Statisztikai Szemle

Közzététel: 2020. április 9.

A tanulmány címe:

A Python programozási nyelvről statisztikusoknak

Szerző:

SÓTI ATTILA, a Széchenyi István Egyetem PhD-hallgatója
E-mail: soti.attila@hallgato.sze.hu

DOI: https://doi.org/10.20311/stat2020.4.hu0324

Sóti Attila

A Python programozási nyelvről statisztikusoknak

On the Python programming language for statisticians

SÓTI ATTILA, a Széchenyi István Egyetem PhD-hallgatója
E-mail: soti.attila@hallgato.sze.hu

A cikk megírásánál az a cél vezérelte a szerzőt, hogy egy olyan bevezető leírást adjon a Python programozási nyelvről, amely könnyen érthető a programfejlesztésben még kezdő, de a statisztikában már jártas szakember számára. A tanulmány lépésről lépésre mutatja be a Python nyelvű programozás szépségeit, felvázol egy utat, amelyen járva könnyen eljutunk egy futtatható kódig, de egyben felhívja a figyelmet a Python nyelvben rejlő buktatókra is. A példaként használt statisztikai módszerek többsége szándékosan alapszintű, egyetlen kivétel a klaszteranalízis. Utolsó lépésként a szerző rámutat az adatvizualizáció fontosságára, és példákon keresztül ismerteti a Python programnyelv lehetőségeit ezen a téren is.

TÁRGYSZÓ: Python, statisztika, klaszteranalízis

This study provides easily accessible introduction to the Python programming language for readers who are beginners in software development but proficient in statistics. The tutorial presents the beauty of Python programming, outlines a path to an easily executable code and also highlights pitfalls of Python. Most of the statistical methods used as an example are deliberately basic, with the exception of cluster analysis. As a final step, the importance of data visualization is highlighted and examples of the possible use of Python in this area are given.

KEYWORD: Python, statistics, cluster analysis

A Python egy több platformon futtatható (Unix, macOS, BeOS, NeXTSTEP, MS-DOS és Windows), könnyen bővíthető, ingyenes programnyelv, amely lehetővé teszi, hogy a programunkat előre megírt elemekből (modulokból) állítsuk össze (*Payne* [2010]). Így más programozási nyelvekhez képest – ahol mindent magunk-

nak kell megírnunk – lényegesen gyorsabban haladhatunk a programfejlesztés során (*Ayer–Miguez–Toby* [2014], *Müllner* [2013]). A Python történetéről annyit kell tudnunk, hogy *Guido van Rossum* és más önkéntesek fejlesztik 1989 óta (*van Rossum–Drake* [2009]). Egy jól kiforrott, de folyamatosan fejlődő programnyelvnek számít, amelyben – a legújabb források szerint – napjainkban a programkódok egy jelentős része készül.[1] A Python nyelvfejlesztés filozófiájának központi része a programkód megértésének megkönnyítése és az egyszerűség megőrzése (*Pilgrim* [2009]). A programozók ezt a viszonylag egyszerű nyelvet összetett problémák megoldására is használhatják (*Oliphant* [2007]). A Python alkalmazásának nem alapfeltétele komplex absztrakciók megértése (*Python Foundation* [2016]) Például a Python automatikusan kezeli az erőforrásokat (memóriát, fájlkezelőket stb.), nincsenek benne mutató adattípusok (pointerek), valamint egy könnyen átlátható kivétel-kezelőrendszerrel van ellátva. Ez lényegesen elősegíti a hibakezelést (*Rossum* [2017]). Előnye még, hogy a Python különböző verziói (Windows-ra, Unixra stb.), oktatóanyagai, kézikönyve, a modulok dokumentációi stb. ingyen letölthetők a hivatalos Python weboldalról (http://www.python.org.)

1. A programfejlesztésről röviden

A programozó a statisztikusokhoz hasonlóan formális nyelvet, képleteket használ az okfejtések leírására *(Moll–Arbib–Kfoury* [1988]). A számítógép rendelkezik egy eljárással, amely úgy dekódolja ezeket a formális nyelven megírt utasításokat, hogy a nyelv minden eleméhez (szavaihoz) egy előre meghatározott akciót rendel. Az a nyelv, amelyet a számítógép megért 1-eskből és 0-ákból áll (gépi kód). Olyan fordítórendszereket kell alkalmaznunk, amelyek a számunkra érhető kulcsszavakat gépi kódra fordítják. Azt a programot, amelyet mi írunk, forrásprogramnak vagy forráskódnak hívjuk. Két módszert alkalmazhatnak a fordítórendszerek arra, hogy a mi forrásprogramunkat átváltsák gépi kódra: az egyik az interpretálás (értelmezés) a másik a compilálás (*Downey–Elkner–Meyers* [2013]). Az interpretálás esetén, amikor végre akarjuk hajtatni a forráskódunkat, egy értelmező (interpreter) programot kell használunk. Ilyen esetben az értelmező a forrásprogram kívánt sorait gépi nyelvű utasításokra fordítja le, amelyeket azonnal végre is hajt. Nem keletkezik másik futtatható (exe) program (*Rossum* [2016a]). A compiler viszont a teljes forráskód egyszeri lefordításából áll. A fordítóprogram a forrásprogram minden sorát elolvassa,

[1] http://pypl.github.io/PYPL.html

és egy új programot állít elő, amelyet tárgykódnak nevezünk. Ez a kód már végrehajtható, és egy külön exe (execute) kiterjesztésű fájlban tárolható.

Mindkét módszernek vannak előnyei és hátrányai. Az interpretáció előnyös, mikor egy programnyelvet tanulunk, mert azonnal tudjuk futtatni a forráskódunk egyes részleteit, valamint azonnal láthatóvá válik az eredmény. Ezt a módszert fogjuk alkalmazni a cikkünk során is (*Janssens* [2014]). Ezzel szemben a fordítást akkor szokták előnyben részesíteni, amikor gyorsabb működésre van szükség. A Python megpróbálja ezt a két módszert kombinálni. Amikor forráskódot futtatunk egy Python fejlesztőkörnyezetében, ez először ún. közbenső kódot – *bytecode*-t – hoz létre, amelyet azután az értelmező végre is hajt. Mivel mi az értelmező segítségével futtatjuk a forráskódunkat, ez lehetővé teszi minden egyes kicsiny programrész közvetlen tesztelését (*Janssens* [2014], *Lutz* [1999]). Ez a módszer elsőre talán összetettnek tűnhet, de nem kell aggódunk, mert a Python fejlesztőkörnyezet ezt automatikusan végrehajtja, nekünk csak a futtatás (*run*) gombra kell kattintanunk.

A programfejlesztés egy bonyolult, több lépésből álló folyamat, amelynek során előfordul, hogy hibákat követünk el. Három típusú hibát különböztetünk meg (*Downey–Elkner–Meyers* [2013]).

– *Szintaktikai hiba*. Ha nem a programozási nyelv szabályai szerint írjuk le a mondatainkat kódírás során, majd ezt megpróbáljuk lefordítani, az értelmező azonnal hibát jelez. Egy rövid üzenettel rávilágít a hiba lehetséges okára is.

– *Szemantikai hiba*. Ilyen típusú hiba esetén a program lefut, csak nem azt az eredményt kapjuk, amit vártunk. Az ilyen logikai hibát már nehezebb megtalálni. Elemezni kell a program eredményét és reprodukálni azokat a műveleteket, amelyeket a programunk végrehajtott.

– *A program végrehajtása közben fellépő hiba (run time error).* Ezen típusú hibák akkor keletkeznek, amikor nem várt körülmények állnak elő a programunk futtatása közben. Például, amikor a programunk egy olyan fájlt szeretne olvasni, amelyhez nincs megfelelő jogosultsága, vagy már nem létezik. Ebben az esetben is az értelmező hibaüzenetekkel segít nekünk megtalálni a hiba okát.

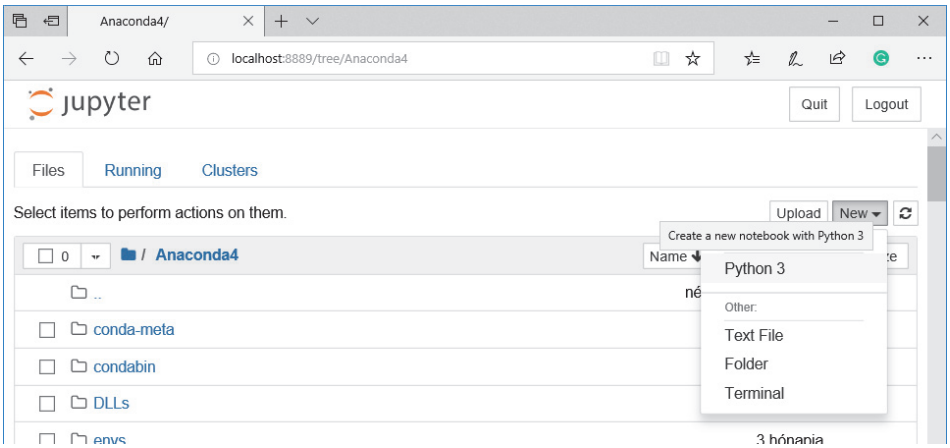
2. A Python

A Szoftverek a statisztikában rovat R-rel foglalkozó cikke (*Hajdu* [2018]) többváltozós statisztikai problémákat dolgozott fel. Ez az írás viszont – bevezető jellege miatt – többnyire egyszerűbb elemzési eszközöket mutat be Pythonban.

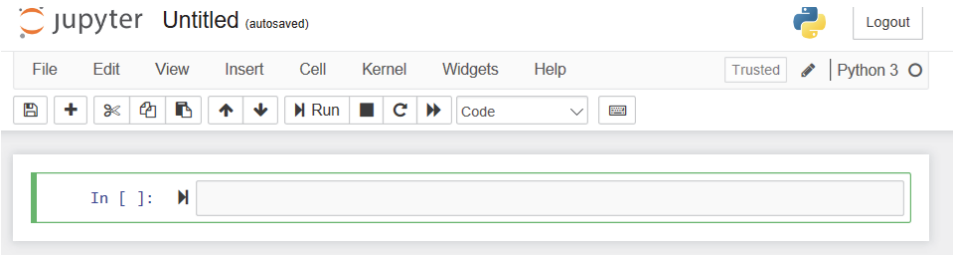
2.1. A keretrendszer telepítése

A Pythonhoz különböző platformokra (Windows, Linux stb.) többfajta keretrendszer létezik. A cikkben bemutatott példákat a Jupyter Notebook kertrendszerben futtattuk, amely ingyenesen telepíthető. (A telepítés leírása a https://jupyter.readthedocs.io/en/latest/install.html weblapon található). Más keretrendszerek is használhatók (Spyder vagy PythonWin). Az aktuális verziókért a Python hivatalos weboldalát (https://www.python.org/downloads/) érdemes böngészni (*Rossum* [2016b]). Mi a Python 3.7.4 verzióját használtuk – amit 2019. július 8-án adtak ki –, amely elég kiforrott és stabil a programjaink futtatásához. A telepítés befejezésével a start menüből a Jupyter Notebook programot indítva a következő böngészőablak jelenik meg. A fejlesztőkörnyezetünk eléréséhez válasszuk a képernyő jobb oldalán levő New/Python 3 menüpontot.

1. ábra. A Jupyter Notebook kezdő ablaka
(First screen of the Jupyter Notebook)



2. ábra. A Jupyter Notebook fejlesztő ablaka
(Jupyter Notebook developer window)



2.2. A Python felépítése

A Python különböző modulokból (*package*-ből) áll. Ezeket egymástól független fejlesztői csoportok (önkéntesek) írják (*Python Software Foundation* [2019]). A modulok tehát olyan fájlok, amelyek különböző függvénycsoportokat foglalnak egybe. Egy Python program általában egy főprogramból és egy vagy több modulból áll, amelyek a bennük levő függvények definícióját tartalmazzák (*Downey–Elkner–Meyers* [2013]). Például a *math* modul olyan matematikai függvények definícióját tartalmazza (*Network Security* [2015]), mint a szinusz-, koszinusz-, négyzetgyök-függvény stb. Ha ezeket a függvényeket szeretnénk használni, elég csak a programunkba beszúrni a következő sort:

from math import * vagy röviden *import math*

Ez jelzi a keretrendszernek, hogy az aktuális forrásprogramba bele kell foglalnia a *math* modul minden függvényét (ezt jelenti a *). Arra is lehetőség van, hogy az import után felsoroljuk azokat a függvényeket, amelyeket a modulból használni szeretnénk. Ilyenkor csak a felsorolt függvények definíciója lesz része a programunknak. Pythonban a függvény valamilyen név és a hozzá tartozó zárójelek formájában jelenik meg. A zárójelben egy vagy több argumentumot adunk meg (*Python Software Foundation* [2019]). A függvénynek mindig van legalább egy visszatérési értéke. (Például, ha $a = abs(-1)$ [ahol *abs* az abszolútérték-függvény], akkor az *a* változó[2] értéke 1 lesz.) A modulokat egymástól független fejlesztők készítik, ezért gyakran megesik, hogy a függvények különböző modulokban más-más módon valósulnak

[2] A Pythonban levő változók névadásának szabályairól, a változók típusairól, valamint arról, hogy miként adunk nekik értéket, részletes leírást találunk magyarul *Gérard Swinenn*: „Tanuljuk meg programozni Python nyelven” című könyvében (https://mek.oszk.hu/08400/08435/08435.pdf).

meg (*Downey–Elkner–Meyers* [2013]). Tanulmányunk egyik célja, hogy összefoglalja a Pythonban meglevő statisztikai függvényeket, mint például az átlagot, a szórást, a bonyolultabbak közül pedig a korrelációt vagy a lineáris regressziót. Példákon keresztül illusztráljuk, hogyan tudjuk használni ezeket, és felhívjuk az olvasó figyelmét az eltérő megvalósítások között található esetleges különbségekre.

2.3. A statisztikai függvényeket tartalmazó modulok

A Python nyelv folyamatosan fejlődik, ezért bármennyire is törekszünk felsorolni az összes általunk ismert és használt modult, ez a felsorolás soha nem lehet teljes (*Menczer–Fortunato–Davis* [2020]). Ha bármilyen függvény vagy meglevő modul kimaradt a felsorolásból, illetve időközben esetleg új modul keletkezik, a szerző szívesen fogadja az ezen esetekről szóló kiegészítéseket.

A Pythonban öt olyan modul van tudomásunk szerint (math, statistics, numpy, scipy.stats, pandas) (*Shell* [2014]), amelyek általános statisztikával foglalkoznak. Ezeket importáljuk forrásprogramunkba a már ismert módon.

3. ábra. Modulok importálása
(Importing modules)

```
In [ ]:
import numpy
import statistics
import scipy.stats
import pandas
import math
```

2.4. Az átlag

Ahhoz, hogy különböző átlagokat (*Briscoe* [2011]) vagy szórást tudjunk számítani, adatokra van szükségünk. Figyelembe kell venni azt is, hogy az adatkészlet tartalmazhat hiányzó értékeket (hiányzó vagy sérült adatot). A *numpy*, *scipy* és *pandas* modulokban az ilyen adat szokásos ábrázolási módja a „*nan*” (not-a-number – nincs érték) (*Menczer–Fortunato–Davis* [2020]). A példaprogramjainkban a legegyszerűbb módon fogjuk megtenni az adatbevitelt, azaz kézzel begépelünk pár adatot, és ezeken fogjuk bemutatni a függvények működését. A változó adattípusát, amelyben ezt beírjuk, sornak nevezzük.

Erről az adattípusról annyit kell tudni, hogy csak számokat vagy hiányzó értékeket tartalmazhat. Úgy rögzítjük, hogy beírunk egy tetszőleges változónevet (példánkban *szam_sor*), majd az egyenlőségjel után szögletes zárójelben, vesszővel elválasztva felsoroljuk a számokat vagy az üres értékeket. (Lásd a 4. ábrát.) A számok lehetnek egészek vagy tizedes törtek is. Az üres érték pedig már az említett *nan*, ami azt jelenti, hogy nem tudom mi van ott, vagyis még véletlenül se keverendő össze a „0” vagy a semmi értékkel.

4. ábra. Adatrögzítés és átlagszámítás
(Data recording and average calculation)

```
szam_sor=[1,5,9,14,3]
szam_sor_nan_al=[1,5,9,14,nan,3]
statistics.mean(szam_sor)
```

```
6.4
```

```
statistics.mean(szam_sor_nan_al)
```

```
nan
```

```
numpy.mean(szam_sor)
```

```
6.4
```

```
numpy.mean(szam_sor_nan_al)
```

```
nan
```

```
numpy.nanmean(szam_sor_nan_al)
```

```
6.4
```

```
pandas_szam_sor_nan_al=pandas.Series(szam_sor_nan_al)
pandas_szam_sor_nan_al.mean()
```

```
6.4
```

Elemezzük a 4. ábra programsorait! Az első két sorban két sortípusú változónak adtunk értéket (*szam_sor* és *szam_sor_nan_al*). Az elsőben csak számok vannak, a másodikban van egy *nan* (nem tudom) érték. Az elsőnek kiszámítjuk az átlagát úgy, hogy meghívjuk a *statistics* modul *mean*() (átlag) függvényét, és paraméterként átadjuk neki a *szam_sor* változót. A Python kiszámítja a számsorban levő számok átlagát (6,4). A második esetben, amikor *nan* érték van a számsorban, a Python az átlagra a *nan* választ adja. Ez azt jelenti: ha nem tudom, mi van ott, nem tudom az

átlagot sem kiszámítani. Ha bármelyik függvényünk *nan* értéket ad vissza, gyanakodjuk arra, hogy a bemeneti adatok között van *nan* érték. Mint már említettük, a függvények elnevezése és működése között is vannak átfedések. Például a *numpy* modulban is van *mean*() függvény, amely hasonlóan működik, mint a *statistics* modul *mean*() függvénye. Azonban a *numpy* modulban van egy másik függvény is *numpy.nanmean*() (lásd a 4. ábrát), amely akkor is kiszámítja az átlagot, ha *nan* érték található a számsorban.

A *pandas* mondulban levő *mean*() függvényt viszont csak úgy tudjuk meghívni, ha először az adatunkat a *pandas* csomagban használatos *series* típusúvá alakítjuk, majd a változóhoz hozzárendelt *mean*() függvény a 4. ábrának megfelelően (ez a típusú függvényhívás is elfogadott a Pythonban) kiszámítja az átlagot. A *pandas*-ban levő *mean*() függvény már alapból tudja kezelni a *nan* értéket tartalmazó adatokat is.

2.5. A súlyozott átlag

A súlyozott átlag – amelyet súlyozott számtani középnek vagy súlyozott aritmetikai átlagnak is nevezünk – a számtani átlag általánosítása. A súlyozott átlagot akkor használjuk, ha az egyes értékek egynél többször fordulnak elő vagy különböző fontosságúak (súlyúak). Legyen *p* súly, amit hozzá kell rendelni a *szam_sor* változó minden értékéhez.

5. ábra. Súlyozott átlag számítása
(Calculation of the weighted average)

```
szam_sor=[1,5,9,14,-3]
p=[0.2,0.3,0.4,0.3,0.7]
numpy.average(szam_sor,weights=p)
```

```
3.8947368421052637
```

```
pandas_szam_sor=pandas.Series(szam_sor)
panads_p=pandas.Series(p)
```

```
(pandas_szam_sor_nan_al*panads_p).sum()/panads_p.sum()
```

```
3.8947368421052637
```

Ahhoz, hogy kiszámítsuk a súlyozott átlagot, szükségünk van a *szam_sor* változóra és *p* súlyváltozóra, valamint a *numpy* csomagban levő *average*()

függvényre. (Lásd az 5. ábrát.) A függvénynek átadjuk a *szam_sort* mint paramétert, és megadjuk (*weigths = p*), melyik súlyváltozót használja. Ha a *pandas* modult szeretnénk alkalmazni a súlyozott átlag számításához, akkor előbb át kell alakítanunk a változókat *pandas.series* formára majd *sum*() összeg függvényhívással kiszámítjuk a sorozat összegét és elosztjuk a súlyok összegével. (Lásd az 5. ábrát.)

2.6. A harmonikus közép

A harmonikus közép a számok reciprokaiból számított számtani közép reciproka (*Wilson* [2019]).

Ezt a *statistics* modul *harmonic_mean*() függvényével tudjuk kiszámítani. Erről a függvényről tudni kell, hogy ha van egy *nan* érték a bemeneti adatok között, akkor *nan* értéket ad vissza. (Lásd a 6. ábrát.) Ha van legalább egy 0, akkor 0-át, ha van egy negatív szám, akkor pedig statisztikai hibát, azaz „*StatisticsError*”-t fog kiírni a képernyőre. A *scipy.stats* csomag *hmean*() függvénye abban az esetben, ha *nan*, 0 vagy negatív érték van a bemeneti adatok között „*ValueError*” üzenetet fog adni.

6. ábra. Harmonikus közép számítása
(Calculation of the harmonic mean)

```
statistics.harmonic_mean(szam_sor)
```

```
2.913968547641073
```

```
statistics.harmonic_mean(szam_sor_nan_al)
```

```
nan
```

```
scipy.stats.hmean(szam_sor)
```

```
2.913968547641073
```

```
scipy.stats.hmean(szam_sor_nan_al)
```

```
C:\Users\Attila\Anaconda4\lib\site-packages\scipy\stats\stats.py:393: RuntimeWarning: invalid value encountered in greater_
equal
  if np.all(a >= 0):

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

2.7. A mértani közép

A mértani közepet a *scipy.stats* csomag *gmean*() függvényével tudjuk kiszámítani. (Lásd a 7. ábrát.) Ha van egy *nan* érték a bemeneti adatok között, akkor *nan* értéket ad vissza. Ha van legalább egy 0, akkor 0-át, ha van egy negatív szám, akkor pedig egy futásidejű figyelmeztetést (*RuntimeWarning*) ír ki. (Lásd a 7. ábrát és a hibák típusai leírást.)

7. ábra. Mértani közép számítása
(Calculation of the geometric mean)

```
scipy.stats.gmean(szam_sor)
```

```
4.5216022974376004
```

```
scipy.stats.gmean([-1,2])
```

```
C:\Users\Attila\Anaconda4\lib\site-packages\scipy\stats\stats.py:330: RuntimeWarning: invalid value encountered in log
  log_a = np.log(np.array(a, dtype=dtype))
```

```
nan
```

2.8. A medián – helyzeti középérték

Ha a sor elemeit nagyság szerint sorba rendezzük, és a középső elemet vesszük, az a medián. Ha a sor elemeinek száma páratlan, akkor ez egyértelmű. Páros számú elem esetén a két középső elem számtani közepét kell vennünk, ez lesz a medián.

A *statistics* modul *median*() függvénye segítségével számítható ki a medián értéke. A 8. ábra alapján a *szam_sor* változó elemei sorba rendezve: [1,3,5,9,14]. A számsor közepén az 5 szerepel, és a *median*() függvény is ezt az eredményt adja. (Lásd a 8. ábrát.) A függvény jól működik, amíg nincs *nan* az elemek között, ha van, akkor viszont nem *nan*-t kapunk, hanem rossz eredményt. Nézzük a 8. ábra példáját: *szam_sor_nan_al* sorba rendezve [1,5,9,14,nan,3] – ilyenkor a *nan* utáni 3-t nem veszi figyelembe (!), ezért van középen a 9 és a 14 – amelynek átlaga 11,5. Ezért a *statistics* modul *median*() függvény használata a bizonytalan működés miatt nem ajánlott. Fontos megjegyezni, hogy páros elemszám esetén van értelme kiszámolni a *median_low*() és a *median_high*() függvényeket. A két függvény, a sorba rendezés után, a sor közepétől jobbra és balra található elemet adja eredményül. Példánkban a 9-t és a 14-t. (Lásd a 8. ábrát.)

8. ábra. Medián számítása (statistics modul)
(Calculation of the median [*statistics* module])

```
szam_sor=[1,5,9,14,3]
szam_sor_nan_al=[1,5,9,14,nan,3]
```

```
statistics.median(szam_sor)
```

```
5
```

```
statistics.median(szam_sor_nan_al)
```

```
11.5
```

```
statistics.median_low(szam_sor_nan_al)
```

```
9
```

```
statistics.median_high(szam_sor_nan_al)
```

```
14
```

9. ábra. Medián számítása (numpy és pandas modul)
(Calculation of the median [*numpy* and *pandas* modules])

```
numpy.median(szam_sor)
```

```
5.0
```

```
numpy.median(szam_sor_nan_al)
```

```
nan
```

```
numpy.nanmedian(szam_sor_nan_al)
```

```
5.0
```

```
pandas_szam_sor_nan_al.median()
```

```
5.0
```

A *numpy* modulnak is van mediánfüggvénye. (Lásd a 9. ábrát.) Ez a függvény már az elvártaknak megfelelően működik *nan*-t tartalmazó sor esetén is (természetesen *nan*-t add vissza). A modulnak van egy *nanmedian*() függvénye, amely nem veszi figyelembe a *nan* értéket, ennek használata ajánlott. (Lásd a 9. ábrát.) A *pandas* modulnak is van *median*() függvénye, amely a *nan* értéket egyáltalán nem veszi fi-

gyelembe, és hibátlanul kiszámítja a számtani középértéket, így használata ajánlott. (Lásd a 9. ábrát.)

2.9. A módusz

A módusz egy adatsor leggyakrabban előforduló eleme.

A *statistics* modul *mode*() függvénye megmutatja a leggyakoribb elemet, viszont hibát ír ki, ha több elem is megfelel a módusz definícióinak („StatisticsError: no unique mode; found 4 equally common values”), ezért kerüljük a használatát. (Lásd a 10. ábrát.) A *scipy.stat* modul *mode*() függvénye ilyen esetben is jól működik, nem ír ki hibát, hanem a legtöbbet előforduló elemek közül a legkisebb elemet írja ki és annak előfordulását. (Lásd a 10. ábrát.) A *pandas* modul is rendelkezik *mode*() függvénnyel, amely jól kezeli a multimodális értékeket, és alapértelmezés szerint figyelmen kívül hagyja a *nan* értékeket.

10. ábra. Módusz számítása
(Calculation of the mode)

```
statistics.mode([1,2,3,4,4,4,4])
```

```
4
```

```
statistics.mode([1,2,3,4])
```

```
---------------------------------------------------------------------------
StatisticsError                           Traceback (most recent call last)
<ipython-input-435-925285040383> in <module>
----> 1 statistics.mode([1,2,3,4])

~\Anaconda4\lib\statistics.py in mode(data)
    504     elif table:
    505         raise StatisticsError(
--> 506                 'no unique mode; found %d equally common values' % len(table)
    507                 )
    508     else:

StatisticsError: no unique mode; found 4 equally common values
```

```
scipy.stats.mode([1,2,3,4,4,4,4])
```

```
ModeResult(mode=array([4]), count=array([4]))
```

```
scipy.stats.mode([1,2,3,4])
```

```
ModeResult(mode=array([1]), count=array([1]))
```

2.10. A szórásnégyzet (variancia) és a szórás (standard deviation)

A szórásnégyzet (variancia) az átlagtól való eltérések négyzetének átlaga, a szórás (standard deviation) pedig ezen értéknek a gyöke.

A *statistics* modulban a *variance*() függvény szolgáltatja a variancia értékét, viszont itt is fellép a már többször emlegetett probléma, hogy a *nan*-t tartalmazó adatsornak *nan* lesz a varianciája is. (Lásd a 11. ábrát.) A *numpy* modulnak a *var*() függvény a szórásnégyzetfüggvénye. Ebben az esetben meg kell adnunk még egy paramétert, a *ddof*-ot (delta degrees of freedom), amely a szabadságfokot állítja be. (Lásd a 11. ábrát.) Itt is a *nan*-t tartalmazó adatsornak *nan* lesz a varianciája, de a *nanvar*() függvény figyelmen kívül hagyja a *nan* értéket. (Lásd a 11. ábrát.) A *pandas* modul is rendelkezik a *var*() függvénnyel. Alapértelmezés szerint figyelmen kívül hagyja a *nan* értékeket, és a *ddof* paraméter alapértelmezésben 1 értéket vesz fel, tehát nem kötelező megadnunk. (Lásd a 11. ábrát.) Ha 0 szabadság fokkal (populációvariancia) szeretnénk számolni, akkor a *statistics* modul *pvarinace*() függvényét kell meghívni, a *numpy* és a *pandas* modul esetében pedig a *ddof* paramétert kell 0-ra állítani.

11. ábra. Variancia számítása
(Calculation of the variance)

```
statistics.variance(szam_sor)
```
```
26.8
```
```
statistics.variance(szam_sor_nan_al)
```
```
nan
```
```
numpy.var(szam_sor,ddof=1)
```
```
26.8
```
```
numpy.var(szam_sor_nan_al,ddof=1)
```
```
nan
```
```
numpy.nanvar(szam_sor_nan_al,ddof=1)
```
```
26.8
```
```
pandas_szam_sor.var()
```
```
26.8
```

A szórásfüggvény *stdev*() vagy *std*() működése megegyezik az ismertetett varianciafüggvény működésével. Ezért mind a három modul esetében csak röviden mutatjuk be a működést. A *statistics* modulban a *stdev*() függvény nem tud a *nan* értékkel számolni. A *numpy* modulban a *std*() és a *nonstd*() függvénynek meg kell adni a *ddfo* = 1 papramétert. A *pandas* csomag *std*() függvénye figyelmen kívül hagyja az esetleges *nan* értékeket. A *ddof* paraméter alapértelmezésben 1 értéket vesz fel. Ha 0 szabadság fokkal (populációszórás) szeretnénk számolni, akkor a *statistics* modul *pstdev*() függvényét kell meghívni, a *numpy* és a *pandas* modul esetében pedig a *ddof* paramétert kell 0-ra állítani.

2.11. A ferdeség (skewnees)

A ferdeség az eloszlás csúcsának a középhelyzethez képest történő eltolódását fejezi ki.

12. ábra. Ferdeség számítása
(Calculation of skewness)

```
scipy.stats.skew(szam_sor, bias=False)
```

```
0.7719466374492348
```

```
pandas_szam_sor.skew()
```

```
0.7719466374492349
```

A *scipy.stats* modulban a *skew*() függvénnyel tudjuk kiszámítani, de fontos, hogy a *bias*, vagyis torzítás paramétert *False* értékre állítsuk. (Lásd a 12. ábrát.) A *pandas* modulnak is van *skew*() függvénye. Szokásos módon, azaz paraméter nélkül kell meghívni, valamint a *nan* értékekkel is jól számol. (Lásd a 12. ábrát.)

2.12. Percentilisek (percentiles)

A percentilisek is definiálhatók, ha az adatok csak egy bizonyos részére vagyunk kíváncsiak. Például az *n* százalékos (vagy *n*-edik) percentilis azt jelenti, hogy az adatok *n* százaléka kisebb, mint a kapott érték (*Szeptycki* [2004]).

A *numpy* modulból a *percentile*() és *nanpercentile*() függvényt tudjuk erre a célra használni. Ebben az esetben egész számként kell a százalékot megadni a függvény

paraméterének. (Lásd a 13. ábrát.) A *pandas* modulban a *quantile*() függvény használható ugyanerre a célra. Itt viszont a második paraméternek a [0,1] intervallumból való számnak kell lennie.

13. ábra. Percentilisek számítása
(Calculation of percentiles)

```
numpy.percentile(szam_sor, 10)
```

```
1.8000000000000003
```

```
numpy.nanpercentile(szam_sor_nan_al,50)
```

```
5.0
```

```
pandas_szam_sor.quantile(0.1)
```

```
1.8000000000000003
```

```
pandas_szam_sor.quantile(0.5)
```

```
5.0
```

2.13. Összefoglaló leíró statisztika

Olyan esetben, ha csak információra van szükségünk az adatokról, de a kapott értékekkel nem szeretnénk tovább számolni, összefoglaló leíró statisztikákat tudunk kérni a Pythonban.

14. ábra. Összefoglaló leíró statisztikák
(Descriptive statistics)

```
scipy.stats.describe(szam_sor)
```

```
DescribeResult(nobs=5, minmax=(1, 14), mean=6.4, variance=26.8, skewness=0.5178375469016621
```

```
pandas_szam_sor.describe()
```

```
count     5.000000
mean      6.400000
std       5.176872
min       1.000000
25%       3.000000
50%       5.000000
75%       9.000000
max      14.000000
dtype: float64
```

A *scipy* és a *pandas* modulok *describe*() függvénye ad leíró statisztikákat az adatainkról, amelyek az eddig felsorolt értékeket (számosság, átlag, szórás stb.) tartalmazzák. (Lásd a 14. ábrát.)

2.14. Két adatsor kapcsolata – korreláció

Két mennyiség kapcsolatának feltárása a mindennapi életben is nagy jelentőségű. Az ilyen kapcsolatok számszerűsítése különösen fontos napjainkban. A korreláció-számítás két adatsor lineáris kapcsolatának intenzitását és irányát méri. A Python segítségével háromféle lineáris korreláció (Pearson-, Spearam- és Kendall-féle tau**)** számítható.

15. ábra. Korreláció számítása (numpy)
(Calculation of correlation [*numpy*)]

```
szam_sor_elso=[1,5,9,14,3,4,7 ,9]
szam_sor_masodik=[1,4,5,12,9,14,3,6]
```

```
numpy.corrcoef(szam_sor_elso,szam_sor_masodik)
```

```
array([[1.        , 0.31244047],
       [0.31244047, 1.        ]])
```

```
numpy.corrcoef(szam_sor_elso,szam_sor_masodik)[0, 1]
```

```
0.31244047052046187
```

A *nupmy* modul *corrcoef*() függvénye egy korelációs mátrixot ad vissza, amelynek a fő átlóján 1-esek vannak, a mellék átlón pedig a korrelációs együttható. (Lásd a 15. ábrát.) A Pythonban a mátrixok sorának és oszlopának számozása 0-tól kezdődik.

A *scipy.stats* modulban levő függvényekkel („*pearson*, *sperman* és a *kendalltau*”) mindhárom korrreláció kiszámítható. (Lásd a 16. ábrát.) Ebben az esetben, egy kételemű sormátrix lesz a visszatérési érték. Az első visszatérési érték a korrelációs együtthatót, a második pedig (a *pvalue*) a szignifikanciaértéket (megfigyelt szignifikanciaszintet) adja meg. Ha csak a korrelációs együthatót szeretnénk megjeleníteni, a függvény után szögletes zárójelben meg kell adnuk egy „0” értéket. (Lásd a 16. ábrát.) Ez azt jelenti, hogy a program a sormátrix első elemét írja ki, mivel – mint

azt már említettük – a Pythonban a mátrixok sorának és oszlopának számozása 0-tól kezdődik.

16. ábra. Korreláció számítása (scipy.stats)
(Calculation of correlation [*scipy.stats*])

```
scipy.stats.pearsonr(szam_sor_elso,szam_sor_masodik)
```

```
(0.3124404705204619, 0.45118277121148426)
```

```
scipy.stats.spearmanr(szam_sor_elso,szam_sor_masodik)
```

```
SpearmanrResult(correlation=0.27545404024004955, pvalue=0.5090540726550732)
```

```
scipy.stats.kendalltau(szam_sor_elso,szam_sor_masodik)
```

```
KendalltauResult(correlation=0.2545875386086578, pvalue=0.38281014365989596)
```

```
scipy.stats.pearsonr(szam_sor_elso,szam_sor_masodik)[0]
```

```
0.27545404024004955
```

17. ábra. Korreláció számítása (pandas)
(Calculation of correlation [*pandas*])

```
pandas_szam_sor.corr(pandas_szam_sor_nan_al)
```

```
0.9268557560322094
```

```
pandas_szam_sor_nan_al.corr(pandas_szam_sor)
```

```
0.9268557560322093
```

```
pandas_szam_sor.corr(pandas_szam_sor_nan_al,method='spearman')
```

```
0.8999999999999998
```

```
pandas_szam_sor.corr(pandas_szam_sor_nan_al,method='kendall')
```

```
0.7999999999999999
```

A *pandas* modulban a *corr*() függvényt használhatjuk a korreláció kiszámítására a már szokásos módon (az egyik változó után ponttal elválasztva hívjuk meg a *corr*() függvényt, a második vátozót átadva paraméterként). Mivel a korreláció

kiszámításánál a paraméterek sorrendje felcserélhető, így ebben az esetben is mindegy, hogy melyik változót írjuk előre és melyiket adjuk át parméterként. Az eredmény nem változik. (Lásd a 17. ábrát.) Ha a Spearman- vagy Kendall-féle korrelációt szeretnénk számítani, akkor ezt a *method* paraméterben kell megadnunk. (Lásd a 17. ábrát.)

2.15. Nemlineáris korrelácó (távolságkorreláció)

A távolságkorreláció a két adatsor közötti lineáris és nemlineáris kapcsolatot is tudja mérni. Ez ellentétben áll a Pearson-féle korrelációval, amely csak két adatsor változói közötti lineáris kapcsolatot képes detektálni. Mivel ez egy viszonylag új fogalom – amit *Székely Gábor* (*Székely–Rizzo–Bakirov* [2007]) magyar matematikus definiált –, ezért ennek kiszámítására a Pythonban a *dcor* modult fejlesztették ki.

18. ábra. Távolságkorreláció számítása
(Calculation of distance correlation)

```
import dcor
```

```
dcor.distance_correlation(pandas_szam_sor,pandas_szam_sor_nan_al)
```

```
0.9637326383899596
```

```
dcor.distance_stats(pandas_szam_sor,pandas_szam_sor_nan_al)
```

```
Stats(covariance_xy=3.924079509897832, correlation_xy=0.9637326383899596, variance_x=3.6097645352571135, variance_y=4.59286
4030210344)
```

A *dcor*-t[3] ugyanúgy tudjuk a programunkba importálni, mint bármely más eddig használt modult. Két függvényét említjük itt meg (részletes leírása megtalálható a 3. lábjegyzetben megadott linken). A *distance_correlation*() függvénynek meg kell adnunk a két adatsorunkat – amelyek *nan*-t nem tartamazhatnak –, és a távolságkorrelációt kapjuk eredményül. (Lásd a 18. ábrát.) A *distance_stats*() függvény egy leírást ad vissza, amely a fontosabb adatokat (távolságkovariancia, távolságkorreláció, és a két adatsor varianciája) tartalmazza. (Lásd a 18. ábrát.)

[3] A *dcor* modul leírását lásd https://readthedocs.org/projects/dcor/downloads/pdf/latest/

2.16. Lineáris regresszió

Ha két adatsor lineáris kapcsolatban van egymással, akkor az egyik segítségével megbecsülhetjük a másik értékét. Szükségünk van a függő és független változó kiválasztására, de ez nem biztos, hogy oksági kapcsolatot is jelent (*Pödör* [2016]). A lineáris regresszió egy paraméteres regressziós modell (*Cserháti* [2004]), amely feltételezi a függő és független változó közötti (paramétereiben) lineáris kapcsolatot Ez az ábrázolásban azt jelenti, hogy a lineáris regresszió becslése során a mintavételi adatok pontfelhőjére igyekszünk egyenest illeszteni.

19. ábra. Lineáris reggreszió számítása
(Calculation of linear regression)

```
scipy.stats.linregress(pandas_szam_sor,pandas_szam_sor_nan_al)
```

```
LinregressResult(slope=1.1902985074626864, intercept=-2.4179104477611935, rvalue=0.9268557560322093, pvalue=0.0234844782514
67857, stderr=0.2783542074859445)
```

A *scipy.stat* modul *linregress*() függvénye visszaadja az illesztett egyenes paramétereit: a meredekséget és a tengelymetszetet, a korrelációt (*rvalue*), a megfigyelt szignifikanciaszintet (*pvalue*), valamint a becsült hibát (*stderr*). (Lásd a 19. ábrát.) Most már minden adatunk megvan, hogy megjelenítsük a lieáris regresszió egyenesét. Nézzük lépésenként a megjelenítést. Elsőként szükségünk van a *matplotlib.pyplot* modulra, amit röviden csak „plt”-nek fogunk hívni („import matplotlib.pyplot as plt” – az as után bármilyen rövidítést megadhatunk).

20. ábra. Lineáris regreszió számítása – értékadás
(Calculation of linear regression – the assignment)

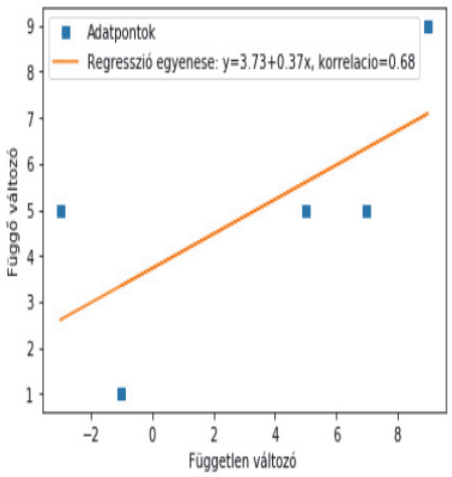
```
meredekseg, tengelymetszet, korrelacio, p, stderr = scipy.stats.linregress(pandas_szam_sor,pandas_szam_sor_nan_al)
```

Második lépésben kiszámítjuk a szükséges adatokat (meredekség, tengelymetszet, korreláció). Ezeket változókban tároljuk úgy, hogy az egyenlőség bal oldalán felsoroljuk a változóink nevét (*meredekség*, *tengelymetszet*, *korreláció*, *p*, *stderr*), így, ha lefuttatjuk a programsort, a változók felveszik a kívánt értékeket. (Lásd a 20. ábrát.)

Utolsó lépésként kirajzoltatjuk az eredményt. A négyzetek az adatpontokat, míg a vonal a regressziós egyenest jelöli. A jelmagyarázatban az egyenlet és a korrelációs együttható szerepel (line változó összeállítása: szöveg + zárójelben a változó értéke. (Lásd a 21. ábra harmadik sorát.)

21. ábra. Lineáris regresszió ábrázolása
(Representation of linear regression)

```
# Adatok megjenítése
import matplotlib.pyplot as plt
line = f'Regresszió egyenese: y={tengelymetszet:.2f}+{meredekseg:.2f}x, korrelacio={korrelacio:.2f}'
fig, ax = plt.subplots()
ax.plot(pandas_szam_sor_1, pandas_szam_sor_2, linewidth=0, marker='s', label='Adatpontok')
ax.plot(pandas_szam_sor_1, tengelymetszet + meredekseg * pandas_szam_sor_1, label=line)
ax.set_xlabel('Független változó ')
ax.set_ylabel('Függő változó')
ax.legend(facecolor='white')
plt.show()
```



Ennek a cikknek a keretein belül sajnos nem tudunk elég részletességgel kitérni a Python adatábrázolási lehetőségeire. Arra viszont szeretnénk felhívni a figyelmet, hogy ez párosros programmal könnyen megoldható, amire a következő alfejezetben mutatunk egy példát.

2.17. A korrelációs mátrixok hőtérképe

A három sorból és oszlopból álló mátrix áttekintése egyszerű feladat. Egy vizuális hőtérkép, ahol minden mező színe megfelel a mátrix egy értékének, már elég látványos. Bemutatjuk, hogy egy rövid programmal, hogyan tudjuk ezt megjeleníteni.

Mindenekelőtt szükségünk van egy korrelációs mátrixra.

22. ábra. Korrelációs mátrix
(The correlation matrix)

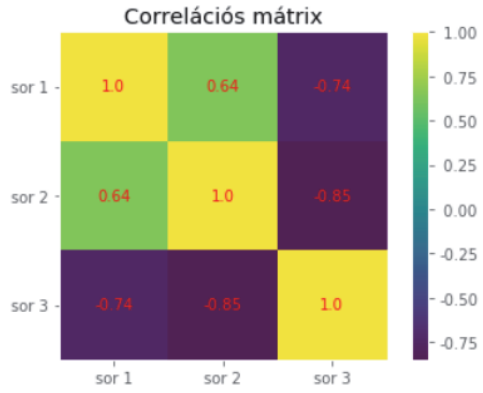
```
matrix=numpy.array([[5, 6, 7, 10, 14, 15, 16, 20, 18, 19],
                    [2, 1, 4, 5,4, 12, 18, 25, 96, 48],
                    [5, 3, 2, 2, 0, -2, -8, 0, -15, -16]])
```

```
corr_matrix=numpy.corrcoef(matrix).round(decimals=2)
```

Első lépésben megadunk három tetszőleges adatsort, majd kiszámítjuk ennek a korrelációs mátrixát a számunkra már ismert *corrcoef*() függvénnyel. Mindezt annyival kiegészítjük, hogy az eredményt kéttizedes pontosságra kerekítjük a *round*() függvénnyel. (Lásd a 22. ábrát.)

23. ábra. Korrelációs mátrix hőtérképe
(Heat map of the correlation matrix)

```
fig, ax = plt.subplots()
im = ax.imshow(corr_matrix)
ax.set_title('Correlációs mátrix')
ax.xaxis.set(ticks=(0, 1, 2),ticklabels=('sor 1', 'sor 2', 'sor 3'))
ax.yaxis.set(ticks=(0, 1, 2),ticklabels=('sor 1', 'sor 2', 'sor 3'))
for i in [0,1,2]:
    for j in [0,1,2]:
        ax.text(j, i, corr_matrix[i, j], ha='center', va='center', color='red')
cbar = ax.figure.colorbar(im, ax=ax, format='% .2f')
ax.grid(False)
plt.show()
```



A színek segítenek, hogy könnyebben átlássuk az eredményt. Ebben a példában a sárga szín az 1-es számot, a zöld a 0,64-ot, a kék pedig a negatív számokat jelöli. (Lásd a 23. ábrát.)

2.18. Klaszterek létrehozása

A klaszterek létrehozása egy olyan folyamat, amely során az adathalmaz elemeit csoportokba rendezzük úgy, hogy az egy csoportban levő elemek egymáshoz viszonyított hasonlósága nagy, a különböző csoportokban levőké pedig kicsi. A létrehozott csoportokat klaszthereknek nevezzük.

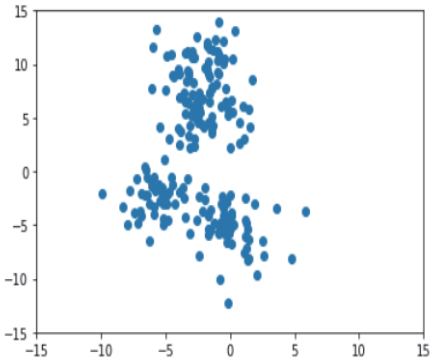
K-közép klaszterelemzés

A klaszterelemzési módszerek közül a K-közép (K-means) algoritmus minden egyes elemet ahhoz a klaszterhez sorol, amelyiknek a középpontja a legközelebb esik az adott elemhez (*Arthur–Vassilvitskii* [2007]). Általában az adatok többdimenziósok, és számos adattáblában találhatók egy adatbázison belül. Az általam bemutatott tesztadatok kétdimenziósak. Ez inkább illusztratív célra megfelelő, vizuálisan jól megjeleníthető, de nem tipikus. Első lépésként hozzunk létre tesztadatokat a Pythonban.

A *sklearn.datasets* modul *make_blobs*() függvénye segít nekünk a tesztadatok elkészítésében. Első lépés a függvény importálása. Ezt a modult azzal a céllal hozták létre, hogy megkönnyítse a tesztadatok létrehozását. Ezt ki is használjuk, mégpedig úgy, hogy egy speciális adattípust, *blobot* generálunk. A *blobok* olyan objektumok, amelyek nagy mennyiségű adat tárolására képesek. A *make_blobs*() függvénnyel egy blobot hozunk létre, amely 200 (*n_samples*) elemből álló tesztadatot tartalmaz. A függvénynek megadtuk, hogy kétdimenziós pontkoordinátákat adjon vissza (*n_features*), amelyek 4 csoportra legyenek elosztva (*centers*), egy csoporton belül 1,8 legyen a szórás (*cluster_std*). Lehetőség van még beállítani, hogy mindig ugyanazokat az adatokat generálja a függvény, ezt a *random_state* = 50 paraméter biztosítja. A függvény visszatérési értékei a pontok koordinátái, amelyeket ki is rajzoltatunk. (Lásd a 24. ábrát.)

24. ábra. Tesztadatok létrehozása
(Creating test data)

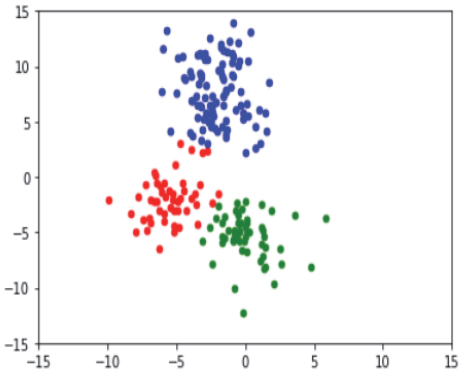
```
from sklearn.datasets import make_blobs
data = make_blobs(n_samples=200, n_features=2, centers=4, cluster_std=1.8, random_state=50)
points = data[0]
plt.scatter(data[0][:,0], data[0][:,1])
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.show()
```



25. ábra. K-közép – Klaszterek létrehozása
(*K*-mean – Creating clusters)

```
from sklearn.cluster import KMeans
y_km = KMeans(n_clusters=3).fit_predict(points)
```

```
plt.scatter(points[y_km ==0,0], points[y_km == 0,1], s=20, c='red')
plt.scatter(points[y_km ==1,0], points[y_km == 1,1], s=20, c='blue')
plt.scatter(points[y_km ==2,0], points[y_km == 2,1], s=20, c='green')
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.show()
```



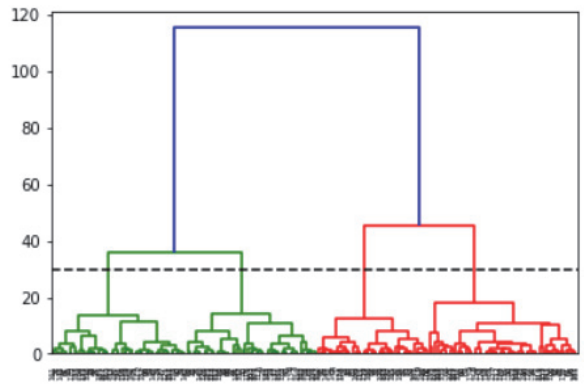
A klaszterek létrhozásahoz a *sklearn.cluster* modul *Kmeans*() függvényét használjuk (importáljuk a programunkba). Második lépésben *n_cluster* = 3 paraméterrel megadjuk a létrehozandó klaszterek számát, és a *fit_perdict*() függvénynek átadjuk a pontok koordinátáit, amely kiszámolja, hogy az egyes pontok melyik klaszterbe kerüljenek. A következő programrészlet különböző színekkel kirajzolja a klaszterek elemeit. (Lásd a 25. ábrát.)

Hierarchikus klaszterelemzés

A hierarchikus klasztert lehet felosztó (divisive) és összevonó (agglomerative) módszerrel készíteni. A felosztó eljárás az egész adattömböt egyetlen klaszternek tekinti, és egyre kisebb klaszterekre osztja, míg a végén minden elem külön klasztert képez (*Contreras–Murtagh* [2015]). Az összevonó eljárás minden egyes elemet külön klaszternek tekint, és összekapcsolja őket egyre nagyobb klaszterekbe, míg a végén egyetlen, az összes elemet tartalmazó klasztert nem kapunk. Az eredményt általában egy fa (*dendrogram*) formájában szokták ábrázolni (*Podani–Schmera* [2006]), ahol az x tengely tartalmazza a minták számát, az y tengely pedig a minták közötti távolságot.

26. ábra. Klaszter létrehozása dendrogrammal – Felosztó módszer
(Creating a cluster with dendrogram – Divisive method)

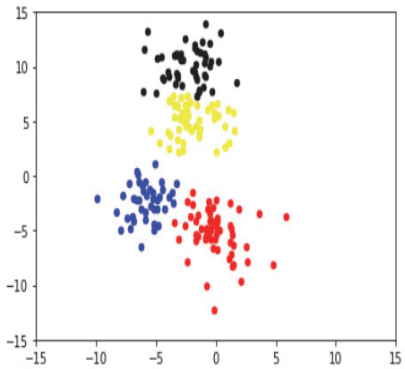
```
# import
import scipy.cluster.hierarchy as sch
# dendrogram készités
plt.axhline(y=30, color='black', linestyle='--')
dendrogram = sch.dendrogram(sch.linkage(points, method='ward'))
```



A *scipy.cluster.hierarchy* modul (amelyet első lépesben importálni kell) *dendrogram*() függvénye állíja elő a kívánt ábrát, miután a *linkage*() függvény elvégzi a hierarchikus csoportosítást. (Lásd a 26. ábrát.) Ha egy szinten elvágjuk a fát, akkor azon az adott ponton tudjuk értelmezni a klaszterek kialakítását. Ebben az esetben mi a 30-at választottuk a minták közötti távolság maximális értékének, amelyet a szaggatott fekete vonal jelez. (Lásd a 26. ábrát.)

27. ábra. Összevonó klaszterkészítés
(Creating a cluster [agglomerative method])

```
#  cluster készítés
hc = AgglomerativeClustering(distance_threshold=30 , n_clusters=None,affinity = 'euclidean', linkage = 'ward')
# megjelenítés
y_hc = hc.fit_predict(points)
plt.scatter(points[y_hc ==0,0], points[y_hc == 0,1], s=20, c='red')
plt.scatter(points[y_hc==1,0], points[y_hc == 1,1], s=20, c='black')
plt.scatter(points[y_hc ==2,0], points[y_hc == 2,1], s=20, c='blue')
plt.scatter(points[y_hc ==3,0], points[y_hc == 3,1], s=20, c='yellow')
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.show()
```



A klaszterek létrehozásához a *sklearn.cluster* modul *AgglomerativeClustering*() függvényét használtuk. Paraméterként a *distance_threshold* = 30 küszöbértéket állítottuk be. Ezen érték felett (maximális távolság) a klaszterek nem egyesülhetnek. Az *affinity* = '*euclidean*' paraméter értéknél pedig beállítottuk, hogy az euklideszi távolságot használja erre a célra. (Lásd a 27. ábrát.)

Klaszteranalízis jósági mutatói – sziluett együttható

A sziluett együttható[4] azt jelzi, hogy egy pont mennyiben hasonlít a saját klaszteréhez, összehasonlítva más klaszterekkel.

28. ábra. Sziluett együttható kiszámítása
(Calculation of the silhouette coefficient)

```
# KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
kmeans = KMeans(n_clusters=4).fit(points)
predictions = kmeans.predict(points)
silhouette_avg = silhouette_score(points, predictions)
print ("Sziluett együttható átlaga (KMeans):" ,silhouette_avg)
```

```
Sziluett együttható átlaga (KMeans): 0.49431551874730656
```

```
#  AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters=4, affinity = 'euclidean', linkage = 'ward')
predictions = hc.fit_predict(points)
y_hc = hc.fit(points)
silhouette_avg = silhouette_score(points, predictions)
print ("Sziluett együttható átlaga (AgglomerativeClusterings):" ,silhouette_avg)
```

```
Sziluett együttható átlaga (AgglomerativeClusterings): 0.48757123685543646
```

A *silhouette_score*() függvény megadja a sziluett együttható átlagos értékét. Ezzel az értékkel megítéljük, melyik klaszterképzés adott jobb eredményt. Két nagyon közeli értéket kaptunk: 0,49-et és 0,48-at. (Lásd a 28. ábrát.) Általánosságban az összevonó klaszterképzés előnye a K-közép klaszterképzéssel szemben az, hogy pontosabb eredményeket ad, hátránya viszont, hogy több számolást igényel.

3. Összefoglalás

A statisztika, mint korunk egyik nagy léptékben fejlődő tudományága, számos kihívás előtt áll ma, és fog állni a közeljövőben is. Ma már nem csak egy szűk szakemberréteg böngészi az adatokat. A hasznos információra mint értékre tekintünk az élet minden területén. Bármely rangos közéleti újságtól, ismeretterjesztő publikáció-

[4] A klaszteranalízis jósági mutatóiról jó áttekintést ad *Szüle* [2019].

tól elvárható, hogy eredményeit adatokkal támassza alá. Nagyon fontos, hogy ezt tudományos alapossággal, de közérthető és látványos módon tegye. Ebben segíthetnek bennünket a világcégek által kínált (SAS, SPSS) statisztikai programcsomagok. Viszont korlátoznak is egyben, mert sokszor fizetnünk kell értük (*USC* [2010]), nem lehet őket olyan könnyen személyre szabni, vagy esetleg más platformon is effektíven használni. A cikk nem titkolt célja, hogy a számok nyelvén jól beszélő szakemberek érdeklődését felkeltse a Python programnyelv iránt, amelyre olyan lehetőségként kell tekinteni, hogy valóban sok befektetett energiát igényel, de szinte korlátlan és ingyenes, valamint lehetőséget nyújt a szakmán belüli fejlődésre (*Shaw* [2013], *Unpingco* [2016]), azaz az adatok tudományos elemzésére és látványos bemutatására (*Solem* [2012]).

Irodalom

ARTHUR, D. – VASSILVITSKII, S. [2007]: *K-means++: The Advantages of Careful Seeding.* Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms. 7–9 January. New Orleans. https://theory.stanford.edu/~sergei/papers/kMeansPP-soda.pdf

AYER, V. M. – MIGUEZ, S. – TOBY, B. H. [2014]: Why scientists should learn to program in Python. *Powder Diffraction*. Vol. 29. Supplement S2. pp. S48–S64. https://doi.org/10.1017/S0885715614000931

BRISCOE, T. [2011]: *Introduction to Linguistics for Natural Language Processing.* Computer Laboratory University of Cambridge. Cambridge. https://www.cl.cam.ac.uk/teaching/1314/L100/introling.pdf

CONTRERAS, P. – MURTAGH, F. [2015]: Hierarchical clustering. In: *Hennig, C. – Meila, M. – Murtagh, F. – Rocci, R.* (eds.): *Handbook of Cluster Analysis*. Chapman & Hall/CRC. Bosa Roca. pp. 103–120. https://doi.org/10.1201/b19706

CSERHÁTI Z. [2004]: Az outlierek meghatározása és kezelése gazdaságstatisztikai felvételekben. *Statisztikai Szemle*. 82. évf. 8. sz. 728–746. old. http://www.ksh.hu/statszemle_archive/all/2004/2004_08/2004_08_728.pdf

DOWNEY, A. B. – ELKNER, J. – MEYERS, C. [2013]: *How to Think Like a Computer Scientist: Learning with Python.* Manual. https://www.amazon.com/How-Think-Like-Computer-Scientist/dp/0971677506

HAJDU O. [2018]: Többváltozós statisztikai R Open alkalmazások. *Statisztikai Szemle*. 96. évf. 10. sz. 1021–1047. old. https://doi.org/10.20311/stat2018.10.hu1021

JANSSENS, J. [2014]: *Data Science at the Command Line: Facing the Future with Time-Tested Tools*. First Edition. O'Reilly Media. Sebastopol.

LUTZ, M. [1999]: Using Python. *USENIX & SAGE*. Vol. 24. pp. 36–42. https://www.mendeley.com/catalogue/8fdf1977-7c33-3c3a-8ec0-4dd777dcb064/

MENCZER, F. – FORTUNATO, S. – DAVIS, C. A. [2020]: *Appendix A – Python Tutorial. A First Course in Network Science*. Cambridge University Press. Cambridge. pp. 221–237. https://doi.org/10.1017/9781108653947.010

MOLL, R. N. – ARBIB, M. A. – KFOURY, A. J. [1988]: *An Introduction to Formal Language Theory.* Springer Verlag. Berlin. https://doi.org/10.1007/978-1-4613-9595-9

MÜLLNER, D. [2013]: Fastcluster: fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software*. Vol. 53. Issue 9. pp. 1–18. https://doi.org/10.18637/jss.v053.i09

NETWORK SECURITY [2015]: *Doing math with Python*. Issue 10. p. 4. https://doi.org/10.1016/s1353-4858(15)30088-x

OLIPHANT, T. E. [2007]: Python for scientific computing Python overview. *Computing in Science and Engineering*. Vol. 9. No. 3. pp. 10–20. https://www.researchgate.net/publication/3422935_ Python_for_Scientific_Computing

PAYNE, J. [2010]: *Beginning Python: Using Python 2.6 and Python 3.1.* http://index-of.es/Python/Beginning%20Python%20Using%20Python%202.6%20and%20Python%203.1.p dfPilgrim, M. (2009). Dive into python 3. In *Dive Into Python 3*. https://doi.org/10.1007/978-1-4302-2416-7

PODANI, J. – SCHMERA, D. [2006]: On dendrogram-based measures of functional diversity. *Oikos*. Vol. 115. Issue 1. pp. 179–185. https://doi.org/10.1111/j.2006.0030-1299.15048.x

PÖDÖR Z. [2016]: Többváltozós lineáris regresszió a gyakorlatban. *Dimenziók: Matematikai Közlemények*. 4. köt. 49–54. old. https://doi.org/10.20312/dim.2016.07

PYTHON FOUNDATION [2016]: *About Python.* Python.org. https://www.python.org/

PYTHON SOFTWARE FOUNDATION [2019]: *Python 3.7.4 documentation*. https://docs.python.org/3.7/

SOLEM, J. E. [2012]: *Programming Computer Vision with Python.* http://programmingcomputervision.com/downloads/ProgrammingComputerVision_CCdraft.pdf

SHAW, Z. [2013]: *Learn Python: The Hard Way.* (3rd Edition.) Addison–Wesley Educational Publishers Inc. Boston.

SHELL, S. [2014]: *An Introduction to Numpy and Scipy*. University of California. Santa Barbara.

SZÉKELY, G. J. – RIZZO, M. L. – BAKIROV, N. K. [2007]: Measuring and testing dependence by correlation of distances. *Annals of Statistics*. Vol. 35. No. 6. pp. 2769–2794. https://doi.org/10.1214/009053607000000505

SZEPTYCKI, P. [2004]: Percentiles. *Real Analysis Exchange*. Vol. 29. Issue 1. pp. 461–464. http://dx.doi.org/10.14321/realanalexch.29.1.0461

SZÜLE B. [2019]: Klaszterszám-meghatározási módszerek összehasonlítása. *Statisztikai Szemle.* 97. évf. 5. sz. 421–438. old. http://dx.doi.org/10.20311/stat2019.5.hu0421

UNPINGCO, J. [2016]: *Python for Probability, Statistics, and Machine Learning*. Springer. Berlin. https://doi.org/10.1007/978-3-319-30717-6

USC (UNIVERSITY OF SOUTHERN CALIFORNIA) [2010]: *Installing SPSS for Windows.* https://itservices.usc.edu/stats/spss/installwin/

VAN ROSSUM, G. – DRAKE, F. L. [2009]: *The Python Language Reference Manual*. Network Theory Ltd. Bristol.

VAN ROSSUM, G. [2016a]: *Extending and Embedding Python.* Samurai Media Limited. Thames Ditton.

VAN ROSSUM, G. [2016b]: *Python Setup and Usage.* https://www.academia.edu/38835231/Python_Setup_and_Usage_Release_3.7.3_Guido_van_Rossum_and_the_Python_development_team

VAN ROSSUM, G. [2017]: *Curses Programming with Python.* https://www.coursehero.com/file/31477266/howto-cursespdf/

WILSON, D. J. [2019]: The harmonic mean *p*-value for combining dependent tests. *Proceedings of the National Academy of Sciences of the United States of America.* Vol. 116. No. 4. pp. 1195–1200. https://doi.org/10.1073/pnas.1814092116