BEVEZETÉS A PYTHON PROGRAMOZÁSBA
(Informatikai szakközépiskola – 10. évfolyam)

1. Alapfogalmak

1.1 Mintaprogram - Krumpli

Feladat – Krumpli
Egy cserkésztáborban készül a vacsora, de a konyhafőnök nem tudja, mennyi krumplit vegyen. A fejadagot ismeri az 0,4 kg, de a szükséges mennyiséget számítógép segítségével szeretné kiszámolni az aktuális létszámtól függően. Írjuk meg a programot a konyhafőnöknek! A program kérje be a létszámot, majd írja ki a szükséges mennyiséget a következőképpen:
Létszám? 20
A szükséges mennyiség: 20 * 0.4 kg = 8.0 kg

```
# -*- coding: ISO-8859-2 -*-
letszamstr = raw_input("Létszám? ")
letszam = int(letszamstr)
mennyiseg = letszam * 0.4
print "A szükséges mennyiség: ",letszam," * 0,4 kg = ",mennyiseg," kg"
```

Az első sor a magyar ékezetes karakterek helyes megjelenítéséhez szükséges kódtábla elérését biztosítja.
A második sorban a konzolról beolvasunk egy sztringet a *letszamstr* változóba, miközben a konzolra kiírjuk a "Létszám? " szöveget.
A harmadik sorban a *letszamstr* szövegből egész típust készítünk, és belerakjuk a *letszam* változóba.
A negyedik sorban a létszámot megszorozzuk a fejadaggal, és ezt a *mennyiseg* nevű változóban tároljuk.
Az ötödik sorban kiíratjuk a konzolra a szükséges szöveget. A fixen kiírandó szövegeket " " jelek között, és hozzájuk vesszővel fűzzük hozzá a változó mennyiségeket.

1.2 Ékezetes és speciális karakterek

A 2.3 verziótól kezdve a magyar nyelvet használóknak ajánlatos minden Python scriptjük elejére beírni a következő pszeudo-commentek egyikét (kötelezően az első vagy a második sorba) :
 # -*- coding: ISO-8859-2 -*-
vagy :
 # -*- coding:Utf-8 -*-

Ezek a pszeudo-commentek azt jelzik a Pythonnak, hogy a scriptben : vagy a fő nyugat-európai nyelvek ékezetes karakterkészletét használjuk egy byteon kódolva az ISO-8859 norma szerint; vagy a Unicodenak nevezett két byteos kódolást használjuk.

A Python mindkét rendszert tudja használni, de meg kell neki adnunk, hogy melyiket használjuk. Ha az operációs rendszerünk úgy van konfigurálva, hogy a billentyűleütések Utf-8 kódokat generálnak, akkor konfiguráljuk úgy a szövegszerkesztőnket, hogy az is ezt a kódot használja és tegyük a fönt megadott második pszeudo-commentet minden scriptünk elejére. Ha az operációs rendszerünk a régi norma (ISO-8859) szerint működik, akkor inkább az elsö pszeudo-commentet kell használnunk.

Ha semmit sem adunk meg, akkor időnként figyelmeztető üzeneteket fogunk kapni az interpretertől és esetleg némi nehézségeket fogunk tapasztalni amikor az IDE környezetben szerkesztjük scriptjeinket (speciálisan Windows alatt).

Függetlenül attól, hogy egyik, vagy másik normát, vagy egyiket sem használjuk, a scriptünk korrekt módon fog végrehajtódni. Ahhoz, hogy a saját rendszerünkön tudjuk a kódot szerkeszteni, a megfelelő opciót kell választani.

1.3 Adatok és változók

Egy számítógépprogram lényegét az adatokkal végzett műveletek jelentik. Ezek az adatok nagyon különbözőek lehetnek (lényegében minden, ami digitalizálható), de a számítógép memóriájában végleg bináris számok véges sorozatává egyszerűsödnek.

Ahhoz, hogy a program (bármilyen nyelven is legyen megírva) hozzá tudjon férni az adatokhoz, nagyszámú különböző típusú változót használ.

Egy programozási nyelvben egy változó majdnem mindegy milyen változónévként jelenik meg, a számítógép számára egy memóriacímet jelölő hivatkozásról van szó, vagyis egy meghatározott helyről a RAM-ban.

Ezen a helyen egy jól meghatározott érték van tárolva. Ez az igazi adat, ami bináris számok sorozata formájában van tárolva, de ami az alkalmazott programozási nyelv szemében nem feltétlenül egy szám. Szinte bármilyen « objektum » lehet, ami elhelyezhető a számítógép memóriájában, mint például: egy egész, egy valós szám, egy vektor, egy karakterlánc, egy táblázat, egy függvény, stb.

A programozási nyelv különböző változótípusokat (egész, valós, karakterlánc, lista, stb.) használ a különböző lehetséges tartalmak egymástól történő megkülönböztetésére.

1.4 Változónevek és foglalt szavak

A változóneveket mi választjuk meg meglehetősen szabadon. Ennek ellenére törekednünk kell, hogy jól válasszuk meg őket. Előnyben kell részesíteni az elég rövid neveket, amik világosan kifejezik, hogy mit tartalmaz az illető változó. Például: az olyan változónevek, mint magasság, magas, vagy mag jobbak a magasság kifejezésére, mint x.

Egy jó programozónak ügyelni kell arra, hogy az utasításssorait könnyű legyen olvasni.

Ezen kívül a Pythonban a változóneveknek néhány egyszerű szabálynak kell eleget tenni :

A változónév az (a - z , A - Z) betűk és a (0 - 9) számok sorozata, aminek mindig betűvel kell kezdődni.

Csak az ékezetnélküli betűk a megengedettek. A szóközök, a speciális karakterek, mint pl.: $, #, @, stb. nem használhatók. Kivétel a _ (aláhúzás).

A kis- és nagybetűk különbözőnek számítanak.

Figyelem : Jozsef, jozsef, JOZSEF különböző változók. Ügyeljünk erre !

Váljon szokásunkká, hogy a változóneveket kisbetűkkel írjuk (a kezdő betűt is) . Egy egyszerű konvencióról van szó, de ezt széles körben betartják. Nagybetűket csak magának a szónak a belsejében használjunk, hogy fokozzuk az olvashatóságot, mint például: tartalomJegyzék.

További szabály: nem használható változónévként az alább felsorolt 28 «foglalt szó » (ezeket a Python használja) :

and	**assert**	**break**	**class**	**continue**
def	**del**	**elif**	**else**	**except**
exec	**finally**	**for**	**from**	**global**
if	**import**	**in**	**is**	**lambda**
not	**or**	**pass**	**print**	**raise**
return	**try**	**while**	**yield**	

1.5 Hozzárendelés vagy értékadás

A változóhoz való « érték hozzárendelés » vagy « értékadás » kifejezések egyenértékűek. Egy olyan műveletet jelölnek, amely kapcsolatot teremt a változónév és a változó értéke (tartalma) között.

A Pythonban számos más nyelvhez hasonlóan a hozzárendelés műveletét az egyenlőségjel reprezentálja1 :

```
n = 7                       # n-nek a 7-et adjuk értékül
msg = "Mi újság ?"          # A "Mi újság ?" értéket adjuk msg-nek
pi = 3.14159                # pi nevű változóhoz hozzárendeljük az értékét
```

A fenti három értékadó utasítás mindegyikének a hatása több művelet végrehajtása a számítógép memóriájában: :

- változónév létrehozása és bejegyzése a memóriába ;
- változónévhez jól meghatározott típus egy speciális érték létrehozása és tárolása a memóriában ;
- kapcsolat létesítése a változónév és a megfelelő érték memóriahelye között (mutatók belső rendszere segítségével)

1.6 Változó értékének a kiírása

A fenti gyakorlat következményeként három változónk van: *n*, *msg*, *pi*. Programban mindig a **print** utasítást használjuk az érték kiíratásához:

print *msg*

A print utasítás szigorúan csak a változó értékét írja ki, úgy ahogyan az kódolva van.

1.7 Operátorok és kifejezések

Az értékeket és a rájuk hivatkozó változókat operátorok segítségével kifejezésekké kombináljuk.

```
a, b = 7.3, 12
y = 3*a + b/5
```

Ebben a példában az *a* és *b* változókhoz először hozzárendeljük a 7.3 és 12 értékeket. A Python automatikusan a « valós » típust rendeli az *a* és az « egész » típust a *b* változóhoz. A példa második sorában az új *y* változóhoz rendeljük hozzá egy kifejezés eredményét, ami a * , + és / operátorokat kombinálja az *a*, *b*, 3 és 5 operandusokkal. Az operátorok speciális szimbólumok, amiket olyan egyszerű matematikai műveletek reprezentálására használunk, mint az összeadás vagy a szorzás. Az operandusok az értékek, amiket az operátorok segítségével kombinálunk.

A Python minden egyes kifejezést kiértékel, amit beírunk neki, legyen az akármilyen bonyolult és ennek a kiértékelésnek az eredménye mindig egy érték. Ehhez az értékhez automatikusan hozzárendel egy típust, ami függ attól, hogy mi van a kifejezésben. A fenti példában *y* valós típusú lesz, mert a kiértékelt kifejezés legalább egy valós változót tartalmaz. Nemcsak a négy matematikai alapművelet operátora tartozik a Python operátoraihoz. Hozzájuk kell venni a hatványozás operátorát **, néhény logikai operátort, a karakterláncokon működő operátorokat, az azonosságot és tartalmazást tesztelő operátorokat, stb. Minderről a későbbiekben lesz szó.

Rendelkezésünkre áll a modulo operátor, amit a % szimbólum jelöl. Ez az operátor egy számnak egy másik számmal való egészosztásából származó maradékát adja meg. A későbbiekben nagyon hasznos lesz, amikor azt vizsgáljuk, hogy egy *a* szám osztható-e egy *b* számmal. Elég azt megnézni, hogy az *a* % *b* eredménye egyenlő e nullával.

1.8 A műveletek prioritása

Ha egy kifejezésben egynél több operátor van, akkor a műveletek végrehajtásának sorrendje a prioritási szabályoktól függ. A Pythonban a prioritási szabályok ugyanazok, mint amiket matematikából ismerünk:

zárójelek. Ezek a legmagasabb prioritásúak. Azt teszik lehetővé, hogy az általunk kívánt sorrendben történjen egy kifejezés kiértékelése.
Így 2*(3-1) = 4 , és (1+1)**(5-2) = 8.

hatványozás. A hatványok kiszámolására a többi művelet előtt kerül sor.
Így 2**1+1 = 3 (és nem 4), és 3*1**10 = 3 (és nem 59049 !).

szorzás és osztás, azonos prioritással. Ezeket az összeadás és kivonás előtt értékeli ki.
Így 2*3-1 = 5 (és nem 4), és 2/3-1 = -1 (Alapértelmezésben a Python egészosztást végez.)

Ha két operátor azonos prioritású, akkor a végrehajtásuk balról jobbra történik.
Így az 59*100/60 kifejezésben először a szorzást, majd ezt követően , az 5900/60 osztást végzi el, aminek az eredményet 98. Ha először az osztást végezné el, akkor az eredmény 59 lenne (emlékezzünk rá, hogy itt egy egészosztásról lenne szó).

1.9 Kompozíció

Eddig egy programozási nyelv különböző elemeit tanulmányoztuk : a változókat, a kifejezéseket és az utasításokat, de azt nem vizsgáltuk, hogy hogyan lehet ezeket egymással kombinálni.

Egy magasszintű programozási nyelv egyik erőssége, hogy különböző elemek kombinálásával összetett utasításokat hozhatunk létre. Így például, ha tudjuk hogyan adunk össze két számot és hogyan íratunk ki egy értéket, akkor a két utasítást egyetlen utasítássá kombinálhatjuk :

```
print 17 + 3
```

Ez a nyilvánvalónak tűnő tulajdonság fogja lehetővé tenni az összetett algoritmusok világos és tömör programozását.

```
h, m, s = 15, 27, 34
print "az éjfél óta eltelt másodpercek száma = ", h*3600 + m*60 + s
```

Figyelem : van egy megszorítás arra nézve, hogy miket lehet kombinálni :
Amit egy kifejezésben az egyenlőségjel baloldalán helyezünk el, annak mindig egy változónak kell lenni, nem pedig egy kifejezésnek. Ez abból a tényből fakad, hogy az egyenlőségjelnek nem ugyanaz a jelentése, mint a matematikában. Így például az m + 1 = b utasítás szabálytalan.

A matematikában viszont elfogadhatatlan *a* = *a* + 1 -et írni, pedig ez az írásmód igen gyakori a programozásban. Az *a* = *a* + 1 utasítás az a változó értékének eggyel történő megnövelését jelenti.

Feladat – Henger
Kérjük be a konzolról egy henger sugarát és magasságát cm-ben, majd
- *írjuk ki a henger térfogatát!*
- *Írjuk ki a henger súlyát, ha ez tömör vashenger, és ha fahenger!*

A kiírásokban a számokat kerekítsük 2 tizedesre!

```
# -*- coding: ISO-8859-2 -*-
from math import *
rstr = raw_input("Sugár (cm)? ")
mstr = raw_input("Magasság (cm)? ")
r = float(rstr)
m = float(mstr)
terf = r * r * pi * m
vas = terf * 7.8
fa = terf * 0.7
print "Térfogat: ",round(terf,2)," cm3"
print "Vashenger: ",round(vas,2)," g"
print "Fahenger: ",round(fa,2)," g"
```

A második sorban a *math* modulra hivatkozunk, hogy el tudjuk érni annak beépített függvényeit, jelen esetben a pi-t.
A **float** függvény a paraméterében megadott sztringet valós számmá alakítja át.
A **round** függvény az adott valós számot a megadott számú tizedesjegyre kerekíti.

1.10 Függvénymodul importálása

Már találkoztunk a nyelvbe beépített függvények fogalmával, mint amilyen például a **float**() függvény, ami a karakterláncot valós számmá alakítja, vagy az **int**(), amely egész számmá alakít. Magától értetődő, hogy nem lehet az összes elképzelhető függvényt a Python standardba belevenni. A nyelvbe beépített függvények száma viszonylag csekély : ezek azok, amik nagyon gyakran használhatók. A többiek moduloknak nevezett külön fájlokban vannak csoportosítva.

A modulok tehát fájlok, amik függvénycsoportokat fognak egybe. A későbbiekben majd meglátjuk, hogy kényelmesebb egy nagyméretű programot több kisebb méretű részre felbontani, hogy egyszerűsítsük a karbantartást. Egy jellegzetes Python-alkalmazás tehát egy főprogramból és egy vagy több modulból áll, melyek mindegyike kiegészítő függvények definícióit tartalmazza.

Nagyszámú modult adnak hivatalosan a Pythonnal. Más modulokat más szolgáltatóknál találhatunk. Gyakran egymással rokonságban lévő függvénycsoportokat próbálnak meg ugyanabba a modulba összefogni, amit könyvtárnak nevezünk.

A *math* modul például számos olyan matematikai függvény definícióját tartalmazza, mint a sinus, cosinus, tangens, négyzetgyök, stb. Ezeknek a függvényeknek a használatához elég a scriptünk elejére beszúrni a következő sort :

from *math* **import** *

Ez a sor jelzi a Pythonnak, hogy az aktuális programba bele kell venni a math modul minden függvényét (ezt jelzi a *). Ez a modul egy matematikai függvénykönyvtárat tartalmaz.
A scriptbe például az alábbiakat írjuk:

gyok = **sqrt**(*szam*) ez a *gyok* nevű változóhoz rendeli a *szam* négyzetgyökét,
sinusx = **sin**(*szog*) ez a *sinusx* nevű változóhoz rendeli a *szog* szinuszát radiánban, stb.

- A függvény valamilyen név és a hozzá kapcsolt zárójelek formájában jelenik meg
  példa : **sqrt**()
- A zárójelekben egy vagy több argumentumot adunk át a függvénynek
  példa : **sqrt**(121)
- a függvénynek van egy visszatérési értéke (azt is mondjuk, hogy « visszaad » egy értéket)
  példa : 11.0

Feladat – Bank
Ha beteszünk a bankba egy adott összeget, adott éves kamatszázalékra, adott hónapra, mennyi pénzt vehetünk majd fel az idő lejártakor?

```
# -*- coding: ISO-8859-2 -*-
from math import *
osszegstr = raw_input("Összeg(Ft)? ")
honapstr = raw_input("Hónap? ")
kamatstr = raw_input("Kamat%? ")
osszeg = float(osszegstr)
honap = float(honapstr)
kamat = float(kamatstr)
ujosszeg = osszeg * pow(1+kamat/100,honap/12)
print honap," hónap múlva ",round(ujosszeg,0)," Ft-ot vehet ki."
```

FELADATOK:

1. Kérd be a konzolról a felhasználó nevét, majd írd ki a következő jókívánságot:
Kedves <<X>>! Sikeres Python programozást!

2. Kérd be a konzolról egy téglatest három élének hosszúságát, majd írd ki a hasáb felszínét és térfogatát!

3. Tárold konstansokban a krumpli, a hagyma és a padlizsán egységárát! Írj olyan programot, amely bekéri, hogy miből mennyit óhajt a vásárló, majd készítsen egy számlát a következő formában:

Krumpli	:	2.5 kg	*	70 Ft/kg	=	175 Ft
Hagyma	:	3.0 kg	*	98 Ft/kg	=	294 Ft
Padlizsán	:	10.0 kg	*	200 Ft/kg	=	2000 Ft
Összesen						2469 Ft

4. Kérje be a gömb sugarát, majd írja ki a gömb felszínét és térfogatát!

5. Ha a számla ÁFA összege a számla nettó értékének egy adott százaléka, akkor hány százalék ÁFÁ-t tartalmaz a számla bruttó összege? Készíts a problémára egy kisegítő programot! Például 25%-os ÁFA esetén a számla 20% ÁFÁ-t tartalmaz, 12%-os ÁFA esetén a számla ÁFA tartalma 10,71%.

6. Feri pénzt kap. Hogy mennyit, azt kérje be a program. A kifizetéshez 5000, 1000, 500 és 100 Ft-os címletek állnak rendelkezésre – a maradékot Feri nem kapja meg. Feltételezzük, hogy minden címletből van elég, és a lehető legkevesebb számú pénz kerül kiosztásra. Milyen címletből hányat kapott Feri, és mennyit hagyott ott ajándékba.

2. Szelekciók

Ha igazán hasznos programokat szeretnénk írni, akkor olyan technikákra van szükség, amik lehetővé teszik a programvégrehajtás különböző irányokba történő átirányítását attól függően, hogy milyen feltételekkel találkozunk. Ehhez olyan utasítások kellenek, amikkel tesztelni lehet egy bizonyos feltételt és következményként módosítani lehet a program viselkedését.

2.1 Egyágú szelekció - if

Az **if** utasítás a legegyszerűbb ezek közül a feltételes utasítások közül.

```
a = 150
if (a > 100):
    print "a meghaladja a százat"
```

Azt a kifejezést, amit zárójelek közé tettünk, mostantól fogva feltételnek nevezzük. Az **if** ennek a feltételnek a tesztelését teszi lehetővé. Ha a feltétel igaz, akkor a « : » után beljebb igazított utasítást hajtja végre a Python. Ha a feltétel hamis, semmi sem történik. Jegyezzük meg, hogy az itt alkalmazott zárójelek opcionálisak a Pythonban. Ezeket az olvashatóság javítása érdekében alkalmazzák. Más nyelvekben kötelezők lehetnek.

Feladat – Fizetés
Kérjük be a konzolról egy alkalmazott fizetését! Ha ez a fizetés 100 000 forintnál nem nagyobb, akkor emeljük meg 25%-kal! Végül írjuk ki az alkalmazott fizetését!

```
# -*- coding: ISO-8859-2 -*-

fizstr = raw_input("Fizetés? ")
fiz = int(fizstr)

if (fiz < 100000) :
   fiz = fiz * 1.25

print "Az új fizetés: ",fiz
```

2.2 Kétágú szelekció – if .. else

```
a = 20
if (a > 100):
    print "a meghaladja a százat"
 else:
    print "a nem haladja meg a százat"
```

Az **else** (« különben ») utasítás egy alternatív végrehajtás programozását teszi lehetővé, így a programozónak két lehetőség között kell választani.

Feladat – Jó szám
Kérjünk be a konzolról egy valós számot! A szám akkor jó, ha 1000 és 2000 közötti páros egész (a határokat is beleértve). Írjuk ki, hogy a szám jó, vagy nem jó!

```
# -*- coding: ISO-8859-2 -*-

szamstr = raw_input("Szám? ")
szam = int(szamstr)

if (szam >= 1000) and (szam <= 2000) and (szam % 2 == 0):
   print "A szám jó."
else:
   print "A szám nem jó."
```

2.3 Relációs operátorok

Az **if** utasítás után kiértékelt feltétel a következő relációs operátorokat tartalmazhatja :

```
x == y          # x egyenlő y -nal
x != y          # x nem egyenlő y -nal
x > y           # x nagyobb, mint y
x < y           # x kisebb, mint y
x >= y          # x nagyobb, vagy egyenlő mint y
x <= y          # x kisebb, vagy egyenlő mint y

a = 7
if (a % 2 == 0):
    print "a páros"
    print "mert 2-vel való osztása esetén a maradék nulla"
 else:
    print "a páratlan"
```

Jól jegyezzük meg, hogy két érték egyenlőségét a dupla egyenlőségjel operátorral teszteljük, nem pedig az egyszeres egyenlőségjellel1. (Az egyszeres egyenlőségjel egy értékadó operátor.)

2.4 Logikai operátorok

A logikai kifejezésekre alkalmazhatók a **not** (nem), az **and** (és) és az **or** (vagy) műveletek. A műveletek eredményeit a következő igazságtáblák szemléltetik: (true = igaz; false = hamis)

x	**not** *x*
true	false
false	true

x	*y*	*x* **and** *y*
true	true	true
true	false	false
false	true	false
false	false	false

x	*y*	*x* **or** *y*
true	true	true
true	false	true
false	true	true
false	false	false

2.5 Többágú szelekciók – else..elif..else

Az **elif** (az « else if » összevonása) utasítást használva még jobb megoldást tudunk adni :

```
a = 0
if a > 0 :
    print "a pozitív"
elif a < 0 :
    print "a negatív"
else:
    print "a nulla"
```

Feladat – Kor

Olvassunk be egy nem negatív egész számot, valakinek az életkorát! Kortól függően írjuk ki a megfelelő szöveget:

0 – 13 évig: Gyermek
14 – 17 évig: Fiatalkorú
18 – 23 évig: Ifjú
24 – 59 évig: Felnőtt
60 évtől Idős

```
# -*- coding: ISO-8859-2 -*-

korstr = raw_input("Életkor? ")
kor = int(korstr)

if (kor <= 13):
    print "Gyermek"
elif (kor <= 17):
    print "Fiatalkorú"
elif (kor <= 23):
    print "Ifjú"
elif (kor <= 59):
    print "Felnőtt"
else:
    print "Idős"
```

2.6 Összetett utasítások – Utasításblokkok

Az **if** utasítással használt konstrukció az első összetett utasításunk. Hamarosan másfélékkel is fogunk találkozni. A Pythonban minden összetett utasításnak mindig ugyanaz a szerkezete : egy fejsor, ami kettőspontra végződik, ez alatt következik egy vagy több utasítás, ami a fejsor alatt be van húzva.

Fejsor:
a blokk első utasítása
... ...
... ...
a blokk utolsó utasítása

Ha a fejsor alatt több behúzott utasítás van, akkor mindegyiknek pontosan ugyanannyira kell behúzva lenni (például 4 karakterrel kell beljebb lenni). Ezek a behúzott utasítások alkotják az utasításblokkot. Az utasításblokk : egy logikai együttest képező utasítások sorozata, ami csak a fejsorban megadott feltételek teljesülése esetén hajtódik végre. Az előző bekezdés példájában az **if** utasítást tartalmazó sor alatti két behúzott sor egy logikai blokkot alkot: ez a két sor csak akkor hajtódik végre, ha az **if** -el tesztelt feltétel igaz, vagyis ha a 2-vel való osztás maradéka nulla.

2.7 Egymásba ágyazott utasítások

Komplex döntési struktúrák létrehozásához egymásba ágyazható több összetett utasítás.

```
if torzs == "gerincesek":                                  # 1
   if osztaly == "emlősök":                                # 2
      if rend == "ragadozók":                        # 3
         if csalad == "macskafélék":           # 4
            print "ez egy macska lehet"        # 5
      print "ez minden esetre egy emlős"        # 6
   elif osztaly == 'madarak':                     # 7
      print "ez egy kanári is lehet"            # 8
print"az állatok osztályozása összetett"                  # 9
```

A programrészlet csak abban az esetben írja ki az « ez egy macska lehet » -et, ha az első négy feltétel igaz.

Az « ez minden esetre egy emlős » kiírásához szükséges és elégséges, hogy az első két feltétel igaz legyen. Az utóbbi mondatot (6. sor) kiíratő utasítás ugyanazon a behúzási szinten van, mint az **if** rend == "ragadozók" (3. sor). A két sor tehát ugyanannak a blokknak a része, ami akkor hajtódik végre, ha az 1. és 2. sorban tesztelt feltételek igazak.

Az « ez talán egy kanári » szöveg kiírásához az szükséges, hogy a törzs nevű változó a « gerincesek » értéket, és az osztály nevű változó a « madarak » értéket tartalmazza.

A 9. sor mondatát minden esetben kiírja, mivel ugyanannak az utasításblokknak a része, mint az 1. sor.

2.8 A Python néhány szintaktikai szabálya

Az előzőek alapján összefoglalunk néhány szintaktikai szabályt :

Az utasítások és a blokkok határait a sortörés definiálja

Számos programozási nyelvben minden sort speciális karakterrel kell befejezni (gyakran pontos vesszővel). A Pythonban a sorvégejel játsza ezt a szerepet. (A későbbiekben majd meglátjuk, hogy hogyan hágható át ez a szabály, hogy egy összetett kifejezést több sorra terjesszünk ki. Egy utasítássort kommenttel is befejezhetünk. Egy Python komment mindig a # karakterrel kezdődik. Ami a # karakter és a LF (linefeed) karakter között van, a fordító figyelmen kívül hagyja.

A nyelvek többségében az utasításblokkot speciális jelekkel kell határolni (néha a begin és end utasításokkal) A C++ -ban és Java -ban, például, az utasításblokkot kapcsos zárójelekkel kell határolni. Ez lehetővé teszi, hogy az utasításblokkokat egymás után írjunk anélkül, hogy a behúzásokkal és a sorugrásokkal foglalkoznánk. Azonban ez zavaros, nehezen olvasható programok írásához vezethet. Ezért minden programozónak, aki ezeket a nyelveket használja azt tanácsolom, hogy alkalmazzák a sorugrásokat és a behúzásokat is a blokkok jó vizuális határolására.

A Pythonban használnunk kell a sorugrásokat és a behúzásokat, viszont nem kell más blokkhatároló szimbólumokkal foglalkoznunk. Végeredményben a Python olvasható kódok írására és olyan jó szokások felvételére kényszerít bennünket, amiket meg fogunk őrizni más programozási nyelvek használata során is.

Összetett utasítás = Fej , kettőspont , behúzott utasítások blokkja

Az utasításblokkok mindig egy jóldefiniált utasítást tartalmazó fejsorhoz kapcsolódnak (if, elif, else, while, def, ...), ami kettőspontra végződik.

A blokkokat behúzás határolja: egyazon blokk minden sorának pontosan ugyanúgy kell behúzva lenni (vagyis ugyanolyan számú betűközzel kell jobbra eltolva lenniük2). A behúzásra akármennyi szóközt használhatunk, a programok többsége a 4 többszöröseit alkalmazza.

Jegyezzük meg, hogy a legkülső blokknak (1. blokk) a bal margón kell lenni (semmibe sincs beágyazva).

A space-eket és a kommenteket rendszerint figyelmen kívül hagyja az interpreter

A sor elején lévő - a behúzásra szolgáló - szóközöktől eltekintve az utasítások és kifejezések belsejében elhelyezett szóközöket majdnem mindig figyelmen kívül hagyjuk (kivéve, ha ezek egy karakterlánc részét képezik). Ugyanez vonatkozik a kommentekre : ezek mindig a # karakterrel kezdődnek és az aktuális sor végéig tartanak.

FELADATOK:

1. Kérd be egy telek oldalait méterben! Írd ki a telek területét négyszögölben! (1 négyszögöl = 3,6 m^2). Ha a telek 100 négyszögölnél kisebb, akkor írja ki, hogy túl kicsi!

2. Van egy henger alakú hordónk, melybe nem tudjuk, hogy belefér-e a rendelkezésre álló bor. Kérd be a bor mennyiségét literben, majd a hordó összes szükséges adatát cm-ben. Adj tájékoztatást, hogy hány literes a hordó, és hogy belefér-e a hordóba a bor! Ha belefér, akkor add meg, hogy mennyi férne még bele! Írd ki százalékosan is a telítettséget! Az adatokat egészre kerekítve írd ki!

3. Kérj be egy évszámot! Ha a beütött szám negatív, akkor adj hibajelzést, ha nem, akkor állapítsd meg, hogy az évszám osztható-e 17-tel, vagy nem!

4. Kérd be Zsófi, Kati és Juli születési évét. Írd ki a neveket udvariassági sorrendben (előre az idősebbeket…)!

5. Kér be egy egyjegyű, nem negatív számot! Írd ki a szám szöveges formáját (1=egy, 2=kettő stb.)

6. Kérj be egy egész óra értéket. Ha a szám nem 0 és 24 óra között van, akkor adjon hibaüzenetet, egyébként köszönjön el a program a napszaknak megfelelően! 4-9: Jó reggelt!, 10-17: Jó napot!, 18-21: Jó estét!, 22-3: Jó éjszakát!

7. Egy dolgozatra annak pontszámától függően a következő osztályzatot adják:

elégtelen	**(1):**	**0 – 29**
elégséges	**(2):**	**30 – 37**
közepes	**(3):**	**38 – 43**
jó	**(4):**	**44 – 49**
jeles	**(5):**	**50 – 55**

Kérje be a dolgozat pontszámát, majd írja ki az osztályzatot számmal és betűvel!

3. Iterációk

Az egyik dolog, amit a számítógépek a legjobban csinálnak, az az azonos feladatok hiba nélküli ismétlése. Az ismétlődő feladatok programozására léteznek eljárások. Az egyik legalapvetőbbel fogjuk kezdeni : a **while** utasítással létrehozott ciklussal.

3.1 A while utasítás

```
a = 0
while (a < 7): # (ne felejtsük el a kettőspontot !)
   a = a + 1 # (ne felejtsük el a behúzást !)
... print a
```

A **while** jelentése : « amíg ». Ez az utasítás jelzi a Pythonnak, hogy az utána következő utasításblokkot mindaddig folyamatosan kell ismételnie, amíg a feltétel igaz.

Mint az előző fejezetben tárgyalt **if** utasítás, a **while** utasítás is egy összetett utasítást kezd meg. A kettőspont a sor végén a megismétlendő utasításblokkot vezeti be, aminek kötelezően beljebb igazítva kell lenni. Ahogyan azt az előző fejezetben megtanultuk, egyazon blokk valamennyi utasításának azonos mértékben kell behúzva lenni (vagyis ugyanannyi szóközzel kell jobbra eltolva lenniük).

Létrehozhatunk tehát olyan programhurkokat, ami bizonyos számú alkalommal megismétli a behúzott utasítások blokkját. Ez a következőképpen működik :

A **while** utasítás esetén a Python a zárójelben levő feltétel kiértékelésével kezdi. (A zárójel opcionális. Csak a magyarázat világossá tétele érdekében használatos)

Ha a feltétel hamis, akkor a következő blokkot figyelmen kívül hagyja és a programvégrehajtás befejeződik1.

Ha a feltétel igaz, akkor a Python a ciklustestet alkotó teljes utasításblokkot végrehajtja, vagyis :

- az $a = a + 1$ utasítást, ami 1-gyel növeli az *a* változó tartalmát
- a **print** utasítást, ami kiíratja az *a* változó aktuális értékét

amikor ez a két utasítás végrehajtódott, akkor tanúi voltunk az első iterációnak, és a programhurok, vagyis a végrehajtás visszatér a while utasítást tartalmazó sorra. Az ott található feltételt újra kiértékeli és így tovább.

Ha példánkban, az $a < 7$ feltétel még igaz, a ciklustest újra végrehajtódik és folytatódik a ciklus.

Megjegyzések :

A feltételben kiértékelt változónak a kiértékelést megelőzően léteznie kell. (Egy értéknek kell már hozzárendelve lenni.)

Ha a feltétel eredetileg hamis, akkor a ciklustest soha sem fog végrehajtódni.

Ha a feltétel mindig igaz marad, akkor a ciklustest végrehajtása vég nélkül ismétlődik (de legalábbis addig, amíg a Python maga működik). Ügyelni kell rá, hogy a ciklustest legalább egy olyan utasítást tartalmazzon, ami a **while**-lal kiértékelt feltételben megváltoztatja egy beavatkozó változó értékét úgy, hogy ez a feltétel hamissá tudjon válni és a ciklus befejeződjön.

Végtelen ciklus példája (kerülendő) :

```
n = 3
while n < 5:
  print "hello !"
```

Feladat – Bank

Van egy kis megtakarított pénzem. Arra vagyok kíváncsi, hogy hány hónap múlva éri el ez az összeg a bankban a 100 000 Ft-ot, ha havi 2%-os kamattal számolhatok?

```
# -*- coding: ISO-8859-2 -*-

penzstr = raw_input("Alaptőke? ")
penz = float(penzstr)
honap = 0

while (penz < 100000):
   penz = penz * 1.02
   honap = honap + 1

print honap," hónap múlva felvehetsz ",penz," Ft-ot"
```

Feladat – Jegyek száma

Kérjünk be a konzolról egy számot! Írjuk ki a jegyeinek a számát!

```
# -*- coding: ISO-8859-2 -*-
szamstr = raw_input("Szám? ")
szam = int(szamstr)
seged = szam
jegySzam = 0

while (seged != 0):
   seged = seged / 10
   jegySzam = jegySzam + 1

print szam," jegyeinek száma: ",jegySzam
```

Feladat – Oszthatóság
Két szám között határozzuk meg az első olyan számot, amelyik osztható egy megadott számmal!

```
# -*- coding: ISO-8859-2 -*-

kezdstr = raw_input("Egyik szám? ")
vegstr = raw_input("Másik szám? ")
osztostr = raw_input("Osztó? ")
kezd = int(kezdstr)
veg = int(vegstr)
oszto = int(osztostr)

if (kezd > veg):
   seged = kezd
   kezd =veg
   veg = seged

while (kezd % oszto != 0) and (kezd <= veg):
   kezd = kezd + 1

if (kezd > veg):
   print "Nem található ilyen szám"
else:
   print "Az első ilyen szám: ",kezd
```

3.2 Léptető ciklus - for

```
for i in range(5):
   print i
```

Eredmény:
0
1
2
3
4

```
for i in range(3,5):
   print i
```

Eredmény:
3
4

```
for i in range(3,15,4):
   print i
```

Eredmény:

3
7
11

```
for i in range(20,-20,-6):
   print i
```

Eredmény:
20
14
8
2
-4
-10
-16

Feladat – Fizetés
Most 2009-et írunk. Írjuk ki, hogy mostantól 2026-ig melyik évben mennyi lesz József fizetése, ha évenként 12%-kal növekszik! József jelenlegi fizetését a konzolról kérjük be!

```
# -*- coding: ISO-8859-2 -*-

fizstr = raw_input("Fizetés? ")
fiz = float(fizstr)

for i in range(2009,2027):
   print i,"-ben a fizetés: ",round(fiz,0)
   fiz = fiz * 1.12
```

Feladat – Szökőév
Írjuk ki megadott két évszám között az összes szökőévet, majd a szökőévek számát!

```
# -*- coding: ISO-8859-2 -*-
kezdstr = raw_input("Egyik évszám? ")
vegstr = raw_input("Másik évszám? ")
kezd = int(kezdstr)
veg = int(vegstr)
szokoevSzam = 0

if (kezd > veg):
   seged = kezd
   kezd =veg
   veg = seged

for ev in range(kezd,veg+1):
   if (ev % 400 == 0) or ((ev % 100 != 0) and (ev % 4 == 0)):
      print ev
      szokoevSzam = szokoevSzam + 1

print "A szökőévek száma: ",szokoevSzam
```

3.3 Adatok feldolgozása végjelig

Feladat – Kör kerülete
A felhasználó adott sugarú körök kerületére kíváncsi. Amikor ő beüt egy sugarat, mi kiírjuk a kör kerületét. Ha már nem kíváncsi több eredményre, akkor a kör sugaránál nullát (végjelet) kell ütnie!

```
# -*- coding: ISO-8859-2 -*-
from math import *

r = 10

while (r != 0):
   r = float(raw_input("A kör sugara: "))
   k = 2 * r * pi
   if (r != 0):
      print "A kör kerülete: ",k
```

3.4 Megszámlálás

Feladat – Megszámol
Kérjünk be számokat a felhasználótól 0 végjelig. Írjuk ki a bevitt számok számát!

```
# -*- coding: ISO-8859-2 -*-

szam = 10
db = 0

while (szam != 0):
   szam = float(raw_input("Szám: "))
   if (szam != 0):
      db = db + 1

print "Darabszám: ",db
```

Feladat – Prímszámok
Adott két szám között hány darab prímszám van? Írjuk is ki őket!

```
# -*- coding: ISO-8859-2 -*-

db = 0
kezdet = int(raw_input("Egyik szám? "))
veg = int(raw_input("Másik szám? "))

if (kezdet > veg):
   seged = kezdet
   kezdet = veg
   veg = seged

for szam in range(kezdet,veg+1):
   i=2
```

```
    while (i <= szam/2) and (szam % i != 0):
        i = i + 1
    if (i > szam / 2):
        print szam
        db = db + 1

print "A prímek száma: ",db
```

3.5 Összegzés, átlagszámítás

Feladat – Átlag
Olvassuk be a számokat 0 végjelig, majd írjuk ki ezek összegét, darabszámát és átlagát!

```
# -*- coding: ISO-8859-2 -*-

db = 0
osszeg = 0.0
szam = 10.0

while (szam != 0):
    szam = float(raw_input("Szám? "))
    if (szam != 0):
        osszeg = osszeg + szam
        db = db + 1

if (db > 0):
    print "A számok összege: ",osszeg
    print "Darabszám: ",db
    atlag = osszeg/db
    print "Átlag: ",atlag
else:
    print "Nincs beolvasott szám."
```

3.6 Minimum- és maximumkiválasztás

Feladat – Maximális számla
Kérjünk be a felhasználótól számlaösszegeket! A bevitel befejeződik, ha az összegnél nullát ütnek. Írjuk ki a legnagyobb összegű számla sorszámát és összegét!

```
# -*- coding: ISO-8859-2 -*-

maxOsszeg = -1
szam = 0
osszeg = 10
db = 0

while (osszeg != 0):
    osszeg = int(raw_input("Összeg? "))
```

```
   db = db + 1
   if (osszeg != 0) and (osszeg > maxOsszeg):
      maxOsszeg = osszeg
      szam = db

if (maxOsszeg > 0):
   print "A maximális számla száma: ",szam
   print "A maximális számla összege: ",maxOsszeg
else:
   print "Nincs érvényes bevitel."
```

3.7 Menükészítés

Feladat – Menü
Készítsünk egy menüt! Három dologból lehet választani: egyik funkció, másik funkció és kilépés. Jelenítsük meg a választási lehetőségeket: <E>gyik / <M>ásik / <V>ége ?
Az Egyik, illetve a Másik funkció választására ismételten jelenjen meg egy-egy szöveg, a Vége választására pedig legyen vége a programnak.

```
# -*- coding: ISO-8859-2 -*-

menu = "M"

while (menu != "V") and (menu != "v"):
   menu = raw_input("<E>gyik / <M>ásik / <V>ége ? ")
   if (menu == "E") or (menu == "e"):
      print "Egyik funkció."
   if (menu == "M") or (menu == "m"):
      print "Másik funkció."
```

FELADATOK

1. Szüretkor sorban lemérik a puttonyokban lévő szőlő súlyát. Készítsünk programot, amely segítségével ezek az értékek feldolgozhatók. Kíváncsiak vagyunk a nap végén, hogy összesen hány puttonnyal, és hány kg szőlőt szüreteltek. Természetesen a puttonyok számát előre nem tudjuk.

2. Egészítsük ki az előző feladatot: A szőlőt 1000 kg teherbírású gépkocsikkal szállítják el a helyszínről. Minden esetben, amikor a szőlő mennyisége meghaladta az 1000 kg-ot, akkor a program figyelmeztessen: Mehet a kocsi!

3. Munkásokat veszünk fel egy adott munkára. A szükséges létszámot a program elején megkérdezzük. Az emberek csoportosan jelentkeznek. Üssük be sorban a jelentkező csoportok létszámait! Ha megvan a szükséges létszám, akkor írjuk ki, hogy „A létszám betelt”, valamint azt, hogy hány főre van szükség az utolsó csoportból!

4. Kérjük be, hogy a héten mennyi kalóriát fogyasztottunk az egyes napokon! Ezután írjuk ki az összes kalóriafogyasztásunkat, valamint a napi átlag kalóriafogyasztást!

5. Kérj be egy egész számot, és állapítsa meg, hogy hány 0 jegy szerepel benne!

6. Hány darab négyzetszám van 1000-ig? Írd is ki a négyzetszámokat!

7. Kérdezd meg a felhasználótól, hogy mennyi pénze van, és milyen évi kamatozással tette azt be a bankba! Ezután a program tegye lehetővé, hogy az illető többször megkérdezhesse, hogy melyik évben mennyi pénze lesz.

8. Van egy bizonyos összegünk, amelyből számlákat szeretnénk kifizetni, de maximum tízet. Kérd be a rendelkezésre álló összeget, majd sorban a számlaösszegeket! Ha a számlák száma eléri a tízet, vagy az adott számlára már nincs pénz, adjon a program egy figyelmeztetést! Végül írd ki, hogy hány számlát sikerült kifizetni, és mennyi a ténylegesen kifizetendő összeg!

9. Kérj be egy határszámot, majd írd ki eddig a számig az összes prímszámot!

10. Kérdezzük meg a felhasználótól, hogy minek a területét szeretné kiszámítani: négyzetét, téglalapét vagy körét? Kérjük be a szükséges adatokat és írjuk ki az eredményt! Ezután újra kínáljuk fel a lehetőségeket! Tegyük lehetővé azt is, hogy a felhasználó kiszálljon a programból!

11. Kérjük be, hogy a héten mennyi kalóriát fogyasztottunk az egyes napokon! Ezután írjuk ki, hogy hányadik napon fogyasztottuk a legtöbb, illetve a legkevesebb kalóriát!

4. Függvények írása

A programozás annak a művészete, hogy a számítógépet olyan feladatok elvégzésére tanítjuk meg, amiket előzőleg nem tudott végrehajtani. Az egyik legérdekesebb erre szolgáló módszer az, hogy felhasználói függvények formájában új utasításokat illesszünk az általunk használt programozási nyelvbe.

4.1 Függvény fogalma, szintaktikája

Az eddig írt programjaink mind nagyon rövidek voltak, mivel a célunk csak az volt, hogy elsajátítsuk a nyelv elemeit. Amikor majd valódi projekteket kezdünk fejleszteni, gyakran rendkívül bonyolult problémékkal fogunk találkozni és a programsorok száma sokasodni fog.

Egy probléma hatékony megközelítése gyakran a problémának több, egyszerűbb alproblémára való felbontásából áll, amiket azután külön vizsgálunk. (Ezek az alproblémák esetleg tovább bonthatók még egyszerűbb alproblémákra és így tovább.

Másrészt gyakran előfordul, hogy ugyanazt az utasítás sorozatot többször kell alkalmaznunk egy programban és nyilván nem kívánjuk azt rendszeresen megismételni.

A függvények és az objektumosztályok eltérő alprogram-struktúrák, amiket a magasszintű nyelvek alkotói azért találtak ki, hogy a fent említett nehézségeket megoldják. A Python függvénydefiníciójának leírásával kezdjük. Az objektumokat és osztályokat később fogjuk tanulmányozni.

Már találkoztunk különböző előre programozott függvényekkel. Most lássuk, hogyan definiáluk mi magunk új függvényeket.

A függvénydefiníció szintaxisa a Pythonban következő :

def *függvényNeve*(*paraméterlista*):
   ...
   utasításblokk
   ...

Függvénynévnek a nyelv foglalt szavai kivételével bármilyen nevet választhatunk azzal a feltétellel, hogy semmilyen speciális vagy ékezetes karaktert sem használhatunk (az aláhúzás karakter « _ » megengedett). A változónevekhez hasonlóan főként a kisbetűk használata javasolt, nevezetesen a szavak elején (a nagybetűvel kezdődő szavakat fenn fogjuk tartani az osztályok számára, amiket a későbbiekben fogunk tanulmányozni).

A **def** utasítás az **if** -hez és a **while** -hoz hasonlóan egy összetett utasítás. Az a sor, amelyik ezt az utasítást tartalmazza kötelezően kettősponttal végződik, ami egy utasításblokkot vezet be, amit nem szabad elfelejtenünk behúzni.

A paraméterlista határozza meg, hogy argumentumként milyen információkat kell megadni, ha majd használni akarjuk a függvényt. (A zárójel üresen is maradhat, ha a függvénynek nincs szüksége argumentumokra).

Egy függvényt gyakorlatilag úgy használunk, mint akármilyen más utasítást. A programtörzsben a függvényhívás a függvény nevéből és az azt követő zárójelekből áll.

Ha szükséges, a zárójelben adjuk meg azokat az argumentumokat, amiket át akarunk adni a függvénynek. Elvileg a függvénydefinícióban megadott mindegyik paraméter számára meg kell adni egy argumentumot, bár adhatók alapértelmezett értékek ezeknek a paramétereknek.

Feladat – Függvényminta
Kérjünk be számokat a konzolról 0 végjelig! Minden egyes számról állapítsuk meg, hogy az hány jegyű! A program elején és végén, valamint az egyes eredmények kiírása után húzzunk egy-egy 20 hosszúságú vonalat!.

```
# -*- coding: ISO-8859-2 -*-

def jegyekSzama(n):
   db = 0
   while (n != 0):
      db = db + 1
      n = n / 10
   return db

def vonal():
   s = ""
   for i in range(20):
      s = s+"-"
   print s

vonal()
szam = 10

while (szam != 0):
   szam = int(raw_input("Szám? "))
   if (szam != 0):
      jsz = jegyekSzama(szam)
      print "A számjegyek száma: ",jsz
```

```
    vonal()

vonal()
```

4.2 Formális és aktuális paraméter

A ***jegyekSzama*** függvény a futáskor kapott *szam* aktuális értékéből számítja ki a jegyek számát.

A függvény fejében deklarált paramétert **formális paramétern**ek nevezzük, hiszen ez egészen addig nincs használatban, míg a függvényt meg nem hívjuk valamilyen **aktuális paramétern**rel. Egy függvénynek akárhány (nulla, egy vagy több) formális paramétere lehet. A formális paraméterlista a formális paraméterek vesszővel elválasztott sorozata.

Az aktuális paraméter bármilyen kifejezés lehet. Példánkban a függvény hívásakor lényegében a függvény futása előtt bekövetkezik az *n = szam* **paraméterátadás**. Tehát a formális paraméter felveszi az aktuális paraméter értékét.

4.3 Visszatérés a függvényből

Ha a függvény tartalmaz **return** utasítást, akkor a vezérlés visszakerül a főprogramhoz, és a függvény futása befejeződik. A függvényhívás a **return** utáni kifejezés aktuális értékét fogja felvenni.

Ha nincs **return**, akkor az alprogram befejeződése adja vissza a vezérlést a főprogramnak, ilyenkor a függvénynek nincs visszatérési értéke. (Egyes programnyelvekben az ilyen alprogramokat nevezik eljárásnak)

Egy függvényt önmagából újra meghívhatunk, mielőtt a vezérlés újra visszakerülne a főprogramhoz. Ezt a jelenséget rekurziónak nevezzük. Ilyen esetben a függvényhívásnak egy az előzőtől teljesen független példánya születik meg. A különböző hívások mind különálló, saját lokális változókkal dolgoznak. Rekurzió esetén ügyelni kell arra, hogy ne keveredjünk végtelen ciklusba. A rekurziónak kell, hogy legyen leállító feltétele!

Feladat – Függvényminta2
*Kérjünk be számokat a konzolról 0 végjelig! Minden egyes számról állapítsuk meg, hogy az hány jegyű! A program elején húzzunk egy 50 hosszú, @ karakterekből álló vonalat! Az egyes eredmények kiírása után húzzunk egy-egy 20 hosszúságú, - jelekből álló vonalat!. A program végén egy 50 hosszú * karakterekből álló vonalat!*

```
# -*- coding: ISO-8859-2 -*-

def jegyekSzama(n):
    db = 0
    while (n != 0):
        db = db + 1
        n = n / 10
    return db

def vonal(mibol,mennyit):
    s = ""
    for i in range(mennyit):
        s = s+mibol
    print s
```

```
vonal("@",50)
szam = 10

while (szam != 0):
    szam = int(raw_input("Szám? "))
    if (szam != 0):
        jsz = jegyekSzama(szam)
        print "A számjegyek száma: ",jsz
        vonal("-",20)

vonal("*",50)
```

Feladat – Háromszög
Írjunk olyan programot, amely egy háromszög három oldalából kiszámítja annak kerületét és területét!

```
# -*- coding: ISO-8859-2 -*-
from math import *

def kerulet(a,b,c):
    return a+b+c

def terulet(a,b,c):
    s = kerulet(a,b,c)/2
    return sqrt(s*(s-a)*(s-b)*(s-c))

x = float(raw_input("a: "))
y = float(raw_input("b: "))
z = float(raw_input("c: "))
print "Kerület: ",round(kerulet(x,y,z),2)
print "Terület: ",round(terulet(x,y,z),2)
```

Feladat – Faktoriális
Írjunk olyan programot, amely kiírja egy adott szám faktoriálisát!

```
# -*- coding: ISO-8859-2 -*-

def faktorialis(n):
    f = 1
    for i in range(1,n+1):
        f = f * i
    return f

szam = int(raw_input("Szám: "))
print szam,"! = ",faktorialis(szam)
```

Feladat – Faktoriális2
Írjunk olyan programot, amely kiírja egy adott szám faktoriálisát! Alkalmazzunk rekurzív algoritmust!

```
# -*- coding: ISO-8859-2 -*-

def faktorialis(n):
    if (n == 0):
        return 1
    else:
        return n * faktorialis(n-1)

szam = int(raw_input("Szám: "))
print szam,"! = ",faktorialis(szam)
```

Feladat – Fibonacci
Írjunk olyan programot, amely kiírja a Fibonacci-sorozat első n elemét!

```
# -*- coding: ISO-8859-2 -*-

def fibonacci(n):
    if (n == 1) or (n == 2):
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

szam = int(raw_input("Sorszám: "))
for i in range(1,szam+1):
    print fibonacci(i)
```

FELADATOK

1. Írj olyan függvényt, amelynek két paramétere van, és az első paraméterben megadott számszor leírja a második paraméterben megadott szöveget!

2. Írj olyan függvényt, amelynek két paramétere van, és kiírja a paraméterek közötti páros számokat!

3. Írj olyan függvényt, amely visszaadja egy szám kétszeresét!

4. Írj olyan függvényt, amely egy kör sugarából visszaadja annak területét!

5. Írj olyan függvényt, amely egy gömb sugarából visszaadja annak térfogatát!

6. Írj olyan függvényt, amely visszatér a két paraméterében megadott egész szám között egész számok összegével, a határokat is beleértve!

7. Írj olyan függvényt, amely eldönti, hogy egy adott szám prímszám-e?

5. Szövegek (Stringek) kezelése

A programozási nyelvek többségében az alfanumerikus adatok kezelésére létezik egy ***karakterlánc*** (vagy angolul ***string***) nevű adatszerkezet.

A Pythonban string típusú adat bármilyen karaktersorozat, amit vagy szimpla idézőjelek (aposztrof), vagy dupla idézőjelek határolnak

Példák :

```
mondat1 = 'a kemény tojást.'
mondat2 = '"Igen", válaszolta,'
mondat3 = "nagyon szeretem"
print mondat2, mondat3, mondat1
"Igen", válaszolta, nagyon szeretem a kemény tojást.
```

A 3 változó : mondat1, mondat2, mondat3 tehát string típusú változó.

Jegyezzük meg, hogy az olyan stringeket, melyekben aposztrofok vannak, idézőjelekkel határoljuk, míg az idézőjeleket tartalmazó stringeket aposztrofokkal határoljuk. Azt is jegyezzük meg, hogy a print utasítás a kiírt elemek közé egy betűközt szúr be.

A « \ » (backslash) néhány kiegészítő finomságot tesz lehetővé :

Lehetővé teszi, hogy egy parancsot, ami nem fér el egy sorban, azt több sorba írjunk (ez bármilyen típusú parancsra érvényes).

Egy karakterlánc belsejében a backslash speciális karakterek (sorugrás, aposztrofok, dupla idézőjelek, stb.) beszúrását teszi lehetővé. *Példák* :

```
txt3 = "E\'meg kicsoda ? kérdezte."
print txt3
E'meg kicsoda ? kérdezte.

hello = "Ez egy hosszú sor\n ami több szövegsort\
tartalmaz (Azonos módon \n működik, mint a C/C++.\n\
    Jegyezzük meg,hogy a white space-ek\n a sor elején lényegesek.\n"
print hello
Ez egy hosszú sor
 ami több szövegsort tartalmaz (Azonos módon
 működik, mint a C/C++.
   Jegyezzük meg,hogy a white space-ek
 a sor elején lényegesek..
```

Megjegyzések :

- A \n egy sorugrást idéz elő.
- A \' lehetővé teszi, hogy aposztrofokkal határolt karakterláncba aposztrofot szúrjunk be.
- A kis/nagybetű lényeges a változónevekben (Szigorúan tiszteletben kell tartanunk a kezdeti kis/nagybetű választást).

« Háromszoros idézőjelek » :

Hogy egy karakterláncba könnyebben szúrjunk be speciális vagy « egzotikus » karaktereket anélkül, hogy a backslasht alkalmaznánk, vagy magát a backslasht tudjuk beszúrni, a karakterláncot háromszoros aposztroffal vagy háromszoros idézőjellel határolhatjuk :

```
a1 = """
Használat: izee[OPTIONS]
{ -h
  -H host
}"""
```

print *a1*

Használat: izee[OPTIONS]
{ -h
  -H host
}

A karakterláncok az összetett adatoknak nevezett általánosabb adattípus egy speciális esetét képezik. A körülményektől függően a karakterláncot hol mint egyetlen objektumot, hol mint különálló karakterek együttesét akarjuk kezelni. Egy olyan programozási nyelvet, mint a Pythont tehát el kell látni olyan eljárásokkal, amik lehetővé teszik egy karakterlánc egyes karaktereihez való hozzáférést. Látni fogjuk, ez nem olyan bonyolult :

Egy karakterláncot a Python a szekvenciák kategória egy objektumának tekint. A szekvenciák elemek rendezett együttesei. Ez egyszerűen azt jelenti, hogy egy string karakterei mindig egy bizonyos sorrendben vannak elrendezve. Következésként a string minden egyes karakterének meghatározható a szekvenciabeli helye egy index segítségével.

Ahhoz, hogy egy adott karakterhez hozzáférjünk, a karakterláncot tartalmazó változó neve után szögletes zárójelbe írjuk a karakter stringbeli pozíciójának megfelelő numerikus indexet.

Figyelem : amint azt egyebütt ellenőrizhetjük, az informatikában az adatokat majdnem mindig nullától kezdve számozzuk (nem pedig egytől). Ez a helyzet egy string karakterei esetében :

Példa :

ch = "Stéphanie"
print *ch*[0], *ch*[3]
S p

5.1 Elemi műveletek karakterláncokon

A Pythonnak számos, karakterláncok kezelésére szolgáló függvénye van (kis/nagybetűs átalakítás, rövidebb karakterláncokra való darabolás, szavak keresése, stb

Elemi műveletek:

Több rövid karakterláncból összerakhatunk egy hosszabbat. Ezt a műveletet összekapcsolásnak nevezzük és a + operátort használjuk rá a Pythonban. (Ez az operátor számokra alkalmazva az összeadás műveletére, míg karakterláncokra alkalmazva az összekapcsolásra szolgál).

Példa :

a = 'A kis halból'
b = ' nagy hal lesz'
c = *a* + *b*
print *c*
A kis halból nagy hal lesz

Meghatározhatjuk egy karakterlánc hosszát (vagyis a karakterek számát) a **len**() függvény hívásával :

print len(*c*)
29

Egy számot reprezentáló karakterláncot számmá alakíthatunk.

Példa :

```
ch = '8647'
> print ch + 45
==> *** error *** nem adhatunk össze egy stringet és egy számot

 n = int(ch)
 print n + 65
8712                    # OK : 2 számot összeadhatunk
```

Ebben a példában az **int**() belső függvény a stringet számmá alakítja. A **float**() függvény segítségével valós számmá lehet alakítani egy karakterláncot.

Feladat – E betűk
Írjunk olyan programot, amely egy stringben megszámolja az e betűket!

```
# -*- coding: ISO-8859-2 -*-

def ebetukSzama(szoveg):
    db = 0
    for i in range(len(szoveg)):
        if (szoveg[i] == 'e') or (szoveg[i] == 'E'):
            db = db + 1
    return db

x = raw_input("Szöveg? ")
print "Az e betűk száma: ",ebetukSzama(x)
```

Feladat – Csillagozás
*Írjon egy programot, ami egy új változóba másol át egy karakterláncot úgy, hogy csillagot szúr be a karakterek közé. Így például, « gaston »-ból « g*a*s*t*o*n » lesz.!*

```
# -*- coding: ISO-8859-2 -*-

def csillagoz(szoveg):
    uj = ""
    for i in range(len(szoveg)-1):
        uj = uj + szoveg[i] + "*"
    uj = uj + szoveg[len(szoveg)-1]
    return uj

x = raw_input("Szöveg? ")
print csillagoz(x)
```

Feladat – Megfordít
Írjon egy programot, ami egy új változóba fordított sorrendben másolja át egy karakterlánc karaktereit. Így például « zorglub » -ból « bulgroz » lesz.!

```
# -*- coding: ISO-8859-2 -*-

def fordit(szoveg):
   uj = ""
   for i in range(len(szoveg)-1,-1,-1):
      uj = uj + szoveg[i]
   return uj

x = raw_input("Szöveg? ")
print fordit(x)
```

A karakterláncok esetében is működik mindegyik relációs. Ez nagyon hasznos lesz a szavak névsorba rendezésénél :

Ezek az összehasonlítások azért lehetségesek, mert a stringeket alkotó alfabetikus karaktereket a számítógép bináris számok formájában tárolja memóriájában, amiknek az értéke a karaktereknek az abc-ben elfoglalt hellyével van összekapcsolva. Az ASCII kódrendszerben például A=65, B=66, C=67, stb.1

A Python számos előre definiált függvényt bocsát a rendelkezésünkre, hogy könnyebben tudjunk mindenféle karakterműveletet végezni :

Az **ord**(ch) függvény bármilyen karaktert elfogad argumentumként. Visszatérési értékként a karakternek megfelelő ASCII kódot adja meg. Tehát **ord**('A') visszatérési értéke 65.

A **chr**(num) függvény ennek pontosan az ellenkezőjét teszi. Az argumentumának 0 és 255 közé eső számnak kell lenni, a 0-t és 255-öt is beleértve. Visszatérési értékként a megfelelő ASCII karaktert kapjuk meg : tehát **chr**(65) az A karaktert adja vissza.

5.2 A karakterláncok, mint objektumok

Az objektumokon metódusok (vagyis ezekhez az objektumokhoz kapcsolt függvények) segítségével is végezhetünk műveleteket. A Pythonban a stringek objektumok. A megfelelő metódusok használatával számos műveletet végezhetünk rajtuk. Íme néhány a leghasznosabbak közül :

split() : egy stringet alakít át substringek listájává. Mi adhatjuk meg argumentumként a szeparátor karaktert. Ha nem adunk meg semmit sem, akkor az alapértelmezett argumentumérték egy szóköz :

```
c2 ="Votez pour moi"
a = c2.split()
print a
['Votez', 'pour', 'moi']

c4 ="Cet exemple, parmi d'autres, peut encore servir"
c4.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

join(lista) : egyetlen karakterlánccá egyesít egy stringlistát. (Ez a metódus az előző inverze.)
Figyelem : a szeparátor karaktert (egy vagy több karaktert) az a string fogja megadni, amelyikre a metódust alkalmazzuk, az argumentuma az egyesítendő stringek listája :

b2 = ["Salut","les","copains"]
print " "***.join***(*b2*)
Salut les copains
print "---"***.join***(b2)
Salut---les---copains

find(sch) : megkeresi az sch substring pozícióját egy stringben :
ch1 = "Cette leçon vaut bien un sajt, sans doute ?"
ch2 = "sajt"
print *ch1*.***find***(*ch2*)
25

count(sch) : megszámolja a sch substring előfordulásainak számát a stringben :
ch1 = "Le héron au long bec emmanché d'un long cou"
ch2 = 'long'
print *ch1*.***count***(*ch2*)
2

lower() : kisbetűssé alakít egy stringet :
ch ="ATTENTION : Danger !"
print *ch*.***lower***()
attention : danger !

upper() : nagybetűssé alakítja a karakterláncot :
ch = "Merci beaucoup"
print *ch*.***upper***()
MERCI BEAUCOUP

capitalize() : a karakterlánc első betűjét nagybetűvé alakítja :
b3 = "quel beau temps, aujourd'hui !"
print *b3*.***capitalize***()
"Quel beau temps, aujourd'hui !"

swapcase() : minden nagybetűt kisbetűvé alakít és visszont :
ch5 = "La CIGALE et la FOURMI"
print *ch5*.***swapcase***()
lA cigale ET LA fourmi

strip() : eltávolítja a string elején és végén lévő betűközöket :
ch = " Monty Python "
ch.***strip***()
'Monty Python'

replace(c1, c2) : A stringben valamennyi c1 karaktert c2 -vel helyettesíti :
ch8 = "Si ce n'est toi c'est donc ton frère"
print *ch8*.***replace***(" ","*")
Si*ce*n'est*toi*c'est*donc*ton*frère

index(c) : megadja a c karakter stringbeli első előfordulásának indexét :
ch9 ="Portez ce vieux whisky au juge blond qui fume"
print *ch9*.***index***("w")
16

Ezeknek a metódusoknak a többségében kiegészítő argumentumokkal pontosan meg lehet adni, hogy a karakterlánc melyik részét kell kezelni. *Példa :*

```
print ch9.index("e")          # a string elejetől keresi
4                                    # és megtalálja az első 'e'-t
print ch9.index("e",5)        # csak az 5-os indextől keres
8                                    # és megtalálja a masodik 'e'-t
print ch9.index("e",15)       # a 15-dik karaktertől keres
29                                   # és megtalálja a negyedik 'e'-t
```

5.3 Karakterláncok formázása

Ez a technika minden olyan esetben nagyon hasznos, amikor különböző változók értékeiből kell egy komplex karakterláncot konstruálni.

Tegyük fel például, hogy írtunk egy programot, ami egy vizes oldat színét és hőmérsékletét kezeli. A színt egy *szin* nevű változóban tároljuk és a hőmérsékletet egy *homers* nevű változóban (float típusú változó). Azt akarjuk, hogy a programunk egy új karakterláncot hozzon létre ezekből az adatokból. Például egy ilyen mondatot : « Az oldat színe pirossá változott és hőmérséklete eléri a 12,7 °C-t ».

Ezt a karakterláncot a konkatenáció operátor (a + szimbólum) segítségével összeállíthatjuk részekből, de a **str**() függvényt is használnunk kellene, hogy a float típusú változóban tárolt numerikus értéket stringgé alakítsuk.

A Python egy másik lehetőséget is kínál. A karakterlánc a **%** operátor segítségével előállítható két elemből : a baloldalon egy formázó stringet adunk meg (valamilyen mintát) ami konverziós markereket tartalmaz és jobboldalon (zárójelben) egy vagy több objektumot ami(ke)t a Pythonnak a karakterláncba a markerek helyére kell be szúrni.

Példa :

```
szin ="zöld"
homers = 1.347 + 15.9
print "A színe %s és a hömérséklete %s °C" % (szin,homers)
A színe zöld és a hőmérséklete 17.247 °C
```

Ebben a példában a formázó string két **%s** konverziós markert tartalmaz, amiket a Python a *szin* és a *homers* változók tartalmával helyettesít.

A **%s** marker bármilyen objektumot elfogad (stringet, egészet, float-ot, ...). Más markereket alkalmazva más formázással lehet kísérletezni. Próbáljuk meg például a második markert **%s** -t **%d**-vel helyettesíteni, vagy például **%8.2g** -vel. A **%d** marker egy számot vár és azt egésszé alakítja át; a **%f** és **%g** markerek valós számokat várnak és meg tudják határozni a kiírás szélességét és pontosságát.

6. A listák

Az előző fejezetben tárgyalt stringek az összetett adatokra voltak az első példák. Az összetett adatokat arra használjuk, hogy struktúráltan csoportosítunk értékegyütteseket. A lista

definíciója a Pythonban : szögletes zárójelbe zárt, vesszővel elválasztott elemek csoportja.
Példa :
nap = ['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
print *nap*
['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']

Ebben a példában *nap* nevű változó értéke egy lista.

Megállapíthatjuk : a választott példában a listát alkotó elemek különböző típusúak lehetnek. Az első három elem string, a negyedik egész, az ötödik valós típusú, stb. (A későbbiekben látni fogjuk, hogy egy lista maga is lehet egy listának eleme !). Ebben a tekintetben a lista fogalma meglehetősen különbözik a tömb (array) vagy az « indexelt változó » fogalmától, amivel más programozási nyelvekben találkozunk.

Jegyezzük meg, hogy a listák is szekvenciák, úgy mint a karakterláncok, vagyis objektumok rendezett csoportjai. A listát alkotó különböző elemek mindig ugyanabban a sorrendben vannak elrendezve, mindegyikükhöz külön hozzá tudunk férni, ha ismerjük a listabeli indexüket. Ezeknek az indexeknek a számozása nullától indul, nem pedig egytől, a karakterláncokhoz hasonlóan.
Példák :
nap = ['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
print *nap*[2]
szerda
print *nap*[4]
20.357

A stringektől (amik egy nem módosítható adattípust jelentenek) eltérően egy listának meg lehet változtatni az elemeit :

print *nap*
['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
nap[3] = *nap*[3] +47
print *nap*
['hétfő', 'kedd', 'szerda', 1847, 20.357, 'csütörtök', 'péntek']

A lista meghatározott elemét a következő módon helyettesíthetjük más elemmel :
nap[3] = 'Július'
print *nap*
['hétfő', 'kedd', 'szerda', 'Július', 20.357, 'csütörtök', 'péntek']

A **len**() belső függvény, amivel már a stringeknél találkoztunk, a listákra is alkalmazható. A listában levő elemek számát adja vissza :
len(*nap*)
7

Egy másik belső függvény - a **del**() - segítségével (az indexe alapján) bármelyik elemet törölhetjük a listából :
del(*nap*[4])
print *nap*
['hétfő', 'kedd', 'szerda', 'Július', 'csütörtök', 'péntek']

Ugyanígy lehetőség van arra, hogy hozzáfűzzünk egy elemet egy listához, de ahhoz, hogy ezt megtegyük meg kell gondolni, hogy a lista egy objektum, aminek az egyik metódusát fogjuk használni:

```
nap.append('szombat')
print nap
['hétfő', 'kedd', 'szerda', 'Július', 'csütörtök', 'péntek', 'szombat']
```

A fenti példa első sorában az ***append***() metódust alkalmaztuk a 'szombat' argumentummal a nap objektumra. Az ***append***() metódus egy olyan függvényfajta, ami valamilyen módon a « lista » típusú objektumokhoz van kapcsolva, vagy az objektumokba van integrálva. A függvénnyel használt argumentum természetesen az az elem, amit a lista végéhez akarunk fűzni.

Jegyezzük meg, hogy egy metódust úgy alkalmazunk egy objektumra, hogy egy ponttal kapcsoljuk őket össze. (Elől áll annak a változónak a neve, ami egy objektumra hivatkozik, utána a pont, majd a metódus neve, ez utóbbit mindig egy zárójelpár követi).

Elemezzük például az alábbi kis scriptet és magyarázzuk meg a működését :

```
nap = ['hétfő', 'kedd', 'szerda', 'csütörtök', 'péntek', 'szombat', 'vasarnap']
a, b = 0, 0
while a<25:
    a = a + 1
    b = a % 7
    print a, nap[b]
```

Az 5. sorban a « modulo » operátort használjuk, amivel már előzőleg találkoztunk és ami jó szolgálatokat tehet a programozásban. Számos nyelvben (a Pythonban is) a % jel reprezentálja.

Feladat – Szétválogatás

Írjon egy programot, ami pozitív egész számokat kér be 0 végjelig, majd szétválogatja őket egy-egy listába, aszerint, hogy párosak, vagy páratlanok!

```
# -*- coding: ISO-8859-2 -*-

szam = 10
paros = []
paratlan = []

while (szam != 0):
   szam = int(raw_input("Szám? "))
   if (szam != 0):
      if (szam % 2 == 0):
         paros.append(szam)
      else:
         paratlan.append(szam)

print "Párosak:    ",paros
print "Páratlanok: ",paratlan
```

6.1 Lista módosítására szolgáló haladó « slicing » (szeletelési) technikák

Egy beépített utasítással (**del**) törölhetünk egy elemet egy listából, illetve egy beépített metódussal (***append***()) hozzáfűzhetünk egy elemet a listához. Ha jól elsajátítottuk a « szeletelés » (slicing) elvét, akkor a [ ] operátorral ugyanezt az eredményt kaphatjuk. Ennek az operátornak a használata egy kicsit nehezebb, mint az erre a feladatra szolgáló utasításoké vagy metódusoké, de több rugalmasságot enged meg :

Egy vagy több elem beszúrása egy lista tetszőleges helyére

```
szavak = ['sonka', 'sajt', 'lekvár', 'csokoládé']
szavak[2:2] =["méz"]
szavak
['sonka', 'sajt', 'méz', 'lekvár', 'csokoládé']
szavak[5:5] =['kolbász', 'ketchup']
szavak
['sonka', 'sajt', 'méz', 'lekvár', 'csokoládé', 'kolbász', 'ketchup']
```

Ha a [ ] operátort az egyenlőségjel baloldalán használjuk, hogy eleme(ke)t szúrjunk be a listába vagy töröljünk a listából, akkor kötelező megadni a céllista egy « szeletét » (azaz két indexet a zárójelben); nem pedig egy elszigetelt elemet a listában.

Az egyenlőségjel jobboldalán megadott elemnek magának is egy listának kell lenni. Ha csak egy elemet szúrunk be, akkor azt szögletes zárójelek közé kell tenni, hogy először egy egyelemű listává alakítsuk. Jegyezzük meg, hogy a szavak[1] elem nem lista (ez a « sajt » karakterlánc), míg a szavak[1:3] elem egy lista.

Elemek törlése / helyettesítése

```
szavak[2:5] = []                                    # [] üres listát jelöl
szavak
['sonka','sajt','kolbász', 'ketchup']
szavak[1:3] = ['saláta']
szavak
['sonka', 'saláta', 'ketchup']
szavak[1:] = ['majonéz', 'csirke', 'paradicsom']
szavak
['sonka', 'majonéz', 'csirke', 'paradicsom']
```

A példa első sorában a [2:5] szeletet egy üres listával helyettesítjük, ami egy törlésnek felel meg.

A negyedik sorban egyetlen elemmel helyettesítünk egy szeletet. (Mégegyszer jegyezzük meg, hogy ennek az elemnek lista formájában kell megjelenni).

A 7-ik sorban egy kételemű szeletet egy három elemű listával helyettesítünk.

6.2 Számokból álló lista létrehozása a range() függvénnyel

Ha számsorokat kell kezelnünk, akkor azokat nagyon könnyen létrehozhatjuk a következő függvénnyel :

```
range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A **range**() függvény növekvő értékű egész számokból álló listát generál. Ha a **range**() -et egyetlen argumentummal hívjuk, akkor a lista az argumentummal megegyező számú értéket fog tartalmazni, de az értékek nullától fognak kezdődni (vagyis a **range**(*n*) a 0-tól *n*-1 -g terjedő egész számokat hozza létre).

Jegyezzük meg, hogy a megadott argumentum soha sem szerepel a létrehozott listában.

A **range**()-et két vagy három, vesszővel elválasztott argumentummal is használhatjuk speciálisabb számsorok létrehozására :

range(5,13)
[5, 6, 7, 8, 9, 10, 11, 12]
range(3,16,3)
[3, 6, 9, 12, 15]

6.3 Lista bejárása a for, range() és len() segítségével

A for utasítás ideális egy lista bejárásához :

```
mondat = ['La','raison','du','plus','fort','est','toujours','la','meilleure']
for szo in mondat:
    print szo,
```

La raison du plus fort est toujours la meilleure

Egy szekvencia (lista vagy karakterlánc) indexeinek automatikus előállításához nagyon praktikus a **range**() és a **len**() függvények kombinálása. *Példa :*

```
mese = ['Maître','Corbeau','sur','un','arbre','perché']
for index in range(len(mese)):
  print index, mese[index]
```

A script végrehajtásának eredménye :

0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché

6.4 A dinamikus típusadás egy következménye

A **for** utasítással használt változó típusa a bejárás folyamán folyamatosan újra van definiálva. Még akkor is, ha egy lista elemei különböző típusúak, a **for** segítségével úgy járhatjuk be ezt a listát, hogy nem kapunk hibaüzenetet, mert a ciklusváltozó típusa automatikusan alkalmazkodik az adat olvasása során annak típusához. *Példa* :

```
divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
for item in divers:
    print item, type(item)
```

3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
Linux is not Windoze <type 'str'>

A fönti példában a beépített **type**() függvényt arra használjuk, hogy megmutassuk az *item* változó tényleg megváltoztatja a típusát minden iterációban (ezt a Python dinamikus típusadása teszi lehetővé)

6.5 Műveletek listákon

A + (konkatenáció) és * (szorzás) operátorokat alkalmazhatjuk a listákra :

```
gyumolcsok = ['narancs','citrom']
zoldsegek = ['póréhagyma','hagyma','paradicsom']
gyumolcsok + zoldsegek
['narancs','citrom','póréhagyma','hagyma','paradicsom']
gyumolcsok * 3
['narancs','citrom','narancs','citrom','narancs','citrom']
```

A * operátor különösen hasznos n azonos elemből álló lista létrehozásakor :

```
het_nulla = [0]*7
het_nulla
[0, 0, 0, 0, 0, 0, 0]
```

Például tételezzük fel, hogy egy olyan *B* listát akarunk létrehozni, ami ugyanannyi elemet tartalmaz, mint egy másik *A* lista. Ezt különböző módon érhetjük el, de az egyik legegyszerűbb megoldás a : *B* = [0]*len(*A*)

6.6 Tartalmazás igazolása

Az **in** utasítás segítségével könnyen meghatározhatjuk, hogy egy elem része e egy listának :

```
v = 'paradicsom'
if v in zoldsegek:
    print 'OK'
OK
```

6.7 Lista másolása

Tegyük fel, hogy van egy *fable* nevű listánk, amit át szeretnénk másolni a *phrase* nevű új változóba. Az olvasó első ötlete bizonyára egy egyszerű értékadás :

```
phrase = fable
```

Ha így járunk el, nem hozunk létre valódi másolatot. Az utasítást követően még mindig csak egy változó van a számítógép memóriájában. Amit létrehoztunk az csak egy új hivatkozás a listára. Próbáljuk ki például :

```
fable = ['Je','plie','mais','ne','romps','point']
phrase = fable
fable[4] ='casse'
phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Ha a *phrase* változó valóban a lista másolatát tartalmazná, akkor az a másolat független lenne az eredetitől. Akkor viszont nem tudná módosítani egy olyan utasítás, mint

amit a harmadik sorban a *fable* változóra alkalmaztunk. Kísérletezhetünk még akár a *fable*, akár a *phrase* tartalmának más módosításaival is. Mindegyik esetben meg állapíthatjuk, hogy az egyik lista módosítása tükröződni fog a másik listában és viszont.

Valójában mind a két név ugyanazt az objektumot jelöli a memóriában. Erre a helyzetre az informatikusok azt mondják, hogy a *phrase* név a *fable* név aliasa.
A későbbiekben látni fogjuk az aliasok hasznát. Most azt szeretnénk, ha volna egy olyan eljárásunk, amivel létre tudunk hozni egy valódi listamásolatot. Ez például a lista elemeinek egyenként történő, új listába való bemásolásával valósíthatjuk meg.

Feladat – Lista másolása
Írjon egy programot, ami egy adott listáról másolatot készít!

```
# -*- coding: ISO-8859-2 -*-

lista1 =
['január','február','március','április','május','június','július','augusztus','szeptember','október','nov
ember','december']
lista2 = []

for i in range(len(lista1)):
   lista2.append(lista1[i])

print lista2
```

7. Véletlenszámok

A Python ***random*** modulja egész sor - különböző eloszlású - véletlenszámokat generáló függvényt kínál. A modul összes függvényét importálhatjuk a következő utasítással :

from random import *

Az alábbi függvény nulla és egy értékek közé eső véletlen valós számokat hoz létre. Argumentumként a lista méretét kell megadni :

```
def veletlen_lista(n):
    s = [0]*n
    for i in range(n):
        s[i] = random()
    return s
```

Először egy nullákból álló n elemű listát hoztunk létre, majd a nullákat véletlen számokkal helyettesítettük.

7.1 Egész számok véletlen sorsolása

Saját projektek fejlesztése során, gyakran előfordul, hogy egy olyan függvényre lesz szüksége, amivel bizonyos határok közé eső véletlen egész számokat tud generálni. Például ha egy játékprogramot akarunk írni, amiben a kártyákat véletlenszerűen kell húzni (egy 52 lapos készletből), akkor biztosan hasznos lenne egy olyan függvény, ami képes lenne 1 és 52 között (az 1-et és 52-t is beleértve) egy véletlen számot sorsolni.

Erre a ***random*** modul **randrange**() függvénye használható.

Ez a függvény 1, 2, 3 argumentummal használható.

Egyetlen argumentummal használva : nulla és az argumentum eggyel csökkentett értéke közé eső egész számot ad visszatérési értékül. *Például* : **randrange**(6) egy 0 és 5 közé eső számot ad visszatérési értéknek.

Két argumentummal használva a visszatérési értéke az első argumentum és az eggyel csökkentett második argumentum értéke közé eső szám. *Például* : **randrange**(2, 8) egy 2 és 7 közé eső számot ad visszatérési értéknek.

Ha egy harmadik argumentumot is megadunk, akkor az azt adja meg, hogy a véletlenszerűen sorsolt egészeket egymástól a harmadik argumentummal definiált intervallum választja el. *Például* a **randrange**(3,13,3) a 3, 6, 9, 12 sorozat elemeit fogja visszatérési értékként adni. :

```
for i in range(15):
   print random.randrange(3,13,3),
```

3 12 6 9 6 6 12 6 3 6 9 3 6 12 12

Feladat – Dobókocka
Írjon egy programot, ami 20 dobást szimulál egy dobókocka esetén! Írja ki a dobások eredményét, majd a dobások átlagát!

```
# -*- coding: ISO-8859-2 -*-
from random import *

kocka=[0]*20
osszeg = 0.0

for i in range(20):
   kocka[i]=randrange(1,7)
   osszeg = osszeg + kocka[i]

print kocka
atlag = osszeg / 20
print "Átlag: ",atlag
```

8. Osztályok, objektumok, attributumok

Most azt vizsgáljuk meg, mi magunk hogyan definiálhatunk új objektumosztályokat. A téma viszonylag nehéz. Fokozatosan közelítjük meg. Nagyon egyszerű objektumosztályok definícióival kezdjük, amiket a későbbiekben tökéletesítünk.

A reális világ objektumaihoz hasonlóan az informatikai objektumok is nagyon egyszerűek, vagy igen komplikáltak lehetnek. Különböző részekből állhatnak, amik maguk is objektumok. Az utóbbiakat más, egyszerűbb objektumok alkothatják, stb.

8.1 Az osztályok haszna

Az osztályok az objektum orientált programozás (Object Oriented Programming vagy OOP) fő eszközei. Ezzel a fajta programozással a komplex programokat egymással és a külvilággal kölcsönható objektumok együtteseként struktúrálhatjuk.

A programozás ilyen megközelítésének az az egyik előnye, hogy a különböző objektumokat (például különböző programozók) egymástól függetlenül, az áthatások kockázata nélkül alkothatják meg. Ez az egységbezárás (encapsulation) elvének a

következménye. Eszerint : az objektum feladatának ellátására használt változók és az objektum belső működése valamilyen formában « be vannak zárva » az objektumba. Más objektumok és a külvilág csak jól definiált eljárásokkal férhetnek hozzájuk.

Az osztályok alkalmazása egyebek között lehetővé teszi a globális változók maximális mértékű elkerülését. A globális változók használata veszélyeket rejt, különösen a nagyméretű programok esetében, mert a programtestben az ilyen változók mindig módosíthatók vagy átdefiniálhatók. (Ennek különösen megnő a veszélye, ha több programozó dolgozik ugyanazon a programon).

Az osztályok alkalmazásából eredő másik előny az a lehetőség, hogy már meglévő objektumokból lehet új objektumokat konstruálni. Így egész, már megírt programrészeket lehet ismételten felhasználni (anélkül, hogy hozzájuk nyúlnánk !), hogy új funkcionalitást kapjunk. Ezt a leszármaztatás és a polimorfizmus teszik lehetővé.

A leszármaztatás mechanizmusával egy « szülő » osztályból « gyermek » osztályt konstruálhatunk. A gyermek örökli a felmenője minden tulajdonságát, funkcionalitását, amikhez azt tehetünk hozzá, amit akarunk.

A polimorfizmus teszi lehetővé, hogy különböző viselkedésmódokkal ruházzuk fel az egymásból leszármaztatott objektumokat, vagy a körülményektől függően önmagát az objektumot.

Az objektum orientált programozás opcionális a Pythonban. Sok projekt sikeresen végigvihető az alkalmazása nélkül olyan egyszerű eszközökkel, mint a függvények. Viszont tudjunk róla, hogy az osztályok praktikus és hatékony eszközök. Megértésük segíteni fog a grafikus interface-ek (Tkinter, wxPython) megtanulásában és hatékonyan készít elő olyan modern nyelvek alkalmazására, mint a C++ vagy a Java.

8.2 Egy elemi osztály (class) definíciója

A **class** utasítással hozunk létre új objektumosztályt. Az utasítás használatát egy elemi objektumtípus - egy új adattípus - definiálásán tanuljuk meg. Az eddig használt adattípusok mind be voltak építve a nyelvbe. Most egy új, összetett adattípust definiálunk : a Pont típust. A matematikai pont fogalmának felel meg. A síkban két számmal jellemzünk egy pontot (az *x* és *y* koordinátáival). Matematikai jelöléssel a zárójelbe zárt *x* és *y* koordinátáival reprezentáljuk. Például a (25,17) pontról beszélünk. A Pythonban a pont reprezentálására **float** típusú koordinátákat fogunk használni. Ezt a két értéket egyetlen entitássá vagy objektummá szeretnénk egyesíteni. Ezért egy ***Pont***() osztályt definiálunk) :

class *Pont*:
"Egy matematikai pont definíciója"

Az osztályok definíciói bárhol lehetnek a programban, de általában a program elejére (vagy egy importálandó modulba) tesszük őket. A fenti példa valószínűleg az elképzelhető legegyszerűbb példa. Egyetlen sor elég volt az új ***Pont***() objektumtípus definiálásához. Rögtön jegyezzük meg, hogy :

- A class utasítás az összetett utasítások egy új példája. Ne felejtsük le a sor végéről a kötelező kettőspontot és az azt követő utasításblokk behúzását. Ennek a blokknak legalább egy sort kell tartalmazni. Megállapodás szerint, ha a **class** utasítást követő első sor egy karakterlánc, akkor azt kommentnek tekinti a Python és automatikusan beágyazza az osztályok dokumentációs szerkezetébe, ami a Python integráns részét képezi. Váljon szokásunkká, hogy erre a helyre mindig az osztályt leíró karakterláncot teszünk !.

- Emlékezzünk arra a konvencióra, hogy az osztályoknak mindig nagybetűvel kezdődő nevet adunk. A szövegben egy másik konvenciót is betartunk : minden osztálynévhez zárójeleket kapcsolunk, ahogyan a függvénynevekkel is tesszük.

Definiáltunk egy ***Pont***() osztályt, amit mostantól kezdve **Pont** típusú objektumok példányosítással történő létrehozására használhatunk. Hozzunk létre például egy új p91 objektumot :
p9 = ***Pont***()

A *p9* változó egy új ***Pont***() objektum hivatkozását tartalmazza. Azt is mondhatjuk, hogy a *p9* egy új példánya a ***Pont***() osztálynak.
Figyelem : utasításban hívott osztályt mindig zárójeleknek kell követni a függvényekhez hasonlóan (még akkor is, ha semmilyen argumentumot sem adunk meg). A későbbiekben majd meglátjuk, hogy az osztályok hívhatók argumentumokkal.
Jegyezzük meg : egy osztálydefiníciónál nincs szükség zárójelekre (szemben a függvény definiciókkal) kivéve, ha azt akarjuk, hogy a definiált osztály egy másik - már létező - osztály leszármazottja legyen.

Az új *p9* objektumunkon mostmár elvégezhetünk néhány elemi műveletet. *Példa* :
print *p9****.__doc__***
Egy matematikai pont definíciója

A különböző Python objektumok dokumentációs stringjei az előredefiniált ***__doc__*** attribútumhoz vannak rendelve.
print *p9*
<__main__.Pont instance at 0x403e1a8c>

A Python-üzenet azt jelenti, hogy a *p9* a főprogram szintjén definiált ***Pont***() osztály egy példánya. A memória egy jól definiált helyén helyezkedik el, aminek a címe hexadecimálisan van megadva.

Ez az osztály ettől kezdve saját változókkal és metódusokkal rendelkezhet. Az egyik kitüntetett metódus a konstruktor, ami az objektum kezdeti értékeinek beállítását oldja meg. A konstruktor neve kötelezően: ***__init__***

Feladat – Raktárprogram
Adott egy zöldségraktár, melyben pillanatnyilag egyetlen árut, paradicsomot raktározunk.
A raktárba gyakran teszünk be, illetve veszünk ki onnan paradicsomot.
A paradicsom pillanatnyi egységára 300 Ft, de ez változhat
Készítsünk olyan programot, amely segítségével rögzíteni tudjuk a megfelelő adatokat, és bármikor jelentést tudunk adni a paradicsom aktuális mennyiségéről, egységáráról, értékéről!
Menüből lehessen kiválasztani, hogy beteszünk-e, vagy kiveszünk paradicsomot, elletve, hogy emeljük, avagy csökkentjük a paradicsom egységárát! Minden művelet során adjunk jelentést!

```
# -*- coding: ISO-8859-2 -*-

class Aru:

    def __init__(self,aruNev,aruEgysegar):
        self.nev = aruNev
        self.egysegar = aruEgysegar
        self.menny = 0
```

```
    def setEgysegar(self,aruEgysegar):
        if (aruEgysegar >= 0):
            self.egysegar = aruEgysegar

    def getAr(self):
        return self.menny * self.egysegar

    def hozzatesz(self,aruMenny):
        if (aruMenny > 0):
            self.menny = self.menny + aruMenny

    def elvesz(self,aruMenny):
        if (aruMenny > 0) and (aruMenny <= self.menny):
            self.menny = self.menny - aruMenny

    def __doc__(self):
        print self.nev," Egységár: ",self.egysegar," Mennyiség: ",self.menny," Ár: ",self.getAr()

aru = Aru("Paradicsom",300)

m = 'A'
while not((m == 'v') or (m == 'V')):
    print "B: Paradicsom berakása a raktárba"
    print "K: Paradicsom kivétele a raktárból"
    print "E: Egységár módosítása"
    print "V: Vége"
    m = raw_input("Választás? ")
    if (m == 'B') or (m == 'b'):
        x = int(raw_input("Mennyit tesz a raktárba? "))
        aru.hozzatesz(x)
        aru.__doc__()
    elif (m == 'K') or (m == 'k'):
        x = int(raw_input("Mennyit vesz ki a raktárból? "))
        aru.elvesz(x)
        aru.__doc__()
    elif (m == 'E') or (m == 'e'):
        x = int(raw_input("Mennyi az új egységár? "))
        aru.setEgysegar(x)
        aru.__doc__()
```

Feladat – Másodfokú egyenlet
Írjunk olyan programot, amely kiszámítja egy másodfokú egyenlet gyökeit annak együtthatóiból!

```
# -*- coding: ISO-8859-2 -*-
from math import *

class MasodfokuEgyenlet:
```

```
    def __init__(self,szam1,szam2,szam3):
        self.a = szam1
        self.b = szam2
        self.c = szam3

    def diszkriminans(self):
        return self.b * self.b - 4 * self.a * self.c

    def megoldas1(self):
        return (-self.b + sqrt(self.diszkriminans())) / (2 * self.a)

    def megoldas2(self):
        return (-self.b - sqrt(self.diszkriminans())) / (2 * self.a)

    def __doc__(self):
        if (self.a == 0):
            print "Az egyenlet nem másodfokú."
        elif (self.diszkriminans() < 0):
            print "Az egyenletnek nincs valós megoldása."
        else:
            print "Az egyenlet egyik megoldása: ",self.megoldas1()
            print "Az egyenlet másik megoldása: ",self.megoldas2()

m = 'A'
while not((m == 'v') or (m == 'V')):
    print "E: A másodfokú egyenlet együtthatóinak bevitele"
    print "V: Vége"
    m = raw_input("Választás? ")
    if (m == 'E') or (m == 'e'):
        a = float(raw_input("a? "))
        b = float(raw_input("b? "))
        c = float(raw_input("c? "))
        egyenlet = MasodfokuEgyenlet(a,b,c)
        egyenlet.__doc__()
```

8.3 Öröklődés

A napjaink leghatékonyabb programozási technikájának tekintett objektum orientált programozásnak (Object Oriented Programming vagy OOP) az osztályok a fő eszközei. Ennek a programozási típusnak az egyik fő előnye az, hogy mindig felhasználhatunk egy már meglévő osztályt egy új, néhány eltérő vagy kiegészítő funkcionalitással rendelkező osztály létrehozására. Az eljárást leszármaztatásnak nevezzük. Egy általánostól a speciális felé haladó osztály hierarchia kialakítását teszi lehetővé.

Feladat – Síkidomok öröklődése
Írjunk olyan programot, amely kiszámítja a négyzet, a téglalap és a kör kerületét, területét! Az egyes síkidomok osztályok legyenek, megfelelő metódusokkal, alkalmazzuk az öröklődést!

```
# -*- coding: ISO-8859-2 -*-
```

```
from math import *

class Teglalap:

    def __init__(self,szam1,szam2):
        self.a = szam1
        self.b = szam2

    def kerulet(self):
        return 2*(self.a + self.b)

    def terulet(self):
        return self.a * self.b

    def __doc__(self):
        print "Kerület: ",self.kerulet()
        print "Terület: ",self.terulet()


class Negyzet(Teglalap):

    def __init__(self,szam1):
        Teglalap.__init__(self,szam1,szam1)

    def __doc__(self):
        Teglalap.__doc__(self)


class Kor(Negyzet):

    def kerulet(self):
        return 2 * self.a * pi

    def terulet(self):
        return self.a * self.a * pi

    def __doc__(self):
        Negyzet.__doc__(self)

m = 'A'
while not((m == 'v') or (m == 'V')):
    print "T: Téglalap"
    print "N: Négyzet"
    print "K: Kör"
    print "V: Vége"
    m = raw_input("Választás? ")
    if (m == 't') or (m == 'T'):
        a = float(raw_input("a? "))
        b = float(raw_input("b? "))
        teglalap = Teglalap(a,b)
```

```
    teglalap.__doc__()
if (m == 'n') or (m == 'N'):
    a = float(raw_input("a? "))
    negyzet = Negyzet(a)
    negyzet.__doc__()
if (m == 'k') or (m == 'K'):
    r = float(raw_input("r? "))
    kor = Kor(r)
    kor.__doc__()
```

VÉGE AZ ELSŐ RÉSZNEK