Változótípusok és számolás pythonban

**Ha valaki az igazságot és a törvényszerűséget keresi, nem tehet különbséget kicsiny és nagy problémák között.
Aki a kicsiny dolgokban nem veszi komolyan az igazságot, abban az emberben nagy dolgokkal kapcsolatban sem bízhatunk meg.**
Albert Einstein

Ebben a Notebookban bemutatjuk a python nyelv szintaxisát és néhány alapvető adat típust és adatstruktúrát. Illetőleg megnézzük hogyan lehet a pythonban egyszerű matematikai műveleteket elvégezni.

Adattípusok

Komment (megjegyzés)

A Pythonban a megjegyzések kettős-kereszttel ('#') kezdődnek és a sor végéig tartanak. Egy megjegyzés lehet sor elején, vagy követhet szóközt, tabulátor-karaktert, de ha egy karakterlánc (string) belsejébe teszed, az nem lesz megjegyzés (lásd a példában!). A kettős-kereszt karakter egy karakterláncon belül csak egy kettős-keresztet jelent.

In [1]:

```
# ez az első megjegyzés
SPAM = 1                 # ez a második megjegyzés
                         # ... és ez a harmadik.
STRING = "# Ez nem megjegyzés, mert idézőjelekben van."
```

A comment csak a programozónak szól emlékeztetőül, vagy más programozók megértését segíti. Noha gyakran elmarad, a kódok rendszeres dokumentálása megjegyzésekkel, később (amikor már a kód részleteit a feledés homály borítja) nagyon kifizetődő lehet.

Logikai Bool-változók és feltételes végrehajtás

Ahhoz hogy a program reagálni tudjon bemenetekre, vagy egy már kiszámolt részeredménytől függhessen további működése, szükség van feltételek vizsgálatára. Gondoljunk például a másodfokú egyenlet megoldóképletére! Attól függően, hogy a diszkrimináns pozitív vagy negatív léteznek ill. nem léteznek valós gyökök, így értelemszerűen egy ilyen programnak "döntenie" kell. A döntés, egy feltétel vizsgálatán alapul (pl. poz. vagy neg. diszkrimináns), ami lehet igaz, vagy hamis, angolul *True* ill. *False*. Az igaz és hamis értéket tárolni képes változó típusa: **boolean** (Bool-változó). (Emlékezzünk vissza az **int** ill. **float** típusokra).

Két objektum azonosságáról dupla egyenlőségjellel, azaz a == operátor segítségével tájékozódhatunk.

In [2]:

```
1 == 1  # 1 megeggyezik 1-el ?
```

Out[2]:

True

In [3]:

```
1 == 2 # 1 megeggyezik 2-vel ?
```

Out[3]:

False

In [4]:

```
1 == '1'
```

Out[4]:

False

Ha arról akarunk meggyőződni hogy két objektum nem ugyan az akkor ezt a != operátor segítségével tehetjük meg:

In [5]:

```
1!=2 #1 nem 2 ?
```

Out[5]:

True

In [7]:

```
1 != 1
```

Out[7]:

False

Természetszerűen számok esetén a <,> operátorok segítségével dönthetjük el hogy melyik szám a nagyobb:

In [8]:

```
1>2
```

Out[8]:

False

In [9]:

```
1<2
```

Out[9]:

True

In [10]:

```
1 < '1'
```

```
---------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-10-9ee3bb6f2cf3> in <module>()
----> 1 1 < '1'

TypeError: unorderable types: int() < str()
```

Egy igaz/hamis értéket változóban is tárolhatunk:

Karakterláncok (Stringek)

A számok mellett a Python karakterláncokkal is tud műveleteket végezni. A karakterláncokat egyszeres ('...') vagy dupla idézőjelek ("...") közé lehet zárni. A két jelölés között nincs jelentős különbség. A \ használható arra, hogy a karakterláncbeli idézőjeleket levédjük:

In [11]:

```
'spam eggs'
```

Out[11]:

```
'spam eggs'
```

In [14]:

```
'doesn\'t'
```

Out[14]:

```
"doesn't"
```

In [15]:

```
"doesn't"
```

Out[15]:

```
"doesn't"
```

In [20]:

```
"\"Yes,\" he said."
```

Out[20]:

```
'"Yes," he said.'
```

A kimeneti karakterlánc idézőjelekben jelenik meg, és a speciális karakterek vissza-perrel () levédve. Bár ez néha különbözőnek látszik a bemenettől (az idézőjel fajtája megváltozhat), a két karakterlánc egyenértékű. A print() függvény egy sokkal olvashatóbb kimenetet eredményez, elhagyva az idézőjeleket és kiírva a levédett és speciális karaktereket:

In [22]:

```
'"Isn\'t," she said.'
print('"Isn\'t," she said.')
```

```
"Isn't," she said.
```

In [26]:

```
s = 'First line.\nSecond line.'  # \n újsort jelent
s  # print() nélkül a \n benne van a kimenetben
print(s)  # print()-tel az \n újsort hoz létre

print('C:\some\name')  # \n újsort jelent
print(r'C:\some\name')  # r van az idézőjel előtt --> Meggátolja a \. értelmezését
```

```
First line.
Second line.
C:\some
ame
C:\some\name
```

Karakterláncokat a + művelettel ragaszthatunk össze és *-gal ismételhetünk.

In [27]:

```
3 * 'un' + 'ium'
```

Out[27]:

```
'unununium'
```

Két egymást követő literális karakterláncot (azokat, amik idézőjelben vannak, és nem egy változóban, vagy nem egy függvény hoz létre) az értelmező magától összefűz:

In [28]:

```
'Py' 'thon'
```

Out[28]:

```
'Python'
```

In [29]:

```
prefix = 'Py'
prefix + 'thon'  # csak így tud változót és literális karakterláncot összefűzni
```

Out[29]:

```
'Python'
```

A karakterláncokat indexelhetjük, az első karakterhez tartozik a 0 index, a következőhöz az 1-es index és így tovább. Nincs külön karakter típus; egy karakter egyszerűen egy egy hosszúságú karakterlánc:

In [30]:

```
szo = 'Python'
szo[0]  # karakter a 0 pozícióban
```

Out[30]:

```
'P'
```

Az indexek negatívak is lehetnek, ilyenkor jobbról kezdünk el számolni:

In [31]:

```
szo[-2]  # utolsó előtti karakter
```

Out[31]:

```
'o'
```

Az indexelésen felül a szeletelés is támogatott. Míg az indexelés egyetlen karaktert jelöl ki, a szeletelés egy rész-karakterláncot:

In [33]:

```
szo[2:5]  # karakterek a 2 pozíciótól (benne van) a 5-esig (az már nem)
```

Out[33]:

```
'tho'
```

In [36]:

```
szo[:2]  # karakterek az elejétől a 2-esig (már nincs benne)
```

Out[36]:

```
'Py'
```

In [37]:

```
szo[42]  # a szo csak 6 karakteres
```

```
---------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-37-f7e2d342191a> in <module>()
----> 1 szo[42]  # a szo csak 6 karakteres

IndexError: string index out of range
```

A beépített len() függvény a karakterlánc hosszával tér vissza:

In [38]:

```
s = 'legeslegelkáposztásíthatatlanságoskodásaitokért'
len(s)
```

Out[38]:

```
47
```

Listák (List)

A Python többfajta összetett adattípust ismer, amellyel több különböző értéket csoportosíthatunk. A legsokoldalúbb a lista, amelyet vesszőkkel elválasztott értékekként írhatunk be szögletes zárójelbe zárva. A lista elemeinek nem kell azonos típusúaknak lenniük, bár gyakran minden elem azonos típusú.

In [39]:

```
a = ['spam', 'tojások', 100, 1234]
a
['spam', 'tojások', 100, 1234]
```

Out[39]:

```
['spam', 'tojások', 100, 1234]
```

In [40]:

```
a[3]
```

Out[40]:

```
1234
```

In [43]:

```
a[2] = a[2] + 23
a
```

Out[43]:

```
['spamspamspamspamspamspamspamspamspamspamspamspamspamspamspamspamspamspams
pamspamspamspamspamspam',
 'tojások',
 123,
 1234]
```

A listák egymásba ágyazhatóak, azaz listába elhelyezhetünk listát elemként:

In [47]:

```
a = ['a', 'b', 'c']
nn = [1, 2, 3]
x = [a, nn]
x
```

Out[47]:

```
[['a', 'b', 'c'], [1, 2, 3]]
```

Bool változók és más változók kapcsolata

Ahogy fenntebb láttuk egy objektumot egész számmá (**int()** függvény) vagy lebegőpontos számmá tudunk konvertálni (**float()** függvény). Objektumokat Bool-változóvá is tudunk konvertálni. Erre szolgál a **bool()** függvény. Alább néhány példát nézzünk meg hogy bizonyos dolgok hogy konvertálódnak a **bool()** függvénnyel.

egy szám vagy egy karakterlánc **True** értéket ad:

In [53]:

```
print(bool(1.0))
print(bool('szoveg'))
```

```
True
True
```

Egy látszólag rosszul működő példa:

In [55]:

```
'b' == ('a' or 'b')
```

Out[55]:

```
False
```

In [56]:

```
'a' == ('a' and 'b')
```

Out[56]:

```
False
```

In [57]:

```
'b' == ('a' and 'b')
```

Out[57]:

```
True
```

In [60]:

```
'a' and 'b'
```

Out[60]:

```
'b'
```

A második példa vajon miért **False** ha az első **True** ? A harmadik miért **False** ha a látszólag hasonló negyedik **True**? Ezen a ponton úgy tűnhet hogy az **and** és **or** utasítások nem működnek megfelelően! Ha részletesen kiböngésszük hogy mit is csinál a python értelmező akkor kiderül hogy pontosan azt teszi amire megkértük.. DE ez nem pontosan az amire elsőnek gondolnánk! Az első két példa tehát nem azt ellenőrzi hogy a **==** előtt álló karakter az szerepel-e a zárójelben! Mi is történik pontosan? Amikor a python egy **or** kifejezéssel találkozik (a zárójeleken belül) sorba megy az **or** kifejezés által kapcsolatba hozott elemeken és az első **True** - ként kiértékelhető objektum **értékét** adja vissza, thát nem **True** vagy **False**-t hanem a változó értékét. Ez azért van így mert ha egy **or** kifejezés egyik tagja igaz akkor az egész kifejezés igaz. Az **and** ezzel szemben addig lépdel végig az alkotó kifejezéseken amíg meggyőződött róla hogy mindegyik kifejezés igaznak számít és a legutolsó értéket adja vissza. A bool operációk ezen viselkedését *rövid zár*-nak hívják és sok programozási nyelvben hasonlóan működik. Tehát ha arról akarunk meggyőződni hogy egy kifejezés valamely kifejezés csoport egyikével megegyezik akkor az alábbiak szerint szükséges eljárni:

In [65]:

```
print((('a' == 'a') or ('a' == 'b')))
print((('b' == 'a') or ('b' == 'b')))
```

```
True
True
```

Előfordul azonban sokszor hogy az a csoport amihez egy lehetséges elemet össze kell hasonlítani igen népes. Például hogy mondjuk meg hogy egy karakterlánc szerepel egy adott hosszú listában? Erre alkalmazható az **in** kulcsszó:

In [66]:

```
Tage_der_Woche=['Montag','Dienstag','Mittwoch','Donnerstag','Freitag','Samstag','Son
ntag']
print('hetfo' in Tage_der_Woche)
print('Montag' in Tage_der_Woche)
```

```
False
True
```

Számok

A Pythonban, mint a programozási nyelvekben többféle számtípust tudunk megkülönböztetni. A különbségtétel oka, hogy miként tudja a program a memóriában tárolni a számokat illetve a CPU (a számítógép számolási feladatait ellátó részegység) miként képes kezelni őket.

A Pythonban megkülönböztetjük többek között az egész (integer) számokat:

In [67]:

```
A=2
print(type(A))
print(A)
```

```
<class 'int'>
2
```

Vannak lebegőpontos számok (tizedes ponttal ellátott **véges hosszú** számok):

In [68]:

```
B=3.25
print(type(B))
print(B)
```

```
<class 'float'>
3.25
```

Python mint számológép

A notebookot lehet úgy használni akárcsak egy sima számológép: be lehet írni egy kifejezést, és az kiszámolja az értékét. A kifejezések nyelvtana a szokásos: a +, -, * és / műveletek ugyanúgy működnek, mint a legtöbb nyelvben (például Pascal vagy C); zárójeleket (()) használhatunk a csoportosításra. Például:

In [69]:

```
2 + 2
```

Out[69]:

```
4
```

In [70]:

```
50 - 5*6
```

Out[70]:

```
20
```

Az egész számok (pl. 2, 4, 20) alkotják az **int** típust, azok, a valós számok (pl 5.0, 1.6) pedig a **float** típust. A python 3-ban az osztás (/) mindig lebegőpontos értékeket ad vissza. Lefelé kerekítő egészosztás elvégzéséhez, amellyel egész értéket kapunk a // operátor használható; az osztás maradékához a %:

In [71]:

```
(50 - 5*6) / 4
```

Out[71]:

```
5.0
```

In [72]:

```
8 / 5  # az osztás egész számok esetén is lebegőpontos eredményt ad.
```

Out[72]:

```
1.6
```

In [73]:

```
17 / 3  # hagyományos osztás lebegőpontos eredménnyel
```

Out[73]:

```
5.666666666666667
```

In [74]:

```
17 // 3  # az egészosztással megszabadulunk a törtrésztől
```

Out[74]:

5

In [75]:

```
17 % 3  # a % operátor az osztás maradékával tér vissza
```

Out[75]:

2

In [76]:

```
5 * 3 + 2  # eredmény * osztó + maradék
```

Out[76]:

17

In [77]:

```
5 ** 2  # 5 négyzete
```

Out[77]:

25

In [78]:

```
2 ** 7  # 2 7-dik hatványa
```

Out[78]:

128

Biztos jól számolunk? A lebegőpontos számok véges hosszúak, és a számítógép csak adott hosszú részt vesz figyelembe!

In [79]:

```
C=0.1 - 0.1000000000000000055511151231257827021181583404541015625   #  Biztos jól sz
ámolunk?
print(C)
```

0.0

In [80]:

```
D=(1/29999999999999999)*30000000000000000
print(D-1)
```

0.0

Az itt látható hiba nagyon gyakori hiba a numerikus programokban. Nagyon figyeljünk a számábrázolási hibára, hiszen ez az apró eltérés, hibás működéshez vezethet (amit nem mindig jelez a rendszer hibaüzenettel!):