă container

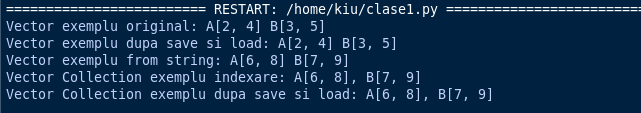
Care va conţine un set de obiecte din sarcina anterioară. De exemplu, clasa ***VectorCollection*** va conține obiecte clasa ***Vector***.

Pentru o clasă de container, furnizați:

* metode speciale:
  + ***\_\_init\_\_(self, ...)*** - inițializare cu necesar parametrii;
  + ***\_\_str\_\_(self)*** - reprezentarea obiectului într-o manieră prietenoasă cu oamenii formă;
  + ***\_\_getitem\_\_()*** - indexare și tăiere pentru clasa container.
* câmpuri, metode, proprietăți:
  + câmpul ***\_date*** - conține un set de date;
  + metoda ***add(self, value)*** - adaugă un element de valoare containerului;
  + metoda ***remove(self, index)*** - elimină un element din container prin ***index***;
  + metoda ***save(self, filename)*** - salvează obiectul în fișier ***filename***;
  + metoda ***load(self, filename)*** - încarcă un obiect din fișier ***filename*** .

|  |
| --- |
| class Vector():  def \_\_init\_\_(self, Ax, Bx, Ay, By):  self.\_Ax = Ax  self.\_Bx = Bx  self.\_Ay = Ay  self.\_By = By    def from\_string(cls, str\_value: str):  return cls.\_\_init\_\_()    def \_\_str\_\_(self):  return f"A[{self.Ax}, {self.Ay}] B[{self.Bx}, {self.By}]"    def save(self, filename):  with open(filename, "w") as file:  file.write(str(self.Ax))  file.write(str(self.Bx))  file.write(str(self.Ay))  file.write(str(self.By))    def load(self, filename):  with open(filename, "r") as file:  args\_list = file.read()  Ax = int(args\_list[0])  Bx = int(args\_list[1])  Ay = int(args\_list[2])  By = int(args\_list[3])    @classmethod  def from\_string(cls, str\_value):  args\_list = str\_value.split(' ')  Ax = int(args\_list[0])  Bx = int(args\_list[1])  Ay = int(args\_list[2])  By = int(args\_list[3])  return cls(Ax, Bx, Ay, By)    @property  def Ax(self):  return self.\_Ax;    @Ax.setter  def Ax(self, value):  self.\_Ax = value    @property  def Bx(self):  return self.\_Bx;    @Bx.setter  def Bx(self, value):  self.\_Bx = value    @property  def Ay(self):  return self.\_Ay;    @Ay.setter  def Ay(self, value):  self.\_Ay = value    @property  def By(self):  return self.\_By;    @By.setter  def By(self, value):  self.\_By = value    class VectorCollection():  def \_\_init\_\_(self, data):  self.\_date = [data]    def \_\_str\_\_(self):  return "Vector Collection class"    @property  def date(self):  return self.\_date;    @date.setter  def date(self, value):  self.\_date = value    def add(self, value):  self.\_date.append(value)    def remove(self, index):  del self.\_date[index]    def save(self, filename):  with open(filename, "w") as file:  for index, vector in enumerate(self.\_date):  file.write(str(index))  file.write(str(vector.Ax))  file.write(str(vector.Bx))  file.write(str(vector.Ay))  file.write(str(vector.By))    def load(self, filename):  with open(filename, "r") as file:  args\_list = [int(i) for i in file.read()]  i = 0  while i < len(args\_list):  index = int(args\_list[i])  self.\_date[index].Ax = int(args\_list[i+1])  self.\_date[index].Bx = int(args\_list[i+2])  self.\_date[index].Ay = int(args\_list[i+3])  self.\_date[index].By = int(args\_list[i+4])  i = i+5    def \_\_getitem\_\_(self, index):  return f"A[{self.date[index].Ax}, {self.date[index].Ay}], B[{self.date[index].Bx}, {self.date[index].By}]"        exempluVector1 = Vector(2, 3, 4, 5)  print("Vector exemplu original: " + exempluVector1.\_\_str\_\_())  exempluVector1.save("file.txt")  exempluVector1.load("file.txt")  print("Vector exemplu dupa save si load: " + exempluVector1.\_\_str\_\_())  exempluVector2 = Vector.from\_string("6 7 8 9")  print("Vector exemplu from string: " + exempluVector2.\_\_str\_\_())    exempluColectie = VectorCollection(exempluVector1)  exempluColectie.add(exempluVector2)  print("Vector Collection exemplu indexare: " + exempluColectie[1])  exempluColectie.save("fisier.txt")  exempluColectie.load("fisier.txt")  print("Vector Collection exemplu dupa save si load: " + exempluColectie[1]) |

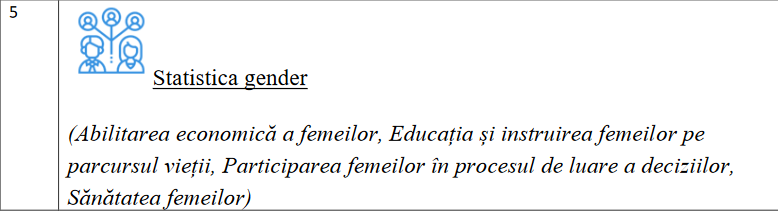
Rezultatul executiei:



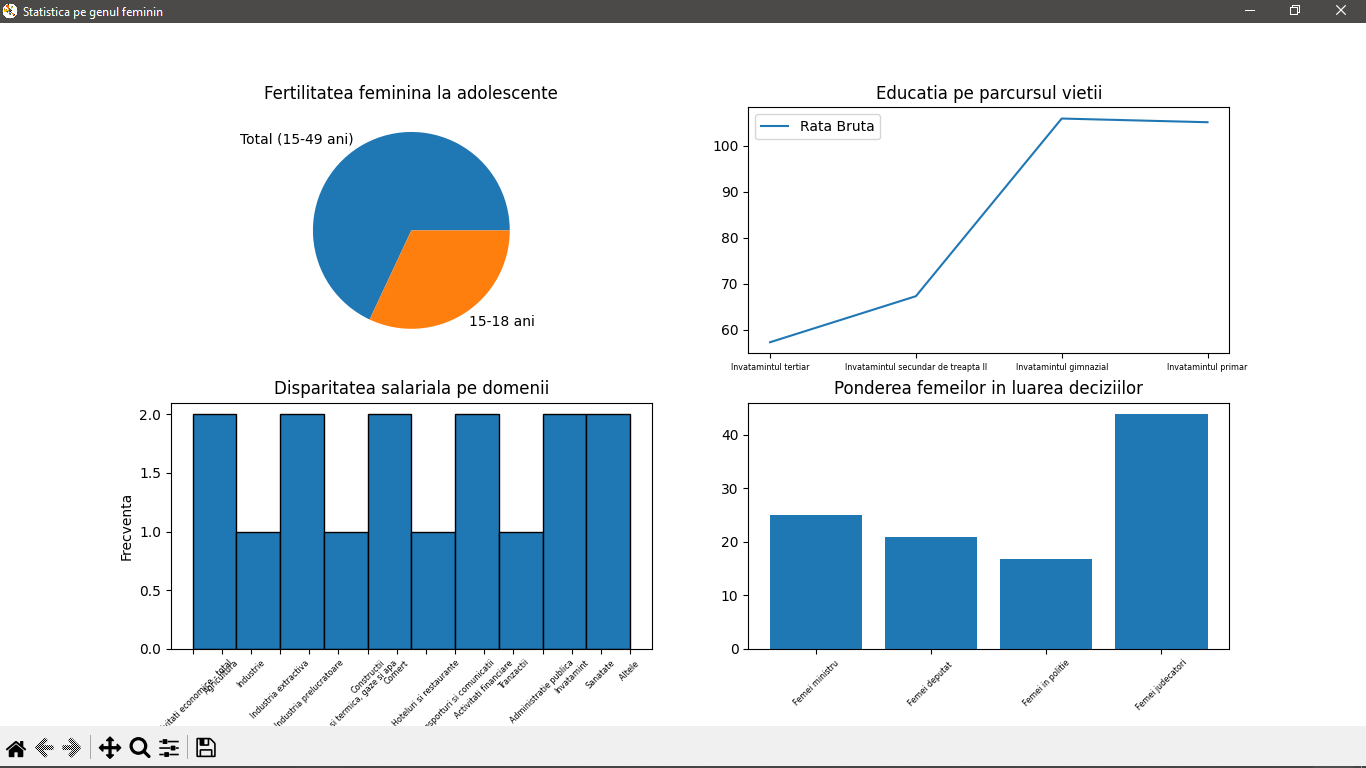
# Prelucrarea datelor și reprezentarea grafică

**Faceți cunoștință cu site-ul statistica.md**

* Având datele de pe site-ul respectiv:
* Alegeți un domeniu de interes din tabelul nr. 1
* Să se execute pe datele selectate calcule statistice la alegere nu mai puțin de 4 (din temele Probabilităţi şi statistică disciplinelor trecute).
* Rezultatele să fie reflectate în diferite tipuri de diagrame (4 în 1, de ex. de tip histogramă, linie(punctată), bare și plăcintă(pie)) cu explicații de rigoare (ca în figura nr 1 – se anexează).



|  |
| --- |
| import matplotlib.pyplot  import matplotlib.gridspec  import pandas  # Primul tabel  dataframe1=pandas.read\_excel("C:/Users/crist/Downloads/uni/ANUL2/sem2/mipi/abilitate\_economica.xlsx")  x\_axis\_dataframe1 = dataframe1["Activitati"]  y\_axis\_dataframe1 = dataframe1["total"]  # Al doilea tabel  dataframe2=pandas.read\_excel("C:/Users/crist/Downloads/uni/ANUL2/sem2/mipi/educatie.xlsx")  x\_axis\_dataframe2 = dataframe2["Domeniul"]  y\_axis\_dataframe2 = dataframe2["Rata bruta"]  # Al treilea tabel  dataframe3=pandas.read\_excel("C:/Users/crist/Downloads/uni/ANUL2/sem2/mipi/fertilitate.xlsx")  x\_axis\_dataframe3 = dataframe3["Grupa de varsta"]  y\_axis\_dataframe3 = dataframe3["fertilitatea"]  # Al patrulea tabel  dataframe4=pandas.read\_excel("C:/Users/crist/Downloads/uni/ANUL2/sem2/mipi/decizii.xlsx")  x\_axis\_dataframe4 = dataframe4["Domenii"]  y\_axis\_dataframe4 = dataframe4["Ponderea"]  # Crearea sub graficelor  gridspec = matplotlib.gridspec.GridSpec(2, 2)  fig = matplotlib.pyplot.figure(num="Statistica pe genul feminin")  # Primul grafic - histograma  graph1 = fig.add\_subplot(gridspec[1, 0])  graph1.hist(x\_axis\_dataframe1, edgecolor="black")  graph1.set\_xticklabels(list(x\_axis\_dataframe1))  graph1.set\_ylabel("Frecventa")  graph1.set\_title("Disparitatea salariala pe domenii")  graph1.tick\_params(axis="x", rotation=45, labelsize="xx-small")  # Al doilea subgrafic - grafic  graph2 = fig.add\_subplot(gridspec[0, 1])  graph2.plot(x\_axis\_dataframe2.values, y\_axis\_dataframe2.values)  graph2.legend(["Rata Bruta"])  graph2.set\_title("Educatia pe parcursul vietii")  graph2.tick\_params(axis="x", labelsize="xx-small")  # Al treilea subgrafic - circular  graph3 = fig.add\_subplot(gridspec[0, 0])  graph3.pie(y\_axis\_dataframe3, labels=x\_axis\_dataframe3)  graph3.set\_title("Fertilitatea feminina la adolescente")  graph3.tick\_params(labelsize="xx-small")  # Al patrulea subgrafic - bare  graph4 = fig.add\_subplot(gridspec[1, 1])  graph4.bar(x\_axis\_dataframe4, y\_axis\_dataframe4)  graph4.set\_title("Ponderea femeilor in luarea deciziilor")  graph4.tick\_params(axis="x", rotation=45, labelsize="xx-small")  # Crearea graficului  fig.show() |



# Concluzii

În concluzie, OOP în Python oferă o modalitate eficientă de organizare și structurare a codului. Clasele permit definirea și gestionarea obiectelor personalizate, în timp ce modulele facilitează reutilizarea și împărțirea codului în unități funcționale. Modulele numerice, cum ar fi NumPy și Pandas, sunt esențiale pentru manipularea datelor numerice și tabulare într-un mod eficient.

Prelucrarea și vizualizarea datelor devin accesibile și puternice prin intermediul modulelor specializate, oferind funcționalități pentru analiza, filtrarea și reprezentarea grafică a informațiilor. Aceste abordări oferă dezvoltatorilor un set de instrumente robust pentru a crea aplicații eficiente și ușor de întreținut în Python. Programarea în Python, cu accent pe OOP și gestionarea modulară a codului, facilitează dezvoltarea de aplicații scalabile și orientate pe performanță.