Python mogyoróhéjban Összeállította: Siki Zoltán

Tartalomjegyzék

Bevezetés

A Python egy széles körben elterjedt általános célú szkript nyelv. Lehetőséget biztosít objektum orientált programozásra is. Számos bővítő modullal rendelkezik, például GDAL, OGR, numpy, OpenCV, PCL, stb. Mivel a Python futtatókörnyezet számos operációs rendszeren (Linux, OS X, Android, Windows) rendelkezésre áll, egyszerűen készíthetünk platform független alkalmazásokat. Számos szoftverben futtathatunk Python szkripteket, pl. GIMP, QGIS, Apache, PostgreSQL, GRASS, stb. A Python értelmező a forráskódot byte kóddá fordítja ezért a Python programok gyorsabban futnak mint más szkript nyelven írt programok.

Manapság még széles körben használják a Python 2.x változatát (de a fejlesztését 2020i-ban befejezték). A 3.x változat 2008 decemberen indult útjára. A nagy verziószám változása miatt a kompatibilitás nincs meg a két verzió között. Most már mindenkinek 3.4 vagy későbbi verzió ajánlott. A példákat 3.8.2 verzióban teszteltem Ubuntu 20.04 rendszeren, de valószínűleg 2.7.x verzióban is működnek.

A Python telepítésére az operációs rendszer függvényében többféle telepítési mód áll a felhasználók rendelkezésére. A térinformatikát használók számára az OSGeo4W telepítővel telepített QGIS esetén a Python és a szükséges bővítő modulok is felkerülnek a gépünkre, melyeket az OSGeo4W Shell ablakban használhat.

A Pythont használhatjuk interaktívan is, az értelmezővel azonnal végrehajtatjuk az utasításainkat. Az operációs rendszer parancs ablakában indítsuk el a Python értelmezőt.

```
> python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
```

```
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Előfordulhat, hogy elegendő a **python** parancs beírása, de sok rendszeren a 2-es és 3-as verzió is megtalálható. **>>>** jel a Python prompt, ha ezt látjuk, akkor beírhatjuk a Python utasításainkat utána. A Python értelmezőből az **exit()** paranccsal vagy Linuxon a Ctrl/D, Windows-on a Ctrl/Z majd Enter billetyű kombinációval léphet ki.

Alapok

Először használjuk interaktívan a Python értelmezőt és ismerkedjünk meg az adattípusokkal, egyszerű kifejezésekkel. A **>>>** jeleket nem kell beírnunk, a Python a # karakter utáni szöveget a sor végéig megjegyzésnek értelmezi, ezeket nem fontos beírni.

```
>>> 5 * 4
20
>>> 3 ** 4                              # hatványozás
81
>>> (2 + 8) // 3                # egész osztás
3
>>> (2 + 8) / 3                 # lebegőpontos osztás
3.3333333333333335
>>> 56.12 // 12.34              # eész osztás
4.0
```

A kiszámított értékeket változókban megőrizhetjük, ilyenkor nem jelenik meg a művelet eredménye.

```
>>> a = 5 * 4
>>> print(a)                    # változó értékének kiiratása
>>> l = True                    # logikai érték True vagy False
>>> c = None                    # a None a definiálatlan értéket jelenti
>>> s = "Hello world"   # karakterlánc szimpla aposztróf is jó
```

Python-ban nincs automatikus típus konverzió a változókra, konstansokra, eltérően más szkript nyelvektől (pl. JavaScript). A változókat nem kell deklarálni, a változó típusát az első értékadás állítja be. A type függvénnyel le tudjuk kérdezni egy változó típusát.

```
>>> b = '16'
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> type(a)                             # Python 3-ban az alaptípusok is osztályok
<class 'int'>
>>> type(b)
<class 'str'>
>>> type(b) is str              # is operátor
True
>>> a + int(b)                  # típus konverzió
36
>>> a * b                               # karakterlánc többszörözése
'1616161616161616161616161616161616161616'
```

```
>>> s[0]                                # karakterlánc első eleme
'H'
>>> s[-1]                               # karakterlánc utolsó eleme
'd'
>>> s[0] = "h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s[1:3]                              # index tartomány
'el'
>>> s[:5]                               # első 5 karakter
'Hello'
>>> s[6:]                               # 6. karaktertől a végéig
'world'
>>> s[-5:]
'world'
>>> len(s)                              # karakterlánc hossza
11
>>> s[:5] + ' Dolly'    # karakterláncok összefűzése
'Hello Dolly'
```

A matematikai függvények (sin, cos, stb.) és néhány konstans (pl. pi) egy külön modulban található (math).

```
>>> sin(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> import math
>>> math.sin(1)                 # szög radiánokban
0.8414709848078965
>>> "{:6.4f}".format(math.sin(1)) # formatizálás
'0.8415'
```

Modulokat kétféleképpen importálhatunk. Az **import** *modul_név* parancs használata esetén a modul elemeire úgy hivatkozhatunk, hogy elé írjuk a modul nevét. A másik módszer esetén **from** *modul_név* **import** *elem* nem kell a modul nevét használnunk a modul elemére hivatkozásnál, viszont ilyenkor könnyen név ütközés fordulhat elő.

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Ez utóbbi változat esetén vesszővel elválasztva a modul több felhasználandó elemét is megadhatjuk vagy * karakterrel az összes elemet importálhatjuk (pl. from math import *).

Adatszerkezetek

Lista

Az egyszerű változók mellett listákat is használhatunk. A lista egy rendezett adathalmaz, elemeire 0-tól kezdődő index-szel hivatkozhatunk. A lista elemei tetszőleges, különböző típusú értékek lehetnek, akár listák is.

```
>>> l1 = []                              # üres lista létrehozása
>>> l1 = list()                 # üres lista létrehozása
>>> l2 = ['alma', 5, 4] # eltérő adattípusok egy listában
>>> l2[0]                               # indexelés egy elem eléréséhez
'alma'
>>> l2[1:]                              # index tartomány mint szövegláncnál
[5, 4]
>>> len(l2)
3
>>> l2[0] = 3                   # listaelemek módosíthatók
>>> l2[3]                               # nem létező listaelem
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> l2[3] = 5                   # új elem értékadással nem hozható létre
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> l2.append(5)                # lista bővítése
>>> print(l2)
[3, 5, 4, 5]
>>> l2[0:3] = [1]               # lista részének cseréje
>>> print(l2)
[1, 5]
>>> del l2[0]                   # listaelem törlése
>>> l2
[5]
>>> l3 = [[1, 2], [6, 4]]       # listák listája
>>> l3[0][1]
2
>>> l4 = l3 + [3, 7]    # listák összege
>>> l4
[[1, 2], [6, 4], 3, 7]
>>> l4 = l3 + [[3, 7]]
>>> print(l4)
[[1, 2], [6, 4], [3, 7]]
>>> help(list)                  # lista objektum leírása
```

Listák összefűzését a "+" művelettel végezhetjük el, a "*" operátor a karakterláncokhoz hasonlóan ismétlést jelent. Az **in** művelettel megvizsgálhatjuk, hogy a lista tartalmaz-e egy elemet. A listákkal kapcsolatban több függvényt használhatunk (például sorted, reversed, min, max) illetve további metódusok tartoznak hozzá (pl. pop, sort, reverse).

A tuple egy nem módosítható lista, a listával azonos módon használható, kivéve a módosító utasításokat (a karakterlánchoz hasonlóan). A tuple kifejezésre nincs elfogadott magyar fordítás, néhányan vektornak mondják, de az a listára is igaz lenne, adatbáziskezelésben rekord értelemben használják, a matematikában használják rá a szám n-es kifejezést, de a Pythonban nem csak számokat tárolhat, ... Én maradtam a tuple-nél (kiejtése: tapló :)

```
>>> t = (65, 6.34)              # tuple létrehozása zárójel elhagyható
>>> t[0]
65
>>> u =
>>> t.append(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'tuple' object has no attribute 'append'
>>> u = tuple()                 # üres tuple létrehozása u = () is lehetne
>>> v = 2,                              # egy elemű tuple létrehozása
```

A listákkal, tuple-kkel kapcsolatban gyakran nagyon tömör kódot írhatunk az elemekkel végzendő műveletre. Ezt angolul list comprehension-nak, magyarul lista értelmezésnek vagy feldolgozásnak mondhatjuk.

```
>>> l = [2, 6, -3, 4]
>>> l2 = [x**2 for x in l]             # minden listaelem négyzet
>>> l2
[4, 36, 9, 16]
>>> [x**0.5 for x in l if x > 0]        # pozitív listaelemek gyöke
[1.4142135623730951, 2.449489742783178, 2.0]
>>> from math import sin
>>> list(map(sin, l))                   # mint [sin(p) for p in l]
[0.9092974268256817, -0.27941549819892586, -0.1411200080598672, -0.7568024953079282]
>> [a for a in l if a % 2]              # páratlan számok a listából
[-3]
```

Halmaz

A halmaz egy rendezetlen, ismétlődő elemeket nem tartalmazó adatszerkezet. Halmazokat szövegláncokból, listákból, tuple-okból hozhatunk létre egyszerűen. A halmaz módosítható (mutable) adatszerkezet.

```
>>> a = "abcabdseacbfds"
>>> s = set(a)                                      # halmaz létrehozás
>>> s
{'e', 'a', 'f', 'd', 's', 'c', 'b'}
>>> t = set((1, 3, 2, 4, 3, 2, 5, 1))
>>> t
{1, 2, 3, 4, 5}
>>> e = set()                                       # üres halmaz létrehozás
>>> u = set(['alma', 'szilva', 'barack', 'alma'])
>>> u
{'alma', 'barack', 'szilva'}
```

A halmazokkal műveleteket is végezhetünk.

```
>>> a = set('abcdefgh')
>>> b = set('fghijklm')
>>> a -b                                        # halmazok különbsége
{'e', 'a', 'd', 'c', 'b'}
>>> a | b                                       # halmazok uniója
{'e', 'a', 'g', 'l', 'f', 'i', 'd', 'c', 'b', 'j', 'k', 'm', 'h'}
>>> a & b                                       # halmazok metszete
{'g', 'f', 'h'}
>>> a ^ b                                       # halamzok szimmetrikus különbsége
{'k', 'e', 'a', 'l', 'd', 'c', 'j', 'b', 'i', 'm'}
```

Szótár

A szótár egy rendezetlen adathalmaz ahol az elemekhez egy kulcs járul, mellyel megcímezhetjük (asszociatív tömb, hash tábla, struktúra más programnyelvekben). A szótár elemei lehetnek listák és további szótárak is. A szótárak módosíthatók, bővíthetők (mutable

```
>>> szotar = {}                          # üres szótár létrehozása
>>> szotar['elso'] = 4          # egyszerűen bővíthető
>>> szotar[5] = 123                      # szám is lehet az index/kulcs
>>> szotar['elso'] = 'alma'     # módosítható egy létező elem
>>> print(szotar)
{'elso': 'alma', 5: 123}
>>> 'elso' in szotar            # létezik a kulcs/index?
True
>>> sz = {'b': 1, 12: 4, 'lista': [1, 5, 8]}    # inicializálás
>>> tomb = {(1,1): 2, (1,2): 4, (2,1): -1, (2,2): 6}    # tuple az index
>>> tomb[1,1]
2
```

Python programok

A Python interaktív használata az ad-hoc, egyszerű feladatok megoldásához, egy-egy utasítás kipróbálásához megfelelő. A produktív felhasználás esetén a Python utasításainkat fájlokban (modulokban) helyezzük el és azokat hajtatjuk végre a Python értelmezővel. A megoldandó feladatot kisebb részekre bonthatjuk függvények illetve osztályok segítségével, a megoldás során elágazó és ciklus utasításokat használunk. A programok készítése előtt ki kell emelni a Python egy sajátosságát, a Python programokban a kód blokkokat a sor elején található szóközökkel jelöljük. Más nyelvekben a blokk elejét és végét jelölik meg, például '{' és '}' zárójelekkel. A nyelv ezen tulajdonsága kikényszeríti a könnyen olvasható kód készítését, másik oldalon nagyobb figyelmet igényel a sorok írásánál. A szóköz és tabulátor karaktereket nem lehet keverni, a négy szóközös tagolás ajánlott. Összetettebb programok esetén valamilyen fejlesztő környezetet érdemes használni, ahol a hibákat hatékonyabban lokalizálhatjuk. Ezek közül az egyszerűen kezelhető IDLE3, melyet a Pythonnal együtt fejlesztenek és teljes egészében Python nyelven íródott. Számos további nyílt forráskódú és fizetős integrált fejlesztői környezet (IDE) létezik (PyCharm, Eric, stb.). A következőkben bemutatott példák beviteléhez egy egyszerű szövegszerkesztő (Notepad, gedit) is megfelelő, de célszerű inkább olyat választani, mely a forráskód különböző elemeit kiszínezi (ilyen például a Notepad++, vim, Idle). Így könnyebben olvasható a kód és észrevehetjük a gépelési hibákat.

Készítsük el első programunkat, mely az első 100 egész szám összegét számítja ki (számlálással vezérelt ciklus). Nyissunk egy új fájlt a kedvenc szövegszerkesztőnkben (pl. vim, nedit (Linux), Notepad++, Jegyzettömb (Windows)) vagy az idle környezetben és vigyük be a következő kódot (sum.py):

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
""" első Python programom
        1 - 100 közötti egész számok összege
"""
s = 0
for i in range(1,101):              # a kettőspont jelzi a blokk elejét
        s += i
print(s)                            # ez a ciklus vége
```

Az első két sorban két speciális megjegyzés található. Az első sor a Linux buroknak szól, ha futtathatóvá tesszük a fájlt (*chmod +x sum.py*), akkor elegendő a fájl nevét megadni a parancssorban (pl. ha az aktuális könyvtárban található a programunk *./sum.py*, lásd a következőkben). A második sor a fájlban használt karakterkódolás adja meg, meg kell adnunk, ha ékezetes betűket használunk a kommentekben.

Három dupla vagy szimpla aposztóffal több sorra kiterjedő megjegyzést kezdhetünk, melyet három aposztróffal zárhatunk le. Az ilyen megjegyzésekből automatizáltan generálhatunk fejlesztői dokumentációt illetve a help Python függvény a modul vagy a függvény elején elhelyezett ilyen megjegyzés tartalmát írja vissza. A programot sum.py névvel mentsük ki a háttértárolóra. A programunkat (modulunkat) futtathatjuk a Python értelmezőből (a Pythont abból a könyvtárból indítsuk, ahová a sum.py fájlt mentettük):

```
> python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sum
5050
>>> help(sum)
Help on module sum:

NAME
        sum

DESCRIPTION
        első Python programom
        1 - 100 közötti egész számok összege
...
```

A parancssorból közvetlenül is futtathatjuk a programokat, általában ez a szokásos.

```
> python3 sum.py
5050
```

Ez utóbbi változat lehetővé teszi, hogy más szkriptekbe beépítsük a programunkat. Vagy futtathatóvá téve a program fájlt, Linuxon így is működhet.

```
> chmod +x sum.py               # ez csak egyszer kell
> ./sum.py
5050
```

Módosítsuk a programunkat, hogy a parancssorban megadhassuk az összegzés felső határát. Ehhez a *sys* modul *argv* listáját használhatjuk. Az alábbi kódot a sum1.py fájlba mentsük el.

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
""" első Python programom
        1 - argv[1] közötti egész számok összege
"""
from sys import argv
if len(argv) < 2:
        print("usage: {} number".format(argv[0]))
        exit(1)
s = 0
for i in range(1,int(argv[1])+1):  # a kettőspont jelzi a blokk elejét
        s += i
print(s)                           # ez a ciklus vége
```

```
> python3 sum1.py 50
1275
```

Ez az első program nem mondható Pythonikusnak (Pythonic). Ez alatt azt értjük, hogy nem a Python nyelv kínálta optimális megoldást választottuk (mert a *for* ciklust akartuk bemutatni). A sum2.py fájlba már a Pythonikus megoldást írjuk:

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
""" első Python programom
        1 - argv[1] közötti egész számok összege
"""
from sys import argv
if len(argv) < 2:
        print("usage: {} number".format(argv[0]))
        exit(1)
print(sum(range(1, int(argv[1])+1)))
```

Ha lehetséges kerüljük el a ciklusokat vagy helyettesítsük lista feldolgozással. Így hatékonyabb lesz a kódunk.

> ***Note***
>
> Mi történne, ha nem egész számot adnánk meg a parancssorban? Próbálja ki!

További algoritmus szerkezeteket is biztosít a Python. A programozás során a *for* ciklus mellet *while* ciklust is használhatunk (feltétellel vezérelt), a ciklusok végrehajtását a *continue* és *break* utasításokkal befolyásolhatjuk. Egy speciális utasítás a *pass*, mely nem csinál semmit, helykitöltésre használható. A döntéseknél az *if elif else* utasítást használhatjuk. Lásd a további példákban.

Szövegfájlok kezelését is egyszerűen megoldhatjuk Pythonban. Mintaként készítsünk egy programot, mely a parancssorban megadott szövegfájlból kiválasztja szintén a parancssorban megadott szót tartalmazó sorokat (grep.py).

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
""" adott szöveget tartalmazó sorok kiírása egy fájlból """
from os import path                                          # file létezik-hez
from sys import argv                                    # paraméterekhez

if len(argv) < 3:
        print("Usage: {} <fájl> <keresett_szöveg>".format(argv[0]))
        exit(1)
if not path.exists(argv[1]):
        print("{} fájlt nem találom".argv[1])
        exit(2)

with open(argv[1]) as f:
        for line in f:                                       # soronként olvasás
                if line.find(argv[2]) >= 0:
                        print(line.strip('\n\r'))
```

A fenti kódban a standard *os* Python modult használjuk a fájl létezésének ellenőrzésére. Az *os* modul számos további lehetőséget tartalmaz. A *with* blokk egy jó példa a Pythonikus kódra. A legtöbb nyelven először megnyitjuk a input fájlt, feldolgozzuk a tartalmát és végül lezárjuk. A *with* a blokk lezárásával együtt a fájlt is lezárja.

A *find* és *strip* függvények a szöveglánc objektumhoz tartoznak. Az összes lehetőséget a *help(str)* paranccsal nézheti meg. A *line.strip('\n\r')* leveszi a kiírás előtt a sorvég karaktereket, melyeket a beolvasott sor tartalmaz.

> ***Note***
>
> A különböző operációs rendszerek eltérő sorvég jeleket használnak. Linux: \n, Windows: rn, OSX: r. a fenti kód minden operációs rendszeren helyesen működik.

Saját függvényeket definiálhatunk a *def* kulcsszóval. A függvényeknek nem kötelező értéket visszaadniuk. A visszaadott érték összetett adatszerkezet is lehet, például lista vagy szótár is. Néhány egyszerű függvény (a func.py fájlba írjuk be a következőket):

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
from math import (sqrt, prod)

def celsius(fahrenheit=0):
        "celsius to fahrenheit conversion"
        return (fahrenheit - 32) * 5.0 / 9.0

def root(a, b, c):
        "roots of a quadratic equation"
        d = b**2 - 4 * a * c
        if d == 0:
                return -b / 2.0 / a
        elif d > 0:
                return ((-b + sqrt(d)) / 2.0 / a, (-b - sqrt(d)) / 2.0 / a)
        else:
                pass    # complex roots solved later
        return None

def f(n):
        "factorial calculation with while"
        f = 1
        while n > 0:
                f *= n
                n -= 1
        return f

def fact(n):
        "factorial calculation with for"
        f = 1
        for i in range(1, n+1):
                f *= i
        return f

def factor(n):
        "recursive faktorial calculation"
```

```
        if n <= 1:
                return 1
        return n * factorial(n-1)

def factorial(n):
        """ pythonikus megoldás 3.8+ verzióban """
        return prod(range(1, n+1))
```

A függvény paraméterekhez adhatunk alapértelmezett értéket, lásd fahrenheit paraméter a celsius függvényben. A függvények hívásánál a paraméterek értékét a definíció sorrendjében adhatjuk meg vagy a paraméter nevének megadásával tetszőleges sorrendben (lásd az alábbi példákat).

A négy faktoriális függvény implementáció numerikusan helyes megoldást add. A programozásban, mint sok más helyen több jó megoldás készíthető ugyanarra a problémára. Hatékonyság szempontjából a harmadik és negyedik megoldás jobb az előzőeknél. A harmadik, rekurzív függvény nagyon nagy faktoriális esetén a verem (stack) megtelése miatt hibára fut.

```
> Python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from func import (celsius, root, f, fact, factorial)
>>> celsius()                       # 0 fahrenheit
-17.77777777777778
>>> celsius(-20)
-28.88888888888889
>>> root(c=5, b=-2, a=2)            # nincs valós gyök
>>> root(c=-5, b=-2, a=2)
(2.1583123951777, -1.1583123951777)
>>> celsius('alma')                 # nem szám paraméter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/siki/tutorials/hungarian/python/code/func.py", line 7, in celsius
        return (fahrenheit - 32) * 5.0 / 9.0
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Feladatok

- Próbáljuk meg az *x2 – 3 x +5 = 0* egyenlet gyökeit kiszámítani *(root(0, -3, 5))*! Írjuk át a függvényt a *try* utasítás használatával, hogy elkerüljük a hibaüzenetet!
- Próbáljuk kiszámítani a 1000! mindhárom függvényünkkel (f, fact, factorial)!
- Melyik a legyorsabb a három faktoriális függvény közül. Használja a *timeit* modult!
- Mi történik, ha negatív paramétert adunk meg a faktoriális függvénynek?
- Készítsünk függvényt a irányszög számításra (math.atan2)!
- Készítsünk függvényt a DMS szögek radiánba átváltására!

Objektumok

A Python objektum orientált programozást a 2D-s pontok osztályának elkészítésén keresztül mutatjuk be. A Python nyelvben minden osztály (például a lista vagy a szótár is, de a függvények is). Egy point2d.py nevű fájlban kezdjük el az osztály kódjának elkészítését.

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

class Point2D(object):                                  # object a bázis osztály
        """ kettő dimenziós pontok osztálya
        """
        def __init__(self, east = 0, north = 0):    # __init__ a konstruktor
                """ Pont inicializálás

                        :param east: első koordináta
                        :param north: második koordináta
                """
                self.east = east              # tagváltozó létrehozása
                self.north = north

        def abs(self):
                """ helyvektor hossza (absolút érték)
                        :returns: absolút érték
                """
                return (self.east**2 + self.north**2)**0.5
```

A *class* kulcsszóval kezdődik az új osztály definíciója. A zárójelben azokat az osztályokat sorolhatjuk fel, melyekből öröklődéssel alkul ki az új osztáy (lásd az öröklődés részt). Az *object* osztály a Python legelemibb osztálya ebből származik az összes többi osztály.

Az *__init__* metódus (az osztályhoz tartozó függvény) feladata az osztály példányainak az inicializálása. Az osztály egy példányának létrehozása során automatikusan meghívja a Python.

A *self* változó az osztály minden (nem statikus vagy osztály) metódusának az első paramétere és az objektum példányt jelenti, ezen keresztül hivatkozhatunk az objektum elemeire pl. self.east. A többsoros megjegyzés a sphinx programnak megfelelően készült, hogy automatikusan lehessen dokumentációt generálni a kódból (a Python dokumentációja is így készül).

Próbáljuk ki a fenti kódunkat.

```
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from point2d import Point2D
>>> p = Point2D()                  # példányosítás, 0, 0 pont
>>> p1 = Point2D(5)                # 5, 0 pont
>>> p2 = Point2D(2, -2)            # 2 – 2 pont
>>> print(p2.east)                 # tagváltozó értéke
2
>>> print(p2.north)
-2
>>> p.abs()
0.0
>>> p2.abs()
2.8284271247461903
>>> print(p2)
<point2d.Point2D object at 0x7f0ac1121af0>      # ??????????
>>> print(p2.__dict__)
{'east': 2, 'north': -2}
>>> print(p2.__doc__)
kettő dimenziós pontok osztálya
```

```
>>> print(p2.abs.__doc__)
helyvektor hossza (absolút érték)
    :returns: absolút érték
>>> help(Point2D)
Help on class Point2D in module point2d:

class Point2D(builtins.object)
 |  Point2D(east=0, north=0)
 |
 |  kettő dimenziós pontok osztálya
...
```

A *print(p2)* utasítás nem azt az eredményt adja amit szeretnénk. Az egyess osztályokhoz speciális függvényeket definiálhatunk (mint pl. az *__init__*), ezek jellemzője, hogy két aláhúzás karakterrel kezdődik és végződik a nevük. Az *__init__* függvény funkciója hasonló más objektum orientált nyelven használt konstruktorra, ez fut le a példányok létrehozásakor, itt biztosíthatjuk, hogy ne legyen inicializálatlan tagváltozónk. A destruktort a *__del__* függvény implementálásával valósíthatjuk meg. A dinamikus tárfoglalás hiányában a Pythonban erre ritkábban van szükség.

A *print* utasítás az osztály *__str__* metódusát hívja meg. A fenti példában ennek alapértelmezett változatának eredményét láthattuk az object osztályból (a példány típusa és a memória címe).

```
# a Point2D osztály kódjához adja hozzá
            def __str__(self):
                    """ Pont szöveggáccá alakítása kiiratáshoz

                            :returns: a koordináták szövegláncként
                    """
                    return "{:.3f}; {:.3f}".format(self.east, self.north)
```

```
> python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from point2d import Point2D
>>> p2 = Point2D(2, -2)
>>> print(p2)
2.000; -2.000
```

Python nyelvben az osztályokhoz további speciális függvényekkel műveleteket is definiálhatunk. A pontokat mint helyvektorokat is használhatjuk, készítsük el a helyvektorok összeadását. Ez egy művelet felülbírálása lesz (operator overriding), ha két Point2D objektum példény közé "+" jelet teszünk, akkor a Python az osztály *__add__* metódusát hívja meg.

```
# az alábbi kódot a Point2D osztályhoz adjuk hozzá

        def __add__(self, p):
                """ Két pont összeadása
                        :param p: hozzáadandó pont
                        :returns: a két pont összegéből képzett Point2D példányt
                """
                return Point2D(self.east + p.east, self.north + p.north)
```

Majd próbáljuk ki!

```
> python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from point2d import Point2D
>>> p1 = Point2D(1, -2)
>>> p2 = Point2D(4, 1)
>>> print(p1 + p2)
5.000; -1.000
```

Összezártság, elzártság

Az objektum orientál programozásban az összezártság azt fejezi ki, hogy az osztály metódusai az osztály változóit közvetlenül elérhetik (a *self* változón keresztül), nem kell azokat a paraméter listán átadni. Az elzártság azt jelenti, hogy az objektum példány változóihoz kívülről nem lehet közvetlenül hozzáférni. Az elkészített Point2D osztályunk az objektum orientált programozás ezen fontos kívánalmának, az elzártságnak nem tesz eleget. A pont osztály egy példányának a koordinátáit közvetlenül megváltoztathatjuk, ez azzal a következménnyel járhat, hogy

- a példány tagváltozóit az osztály felhasználója egy programhibából következően is átírhatja
- a példány tagváltozóit ellenőrzés nélkül is felül lehet írni. például a koordinátákat szövegláncként adjuk meg, ami csak később vezet hibára

```
> python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from point2d import Point2D
>>> p1 = Point2D(1, -2)
    >>> print(p1)
    >>> p1.east = 5
    >>> print(p1)
5.000; -2.000
    >>> p1.north = '122'                    # !!!!!!
    >>> p1.abs()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/home/siki/tutorials/hungarian/python/code/point2d.py", line 20, in abs
            return (self.east**2 + self.north**2)**0.5
    TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Ennek kivédésére privát tagváltozókat ("__" karakterekkel kezdődő név) vagy védett (protected) tagváltozókat ("_" karakterrel kezdődő név) és getter/setter metódusokat használhatunk. Írjuk át az osztályunkat, hogy csak egy-egy metóduson keresztül lehessen megváltoztatni a tagváltozókat. A privát tagváltozókat csak annak az osztályhoz tartozó metódusok módosíthatják. A védett változókat az osztályhoz tartozó metódusok mellett az osztályból öröklődéssel származtatott osztályok metódusai is módosíthatják. A Python osztályok privát tagváltozóinak neve két aláhúzás karakterrel kezdődik.

Írjuk át a Point2D osztályt a tagváltozók eléréshez készítsünk külön metódusokat, melyekbe ellenőrzéseket is elhelyezhetünk. Ez az *__init__* és a többi metódus módosítását is megköveteli. Használjunk privát tagváltozókat (point2d_1.py).

```python
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

class Point2D(object):                          # object a bázis osztály
        """ kettő dimenziós pontok osztálya
        """
        def __init__(self, east = 0, north = 0):    # __init__ a konstruktor
                """ Pont inicializálás

                        :param east: első koordináta
                        :param north: második koordináta
                """
                self.setEast(east)
                self.setNorth(north)

        def setEast(self, east):
                """ első koordináta beállítása """
                if type(east) in (int, float):
                        self.__east = east
                else:
                        raise Exception("hibás koordináta típus")

        def setNorth(self, north):
                """ második koordináta beállítása """
                if type(north) in (int, float):
                        self.__north = north
                else:
                        raise Exception("hibás koordináta típus")

        def getEast(self):
                """ első koordináta lekérdezése """
                return self.__east

        def getNorth(self):
                """ második koordináta lekérdezése """
                return self.__north

        def abs(self):
                """ helyvektor hossza (absolút érték)
                        :returns: absolút érték
                """
                return (self.__east**2 + self.__north**2)**0.5

        def __str__(self):
                """ Pont szöveggáccá alakítása kiiratáshoz

                        :returns: a koordináták szövegláncként
                """
                return "{:.3f}; {:.3f}".format(self.__east, self.__north)

        def __add__(self, p):
                """ Két pont összeadása
                        :param p: hozzáadandó pont
                        :returns: a két pont összegéből képzett Point2D példányt
                """
                return Point2D(self.__east + p.getEast(), self.__north + p.getNorth())
```

Ebben a megoldásban a setter metódusokat további vizsgálatokkal egészíthetjük ki, hogy például az EOV koordináta tartományokat ellenőrizzük. Az előző megoldáshoz képest védettebbek a tagváltozók a véletlen módosítástól, a változó neve helyett mindenhol a getter/setter függvényeket kell hívni, ami miatt többet kell gépelni.

Végül nézzük meg azt a megoldást, mely lehetővé teszi a tagváltozóra közvetlen hivatkozást és a getter/setter metódusok közvetett hívását. A Pythonban javasolt megoldás a @property illetve a @név.setter dekorátor használata. A osztályhoz hozzáadtunk még egy *__abs__* metódust, melyet az *abs* függvény hív meg, ha a paramétere egy Point2D objektum példánya.

Mivel ebből az osztályból később más osztály szeretnénk öröklődéssel származtatni, módosítsuk a privát tagváltozókat védett tagváltozókká (dupla aláhúzás helyett egy aláhúzás legyen a név elején, point2d_2.py).

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

class Point2D(object):                          # object a bázis osztály
        """ kettő dimenziós pontok osztálya
        """
        def __init__(self, east = 0, north = 0):    # __init__ a konstruktor
                """ Pont inicializálás

                        :param east: első koordináta
                        :param north: második koordináta
                """
                self.east = east                        # setter implicit hívása
                self.north = north

        @property
        def east(self):
                """ első koordináta lekérdezése """
                return self._east

        @property
        def north(self):
                """ második koordináta lekérdezése """
                return self._north

        @east.setter
        def east(self, east):
                """ első koordináta beállítása """
                if type(east) in (int, float):
                        self._east = east
                else:
                        raise Exception("hibás koordináta típus")

        @north.setter
        def north(self, north):
                """ második koordináta beállítása """
                if type(north) in (int, float):
                        self._north = north
                else:
                        raise Exception("hibás koordináta típus")

        def abs(self):
                """ helyvektor hossza (absolút érték)
                        :returns: absolút érték
                """
                return (self._east**2 + self._north**2)**0.5
```

```
        def __abs__(self):
                """ abs függvényhez """
                return self.abs()

        def __str__(self):
                """ Pont szöveggáccá alakítása kiiratáshoz

                        :returns: a koordináták szövegláncként
                """
                return "{:.3f}; {:.3f}".format(self._east, self._north)

        def __add__(self, p):
                """ Két pont összeadása
                        :param p: hozzáadandó pont
                        :returns: a két pont összegéből képzett Point2D példányt
                """
                return Point2D(self._east + p.east, self._north + p.north)
```

A @property dekorátorral bevezetett lekérdező függvénynek meg kell előznie a kódban a @xxx.setter dekorátorral indított metódust.

Öröklödés, többértelműség

Az öröklődés segítségével már létező osztályok funkcionalitását bővíthetjük ki, anélkül, hogy a szülő osztály kódján változtatnánk. Az öröklődés bemutatására készítsünk egy Point3D osztályt. A származtatott osztályban a szülő osztály metódusait felülbírálhatjuk szükség esetén, illetve újabb metódusokat hozhatunk létre. A származtatott osztály örökli a szülő osztály tagváltozóit és újabb tagváltozókat definiálhatunk. Az objektumorientált programozásban a többértelműség azt is jelenti, hogy több ugyanolyan nevű függvényt hozhatunk létre eltérő paraméterlistával (a paraméterek típusa vagy száma eltérő). Ezt a Pythonban csak korlátozottan alkalmazhatjuk, mivel a függvények paramétereihez nem adhatunk meg változó típust. Az előző példában az *__abs__* függvény egy példa. Ezt akkor hívja meg a Python értelmező, ha az *abs* függvény argumentuma egy Point2D típusú változó, de emellett egész vagy valós értékekkel is használhatjuk (point3d.py).

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
from point2d_2 import Point2D

class Point3D(Point2D):
        """ háromdimenziós pontok osztálya

                        :param east: első koordináta
                        :param north: második koordináta
                        :param elev: magasság
        """

        def __init__(self, east = 0, north = 0, elev = 0):
                Point2D.__init__(self, east, north)
                self.elev = elev

        @property
        def elev(self):
                return self._elev

        @elev.setter
```

```
        def elev(self, elev):
                if elev < 10000:
                        self._elev = elev
                else:
                        raise ValueError('elev must be less than 10000')

        def abs(self):
                """ helyvektor hossza (absolút érték)
                        :returns: absolút érték
                """
                return (self._east**2 + self._north**2 + self._elev**2)**0.5

        def __abs__(self):
                """ abs függvényhez """
                return self.abs()

        def __str__(self):
                """ String representation of the 2D point

                        :returns: point coordinates as string (e.g. 5; 3)
                """
                return super(Point3D, self).__str__() + "; {:.3f}".format(self._elev)
```

A Point3D osztály *__init__* metódusában felhasználjuk a bázis osztály inizializáló metódusát a *Point2D.__init__(self, east, north)*, mely csak a kelet és észak koordinátákat állítja be, utána kiegészítjük a magasság inicializálásával (hasonlóan jártunk el az *__str__* metódus esetén). Nem duplikáltuk a kódot, mert ha a 2D-s pontok osztályának készítője módosítja az osztályát az automatikusan kihat a származtatott osztályra. A kód duplikálása esetén ez nem lenne igaz.

A bázis osztály metódusait a származtatott osztály örökli, így nem kell a kelet és észak koordináta setter/getter metódusait megismételni.

A Point3D osztály *__abs__** metódusa felülírja ebben az osztályban az abszolút érték számítását.

Probáljuk ki a háromdimenziós pontok osztályát:

> python3 Python 3.8.2 (default, Jul 16 2020, 14:00:26) [GCC 9.3.0] on linux Type "help", "copyright", "credits" or "license" for more information. >>> from point3d import Point3D >>> p1 = Point3D(600000,200000,100) >>> print(p1) >>> print(p1.abs()) 632455.53993937 >>> print(abs(p1)) 632455.53993937 >>> print(abs(-1)) # többértelműség az abs függvény a paramétereének megfelelő osztály __abs__ metódusát hívja 1 >>> print(abs('abc')) # karakterláncokra nincs abszolút érték Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: bad operand type for abs(): 'str'

Feladatok

- Készítse el a 3D pontok összeadását megvalósító metódust.
- Származtasson egy kör osztályt a Point2D osztályból

Python térinformatikai alkalmazása

Mielőtt egy programozási feladatot meg szeretnénk oldani érdemes körülnézni az interneten, hátha valaki már már készített egy Python modult, mely segít a megoldásban. Az alábbiakban néhány térinformatikai példát mutatok be nyílt forráskódú Python modulok felhasználásával.

A legtöbb Python bővítő modult a PyPi (https://pypi.org) oldalon találhatjuk. Az itt található modulokat a *pip* programmal telelpíthetjük a parancssorból.

-- code:

```
> pip install csomag_név
```

A példákbn felhasznált adatállományok a http://www.agt.bme.hu/ftp/foss/mo.zip címről letölthetőek.

OGR, ezdxf

Nézzünk egy egyszerű példát. Oldjuk meg azt a feladatot, hogy egy shape fájl egyik attribútumából készítsünk feliratot egy AutoCAD DXF fájlban, a felirat beszúrási pontja legyen a geometriai elem súlypontja felületek esetén és az első pontja törvonal vagy pont elem esetén. A shape fájl olvasására az OGR könyvtárat használjuk, mely vektoros térinformatikai formátumok kezeléséhez készítettek. Az OGR a GDAL könyvtár része, a PyPi oldalról a GDAL modult kell telepítenie. A DXF fájl létrehozására az ezdxf modult, használjuk, mely szintén megtalálható a PyPi oldalon (shp2dxftxt.py).

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
""" shp attribútum átvitele DXF szövegbe
        pont típusú rétegnél a szöveg a pontba kerül
        törtvonal rétegnél az első pontba
        felület típusú rétegnél a szöveg a centrálisba kerül
        opcionálisan megadható egy oszlop a szöveg szögével és egy szöveg méret
"""
from osgeo import ogr                   # shp olvasáshoz
import ezdxf                                    # dxf iráshoz
from os import (path, unlink)
import sys
if len(sys.argv) < 4:
   print("usage: {} <input_shp> <output_dxf> <txt_column> [rotation_column] [text_height]".format(sys.argv[0]))
   exit(1)
# input réteg
shpfile = sys.argv[1]
if not path.exists(shpfile):
   print("Shape fájlt nem találom")
   exit(2)
# input shape megnyitása
inDriver = ogr.GetDriverByName("ESRI Shapefile")
inDataSource = inDriver.Open(shpfile, 0)
inLayer = inDataSource.GetLayer()
# output dxf file
dxffile = sys.argv[2]
dxf = ezdxf.new(dxfversion='AC1015') # AutoCAD R2000
if path.exists(dxffile):
        unlink(dxffile)
modelSpace = dxf.modelspace()
col = sys.argv[3]        # oszlopnév a felirathoz
rot = 0                  # alapértelmezett szövegirány 0 fok
if len(sys.argv) > 4:
        rotcol = sys.argv[4]
h = 1                    # alapértelmezett szöveg magasság
if len(sys.argv) > 5:
        h = float(sys.argv[5])
# térképi elemek feldolgozása
for feature in inLayer:
        geom = feature.GetGeometryRef()
        if geom.GetGeometryName() == "POLYGON":
                pp = geom.Centroid()    # szöveg a centrálisba
    p = (pp.GetX(), pp.GetY(), pp.GetZ())
        else:
                p = geom.GetPoint(0)   # szöveg az első pontba
        label = feature.GetField(col)
        if len(sys.argv) > 4:
                rot = feature.GetField(rotcol)
        modelSpace.add_text(label, \
                dxfattribs={'height': h, 'rotation': rot}).set_pos(p)
dxf.saveas(dxffile)
```

A fenti kódban a paraméterek átvétele, ellenőrzése után a réteg elemeit egyesével dolgozzuk fel. A DXF fájlt a memódiában hozzuk létre és bővítjük, majd a legvégén írjuk ki a megadott fájlba.

A használatra példák:

```
python3 shp2dxftxt.py varos.shp varosnev.dxf NEV
python3 shp2dxftxt.py folyo.shp folyonev.dxf NEV
python3 shp2dxftxt.py megye.shp megyenev.dxf Nev
```

Feladatok

- Alakítsa át a programot, hogy a DXF fájl létrehozását is az OGR könyvtárral valósítsa meg
- Ellenőrizze, hogy különböző kódlapok esetén az ékezetes karakterek helyesen kerülnek-e át a DXF-be, szükség esetén módosítsa a kódot

GDAL

A GDAL könyvtár a raszteres térinformatikai adatok kezelését teszi lehetővé. A Mátrát és a Bükköt tartalmazó domborzatmodellből (TIF) készítsünk kimutatást egy megadott magasság alatti terület nagyságáról. A program elkészítéséhez a GDAL könyvtárat használjuk (dtmarea.py).

```
#!/usr/bin/env python3
    # -*- coding: UTF-8 -*-
    from osgeo import gdal
    from gdalconst import GA_ReadOnly
    import struct
    from sys import argv

    if len(argv) < 3:
            print("Használat: {} dtm_tiff magasság".format(argv[0]))
            exit(1)
    data = gdal.Open(argv[1], GA_ReadOnly) # DTM megnyitása
    if data is None:
            print("TIFF nyitási hiba")
            exit(2)
    try:
            height = float(argv[2])
    except ValueError:
            print("Hibás magasság")
            exit(3)
    geotr = data.GetGeoTransform()
    band = data.GetRasterBand(1)                    # első sáv a képből
    no_data = band.GetNoDataValue()                 # nincs adat érték lekérdezése
    fmt = "<" + ("f" * band.XSize)                  # formátum 1 sorhoz, 32 bit valós
    pixel_area = abs(geotr[1] * geotr[5])   # egy pixel területe
    area = 0.0                                      # terület összegzéshez

    for y in range(band.YSize):                             # minden pixel sorra
            scanline = band.ReadRaster(0, y, band.XSize, 1, band.XSize, 1, band.DataType)
            values = struct.unpack(fmt, scanline)   # magasságok beolvasása és listává alakítása
            for value in values:
                    if value < height and value != no_data:
                            area += pixel_area
    print(area)
```

Feladatok

- Alakítsa át a programot, hogy két magasság közé eső területet számoljon

- Alakítsa át a programot, hogy egy függvény számítsa a területet
- Alakítsa át a programot, hogy az átlag magasságot számítsa ki

Shapely

A Shapely modul segítségével a GEOS könyvtár funkcionalitását érhetjük el. A GEOS C++ nyelven írt, geometriai számítások végrehajtására készített programkönyvtár. A PostGIS is a GEOS-t használja. Az alábbi példában a varosok köré 30 km-es övezetet készítünk egy új rétegbe, a Shapely mellett a már korábban használt OGR modul segítségével olvassuk a shape fájlokat (buffer.py).

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

from osgeo import ogr
import shapely.wkt
import os.path

shapefile = ogr.Open("varos.shp")       # input pont shape
if shapefile is None:
        print("Nem találom a varos.shp fájlt")
        exit(1)
layer = shapefile.GetLayer(0)

driver = ogr.GetDriverByName("ESRI Shapefile")
outshp = "varosb.shp"
if os.path.exists(outshp):              # létező output törlése
        driver.DeleteDataSource(outshp)
dstFile = driver.CreateDataSource("varosb.shp") # output shape létrehozása
dstLayer = dstFile.CreateLayer("layer", geom_type=ogr.wkbPolygon)
field = ogr.FieldDefn("id", ogr.OFTInteger)     # id mező létrehozása
dstLayer.CreateField(field)

for i in range(layer.GetFeatureCount()):
        feature = layer.GetFeature(i)       # pont az inputból
        geometry = feature.GetGeometryRef()
        wkt = geometry.ExportToWkt()        # átalakítás wkt formátumba
        p = shapely.wkt.loads(wkt)          # átalakítás shapely geometriába
        pb = p.buffer(30000)                # 30 km buffer
        wktb = shapely.wkt.dumps(pb)        # export wkt-ba
        feature = ogr.Feature(dstLayer.GetLayerDefn())
        feature.SetGeometry(ogr.CreateGeometryFromWkt(wktb))
        feature.SetField("id", i)           # id
        dstLayer.CreateFeature(feature)

dstFile.Destroy()                       # mentés és output lezárás
```

Feladatok

- Alakítsa át a programot, hogy a parancssorból kapja meg az input shape fájl nevét és az övezet méretét
- Próbálja ki, hogy nem pont típusú rétegre használható-e a program

Internetes tartalom elérése

A Python *urllib* könyvtár segítségével a web szerverek szolgáltatásait elérhetjük. Egy példén keresztül a tanszéki honlapon elérhető cm-es pontosságot biztosító ETRS89-EOV átszámítási szolgáltatást használjuk.

```
> python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib.request
>>> req = urllib.request.urlopen("http://www.agt.bme.hu/on_line/etrs2eov/etrs2eov.php?e=650000&n=240000&sfradio=single&format=TXT").read()
>>> print(req)
b'1 19.0474474 47.5039331\n'
```

A fenti példában HTTP GET típusú kérést mutattunk be. Ha több adat átküldése szükséges a szerverhez, akkor a POST típusú kérések célszerűbbek. Az alábbi példában a POST adatküldést mutatjuk be.

```
> python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib.request
>>> import urllib.parse
>>> values = {'e': 650000, 'n': 240000, 'sfradio': 'single', 'format': 'TXT'}
>>> data = urllib.parse.urlencode(values).encode('ascii')
>>> req = urllib.request.Request("http://www.agt.bme.hu/on_line/etrs2eov/etrs2eov.php, data)
>>> res = urllib.request.urlopen(req).read()
>>> res
b'1 19.0474474 47.5039331\n'
```

Feladatok

- Írjon egy programot, mely a parancssorból beolvasott ETRS89 vagy EOV koordinátát a másik koordinátarendszerbe számít át
- Készítsen programot, mely egy szövegfájlban tárolt pontokat számít át a webes szolgáltatással

Kapcsolódás relációs adatbázishoz

Python nyelvből számos relációs adatbáziskezelőhöz kapcsolódhatunk. A Python DB API segítségével olyan kódot készíthetünk, mely minimális mértékben függ csak a használt adatbáziskezelőtől. A DB API moduljai objektumokat kínálnak a programozóknak és nevük általában a „db” betűkkel végződik, pl. MySQLdb (MySQL), de psycop2 (PostgreSQL). A megfelelő adatbáziskezelőhöz tartozó DB API modult telepítenünk kell a számítógépünkre, ezután a Python programunkba importálhatjuk a modult. A megfelelő adatbáziskezelőhöz tartozó DB API modult telepítenünk kell a számítógépünkre, ezután a Python programunkba importálhatjuk a modult.

A PostgreSQL DB API modult psycopg2-binary névvel találjuk meg (tablelist.py).

```
> pip3 install psycopg2-binary
```

Az alábbi mintapéldában egy tábla tartalmát iratjuk ki.

```
# -*- coding: UTF-8 -*-
import psycopg2 as db   # PostgreSQL meghajtó betöltése
from sys import argv
```

```
if len(argv) < 5:
        print("Használat: {} database user password table".format(argv[0]))
        exit(1)
con = None                              # kapcsolat változó inicializálása
try:
        con = db.connect(database=argv[1], user=argv[2], password=argv[3])
        cur = con.cursor()                                              # cursor a lekérdezéshez
        cur.execute('SELECT * FROM {}'.format(argv4])   # adatok lekérdezése
        rec = cur.fetchone()                                    # következő sor a lekérdezésből
        while rec:
                print(rec)
                rec = cur.fetchone()
except db.DatabaseError as e:
        print('Hiba: {}'.format(e))                             # hibaüzenet kiírása
        exit(1)
finally:
        if con:                                                                 # kapcsolat lezárása
                con.close()
```

A Python programból egy adatbázis kapcsolat objektumot kell létrehozni, majd ezen a kapcsolaton keresztül SQL utasításokat hajthatunk végre az adatbázison. Lekérdezésekhez úgynevezett cursor-t használunk, melynek segítségével akár egyesével kaphatjuk meg és dolgozhatjuk fel a lekérdezés eredményét. A kapcsolat objektum létrehozásakor több paramétert adhatunk meg, melyek többségére alapértelmezett érték is rendelkezésre áll (itt a gép: localhost és port: 5432 alapértelmezéseket használtuk).