Alkalmi Python - 10. rész

PCLinuxOS Magazine - 2019. november

Írta: Peter Kelly (critter)

Hírolvasó

Ehhez telepítened kell a fondorlatos elnevezésű **python3-beautifulsoup4**-et a tárolókból. A modul html és xml fájlok elemzésére használatos, amit néha web-kaparászás néven is ismernek. Szükséged lesz még a python3-request-re is, ami feltehetően már telepítve van.

Nem igazán tudván, hogyan írjam le az rss hírforrás értelmét, noha gyakran használtam a kifejezést, ezért ezt másoltam ki egy weblapról:

*„Az **RSS** a Really Simple Syndication rövidítése. Ez egyszerű módja hírösszefoglaló-listák, frissítési értesítések és esetenként tartalom széles körű megosztásának. Számítógépprogramok használják fel, amik a hírösszefoglalókat és értesítéseket a könnyebb olvashatóság érdekében rendezik."*

Szeretek a hírekkel tisztában lenni és Britanniában élőként a BBC hírösszefoglalóit használom gyakran. A „feed"-ek által adott információ a főbb hírszolgáltatók által jelentett esemény nagyon rövid összefoglalója, de tartalmaz egy hivatkozást részletesebb információhoz. Elhatároztam, hogy készítek egy alkalmazást, ami megmutatja a rövid verziót, de megnyitja az egészet, a teljes weblapot, képeket és mindent, amikor a számomra érdekes elemre kattintok.

Ez valójában az alkalmazáskereső egy alternatívája . Kapunk némi információt, megjelenítjük és amikor az egyik információs elem egérkattintást kap, akkor az elem elindul, vagy ebben az esetben további információk jelennek meg. A megjeleníteni szándékozott információ a hivatkozással kapcsolt weblap. Kell még egy az eredeti listához visszatérítő útvonal.

A kód elég rövid, olyan 75 sor körüli és ennek majd felét korábban már láttuk. Ugyanakkor van néhány új elem is.

Annak érdekében, hogy megjelenítsük az internetes információkat, szükségünk van egy Qt5 interfész kapcsolóra – a QwebEngineView-ra a QtEngineWidgets-ből. Importálhatjuk ezt a kódunk elején.

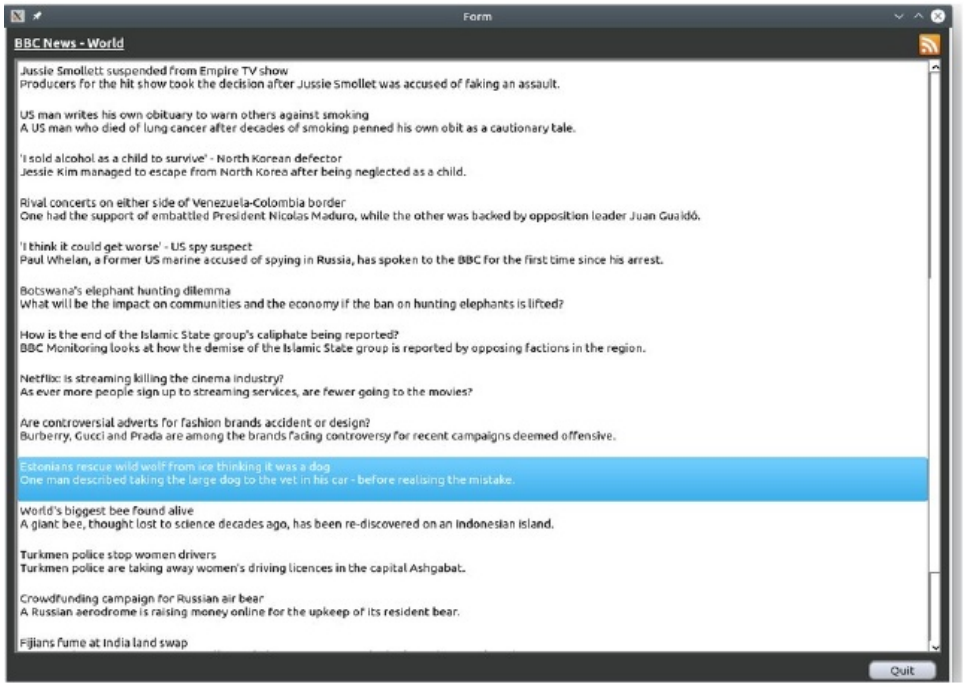
Az importált requests modul „lekérdez" egy weblapot az internetről és visszaadja a tartalmát. Tároljuk, majd keressük a minket érdeklő tartalmat a beautifulsoup eljárásokkal. Úgy vélem, ezt hívják „web scaping"-nek (webes kaparászásnak).

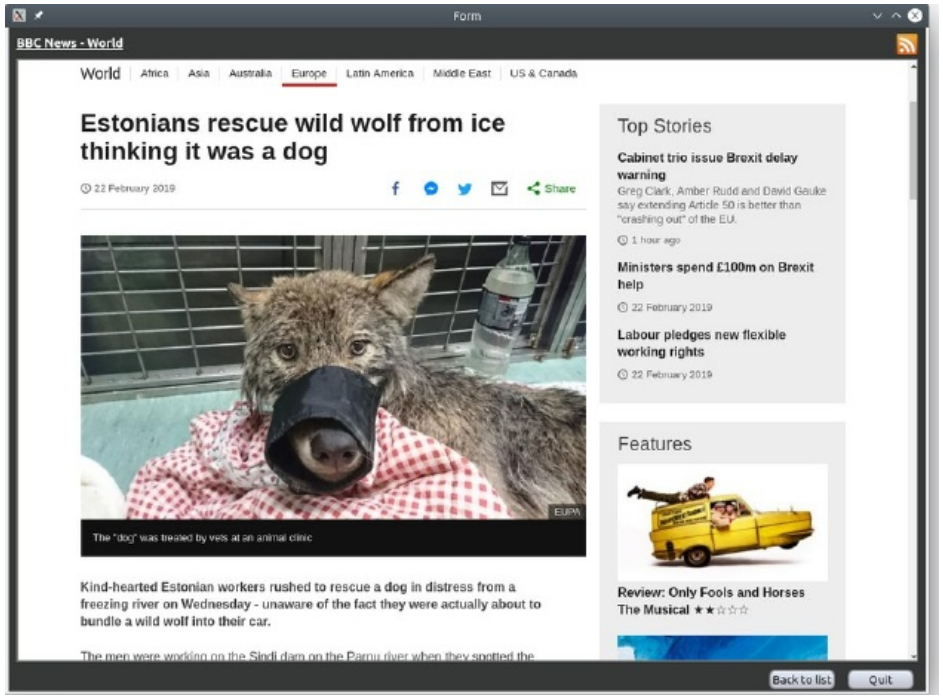
Az egyik rendszeremen a kód nem indult el geany alól, de jól működött, amikor a Dolphine fájlkezelőből, vagy parancssorból hajtottam végre. A problémát sikerült kiküszöbölni, amikor egy egyszerű beállítást megváltoztattam. A Geany-ban Build (Összeállítás)→ Set Build (Összeállítási parancsok megadása) parancsot és váltsd át a „python"-t „python3"-ra mind a compile (független parancsok), mind az execute (Parancs futtatása) parancsdobozokban. Ne felejtsd el végrehajthatóvá tenni a kódodat.

A példában használt bbc forrás **itt** érhető el:

Sok további érhető el, néhány kipróbálásra érdemes közülük:

UK hírek http://feeds.skynews.com/feeds/rss/uk.xml
World news http://feeds.skynews.com/feeds/rss/world.xml
US news http://feeds.skynews.com/feeds/rss/us.xml
Techology news http://feeds.skynews.com/feeds/rss/technology.xml
Strange news items http://feeds.skynews.com/feeds/rss/strange.xml
(Magyar rss-ket innen lehet kinézni: https://rss.lap.hu/ - fordító.)





A felhasználói felület a Designer-ben

A felhasználói felület a szokásos. A sablonkönyvtár egy másolatával indítottam, eltávolítottam azt, amire nincs szükség és hozzáadtam az ikont, majd átméreteztem az űrlapot és hozzáadtam, átneveztem a következő elemeket. A fejléc bal felső részéhez tartozik egy objectName lbl_titles a szövege „Title” változik futás közben, megjelenítve az éppen nézett oldal címét. A fejléc jobb felső oldalánál nincs szöveg, csak az rss logó, amit a usr/share/icons-ból „kölcsönöztem”. Két nyomógombot, amikhez objectName btn_Back és btn_Quit tartozik, jobbra lent helyeztem el, miközben nagy fehér négyszög középen egy tároló widget alapbeállítás szerinti objectName stackedWidget-tel van. Az első oldalon van egy QlistWidget, amit én news_list-nek neveztem el. A második oldalon található egy webEngineView widget, objectName weEngineView-val. Ezek az objectName-ek fontosak, hogy az alkalmazás kódja felismerje. Néhány stíluslapot adtam a felület alapmegjelenésének megváltoztatásához, de ez az én személyes heppem. A kezelői felület teljes méretét 1170, 850-ben állapítottam meg. Mikor a felület elkészült, mentsd és szerkeszd meg az update_res.sh-t, hogy az új fájlneveket tartalmazza. Futtasd az update_res.sh-t a reader_ui.py létrehozása érdekében.

Íme a „becsukott” kód:

newsreader.py

```
1   #!/usr/bin/python3
2   # -*- coding: utf-8 -*-
3
4   import sys
5   from PyQt5.QtWidgets import *
6   from PyQt5.QtCore import *
7   from PyQt5.QtGui import *
8   from PyQt5.QtWebEngineWidgets import QWebEngineView
9
10  from bs4 import BeautifulSoup
11  import requests
12
13  import reader_ui
14
15
16  class News(QMainWindow, reader_ui.Ui_Form):
17
18      def __init__(self):
44
45      def show_list(self):
50
51      def show_html(self, curr):
59
60      def keyPressEvent(self, e):
65
66      def exitApplication(self):  # Exit point
69
70
71  if __name__ == '__main__':
77
```

A kód

A 2. sor közli a python3 interpreterrel, hogy utf-8 kódolást használjon a szöveghez. A hír elemek gyakran tartalmaznak olyan betűket, amiket az ascii nem definiált, ezzel biztosítjuk az ilyen karakterek helyes megjelenítését.

- * - coding: utf-8 - * -

A BeautifulSoup modul bs4 nevet kapott, de nekünk csak a BeautifulSoup-ot kell importálnunk. Lásd: https://www.crummy.com/software/BeautifulSoup/bs4/doc-ot a további információkért.

Az __init__ eljárás

```
def __init__(self):
    super(self.__class__, self).__init__()
    self.setupUi(self)

    self.btn_Quit.clicked.connect(self.exitApplication)
    self.btn_Quit.setToolTip('click or press the escape key to exit')
    self.btn_Back.setVisible(False)
    self.btn_Back.setEnabled(False)
    self.btn_Back.setToolTip(
        'click or press back arrow ← to return to the list')
    self.btn_Back.clicked.connect(self.show_list)
    page = requests.get("http://feeds.bbci.co.uk/news/world/RSS.xml")
    soup = BeautifulSoup(page.content, 'html.parser')
    title = soup.find('title').get_text()
    self.lbl_title.setText(title)

    self.url = []
    for i in range(1,26):
        try:
            t = soup.find_all('title')[i + 1].get_text()
        except IndexError:
            break
        d = soup.find_all('description')[i].get_text()
        item = t + '\n' + d + '\n'
        self.news_list.addItem(item)
        self.url += [soup.find_all('guide')[i - 1].get_text()]

    self.news_list.itemClicked.connect(self.show_html)
```

A szokásos beállító kódot követően a Quit-gomb az applicationExit eljáráshoz kapcsolódik. Az első szokatlan kód az, ahol a Back gomb attribútumai között a Visible-t (láthatóság) és Enable-t (engedélyezett) False-ra (hamis) állítjuk, amikor az alkalmazás indításakor nem kell a visszaléptetés, mivel nincs hova. A vissza gomb a listamegjelenítő eljáráshoz kapcsolódik.

A requests (lekérések) modulok get eljárása arra szolgál, hogy a hírfolyam címéről lekérjen egy weblapot és az eredményt a név szerinti oldal számára jelölje ki. Az eredmény nem szöveg, hanem válasz, egy objektum, amit a BeautifulSoup-nak adunk át. Ezután közöljük a BeautifulSoup-pal, hogy a html.parser-ét a page.content-el együtt használja és készítsen egy referenciát az eredményhez a name soup-ot használva. Ezután a soup find eljárását használjuk, hogy megtalálja a 'title' nevű szöveget és tárolja azt a szöveget a 'title'-változóban. Ezt a szöveget a lbl_title-ban, a felhasználói felület bal felső részében használjuk, így tudjuk melyik oldalt nézzük éppen.

Ha a fenti komplikáltnak látszana, az is, de ez a kis kód bármilyen hasonló weblapra alkalmazható.

Self.url = [] üres listát készít

Megtalálván egy címet ('title'), ami a weblap címe, most egy második és további címeket kell megtalálni, amik a cikk címek (**article titles**) és a hozzájuk tartozó cikk leírásokat (**article descriptions**).

Mivel nem tudjuk, hogy mennyi cikk lesz a weblapon, be kell állítanunk egy határt és tesztelni kell, hogy mi történik, amikor elérjük a limitet, vagy amikor a python dob egy kivételt (egy hibát).

A try/except hurok a kivételeket kezelő szabványos eljárás. Próbálunk valamit csinálni, és ha sikeres, akkor folytatjuk. Ha nem sikeres, akkor kivételt indít, de mi ekkor elkapjuk és kilépünk a hurokból. A határt a range függvénynél 25 elemben határoztam meg.:

for i in range(1,26) # 26-ig de azt el nem érően

Ez fura, mivel mi a második címet akarjuk, ám az első leírás és az indexelés is nulláról indul.

	1st title	site title
i from range		
1	2nd title	article title
	1st description	article description
	1st guide	article guide

```
2       3rd title               article title
        2nd description         article description
        2nd guide               article guide

3       4th title               article title
        3rd description         article description
        3rd guide               article guide
…
…
```

A cím (title) és leírás (description) t és d nevet kapott és tárolunk egy címet, egy új sort, egy leírást és egy új sor karaktert egy item nevű sztringben, és hozzáadjuk a new_list listwidget-ben.

Ez a sor: **self.url +- [soup.find_all('guide')[i – 1].get_text()]** hozzá adja a listaelem url-jét a list objektumhoz, amit korábban hoztunk létre. Levonunk 1-et i-ből, hogy visszaállítsuk a 0 bázisú indexelést a self.url lista objektumában.

A list widget **itemClicked** eseménye a show_html eljáráshoz van kapcsolva.

Kell egy kis idő megérteni ezt, de gondolj az eredményre. Bármilyen rss hírösszefoglaló az interneten, amit csak elérünk és olyan kevés, vagy sok információt hívhatunk le rákattintva, amennyit csak akarunk.

A show_html eljárás

```
def show_html(self, curr):
    self.btn_Back.setDisabled(False)
    self.btn_Back.setVisible(True)
    new_url = QUrl(self.url[self.news_list.currentRow()])
    self.stackedWidget.setCurrentIndex(1)
    self.webEngineView.load(new_url)
```

Először megjelenítjük és elérhetővé tesszük back bombot.

Ezután vesszük a clicked item-hez kapcsolódó url-t, ezeket akkor tároltuk el, amikor létrehoztuk a list widget tartalmát.

Mozgatás a stacked widget 2. lapjára – webes nézet.

Az url betöltése a web nézőbe a megjelenítéshez.

A show_list eljárás

```
def show_list(self):
    self.stackedWidget.setCurrentIndex(0)
    self.btn_Back.setDisabled(True)
    self.btn_Back.setVisible(False)
```

Belépés a stacked widget 1. oldalára - a cikklista.
A back gomb elrejtése és kikapcsolása.

A kód maradéka megegyezik a már sokszor használt kóddal.

A keypressEvent eljárása (gomb lenyomása)

```
def keyPressEvent(self, e):
    if e.key() == Qt.Key_Escape:
        self.exitApplication()  # cancel the app
    if e.key == Qt.Key_Left:
        self.show_list()
```

Az exitApplication eljárás (kilépés alkalmazásból)

```
def exitApplication(self):  # Exit point
    self.close()
    sys.exit()
```

A végső kód

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    form = News()
    form.show()
    app.exec_()
```

Modulok és hatókör

A modul egy végrehajtható kódot tartalmazó fájl, amit bármilyen más programfájl importálhat. A kód lehet python kód, lefordított c kód, vagy egy a sok más típusból. Minden eddig megírt fájlunk használható modulként.

Amikor importáltuk a subprocess (al-folyamat) modult, hozzáfértünk a Popen eljáráshoz (további 30 más eljárással együtt). A Popen használatához annak a modulnak a nevével kell előtagként ellátnunk, ahonnan származik: subprocess.Popen(). Ez pontozott jelölésként ismert és az előtagot névtérnek (namespace) nevezik. Az aktuális modul névterét. az alkalmazásunk végrehajtási kódját, mindig __main__ -nek hívják. Emiatt van ez a sor:

if __name__ = = __main__:

Gondold át a következőket:

my_maty.py

```
#!/usr/bin/env python3

def power(x, y): # define a function
  print('{} ** {} = {}'.format(x, y, x ** y))

if _ _name__ == '__main__':
   power(2, 10) # test code
   power(2, 0.5)
  power(2, -3)
```

mathematics.py

```
import my_math # import a module

class newMath: # create a class
  def __init__(self, x, y): # intialize the class
    self.x = x
    self.y = y
  def power(self): # define a class method
    print('{} raised to the power {} is {}'.format(x, y, x ** y))
def power(x, y): # define a local function
    print( '{} raised to the {}th power is {}'.format(x, y, x ** y))


if __name__ == '__main__':
   x, y = 3, 4 # define 2 variables
   n = newMath(x, y) # create a class instance


   my_math.power(x, y) # imported function
   n.power() # class method
   power(x, y) # local function
```

Ha futtatjuk a my_mathematics.py-t, akkor a fájl végénél lévő teszt-kód lefut.

```
my_math.py ==> 2 ** 10 = 1024
           ==> 2 ** 0.5 = 1.41421356237
           ==> 2 ** -3 = 0.125
```

Ugyanakkor, ha importáljuk a my_math-ot (a py nem kell, vagy engedélyezett), akkor a teszt-kód nem fut le, de hozzáférünk a hatványozási függvényhez, amit tartalmaz.

```
mathematics.py ==> 3 ** 4 = 81
               ==> 3 raised to the power 4 is 81
               ==> 3 raised to the 4th power is 81
```

Három függvényünk, eljárásunk van itt, ami ugyanazt a számítást végzi el, és azonos eredményt ad, ám a kimenete másmilyen. A Python felismeri az általunk használt előtagból, hogy melyik függvényt hívjuk meg:

* my_math.power(x, y) meghívja a my_math névtérrel meghatározott függvényt. A névtér egy olyan hely, ahol a függvények, az eljárások, a változók és attribútumok neveit tárolják. A my_math modul importálása beolvassa a tartalmazott függvények és változók neveit és eltárolja egy általa my_math néven létrehozott névtérben.

* Az n változó a newMath osztály egy példányára hivatkozik, így az n.power a névtérben tárolt newMath egy eljárására hivatkozik a név alapján.

* A call power(x, y) függvény nincs előtaggal ellátva, ezért a python először az aktuáli (__main__) névtérben keres egy power nevű függvényt, és ha nem találta meg, akkor a builtin nevű névtérben keresi, amit a python az indításkor automatikusan hoz létre és tesz elérhetővé. Ez a névtér tartalmazza a python összes beépített sajátosságát.

A python különféle helyeken történő névkeresésének legutolsó bonyodalmát scope-nak (hatókör) hívják. A scope az, ahol a név érvényes. A python szigorú rendben nézi át a helyeket: helyi, csatolmány, globális és beépített - szigorúan ebben a sorrendben. Ezt hívják LEGB-szabálynak. A globális nevek azok, amik az aktuális névterünkben elérhetőek, a beépítetteket pedig már érintettem. Helyi nevek, amiket helyileg deklarálunk egy kódblokkban, például függvények. A csatolmányok az a névtér ami az aktuális kódblokkot tartalmazza. Egy példa talán világosabbá teheti mindezt.

Scope.py

```
#!/usr/bin/env python3

v = 0
def f1():
   v = 1
   print('v in f1', v)
   def f2():
      print('v in f2', v)
   f2()

print('v', v)
f1()
f2()
```

Output:

```
v 0
v in f1 1
v in f2 1
Traceback (most recent call last):
  File "scope.py", line 15, in <module>
  f2()
NameError: name 'f2' is not defined
```

Ebben a programban a v név inicializáláskor az integer 0 referenciája, van egy f1 függvény definiálás, ami ezután egy belső f2 függvényt definiál. Először a v által hivatkozott érték kimenetre megy és 0-t kapunk, ami az inicializáláskor adtunk neki a globális tartományban. Ezután meghívjuk az f1-et, ami létrehozza a saját v változóját integer 1 értékkel ellátva és kiadja. Ezután jön az f2 definiálása, de annak nincs helyi v változója, ezért a print kifejezés a következő névtérben keres - csatolmány, ami az azt tartalmazó f1 névtere. Itt talál egy v változót, ez az integer 1-re hivatkozik és ez lesz a kimenet, amikor ezután az f2-t meghívják. Ekkor az f1 függvény már lefutott és a vezérlés visszakerül a globális térbe. Az f2 meghívásra kerül, de ez a globális névtérben nincs, így hibajelzést ad ki.

Mindez komplikáltnak tűnhet, de szükséges ahhoz, hogy a kód különböző rétegeiben definiált neveket elkülönítve tarthassuk. Ezt demonstrálta a mathematics.py-ban a három hatványozási függvény meghívása.

Kód importálásakor négy lehetőségünk van:

* import modulnév. Ez az adott nevű modul összes nevét importálja, az elérhetőségükhöz pedig a modulnév előtaggal kell ellátnunk egy pont elválasztóval. Ez a legbiztonságosabb eljárás.

* from modulnév import eljárásnév. Ez lehetővé teszi adott nevű eljárás elérését előtag nélkül, ám a hátránya, hogy előfordulhat névütközés a szülő kódban, ezért kiemelt figyelmet kell szentelni ennek.

* from modulnév import *. Ezzel hozzáférést kapunk a modul összes eljárásához, attribútumához stb, anélkül hogy pont-hivatkozást kellene alkalmaznunk. Ezt a módszert gyakran helytelenítik, mivel nem határozzuk meg az importálandó neveket, fokozva a névütközés veszélyét. Ezt az eljárást alkalmaztam a PyQt modulok importálására, ám mivel ezek a modulok mind nagy Q-val kezdődnek, így a veszély könnyen elkerülhető.

* from modulnév import eljárásnév as m. Amikor a modulnév, vagy az eljárásnév hosszú, vagy nehezen kezelhető, ez használható az olvashatóság segítésére. Az utóbbira példa a decimális modul, ami támogatja a változó pontosságú számítást.

Az utóbbi opcióra példa, a decimális modul támogatja változó pontosságú számítást

Legyen ez:

```
import decimal
2.1 +2.7                        ==> 4.800000000000001

decimal.getcontext().prec = 6

decimal.Decimal(2.1) 3 decimal.Decimal(2.7) ==> Decimal('4.80000)
```

Vagy ezt is tehetjük:

```
from decimal import Decimal as dec
from decimal import getcontext as gc

2.1 + 2.7                       ==> 4.800000000000001
gc().prec = 6

dec(2.1) + dec(2.7)             ==> Decimal('4.80000')
```

Amikor sok hasonlókód van, akkor nagyon meggyorsítja a dolgokat és javítja a kód olvashatóságát.

Még egy dolgot szeretnék elmagyarázni és ez a titokzatos self paraméter, ami az

osztályokban jelenik meg. A fenti mathematics.py-ban csináltam egy új newMath nevű osztályt benne két eljárással: az __init__() és a power(). Az osztályokat nem használjuk közvetlenül, egy új típusú objektumot definiálnak, miképpen egy sztring, vagy integer formájában.

Az új objektumtípus használatához el kell készítenünk egy példányát, miként integert, vagy sztringet szokás.

```
s = “I am a string”       # s is an instance of a string object
x = 2                     # x is an instance of an integer object
n = newMath(x, y)         # n is an instance of the class newMath
```

Az első két példában a Python az idézőjelből és a konstans -ból tudja, hogy milyen objektumot készítsen. A newMath osztály 2 paramétert vár, ezért 2 változót adunk át a definícióban. Az osztály kódjában a 2 eljárásnak tudnia kell, hogy épp melyik példánnyal dolgozik és ezeket automatikusan elhelyezi az egyes eljárások első paraméterébe. Noha mi csak 2 változót adunk át az __init__ eljárásnak, mégis 3-at kap, az osztály objektumot, valamint az x és y változót. Konvenció szerint az első paramétert self-nek hívják. Ennek szellemében a hívással a self.x és a self.y kapják a két változót. A hatványozás-eljárás a self-fel hivatkozott objektumokkal foglalkozik és kapja a helyes x és y értéket.

Itt van egy egyszerűbb példa egy Point nevű osztályt létrehozásáról, ami egy sík pontjának 2 koordinátáját reprezentálja, 0 értéket adva a koordinátáknak. Ezután az osztályból két példányt készít: start_point változó átadása nélkül, így az induló értékként 0, 0 alkalmaz, az end_point (vég) koordinátái 3, 4. Az első print utasítás jelzi, hogy 2 önálló objektumunk van, a második pedig átveszi a pontokat és kinyomtatja a koordinátákat, miközben a harmadik kinyomtatja a Püthagorasz-tétel alapján, a koordináták felhasználásával kiszámolt eredményt (ipython3-t használva).

```
In [1]: class Point:
  ...:     def __init__(self, x=0, y=0):
  ...:         self.x = x
  ...:         self.y = y
  ...:

In [2]: start_point = Point()

In [3]: end_point = Point(3, 4)

In [4]: print(start_point, end_point)
<__main__.Point object at 0x7f3392844f60> <__main__.Point object at
0x7f3392844ef0>

In [5]: print(start_point.x, start_point.y,  end_point.x, end_point.y)
0 0 3 4

In [6]: x_distance = end_point.x - start_point.x

In [7]: y_distance = end_point.y - start_point.y

In [8]: distance = (x_distance ** 2 + y_distance ** 2) ** 0.5

In [9]: print(
     'The distance between start_point and end_point is {}'.format(distance))
```

A start_pont és az end_point közötti távolság 5.0.

Szerkesztő megjegyzése: *az Alkalmi Python cikksorozatban szereplő kódok* ***innen*** *letölthetőek.*