<-- vissza

Tartalomjegyzék

Osztályok és kivételek

Osztályok

Az osztályokat alapvet?en egységbezárásra használják. Ami által a program tagolhatósága és újrahasznosítása n?.

Osztályok változókból és metódusokból állnak.

```
class MyClass:
  pass
```

Az el?bbi kód egy alap gy?jt? osztályt reprezentál.

```
class MyClass:
  gvar = 667

  def fv(self):
    pass
```

A **gvar** változó a MyClass osztály szint? azaz az osztály nevével elérhet? nem kell példányosítani. Az **fv** metódus csak példányon, vagy létez? példánnyal hívható meg.

Eddig definíciókat írtunk. Olyan mint a fv definíció, ez még magában nem fut le, csak definiálunk egy típust. Az osztályokat a program futása során példányosítjuk.

```
x = MyClass()
```

```
# hasonloan mint mondjuk egy listát
l = list()
# vagy mas tipusokat ugyanis minden elem a python-on belül osztály lesz
d = dict()
```

Amikor egy osztályt példányosítunk akkor a példányon elérjük az osztály definíciójában definiált változókat és függvényeket.

```
x = MyClass()
x.fv()
MyClass.fv(x) # a két fv. hívás megegyezik
```

Az osztály elemeit nem szükséges a definíció során megadni. Utólag is hozzáadhatjuk.

```
x.count = 1
x.count += 1
# es szinten torolheto is
del(x.count)
```

A python osztályok alapvet?en nem valók absztrakt definiálásra, ebben van az egyszer?sége.

Ha a osztály definíciójában használunk változókat, akkor azokat látják a létrehozott példányok is, de csak addig míg felül nem írják...

öröklés

```
>>> class A(object):
...   pass
...
>>> a = A()
>>> a.t = 5
>>> a.t
5
```

Alapvet?en minden osztály az **object** alosztálya. Az alosztály örökli az ?s minden elemét. Így már létez? m?ködést tudunk az új típus (osztály) - ra tenni. Ezzel újrahasznosítani a már létez? kódot.

Pythonon belül egyszerre több osztálytól is leszármazhatunk.

```
class A(B, C):
  pass
```

Itt az A osztály a B és a C -t?l is leszármazik.

Pythonon belül minden publikus változó, azaz ha én látom az osztály példányát akkor elérek bel?le mindent. Ezért úgymond kitaláltak egy konvenciót, amit illik betartani: Megegyezés szerint.

- **_elem** - fél privát (csak kritikus esetben írjuk direkt)
- **__elem** - privát (soha ne írjuk felül)

isinstance - lehet vizsgálni változóról, hogy milyen példány.

```
>>> isinstance(object, int)
False
>>> isinstance(int, object)
True
>>> isinstance(x, object)
```

```
True
```

issubclass - a leszármazást lehet vele vizsgálni hasonló képen

Több osztályt is megadhatunk leszármazás során

```
class A(B,C):
  pass
```

konstruktor, operátorok

Az **__init__** metódus az osztály konstruktorát reprezentálja. Amikor létrehozunk egy példányt az osztályból ez a függvény hívódik meg.

A **__repr__** - a repr() által visszaadott elemet definiálhatjuk, hasonló képen a **__str__** a str() fv-vel.

```
class Position3D:
  def __init__(self,x,y,z):
    self.pos = (x,y,z)

  def get_y():
    return self.pos[1]

  def __repr__(self):
    return 'This position is in:\n\tx: {0:d}, y: {1:d}, z:{2:d}'.format(self.pos[0],self.pos[1],se
```

További operátorok:

- iterátor lekérése (iter fv.)

__iter__,

- +, -, /, stb. operátorok

__mul__, __add__,

- lista elem lekérés, slice ([], [:])

__getitem__, __setitem__, __getslice__

- attribútum lekérése (.)

__getattr__,

- hívás (())

__call__,

- törlése (del fv.)

__del__,

- hossza (len fv.)

__len__,

- összehasonlító operátorok (==, <, <=, >, >=, !=)

__eq__, __lt__, __le__, __gt__, __ge_, __ne__

- stb

__cmp__,

syntax

super() függvény

a super fv egy típust vár paraméterként és visszaad egy "http://wiki.math.bme.huproxy"http://wiki.math.bme.hu-t amin keresztül elérjük az ?s függvényeit. A proxy-n az operátorok nem használhatók!

super(<type>) - type ?sének közvetít?jét adja vissza super(<type>, <obj>) - type ?sének közvetít?jét adja vissza ha az obj type típusú super(<type1>, <type2>) - az type1 type2 ?sének közvetít?jét adja vissza (többes öröklés esetén)

```
class A(object):
  def __init__(self, x):
    self.hey = x * x

class B(A):
  def __init__(self, x, y):
    self.zed = y / x
    super(B, self).__init__(x)

print B
a = A(5)
print a.hey
b = B(5, 30)
print b.zed, b.hey
```

super függvény leírása

Kivételek

A kivétel olyan nyelvi elem, ami a keletkezésénél megszakítja a program futását, majd amíg nincs lekezelve mozog a program kezdeti blokkjáig, ahol az interpreter elnyeli és kilép a program futtatásából.

Ha a kivétel valahol le van kezelve, és a lekezelés során nem lépünk ki a programból, akkor a program a lekezel? blokk után fogja folytatni a futását.

Kivétel keletkezik ha nullával próbálunk osztani, ha a behúzás a program során nem egységes, ill. ha egy fájlt nem tudunk megnyitni olvasásra...

kivételek kezelése

A kivételekre lehet azért számítani, és ha kell akkor megfelel?en lekezelni a következ? módon:

```
try:
  ...
except <exp> as <v>:
```

```
  ...
else:
  ...
finally:
  ...
```

Kivételek elemei:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to printed directly
...     x, y = inst          # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

A kivételek is osztályok, és azok is leszármaztathatók.

```
>>> class B():
...   pass
...
>>> class C(B):
...   pass
...
>>> class D(C):
...   pass
...
>>> for c in [B,C,D]:
...   try:
...     raise c()
...   except D:
...     print "d"
...   except C:
...     print 'c'
...   except B:
...     print 'b'
...
b
c
d
```

egy magyarázat

with értelmezése

A **with** kulcsszó alapvet?en er?források lefoglalására majd a with blokk elhagyása után azok felszabadítására használható. B?vebb értelemben pedig bármire, amit a blokk el?tt és után mindig le szeretnénk futtatni.

```
with obj as var:
  ...code block
```

Egyenérték? kód:

```
__enter__()
try:
  doSth()
finally:
  __exit__()

# azaz

with <valami> as <valt>:
  doSth()
```

A mélyére ásva a with akkor használható, ha az objektumnak amin értelmezett (obj) definiált az **__enter__** és a **__exit__** operátor. Ugyanis ahogy látszik az egyenérték? kódon ezeket hajtja végre a blokk elején és végén. Az as után álló var pedig az __enter__ visszatérési értéke lesz, amit felhasználhatunk a blokkon belül.

egy magyarázat

iterátorok

iterátorok segítségével tudjuk a sorozatokat bejárni. Az **iter** fv képes visszaadni a sorozat iterátorát amin a **next** fv segítségével lehet lépegetni.

```
>>> t='alma'
>>> i=iter(t)
>>> i
<iterator object at 0xb73c4dec>
>>> i.next()
'a'
>>> i.next()
'l'
>>> i.next()
'm'
>>> i.next()
'a'
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

yield hogyan

Memória barát funkcionális program elem, generátor

generátor

a generátor egy olyan különleges iterátor, aminek az elemei nem foglalnak helyet a memóriában, hanem lépésr?l lépésre "http://wiki.math.bme.hugenerálódik"http://wiki.math.bme.hu. A generátor elemein csak egyszer tudsz végig menni. egy magyarázat

generátor kifejezés

```
(i*i for i in range(1,11))
```

Ez a kód egy generátort eredményez, ami 1 - 10 négyzetét adja vissza elemeiként. b?vebb leírás

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
```

```
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```