Klaseak

Klaseek **objektuei zuzendutako programazioa** izeneko teknika aplikatzeko aukera ematen digute. Arazo konplexuak konpontzeko beste estrategia bat da. Funtzioekin, arazo bat arazo txikietan banatzen dugu. Aldiz, objektuetara zuzendutako programazioarekin, arazoa klasetan banatzen saiatzen gara. Baina nola? Arazoaren zati den guztia irudikatuz, klaseak erabiliz. Hots, arazoan existitzen diren gauzak objektuak bilakatuz!

Demagun Mario Kart bezalako lasterketa joko baten programa egin behar dugula. Objektuetara zuzendutako programazioa erabiliz, jokoaren elementuak honako klase hauekin irudika ditzakegu:

- Pertsonaia, bere izenarekin eta beste ezaugarri batzuekin.
- Autoa, bere ezaugarri hauekin: abiadura, erresistentzia, azelerazio funtzioak, etab.
- Zirkuitua, luzerarekin, tunelekin, sariekin eta abarrekin.

Nola sortu klaseak

Klase bat programazio egitura bat da, **gauza bat** bere propietate eta metodoekin errepresentatzeko aukera ematen diguna. Hau da, klase batek:

- Atributu edo propietateak ditu: klasearen aldagaiak.
- Gauzak egiten ditu: funtzioak.

Adibidez, hurrengo klaseak katu oso sinple bat errepresentatzen du, miauka egiteko funtzio batekin:

```
class Katua:
  def miaukatu(self):
    print ("Miau")
```

Ikus daitekeenez, klasea definitzeko, `class` hitza erabiliko dugu, eta, ondoren, klasearen izena, lehenengo letra larriz. Bloke horren barruan doan guztia klasearen parte izango da.

Bestalde, kontuan izan behar duzu klase baten funtzio **guztiek** `self` parametro berbera izan behar ditutela, nahiz eta ez erabili. Parametro hori klaseari berari dagokio, eta haren propietateei eta funtzioei erreferentzia egiteko erabiltzen da, gero ikusiko dugun bezala.

Klasea vs instantzia

Klasea katuaren definizioa besterik ez da. Baina gure programan katu bat sortzeko, **instantzia** bat sortu behar dugu. Honela egiten da:

```
katu = Katua()
katu.miaukatu()
```

Pantailan honako hauek ikusiko ditugu:

```
Miau
```

Instantzia objektu zehatz bat da. Klasea definizioa baino ez da: katu batek izena eta `miaukatu` funtzioa ditu. Instantzia objektu zehatz bat da: `katu`.

5.0 Ariketa

Idatzi `Pertsona` izeneko klase bat, lo egiteko, jateko eta agurtzeko metodoak dituen funtzioak definituko duen programa bat. Funtzio edo metodo bakoitzaren barruan testuren bat atera behar duzu kontsola bidez. Sortu klasearen instantzia bat eta deitu metodo desberdinetara.

```
class Pertsona:
    def loEgin(self):
        print("ZZZZZ...")

    def jan(self):
        print("Ñam, Ñam...")

    def kaixo(self):
        print("Kaixo, zer moduz!")

pertsona = Pertsona()

pertsona.loEgin()
pertsona.jan()
pertsona.kaixo()
```

Emaitza:

```
ZZZZZ...
Ñam, Ñam...
Kaixo, zer moduz!
```

Eraikuntza funtzioa

Lehenago definitu dugun katu horri izen bat emango diogu. Horretarako, funtzio berezi bat sortuko dugu, `__init___` izenekoa. **eraikuntza-funtzioa** esaten zaio (*constructor* ingelesez). Funtzio hau klaseko objektu bat sortzen denean deitzen da, eta, beraz, klasearen propietateak hasieratzeko lekurik egokiena da:

```
class Katua:
  def __init__ (self, izena):
    self.izena = izena

  def miaukatu (self):
    print("Miau,", self.izena, "naiz.")
```

Orain, `Katua` klaseko objektuak sortzen ditugunean, izen bat emango diogu, eta hau `izena` atributuan geratuko da:

```
katua = Katua("Pixi")
katua.miaukatu()

besteKatua = Katua("Txeto")
besteKatua.miaukatu()
```

Eta kasu honetan ikusiko dugu:

```
Miau, Pixi naiz.
Miau, Cheto naiz.
```

5.1 Ariketa

Idatzi programa bat `Kaixo` izeneko klase bat definitzeko. Klaseak funtzio eraikitzaile bat izan behar du, `agurra` izeneko atributu batekin. Atributu hori `"kaixo"` hitzarekin hasiko da. Horrez gain, `esanKaixo` izeneko metodo bat erantsiko duzu, eta, pantailan, agurraren edukia erakutsiko du.

```
class Kaixo:
    def __init__(self):
        self.agurra = "Kaixo"

    def esanKaixo (self):
        print(self.agurra)


kaixo = Kaixo()
kaixo.esanKaixo()
```

Emaitza:

```
Kaixo
```

Herentzia

Herentzia kodea berrerabiltzeko klaseek duten mekanismo bat da. Demagun katakume bat definitzen duen klase bat egin nahi dugula. Katu klaseak egiten duen gauza bera egitea nahi dugu, baina, gainera, zurrunga egitea. `Katakumea` klaseak `Katua` klasetik heredatu ahal izango luke, honela:

```
class Katakumea (Katua):
  def purrustatu (self):
    print ("Purrrr")
```

Orain honako hau egin dezakegu. `Katakumea` objektu bat sortzea, `Katua` klasearen propietate berberekin. Automatikoki jasoko ditu `izena` atributua eta `miaukatu` metodoa:

```
katutxoa = Katakumea ("Lucifur")
katutxoa.purrustatu()
katutxoa.miaukatu()
```

Honela ikusiko litzateke:

```
Purrrr
Miau, Lucifur naiz.
```

super()

Azpiklase bat edo klase bat beste baten *semea* sortzen duzunean, heredatzen duzun klasetik `super()` funtzioa erabil dezakezu oinordetzan hartutako klasearen funtzioei deitzeko. Adibidez, aurreko kasuan, `Katakumea` azpimailatik berezko eraikitzaile bat gehi genezake, eta `Katua` superklasearen eraikitzaileari ere dei diezaiokegu:

```
class Katakumea (Katua):
  def __init__ (self, izena):
    super () .__init__ (izena)
    print ("Katakumea sortua")

  def purrustatu (self):
    print ("Purrrr")
```

5.2 Ariketa

Idatz ezazu `Janaria` klasea definituko duen programa bat, `izena` atributuarekin. Sor ezazu `Fruitua` izeneko azpiklase bat, `Janaria` izena eta `bitaminak` jasotzen dituen eraikitzaile batekin, eta `info` izeneko metodo bat, bere informazio guztia itzultzen duena. Sortu instantzia bat `Fruitua` klasea probatzeko.

```
class Janaria:
    def __init__(self, izena):
        self.izena = izena


class Fruitua(Janaria):
```

```
    def __init__(self, izena, bitaminak):
        super().__init__(izena)
        self.bitaminak = bitaminak

    def info(self):
        # return f"{self.izena} {self.bitaminak}";
        return self.izena + " " + str(self.bitaminak)

postrea = Fruitua("Kiwi", ["A", "C"])
print(postrea.info())
```

Emaitza:

```
Kiwi ['A', 'C']
```

Enkapsulazioa

Katuaren adibidean, zuzenean sar daiteke `izena` propietatea. Horretarako, objektuetan nahikoa da horrelako zerbait jartzea:

```
objektua.AtributuIzena
```

Katuak `izena` izeneko propietate bat du.

```
nireKatua = Katua("Pixi")
print (nireKatua.izena) #Pixi
nireKatua.izena = "Pixel"
nireKatua.miaukatu () #Miau, Pixel naiz
```

Propietate bat hain zuzenean eskuratzea ez dago gaizki, baina programatzaile onak klasea *kapsulatzen* saiatzen dira. Zer esan nahi du horrek? Ezin direla zuzenean haren propietateak irakurri edo aldatu. Klasea erabiltzen dutenentzat beharrezkoa dena bakarrik eskeini behar zaie. Bestela esanda, programatzaileek "kutxa beltzak" diruditen klaseak sortzen saiatu behar dute. Izena bezalako propietateen kasuan, Python-en metodo hauek gehitu daitezke:

Izenaren jabetzaren balioa itzultzeko metodo bat, *getter* ere esaten zaiona:

```
  @property
  def izena ():
    return self._izena
```

Izenaren balioa aldatu ahal izateko metodo bat, *setter* ere esaten zaiona:

```
@izena.setter
def izena (izena):
  if izena! = "":
    self._izena = izena
```

Klasea honela geratuko litzateke:

```
class Katua:
  def __init__(self, izena):
    self._izena = izena

  @property
  def izena(self):
    return self._izena

  @izena.setter
  def izena(self, izena):
    if izena != "":
      self._izena = izena

  def miaukatu(self):
    print("Miau, ni naiz ", self.izena)
```

Ikus ezazu propietatea `izena` dela orain. Jabetza hori "pribatua" dela adierazteko modu bat da, eta ez litzatekeela klasetik kanpo eskuratu behar. Orain, `Katua` izeneko klasea erabiltzen dugunean, funtzio berri horien bidez egingo da.

```
nireKatua = Katua ("Pixi")
print (nireKatua.izena) # deitu `def izena ()` metodora
nireKatua.izena = "Pixel" #deitu `def izena (izena) metodora
nireKatua.miaukatu () # Miau, ni naiz Pixel
```

5.3 Ariketa

Idatz ezazu `Ibilgailua` klasea definituko duen programa bat, `matrikula` atributuarekin, *getter/setter* metodoekin eta `abiarazi` izeneko beste metodo batekin. Sortu `Kotxea` izeneko azpiklase bat, `Ibilgailua` klasea zabaltzeko, zein eraikitzaile batean `matrikula`, `modeloa` eta `kolorea` jasoko dituen, eta informazio guztia itzultzeko funtzio bat izango duena. Sortu instantzia bat `Kotxea` klasea probatzeko.

```
class Ibilgailua:
    def __init__(self, matrikula):
        self._matrikula = matrikula

    @property
    def matrikula (self):
```

```
        return self._matrikula

    @matrikula.setter
    def matrikula (self, matrikula):
        self._matrikula = matrikula

    def abiarazi (self):
        print("Abiarazten ", self._matrikula)


class Kotxea(Ibilgailua):
    def __init__(self, matrikula, modeloa, kolorea):
        super().__init__(matrikula)
        self._modeloa = modeloa
        self._kolorea = kolorea

    def info (self):
        return f"{self.matrikula} {self._modeloa} {self._kolorea}"


kotxea = Kotxea("0042ASI", "Opel Corsa", "Zuria")
kotxea.abiarazi()
print(kotxea.info())
```

Emaitza:

```
0042ASI Abiarazten
0042ASI Opel Corsa Zuria
```

Zer abantaila izan dezake enkapsulazioak?

Funtsean, "kanpotik" ezin da objektua kontrolik gabe manipulatu. Horregatik kutxa beltz bat bezalakoa da. Bideo-kontsola batekin konparatu daiteke. Joko bat jokatzeko eskuz ireki eta soldatu beharko bazenu, ziurrenik kontsola apurtuko zenuke. Horregatik, aparatuak kutxa beltz gisa diseinatzen dira, kanpotik manipulatzeko aukera batzuk baino ez dizute ematen. `Katua` klasearen kasuan, ez dugu uzten izena zuzenean aldatzen. *setter* funtzioaren bidez, esleitu nahi den izena zuzena dela kontrola dezakegu.

Klaseak klase barruan

Objektuetara zuzendutako programazioarekin, klaseen bidez mundu errealeko gauzak irudikatzen saiatzen gara. Eta klase horiek bata bestearekin lotuta egon daitezke.

Adibidez, demagun zure ikastola klaseen bidez errepresentatu nahi dugula. Ikastetxe batek gelak izan ditzake, gela batek ikasleak eta irakasleak izan ditzake, etab. Klaseek, beraz, beste mota batzuetako propietateak edo beste klase batzuetako zerrendak ere izan ditzakete.

```
class Ikaslea:
  def __init__ (self, izena):
    self.izena = izena

class Gela:
  def __init__ (self):
    self._ikasleak = []

  def sartuIkaslea (self, ikaslea):
    self._ikasleak.append (ikaslea)

  def listaPasa (self):
    for ikaslea in self._ikasleak:
      print (ikaslea.izena)


ikasle1 = Ikaslea ("Gumball")
ikasle2 = Ikaslea ("Darwin")

gela = Gela ()
gela.sartuIkaslea(ikasle1)
gela.sartuIkaslea(ikasle2)
gela.listaPasa()
```

Emaitza:

```
Gumball
Darwin
```

Diseinuak behar bezain konplexuak izan daitezke behar duguna irudikatzeko.

5.4 Ariketa

Idatz ezazu programa bat, `Pilotoa` eta `get/set` funtzioak dituena. Era berean, sor ezazu `Hegazkina` izeneko klase bat, `modeloa`, `pilotoa` eta `kopilotoa` atributuarekin, `get/set` funtzioekin, eta `info` metodo batekin atributuen balioak pantailaratzeko. Sortu instantzia bat bi klaseak probatzeko.

```
class Pilotoa:
    def __init__(self, izena):
        self._izena = izena

    @property
    def izena (self):
        return self._izena

    @izena.setter
    def izena (self, izena):
```

```
        self._izena = izena



class Hegazkina:
    def __init__(self, modeloa, pilotoa, koPilotoa):
        self._modeloa = modeloa
        self._pilotoa = pilotoa
        self._koPilotoa = koPilotoa

    @property
    def modeloa (self):
        return self._modeloa

    @modeloa.setter
    def modeloa (self, modeloa):
        self._modeloa = modeloa

    def info (self):
        return f"{self._modeloa} modeloa, {self._pilotoa.izena} eta
{self._koPilotoa.izena}rekin hegaldia egiten"


pilotoa1 = Pilotoa("Han Solo")
pilotoa2 = Pilotoa("Murdock")
hegazkinTxikia = Hegazkina("AirBluff 727", pilotoa1, pilotoa2)

print(hegazkinTxikia.info())
```

Emaitza:

```
AirBluff 727 modeloa, Han Solo eta Murdockekin hegaldia egiten
```

Metodo estatikoak

Normalean, klase bat erabili ahal izateko, haren instantzia bat sortzen dugu beti, aurreko adibidean egiten genuen bezala:

```
ikasleBat = Ikaslea ("Gumball")
```

Batzuetan, agian, kopiarik egin nahi ez dugun klase bat sortu nahi dugu, zeregin zehatz bat egiteko bakarrik balio duena. Funtzio bat balitz bezala.

Adibidez, izen bat emanda, formatu zuzena ematen dion klase bat egin dezakegu, lehenengo letra larriarekin eta gainerako letra xeheekin:

```
class Formatua:
  @staticmethod
  def zuzendu (izena):
    return izena [0] .upper () + izena [1:] .lower ()

print (Formatua.zuzendu("gUmBaLl"))
```

Emaitza:

```
Gumball
```

5.5 Ariketa

Idatzi programa bat, `Zenbakia` izeneko klase bat definituko duena, eta funtzio estatiko bat, `ausazkoa (max)` izenekoa. Funtzio honek zenbaki bat itzuli behar du `0` eta `max` tarteen barruan.

```
import random

class Zenbakia:
    @staticmethod
    def ausazkoa (max):
      return random.randint(0, max)

for i in range(5):
  print(Zenbakia.ausazkoa(10))
```

Emaitza:

```
4
3
0
9
1
```

ADI Zergatik egin klase bat horrelako funtzio batekin eta ez zuzenean funtzio batekin? klase batean egitea baliagarria izan daiteke leku berean funtzio estatiko bat edo gehiago sartu nahi ditugunean, eta ez ditugu instantzia desberdinak sortu nahi, funtzio zehatzak erabili baizik.

ADI Ez ahaztu klaseak eta objektuei zuzendutako programazioa diseinu estilo bat dela: kasu askotan, programaren erabilera errazteko eta kodea modu argiagoan banatzeko erabiltzen dira. Hasieran azaldu den bezala, arazoak konpontzeko bide bat eskeintzen digute.

Proposatutako ariketak

5.0 Ariketa

Idatzi `Instrumentua` izeneko klase bat definitzen duen programa bat. Eraikitzaileak parametro hauek izan behar ditu, hurrenez hurren: `izena` eta `mota`. Gainera, `jo` izeneko funtzio bat gehitu behar duzu, mezu bat aterako duena. Baita `info` funztioa, atributuen informazioa duen testu bat itzultzeko. Sortu klasearen instantzia bat eta deitu bere funtzioetara.

```
class Instrumentua:
    def __init__(self, izena, mota):
        self._izena = izena
        self._mota = mota

    def jo (self):
        print("Jotzen ", self._izena, "jotzen")

    def info (self):
        return f"{self._izena} {self._mota}"


nireGitarra = Instrumentua("Gitarra", "klasikoa")
nireGitarra.jo()
print(nireGitarra.info())
```

Emaitza:

```
Gitarra jotzen
Hari-gitarra
```

5.1 Ariketa

Idatz ezazu programa bat, `IzenZuzena` izeneko klase bat definituko duena, eta, bertan, `__init__` eraikitzailea bat `izena` eta `abizena` atributuak hasieratzeko. Gainera, klaseak `zuzendu` izeneko metodo bat izan behar du, `izena` eta `abizena` atributuak zuzenduko dituena. `zuzendu` metodoak, lehenengo hizkiak larri eta gainerakoak xehe itzuliko ditu.

```
class IzenZuzena:
  def __init__(self, izena, abizena):
    self._izena = izena
    self._abizena = abizena

  def zuzendu (self):
    return self._zuzendu(self._izena) + " " + self._zuzendu(self._abizena)


  def _zuzendu (self, hizkiak):
    return self._lehenengo(hizkiak) + self._gainera(hizkiak)
```

```
  def _lehenengo (self, hizkiak):
    return hizkiak[0].upper()


  def _gainera (self, hizkiak):
    return hizkiak[1:len(hizkiak)].lower()


zuzentzailea = IzenZuzena("JUAN", "PÉREZ")
print(zuzentzailea.zuzendu())
```

Emaitza:

```
Juan Pérez
```

5.2 Ariketa

Idatz ezazu programa bat `Batuketa`, izeneko klase bat definitzeko, eta bi zenbakirekin hasieratzeko.

```
Batuketa(28, 14)
```

Bientzako `get` eta `set` funtzioak barne hartzen ditu, eta balio negatiboa esleitzen saiatzen bazaizkie `0` esleitzea kontrolatu behar duzu. Gainera, bi zenbakien batura itzuliko duen `batu` funtzioa izango dute.

```
class Batuketa:
    def __init__(self, balio1, balio2):
        self.balio1 = balio1
        self.balio2 = balio2

    @property
    def balio1 (self):
        return self._balio1

    @balio1.setter
    def balio1 (self, balio1):
        if balio1 > 0:
            self._balio1 = balio1
        else:
            self._balio1 = 0

    @property
    def balio2 (self):
        return self._balio2

    @balio2.setter
    def balio2 (self, balio2):
```

```
        if balio2 > 0:
            self._balio2 = balio2
        else:
            self._balio2 = 0

    def batu (self):
        return self._balio1 + self._balio2


batuketa = Batuketa(28, 14)
print(batuketa.batu())

batuketa.balio1 = 600
batuketa.balio2 = 66
print(batuketa.batu())
```

Emaitza:

```
42
666
```

5.3 Ariketa

Sortu programa bat `Txanpona` izeneko klase batekin. Klaseak eraikitzaile huts bat izan behar du, eta funtzio bakar bat, `bota` izenekoa. Horren emaitza "aurpegiaren" edo "gurutzearen" artean ausaz aukeratutako *string* bat izan behar da. Sortu klasearen instantzia bat probatzeko.

```
import random

def ausazkoa (max):
    return random.randint(0, max)

class Txanpona:
    def bota (self):
        aldeak = ["aurpegi", "gurutze"]
        zenbakia = ausazkoa(1)

        return aldeak[zenbakia]

txanpona = Txanpona()

for i in range(10):
    print(txanpona.bota())
```

Emaitza:

```
aurpegia
aurpegia
gurutzea
gurutzea
gurutzea
...
```

5.4 Ariketa

Sortu programa bat N aurpegiko dado baten portaera simulatzeko, `Dadoa` izeneko klase batekin. Sortu klaseko instantzia bat jaurtiketak probatzeko.

- `def __init__ (self, aldeak, zeroOnartu = False)`: aurpegi-kopurua gordetzen duen atributua eta `zeroOnartu` boolearra: dadoak 0 balioa itzul dezakeen esaten digun atributua. Lehenetsitako balioa `False` da.
- `def aldeak (self, aldeak):` Parametrodun setter funtzioa. `aldeak` atributua ezartzen du.
- `def zeroOnartu (self):`: parametrodun setter funtzioa, boolear atributua ezartzen du.
- `def bota (self)`: funtzio honek dadoaren jaurtiketa simulatzen du eta emaitza bat itzultzen du. Kontuan hartu behar duzu "Onartu zeroa" atributua.

Sor itzazu 6 aurpegiko dado bat, 10 aurpegiko dado bat eta zeroak ahalbidetzen dituen 20 aurpegiko dado bat sortzen duten instantziak, eta egizu bakoitzetik 100 jaurtiketa.

```
import random

def ausazkoa (maximoa):
    return random.randint(0, maximoa)

class Dadoa:
    def __init__(self, aldeak = 6, zeroOnartu = False):
        self._aldeak = aldeak
        self._zeroOnartu = zeroOnartu

    @property
    def aldeak (self):
        return self._aldeak

    @aldeak.setter
    def aldeak (self, aldeak):
        self._aldeak = aldeak

    @property
    def zeroOnartu (self):
        return self._zeroOnartu

    @zeroOnartu.setter
    def zeroOnartu (self, zeroOnartu):
        self._zeroOnartu = zeroOnartu
```

```
    def bota (self):
        zenbaki =  ausazkoa(self._aldeak)

        if not self._zeroOnartu:
            zenbaki = zenbaki + 1

        return zenbaki



dadoa = Dadoa()
for i in range(10):
    print(dadoa.bota())
```

Emaitza:

```
4
3
2
4
3
...
```

5.5 Ariketa

Sortu bi klase dauzkan programa bat:

1- `Jokalaria` klasea, `izena`, `posizioa` eta `zenbakia` propietateak dituena. Parametro horiek guztiak dituen eraikitzaile bat ere sortu, eta ezaugarri guztiak itzuliko dituen `txosten` izeneko funtzio bat. 2 - `Taldea` klasea, `izena`, `fundazioa`, `aurrekontua` propietateekin eta `jokalariak`: `Jokalaria` motako instantziak gordetzeko propietatea (zerrenda izango dena). Honako ezaugarri hauek dituen eraikitzaile bat izan behar du. `Taldea` klaseak beste bi funtzio izango ditu: `jokalariaFitxatu` eta `jokalariakErakutsi`:

- `def jokalariaFitxatu (self, jokalaria)`, zerrendara jokalariak gehitzeko.
- `def jokalariakErakutsi (self)`, jokalarien datu guztiak jasotzen dituen kate bat itzultzeko

Gainera, gehitu bi jokalari eta talde bat sortzeko beharrezkoa den kodea, eta horri jokalariak gehitu eta erakutsiko dituzu.

```
class Jokalaria:
    def __init__(self, izena, posizioa, zenbakia):
        self._izena = izena
        self._posizioa = posizioa
        self._zenbakia = zenbakia

    def txosten (self):
```

```
        return f"{self._izena} {self._posizioa} {self._zenbakia}"


class Taldea:
    def __init__ (self, izena, sortzea, aurrekontua):
        self._izena = izena
        self._sortzea = sortzea
        self._aurrekontua = aurrekontua
        self._jokalariak = []

    def jokalariaFitxatu (self, jokalaria):
        self._jokalariak.append(jokalaria)

    def jokalariakErakutsi (self):
        for jokalaria in self._jokalariak:
            print(jokalaria.txosten())


jokalaria1 = Jokalaria("Maradona", "Aurrelari", 10)
jokalaria2 = Jokalaria("Beckenbauer", "Defentsa", 4)

print(jokalaria1.txosten())

ekipoa = Taldea("New Team", 1983, 39944.45)
ekipoa.jokalariaFitxatu(jokalaria1)
ekipoa.jokalariaFitxatu(jokalaria2)

ekipoa.jokalariakErakutsi()
```

Emaitza:

```
Maradona Aurrelaria 10
Beckenbauer Defentsa 4
```

5.6 Ariketa

Sortu ondorengo klaseak definitzen dituen programa bat:

1 - `Dispositiboa` klasea: `izena` eta `prezioa` atributuak ditu. Eraikitzaile bat atributuekin, `set` eta `get` eta `toString` funtzioak gehitu (atributuak erakusteko) 2 - `Mugikorra` klasea: `Dispositiboa` motako azpiklasea da, eta `zenbakia` atributua gehitu behar zaio. Sortu eraikitzailea eta `def toString (self)` metodoa, superklasekoak aprobetxatuz. Gehitu `def deitu (self, zenbakia)` funtzioa, pantailatik `"Deitzen [zenbakia]"` esaten duena. 3 - `Ordenadorea` klasea: `Dispositiboa` azpiklasea da, `prozesadorea` atributua gehitu behar zaio. Sortu eraikitzailea eta `def toString (self)` funtzioa superklasekoak aprobetxatuz Gainera, mugikorra eta ordenagailua sortzeko behar den kodea idatzi.

```python
class Dispositiboa:
    def __init__(self, izena, prezioa):
        self._izena = izena
        self._prezioa = prezioa

    def getIzena():
        return self._izena

    def setIzena(izena):
        self._izena = izena

    def getPrezioa():
        return self._prezioa

    def setPrezioa(prezioa):
        self._prezioa = prezioa

    def toString(self):
        return f"{self._izena} {self._prezioa}";


class Mugikorra(Dispositiboa):
    def __init__(self, izena, prezioa, zenbakia):
        super().__init__(izena, prezioa)
        self._zenbakia = zenbakia

    @property
    def zenbakia(self):
        return self._zenbakia

    @zenbakia.setter
    def zenbakia(self, zenbakia):
        self._zenbakia = zenbakia

    def toString(self):
        return f"{super().toString()} {self._zenbakia}"

    def deitu(zenbakia):
        print("Deitzen", zenbakia)


class Ordenagailua(Dispositiboa):
    def __init__(self, izena, prezioa, prozesadorea):
        super().__init__(izena, prezioa)
        self._prozesadorea = prozesadorea

    @property
    def prozesadorea(self):
        return self._prozesadorea

    @prozesadorea.setter
    def prozesadorea(self, prozesadorea):
        self._prozesadorea = prozesadorea
```

```
    def toString(self):
        return f"{super().toString()} {self._prozesadorea}"


ordenagailua = Ordenagailua("Dell", 4553.4, "Lentium 4")
telefonoa = Mugikorra("Chanmhung", 434.4, 665745345)

print("Ordenagailua:", ordenagailua.toString())
print("Telefonoa:", telefonoa.toString())
```

Emaitza:

```
Dell 4553.4 Lentium 4 ordenagailua
Chanmhung telefonoa 434.4 665745345
```

5.7 Ariketa

Txanogorritxo proiektua sortuko dugu, non protagonistak janari saski bat kudeatzen duen. Janari mota askotakoa izango da. Hauek dira egin beharreko klaseak:

1 - `Janaria` klasea: `izena` eta `pisua` atributuak ditu. Eraikitzaile bat atributuak, `set` eta `get` eta `toString` funtzio bat erabiltzen, atributuak erakutsiz. 2 - `Fruta` klasea: `Janaria` klasearen azpiklasea da, eta `bitamina` atributua gehitu behar zaio. Sortu eraikitzailea eta `toString` funtzioa superklaseko kodea berrerabiliz. 3 - `Goxokia` klasea: `Janaria` klasearen azpiklasea da, eta `kaloria` atributua gehitu behar zaio. Sortu eraikitzailea eta `toString` funtzioa, superklaseko kodea berrerabiliz. 4 - `Saskia` klasea, `janariak` izeneko atributua du, eta janari klaseko (edo azpiklaseko) elementu zerrenda bat da: `Futua` eta `Goxokia`. Eraikitzailean hasten da. Hiru funtzio ditu:

- `def sartuJanaria (hau, janari)` saskian janari bat sartzen du.
- `def pisuGuztira (hau)` saskiko janariaren guztizko pisua itzultzen du.
- `def toString (self )` saskiko janari guztia erakusteko.

Gainera, erantsi behar den kodea instantzia hauek sortzeko: `Fruta` eta `Goxokia` klaseen instantziak sortu, eta erantsi `Saskia` motako instantzia bati.

```
class Janaria:
    def __init__(self, izena, pisua):
        self._izena = izena
        self._pisua = pisua

    @property
    def izena (self):
        return self._izena

    @izena.setter
    def izena (self, izena):
        self._izena = izena
```

```
    @property
    def pisua (self):
        return self._pisua

    @pisua.setter
    def pisua (self, pisua):
        self._pisua = pisua

    def toString (self):
        return f"{self._izena} {self._pisua}"


class Fruta(Janaria):
    def __init__(self, izena, pisua, bitamina):
        super().__init__(izena, pisua)
        self._bitamina = bitamina

    @property
    def bitamina (self):
        return self._bitamina

    @bitamina.setter
    def bitamina (self, bitamina):
        self._bitamina = bitamina

    def toString (self):
        return f'{super().toString()} {self._bitamina}'


class Goxokia(Janaria):
    def __init__(self, izena, pisua, kaloria):
        super().__init__(izena, pisua)
        self._kaloria = kaloria

    @property
    def kaloria (self):
        return self._kaloria

    @kaloria.setter
    def kaloria (self, kaloria):
        self._kaloria = kaloria

    def toString (self):
        return f'{super().toString()} {self._kaloria}'


class Saskia:
    def __init__(self):
        self._janariak = []

    def sartuJanaria (self, janari):
        self._janariak.append(janari)
```

```
    def pisuGuztira (self):
        guztira = 0
        for janaria in self._janariak:
            guztira += janaria.pisua

        return guztira

    def toString (self):
        informazioa = ""
        for janaria in self._janariak:
            informazioa = informazioa + janaria.toString() + "\n"

        return informazioa


txintxa = Goxokia("Bomer", 0.2, 100)
gominoa = Goxokia("Marrubia", 0.3, 210)
udarea = Fruta("Udarea", 0.1, "B")
sagarra = Fruta("Sagarra", 0.15, "A")

saskia = Saskia()
saskia.sartuJanaria(txintxa)
saskia.sartuJanaria(gominoa)
saskia.sartuJanaria(udarea)
saskia.sartuJanaria(sagarra)

print("Saskiaren edukia:", saskia.toString())
print("Pisua guztira:", saskia.pisuGuztira())
```

Emaitza:

```
Saskiaren edukia: Bomer 0.2 100
Marrubia 0.3 210
Udarea 0.1 B
Sagarra 0.15 A

Pisua guztira: 0.75
```