Tartalomjegyzék

- 1 Bevezetés a Python nyelvbe
  - 1.1 Python kód futtatása
    - 1.1.1 Interaktív módon
    - 1.1.2 Fájlból
    - 1.1.3 Argumantum átadása
  - 1.2 Kódolás stílusa
  - 1.3 Docstring
  - 1.4 A Python, mint számológép
    - 1.4.1 Azonosítók
    - 1.4.2 Objektumhivatkozások
      - 1.4.2.1 Azonossági m?velet
      - 1.4.2.2 Összehasonlító m?veletek
      - 1.4.2.3 Tagsági m?veletek
    - 1.4.3 Egész szer? adattípusok: egész (int, long), logikai (bool)
    - 1.4.4 Beépített lebeg?pontos típusok (float, complex)
    - 1.4.5 Karakterláncok (str)
      - 1.4.5.1 Metódusai
      - 1.4.5.2 Reguláris kifejezések
        - 1.4.5.2.1 A match objektumok
        - 1.4.5.2.2 A regex objektumok
        - 1.4.5.2.3 Példa
    - 1.4.6 Listák
      - 1.4.6.1 Lista létrehozása, listaértelmezés (list comprehension)
      - 1.4.6.2 Lista metódusok
    - 1.4.7 Szótár, hasító tábla (dictionary, hash table)
    - 1.4.8 Operátorok precedenciája
- 2 A procedurális programozás elemei
  - 2.1 Vezérl? utasítások
    - 2.1.1 if
    - 2.1.2 for, break, else, continue
    - 2.1.3 while, braek, else
    - 2.1.4 pass
    - 2.1.5 Kivételkezelés
  - 2.2 Függvények
    - 2.2.1 Opcionálisan megadható argumentumok alapértelmezett értékkel
    - 2.2.2 Függvény tetsz?leges számú argumentummal
    - 2.2.3 Függvényhívás kulcsszavas argumentumokkal
    - 2.2.4 Argumentumlista kicsomagolása
    - 2.2.5 lambda-függvény
    - 2.2.6 Funkcionális programozás elemei
      - 2.2.6.1 Iterátor (bejáró)
      - 2.2.6.2 Generátorkifejezés
      - 2.2.6.3 A map és a filter függvények

- 2.3 Input / Output
  - 2.3.1 Help
  - 2.3.2 Output
  - 2.3.3 Input
  - 2.3.4 Fájlkezelés
- 3 Az objektumorientált programozás alapjai
  - 3.1 Néhány alapfogalom
  - 3.2 Bevezet? példák
  - 3.3 Objektum tulajdonságai
    - 3.3.1 Dekorátor
      - 3.3.1.1 Scope (hatókör), névtér
      - 3.3.1.2 Dekorálunk
- 4 Egyebek
  - 4.1 Modulok
  - 4.2 Csomagok (packages)

Bevezetés a Python nyelvbe

Az el?adáshoz els?számú olvasmány a Python tutorial.

A Pythont ismerjük a Sage-b?l. Különbségek:

- ^ helyett ??, / osztást, // egészosztást jelöl.
- [a..b] helyett range(a,b), vagy xrange(a,b)
- A megszokott, beépített matematikai függvények hiányoznak (find_root, plot, is_prime), de különböz? modulokban sokuk megtalálható.
- Nincsenek szimbolikus változók

A Python egy olyan általános körben használható magas szint? programozási nyelv, aminek az egyik alapelve az olvasható kód írása egy nagyon tiszta szintaxis használatával. 1991-ben alkotta meg Guido Van Rossum.

További jellemz?k

- objektum orientált, procedurális, funkcionális
- sok beépített modul a fejlesztés megkönnyítésére
- dinamikus típus kezelés
- automatikus memóriakezelés
- többféle megvalósítás (CPython, Jython, IronPython, PyPy, Python for S60)
- open-source a f?bb platformokra
- sokkal tömörebb sok más nyelvnél

Nevét a Monthy Python ihlette, nem az állat.

Filozófiája megkapható a 'this' modul betöltésével:

```
import this
```

Python kód futtatása

Interaktív módon

Egyszer?en a python parancs terminálból indítva:

```
$ python
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Sokkal jobb környezetet biztosít az ipython:

```
$ ipython
Python 2.7.5+ (default, Feb 27 2014, 19:37:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Fájlból

- Nyiss egy új file-t pl. a *gedit*-ben, mentsd el "http://wiki.math.bme.hukerdez.py"http://wiki.math.bme.hu néven egy könyvtáradba!
- Írd bele a következ? python kódot (ne használj ékezeteket):

```
s = input("Mondj egy szamot: ")
print "Ennel eggyel kisebbet mondtal: ", s + 1
```

- Mentsd el, és futtasd a scriptedet! (*python kerdez.py*)
- Most kicsit kiegészítjük a scriptet, hogy tartalmazhasson ékezetes bet?ket, és hogy kényelmesebben futtatható legyen (a *python* parancs begépelése nélkül is):

```
#!/usr/bin/python
#coding=UTF-8
s = input("Mondj egy számot: ")
print "Ennél eggyel kisebbet mondtál: ", s + 1
```

- Mentsd el, és adj rá futtatási jogot csak magadnak (`chmod +x kerdez.py`)!
- Futtasd így: `kerdez.py` vagy így: `./kerdez.py`
- A kód második sora lehet ez is:

```
# -*- coding: utf-8 -*-
```

Argumantum átadása

```
#!/usr/bin/env python
#coding=utf-8
# Összead tetsz?leges számú számot
import sys  # modul a rendszerfügg? információkhoz
```

```
osszeg = sum(float(arg) for arg in sys.argv[1:])
print 'Összeg =', osszeg
```

Ha e fájl az osszeg.py nev? fájlba van mentve, akkor

```
$ ./osszeg.py 12 23.5 1
36.5
```

Kódolás stílusa

Stílus (code style) a PEP 8 alapján

- használj mindig 4 space-t minden egyes szinthez, de a folytatósort kezd még beljebb,
- a nagyobb kódrészeket tagold üres sorokkal (függvény, osztály, nagyobb kód blokk)
- használj space-t a vesz? után és a legmagasabb szinten lév? operátorok körül
- használj docstring-et és ahol lehet a megjegyzés a saját sorára vonatkozzon, vagy azonos mértékben behúzva arra a blokkódra
- ahol lehet használj ASCII karakterkódolást
- 79 karakternél ne legyen hosszabb egy sor
- CamelCase elnevezési konvenciót kövesse az osztályok neve és lower_case_with_underscores a függvények és változók nevei

Érdemes megnézni a Google python code style ajánlását is.

Docstring

A hivatkozás nélküli string elemet szokás használni megjegyzések írására és dokumentálásra.

```
"""This is a class of example.

TODO: needs implementation.
"""
```

Els? sort nagybet?vel kezdjük és pontal zárjuk. Egy összefoglaló mondat legyen. Majd egy üres sort hagyva részletesen leírhatunk minden funkciót amit az osztály vagy függvény tartalmaz. Megadhatjuk, hogy a függvény bizonyos hívásaira mi legyen a válasz,

Hasznos linkek:

- hogyan érdemes használni a docstringet
- tesztelés docstring segítségével

```
#coding=utf-8
def lnko(a, b):
    """Kiszámolja két szám legnagyobb közös osztóját.

    >>> lnko(36, 8)
    4
    >>> lnko(-36, 4)
    4
    >>> lnko(3, 0)
    3
    """

    a, b = abs(a), abs(b)
    while a != 0:
```

```
        a, b = b%a, a
    return b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Ha `lnko.py` a neve, akkor egy

```
python lnko.py
```

parancs után nem kapunk semmit (mert a teszt hiba nélkül lefutott), azonban a -v opciót is használva látjuk a részleteket is:

```
$ python lnko.py -v
Trying:
    lnko(36, 8)
Expecting:
    4
ok
Trying:
    lnko(-36, 4)
Expecting:
    4
ok
Trying:
    lnko(3, 0)
Expecting:
    3
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.lnko
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

A Python, mint számológép

Azonosítók

Az adatokat többszöri felhasználásra *azonosítóval* (*névvel*) láthatjuk el.

- a név bet?vel vagy aláhúzással kezd?dhet: [_a-zA-Z]
- a név további karakterei az el?bbieken felül számok is lehetnek: [_a-zA-Z0-9]
- elméletileg bármilyen hosszú lehet a név
- név nem lehet foglalt szó
- nagybet?-kisbet? érzékeny, tehát a val1 név nem azonos a Val1 névvel

Másképp, Backus?Naur formában leírva:

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
```

```
digit      ::=  "0"..."9"
```

A foglalt szavak: `and del from not while as elif global or with assert else if pass yield break except import print class exec in raise continue finally is return def for lambda try` De ne használjuk a Python beépített neveinek, függvényeinek, kivételeinek neveit sem. Ezek megkaphatók a `dir(__builtins__)` paranccsal:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', ......
```

Objektumhivatkozások

Az értékadás (=) esetén valójában **objektumhivatkozás** történik, azaz az egyenl?ség bal oldalán álló névhez egy hivatkozás kapcsolódik, mely az egyenl?ség jobb oldalán álló objektumra mutat. Ez érthet?vé teszi a következ? kódot:

```
>>> a = 1
>>> b = a
>>> a = 2
>>> a
2
>>> b
1
```

A Python **dinamikusan típusos** nyelv, azaz az objektum, amire egy objektumhivatkozás mutat, lecserélhet? egy más típusú objektumra (nem kell a változók típusát deklarálni).

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> a = "b"
>>> type(a)
<type 'str'>
```

Azonossági m?velet

Az alábbi példában a és b még azonos, mert ugyanarra az objektumra hivatkoznak, de a c és d már **nem azonosak**, bár egyenl?k:

```
>>> a = 2
>>> b = 2
>>> a is b
True
>>> c = [1, 2, 3]
>>> d = [1, 2, 3]
>>> c is d
False
>>> c == d
True
```

Összehasonlító m?veletek

Nem csak a számok, de karakterláncok, sorozatok, listák... is összehasonlíthatók (==, !=, <, >, <=, >=):

```
>>> a = 23
>>> b = 56
>>> a == b
False
>>> a != b
True
>>> a <= b
True
>>> c = "abc"
>>> d = "abcd"
>>> c == d[:3]
True
>>> c < d
True
>>> (2, 1, 3) <= (2, 1, 4)
True
```

Tagsági m?veletek

in, not in:

```
>>> l = [1, 2, "cica"]
>>> 2 in l
True
>>> "macska" not in l
True
>>> "c" in l[2]
True
>>> "ci" in l[2]
True
>>> "ci" in l
False
```

Egész szer? adattípusok: egész (int, long), logikai (bool)

int (egész) és hosszú egész (long)

```
>>> a = 12
>>> b = 0b011101
>>> c = 0o3701
>>> d = 0xff0c
>>> a, b, c, d
(12, 29, 1985, 65292)
>>> e = 2222222222222222222222222222222222222222
>>> e
2222222222222222222222222222222222222222L
>>> type(a), type(b), type(e)
(<type 'int'>, <type 'int'>, <type 'long'>)
```

M?veletek egészekkel: +, -, *, / (float eredményt ad), //, % (maradék), **, abs, pow, round, | (OR butenként), ^ (XOR bitenként), & (AND bitenként), <<, >> (eltolás bitenként), ~ (bitenkénti NOT)

Logikai (bool) típus:

Logikai értékek: False (0 vagy 0 hosszúságú), True (nem 0 vagy nem 0 hosszúságú)

Logikai m?veletek: and, or, not

```
>>> True and False
False
>>> 1 and 0
0
>>> 3 and 0
0
>>> 3 or 1
3
>>> 1 or 3
1
>>> 1 or 1 and 0  ### el?bb az and, aztán az or == 1 or (1 and 0)
1
>>> not 67
False
>>> not 0
True
```

Beépített lebeg?pontos típusok (float, complex)

Lebeg?pontos szám megadása:

```
>>> 2.3, -1.2e3, -1.2e-2
(2.3, -1200.0, -0.012)
```

Matematikai függvények:

```
import math
```

után. Ld. [1]

Komplex szám imaginárius része után **j** bet?, de * nincs! Komplex **jellemz?i (attribute)** a real és az imag, **tagfüggvénye (method)** conjugate():

```
>>> z = 3.1 + 2.2j
>>> z
(3.1+2.2j)
>>> z.real
3.1
>>> z.imag
2.2
>>> z.conjugate()
(3.1-2.2j)
```

Karakterláncok (str)

A karakterláncok megadása: "http://wiki.math.bme.hu..."http://wiki.math.bme.hu, '...' vagy "http://wiki.math.bme.hu"http://wiki.math.bme.hu"http://wiki.math.bme.hu..."http://wiki.math.bme.hu"http://wiki.math.bm módon történhet:

```
>>> a="""itt 'ez' meg "az" van"""
>>> a
'itt \'ez\' meg "az" van'
```

```
>>> print a
itt 'ez' meg "az" van
>>> type(a)
<type 'str'>

>>> c = 'aa\nbb'
>>> c
'aa\nbb'
>>> print c
aa
bb

>>> d = r'aa\nbb'  # az r bet? után minden karakter magát jelenti
>>> d
'aa\\nbb'
>>> print d
aa\nbb
```

Véd?kódok (eszkép karakterek, escape characters): \ (folytatás új sorban), \\ (\), \' ('), \"http://wiki.math.bme.hu ("http://wiki.math.bme.hu), \n (új sor), \t (tab). Ha a karakterlánc elé **r** bet?t írunk, a véd?kódok nem érvényesek.

M?veletek karakterláncokkal: indexelés és szeletelés:

```
lánc[sorszám]
lánc[kezdet:vég]
lánc[kezdet:vég:lépés]
```

továbbá az **+** (összef?zés) és a ***** (többszörözés) m?veletek:

```
>>> a = "ho"
>>> b = "rgasz"
>>> 3*a + b
'hohohorgasz'

>>> c = _       # _ az el?z? eredmény
>>> c
'hohohorgasz'
>>> c[:2]+c[6:]
'horgasz'
>>> c[1:7:2]
'ooo'
>>> c[1:6:2]
'ooo'
>>> c[-1::-1]
'zsagrohohoh'
>>> c[-3:4:-1]
'agro'
```

Az indexekre kétféleképp gondolhatunk: 1. a második index már nincs (ennek pl. az az értelme, hogy egy intervallum végét, és a következ? elejét azonos érték jelzi, nem kell 1-et hozzáadni), 2. az indexeket az elemek közé képzeljük, vagyis az elemek határait indexeljük:

```
 +---+---+---+---+---+---+---+
 | h | o | r | g | a | s | z |
 +---+---+---+---+---+---+---+
 0   1   2   3   4   5   6   7
-5  -6  -5  -4  -3  -2  -1
```

A karakterláncok nem változtatható (immutable) objektumok, vagyis a m?veletek, tagfüggvények alkalmazása után új karakterlánc keletkezik:

```
>>> a = "aaaa"
>>> a[1] = b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Metódusai

Metódusok (tagfüggvények, methods): részletesen lásd [2].

```
>>> c = 'this is a STRING'
>>> c.capitalize()
'This is a string'
>>> c
'this is a STRING'
>>> c.upper()
'THIS IS A STRING'
>>> c.lower()
'this is a string'
>>> c.swapcase()
'THIS IS A string'
>>> c.isalpha()    # nem csak az ábécé bet?ib?l áll
False
>>> d = u"? egy KARAKTERLÁNC"   ## Unicode, ékezetes bet?ket is kezel
>>> d
u'\u0151 egy KARAKTERL\xc1NC'
>>> print d.capitalize()
? egy karakterlánc
>>> print d.upper()
? EGY KARAKTERLÁNC
>>> print d.lower()
? egy karakterlánc
```

Néhány egyszer?bb formázási metódus:

```
>>> c.center(20)
'  this is a STRING  '
>>> c.rjust(20)
'    this is a STRING'
>>> c.ljust(20)
'this is a STRING    '
```

Egy példa táblázatos kiírásra:

```
>>> for x in range(1, 6):
...     print repr(x).rjust(2), repr(x*x).rjust(3),   # vessz?vel végz?dik!
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125

>>> print "\n    valami    \n"  # elején, végén ,,white space''
```

```
    valami

>>> "\n  valami  \n".strip()    # fálból való beolvasásnál fontos lehet!
'valami'
>>> "\n  valami  \n".lstrip()
'valami  \n'
>>> "\n  valami  \n".rstrip()
'\n  valami'
>>> c
'this is a STRING'
>>> c.partition("is")           # három részre vágja a karakterláncot
('th', 'is', ' is a STRING')
>>> c.rpartition("is")
('this ', 'is', ' a STRING')
>>> c.replace("is", "IS")       # csere
'thIS IS a STRING'
>>> c.split()                   # szavakra vág
['this', 'is', 'a', 'STRING']
>>> c.split("is")               # adott karakterlánccal vág
['th', ' ', ' a STRING']
>>> print " ==> ".join(['rovar', 'bogár', 'katica'])
rovar ==> bogár ==> katica
```

format metódus (részletes leírás itt.

A karakterláncba tett { } zárójelpárok közt adható meg, hogy mi jelenjen meg és hogyan a format argumentumai közül.

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s"
format_spec       ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill              ::=  <any character>
align             ::=  "<" | ">" | "=" | "^"
sign              ::=  "+" | "-" | " "
width             ::=  integer
precision         ::=  integer
type              ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x"
```

Az el?z? példa másik megoldása:

```
>>> for x in range(1,6):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
```

Az argumentum sorszáma szerint:

```
>>> s = "k{}r{}m"
>>> print s.format('a', 'o'), s.format('o', 'o'), s.format('ö', 'ö')
karom korom köröm
```

```
>>> s = "k{0}r{0}m".format("ö")
>>> print s
köröm
>>> s = "k{1}r{0}m".format("á", 'a')
>>> print s
karám
```

Az argumentum neve, tulajdonsága, indexe szerint:

```
>>> print 'A középpont: ({x}, {y})'.format(x=3, y=5)
A középpont: (3, 5)
>>> print '{0} valós része {0.real}, imaginárius része {0.imag}.'.format(c)
(4-3j) valós része 4.0, imaginárius része -3.0.
>>> t = (1, 2, 3)
>>> 'x={0[0]}, y={0[1]}, z={0[2]}'.format(t)
'x=1, y=2, z=3'
```

Igazítás:

```
>>> '{:<20}'.format('balra')
'balra               '
>>> '{:>20}'.format('jobbra')
'              jobbra'
>>> '{:*^20}'.format(' kozepre ')
'***** kozepre ******'
```

Számok:

```
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> '{:+f}; {:+f}'.format(1.1234567, -1.1234567)
'+1.123457; -1.123457'
>>> '{: f}; {: f}'.format(1.1234567, -1.1234567)
' 1.123457; -1.123457'
>>> '{:f}; {:f}'.format(1.1234567, -1.1234567)
'1.123457; -1.123457'
>>> '{:2.2f}; {:2.2f}'.format(1.1234567, -1.1234567)
'1.12; -1.12'
>>> '{: 2.2f}; {: 2.2f}'.format(1.1234567, -1.1234567)
' 1.12; -1.12'
>>> '{: .2f}; {: .2f}'.format(1.1234567, -1.1234567)
' 1.12; -1.12'
>>> '{: .2%}; {: .2%}'.format(1.1234567, -1.1234567)
' 112.35%; -112.35%'
```

Reguláris kifejezések

Olvasnivalók:

[python.org: Regular expression HOWTO]

[python.org: Regular expressions]

[tutorialspoint.com: Python regexp összefoglaló]

Vannak online tesztoldalak, ahol programozás nélkül lehet próbálgatni a reguláris kifejezéseket. Ezek egyike a [regex101.com].

Példák reguláris kifejezésekre Írjunk olyan reguláris kifejezést, mely illeszkedik az alábbiakban megadott mintára!

Feladat: néggyel osztható 2-jegy? szám

```
[02468][048]|[13579][26]
```

Feladat: szök?év

```
([1-9][0-9])(0[48]|[2468][048]|[13579][26])|(([2468][048]|[13579][26])00)
```

Feladat: nagy bet?kkel írt római számok

Évezred: `M{0,4}`, évszázad: `CM|CD|D?C{0,3}`, évtized: `XC|XL|L?X{0,3}`, év: `IX|IV|V?I{0,3}`. Akkor mégis mi a hiba az alábbi megoldással?

```
M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})
```

Hogy illeszkedik az üres sztringre is! Megoldás

```
\b(?=[MDCLXVI])M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b
```

Feladat: pozitív egészek, esetleg a sokjegy? számok hármasával csoportosítva vannak és a csoportok szóközzel elválasztva

```
[1-9](\d{0,2}( \d{3})+|\d*)
```

Ugyanez szóhatárok közt:

```
\b[1-9](\d{0,2}( \d{3})+|\d*)\b
```

Feladat: HTML-kódban hexadecimális színkód (3 vagy 6 hexa szám)

```
([0-9A-Fa-f]{3}){1,2}
```

Feladat: a yyyy-mm-dd formátum szerinti érvényes dátum az 1600-as évekt?l

```
"""
(
 (1[6-9]\d\d|[2-9]\d{3})          # tetsz?leges év
 -(0[13456789]|1[0-2])            # nem február
 -(0[1-9]|[12]\d|30)              # 1-30
|
 (1[6-9]\d\d|[2-9]\d{3})          # tetsz?leges év
 -02-(0[1-9]|1\d|2[0-8])          # február
|
 (1[6-9]\d\d|[2-9]\d{3})          # tetsz?leges év
 -(0[13578]|1[02])                # 31-napos hónap
 -31                              # 31
|
 (                                ###### szök?év
  (1[6-9]|[2-9]\d)                # els? két jegy
  (0[48]|[2468][048]|[13579][26]) # 4-gyel osztható nem évszázad
 |                                # vagy
  (([2468][048]|[13579][26])00)   # 400-zal osztható
```

```
 )                                  ###### szök?év eddig
 -02-29                             # február 29
)
"""
```

Feladat: '-jelek vagy "http://wiki.math.bme.hu-jelek közé zárt szöveg

```
(['"http://wiki.math.bme.hu])[^\1]*\1
```

A modul betöltése:

```
import re
```

A match objektumok

```
re.match(pattern, string, flags=0)
re.search(pattern, string, flags=0)
```

A match objektum metódusai: group(num=0), groups()

```
>>> line = "gyöngyön elemelem vilmoskörte"
>>>
>>> matchObj = re.match(r'\b(.+)\1\b', line, re.M|re.I)
>>>
>>> if matchObj:
...    print "matchObj.group() : ", matchObj.group()
...    print "matchObj.group(1) : ", matchObj.group(1)
...    print "matchObj.groups() : ", matchObj.groups()
... else:
...    print "No match!!"
...
matchObj.group() :  gyöngyön
matchObj.group(1) :  gyön
matchObj.groups() :  ('gy\xc3\xb6n',)
>>> matchObj = re.search(r'\b(.+)\1\b', line, re.M|re.I)
>>>
>>> if matchObj:
...    print "matchObj.group() : ", matchObj.group()
...    print "matchObj.group(1) : ", matchObj.group(1)
...    print "matchObj.groups() : ", matchObj.groups()
... else:
...    print "No match!!"
...
matchObj.group() :  gyöngyön
matchObj.group(1) :  gyön
matchObj.groups() :  ('gy\xc3\xb6n',)
```

Iterátor match objektumra:

```
>>> iterObj = re.finditer(r'\b(.+)\1\b', line, re.M|re.I)
>>>
>>> for j in iterObj:
...     print j.group()
...
gyöngyön
elemelem
```

Helyettesítés:

```
>>> print re.sub(r'\b(.+)\1\b', r'\1|\1', line, re.M|re.I)
gyön|gyön elem|elem vilmoskörte
>>>
>>> print re.sub(r'\b(.+)\1\b', r'\1|\1', line, 1, re.M|re.I)
gyön|gyön elemelem vilmoskörte
>>> print re.subn(r'\b(.+)\1\b', r'\1|\1', line, re.M|re.I)
('gy\xc3\xb6n|gy\xc3\xb6n elem|elem vilmosk\xc3\xb6rte', 2)
>>> print re.split(r'[,.\s]+', 'szavak, ige,    ez. Meg      az')
['szavak', 'ige', 'ez', 'Meg', 'az']
```

A regex objektumok

A **compile** függvény egy reguláris kifejezésb?l egy regex objektumot hoz létre, amelynek használhatóak a search, match... metódusai. Körülményesebb, mint az el?z?ekben leírt módszerek, de bonyolultabb esetekben, és többször használatos reguláris kifejezésre ezt érdemes használni.

```
compile(pattern, flags=0)
```

Ez a kód

```
prog = re.compile(pattern)
result = prog.match(string)
```

ekvivalens ezzel:

```
result = re.match(pattern, string)
```

A reguláris kifejezés objektum metódusai:

```
search(string[, pos[, endpos]])
```

```
>>> pattern = re.compile("a.")
>>> pattern.search("hagyma")
<_sre.SRE_Match object at 0xaf83d8>
>>> _.group()
'ag'
>>> pattern.search("hagyma", 2)
>>> pattern.search("hagymaleves", 2)
<_sre.SRE_Match object at 0xaf8440>
>>> _.group()
'al'
>>> pattern.search("hagymaleves", 2, 4)
>>>
```

A match m?ködése hasonló, csak az elejére illeszt.

Példa

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Elmehetek messzire, de mondjuk ne nagyon"
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Elmeehetk msiserze, de mduonjk ne ngayon'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Emeelhtek mrszesie, de mduojnk ne naygon'
```

Listák

Ugyanúgy lehet **szeletelni (slice)**, összeadni, többszörözni, mint a karakterláncot, de **a lista változtatható (mutable)**:

```
>>> x = [1, 2, 3, 4]
>>> x[:2]*2 + x[-1:]
[1, 2, 1, 2, 4]
>>> x[:2]*2 + x[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x[0::2]
[1, 3]
>>> x[-1::-1]
[4, 3, 2, 1]
```

Az értékadás (=) csak objektumhivatkozással jár, nem történik adatmásolás. Ennek következtében egy `y = x` parancs után, ahol x lista (vagy valamilyen más gy?jteményes adattípus), az y ugyanarra az objektumra fog mutatni. Így ha megváltoztatjuk x-et vagy y-t, változik a másik is. Az objektumhivatkozások megtörténhetnek egy szinttel mélyebben is, a (`z = x[:]` kód esetén az x elemeire mutató hivatkozások másolódnak, de ekkor sem jön létre a teljes objektumról másolat. Ezt hívjuk **sekély másolásnak (shallow copy)**.

Egy másik példa sekély másolásra:

Mély másolás (deep copy), amikor valóban új példány keletkezik az objektumból:

```
import copy
w = copy.deepcopy(x)
```

Lista létrehozása, listaértelmezés (list comprehension)

Lista létrehozható értékadással, [] az üres lista. A range parancs is listát ad vissza:

```
>>> range(3)
[0, 1, 2]
>>> range(3, 6)
[3, 4, 5]
>>> range(1,10,3)
[1, 4, 7]
```

Rövid listák létrehozásának egyszer? módja a listaértelmezés. Általános alakja:

```
[expression for expr in sequence1
            if condition1
            for expr2 in sequence2
            if condition2
            ...
            for exprN in sequenceN
            if conditionN]
```

Kis programrészek helyettesíthet?k vele. Például ***soroljuk fel az 1890 és 1915 közé es? szök?éveket!***

```
szoko = []
for ev in range(1890, 1922):
    if (ev%4 == 0 and ev%100 != 0) or (ev%400 == 0):
        szoko.append(ev)
print szoko
[1892, 1896, 1904, 1908, 1912, 1916, 1920]
```

Lépésenként egyre összetettebb listaértelmezéssel állítsuk el? ugyanezt:

```
>>> szoko = [ev for ev in range(1890, 1915)]
>>> szoko    # ez még az összes év
[1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1
>>> szoko = [ev for ev in range(1890, 1915) if ev%4 == 0]
>>> szoko    # ez a 4-gyel osztható évek listája
[1892, 1896, 1900, 1904, 1908, 1912]
>>> szoko = [ev for ev in range(1890, 1915)
                if (ev%4 == 0 and ev%100 != 0) or ev%400 == 0]
>>> szoko
[1892, 1896, 1904, 1908, 1912]
```

Egy egyszer? algoritmus egy különböz? elemekb?l álló lista összes permutációja listájának el?állítására:

```
def perm(lista):
    if lista:
        return [[x] + p
                for x in lista
                for p in perm([y for y in lista if y != x])]
    else:
        return [[]]
```

Például:

```
>>> perm([])
[[]]
>>> perm([1, 2, 3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

***** Egy nehezebb feladat: ***Tegyünk fel nyolc vezért egy sakktáblára úgy, hogy semelyik kett? ne üsse egymást! Soroljuk fel az összes ilyen vezérelhelyezést!***

Az ilyen bástyaelhelyezések száma 8! = 40320. A vezérek (királyn?k) azonban átlósan is üthetik egymást. Legyen x egy permutációja a [0, 1, 2, 3, 4, 5, 6, 7] számoknak. Az i-edik sorban a vezér legyen az x[i]-edik oszlopban. Az i-edik és j-edik sorban lév? vezér pontosan akkor üti egymást, ha abs(i - j) != abs(x[i] - x[j]). A permutációk listázásához töltsük be az itertools csomag permutations függvényét, mely a for ciklus minden ciklusában ad egy következ? permutációt, amíg a végére nem ér (így nem kell felsorolni mind a 8! permutációt egyetlen listában). Az all függvény igaz, ha az argumentumában lév? lista minden eleme igaz. Így a megoldás:

```
from itertools import permutations
for x in permutations(range(8)):
    if all([abs(i - j) != abs(x[i] - x[j]) for i in range(8) for j in range(i)]):
        print x
```

Egy logikai feladat

Sokan Einsteinnek tulajdonítják, de erre semmi bizonyíték nincs, s?t, a feladat szövege alapján szinte biztos, hogy nem t?le származik. Angol alapváltozatának neve „Zebra puzzle"http://wiki.math.bme.hu.

Az elterjedt magyar változat a következ?: Van 5 különböz? szín? ház. Minden házban él egy-egy ember, mindegyik más nemzetiség?. Az öt tulajdonos különböz? italt fogyaszt, különféle cigit szív és más-más állatot tart, mindegyikb?l pontosan egyet.

1. A skót a piros házban lakik.
2. A svéd kutyát tart.
3. A dán teát iszik.
4. A zöld ház a fehér ház bal oldalán van.
5. A zöld ház tulajdonosa kávét iszik.
6. Az a személy aki Pall Mall-t szív madarat tart.
7. A sárga ház tulajdonosa Dunhill-t szív.
8. Az az ember aki a középs? házban lakik tejet iszik.
9. A norvég az els? házban lakik.
10. Az ember aki Blend cigit szív amellett lakik aki macskát tart.
11. Az az ember aki lovat tart amellett lakik aki Dunhill cigit szív.
12. A tulaj aki Blue Mastert szív, sört iszik.
13. A német Prince-t szív.
14. A norvég a kék ház mellett lakik.
15. Az ember aki Blend-et szív, a vizet ivó ember szomszédja.

A kérdés: Melyik tart halat és ki iszik vizet?

Írjunk programot, mely megválaszolja e kérdést, gyorsan lefut. Igazoljuk, hogy a magyar változat gyengéje, hogy az utolsó feltétel elhagyható, úgy is csak egy megoldás van. Hogyan változik a megoldás, ha a 4. feltételben kicseréljük a két házat?

Az el?z? perm algoritmust használó megoldás, ami becslésünk szerint több órát fut:

```
def jobbra(h1, h2):
    return h1 - h2 == 1

def szomszed(h1, h2):
    return abs(h1 - h2) == 1

sorrend = perms([1, 2, 3, 4, 5])
print [(skot, sved, dan, nemet, norveg, hal, viz)
      for [piros, zold, feher, sarga, kek] in sorrend
      for [skot, sved, dan, nemet, norveg] in sorrend
      for [kave, tea, tej, sor, viz] in sorrend
      for [PallMall, Dunhill, Blend, BlueMaster, Prince] in sorrend
      for [kutya, madar, macska, lo, hal] in sorrend
      if skot is piros              # 1
      if sved is kutya              # 2
      if dan is tea                 # 3
      if jobbra(zold, feher)        # 4
      if kave is zold               # 5
      if PallMall is madar          # 6
```

```
      if Dunhill is sarga           # 7
      if tej is 3                   # 8
      if norveg is 1                # 9
      if szomszed(Blend, macska)    #10
      if szomszed(Dunhill, lo)      #11
      if BlueMaster is sor          #12
      if nemet is Prince            #13
      if szomszed(norveg, kek)      #14
      if szomszed(Blend, viz)       #15
    ]
```

Javított változat:

```
sorrend = perms([1, 2, 3, 4, 5])
print [(skot, sved, dan, nemet, norveg, hal, viz)
      for [piros, zold, feher, sarga, kek] in sorrend
      if jobbra(zold, feher)        # 4
      for [skot, sved, dan, nemet, norveg] in sorrend
      if skot is piros              # 1
      if norveg is 1                # 9
      if szomszed(norveg, kek)      #14
      for [kave, tea, tej, sor, viz] in sorrend
      if dan is tea                 # 3
      if kave is zold               # 5
      if tej is 3                   # 8
      for [PallMall, Dunhill, Blend, BlueMaster, Prince] in sorrend
      if Dunhill is sarga           # 7
      if BlueMaster is sor          #12
     #if szomszed(Blend, viz)       #15
      for [kutya, madar, macska, lo, hal] in sorrend
      if sved is kutya              # 2
      if PallMall is madar          # 6
      if szomszed(Blend, macska)    #10
      if szomszed(Dunhill, lo)      #11
      if nemet is Prince            #13
    ]
```

Lista metódusok

```
>>> l = [1, 3, 5]
>>> l.append(2)             # egy elem hozzáadása
>>> l.extend([6, 3, 1]); l  # iterálható hozzáadása
[1, 3, 5, 2, 6, 3, 1]
>>> l.count(3)              # hányszor fordul el?
2
>>> l.index(3)              # hanyadik helyen
1
>>> l.index(3, 2)           # a 2 index után hol fordul el? el?ször
5
>>> l.insert(1, 9); l       # beszúr
[1, 9, 3, 5, 2, 6, 3, 1]
>>> a = l.pop(); l; a       # a végér?l eldob egy elemet, és azt visszaadja értékként
[1, 9, 3, 5, 2, 6, 3]
1
>>> a = l.pop(4); l         # a megadott helyr?l eldob
[1, 9, 3, 5, 6, 3]
>>> a
2
>>> l.sort()                # rendez
>>> l
[1, 3, 3, 5, 6, 9]
```

```
>>> l.reverse(); l          # fordítva rendez
[9, 6, 5, 3, 3, 1]
>>> l.remove(3); l          # adott elemet eldob (az els? el?fordulását)
[9, 6, 5, 3, 1]
>>> l.remove(3); l
[9, 6, 5, 1]
>>> a = l.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Szótár, hasító tábla (dictionary, hash table)

A **hash tábla** (hash table, **hasító tábla**) egy olyan adatszerkezet, amely egy hash függvény (hasító függvény) segítségével állapítja meg, hogy melyik kulcshoz milyen érték tartozik. A Python dict adattípusa hash tábla. A hash függvény minden objektumhoz egy véges tartományba es? egész számot rendel. A hash tábla kezelésekor kezelni kell az esetleges ütközéseket, azaz azokat az eseteket, amikor két különböz? kulcshoz azonos hash-érték tartozik. A Python a hash függvény értékkészletébe es? egészeken az identikus leképezés, tehát

```
>>> hash(1)
1
>>> hash(1.0)
1
>>> hash(1123)
1123
>>> hash(-1123)
-1123
>>> hash(111111111111111111111111111)
497554224548382901
```

Így e függvényhez könny? **ütközést** kreálni (a születésnapi paradoxon is besegít), de a gyakorlatban nagyon ritka, és a hash tábla kezelés föl van ilyen esetekre készülve (a kriptográfiai célokra használt hash függvényt úgy kell megkonstruálni, hogy ütközést belátható id?n belül ne lehessen könnyen találni).

```
>>> hash(0.1)
2576882278
>>> hash(2576882278)
2576882278
```

Csak változtathatatlan (immutable) objektumoknak lehet hash értéke:

```
>>> l = [1, 2]     # változtatható (mutable)
>>> c = "hash"     # változtathatatlan (immutable)
>>> hash(c)
7799588877615763652
>>> hash(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Hash tábla, azaz szótár létrehozása:

Operátorok precedenciája

Az operátorok között, mint a matematikában itt is, van precedencia sorrend:

Operator	Description
lambda	Lambda expression
if ? else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, identity test, including membership test
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
`+, -`	Addition and subtraction
*, //, /, %	Multiplication, division, remainder
`+x, -x, ~x`	Positive, negative, bitwise not
**	Exponentiation
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key:datum...}, `expressions...`	Binding or tuple display, list display, dictionary display, string conversion

A sage, python a kifejezéseket balról jobbra értékeli ki, kivéve az értékadásnál, amikor el?bb a jobb oldalt értékeli ki, majd a bal oldalt. Pl. a logikai kifejezés elemeit ha mar felesleges nem értékeli ki.

A procedurális programozás elemei

Vezérl? utasítások

if

```
>>> x = int(raw_input("Adj meg egy egész számot: "))
Adj meg egy egész számot: 42
>>> if x < 0:
...     print "ez negatív"
... elif x == 0:
...     print "ez nulla"
... elif x == 1:
...     print "ez egy"
... else:
...     print "ez sok"
...
ez sok
```

Az elif-ek száma tetsz?leges, és ezzel elkerülhet? a sok behúzás.

A következ? kóddal tesztelhet?, hogy milyen kifejezéseket tekint a python igaznak:

```
x = 5
y = 5.0
if x == y:           # if x is 5:
    # True, 1, 2, 0.00001, [[]], (((),),
    print "igaz, nem 0"
else:
    # False, None, 0, 0.0, 0 + 0j, [], (), (()), {}
    print "hamis, 0, a hossza 0"
```

for, break, else, continue

```
>>> long_words = [u"Antidisestablishmentarianism",
                  u"Donaudampfschiffahrtselektrizitätenhauptbetriebswerkbauunterbeamtengesellschaf
                  u"bejelentkezésszabályozási", u"fosszilisdinoszauruszhányásvilágranglista-megdön
                  u"folyamatellen?rzésiügyosztályvezet?helyettesképesítésvizsgálat-szervezéseitekk
>>> for i in long_words:
...     print i, len(i)
...
Antidisestablishmentarianism 28
Donaudampfschiffahrtselektrizitätenhauptbetriebswerkbauunterbeamtengesellschaft 79
bejelentkezésszabályozási 25
fosszilisdinoszauruszhányásvilágranglista-megdöntés 51
folyamatellen?rzésiügyosztályvezet?helyettesképesítésvizsgálat-szervezéseitekkel 80
```

Döntsük el 10-ig minden egészr?l, hogy prím vagy összetett szám!

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, '=', x, '*', n/x
...             break
...     else:
...         # ha a ciklusban nem talált osztót (nem volt break)
...         print n, 'prím'
...
2 prím
3 prím
4 = 2 * 2
5 prím
6 = 2 * 3
7 prím
8 = 2 * 4
9 = 3 * 3
```

Az **else** a **for**-ral is használható! Szerencsétlen a névhasználat (jobb lenne pl. finally), de a szerkezet praktikus lehet. Az else ágra a ciklus utolsó végrehajtása után kerül a vezérlés (így akkor is, ha a ciklus egyszer sem fut le). Elkerüli viszont az else ágat a vezérlés, ha a break utasítással hagyjuk el a ciklust.

Írjuk ki az 50 alatti páros számokat, de a 3-mal oszthatók helyett *-ot tegyünk!

```
>>> for n in range(2, 50, 2):
...     if n % 3 == 0:
...         print "*",
...         continue
...     print n,
...
```

```
2 4 * 8 10 * 14 16 * 20 22 * 26 28 * 32 34 * 38 40 * 44 46 *
```

while, braek, else

Írjuk ki az 1000 alatti Fibonacci-számokat:

```
>>> n = 1000
>>> a, b = 0, 1
>>> while a < n:
...     print a,  # a vessz? miatt egy sorba kerülnek
...     a, b = b, a + b
...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

A while-nak is lehet else-ága és használható a break.

A pythonban nincs hátul tesztel? ciklusutasítás, a következ?vel helyettesíthet?:

```
while True:
    utasítások
    if kilépési_feltétel:
        break
    (utasítások)
```

pass

Nem csinál semmit, ami sokszor nagyon jól jöhet:

```
while True:
    pass  # Várunk egy billenty?zet interruptot (Ctrl+C)



def func(*args):
    pass   # megírandó
```

Kivételkezelés

Lehet?ségünk van kivételek, hibák kezelésére.

- **try** és **except** blokkok használatával kezelhetjük a hibát adható részeit a kódnak.
- A **try blokkon belül** megpróbáljuk végrehajtani az ott felsorolt utasításokat.
- Ha ez nem sikerül, akkor a felmerül? **hibát az except blokkal "http://wiki.math.bme.hukapjuk el"http://wiki.math.bme.hu.**
- Példa, alább magyarázattal:

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
```

- A while True egy végtelen ciklus, amelyb?l a break utasítással léphetünk ki. A break utasítás csak akkor fut le, ha el?tte a felhasználótól hibátlanul bekértünk egy számot. Ha ez nem sikerült, akkor a try blokkból kilépünk a break utasítás el?tt és ugrunk az except részre. Mivel mindez egy végtelen

while ciklusban van, ezért ezt addig folytatjuk, amíg helyes bemenetet nem kapunk.
- **Tehát megpróbáljuk végrehajtani a try blokkon belüli utasításokat, de amint hibát észlelünk, kilépünk a try blokkból.**
- A hibát az except blokkal kezeljük.
- Az except után megadhatjuk a hiba típusát, így a különböz? hibákat különböz?féleképp kezelhetjük:

```
try:
    z = x/y
except ZeroDivisionError, e:
    z = e # representation: "<exceptions.ZeroDivisionError instance at 0x817426c>"
print z
```

Több különböz? hiba is kezelhet?, különböz? módokon:

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print e
except ValueError:
    print "Could not convert data to an integer."
except:
    raise
```

Az else ág itt akkor fut le, ha nincs kivétel. A következ? program parancssorból indítva megszámolja, hogy egy fájl hány sorból áll:

```
import sys

for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Függvények

Opcionálisan megadható argumentumok alapértelmezett értékkel

```
>>> def fn(a, b, c=1, d=2):
...     print a, b, c, d
...
>>> fn(-1, 0)
-1 0 1 2
>>> fn(-1, 0, 9, 5)
-1 0 9 5
>>> fn(-1, 0, d=5)
-1 0 1 5
```

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok[0] in ('y', 'Y', 'i', 'I'):
```

```
            return True
        if ok[0] in ('n', 'N'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Néhány lehetséges meghívása:

```
ask_ok("Is it OK? ")
ask_ok("May I delete your files? ", 0)
ask_ok("Törölhetem a wincsesztert? ", 1, "Igen vagy nem? ")
```

Függvény tetsz?leges számú argumentummal

```
def fn(*a):
    for i in a:
        print i,
>>> def fn(*a):
...     for i in a:
...         print i,
...
>>> fn(1,2,3,4)
1 2 3 4
```

Kötelez? argumentumok utáni használat:

```
>>> def fn(n, *a):
...     for i in a:
...         print i**n,
...
>>> fn(3, 1, 2, 5)
1 8 125
>>> fn(4, 2, 3, 4, 5)
16 81 256 625
>>> fn(4)
>>>
```

Az alapértelmezett argumentum értéke a függvény **létrehozásakor** lesz kiszámolva, nem a meghívásakor! És csak akkor **egyszer**! Ez a változtatható (mutable) objektumoknál érdekes hatással jár:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(5)
```

A futás eredménye:

```
[1]
[1, 2]
[1, 2, 5]
```

E mellékhatás elkerülésére [] helyett None-t használjunk!

```
def f(a, L=None):
```

```
    if L is None:
        L = []
    L.append(a)
    return L
```

Az argumentumok megadhatók *kulcsszó* = *érték* alakban a kötelez? argumentumok esetén is. Így az argumentumok sorrendje is megváltoztatható,

Függvényhívás kulcsszavas argumentumokkal

```
>>> def bolt(kind, *arguments, **keywords):
...     print "-- Van az Önök boltjában", kind + "?"
...     print "-- Sajnos nincs", kind+".",
...     for arg in arguments:
...         print arg+".",
...     print "\n" + "-" * 40
...     keys = sorted(keywords.keys())
...     for kw in keys:
...         print kw, ":", keywords[kw]
...
>>> bolt('káposzta')
-- Van az Önök boltjában káposzta?
-- Sajnos nincs káposzta.
----------------------------------------
>>> bolt("füstöltsajt", "Nincs",
...      "Rég elfogyott",
...      boltos='Kovács József',
...      vevo="Szabó József")
-- Van az Önök boltjában füstöltsajt?
-- Sajnos nincs füstöltsajt. Nincs. Rég elfogyott.
----------------------------------------
boltos : Kovács József
vevo : Szabó József
```

Argumentumlista kicsomagolása

Egy listában vagy sorban lév? argumentumok kicsomagolása (az el?z? példákban a * a függvény definíciójában szerepelt, itt a híváskor, és listából vagy sorból argumentumlistát készít):

```
>>> l = [1, 9, 2]
>>> range(*l)
[1, 3, 5, 7]
```

lambda-függvény

Rövid, névtelen függvények konstrukciójára:

```
>>> map(len, ["emberek", "halak", "tyukok"])
[7, 5, 6]
>>> map(lambda x: x**2 + 1, [1, 2, 4])
[2, 5, 17]
```

Funkcionális programozás elemei

Iterátor (bejáró)

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> print it
<...iterator object at ...>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

```
for i in iter(obj):
    print i

for i in obj:
    print i
```

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

Generátorkifejezés

Hasonló a listaértelmezéshez.

```
(expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3
             if condition3 ...
             for exprN in sequenceN
             if conditionN )
```

```
obj_total = sum(obj.count for obj in list_all_objects())
```

A generátor egy iterátort ad vissza mely adatok folyamát adja vissza.

```
def generate_ints(N):
    for i in range(N):
        yield i
```

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration

def counter (maximum):
    i = 0
    while i < maximum:
        print "el?tte", i
        val = (yield i)
        print "utána", i
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1

>>> x = counter(8)
>>> x
<generator object counter at 0x12f50f0>
>>> x.next()
el?tte 0
0
>>> x.next()
utána 0
el?tte 1
1
>>> x.next()
utána 1
el?tte 2
2
>>> x.send(6)
utána 2
el?tte 6
6
>>> x.next()
utána 6
el?tte 7
7
>>> x.next()
utána 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A map és a filter függvények

A funkcionális programozás alapelemei a függvényeket objektumokra alkalmazó map és filter függvény. A Pythonban gyakran egyszer? listaértelmezéssel kiválthatók. A map az els? argumentumában lév? függvényt alkalmazza a második (és további) bejárható argumentuma elemeire és listát ad vissza.

```
>>> def upper(s):
...     return s.upper()
>>>
>>> map(upper, ['sentence', 'fragment'])     # listából lista
['SENTENCE', 'FRAGMENT']
>>> map(upper, ('sentence', 'fragment'))     # tuplából is lista lesz
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']

>>> def is_even(x):
...     return (x % 2) == 0
>>> filter(is_even, range(10))
[0, 2, 4, 6, 8]
>>> [x for x in range(10) if is_even(x)]
[0, 2, 4, 6, 8]
```

Input / Output

Help

Interaktív python-ban meghívva a **help()** függvényt egy segítség menübe jutunk. Próbáljátok ide beírni a következõket:

- keywords
- if
- random

Az iPythonban (a sage-hez hasonlóan) egy objektumot követ? pont után nyomott tab-ra kiadja a lehetséges metódusok listáját.

Output

Output már szerepelt a **print** kulcsszóval. Ez ugye bármilyen beépített típust képes volt kiírni, de ha formázni szeretnénk a kiírást akkor több dolog kell.

- **str()** - string-et csinál más típusú objektumból, **print valtozo** és **print str(valtozo)** ekvivalens
- **repr()** - nagyon hasonló a **str()**-hez, az interaktív módban való megjelenítés függvénye, néha mást (pl. kicsit többet) mutat a str-nél, próbáljuk ki: **print str(2.5656454534543654)** és **print repr(2.5656454534543654)**
- A stringek **format()** metódusát a karakterláncoknál vizsgáltuk

Input

- **input()** - Kiírja a neki adott string-et, és vár egy bementet, a visszatérési értéke a bemenet automatikusan értelmezve, pl:

```
m = input("Magassag centimeterben:")
print "Magassag meterben:", m / 100.0
```

- A következõ kód viszont hibát fog okozni:

```
n = input("Neved:")
print "Hello", n
```

- Ez azért van, mert az input azonnal értelmezi a beadott adatot, és a string-eket nem tudja felismerni, õ ekkor változónévre gondol.

- **raw_input()** - Kiírja a neki adott string-et, és vár egy bementet, a visszatérési értéke a bemenet stringként, pl:

```
n = raw_input("Neved:")
print "Hello", n
```

Fájlkezelés

- **open(file_neve, megnyitasai_mod)** - megnyit egy fájlt, elsõ paramétere a neve, második a megnyitási mód: 'w', 'a', 'r' a szokásosak. Pl:

```
f = open('test.txt', 'w')
```

- **write()** - metódusa a file objektumoknak (az elõzõ példában **f** file objektum), a neki adott string-et a file-ba írja. Pl:

```
f.write("Which witch watches which witch's watches?\nKovetkezo sor\n")
```

- **read()** - metódusa a file objektumoknak, beolvassa az egész file-t egy (potenciálisan) jó nagy string-be. Pl:

```
f = open('test.txt', 'r')
s = f.read()
print s
```

- **readline()** - metódusa a file objektumoknak, beolvas egy sort a file-ból, az újsor jelet is beolvassa, az üres sorokat '\n'-ként olvassa be. Ha a file végére ért üres sztringet olvas be. Pl:

```
line = f.readline()
print line
```

- **readlines()** - metódusa a file objektumoknak, beolvassa az összes sort mint listát. Pl:

```
lines = f.readlines()
for line in lines:
    print line
```

- Megjegyzés: a file objektumok iterálhatóak, azaz az elõzõ példa ekvivalens ezzel:

```
for line in f:
    print line
```

- **close()** - metódusa a file objektumoknak, bezárja a file-t. Pl:

```
f.close()
```

A fentiek lényegesen egyszer?síthet?k a **with** paranccsal:

```
with open("filename") as f:
    data = f.read()
    # a data kezelése ide
```

Ez a végén automatikusan le is zárja a megnyitott fájlt.

Az objektumorientált programozás alapjai

[[3] Python tutorial], [[4] Egy jó wikibook oldal]

[[5] Python library az operátorokról]

Néhány alapfogalom

Egységbezárás (encapsulation)
: a procedurális megközelítéssel ellentétben az adatok és a függvények a program nem különálló részét képezik, azok összetartoznak: együtt alkotnak egy objektumot. Az objektum felülete(ke)n (interfész, interface) keresztül érhet? el a többi objektum számára.

Osztály (class)
: egy objektum prototípusa, tulajdonságokkal ellátva, melyek jellemzik az osztály példányait. A tulajdonságok osztály és példányváltozók, és metódusok (tagfüggvények). Pont-jelöléssel érhet?k el.

Attribútum/tulajdonság (attribute)
: egy objektum tulajdonsága, jellemz?je, az objektum nevét követ? pont után írjuk a nevét.

Példány (instance)
: egy osztályhoz tartozó egyedi objektum.

Példányosítás (instantiation)
: egy osztály egy példányának létrehozása.

Példány változó (instance variable)
: metóduson belül definiált változó, mely csak az osztály adott példányához tartozik.

Osztályváltozó (class variable)
: változó, melyet az osztály minden példánya elér.

Metódus (method, tagfüggvény)
: osztályon belül definiált spec. függvény, els? argumentuma tipikusan a self.

Örökl?dés (inheritance)
: származtatással egy már meglév? osztályból egy újat hozunk létre. Ez rendelkezni fog az ?s minden tulajdonságával, amihez továbbiakat ad(hat)unk.

Polimorfizmus/többalakúság (polymorphism)
: különböz? viselkedésmódokkal ruházzuk fel az egymásból származtatott objektumokat.

Névtér (namespace)

megadja, hogy egy név melyik objektumhoz tartozik (leképezés a nevekr?l az objektumokra). (Példák névterekre: foglalt nevek, globális változók, egy függvény lokális nevei).

Hatókör/szkóp (scope)
: a kód azon része, ahol a névtér közvetlenül (a teljes elérési út magadása nélkül) elérhet?.

Bevezet? példák

```
>>> class Osztalyom:
...     "Egy egyszer? példa osztályra"
...     i = 123
...     def f(self, i):
...         self.i = i
...         return 'hello vilag'
...
>>> x = Osztalyom()
>>> x.i
123
>>> x.f(5)
'hello vilag'
>>> x.i
5
>>> Osztalyom.i
123
```

Objektum **kívülr?l** ráaggatott attribútumokkal (ritkán van erre szükség):

```
>>> x.szamlalo = 1
>>> while x.szamlalo < 10:
...     x.szamlalo = x.szamlalo * 2
...
>>> print x.szamlalo
16
>>> del x.szamlalo
```

Konstruktor `__init__`

```
class Komplex:

    def __init__(self, real, imag):
        self.r = real
        self.i = imag

    def konjugalt(self):
        return Komplex(self.r, -self.i)
```

Használatára létrejön egy példány, melynek memóriacímét kapjuk vissza nevének beírása után:

```
>>> z = Komplex(3, 4)
>>> z
<__main__.Komplex instance at 0x184a710>
>>> w = z.konjugalt()
>>> w
<__main__.Komplex instance at 0x184a758>
```

Tulajdonságaik lekérdezhet?k:

```
>>> z.r, z.i
(3, 4)
```

```
>>> w.r, w.i
(3, -4)
```

További tulajdonságokkal ruházzuk fel a Komplex osztály objektumait. Ehhez a speciális metódusokat használjuk:

```
class Komplex:
    """Komplex számok kezelése."""

    def __init__(self, real, imag):
        """Komplex szám létrehozása"""
        self.r = real
        self.i = imag

    def konjugalt(self):
        return Komplex(self.r, -self.i)

    def __add__(self, z):
        """Az infix '+' m?velet használatához."""
        return Komplex(self.r + z.r, self.i + z.i)

    def __repr__(self):
        """Komplex szám megjelenése interaktív módban.
        Ugyanezt kapjuk a print parancsra is."""

        # ez egyel?re rossz, mert a 3 - 4i képe ez lesz: 3 + -4i
        return repr(self.r) + " + " + repr(self.i) + "i"
```

Alkalmazása:

```
>>> z1 = Komplex(3, 4)
>>> z2 = Komplex(1, 5)
>>> z1 + z2
4 + 9i
>>> z1
3 + 4i
>>> z2
1 + 5i
>>> z1 + z2
4 + 9i

>>> type(z1)
<type 'instance'>
>>> isinstance(z1, Komplex)
True
>>> z1.__class__
<class __main__.Komplex at 0x21ce3f8>
>>> z1.__module__
'__main__'
>>> z1.__class__.__name__
'Komplex'
>>> dir(z1)
['__add__', '__doc__', '__init__', '__module__', '__repr__', 'i', 'konjugalt', 'r']
>>> print z1.__doc__
Komplex számok kezelése.
>>> z1.konjugalt()
3 + -4i
>>> z1.__add__(z2)
4 + 9i
```

Egy példa az osztályváltozó alkalmazására:

```
class Alkalmazott:
    """Alkalmazottak osztálya"""
    alkSzamlalo = 0

    def __init__(self, nev, fizetes):
        """Konstruktor az alkalmazottaknak"""

        self.nev = nev
        self.fizetes = fizetes
        Alkalmazott.alkSzamlalo += 1
        self.sorszam = Alkalmazott.alkSzamlalo

    def szamlalot_kiir(self):
        mit = "{0} a {1}. alkalmazott az összesen {2} között"
        print mit.format(self.nev, self.sorszam, Alkalmazott.alkSzamlalo)

    def adatot_kiir(self):
        print "Név: ", self.nev,  ", Fizetés: ", self.fizetes

>>> a1 = Alkalmazott("Jóska", 1000000)
>>> a2 = Alkalmazott("Pista", 200000)
>>> a3 = Alkalmazott("Ildikó", 2000000)
>>> a2.szamlalot_kiir()
Pista a 2. alkalmazott az összesen 3 között
>>> a2.adatot_kiir()
Név:  Pista , Fizetés:  200000
>>> a3.adatot_kiir()
Név:  Ildikó , Fizetés:  2000000
```

Példa az öröklésre. Szül? osztály:

```
class Ellipszis:

    def __init__(self, kozeppont, felx, fely):
        self.kozeppont = kozeppont
        self.felx = felx
        self.fely = fely

    def terulete(self):
        return self.felx*self.fely*3.14159
```

Gyerek osztály:

```
class Kor(Ellipszis):

    def __init__(self, kozeppont, sugar):
        Ellipszis.__init__(self, kozeppont, sugar, sugar)
        self.sugar = sugar

>>> e = Ellipszis((0, 0), 3, 2)
>>> e.felx
3
>>> e.fely
2
>>> e.terulete()
18.849539999999998
>>> k = Kor((2, 0), 3)
>>> k.felx
3
>>> k.fely
3
>>> k.sugar
```

```
3
>>> k.terulete()
28.27431
```

Függvény felülírása:

```
class Kor(Ellipszis):

    def __init__(self, kozeppont, sugar):
        Ellipszis.__init__(self, kozeppont, sugar, sugar)
        self.sugar = sugar

    def terulete(self):
        return self.sugar**2*3.14159
```

Objektum tulajdonságai

```
# Google példa: http://google-styleguide.googlecode.com/svn/trunk/pyguide.html#Python_Language_Rul

import math

class Square(object):
    """A square with two properties: a writable area and a read-only perimeter.

    To use:
    >>> sq = Square(3)
    >>> sq.area
    9
    >>> sq.perimeter
    12
    >>> sq.area = 16
    >>> sq.side
    4
    >>> sq.perimeter
    16
    """

    def __init__(self, side):
        self.side = side

    def __get_area(self):
        """Calculates the 'area' property."""
        return self.side ** 2

    def ___get_area(self):
        """Indirect accessor for 'area' property."""
        return self.__get_area()

    def __set_area(self, area):
        """Sets the 'area' property."""
        self.side = math.sqrt(area)

    def ___set_area(self, area):
        """Indirect setter for 'area' property."""
        self.__set_area(area)

    area = property(___get_area, ___set_area,
                    doc="""Gets or sets the area of the square.""")

    @property
    def perimeter(self):
        return self.side * 4
```

Konvenció:

- _single_leading_underscore: bels? használatra (Pl. `from M import *` nem tölti be ezeket)
- single_trailing_underscore_: kerüljük a konfliktust a kulcsszavakkal (class_)
- __double_leading_underscore: "http://wiki.math.bme.huname mangling"http://wiki.math.bme.hu (a class FooBar osztálybeli __boo kint majd _FooBar__boo -ként érhet? el).
- __double_leading_and_trailing_underscore__: "http://wiki.math.bme.hukülönleges"http://wiki.math.bme.hu objektumok (pl. __init__), saját magunk ilyeneket ne vezessünk be.

Dekorátor

Jó leírások: [[6]], [[7]]

A **dekorátorok** módosítják függvények vagy osztályok kódját!

Scope (hatókör), névtér

Minden új függvény fölépíti saját névterét, egy szótár formájában. Íme a globális és egy lokális névtér:

```
>>> x = 2
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'x': 2, '__doc__': Non
>>> def fn():
...     y = 3
...     print locals()
...
>>> fn()
{'y': 3}
```

Ha egy név lokálisan nem lett létrehozva, a tartalmazó névterekben keresi egyre kijjebb haladva (itt az x nem lett lokálisan létrehozva, de elérhet?):

```
>>> def fn():
...     y = 3
...     print "x, y:", x, y
...     print locals()
...
>>> fn()
x, y: 2 3
{'y': 3}
```

Kérdés: Mi történik az alábbi kód hatására? magyarázzuk meg, mi történik:

```
>>> def fn():
...     y = 3
...     print "x, y:", x, y
...     x = 5
...     print locals()
...
>>> fn()
```

És mi történik, ha a 3. és 4. sort fölcseréljük?

A névtér törl?dik a függvény lefutása után, magasabb szinten az alacsonyabb szint? névterek nem érhet?k el:

```
>>> def fn():
```

```
...     z = 1
...
>>> print z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

A **függvény lezárása** a Pythonnak azt a képességét jelenti, hogy a nem globális névtérben definiált függvény **emlékszik** a definiálás pillanatában érvényes bennfoglaló névterekre:

```
>>> def kulso(x):
...     def belso():
...         print x
...     return belso
...
>>> f1 = kulso(3)
>>> f2 = kulso(5)
>>>
>>> f1()
3
>>> f2()
5
```

Dekorálunk

A küls? függvény az argumentumában átadott függvény eredményét duplázza, és ezt a duplázó függvényt adja vissza. Ezzel **dekorálunk** két különböz? függvényt!

```
>>> def kulso(fn):
...     def belso(*args):
...         print "belül vagyunk"
...         return 2*fn(*args)
...     return belso
...
>>> def fv(n):
...     return n
...
>>> dekoralt = kulso(fv)
>>> dekoralt(5)
belül vagyunk
10
>>> def fadd(a, b):
...     return a + b
...
>>> dekoralt = kulso(fadd)
>>> dekoralt(4, 5)
belül vagyunk
18
```

A `fv = kulso(fv)` rövidítése a `@kulso` a fv függvénydefiníció elé írásával:

```
>>> @kulso
... def fmul(a, b):
...     return a * b
...
>>> fmul(3, 5)
belül vagyunk
30
```

Dekorátor lehet olyan osztály is, mely **meghívható**, azaz amelyben létezik `__call__` függvény:

```
>>> class Dekorator():
...     def __init__(self, f):
...         print "Dekorátor konstruktor"
...         print "meghívjuk a fv-t:", f()
...     def __call__(self):
...         print "Dekorátor meghívása"
...
>>> @Dekorator
... def fn():
...     print "az fn meghívása"
...
Dekorátor konstruktor
meghívjuk a fv-t: az fn meghívása
None
>>> fn()
Dekorátor meghívása
```

Egyebek

Modulok

[[8]]

A modul egy egyszer? Python-kódú fájl definíciókkal és végrehajtható utasításokkal a Python interpreter számára elérhet? könyvtárban. Neve a fájl .py el?tti része lesz, azaz ez kerül a globális __name__ változóba. Legyen az alábbi kód a fibi.py fájlba elmentve.

```
#!/usr/bin/python
#coding=utf8

def fib(n):
    """Visszaadja az n-edik Fibonacci-számot."""
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a

def fibs(n):
    """Kiírja n-ig a Fibonacci-számokat."""
    a, b = 0, 1
    print 0,
    while b <= n:
        print b,
        a, b = b, a+b

if __name__ == "__main__":
    import sys
    print fib(int(sys.argv[1]))
```

Ekkor e modul a következ?képp tölthet? be:

```
>>> import fibi
>>> fibi.fibs(13)
0 1 1 2 3 5 8 13
>>> fibi.fibs(0)
0
>>> fibi.fib(0)
```

```
0
>>> fibi.fib(10)
55
```

A modul tartalma elérhet?, pl:

```
>>> fibi.__name__
'fibi'
>>> print fibi.fib.__doc__
Visszaadja az n-edik Fibonacci-számot.
>>> dir(fibi)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'fib', 'fibs']
```

Egyéb betöltési lehet?ségek:

```
from fibi import fib
from fibi import *
import fibi as f
```

Újratöltés, ha közben átírtuk:

```
reload(fibi)
```

A __name__ fibi, ha betöltjük, de __main__, ha parancssorból futtatjuk, ez használhatóvá teszi parancssorból is:

```
$ python fibi.py 18
2584
```

Csomagok (packages)

Modulok kollekciója, egy egyszer? könyvtárstruktúra nevei definiálják a alcsomagstruktúrát:

```
csomag/
    __init__.py
    elsoalcsomag/
        __init__.py
        elsomodul.py
        masodikmodul.py
    masodikalcsomag/
        __init__.py
        elsomodul.py
        masodikmodul.py
```

Az __init__.py fájlok az adott csomag betöltésekor lefutnak -- lehetnek üres fájlok is! Betöltésük:

```
import csomag
from csomag import elsoalcsomag
from csomag.elsoalcsomag import elsomodul
```