PYTHON segéd **v1.0.1**

Középiskola – informatika / digitális kultúra

2021

Beke Imre

Bevezetés

A Python egy általános célú, magas szintű programozási nyelv, melyet Guido van Rossum holland programozó kezdte el fejleszteni. A nyelv tervezési filozófiája az olvashatóságot és a programozói munka megkönnyítését helyezi előtérbe a futási sebességgel szemben.

A Python többek között a funkcionális, az objektumorientált, az imperatív és a procedurális programozási paradigmákat támogatja.

A Python úgynevezett interpreteres nyelv, ami azt jelenti, hogy nincs különválasztva a forrás- és tárgykód, a megírt program máris futtatható, ha rendelkezünk a Python értelmezővel. A Python értelmezőt több géptípusra (processzorra) és operációs rendszerre is elkészítették. Így a megszokott Ms Windows-os környezeteken kívül, Linuxos vagy Mac Os operációsrendszereket használó hardverekre is lehet alkalmazást fejleszteni.

Telepítés

A fejlesztőrendszer telepítéséhez a Python hivatalos webhelyéről az operációs rendszertől függően választhatunk telepítő készletet. (Download Python | Python.org) Választási szempont, hogy hány bites a rendszerünk (32 / 64) illetve exe vagy msi formátumú telepítőt akarunk használni. Linux operációs rendszereknél a telepítést parancs üzemmódban, egy terminál ablakban deb vagy rpm formátumú telepítőkészletekből lehet megoldani disztribúciótól függően.

A tényleges program vagy alkalmazás fejlesztése legtöbbször valamilyen integrált fejlesztési környezetben (IDE) történik. Az alap környezet az. un IDLE (tkinter GUI-val, telepítéssel rendelkezésre áll.) Ezen kívül több jól használható IDE-t is érdemes kipróbálni. Mindegyiknek van valamilyen hasznos többlet szolgáltatása az alap rendszerhez viszonyítva.

Ezek közül három kiemelhető:

- Thonny nevű (nemcsak x86, hanem több ARM processzoros rendszerben IoT eszközökre történő programfejlesztést tesz lehetővé. (https://thonny.org/)
- Spyder könnyen telepíthető, Ms Windows, Linuxos operációs rendszer környezetekben (https://www.spyder-ide.org/).
- Visual Studio Code, hatékony eszköz bármely nyelvi környezetben lefolytatott programfejlesztés esetén. (https://code.visualstudio.com/download) Azoknak előnyös, akik már korábban MS fejlesztőeszközöket használtak.

Dokumentáció

A Python fejlesztőrendszeréhez és különböző változatokhoz az Interneten keresztül a következő linken érhetők el angol nyelvű dokumentációk, leírások: 3.9.2 Documentation (python.org)

Az önálló tanuláshoz elérhető ész az ingyenes online anyagok közül kiemelendő:

- https://www.w3schools.com/python/default.asp
- https://www.tutorialspoint.com/python/index.htm

Fejlesztőrendszer indítása

A telepítés során beállítható, hogy az adott operációs rendszer grafikus felületén jöjjön létre egy indító ikon, melyen keresztül a konkrét IDE elérhető.

Itt még néhány beállítást lehet és célszerű megtenni:

1. Munka könyvtár beállítása
2. A felhasználói felület tulajdonságai (karaktertípus, szín, téma …)
3. Egyes IDE-k lehetővé teszik, hogy ha több verziója is van a Python nyelvnek telepítve a gépre, akkor melyik verzió legyen az alapértelmezett a fejlesztés során.

Program fejlesztés lépései

1. Probléma felvetése
   Az a fázis, amely során megtörténik a megoldandó feladat megfogalmazása, értelmezése
2. Tervezés
   Ebben a szakaszban elkészül több dokumentum, melyek a program kódolását meghatározza, azt követően a program helyességének ellenőrzését támogatja.
   a. Modell alkotása (pl. lokális, hálózati környezetre készül …)
   b. Az alkalmazásban használt adatok típusainak minden lehetséges adatjellemzőnek, a rendszer adatainak tárolását végző fájlok struktúráinak leírása
   c. Algoritmusok megfogalmazása
   d. Inputok, outputok tervezése
   e. Felhasználói felület tervezése
   f. Rendszer üzemeltetéshez kapcsolódó mentési helyreállítási stratégia megfogalmazása
   g. Tesztelési terv
3. Program kódolása
   A tervezés során létrehozott dokumentáció alapján a program kódolása. A fejlesztendő program összetettségétől függően a kódok megírása.
   a. Közösen használt programelemek kódolása
   b. Részfeladatok kódolása
   c. Teljes kód összeszerkesztése, fordítása
4. Tesztelés, tesztelési stratégia
   a. „Black box” – funkcionális alapon elvégzett teszt
   b. „White box” – forrás program alapján összeállított adatokkal
5. Dokumentálás
   a. Fejlesztői dokumentumok elkészítése
   b. Felhasználói dokumentumok, használati útmutató (telepítés / mentés / helyreállítás)
6. Archiválás
7. Fejlesztési verziók szétválasztása (Git)

Python program szerkezete

A Python nyelven írt programok nagy része az alább összeállított szerkezetnek feleltethető meg.

```
#
#Program neve
#Program szerzője:
#Verzió:
#Dátum
#

# A fejlesztett alkalmazás által használt „gyári” modulok
#
import math             # pl. matematikai modul
import random           # véletlen számokat kezelő modul
import time             # idő/dátum modul
#
# Saját fejlesztésű modulok
#
import modul1__neve   # saját modul neve  (1-es)
import modul2__neve   # saját modul neve  (2-es)
```

```
#Osztályok definíciói
#
class osztaly_1 :     # saját osztály definiálása (1.)
…
…
class osztaly_k :     # saját osztály definiálása (k.)


# A program részére kifejlesztett függvények
#
def fuggveny_1() :    # saját függvény definiálása argumentum nélkül
      # utasítások
      …
      return

def fuggveny_2(arg) : # saját függvény definiálása egy paraméteres
      # utasítások
      …
      return

def fuggveny_3(arg1, arg2): # saját függvény definiálása két
paraméteres
      # utasítások
      …
      return z

def fuggveny_n() :    # saját függvény definiálása n-edik
      # utasítások
      …
      return

def start():
      # pl.a fejlesztett alkalmazás indító függvénye nem kötött a név
      …
      return

#Itt kezdődik, innen indul ténylegesen a „főprogram”
#
# Valamilyen értékadás, adatbeviteli eljárás etc….
start()    # nincs kötött függvény név rögzítve.
```

Program elemek

Ebbe a részbe kerültek a Python nyelv azon szabályai, melyeket a forráskódok megalkotása során mindig eszünkben kell tartani. A szabályok ez a része szintaxis jellegű. Ezek kötött formai, tartalmi előírások, ezt egyszerűen tudomásul kell venni.

A programok létrehozásánál szoktak lenni egyéb nem kötelező szabályok is, melyeket konvenciókként (hallgatólagos megállapodásként) értelmezünk. pl. adatbeviteli mezők között Tab billentyűvel

léptetünk a GUI-ban, F1 billentyű Help-et hív, beszédes nevek használata a változóknál, függvények neveinek létrehozásánál. (stb)

A programokat célszerű ellátni megjegyzésekkel, kommentekkel. A Python nyelvben a sor elején elhelyezett # után írt szöveget nem értelmezi az interpreter utasításnak. Másik lehetőség, ha egy soron belül a szabályosan leírt utasítás után tesszük ki a # jelet és azt követően a megjegyzést. Több soros megjegyzése a ”””” – három macskaköröm után kezdve következő módon kerülhet a forrásprogramba:

”””
A komment szöveg első sora
2. sora
…
n-edik. sora
”””
A kommentet egy második ”””” jel zárja.

Kötött elnevezések

Ezeket a kulcsszavakat csak a programozási nyelv specifikált céljainak megfelelően lehet használni. pl. nem lehet megadni egy változó vagy függvény nevének az itt felsorolt kulcsszavak egyikét se.

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Adattípusok

A Python nyelv hasonlóan más programozási nyelvekhez sok féle adattípus felhasználását támogatja. A következő felsorolásban láthatjuk a Python nyelv beépített adattípusait:

Szöveg vagy Text Type:	`str`
Számok vagy Numeric Types:	`int, float, complex`
Listák / Sequence Types:	`list, tuple, range`
Mapping Type:	`dict`
Set Types:	`set, frozenset`
Logikai vagy Boolean Type:	`bool`
Bináris vagy Binary Types:	`bytes, bytearray, memoryview`

Egy adat vagy egy objektum típusát a Python nyelvben a **`type()`** függvénnyel tudjuk lekérdezni. használata **`type(<változó neve>)`** az argumentumban meg kell adni a keresett, létező változó nevét. Hibaüzenetet kapunk, ha nem létező változóra használjuk a függvényt.
Fontos `az` **`id(<változó neve>)`** függvény, ami az adott változó tárbeli címéről ad információt.

Változó névadása során betartandó fontos szabályok:

- a név **betűvel** vagy ”_” alul vonás karakterrel kezdődhet
- egy név nem kezdődhet számjeggyel
- a nevek **alfanumerikus** karakterekből állíthatók össze
- a kis és nagybetűt megkülönbözteti pl a Txt nem ugyanaz, mint a txT
- `global` címke egy változó neve előtt jelzi, hogy az bárhol elérhető, globális!

Numerikus

Értékadási művelet során a változók automatikusan az érték függvényében kapják meg típusukat.

```
x = 10       # int
y = 25.8     # float
z = 3j       # complex
```

A különböző számtípusok közötti konvertálás az `int(), float(),` and `complex()` metódusokkal lehetséges.

Konvertáljuk `int` -ből `float` -ba:
```
a = float(x)
```

Konvertáljuk `float` -ból `int` -be:
```
b = int(y)
```

Konvertáljuk `int`-ből `complex` formába:
```
c = complex(x,y)      # x a valós, y az imaginárius rész
```

Véletlen számok használata sokszor előfordul pl. tesztadatok generálásakor. A Python nyelvben véletlen számok létrehozásához használatba kell venni a Python nyelv `random` modulját. Néhány függvény, melyekkel véletlenszámokat generálhatunk. pl.

```
import random
random.randrange(start,stop,step) # start-stop közötti véletlenszám
            step a lépésköz. Ha nem adjuk meg, alapértelmezetten 1.
random.randint(start,stop) # start-stop közötti egész
random.random()       # 0, 1 intervallumból float típusú szám.
random.choice(lista)  # A lista egy eleme véletlenszerűen kiválasztva
random.seed()         # véletlenszám generátor inicializálását végzi.
```

Sokszor használt matematikai függvények közül az alább felsoroltak lettek kiválasztva.
Ezen függvények eléréséhez szükséges a `math` modul importálása programunk elején.
pl.

```
import math
```
Legyenek az `x, y, z …` nevekkel jelölt változók `int, float` típusúak

```
math.exp( x )   # exponenciális kifejezés kiszámolja az e^x értékét.
math.fabs( x )  # x abszolút értéke
math.pow(x, y)) # alap x, y hatványkitevő x=10, y=2, értéke 10^2
math.sqrt( x )  # x négyzetgyöke x>0 ha x<0 akkor Error!!!
math.floor( x )# a legnagyobb egész, ami nem nagyobb x-nél
math.sin( x )   # x szög szinusza, a szög radiánban
math.cos( x )   # x szög koszinusza, ahol a szög radiánban adott.
math.radians( x ) # x szög fokban, az eredmény a szög radiánban
math.degrees( x ) # x szög radiánban, az eredmény a szög fokban
```

```
max( x, y, z, .... ) # x,y,z … értékek közül a maximális
min( x, y, z, .... ) # x,y,z … értékek közül a minimális
abs( x )              # x abszolút értéke (nem math modul)
round( x [, n]  )     # x értékét kerekíti n tizedesre
chr(x)                # 0<=x<=255 ASCII karaktert készít
ord(x)               # x karakter ASCII kódját adja vissza
```

Sztring

Szöveg típus, melyet a Pythonban két formában is megadhatunk. 'alma' vagy "alma" mindkettő forma helyes. Több sorból álló szöveg ''' Három darab aposztróffal kezdődő és a végén hasonlóan három darab ''' jellel záródik.

Műveletek sztringekkel:

A szöveg típusú változókkal kapcsolatban több metódus áll rendelkezésre, melyekkel a sztringek kezelését elvégezhetjük. Legyenek a következő változók nevei txt1, txt2 stb.

```
len(txt1)          # megadja a txt1 sztring karaktereinek számát
txt1 in txt2       # ellenőrzi, hogy txt1 benne van e txt2 -ben. True az értéke ha igen.
```

Szövegek „szeletelése".

```
txt1[ 2:5 ]        # txt1 2. pozíciótól 5-ig tartó rész
txt1[ :5 ]         # txt1 elejétől 5-ös pozícióig
txt1[ 2: ]         # txt1 2 pozíciótól a végéig tartó szakasz
```

Szövegek összefűzése. (concatenate)

```
txt1 + txt2        # eredménye a két sztring "összege"
```

Szövegek „módosítását" végző függvények közül pár darab?

```
txt1.upper()       # txt1 minden karakterét nagybetűssé alakítja
txt1.lower()       # txt1 minden karakterét kisbetűssé alakítja
txt1.strip()       # txt1 elejéről/végéről leszedi a szóközöket
txt1.split(',')    # txt1-t széthasítja a paraméterként megadott szeparátoroknál
txt1.replace(txt2,txt3)      # txt1 ben megtalált txt2 sztringet txt3-mal írja felül.
```

Néhány method mellyel egy sztring ellenőrzését végezhetjük

```
txt1.isalpha()    # Ha txt1 alfabetikus karakterekből áll akkor True
txt1.isdigit()    # True, ha minden karakter számjegy
txt1.isprintable() # True, ha minden karakter nyomtatható
txt1.isupper()    # True, ha minden karakter nagybetű
txt1.isspace()    # True, ha minden karakter „whitespace"
```

Logikai

A logikai típusú változók két értéket vehetnek fel: `True` vagy `False`

Több olyan függvény létezik a Python nyelvbe beépítve, melyek logikai értékeket adnak vissza. (pl. sztringek esetében a különböző `isdigit()`, `isalpha()` etc …) Logikai értékkel tér vissza minden olyan művelet, melyek során változók, kifejezések értékeit hasonlítjuk össze. Használjuk a következő változó neveket v1, v2, v3. Tegyük fel azonos típusúak, akkor a következő műveletek eredményei logikai jellegűek lesznek:

v1 == v2 (azonos), v1 < v2 (kisebb), v1 > v2 (nagyobb), v1 != v2 (nem egyenlő), v1 >= v2 (nagyobb egyenlő), v1 <= v2 (kisebb egyenlő) összehasonlítások eredményei `True` vagy `False` lesznek v1 és v2 értéktől függően.

Bitek szintjén értelmezett logikai műveletek:

&	**and**	1 & 1 `True`	0 & 0	0 & 1	1 & 0 esetén `False` értékű a művelet
|	**or**	0 | 0 `False`	0 | 1	1 | 1	1 | 1 esetén `True` értékű a művelet

^ **xor** 0 ^ 0 1 ^ 1 `False` 0 ^ 1 0 ^ 1 esetén `True`
~ **not** ha `True` volt akkor `False` ha `True` volt akkor `False` érték
<< Bitenként balra léptet
>> Bitenként jobbra léptet

Lista

A Python egyik alapvető összetett adatszerkezeti eleme a lista. Listának nevezzük, amit például a C-nyelvben tömbnek vagy vektornak neveznénk. Annyiban különbözik a lista a tömböktől, hogy nemcsak egyféle adattípusból állhat a lista, hanem *tetszőleges típusú elemek vegyesen* szerepelhetnek benne.

Python pl.
lista = [ ] # Üres lista
lista = [1,2,3,4,5] # Numerikus lista
lista = [’a’,’b’,’c’] # Karakterek listája
lista = [’alma’,32,12.0,’b’] # Vegyes lista – karaktersorozat, számok, karakter szerkezete

utolso_elem = lista[-1]

Legyen a listánk a következő,
lista = [1,2,3,4,5,6,7,8,9] # akkor

lista2 = lista[3:5] # eredménye lista2 = [4,5] – az index számolása 0-ról indul !!!

lista1 = [1,2,3,4]
lista2 = [5,6,7,8,9]
lista = lista1 + lista 2 # eredménye a lista = [1,2,3,4,5,6,7,8,9] lesz.

A listákkal elvégezhető műveletek

```
len(lista)                          # - lista elemeinek száma
lista.append( <érték> )             # - új elem felvétele a listába (végére kerül)
lista.insert( <pozíció>, <érték>)   # - új elem beszúrása megadott helyre
lista.extend( <bővitő_lista> )      # - lista bővítése egy másik listával
lista.remove(<érték>)               # – elem törlése értéke alapján
lista.pop()                         # – elem törlése (utolsó elem) törölve
lista.pop(2)                        # egy lista 2-es eleme törölve
lista.index(<n>)                    # hányadik a listában ’n’ indexének lekérdezése
lista.reverse()                      #sorrendjének +fordítása
lista.sort()                        # – lista rendezése növekvő sorrend
lista.sort(reverse=True)            # – lista fordított sorrendben
lista.count(<elem>)                 # adott elem hányszor fordul elő
lista.clear()                       # egy lista elemeinek törlése eredmény üres lista
```

Műveletek

Aritmetikai

A numerikus típusú változókkal elvégezhető műveletek és értelmezésük: Legyenek a =10 és b=2 változók a jelölt értékekkel.

Operátor	Megnevezés	Példa	Eredmény
+	Összeadás	a + b	12
-	Kivonás	a - b	8

*	Szorzás	a * b	20
/	Osztás	a / b	5
%	Osztás maradék	a % b	0
**	Hatványozás	a ** b	100
//	Osztás egész része	a // b	5

Logikai

A logikai kifejezések megfogalmazásánál használható műveleteket az alábbi táblázat tartalmazza.

Relációk	>	Az összehasonlítás eredménye (`True/False`) logikai érték lehet	`x > y`
	<		`x < y`
	==	x azonos y -nal (`True/False`)	`x == y`
	!=	x nem azonos y -nal (`True/False`)	`x != y`
	>=	x nagyobb egyenlő y-nál (`True/False`)	`x >= y`
	<=	x kisebb egyenlő y nal (`True/False`)	`x <= y`
Logikai operátorok	`and`	ha A, B operandusok értéke `True` akkor `True` egyébként `False`	`A and B`
	`or`	Bármely operandus `True` akkor `True`	`A or B`
	`not`	`True` ha x `False` / `False` ha x `True`	`not A`
Lista eleme	`in`		`x in y`
	`not in`		`x not in y`

megjegyzés: gyakori hiba az azonosság jelölésénél, hogy nem a helyes '==' (két egyenlőségjelet használják)

Utasítás

Néhány nagyon egyszerű utasítás, melyek minden programban előfordulnak.

1. Értékadás
   a. <változó> = <érték> # a változónak konkrét tetszőleges adattípus szerinti értéket adunk meg
   b. <változó2> = <változó1> # a változó2 megkapja a korábban már értékadással létrehozott változó1 értékét. Ugyanolyan típusúak lesznek.
   c. <változó> = <függvény() > # A hívott függvény visszaadott értékét kapja a változó
   d. <v1> = <v2> = <v3> = <érték> # mindhárom változó megkapja a jelölt értéket.
2. Adat bevitele billentyűzetről
   a. <változó> = input() # A változó értéke a billentyűzetről adható meg, mindig *szövegtípus* lesz a változó tartalma.
   b. <valtozó> = int (input()) # Ha számot gépelünk be, akkor az egész típusú lesz.
   c. <valtozó> = float (input()) # Ha számot gépelünk be, akkor az valós típusú lesz.
   d. <valtozó> = input(„Prompt szöveg:") # Bevitel elejére kiírja a „Prompt szöveg:"-t
3. Adat megjelenítése képernyőn (konzol)
   a. print(<valtozó>) # valtozó értékét kiírja a képernyőr és 1 sort emel
   b. print(„Szoveg:",<v1>, "\t", <v2>) # Kiírja a Szoveg: szót, v1 értékét elugrik egy tabulátor pozíciót és kiíratja a v2 változó értékét.
   c. '\t' - tabulátor, '\n' soremelést vezérlő karakterek

Döntés (szelekció)

A döntési helyzetek kezelésével lehetséges egy program lefutását szabályozni attól függően, hogy miképpen alakultak különböző változók értékei. Minden programozási nyelv kezeli egy folyamat két/több irányba történő szétválasztását. Python esetében a struktúra a következő: Nagyon fontos a feltételek megadásánál szereplő " **:** " ez a formailag helyes struktúra kialakításához szükséges.

```
if < logikai kifejezés > :   # Ha True, akkor az 1.-es ág hajtódik végre
        # 1. program ág
        …
elif <logikai kif.2 > :      # Ha True, akkor az 2.-es ág hajtódik végre
        # 2. program ág
        …
        …
elif < logikai kif.k > :     # Ha True, akkor az k-dik ág hajtódik végre
        # k. program ág
        …
else  :                      # Egyébként
        #
```

Ciklus (iteráció)

Mondatszerű leírással a ciklusok alapszerkezetei.

SZÁMLÁLÓS CIKLUS (**for** ...)

```
    Ismételd I=1-től N-ig
         utasításblokk
    Ismétlés vége
```

vagy

```
    Ismételd I=N-től 1-ig -1 esével
         utasításblokk
    Ismétlés vége
```

Python nyelvű példa.

```
lista=[1,5,3,2,6,0,4,9,7,8]      #10 elemű számsorozat, tömb
listahossza=len(lista)
''' len() - megadja a lista elemeinek számát
indexük alapján férünk hozzá az elemekhez, kiíratja a lista elemet és
annak a négyzetértékét'''
for i in range(listahossza):
   print('szám: %d, négyzete: %2d' %(lista[i], lista[i]**2))
```

FELTÉTELES CIKLUS (**while** …)

A Pythonban csak "előltesztelős" feltételes ciklus van, nekünk kell létrehozni a "hátultesztelős" ciklust, ha azt akarjuk, hogy legalább egyszer lefusson a ciklusmag, az ismétlendő utasításblokk.

A programozási nyelvek a **while**... valamint a **do...while** *(ismételd...amíg)* vezérlőparancsokat használják. A Pythonban csak a **while...** típus van meg. Ennek a segítségével az iteráció típusának beállítása a következő módon lehetséges

"*Előltesztelős*" ciklus -esetén ismétlődés feltételének vizsgálata ismétlés előtt

```
print('ELŐLTESZTELŐS')
i = int(input('előltesztelős ciklusváltozó kezdőértéke: '))
while i<11:
     print('szám: %2d köbe: %3d' %(i, i**3))
     i += 1
```

"*Hátultesztelős*" ciklus – esetén az ismétlődés feltételének vizsgálata egyszeri végrehajtás után történik meg.

```
print('HÁTULTESZTELŐS')
i = int(input('hátultesztelős ciklusváltozó kezdőértéke: '))
while True:
     print('szám: %2d köbe: %3d' %(i, i**3))
     i += 1
     if i>10:
          break
```

Egy ciklus a `break` utasítására bármikor logikai feltételek vizsgálata nélkül is megszakítható. Ennek következtében a program a ciklust követő első utasítással vagy függvényhívással fog folytatódni.

try … except kezelése

Egy program fejlesztési folyamatában a hibák behatárolása, hiba jellegének megállapítása nagyon fontos lépés. A programozási nyelvek belső szerkezetében vannak ezt a tevékenységet támogató eljárások, változók, melyek értékeinek vizsgálatával egy hiba feltárható és aztán megszüntethető.

```
try:                # próbálja
     x > 3          # a tevékenység, amit ellenőrzünk. Ha nincs gond végrehajtódik és ennek a
```
blokknak a vége után folytatódik a program.
```
except:
     print("Valami rosszul ment")  # Hiba volt
else:
     print("Rendben")      # Lefutott
finally:
     print("A try...except blokk befejezve")  #
```

Ez a programrészlet mutatja, hogy x értékét összehasonlítjuk 3-mal. Ha x nem volt korábban definiálva, akkor a „Valami hiba volt” üzenet jelenik meg, de az egész program futása nem szakad meg. Az x > 3 helyére bármilyen kifejezés, függvény beírható.

Függvény

A függvény egy program, újra felhasználható kód blokkja. amelyet egy kapcsolódó művelet végrehajtására használnak. A függvények használata javítja egy program modularitását és a kódok újra felhasználását eredményezi.

Python nyelv sok beépített függvényt tartalmaz, például `print ().` `input()` stb..

A felhasználó saját függvényeket is létrehozhat. Ezeket a funkciókat felhasználó által definiált függvényeknek nevezzük.

Python nyelvben a függvény létrehozása a következő módon történik

```
def függvény_neve([arg1 [,arg2]]) :   # def kulcsszó után meg kell adni a
```
függvény nevét a és zárójelen belül az opcionálisan felhasználandó külső paramétereket. Lehetséges, hogy egy függvény hívásához nem szükséges semmilyen paraméter, de lehet olyan függvényt is létrehozni, melyben több kívülről átveendő paramétert lehet megadni. A Python nyelv esetében szintaktikai előírás, hogy „**:**” van a def kulcsszó és a függvény neve után.

```
     …
     …      # utasítások, ill. szelekciók, ciklusok , más függvények …
     …
     return
```

példa: egy két paraméteres függvény

```
def osszegzo(a,b) :   #2 paraméteres a függvény
     osszeg = a + b   #
     return osszeg    #  a visszatérési értéket az osszeg nevű változó tartalmazza
```

A Python programozási nyelv elfogadja a függvény rekurzióját is, ami azt jelenti, hogy egy meghatározott függvény hívhatja önmagát. pl.

```
def rekurzio(k):
      if(k > 0):
            result = k + rekurzio(k - 1)
            print(eredmeny)
      else:
            eredmeny = 0
      return eredmeny
```

A **rekurziv** hívás tervezésekor ügyelni kell arra, hogy az algoritmus futásideje ne legyen végtelen, vagy ne kössön le túl sok erőforrást (memória, számolási kapacitást etc…)

File-ok kezelése

Fájlok kezelése minden programozási nyelv szempontjából kiemelt jelentőségű. Egy program szemszögéből a fájlokat két típusba sorolhatjuk. (szöveg illetve bináris)

A fájlokhoz kapcsolódó műveletek fájl létrehozása, fájl tartalmának beolvasása, fájl frissítése, törlése. A fájlokhoz kapcsolódó kulcs műveletek egyike az `open()` vagy nyitási tevékenység. A függvény két paraméterrel rendelkezik. az első a megnyitandó file neve, a második a nyitás módját jelzi.
`open(file_neve, mód)`

A nyitás módjaként két karakterrel adunk jelzést a Python számára:

Első karakter:

`r` olvasásra, hibajelzés van, ha a jelölt fájl nem létezik.
`a` megnyitja a fájlt, „hozzáfűzéssel”, ha nem létezik a fájl, akkor létrehozza.
`w` megnyitja fájlt írásra, ha nem létezik akkor létrehozza.
`x` létrehozza a fájlt, hibaüzenet van, ha már korábban létezett a megadott fájl.

Második karakter:

`t` szöveg típusú a fájl. (alapértelmezett típus)
`b` bináris a fájl.

pl.

```
f = open('minta.txt','wt') # létrehozza a megadott nevű fájlt, ami text típusú lesz.
f = open('minta.txt','a')  # ha létezett a fájl megnyitja és tudjuk módosítani tartalmát,
                           bővithetjük.
f = open('D:\adat\minta.txt', 'r')     # megadott tárhelyen kiválasztott file nyitása
                           olvasásra.
```

Egy szövegfájlból egyetlen sor beolvasása a következő módon zajlik le.

```
f = open("minta.txt", "r") #
print(f.readline())        #  f fájl-ból egy sor beolvas, és kiíratja a print paranccsal.
```

Egy fájl tartalmát sorról sorra beolvassa és kiíratja képernyőre a következő kódrész.

```
f = open("minta.txt", "r")
for x in f :            #mindaddig amíg a fájlon belül EOF előtt.
      print(x)
```

Ha egy (f) fájlt írásra nyitottunk meg, ekkor az `f.write(<változó>)` utasítással az aktuális pozíciónál beírathatjuk a fájlba a <változó> által hordozott tartalmat.

Ha megnyitottunk egy fájlt, beolvastuk a tartalmát, vagy módosítottuk a fájlt a tevékenységet mindig le kell zárni. Erre létezik egy külön parancs. `f.close()`. Ez a parancs csak megnyitott fájlok

esetében működik. Egy alkalmazásból kilépést a fájl zárása művelettel végezzük, mert ezzel aktualizáljuk a fájlt, és adatvesztést előzhetünk meg. (puffer memóriában levő adatok is kiírásra kerülnek!
A fájlok kezeléséhez kapcsolódó egyéb tevékenységek esetén használni kell az `os` modult.

```
import os
os.remove("minta.txt")            #fájl törlése


import os
if os.path.exists("minta.txt"): #létezik a megadott fájl
     os.remove("minta.txt")
else:
     print("A fájl nem létezik")
```

Modul

Modulok szervezésével egy Python környezetben fejlesztett program szerkezete áttekinthetőbbé tehető, valamint kódfejlesztésekben a többször felhasználható kódok kialakítására ösztönöz. Gyorsíthatja a program fejlesztésének folyamatát és a többszörös ellenőrzés/teszt miatt javíthatja egy fejlesztett szoftver minőségét is. A modulok olyan egységek, amelyekben tematikusan összegyűjtve helyezhetünk el különböző állandókat, függvényeket, osztályokat a hozzájuk tartozó metódusokkal egyetemben.

Beépített és saját modul

A Python nyelv több beépített modult tartalmaz. pl. `math, random, datetime`, `turtle,` vagy a `tkinter` (GUI). Bármely modul használatba vétele több módon is lehetséges. A kulcs kifejezés minden esetben az `import`.

```
import math             # a math modult teljes egészében importálja
import math as m        # a math modult m néven veszi használatba
from math import sin    # a math modulból csak a sin() függvény
```

Az átnevezést, akkor célszerű használni, ha túl hosszú esetleg a modul eredeti neve és sok olyan függvényt tartalmaz, amit ténylegesen használnánk. Ezzel ekkor csökkenthetjük a begépelendő forrás kód mennyiségét.

Saját modul létrehozása

Az alábbi pár sorban az az általános struktúra lett felvázolva, amely mutatja egy modul lehetséges kialakítását.

```
# modul neve
#
import math
#
sajat_valtozo1 = 1          # int típusú változó
sajat_valtozó2 = 'szöveg'   # string
#
def sajat_fv():
      …
      return
#
def sajat_fv2(x):
      …
      return
#
#class osztály_neve:        # osztály (saját)
      def __init__ (self,x,y):
            self x = x
            self y = y
      …                     # saját methodusok
      …
#
```

A `sajat_mod.py` néven elmentve ezt a forráskódot már fel is használhatjuk. A modul elemeire egyszerűen hivatkozhatunk programból.

Legyen az applikációnk neve pl. `app.py` amely felhasználja a `sajat_mod.py` -t, akkor az a következő módon fog kinézni.

```
# app.py
import sajat_mod as sm      # importáljuk a sajat_mod -ot sm néven.
#
valt1 = 'xxx'               # valt1 értéket kap
sm.sajat_fv2(valt1)         # meghívjuk a sajat_fv2-t a valt1
argumentummal.
#
#
```

Class

Az osztály adattípus, amely egy adott jellegű objektum sablondefiníciójaként szolgál. Pythonban az osztály `class` kulcsszóval és egy név megadásával hozható létre:

```
class osztály_neve :
```

Amikor objektumot hozunk létre egy osztályból, ezt az osztály *példányosításának* nevezzük. A program ilyenkor az operációs rendszeren keresztül biztosít memóriát egy objektum létrehozásához ezzel a sablonnal (az osztállyal) és a hozzátartozó kiindulási értékekkel. A példányosítás tulajdonképpen létrehozást jelent.

Egy objektum példányosításához zárójelekkel kell kiegészítenie az osztály nevét. Ilyenkor egy objektumot kap, amelyet egy változóhoz is hozzárendelhet az alábbi módon:

```
uj_obj = osztaly_neve()
```

Változók az objektumorientált programozásban is vannak, de itt nem önmagukban vannak definiálva, hanem objektumokhoz vannak *csatolva*. Egy objektum változóira *attribútumokként* lehet hivatkozni. Egy objektumhoz csatolt attribútum két célra szolgálhat:

- **Az objektum leírása**: Egy *leíró* változóra példa lehet egy kocka élhossza, kör sugara.
- **Az állapot ábrázolása**: Egy változó egy objektum állapotának leírására is használható. Az állapotra példa mondjuk egy kapcsoló esetében az, hogy ON vagy OFF állapotban van.

Konstruktor fogalma: az egy speciális függvény, amely csak az objektum első létrehozásakor van meghívva. A konstruktor csak egyszer lesz meghívva. Ebben a metódusban hozhatja létre az attribútumokat, amelyekkel az objektumnak rendelkeznie kell. Emellett kezdőértékeket is rendelhet a létrehozott attribútumokhoz.

A Pythonban a konstruktor neve `__init()__`. Paraméterként a speciális `self` kulcsszót is át kell adnia a konstruktornak. A `self` kulcsszó az objektumpéldányra vonatkozik. Az ehhez a kulcsszóhoz történő hozzárendelés azt jelenti, hogy az attribútum az objektumpéldányé lesz. Ha nem teszi hozzá a `self` attribútumot, akkor az ideiglenes változóként lesz kezelve, amely az `__init()__` végrehajtásának befejezése után nem létezik többé.

Létrehozás

```
class Kocka:          # Kocka nevű osztály dekl.
     def __init__(self, kocka_el):  # kezdő paraméter a kocka él-
hossza)
        self.make = "KOCKA"
        self.a_old = kocka_el

     def k_elh(self):
        return self.a_old * 12
```

Az osztálydefiníciók nem lehetnek üresek, de ha valamilyen oknál fogva mégis tartalom nélküli osztálydefinícióval rendelkezik, akkor használjuk a `pass` utasítást a hiba elkerülése érdekében.

pl.

```
class Pont:
      pass
```

Metódusok az objektumorientált programozásban

A metódusok a használt paradigmától függetlenül egy műveletet hajtanak végre. Ez a művelet lehet egy csak a bemenet alapján végzett számítás, vagy megváltoztathatja egy változó értékét.

Az objektumok metódusainak az OOP-ben két változata van:

- Más objektumok által hívható **külső metódusok**.
- Más objektumok számára nem elérhető **belső metódusok**. Emellett az ilyen metódusok közreműködnek az egy külső metódus hívásával megkezdett tevékenységek végrehajtásában is.

A metódusok a típusuktól függetlenül megváltoztathatják egy objektum egy attribútumának értékét, más szóval az objektum *állapotát*.

Hozzáférési szintek

Hogyan védhetők az osztályok és objektumok a nem kívánt adatmanipulációval szemben? A válasz az, hogy *hozzáférési szintekkel*. Az adatok úgy rejthetők el a külvilág és a többi objektum elől, hogy különleges kulcsszavakkal jelöljük meg az adatokat és függvényeket. Ezek a kulcsszavak az úgynevezett hozzáférés-módosítók.

A Python az attribútumnevekhez fűzött előtagokkal oldja meg az adatok elrejtését. Egy kezdő aláhúzásjel, _ azt jelzi a külvilág számára, hogy ezt az adatot nem tanácsos változtatni.

Egy kezdő aláhúzásjel még megengedi az adat módosítását. Az ilyen adatot a Pythonban *védettnek* nevezzük. Ha két kezdő aláhúzást, használunk __, így az adat *privát* lesz

Az adatvédelmet implementáló nyelvek más-más módon kezelik ezt a problémát. A Python sajátossága, hogy az adatvédelem inkább különböző szintű *ajánlásokból áll*, és nincs szigorúan implementálva.

Minta programok, példák

Tömbök buborék rendezése (bubblesort)

```
# Python 3.x
# Buborék rendezés || Bubble sort
#
def bubbleSort(arr):
      n = len(arr)      # az arr tömb elemszáma
      # Az összes elemre
      for i in range(n):
            swapped = False
            # Utolsó i elem a helyén van
            for j in range(0, n-i-1):
                  # j mozogjon 0-tól n-i-1 ig.
                  # Cserélje meg, a tömb 2 elemét,ha
                  # a talált elem nagyobb, mint a következő elem.
                  if arr[j] > arr[j+1] :
                        arr[j], arr[j+1] = arr[j+1], arr[j]  # !! csere !!
                        swapped = True

            # Ha két elemet nem cserélt fel a belső hurok,
            # akkor szakítsa meg a ciklust
            if swapped == False:
                  break

# Teszt tömb
arr = [64,-5, 34, 25,-65, 12, 22, 11, 1,90,211]

bubbleSort(arr)                     # az arr tömbre használjuk

print ("Rendezett tömb :")          # eredmény kiíratása
for i in range(len(arr)):
      print ("%d" %arr[i],end=" ") # end=" " -- egymás mellé írat!!
```

Fájl generálása

A következő minta program egy fájl létrehozását. Adattartalma a véletlenszám generáláshoz kötött. A program elején beolvastatjuk a generálandó adatok számát, pontosabban a fájl sorainak számát. A véletlenszám (egész) legkisebb és a legnagyobb értékét.

A fájl egy sora úgy egy négyelemű listaként fogható fel. Az első elem egy sorszám (1-től), a második egy valós (`float`) a harmadik egész típusú, a negyedik ismét valós értéket hordoz.

Külön kiemelendő a program futásának idejének mérése. A datetime modul egyik függvényének felhasználásával.

```
# Fájl létrehozása, program futási idejének mérése

import random          #véletlen számok kezelése
import datetime        #dátum / idő

mennyi = int( input("Hány tesztadatot generáljunk? "))
```

```
alap = int( input("Mennyi a tesztadat alapja? "))
maximum = int(input("Mennyi a maximális érték? "))

t = [0,1,2,3]
f= open("adat_file.txt","w+")    # w -- ha nem létezett akkor hozza létre | létezett akkor ürit.
                                 # 0 méretű fájlként nyílik meg
x = datetime.datetime.now()      # MOST - OS rendszeridő
i=1
#
#
for i in range(1,mennyi+1,1):    # tesztadatszám
    for j in range(1,3,1):       # tömb feltöltése
        t[0]=i
        t[1]=float(i+0.1+random.random())       # valós float, adat növekvő
        t[2]=int(alap+maximum*random.random())  # csak az egész rész van figyelembevéve !!!!
        t[3]=float(alap+maximum*random.random()) # valós számérték !!!!

        szoveg="{:d};{:.3f};{};{:.4f}\n"        # formázás a kiírásra kerülő adatoknál

    f.write(szoveg.format(t[0],t[1],t[2],t[3]))  # a szoveg formátummal írja ki a fileba!
#
# Generálás vége
#
y = datetime.datetime.now()
print("Kezdés:   ",x)
print("Befejezés:",y)

f.close()       # fájl bezárása
```

A program lefuttatása után vizsgáljuk meg az `adat_file.txt` fájl tartalmát! (jegyzettömb-bel megnyitva) Érdemes vizsgálni 10, 100, 1000000 sorból álló minta létrehozását!

File konverzió

A következő minta program az `input_f.txt` nevű fájl tartalmát beolvasva átalakítja és az átalakítás (konverzió) eredményét az `output_f.txt` nevű fájlba írja ki.

Az `input_f.txt` minta tartalma (három sor, számokkal és a számokat ’,’ szeparálja.

1,9,3,4,5,6,7
3,9,5,6,7,8,9
5,7,8,9,0,1,2

A konverzió ebben az esetben a tartalom soronként növekvő kiírását jelenti az `output_f.txt` fájlba.

1,3,4,5,6,7,9
3,5,6,7,8,9,9
0,1,2,5,7,8,9

```
#File-konverzió
#
#
def konverzio(tomb):
  #
  #tomb.reverse()        #egyszerűen tömböt forditott sorrendbe kiíratná
  tomb.sort()            #rendezetten ir ki
  return tomb
#
#
#
f=open("input_f.txt","r")   #nyitva olvasásra
g=open("output_f.txt","w")  #nyitjuk feltöltésre, üresen létrehozza.

for x in f:       #file sorrol sorra beolvasva
  print("Input ", x)

  tmp=x.split()   #tmp átmeneti tárolóba beolvas,  1 elemű tömb "\n" -
nélkül
  print("tmp = ", tmp)

  adat=tmp[0].rsplit(",")
  print("adat", adat)  #adat tömbbe szétszedte a tmp-t
  print()

  for i in range(0,7,1):   #pl. egy sor legyen pl. 7 elemű tömb
      adat[i]=int(adat[i])

  #tevékenység a tömbbel
  konverzio(adat)          # a soronkénti konverzió tényleges meghívás ---
konverzio(tomb) függvény!  Feladatonként más
  print("konvertalt adat:",adat,"\n\n")     #eredmény kiíratása képernyőre

szoveg=str(adat[0])+","+str(adat[1])+","+str(adat[2])+","+str(adat[3])+","+
str(adat[4])+","+str(adat[5])+","+str(adat[6])+"\n"
  g.write(szoveg)           #kirás fileba

f.close()                   #file-ok zárása
g.close()
```

A turtle (teknőc) modul használata

Néhány egyszerű minta kód mutatja, miként lehet konzol üzemmódban is 'grafikus' jellegű adatmegjelenítést, ábrákat létrehozni. A Python IDLE-ben elérhető turtle examples programok forrásszinten adnak egyszerű betekintést a modul használatára.

```
#Csillag rajzolása
import turtle

star = turtle.Turtle()
```

```
for i in range(5):
      star.forward(150)       #előre 150 egység
      star.right(144)         #fordulj jobbra 144 fokot

turtle.done()


#Szabályos hatszög rajzolása
#
import turtle      # importáljuk a turtle (teknőc) nevű modul
poligon = turtle.Turtle()    # generál egy poligon nevű objektumot
oldal_szam = 6               # oldalak száma
oldal_hossz = 70             # oldal hossza
szog = 360.0 / oldal_szam    # szög
for i in range(oldal_szam):
      poligon.forward(oldal_hossz) # előre oldal hossznyit
      poligon.right(szog)          # fordulj jobbra
turtle.done()


from turtle import *

reset()
goto(100,100)
a=0
width(2)
while a<=15 :
    a=a+1
    forward(150)
    left(150)
    if a%2 ==0:
        color("red")
    else:
        color("blue")
    width(a)
```

3 dimenziós vektorok – saját modul

Ebben a modulban az O (0,0,0) origóból a P (x,y,z) pontba mutató vektorok kezelésére lett összeállítva néhány függvény. A függvények segítségével a V vektor hosszát, n= valós/egész esetén a V vektor n-szeresét vagy V1 V2 használatával vektorok összegét, különbségét, skalárszorzatát vagy vektoriális szorzatát tudjuk meghatározni. (matematika, fizika feladatok megoldásához használható)

```
import math
#
def vektor_pm(v1,v2,x):  # 3 dimenziós vektorok v1, v2 összege, különbsége
    vektor=[0,0,0]
    if x=="+":
        vektor[0]=v1[0]+v2[0]
        vektor[1]=v1[1]+v2[1]
        vektor[2]=v1[2]+v2[2]
    elif x =="-":
        vektor[0]=v1[0]-v2[0]
        vektor[1]=v1[1]-v2[1]
        vektor[2]=v1[2]-v2[2]
    return vektor

def vektor_szor (v1, szorzo):  # vektor nyújtása, zsugorítása
    vektor=[0,0,0]
    vektor[0]=v1[0] * szorzo
    vektor[1]=v1[1] * szorzo
    vektor[2]=v1[2] * szorzo
    return vektor

def vektor_skszor(v1,v2):      # 2 vektor skaláris szorzata
    ssz = v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2]
    return ssz

def vektor_hossz(v):           # vektor hossza
    hossz=0
    hossz = math.sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2])
    #hossz = math.sqrt(v_mod.vektor_skszor(v))
    return hossz

def vektor_vek_szorz(v1,v2):   # 2 vektor vektoriális szorzata  !!! 3
dimenzió
    vektor=[0,0,0]
    vektor[0] = v1[1]*v2[2] - v1[2]*v2[1]
    vektor[1] = v1[2]*v2[0] - v1[0]*v2[2]
    vektor[2] = v1[0]*v2[1] - v1[1]*v2[0]
    return vektor

def vektor_vektor_szog(v1,v2): # két vektor által bezárt szög
    a=v_mod.vektor_hossz(v1)
    b=v_mod.vektor_hossz(v2)
    c=v_mod.vektor_skszor(v1,v2)
    szog = math.acos(c/(a*b))
```

```
    return szog                 #radiánban !!!
```

Geometriai objektumok – minta class

Objektumok definiálását és a hozzájuk kapcsolódó metódusok szerkesztését egy geometriai feladaton keresztül láthatjuk. Kocka, kör, gyűrű, henger, gömb tulajdonságai vannak összegyűjtve a teljesség igénye nélkül. Ilyenek, mint kör/gömb sugara alapján a kerület, terület vagy gömb felszíne, térfogata. etc… Ezen objektumok használatával néhány matematikai, fizikai vagy egyszerűbb műszaki feladat kiszámolására láthatunk mintát.

```
import math
#
#
class Kocka:          # Kocka nevű osztály dekl.
    def __init__(self, kocka_el):  # inicializálás (kezdő paraméter a
oldal)
        self.make = "KOCKA"
        self.a_old = kocka_el

    def k_elh(self):
        return self.a_old * 12

    def k_fel(self):                  # kocka felülete
        return (self.a_old **2) * 6

    def k_terf(self):                 # kocka térfogat
        return self.a_old ** 3

    def k_tom(self, suruseg):         # tomeg  m = sűrűség * térfogat
        return self.k_terf()*suruseg
#
#
class Hasab:                         # hasáb 90 fok a,b,c oldalakkal
    def __init__(self, a_old, b_old, c_old):
        self.make = "Hasáb"
        self.a_old = a_old
        self.b_old = b_old
        self.c_old = c_old

    def h_fel(self):                # hasáb felszíne
        return (self.a_old * self.b_old)*2 + (self.a_old * self.c_old)*2 +
(self.c_old * self.b_old)*2

    def h_terf(self):               # hasáb térfogata
        return self.a_old * self.b_old * self.c_old
#
class Kor:            # Középpont (0,0)kör/(0,0,0)gömb sugar paraméterrel
    def __init__(self,sugar):
        self.make = "Kör"
        self.sugar = sugar

    def kor_pont(self, x, y):   # 0,0 középpontú self.sugarú kör és pont
viszonya el
```

```
        pont = x**2 + y**2
        if (pont - self.sugar **2)== 0 :
            vissza = 0                          # Körvonalon
        elif pont - self.sugar **2 > 0 :
            vissza = 1                          # Körvonalon kívül
        else:
            vissza = 2                          # Körvonalon belül
        return vissza

    def kor_ker(self):
        return 2*self.sugar * math.pi        # kör kerülete

    def kor_ter(self):                          # kör területe
        return math.pi * self.sugar**2

    def gom_fel(self):                          # gömb felszíne
        return 4*math.pi * self.sugar

    def gom_terf(self):                         # gömb térfogata
        return (4 * math.pi * self.sugar ** 3)/3

    def gom_pont(self,x,y,z):
        pont = x**2 + y**2 + z**2
        if (pont - self.sugar **2) == 0:
            vissza = 0                          # Gömbfelszínen
        elif pont - self.sugar **2 > 0:
            vissza = 1                          # Gömbön kívül
        else:
            vissza = 2                          # Gömbfelszínen belül
        return vissza

    def gom_tom(self,suruseg):      # Gömb tömege m  = terfogat* sűrűség
        return self.gom_terf()*suruseg

    def gyuru_f(self, other_ring):       # két koncentrikus kör által
meghatározott gyűrű felülete
        f = self.kor_ter() - other_ring.kor_ter()
        return math.fabs(f)

    def gyuru_k(self, other_ring):       # két koncentrikus kör által
meghatározott gyűrű kerülete
        return math.fabs(self.kor_ker() + other_ring.kor_ker())

    def henger_v(self, other_ring,h):        # henger térfogata
        return self.gyuru_f(other_ring)*h

    def henger_vm(self,other_ring,h,ro):     # henger tömege
        return self.henger_v(other_ring,h) * ro
```

Hogyan használhatjuk ezeket az osztályokat programunkban. pl. ha az itt leírt forrást beépítettük a programba, akkor nagyon egyszerű alkalmazni ezeket.

#

```
#
koceka = Kocka(2.15)
type(koceka)      #létrejön egy koceka nevű Kocka 2.15 egység élhosszal
print("Kocka térfogata",koceka.k_terf()) # Kiszámoltatjuk a térfogatát
print("Kocka felszíne",koceka.k_fel())   # A felszínét
print("Kocka tömege: ",koceka.k_tom(1.2)) # és 1.2-nyi sűrűséggel a tömegét
a kockának
```

Segédlet

A Google szolgáltatásai közül jól használható a programok tervezéséhez, dokumentálásához, programleírások készítésére a Untitled Diagram - diagrams.net webhelyen található applikáció. Folyamatábrák, UML dokumentumok mellett adatbázistervezéshez is használható grafikus sablonok állnak rendelkezésre. A produktum kimenetként különböző kép-formátumokban áll rendelkezésre, az eredmény letölthető.

Források

Internet:

- 3.9.3 Documentation (python.org)
- Python Tutorial (w3schools.com)
- Python 3 Tutorial - Tutorialspoint
- Turtle Programming in Python - GeeksforGeeks
- Turtle programming in Python (tutorialspoint.com)
- Mit jelent az objektumorientált programozás? - Learn | Microsoft Docs

Tartalom