<-- vissza

Tartalomjegyzék

függvények definiálása és használata

Amikor függvényeket definiálunk, akkor igazán nem fut le semmilyen létrehozott kod, csak egy ellen?rzés, amivel kizárja az interpreter a szintaktikai hibákat (olyan hiba ami a kod szövegéb?l ered).

A függvényben megírt kód csak a függvény hívásakor fut le és a változói akkor lesznek kiértékelve.

```
def fvdef():
  pass # a pass egy olyan parancs ami nem csinal semmit
  return x**2-1
```

Itt meg nem futott le semmi csak a fuggvény hívásakor.

```
fvdef()
```

Maga a függvényben definiált kód csak a meghívásakor fut le.

Meghívni a függvényt a függvény nevével lehet, úgy hogy utána a zárójelben megadjuk a paraméterként szánt értékeket. (Ha nem kell megadni paramétert akkor csak üres zárójelet használunk... "http://wiki.math.bme.hu()"http://wiki.math.bme.hu)

alapértelmezett paraméter érték

Ha csak simán definiálunk egy függvényt amiket paraméterekkel lehet csak használni, akkor azok a paraméterek kötelezöek

```
def a_d_test(fv, vert):
  return fv * vert

a_d_test(5)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a_d_test() takes exactly 2 arguments (1 given)
```

Ezért lehetséges alapértelmezett értékeket beállítani a definiálás során, ez az érték egyszer értékel?dik ki és utána mindig az lesz az alapértelmezett.

```
def a_d_test(fv, vert = 1):
  return fv * vert

a_d_test(5)
5
```

Fontos hogy azok a paraméterek amiket alapértelmezett értékkel akarunk ellátni a paraméterezésben hátulról kell beilleszteni.

```
SyntaxError: non-default argument follows default argument
```

kulcsszavas paraméter érték

Lehetséges a paraméterek neveivel is átadni a kívánt elemeket:

```
a_d_test(vert=2, fv=2)
```

Itt is fontos, hogy ha valamit kulcsszóval akarunk átadni, akkor azt a függvény hívás paraméterlistájának végét?l kell feltölteni.

tetsz?leges számú paraméter kezelése

Lehet?ség van tetsz?leges számú paramétert kezelni a ***<nev>** paraméter elkérésével

```
def as_many(*args):
  for i in args:
    print i,

as_many('trali', 'fari')
trali fari
```

tetsz?leges számú nevesített paraméterek

```
def as_many_as(**args):
  for k,v in args.iteritems():
    print k, "erteke: ", v

as_many_as(elso='trali', masodik='fari')
```

```
elso erteke:  trali
masodik erteke:  fari
```

Ahogy láthatjuk itt nevet adhatunk az értéknek és nem csak a pozíciója határozza meg hogy mihez is lehet kötni az elemet. Fontos hogy mivel nem számíthatunk hogy megkapjuk-e azt a paramétert a függvény hívásakor ezért mindig ellen?rizzük annak meglétét.

Változók érvényességi köre

A nevek egy változóra hivatkoznak. Egy blokk olyan programrész, ami egy egységként hajtódik végre. A mi esetünkben egy függvény törzse blokk (de egy parancs is blokként értelmezett).

Egy blokk futtatása úgynevezett "http://wiki.math.bme.hufuttatási keretben"http://wiki.math.bme.hu történik (administrációs és futtatási okokból).

A scope a láthatóságát jelenti az adot változónak.

- ha változót deklarálunk egy blokkon belül akkor annak láthatósága arra a blokra korlátozódik
- ha a blokkon belül újabb blokkot hozunk létre, akkor ott is értelmezett lesz a változó, kivéve ha a bels? blokkon belül azt nem definiáljuk felül, azaz nem adunk értéket neki
- továbbá ha a küls? blokkbeli változót szeretnénk használni, majd annak értéket adni a bels? blokkban, az hibát fog okozni, mert ugyan bárhol deklarálhatunk változót, de blokkon belül a deklaráció el?tt nem lehet használni azt a változót.

```
def fv():
  return a

a=7
fv()
```

Látható, hogy a küls? blokkban deklarált "http://wiki.math.bme.hua"http://wiki.math.bme.hu változó látható a függvényen belül is.

```
a=7
def sz2():
  a=2
  return a

print a
print sz2()
print a
```

Latható, hogy ha a bels? blokkon belül deklarálunk már létez? névvel új változót, akkor az csak a bels? blokkon belül lesz értelmezett.

```
a=7
def sz3():
  b=a
  a=6
  return b

Traceback (click to the left of this block for traceback)
...
UnboundLocalError: local variable 'a' referenced before assignment
```

Látszik, hogy "http://wiki.math.bme.hua"http://wiki.math.bme.hu ugyan létezik, de mivel a bels? blokkon belül újra deklaráljuk, igy a deklaráció el?tt nem használhatjuk azt.

A hiba egyszer? de nem szép megoldására a python a **global** kulcsszót adja a kezünkbe.

```
a=7
def sz3():
  global a
  b=a
  a=6
  return b
```

Programtervezési irányelveket követve az el?z? hibát más eszközökkel kell megoldani.

Program struktúra

A python lehet?séget ad bonyolult program szétdarabolására hogy az átláthatóbb legyen a fejleszt?k számára A **csomagok és a modulok** kialakításával érhetjük el a darabolást.

csomag

Kiszervezhet? és akár önmagukban is futtatható nagyobb független kódrészletekre használatos. Egy adott területhez tartozó feladatokat látja el. Általában több modul és alcsomagokat tartalmaz.

Pl: numpy - "http://wiki.math.bme.huNumPy is the fundamental package for scientific computing with Python."http://wiki.math.bme.hu

Mit?l lesz valami csomag? Csomaggá az a könyvtár változhat ami tartalmazza a **__init__.py** fájlt. Ez a fájl jelzi az fordító számára, hogy egy csomagról van szó. A __init__.py üresen is elég, de vannak extra változók amiket lehet használni a csomag könnyebb használata érdekében (__all__, __path__, ...)

modul

Kisebb egybefügg? programrészek egysége. Maga a python programot tartalmazó file egy modult képez.

A modul neve a file nevével egyezik amiben van a kód. Így pl.: a protak.py programfájlban lév? modulra a protak névvel tudunk hivatkozni. A modul nevét a programon belül a **__name__** változóval érjük el. Fontos, hogy ha a modult mint program indítjuk akkor a __name__ értéke __main__ lesz! Ezt fel lehet használni pl.:

tortin.py fájl tartalma:

```
def mainp():
    print "This is the main"

  def elp():
    print 'This is not the main'

  if __name__ == '__main__':
    mainp()
  else:
    elp()
```

Majd hivjuk meg konzolbol a

```
python tortin.py
```

illetve interpreterb?l...

```
import tortin
```

(A modul elnevezést szokás használni még azokra a csomagokra is amiket különálló általános problémákat lehet kezelni pl. numpy és azokra amik kiterjesztési csomagok azaz más nyelvhez (pl.: C) való kapcsolódást valósít meg.)

csomagok és modulok felhasználása (import)

A programrészeket felhasználhatjuk a programon belül akárhol, ha azt jelezzük a fordítónak.

A python-on belül ezt az **import** vagy a **from és az import** kulcsszavakkal tehetjük meg a következ? módon.

```
import sys
  from sys import stdin
  from sys import maxint as mi
  from mymodule import *
```

- az **import sys** paranccsal az összes sys modul alatt elérhet? függvényhez és változóhoz hozzáférünk. pl.: sys.maxint vagy sys.ps1
- az **from sys import stdin** paranccsal a programon használhatjuk a sys modulban definiált stdin változót a saját nevén, azaz a fájlon belül elérhet? a stdin változó.
- a **from sys import maxint as mi** nagyon hasonló az el?z?höz csak itt meg megadtuk azt is hogy a mi modulunkban mi a maxint változót a **mi** néven akarjuk elérni.
- végül a **from mymodule import *** sorral azt adtuk meg, hogy mymodule minden eleme (kivéve a "http://wiki.math.bme.hu_"http://wiki.math.bme.hu karakterrel kezd?d?ek) elérhet?ek legyenek. A * használata csomag import esetén kerülend?, modul esetén is megfelel? figyelemmel!

relatív import útvonalak

Bonyolultabb csomagokban ahol több alcsomag és sok-sok modul található az egymáshoz való hivatkozásokhoz használhatunk relatív útvonalakat.

```
from . import torti
  from .. import kerek
  from ..vaz import orr
```

modul és csomag nevek feloldása

Az fordító el?ször a

1. **beépített modulok** között keres, ha ott nem találja, akkor
2. **sys.path** - ban lév? útvonalakat próbálja végig ami a következ?képpen épül fel:
    1. az indítás könyvtárába
    2. PYTHONPATH

elérhet? elemek

Az elérhet? elemeket egy adott modulon belül a **dir()** függvénnyel listázhatjuk ki

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Kiírás, bevitel

Írásra a

`print`

két formáját, fájlba való íráshoz meg

`write`

használható.

```
>>> print a, 'almafa'
5 almafa
>>> repr(a)
'5'
>>> str(a)
'5'
>>> str(a).ljust(6)
'5     '
>>> str(a).rjust(6)
'     5'
>>> str(a).center(6)
'  5   '
```

a print-nek két formálja van:

Print mostani használata

```
>>> print 'a {0} kisebb mint a {1}'.format(5,6)
a 5 kisebb mint a 6
```

Lehet a format elemén belül címezni.

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
```

Részletek

Print régi használata

```
>>> a = 5*3.7
>>> a
18.5
>>> print '5 * 3.6 egyenlo %5.3f.' % a
5 * 3.6 egyenlo 18.500.
```

Itt a **%** jellel adható meg a paraméter helye, majd az utolsó % jel után vannak felsorolva ismét.

```
>>> print '5 * 3.6 egyenlo %5.3f, %04d.' % (a, a)
5 * 3.6 egyenlo 18.500, 0018.
```

Íras, olvasás fájlok terén

Fájlt megnyitni az **open** paranccsal lehetséges, majd utána a **read** al lehet olvasni bel?le. A read egy nem kötelez? paramétert vár, ami meghatározza, hogy mekkora adatot olvasson be, ha nincs megadva akkor a teljes fájlt próbálja beolvasni, ha az lehetséges. ha a read eléri a fájl végét akkor üres string-el tér vissza.

open parancs

```
open(...)
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file object.  This is the
    preferred way to open a file.  See file.__doc__ for further information.
```

read

```
>>> f = open('/home/me/test.txt','r')
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

```
>>> o = open('d:/temp/test.txt','r')
>>> o.read(10)
'Kiscica ka'
>>> o.read(10)
'landjai\n\nA'
>>> o.read(10)
' kiscica f'
>>> o.read(10)
'utott, eve'
>>> o.readline()
'tt, aludt.\n'
>>> o.seek(0)
>>> o.read()
'Kiscica kalandjai\n\nA kiscica futott, evett, aludt.\n\nItt a vege fuss el vele.'
```

readlines, seek, ciklusok file objektumon

```
>>> o.readlines()
[]
>>> o.seek(0)
>>> o.readlines()
['Kiscica kalandjai\n', '\n', 'A kiscica futott, evett, aludt.\n', '\n', 'Itt a vege fuss el vele.
>>> o.seek(0)
>>> for i in o:
...   print i
...
Kiscica kalandjai



A kiscica futott, evett, aludt.



Itt a vege fuss el vele.
<python>

==== write ====
```

```
f.write(string) - a fájl írására a write parancsot használjuk.

<python>
>>> o = open('D:/temp/test.txt', 'w')
>>> o.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading
>>> o.write('En irtam\n\n')
>>> o.close()
>>> o = open('D:/temp/test.txt', 'r')
>>> o.read()
'En irtam\n\n'
```

Ha objektumot szeretnénk kiírni akkor azt át kell alakítanunk az **str()** vagy **repr()** parancsal

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

o.tell() - visszaadja hol is tartok epp a fájlban, a **o.seek()** az aktuális pozíciót tudjuk változtatni.

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # a hatodik byte-ra ugrunk
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # a vége el?tti harmadik byte-ra ugrunk
>>> f.read(1)
'd'
```

És végül ha már nem akarjuk a fájlt használni, akkor hogy felszabadítsuk az általa foglalt er?forrásokat, le kell zárni. Tudjuk ellen?rizni is hogy a fájl éppen zárva van-e, **f.closed**, a fájl closed állapotának való lekérésével.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Egyébb

with segítségével egyszer? er?forrás kezelés **name** attributum

```
>>>
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.name
'/tmp/workfile'
>>> f.closed
True
```