Alkalmi Python - 11. rész

PCLinuxOS Magazine - 2019. december

Írta: Peter Kelly (critter)

Hibakezelés

Minél összetettebb alkalmazásokat készítünk, sajnos annál több hibával találkozunk a kódban, Ahogy a gyakorlatunk és kódolási tudásunk gyarapodik elkerülve az egyszerű „kezdő” hibákat, egyre inkább olyan hibákba ütközünk, amik nehezebb megtalálni és megszüntetni.

Az ilyen hibák kezelését és kiküszöbölését hibakezelésnek (error handling), vagy zsargonban „bug squashing”-nek hívják. A bug kifejezést a számítógéphibára Grace Hopper számítógépes úttörő és USA tengerésztiszt használta először, aki egy molyt talált relébe szorulva. A molyt beragasztotta a számítógép kézzel vezetett naplójába.

A programozási hibák két csoportba sorolhatók:

* **Szintaxis** (syntax) hibák, amiket a kezdő programozók gyakran követnek el. Abból fakadnak, hogy a programozó nem követte a programnyelv mondattanát és a fordító, interpretáló visszautasítja a kódot. Ezek a legkönnyebben megtalálhatók és kezelhetők.

* **Szemantikai** (semantic) hibákat általában nehezebb megtalálni és lehetnek rossz logika szerinti kódok, amik nem megfelelő adatot eredményeznek, vagy eredhetnek a programnyelv eszközeinek helytelen használatából, aminek az eredménye gyakran nem tartalmaz értelmezhetőt. Ami még rosszabb, hogy ezek a hibák általában csak a futtatáskor jelennek meg, például amikor a kód elágazási utasításhoz ér, vagy adatot tesztel.

Ha a kódod szintaxis-hibát tartalmaz, a fordító akkor jelez, amikor futtatni akarod a kódot.

```
>>> for i in range(11):
... print i
File "<stdin>", line 2
print i
^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(i)?
```

A hibajelzés hasznos, mivel megmondja, hogy mi a rossz.

Első látásra a kód jónak tűnik,

```
 >>> x = 2
>>> def square():
... return x * x
...
```

De amikor futtatni akarod ...

```
>>> square(x)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: square() takes 0 positional arguments but 1 was given
```

Jóllehet átadtam a 2 értéket, amire az x változó hivatkozik, de a függvénynek fogalma sincs róla, mivel neki kell a zárójelbe egy paraméter az érték elfogadásához. A hibajelentés ismét tartalmaz hasznos szöveget.

```
>>> def square(x):
... return x * x
... >>> square(x)
4
```

Az interpreter most már boldog és pontosan végrehajtja a kódot.

Ezt könnyű volt megtalálni, de más cikornyásabb hibák miatt kihullhat akár a hajad is. Az egyik leggyakoribb hiba, amikor egy változóban elvárt érték rossz, vagy teljesen hiányzik.

```
>>> for i in range(3):
... if i == 3:
... print('Found it!')
 ...
```

Miért nem nyomtat ki semmit? Nos, a range függvény egy, aa záróértéket nem tartalmazó sorozatot ad, de 0, 1, 2. Tehát a változó sosem egyenlő 3-mal.

```
>>> for i in range(4): # 0, 1, 2, 3
... if i == 3:
... print('Found it!')
...
```

Found it!

Sokkal jobb. Mivel minden programozó tapasztal ilyen típusú hibákat, a Python ad néhány eszközt és eljárást a kezelésükre.

Assert

A Python az assert kifejezést arra szolgáltatja, hogy segítse a feldogozás alatt álló érték ellenőrzését. A kód tesztelésekor fontos tudni, hogy az adatot a függvény megkapja és az eljárások érvényesek. Az érvénytelen értékre példa 0 küldése nevezőbe és az ilyen lehetőségeket ki kell zárni.

Az assert kifejezést kizárólag a tesztelési fázisban történő használatra tervezték, a rendes kódművelet során nem lehet aktív. Az assert vesz egy boolean kifejezést és egy opcionális kifejezést, ami hibajelzésként olvasható. Ha az assert sikertelen, akkor az AssertionError kivételt dobja fel. Amikor egy szigorú tesztelés során assertion error nem jelenik meg, akkor a megadott adatokat érvényesnek lehet tekinteni és az assert kifejezést el lehet távolítani, vagy kikapcsolni, mivel a kimenetükre a végfelhasználónak nem kell megmutatni.

```
>>> def product(*args):
... assert all(args), "0 is not allowed"
... result = 1
... for arg in args:
... result *= arg
... return result .
..
>>> print(product(2, 3, 4))
24
>>> print(product(2, 0, 4))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in product
AssertionError: 0 is not allowed
```

A fenti kódban változó sort ad át a product() függvénynek. A leírót *args-zal csomagolja ki és az assert utasítás ellenőrzi, hogy minden argumentum igaz-e (True). A teszt a Python beépített all() függvényt alkalmazza, ami True (igaz) értéket ad, ha az iterálhatók mind True, ellenkezőleg False (hamis) (a 0-t False-nak értékeli, az összes többi pozitív, vagy negatív True). Ha az ellenőrzés lefutott, folytatódik a program, Ha sikertelen, akkor a futás leáll és egy AssertioError a kimenet, egy általunk meghatározott magyarázó szöveg jelenik meg. A teszt bármi lehet teszésed szerint, ami True-t, vagy False-t értéket adhat.

```
assert 3 > 4, "my test"
```

Ez AssertionError-t ad, mivel a 3 nem nagyobb mint 4 és a kimenet „My test” lesz.

Exceptions (kivételek)

Amikor a program futása során hiba következik be, a Python kivételt jelez. Ha nincs kezelve, ez a kivétel, a program leállását fogja eredményezni annál a pontnál és kiad egy magyarázó üzenetet..

Ebben a példában az int() függvény a „seven” sztringet kapja meg az input függvénytől. Mivel ez nem konvertálható integer-re, egy „ValueError” kivételt emel és a program leáll.

```
>>> n = int(input("Please enter an integer number: "))
Please enter an integer number: seven
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'seven'
```

Ennek elkerüléséhez a kivételt „kezelni” kell és ehhez a Python a try (próba) és except utasítást biztosítja.

Az előző példát ismételve, először megpróbáljuk végrehajtani az int függvényt és folytatjuk a programot ha sikeres. Ha sikertelen, a kivétel utasítás átadja a kivétel típusát és végrehajtja a a követő kódot, bármi legyen is.

```
>>> while True:
... try:
... n = int(input("Please enter an integer number: "))
... break
... except ValueError:
... print("That is not a valid integer! Please try again...")
... Please enter an integer number: seven That is not a valid integer! Please try again
...
```

A többszörös kivétel utasítás alkalmazható különféle hibafeltételek elfogására, de csak egyet hajt végre. Ha a hiba ismeretlen, vagy váratlan, akkor tetszőleges kivétel utasítás alkalmazható. Itt a sys.exc_info függvényt használtam, hogy a kivételről rendelkezésre álló információkat visszaadja. A raise utasítás előhívja a kivételt ismét, a futás leállását okozva. Többszörös kivétel egyetlen kivétel-utasításként is átadható kivételsorozat formájában. Tartalmazhat else utasítást is, ami abban az esetben, ha a próba block rendesen végződik, lefut és nem hajtódik végre, ha kivétel jelenik meg.

```
import sys

try:
f = open('integers.txt')
s = f.readline()
i = int(s.strip())
except IOError as e:
errno, strerror = e.args
print("I/O error({0}): {1}".format(errno, strerror))
except ValueError:
print("No valid integer on line.")
except:
print("Unexpected error:", sys.exc_info()[0])
raise
else:
print("The file contains the integer: ", i)
```

Amennyiben az első kivétel következik be, akkor ezt kapjuk:

I/O error(2): No such file or directory

A valueError kivétel akkor történik, ha a fájlból olvasott sor nem tartalmaz érvényes integert.

No valid integer on line.

A harmadik vizsgálat minden egyéb kivételre vonatkozik. Itt most nincs olvasási jogunk a fájlra. Mivel ez váratlan, a futás valószínűleg meg kell szakítanunk. Mivel a kivételt már kezeltük (a lenti sor kiírásával), ezért ismét kivételt kell emelnünk, de ezúttal nem kezelve és engedélyezve a programnak a futás leállítását.

I/O error(13): Permission denied

Végül, ha az összes vizsgálat lefutott, folytathatjuk a maradék kód végrehajtását.

The file contains the integer: 7

Kivételek emelése

Furcsának tűnhet, hogy kivételt akarjunk emelni, de néha nagyon hasznos lehet. A következő példában a kivételt nem a kód magjában fogtuk meg, hanem a függvényben. A raise (emelés) sokkal tisztább képet ad arról, hogy éppen mi történik.

```
def func():
try:
n = int(input("Please enter an integer number: "))
val = n * 3
return val
except ValueError as e:
print("exception caught in the function func() :-( ", e)
raise
try:
n = func()
except ValueError as e:
print("Oops!", e)
raise print("Got " + str(n))

Please enter an integer number: 7
Got 21
Please enter an integer number: seven
exception caught in the function func() :-( invalid literal for
int() with base 10: 'seven'
Traceback (most recent call last):
File "times_3.py", line 11, in <module>
n = func()
File "times_3.py", line 3, in func
n = int(input("Please enter an integer number: "))
ValueError: invalid literal for int() with base 10: 'seven'
```

A Python rengeteg hasznos beépített kivétellel rendelkezik (a részleteket lásd itt).

Az except finally kipróbálása

A finally utasítás tisztító utasításnak is ismert. Ez a kódrész mindig lefut. Függetlenül attól, hogy kivétel emelése történt-e. Bármikor, ha megnyitottál egy fájlt, a programból kilépés előtt be is kell zárnod. Itt a finally utasítás végzi el ezt nekünk.

```
import sys

try:
f = open('integers.txt')
s = f.readline()
i = int(s.strip())
except IOError as e:
errno, strerror = e.args
print("I/O error({0}): {1}".format(errno, strerror))
```

```
except ValueError:
print("No valid integer on line.")
except:
print("Unexpected error:", sys.exc_info()[0])
raise else:
print("The file contains the integer: ", i)
finally:
if f is not None:
print('Closing the file.')
f.close()
No valid integer on line. # The file contains ‘seven’
Closing the file.
The file contains the integer: 7 # ok
```

A fájl lezárása

Ahogy a gyakorlatod nő, egyre ritkább lesz a szintaxis-hiba, de egyre több második típusú, szemantikus (mondatszerkezeti) hibával szembesülsz.

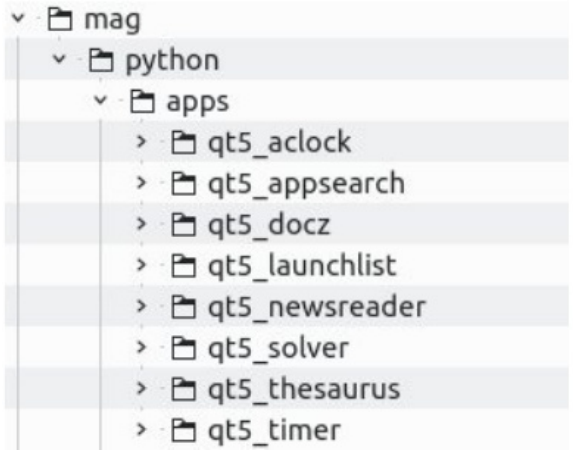
QListWidget, QListWidgetItem

A QT QlistWidget egy kicsit szokatlan, mivel két részből áll:

* magából a ListWidget-ből és a
* ListWidgetItems-ből, ami a ListWidget-et hordozza.

A használata bemutatására egy másik alkalmazásindítót készítek, de ezt a QtDesigner használata nélkül és két Python-fájlt fog tartalmazni: a fő pythonos alkalmazást és egy python-modult a fő fájl rendezetlenségének csökkentésére, könnyebben követhetővé téve a kódot. Valójában már használtuk ezeket a modulokat, de ezek azok a Python-fájlok voltak, amiket a Qt Designer használatával generáltunk, és a PyQt-kód, amit az egyes alkalmazások elején importáltunk. Ahogy azt az előző részben már említettem, semmi misztikum sincs a modulokban, ezek (általában) egyszerű, függvényeket, osztályokat és változókat tartalmazó python-fájlok, amiket a fő kódban történő felhasználásra importálhatunk. Minden, a sorozat kertében írt python-fájl modulnak tekinthető.

A kód egyszerűségének fenntartása érdekében készítek egy indítót a cikk során írt néhány programhoz. Van egy könyvtárszerkezetem, ami így néz ki:(l. következő hasáb teteje)



A qt5_launchlist könyvtár a készítendő alkalmazás számára való. Minden elindítandó alkalmazáshoz kell egy ikon, egy név és egy hely, ahol az alkalmazás található, továbbá egy leírás, ami a listában jelenik meg. Ezek tárolására egy szótárat használok.

A szótárak hasznos eszközök, mivel szerkesztett adatstruktúraként használhatóak, amire itt szükségünk van. A lista minden eleme rendelkezik egy leírással, ami egy attribútumokat, ikont és alkalmazásnevet tartalmazó listához van kapcsolva. Mivel a szótárak elemeket és értékeket tartalmaznak, azokra sztringeket (leírást) használhatunk és az értékekről attribútum-listát. Mindez sima szöveg, ami a fő kódot eléggé zsúfolttá tenné, ezért ezeket modulba teszem és szükség szerint hívom le a részleteket. Rajta, csináljuk meg!

Készíts egy szótárat valahol a rendszeredben a kód számára, lehetőleg ott, ahol a többi, elindítani szándékozott alkalmazás szótárait tárolod. Ha a sorozat előző részeinek alkalmazásait még nem készítetted volna el, akkor letöltheted azokat a magazin weblapjáról innen.

Ezt a modult apps_list.py-nek nevezzük el.

```
#!/usr/bin/env Python3

apps = {'Analogue clock': ['icons/aclock.png',
'qt5_aclock/qt5_aclock.py -nbtl PCLinuxOS -x 3080 -y45'],
'Crossword solver': ['icons/xword.png',
'qt5_solver/solver.py'], 'Rss news reader': ['icons/newsreader.png',
'qt5_newsreader/newsreader.py'],
'Alarm/timer': ['icons/alarm.png', 'qt5_timer/qt_timer.py'],
'Fuzzy application finder': ['icons/appfinder.png',
'qt5_appsearch/qt5_appsearch.py'],
```

'Thesaurus': ['icons/Aiksaurus.png',
'qt5_thesaurus/thesaurus.py'],
'Document finder': ['icons/docz.png',
'qt5_docz/docz.py']}

if __name__ == '__main__':
for item in sorted(apps.items()):
print(item)

A fájlnak egyedül a szótárat kell tartalmaznia, de én hozzáadtam az első és az utolsó három sort, így futtathatom rendes programként. Gyakran alkalmazott eljárás a teszteléshez, de én csak a szótár tartalmát nyomtatom ki.

A fő alkalmazás importálni fogja a szótárat. Íme itt van, összecsukott formában.

```
1   #!/usr/bin/env python3
2
3
4   import sys, os, subprocess
5   from PyQt5.QtGui import QIcon
6   from PyQt5.QtWidgets import QApplication, QDialog, QListWidgetItem, QListWidget
7   from PyQt5.QtCore import QRect
8   import apps_list
9
10
11  '''ensure files are called relative to this directory regardless of
12      where this application was called from'''
13  dir_path = os.path.dirname(os.path.realpath(__file__))
14  os.chdir(dir_path)
15
16  def main():
35
36  def react(item):
46
47  def key_press_event(e):
50
51  def exit_application():
53
54
55  if __name__ == '__main__':
56      main()
57
```

1. sor közli a rendszerrel, hogy Python3-at használjon a szkript fordításához.

4. sor importál néhány modult a szabványos könyvtárból, ahonnan néhány eljárásra van szükségünk a programtörzsben.

5.-7. sorok csak a PyQt-keretrendszer részeit importálják felhasználásra.

8. sor importálja a modulunkat, hogy elérjük az alkalmazás könyvtárát.

11.-14. sorok újak. Az alkalmazás a rendszerben bárhová telepíthető, de tudnunk kell, hogy hol vagyunk és hogyan érhetjük el az elindítandó alkalmazásokat. Ezek a sorok kiderítik, honnan indítottuk a az alkalmazást és átvált arra a könyvtárra. Ezt követően, minden futási hívás ehhez a könyvtárhoz viszonyított.

16. sorral kezdődik a fő függvényt, amit az 55. sor hív meg.

36. sor írja le a main() által meghívott react függvényt, ami reagál az elérhető alkalmazások listáján történő egérkattintásra. A kód maradéka teljesen megegyezik a korábbi alkalmazásokéval, további magyarázat szükségtelen.

A fő függvény

Itt következik az alkalmazás belépési pontja, ezért bontsuk ki:

```
16  def main():
17      app = QApplication(sys.argv)
18      window = QDialog()
19      window.resize(250, 350)
20      windowicon = 'icons/launch.png'
21      window.setWindowIcon(QIcon(windowicon))
22      #window.setStyleSheet("background-color: #1B2224; color: #C5D5D7")
23      applist = QListWidget(window)
24      applist.setGeometry(QRect(10, 10, 230, 330))
25      #applist.setStyleSheet("background-color: #222B2E; color:white; border: 1px solid white")
26      for application in sorted(apps_list.apps):
27          text = application
28          item = QListWidgetItem(text)
29          icon = apps_list.apps.get(application)[0]
30          item.setIcon(QIcon(icon))
31          applist.addItem(item)
32      window.show()
33      applist.itemClicked.connect(react)
34      sys.exit(app.exec_())
35
```

17. sor készít egy példányt a Qapplication-ból, ami elfogad bármilyen argumentumot a sys.ardv()-tól. Ebben az esetben, csak annak az alkalmazásnak egy példányát fogadja, ami végrehajtásáról döntöttünk, vagyis ezt az alkalmazást.

18.-21. sor közli az alkalmazással, hogy:

* Qdialog típusú ablakot használjon (ez egy nagyon egyszerű, kis méretű ablaktípus);

* az ablakot méretezze át 250 px széles és 350 px magasra;

* adjon egy ikont, ami azután az alkalmazás saját ikonja lesz.

22. és 25. sor opcionális. Én a sötét témát szeretem használni és a színbeállítások az én elvárásaimnak felelnek meg,

23. és 24. sor egy QlistWidget-et ad a párbeszéd-ablakhoz és körben 10 pixeles határ hagyását adja meg (a 19. sorhoz képest).

26.-31. sorok:

* a modulunkból importált rendezett szótárak listáján iterál;

* a szótár minden egyes eleméhez veszi az ahhoz tartozó szöveget;

* az ikont index[0]-nak veszi az elem értékéhez;

* a QlistWidgetItems ikont erre az ikonra állítja át;

* Hozzáadja a QlistWidgetItem-et a QlistWidget-hez.

32. sor: minthogy az összes elemet beolvasta és hozzáfűzte a QlistWidget-hez, megjeleníti az ablakot.

33. sor hozzákapcsolódik és végrehajtja a react függvényt, amennyiben valamelyik elemre kattintottak.

A react függvény

```
36  def react(item):
37      selected = item.text()
38      homedir = os.environ['HOME']
39      prefix = homedir + '/mag/python/apps/'
40      selected_app = apps_list.apps[selected][1]
41      if selected_app.find('qt5_aclock.py'):
42          subprocess.Popen(prefix + apps_list.apps[selected][1], shell=True)
43      else:
44          subprocess.Popen(prefix + apps_list.apps[selected][1])
45      exit_application()
```

Ez az, amivel a reagálunk (react) elemen történt kattintásra

36. sor: a szótárelem átkerül a függvényhez és az **item** paraméterben tárolódik.

37. sor: vesszük a kiválasztott szótárelem szövegét, a leírást.

38 sor: az importált rendszermodult felhasználva a felhasználói könyvtárba lép.

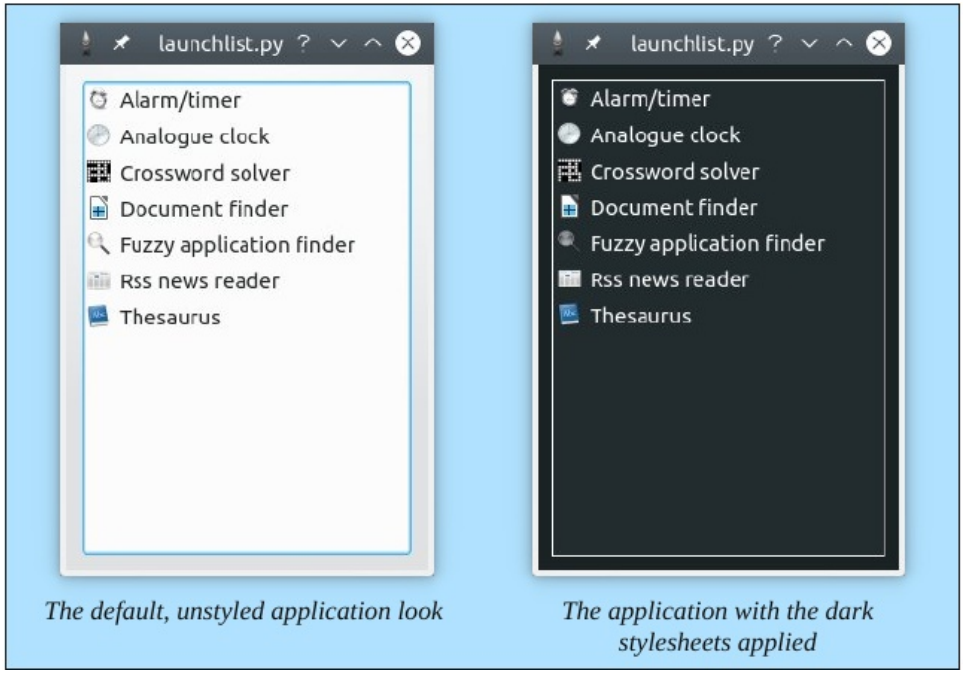
39. sor: létrehoz egy szöveg-sztringet, ami az alkalmazásokat tartalmazó könyvtárra mutat.

40. sor: a kiválasztott alkalmazás nevét a selected_app változóba rakja.

41.-42. sor: amennyiben az analóg óra lett a kiválasztott, ami további paramétereket vár el, akkor a héj (shell) szükséges a parancssor lefordításához, ezért a Popen-eljárásnak kell a „shell=true” jelző, hogy képes legyen az opciókat olvasni és végrehajtani a parancsot.

43.-44. sor átadja valamelyik alkalmazás alkalmazás-sztringjét a Popen-eljárásnak végrehajtásra.

45. sor minden kész, tehát kilép.



The default, unstyled application look

The application with the dark stylesheets applied

A kódot átnézve látom, hogy sok helyen lehetne egyszerűsíteni, fejleszteni, optimalizálni, vagy sokkal profibbra alakítani, de nekem ez így jó. Nem magas minőségű, millió példányban kiadható, szoftveres főnöködet lenyűgöző. Ez egy alkalmi szoftver, alkalmi Pythonban írva. A munka készen van!