



Európai Unió
Európai Szociális
Alap



BEFEKTETÉS A JÖVŐBE

SZÉCHENYI  2020

Hogyan gondolkozz úgy, mint egy informatikus: Tanulás Python 3 segítségével

3. kiadás

**Peter Wentworth, Jeffrey Elkner,
Allen B. Downey and Chris Meyers**

2019. március 19.

Tartalomjegyzék

Copyright / Szerzői jogi megjegyzés	2
Előszó a magyar fordításhoz	3
Előszó	4
Bevezetés	6
A Rhodes helyi kiadása (Rhodes Local Edition - RLE) (2012. augusztusi verzió)	10
Közreműködői lista	13
1. A programozás mikéntje	16
1.1. A Python programozási nyelv	16
1.2. Mi egy program?	18
1.3. Mi a nyomkövetés?	18
1.4. Szintaktikai hibák	18
1.5. Futási idejű hibák	18
1.6. Szemantikai hibák	19
1.7. Kísérleti nyomkövetés	19
1.8. Formális és természetes nyelvek	19
1.9. Az első program	21
1.10. Megjegyzések	21
1.11. Szójegyzék	21
1.12. Feladatok	22
2. Változók, kifejezések, utasítások	24
2.1. Értékek és típusok	24
2.2. Változók	26
2.3. Változónevek, kulcsszavak	27
2.4. Utasítások	28
2.5. Kifejezések kiértékelése	28
2.6. Műveleti jelek és operandusok	29
2.7. Típuskonverziós függvények	29
2.8. Műveletek kiértékelési sorrendje	30
2.9. Sztringkezelő műveletek	31
2.10. Adatbekérés	31
2.11. Függvények egymásba ágyazása	32
2.12. A maradékos osztás művelet	33
2.13. Szójegyzék	33
2.14. Feladatok	34
3. Helló, kis teknőcök!	36

3.1.	Az első teknőc programunk	36
3.2.	Példányok – teknőcök hada	38
3.3.	A for ciklus	40
3.4.	A for ciklus végrehajtási sorrendje	41
3.5.	A ciklus egyszerűsíti a teknőc programunkat	42
3.6.	További teknőc metódusok és trükkök	43
3.7.	Szójegyzék	45
3.8.	Feladatok	45
4.	Függvények	48
4.1.	Függvények	48
4.2.	A függvények is hívhatnak függvényeket	50
4.3.	A programvezérlés	51
4.4.	Paraméterekkel rendelkező függvények	54
4.5.	Visszatérési értékkel rendelkező függvények	54
4.6.	A változók és a paraméterek lokálisak	56
4.7.	Teknőc revízió	56
4.8.	Szójegyzék	57
4.9.	Feladatok	58
5.	Feltételes utasítások	61
5.1.	Boolean értékek és kifejezések	61
5.2.	Logikai operátorok	62
5.3.	Igazságtáblák	62
5.4.	Boolean kifejezések egyszerűsítése	63
5.5.	Feltételes végrehajtás	63
5.6.	Az else ág kihagyása	65
5.7.	Láncolt feltételes utasítások	65
5.8.	Beágyazott feltételes utasítások	66
5.9.	A return utasítás	67
5.10.	Logikai ellentétek	67
5.11.	Típuskonverzió	69
5.12.	Egy teknőc oszlopdiagram	70
5.13.	Szójegyzék	72
5.14.	Feladatok	73
6.	Produkzív függvények	76
6.1.	Visszatérési érték	76
6.2.	Programfejlesztés	77
6.3.	Nyomkövetés a print utasítással	80
6.4.	Függvények egymásba ágyazása	80
6.5.	Logikai függvények	81
6.6.	Stílusos programozás	82
6.7.	Egységteszt	82
6.8.	Szójegyzék	84
6.9.	Feladatok	85
7.	Iteráció	89
7.1.	Értékadás	89
7.2.	A változók frissítése	90
7.3.	A for ciklus újra	90
7.4.	A while utasítás	91
7.5.	A Collatz-sorozat	92
7.6.	A program nyomkövetése	93
7.7.	Számjegyek számlálása	94

7.8.	Rövidített értékadás	95
7.9.	Súgó és meta-jelölés	96
7.10.	Táblázatok	97
7.11.	Kétdimenziós táblázat	98
7.12.	Enkapszuláció és általánosítás	98
7.13.	Még több enkapszuláció	99
7.14.	Lokális változó	99
7.15.	A <code>break</code> utasítás	100
7.16.	Más típusú ciklusok	101
7.17.	Egy példa	102
7.18.	A <code>continue</code> utasítás	103
7.19.	Még több általánosítás	104
7.20.	Függvények	105
7.21.	Értékpár	105
7.22.	Beágyazott ciklus beágyazott adatokhoz	106
7.23.	Newton módszer a négyzetgyök megtalálásához	107
7.24.	Algoritmusok	108
7.25.	Szójegyzék	108
7.26.	Feladatok	110
8.	Sztringek	113
8.1.	Összetett adattípusok	113
8.2.	Sztringek kezelése egy egységként	113
8.3.	Sztringek kezelése részenként	115
8.4.	Hossz	116
8.5.	Bejárás és a <code>for</code>	117
8.6.	Szeletelés	118
8.7.	Sztringek összehasonlítása	118
8.8.	A sztringek módosíthatatlanok	119
8.9.	Az <code>in</code> és a <code>not in</code> operátor	120
8.10.	Egy kereses függvény	120
8.11.	Számlálás ciklussal	121
8.12.	Opcionális paraméterek	121
8.13.	A beépített <code>find</code> metódus	122
8.14.	A <code>split</code> metódus	123
8.15.	A sztringek tisztítása	123
8.16.	A sztring <code>format</code> metódusa	124
8.17.	Összefoglalás	127
8.18.	Szójegyzék	128
8.19.	Feladatok	128
9.	Rendezett n-esek	132
9.1.	Adatsoportosításra használt rendezett n-esek	132
9.2.	Értékadás rendezett n-esel	133
9.3.	Rendezett n-es visszatérési értéként	134
9.4.	Adatszerkezetek alakíthatósága	134
9.5.	Szójegyzék	135
9.6.	Feladatok	135
10.	Eseményvezérelt programozás	136
10.1.	Billentyű leütés események	136
10.2.	Egér események	137
10.3.	Időzített, automatikus események	138
10.4.	Egy példa: állapotautomata	139

10.5. Szójegyzék	141
10.6. Feladatok	142
11. Listák	144
11.1. A lista értékei	144
11.2. Elemek elérése	145
11.3. A lista hossza	145
11.4. Lista tagság	146
11.5. Lista műveletek	147
11.6. Lista szeletek	147
11.7. A listák módosíthatók	147
11.8. Lista törlése	148
11.9. Objektumok és hivatkozások	149
11.10. Fedőnevek	150
11.11. Listák klónozása	150
11.12. Listák és a <code>for</code> ciklus	151
11.13. Lista paraméterek	152
11.14. Lista metódusok	153
11.15. Tiszta függvények és módosítók	155
11.16. Listákat előállító függvények	156
11.17. Sztringek és listák	156
11.18. A <code>list</code> és a <code>range</code>	157
11.19. Beágyazott listák	158
11.20. Mátrixok	158
11.21. Szójegyzék	159
11.22. Feladatok	160
12. Modulok	162
12.1. Véletlen számok	162
12.2. A <code>time</code> modul	165
12.3. A <code>math</code> modul	165
12.4. Saját modul létrehozása	166
12.5. Névterek	166
12.6. Hatókör és keresési szabályok	168
12.7. Attribútumok és a pont operátor	169
12.8. Az <code>import</code> utasítás három változata	169
12.9. Az egységesztelődöt alakítsd modullá	170
12.10. Szójegyzék	170
12.11. Feladatok	171
13. Fájlok	175
13.1. Fájlokról	175
13.2. Első fájlunk írása	175
13.3. Fájl soronkénti olvasása	176
13.4. Fájl átalakítása sorok listájává	177
13.5. A teljes fájl beolvasása	177
13.6. Bináris fájlok kezelése	178
13.7. Egy példa	178
13.8. Könyvtárak	179
13.9. Mi a helyzet az internetről való letöltéssel?	180
13.10. Szójegyzék	180
13.11. Feladatok	181
14. Lista algoritmusok	182
14.1. Tesztvezérelt fejlesztés	182

14.2.	A teljes keresés algoritmus	182
14.3.	Egy valós probléma	183
14.4.	Bináris keresés	186
14.5.	A szomszédos duplikátumok eltávolítása	189
14.6.	Sorbarendezett listák összefűzése	190
14.7.	Alice Csodaországban, ismét!	191
14.8.	Nyolc királynő probléma, első rész	193
14.9.	Nyolc királynő probléma, második rész	196
14.10.	Szójegyzék	197
14.11.	Feladatok	198
15.	Osztályok és objektumok – alapok	200
15.1.	Objektorientált programozás	200
15.2.	Saját, összetett adattípusok	200
15.3.	Attribútumok	202
15.4.	Az inicializáló metódus továbbfejlesztése	202
15.5.	Újabb metódusok hozzáadása az osztályunkhoz	204
15.6.	Példányok felhasználása argumentumként és paraméterként	205
15.7.	Egy példány átalakítása sztringgé	205
15.8.	Példányok, mint visszatérési értékek	206
15.9.	Szemléltetváltás	207
15.10.	Az objektumoknak lehetnek állapotai	207
15.11.	Szójegyzék	208
15.12.	Feladatok	208
16.	Osztályok és objektumok – ássunk egy kicsit mélyebbre	210
16.1.	Téglalapok	210
16.2.	Az objektumok módosíthatók	211
16.3.	Azonosság	212
16.4.	Másolás	213
16.5.	Szójegyzék	214
16.6.	Feladatok	214
17.	PyGame	216
17.1.	A játék főciklusa	216
17.2.	Képek és szövegek megjelenítése	219
17.3.	Tábla rajzolása az N királynő problémához	221
17.4.	Sprite-ok	225
17.5.	Események	228
17.6.	Egy integetős animáció	230
17.7.	Alienek – esettanulmány	233
17.8.	Mérlegelés	234
17.9.	Szójegyzék	234
17.10.	Feladatok	234
18.	Rekurzió	236
18.1.	Fraktálok rajzolása	236
18.2.	Rekurzív adatszerkezetek	238
18.3.	Listák rekurzív feldolgozása	239
18.4.	Esettanulmány: Fibbonacci-számok	240
18.5.	Példa a rekurzív könyvtárakra és fájlokra	241
18.6.	Animált fraktál, PyGame használatával	242
18.7.	Szójegyzék	244
18.8.	Feladatok	245

19. Kivételek	248
19.1. Kivételek elkapása	248
19.2. Saját kivételek létrehozása	249
19.3. Egy korábbi példa áttekintése	250
19.4. A <code>finally</code> ág és a <code>try</code> utasítás	251
19.5. Szójegyzék	252
19.6. Feladatok	252
20. Szótárak	253
20.1. Szótár műveletek	254
20.2. Szótár metódusok	255
20.3. Fedőnevek és másolás	256
20.4. Ritka mátrixok	257
20.5. Memoizálás (a feljegyzéses módszer)	258
20.6. Betűk számlálása	259
20.7. Szójegyzék	259
20.8. Feladatok	260
21. Esettanulmány: A fájlok indexelése	262
21.1. A kereső program	262
21.2. A szótár lemezre mentése	264
21.3. A lekérdező (Query) program	265
21.4. A szerializált szótár tömörítése	266
21.5. Szójegyzék	266
22. Még több OOP	268
22.1. Az <code>Ido</code> osztály	268
22.2. Tiszta függvények	268
22.3. Módosító függvények	269
22.4. Alakítsuk át a <code>novel</code> függvényt metódussá	270
22.5. Egy „aha-élmény”	271
22.6. Általánosítás	272
22.7. Egy másik példa	273
22.8. Operátorok túlterhelése	274
22.9. Polimorfizmus	275
22.10. Szójegyzék	277
22.11. Feladatok	277
23. Objektumok kollekciója	279
23.1. Kompozíció	279
23.2. <code>Kartya</code> objektumok	279
23.3. Osztály attribútumok és az <code>__str__</code> metódus	280
23.4. Kártyák összehasonlítása	281
23.5. Paklik	282
23.6. A pakli kiírása	283
23.7. Pakli keverés	284
23.8. Osztás és a kártyák eltávolítása	285
23.9. Szójegyzék	285
23.10. Feladatok	286
24. Öröklődés	287
24.1. Öröklődés	287
24.2. A kézben tartott lapok	287
24.3. Osztás	288
24.4. A kézben lévő lapok megjelenítése	289

24.5.	A <code>KartyaJatek</code> osztály	290
24.6.	<code>FeketePeterKez</code> osztály	290
24.7.	<code>FeketePeterJatek</code> osztály	292
24.8.	Szójegyzék	294
24.9.	Feladatok	295
25.	Láncolt listák	296
25.1.	Beágyazott referenciák	296
25.2.	A <code>Csomopont</code> osztály	296
25.3.	Listák kollekcióként	297
25.4.	Listák és a rekurzió	298
25.5.	Végtelen listák	299
25.6.	Az alapvető félreérthetőség tétele	299
25.7.	A listák módosítása	300
25.8.	Csomagolók és segítők	301
25.9.	A <code>LancoltLista</code> osztály	301
25.10.	Invariánsok	302
25.11.	Szójegyzék	303
25.12.	Feladatok	303
26.	Verem	304
26.1.	Absztrakt adattípusok	304
26.2.	A verem AAT	304
26.3.	Verem implementációja Python listákkal	305
26.4.	Push és pop	305
26.5.	Verem használata posztfix kifejezés kiértékeléséhez	306
26.6.	Nyelvtani elemzés	306
26.7.	Posztfix kiértékelés	307
26.8.	Kliensek és szolgáltatók	307
26.9.	Szójegyzék	308
26.10.	Feladatok	308
27.	Sorok	309
27.1.	A sor AAT	309
27.2.	Láncolt sor	309
27.3.	Teljesítmény jellemzők	310
27.4.	Javított láncolt sor	310
27.5.	Prioritásos sor	311
27.6.	A <code>Golfozo</code> osztály	313
27.7.	Szójegyzék	313
27.8.	Feladatok	314
28.	Fák	315
28.1.	Fák építése	316
28.2.	A fák bejárása	316
28.3.	Kifejezésfák	316
28.4.	Fabejárás	317
28.5.	Kifejezésfák felépítése	318
28.6.	Hibák kezelése	321
28.7.	Az állati fa	322
28.8.	Szójegyzék	324
28.9.	Feladatok	324
A.	Nyomkövetés	326
A.1.	Szintaktikai hibák	326

A.2.	Nem tudom futtatni a programomat, akármit is csinállok	327
A.3.	Futási idejű hibák	327
A.4.	A program abszolút semmit nem csinál	327
A.5.	A programom felfüggesztődött	327
A.6.	Végtelen ciklus	328
A.7.	Végtelen rekurzió	328
A.8.	A végrehajtás menete	329
A.9.	Amikor futtatom a programom, egy kivételt kapok	329
A.10.	Olyan sok <code>print</code> utasítást adtam meg, hogy eláraszt a kimenet	330
A.11.	Szemantikai hibák	330
A.12.	A programom nem működik	330
A.13.	Van egy nagy bonyolult kifejezés és nem azt csinálja, amit elvárok	331
A.14.	Van egy függvény vagy módszer, amely nem az elvárt értékkel tér vissza	332
A.15.	Nagyon, nagyon elakadtam, és segítségre van szükségem	332
A.16.	Nem, tényleg segítségre van szükségem	332
B.	Egy apró-cseprő munkafüzet	334
B.1.	A jártasság öt fonala	334
B.2.	E-mail küldés	335
B.3.	Írd meg a saját webszerveredet!	336
B.4.	Egy adatbázis használata	337
C.	Ay Ubuntu konfigurálása Python fejlesztéshez	340
C.1.	Vim	340
C.2.	<code>\$HOME</code> környezet	341
C.3.	Bárhonnan végrehajtható és futtatható Python szkript létrehozása	341
D.	A könyv testreszabása és a könyvhöz való hozzájárulás módja	342
D.1.	A forrás megszerzése	342
D.2.	A HTML verzió elkészítése	343
E.	Néhány tipp, trükk és gyakori hiba	344
E.1.	Függvények	344
E.2.	Sztring kezelés	348
E.3.	Ciklusok és listák	349
F.	GNU Free Documentation License	350
F.1.	0. PREAMBLE	350
F.2.	1. APPLICABILITY AND DEFINITIONS	350
F.3.	2. VERBATIM COPYING	351
F.4.	3. COPYING IN QUANTITY	352
F.5.	4. MODIFICATIONS	352
F.6.	5. COMBINING DOCUMENTS	353
F.7.	6. COLLECTIONS OF DOCUMENTS	354
F.8.	7. AGGREGATION WITH INDEPENDENT WORKS	354
F.9.	8. TRANSLATION	354
F.10.	9. TERMINATION	354
F.11.	10. FUTURE REVISIONS OF THIS LICENSE	355
F.12.	11. RELICENSING	355
F.13.	ADDENDUM: How to use this License for your documents	355



A fordítás alapjául szolgáló mű:

Peter Wentworth, Jeffrey Elkner, Allen B. Downey és Chris Meyers: *How to Think Like a Computer Scientist: learning with Python*, 2012 október

(mely a Jeffrey Elkner, Allen B. Downey és Chris Meyers által jegyzett második kiadáson alapul)

Fordította: Biró Piroska, Szeghalmy Szilvia és Varga Imre
Debreceni Egyetem, Informatikai Kar

A fordítás az „Az MTMI szakokra való bekerülést elősegítő innovatív programok megvalósítása a Debreceni Egyetem vonzáskörzetében” című EFOP-3.4.4-16-2017-00023 azonosítójú pályázat keretében valósult meg.

Kapcsolattartó szerző: p.wentworth@ru.ac.za

A forrás fájlok itt találhatóak: <https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle>

Az offline használathoz töltsd le a html vagy a pdf változat zip fájlját (a pdf ritkábban van frissítve) innen <http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/>

Copyright / Szerzői jogi megjegyzés

Copyright (C) Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers.
Permission is granted to copy, distribute and / or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with Invariant Sections being Foreword, Preface, and Contributor List, no
Front-Cover Texts, and no Back-Cover Texts. A copy of the license is
included in the section entitled „GNU Free Documentation License”.

Copyright (C) Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers.
Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére
és / vagy módosítására a GNU Free Documentation Licence feltételei alapján,
az 1.3-as verzió vagy bármely azt követő Free Software Foundation publikálási
verziójának feltételei alapján; nem változtatható szakaszok az előszó,
bevezető és a közreműködői lista, nincs címlapszöveg és nincs hátlapszöveg.
A jelen licenc egy példányát a „GNU Szabad Dokumentációs Licenc” elnevezésű
fejezet alatt találja.

Előszó a magyar fordításhoz

Aszalós László

külkapcsolati dékánhelyettes

Debreceni Egyetem, Informatikai Kar

A nyolcvanas évek közepén – amikor a home-computerekkel Magyarországra is megérkezett a széles tömegeknek szánt számítástechnika – a számítógépek még beépített BASIC-kel érkeztek, így nem igazán volt kérdés, hogy melyik programozási nyelvet válassza a felhasználó.

Az informatika egyre nagyobb térnyerésével az elérhető programozási nyelvek száma ugrásszerűen megnőtt, és az egyes programozási nyelvek evangelistái körömszakadtáig harcolnak azért, hogy a kezdők az ő kedvenc nyelvükön tanuljanak meg programozni.

Érdemes úgy tekinteni a programozásra, mint a kocsit vezetésére. Ha az ember a megkapja a jogosítványt, akkor az felhatalmazza arra, hogy bármilyen kocsit vezessen, legyen az hagyományos vagy automata sebességváltós, jobb- vagy balkormányos. Pár percig, vagy esetleg pár napig gondot okozhat az átállás, de az alapok ugyanazok. Hasonlóképpen, ha valaki megtanul egy programozási nyelvet, akkor viszonylag könnyen képes egy újabbat is elsajátítani, ám hogy virtuóz módon használhassa, évek gyakorlása szükséges.

Akkor mégis mi alapján érdemes kiválasztani az első nyelvet? Úgy gondolom, hogy egy új tevékenységnél a gyors elindulás, majd a folyamatos sikerélmény az, ami arra sarkallja a tanulót, hogy tovább haladjon. Ahogy a BASIC, úgy a Python esetén is pár sorban már tekintélyes dolgokat lehet megoldani. Példaképp a középiskolai programozási verseny feladataihoz rendszerint 10 soros program már elég szokott lenni.

A manapság divatos tényérnyi számítógépek (mint például a Raspberry Pi, vagy a BBC Microbit) esetén a Python az alapvető programozási nyelv. A legjobb amerikai egyetemek még az informatikus hallgatóikat is ezzel a nyelvvel ismertetik meg elsőként. Az informatikát használó egyéb szakterületeken (adatbányászat, bioinformatika, mérnöki tudományok, gépi tanulás stb.) is jellemző a Python elsősége, de a filmgyártás, kereskedelem, közlekedés is aktív használója a Pythonban írt programoknak, sőt a Google egyik hivatalos programozási nyelve. A középiskolai, egyetemi versenyeken egyre nagyobb számban alkalmazható a Python, sőt az emelt szintű érettségit is meg lehet vele oldani.

Egy programozási nyelv használhatóságát az határozza meg, hogy mit kapunk meg alapból, mi az ami elérhető hozzá, és milyen közösség szerveződött köré. A gazdag adattípus-készlet és ezek szabadon kombinálhatósága nagyon leegyszerűsítheti a programjainkat, itt egyből adott, amit más nyelvben le kellene programozni. A Python alapfilozófiája, hogy bőséges alapkönyvtárt tartalmaz, így csak nagyon speciális igények esetén kell pluszban kiegészítőket letöltenünk. Ha erre kényszerülünk, akkor közel 130 ezer csomag közül választhatunk, és ezek száma napról napra nő. Ha az angol nyelv nem jelent problémát, akkor szinte végtelen számú oktatóanyag, tankönyv, fórum, levelezési lista, előadásvideó érhető el. Azon dolgozunk, hogy a magyar nyelvű anyagok, lehetőségek száma növekedjen. Ennek része ez a fordítás is.

Az elmúlt évek tendenciáit figyelve egy évtized múlva már nem igazán lesz olyan szakma, ahol nem lesz szükség alapvető programozási ismeretekre. A kezdeti lépések (akár önálló) megtételéhez ajánljuk ezt a könyvet az általános iskolásoktól a nyugdíjasokig.

Előszó

David Beazley

Mint oktató, kutató és könyvíró, örömmel látom a könyv elkészültét. A Python egy szórakoztató és rendkívül könnyen használható programozási nyelv, amely egyre népszerűbbé vált az elmúlt években. A Guido van Rossum által tíz évvel ezelőtt kifejlesztett Python egyszerű szintaxisa és általános hangulata nagyrészt az ABC-ből származik, az 1980-as években kifejlesztett nyelvből. Ugyanakkor a Python-t azért hozták létre, hogy valódi problémákat oldjon meg, és számos megoldást kölcsönöztek más programozási nyelvekből, például C++-ból, Java-ból, Modula-3-ból és Scheme-ből. A Python egyik leginkább figyelemre méltó tulajdonsága az, hogy széles körben vonzza a professzionális szoftverfejlesztőket, tudósokat, kutatókat, művészeket és oktatókat.

Annak ellenére, hogy Python vonzza a különböző közösségeket, még mindig azon tűnődhetsz, hogy miért Python, vagy miért tanítjuk a programozást Pythonnal? Ezeknek a kérdéseknek a megválaszolása nem egyszerű feladat - különösen akkor, ha a közvélemény a mazochista alternatívák, mint például a C++ és a Java oldalán áll. Azonban, szerintem a legközvetlenebb válasz az, hogy a Python programozás egyszerűen sokkal szórakoztatóbb és produktívabb.

Amikor informatikai kurzusokat tanítok, szeretném lefedni a legfontosabb fogalmakat, továbbá az anyagot érdekesen akarom átadni a diákok számára. Sajnálatos módon a bevezető programozási kurzusoknál az a tendencia, hogy túl sok figyelmet fordítanak a matematikai absztrakcióra, és a diákok bosszantó problémákkal szembesülnek az alacsony szintű szintaxissal, a fordítással és a látszólag félelmetes szabályok végrehajtásával kapcsolatosan. Bár az ilyen absztrakció és formalizmus fontos a professzionális szoftvermérnökök és hallgatók számára, akik tervezik, hogy a tanulmányaikat az informatika területén folytatják, egy bevezető kurzus ilyen megközelítése unalmassá teheti az informatikát. Amikor egy tanfolyamot tanítok, nem akarok inspirálatlan tanulókat az osztályban. Nagyon szeretném látni, hogy érdekes problémákat próbálnak megoldani különböző ötletek feltérképezésével, a nemszokványos megközelítésekkel, a szabályok megszegésével és a saját hibáikból való tanulással. Ilyen módon nem akarom, hogy a szemeszter felében próbáljam rendezni az ismeretlen szintaxis problémákat, az érthetetlen fordító hibaüzeneteket vagy a több száz módon a program által generált általános védelmi hibákat.

Az egyik oka annak, hogy szeretem a Python-t, hogy ez egy nagyon szép egyensúlyt biztosít a gyakorlat és az elmélet között. Mivel a Python egy értelmező, a kezdők gyorsan elsajátíthatják a nyelvet, és szinte azonnal elkezdhetik csinálni a dolgokat, anélkül, hogy elvesznének a fordítás és összeszerkesztés (linkelés) problémáiban. Ezenkívül a Python nagy modulkönyvtárral rendelkezik, melynek használatával mindenféle feladatot elvégezhet a webprogramozástól a grafikaig. Az ilyen gyakorlati megközelítés nagyszerű módja a diákok lefoglalásának, és lehetővé teszi számukra, hogy jelentős projekteket hajtsanak végre. Ugyanakkor a Python kiváló alapot jelenthet a fontos informatikai fogalmak bevezetéséhez. Mivel a Python teljes mértékben támogatja az eljárásokat és az osztályokat, a hallgatókat fokozatosan vezethetjük be olyan témákba, mint az eljárás absztrakció, az adatszerkezetek és az objektum-orientált programozás, amelyek mindegyike alkalmazható később Java vagy C++ kurzusokon is. A Python számos funkciót kölcsönöz a funkcionális programozási nyelvekből, és felhasználható olyan fogalmak bevezetésére, amelyek részletesebben a Scheme és a Lisp kurzusokon szerepelnek.

Jeffrey bevezetőjét olvasva, nagy hatással volt rám az a megjegyzése, miszerint a Python hozzájárult ahhoz, hogy magasabb szintű sikert és alacsonyabb frusztrációt látott, és gyorsabban tudott haladni jobb eredményeket elérve. Bár ezek a megjegyzések a bevezető kurzusra hivatkoznak, én néha Pythont használok ugyanezen okok miatt a Chicagói Egyetem haladóbb végzős szintű informatikai kurzusain. Ezeknél a kurzusoknál folyamatosan szembe kell néznem azzal a rettenetes feladattal, hogy rengeteg nehéz tananyagot fedjek le egy fárasztó kilenches negyedévben. Meg-

lehet, hogy sok fájdalmat és szenvedést okozok egy olyan nyelven, mint a C++, gyakran úgy találtam, hogy ez a megközelítés kontraproduktív - különösen akkor, ha a kurzus olyan témáról szól, amely nem kapcsolódik a programozáshoz. Úgy vélem, hogy a Python használatával jobban összpontosíthatok a tényleges témára, miközben lehetővé teszi a hallgatók számára, hogy jelentős osztály projekteket hajtsanak végre.

Bár a Python még mindig fiatal és fejlődő nyelv, azt hiszem, hogy fényes jövője van az oktatásban. Ez a könyv fontos lépés ebben az irányban. David Beazley, Chicagói Egyetem, a *Python Essential Reference* szerzője.

Bevezetés

Jeffrey Elkner

Ez a könyv az Internet és a szabad szoftver mozgalom által lehetővé tett együttműködésnek köszönhetően jött létre. A három szerzője – egy főiskolai tanár, egy középiskolai tanár és egy professzionális programozó – soha nem találkozott szemtől szembe a munka során, de szoros együttműködést tudtunk végezni, sok más ember segítségével, akik időt és energiát szántak arra, hogy visszajelzéseket küldjenek nekünk.

Úgy gondoljuk, hogy ez a könyv az ilyen jellegű együttműködés előnyeit és jövőbeni lehetőségeit igazolja, amelynek keretét Richard Stallman és a Free Software Foundation hozta létre.

Hogyan és miért fordultunk a Pythonhoz

1999-ben első alkalommal került sor a College Board Advanced Placement (AP) informatikai vizsgájára C++ nyelven. Mint az ország sok más középiskolájában is, a nyelvek lecserélésére vonatkozó döntés közvetlen hatással volt Virginiában az Arlingtoni Yorktown High School informatikai tantervére, ahol tanítok. Addig a Pascal volt a tanítási nyelv mind az első évben, mind az AP kurzusain. Összhangban azzal a korábbi gyakorlattal, amely szerint a hallgatóknak két évig kell ugyanazon nyelvvvel foglalkozniuk, úgy döntöttünk, hogy az 1997-98-as tanév első évfolyamán C++-ra váltunk, hogy kövessük a College Board változtatásait az AP kurzusain a következő évben.

Két évvel később meg voltam győződve, hogy a C++ nem megfelelő választás a hallgatók informatikába való bevezetésére. Bár ez bizonyára egy nagyon erős programozási nyelv, de rendkívül nehéz megtanulni és megtanítani. Folyamatosan harcoltam a C++ nehéz szintaxisával és a sokféle megoldási móddal, melynek eredményeként túl sok diákot „elveszítettem”. Meg voltam győződve, hogy jobb nyelvet kell választanunk az elsőéves osztályoknál, ezért elkezdtem kutatni a C++ alternatíváját.

Szükségem volt egy olyan nyelvre, amely fut a GNU/Linux laborban, valamint a Windows és a Macintosh platformon is, melyekkel a hallgatók többsége rendelkezik. Szerettem volna ingyenes szoftvert használni, hogy a diákok otthon is használhassák jövedelmüktől függetlenül. Olyan nyelvet akartam, amelyet a professzionális programozók használnak, és egy aktív fejlesztői közösség veszi körül. Támogatja mind az eljárás, mind az objektumorientált programozást. És ami még fontosabb, hogy könnyű legyen megtanulni és megtanítani. Amikor ezeket a célokat szem előtt tartva kutattam, a Python kiemelkedett mint ezen feladatok legjobb pályázója.

Megkértem a Yorktown egy tehetséges hallgatóját, Matt Ahren-t, hogy próbálja ki Python-t. Két hónap alatt nem csak a nyelvet tanulta meg, hanem egy pyTicket nevű alkalmazást is írt, amely lehetővé tette a munkatársaink számára, hogy technikai problémákat jelentsenek be weben keresztül. Tudtam, hogy Matt ilyen rövid idő alatt nem tudta volna befejezni az alkalmazást a C++-ban, és ez a teljesítmény, kombinálva Matt pozitív Python értékelésével, azt sugallta, hogy a Python lesz az a megoldás, amit keresek.

Tankönyv keresése

Miután úgy döntöttem, hogy a következő évben Pythont használok mindkét osztályban az informatika bevezetésére, a legégetőbb probléma egy elérhető tankönyv hiánya volt.

Az ingyenes dokumentumok mentettek meg. Az év elején Richard Stallman bemutatta nekem Allen Downey-t. Mindketten írtunk Richard-nak kifejezve érdeklődésünket az ingyenes oktatási anyagok fejlesztéséhez. Allen már írt első éves informatikai tankönyvet, *How to Think Like a Computer Scientist* címmel. Amikor elolvastam ezt a könyvet, rögtön tudtam, hogy használni akarom az osztályaimban. Ez volt a legérthetőbb és leghasznosabb informatikai szöveg, amit láttam. Az algoritmikus gondolkodás folyamatát hangsúlyozta, nem pedig egy adott nyelv jellemzőit. Ezt elolvasva rögtön jobb tanárrá váltam.

How to Think Like a Computer Scientist nemcsak egy kiváló könyv volt, hanem a GNU nyilvános licenc alatt jelent meg, ami azt jelenti, hogy szabadon felhasználható és módosítható a felhasználó igényeinek megfelelően. Miután úgy döntöttem, hogy Pythont használok, eszembe jutott, hogy lefordítom Allen könyvének eredeti Java verzióját az új nyelvre. Bár nem lettem volna képes saját tankönyv megírására, azonban Allen könyve lehetővé tette ezt számomra, bebizonyítva azt, hogy a szoftverfejlesztésben kitűnően alkalmazható kooperatív fejlesztési modell ugyanolyan jól alkalmazható az oktatási anyagoknál is.

Az elmúlt két év munkája a könyvön gyümölcsöző volt mind magam, mind hallgatóim számára, és a hallgatóim nagy szerepet játszottak ebben a folyamatban. Mivel azonnal változtathattam, amikor valaki egy helyesírási hibát vagy egy nehézén érthető részt talált, arra bízattam őket, hogy keressék a hibákat a könyvben, és minden egyes alkalommal bónusz pontokat adtam nekik, amikor javaslatuk valamilyen változást eredményezett a szövegben. Ennek kettős előnye volt, bátorította őket arra, hogy alaposabban olvassák el a szöveget, és megvizsgálták a szöveget a legfontosabb kritikusok, a hallgatók, akik használva tanulták az informatikát.

A könyv második felében az objektumorientált programozási résznél tudtam, hogy valódi programozási tapasztalattal rendelkezőre van szükségem ahhoz, hogy helyes legyen. A könyv befejezetlen állapotban állt az év jelentős részében, amíg a nyílt forráskódú közösség ismét biztosította a befejezéséhez szükséges eszközöket.

Chris Meyerstől kaptam egy e-mailt, aki érdeklődést mutatott a könyv iránt. Chris egy professzionális programozó, aki elmúlt évben kezdett el programozást tanítani Pythonnal Oregonban a Lane Community College in Eugene-ben. A kurzus tanításának lehetősége vezette Christ a könyvhöz, és azonnal elkezdett segíteni. A tanév végéig készített a weboldalunkon egy közösségi projektet: <http://openbookproject.net> *Python for Fun* néven, és a haladóbb hallgatóimmal dolgozott együtt mint mester tanár, irányítva őket, meghaladva az eddigi elvárásokat.

Bevezetés a Python programozásba

Az elmúlt két évben a *How to Think Like a Computer Scientist* fordításának és használatának folyamata megerősítette, hogy a Python alkalmas a kezdő hallgatók tanítására. A Python nagyban leegyszerűsíti a programozási feladatokat, és megkönnyíti a legfontosabb programozási fogalmak tanítását.

Az első példa a szövegből szemlélteti ezt. Ez a hagyományos „Hello, World.” program, amely a könyv Java verziójában így néz ki:

```
class Hello {  
  
    public static void main (String[] args) {  
        System.out.println ("Hello, world.");  
    }  
}
```

a Python verzióban:

```
print("Hello, World!")
```


Bár ez egy triviális példa, a Python előnyei kiemelkednek. A Yorktowni informatikai kurzusnak nincsenek előfeltételei, így sok hallgató úgy tekint erre a példára, mint az első programjára. Néhányan kétségtelenül kissé idegesek, miután hallották, hogy a programozást nehéz megtanulni. A Java változat mindig arra kényszerít, hogy két nem megfelelő megoldás közül válasszak: vagy elmagyarázom a `class Hello, public static void main, String[] args, {, és }` utasításokat, kockáztatva, hogy összezavarok vagy megfélemlítek néhány hallgatót már az elején, vagy elmondom nekik, hogy: Ne aggódjatok most ezen dolgok miatt; majd később fogunk beszélni róla, és ugyanazt kockáztatjuk. Az oktatási cél a kurzus ezen pontján, hogy bevezessük a hallgatókat a programozási nyelv utasításaiba, hogy megírassák az első programjukat, és bevezessük őket a programozási környezetbe. Egy Python program pontosan ezeket a dolgokat kell tegye, és semmi mást.

A programhoz tartozó magyarázó szövegrészek összehasonlítása a könyv különböző változataiban tovább illusztrálja, hogy ez mit jelent kezdetben a hallgatónak. Hét paragrafus magyarázza a Hello, a világ!-ot a Java verzióban; a Python verzióban csak néhány mondat van. Még ennél is fontosabb, hogy a hiányzó hat bekezdés nem foglalkozik a számítógépes programozás alapötleteivel, hanem a Java szintaxisának apró részleteit magyarázza. Úgy találtam, hogy ugyanez történik az egész könyvben. Teljes bekezdések tűnnek el a Python verziójából, mert a Python sokkal egyszerűbb és világosabb szintaxisa miatt szükségtelenek.

A nagyon magas szintű nyelv használata, mint például a Python, lehetővé teszi a tanár számára, hogy elhalassza a gép alacsony szintű részleteiről való beszélgetést, amíg a hallgatók nem rendelkeznek olyan háttérrel, amely szükséges a részletek jobb megértéséhez. Így megadja azt a lehetőséget, hogy a pedagógiai / didaktikailag legfontosabb dolgokkal kezdjen. Ennek egyik legjobb példája az, ahogy a Python a változókat kezeli. Így megadja azt a lehetőséget, hogy a didaktikailag legfontosabb dolgokkal kezdjen. A Java-ban egy változó egy olyan hely neve, ahol az értéket tárolja, ha beépített típusú, és egy objektumhoz való hivatkozás, ha nem. Ezen különbségek megmagyarázása megköveteli, hogy megbeszéljék a számítógép adattárolásának módjait. Így egy változó fogalma szorosan összekapcsolódik a gép hardverével. A változók erőteljes és alapvető fogalma már elég nehéz a kezdő hallgatóknak (mind az informatikában, mind az algebrában). A bájtok és címek nem segítenek az ügyben. Pythonban egy változó olyan név, amely egy dologra utal. Ez egy sokkal intuitívabb koncepció a kezdő hallgatók számára, és sokkal közelebb áll a változó jelentéséhez, amit a matematikai kurzusokon tanultak. Sokkal kevesebb nehézséget okoztam a változók oktatása során ebben az évben, mint korábban, és kevesebb időt töltöttem a hallgatók problémáinak megoldásával.

A függvények szintaxisa egy másik példa arra, hogy a Python hogyan nyújt segítséget a programozás tanításában és tanulásában. A hallgatók mindig nagy nehézségekbe ütköztek a függvények megértése során. A fő probléma középpontjában a függvénydefiníció és a függvényhívás, valamint a paraméter és az argumentum közötti különbség van. A Python a szintaxissal siet a segítségünkre, amely egyenesen gyönyörű. A függvénydefiníciók a `def` kulcsszóval kezdődnek, ezért egyszerűen azt mondom a hallgatóknak: amikor definiálsz egy függvényt, kezd a `def`-el, majd folytasd a definiálandó függvény nevével; ha meghívod a függvényt, egyszerűen hívd (írd) a nevével. A paramétereket a definícióknál, az argumentumokat a hívásoknál használjuk. Nincsenek visszatérési típusok, paramétertípusok, vagy referencia- és értékparaméterek, így a függvényeket fele annyi időben és jobban meg tudom tanítani, mint korábban.

A Python használata javította az informatikai programunk hatékonyságát minden hallgató számára. Egy magasabb szintű általánosabb sikert és alacsonyabb frusztrációt láttam, mint amit addig a C++ vagy a Java használatával tapasztaltam. Gyorsabban haladtam jobb eredményeket elérve. Több hallgató végezte el a kurzust azzal a képességgel, hogy értelmes programokat tudtak írni és pozitív hozzáállást tanúsítottak a programozás iránt.

Egy közösség építése

A világ minden tájáról kaptam e-maileket, olyanoktól, akik ezzel a könyvvel tanítják vagy tanulják a programozást. Egy felhasználói közösség kezdett el felépülni, és sokan járultak hozzá a projekthez azáltal, hogy anyagokat küldtek a társ weboldalakon <http://openbookproject.net/pybiblio>.

A Python folyamatos növekedése mellett a felhasználói közösség növekedése folytatódni és gyorsulni fog. E felhasználói közösség kialakulása és annak lehetősége, hogy a pedagógusok hasonló együttműködést tanúsítsanak, a projekt legfontosabb része volt számomra. Együttműködéssel növelhetjük a rendelkezésre álló anyagok minőségét, és időt takaríthatunk meg. Meghívom Önt, hogy csatlakozzon a közösségünkhöz, és várom a jelentkezését. Kérjük, írjon nekem a következő címen: jeff@elkner.net.

Jeffrey Elkner

Arlingtoni Kormányzati Karrier és Technikai Akadémia (Governor's Career and Technical Academy)

Arlington, Virginia

A Rhodes helyi kiadása (Rhodes Local Edition - RLE) (2012. augusztusi verzió)

Peter Wentworth

Köszönetnyilvánítás ...

2010-től bevezetett kurzusainknál a Java-ról Pythonra váltottunk. Eddig azt látjuk, hogy az eredmények pozitívak. Az idő majd megmondja.

Ezen könyv elődje jó kiindulási pont volt számunkra, különösen a módosításokra vonatkozó szabadelvű engedélyek miatt. Házon belüli jegyzeteink vagy kiadványaink lehetővé teszik számunkra, hogy alkalmazzuk és frissítsük, átszervezzük, hogy lássuk mi az, ami működik és agilitást nyújt számunkra. Biztosítjuk, hogy a kurzusaink minden hallgatója megkapja a jegyzet másolatát – ami nem mindig történik meg, ha költséges tankönyveket írunk.

Nagyon sok köszönet az összes hozzájárulónak és a szerzőknek, hogy kemény munkájukat a Python közösség és a hallgatóink rendelkezésére bocsátották.

Egy kolléga és barát, Peter Warren egyszer azt a megjegyzést tette, hogy a bevezető programozás tanulása ugyanúgy szól a környezetről, mint a programnyelvről.

Nagy rajongója vagyok az IDE-knek (Integrated Development Environments). Szeretem, ha a segítség bele van integrálva a szerkesztőmbé – mint egy olyan egyed, amely támogatja más entitások számára az általánosan elérhető műveleteket – amit egy gombnyomással elérhetek. Szintaxis kiemelést akarok. Szeretnék azonnali szintaxis-ellenőrzést és jól működő automatikus kiegészítést. Szeretnék egy olyan szerkesztőt, amely el tudja rejteni a függvények testét vagy kódját, mert ezáltal elősegíti és ösztönzi a mentális absztrakciók építését.

Különösen rajongok az egyszerű lépésenkénti hibakeresőért és a töréspontokért a beépített kódellenőrzésnél. A program végrehajtásának koncepcionális modelljét próbáljuk kiépíteni a hallgató elméjében, melynek tanításához azt tartom a legjobb megoldásnak, ha a hívási vermet és a változókat láthatóvá tesszük, hogy azonnal ellenőrizni tudjuk az utasítások végrehajtásának eredményét.

Az én filozófiám tehát nem az, hogy egy olyan nyelvet keressek, amit megtaníthatok, hanem az IDE és a nyelv kombinációját keressem, amely egy csomagban van és egy egésként értékelhető.

Nagy változtatásokat hajtottam végre az eredeti könyvön, hogy ezt (és sok más általam is osztott véleményt) tükrözzem, és kétségem sincs afelől, hogy a kurzusaink tapasztalatai alapján ezt további változások fogják követni.

Íme néhány olyan kulcsfontosságú dolog, amelyet másképp közelítettem meg:

- A mi helyzetünk azt követeli meg, hogy a bevezető kurzus anyagát alig három hét alatt adjuk át egy nagy számú hallgatói csoportnak, majd egy féléven keresztül tanítjuk azokat, akik részt vesznek a fő programunkban. Tehát a könyv két részből áll: először az első öt fejezet vesszük a nagy „kipróbálás” részben, a fennmaradó anyagot pedig egy különálló félévben.

- A Python 3-at használjuk. Tisztább, objektum-orientáltabb, és kevesebb ad-hoc jellegű megoldás van benne, mint a Python korábbi verzióinál.
- A PyScriptert használjuk IDE-ként, a Windows rendszeren. És ez része ezen jegyzetnek, képernyőképekkel, stb.
- Elvettem a GASP-t.
- A grafikák során a Turtle modullal kezdünk. Ahogy haladunk tovább, a PyGame-t használjuk a fejlettebb grafikákra.
- Bevezettem az eseményvezérelt programozást a teknős használatával.
- Megpróbáltam több objektumorientált fogalmat korábban elmondani anélkül, hogy a hallgatókat az objektumok egységbe zárására vagy saját osztályok írására kértem volna. Így például a teknősökről szóló fejezetben létrehoztuk a teknősök többszörös példányát, beszéltünk azok tulajdonságairól és állapotáról (szín, pozíció stb.), ezt a metódus hívási stílust kedveljük a mozgatusukra `Eszti.forward(100)`. Hasonlóképpen, ha véletlenszerű számokat használunk, elkerüljük a véletlenszerű modulban lévő „hidden singleton generator”-t – mi inkább a generátor példányát hozzuk létre, és meghívjuk a metódusokat a példányon.
- A listák és a `for` ciklus létrehozásának egyszerűsége a Pythonban nyerőnek tűnik, ezért a hagyományos `input` parancssori adatok helyett előnyben részesítjük a ciklusok és listák használatát, mint ez:

```
1 barátok = ["Peti", "Kati", "Misi"]
2 for f in barátok:
3     meghivo = "Szia " + f + "! Szeretettel meghívlak a szombati bulimra!"
4     print(meghivo)
```

Ez azt is jelenti, hogy a `range` bemutatását korábbra ütemeztem. Úgy gondolom, hogy idővel több lehetőség nyílik kiaknázni a „korai listákat, korai iterációkat” a legegyszerűbb formában.

- Én elvettem a `doctest`-t: ez túl szokatlan nekem. Például a teszt nem sikerül, ha a listanevek közötti távolság nem pontosan ugyanaz, mint a kimeneti karakterlánc, vagy ha a Python egy idézetből álló szöveget ír ki, de a tesztet kettős idézőjelekkel írta le. Az ilyen esetek eléggé összezavarják a hallgatókat (és az oktatókat):

```
1 def addlist(xs):
2     """
3     >>> xs = [2,3,4]
4     >>> addlist(xs)
5     9
6     """
7     return
```

Ha meg tudná elegánsan magyarázni az `xs` paraméter és az `xs` `doctest` változó hatáskörére, valamint élettartamára vonatkozó szabályok közötti különbséget, kérjük ossza meg velem. Igen, tudom, hogy a `doctest` létrehozza a „hátnak mögött” a saját hatókörét, de pontosan ez az a fekete mágia, amit próbálunk elkerülni. A megszokott behúzási szabályoktól függően úgy tűnik, hogy a `doctest`-ek be vannak ágyazva a függvények hatókörébe, de valójában nincsenek. A hallgatók úgy gondolták, hogy a paraméter megadott értékét `xs`-hez rendelik a `doctest`-ben!

Azt is gondolom, hogy a teszt a tesztelt függvényektől való elkülönítése a hívó és a hívott közötti tisztább kapcsolatot eredményez, és jobb esélyt ad arra, hogy pontosan megtanulják az argumentum átadás / paraméter fogalmakat.

Van egy jó egységteszt modul a Pythonban, (és a PyScripter integrált támogatást nyújt, és automatikusan generálja a teszt modulok vázat), de úgy tűnt, hogy túl haladó ez még a kezdők számára, mert megköveteli a több modulból álló csomagok fogalmát.

Ezért a 6. fejezetben (kb. 10 sornyi kóddal) megadtam a saját teszt szerkezetemet, amelyet a diákoknak be kell illeszteniük a fájlba, amilyen dolgoznak.

- Lefutattam a parancssori bemenetet / folyamatot / kimenetet, ahol lehetet. Sok hallgatónk soha nem látott parancsértelmezőt, és vitathatatlanul elég megfélemlítő.
- Visszatértünk a „klasszikus / statikus” megközelítéshez, saját osztályaink és objektumaink írásához. A Python (a cégeknél, az olyan nyelvek mint a Javascript, a Ruby, a Perl, a PHP stb.) nem igazán hangsúlyozza a „lezárt” osztályok vagy „privát” tagok, vagy akár „lezárt példányok” fogalmát.

Tehát az egyik tanítási megközelítés az, hogy minden egyes példányhoz hozzárendel egy üres tárolót, majd ezt követően lehetővé teszi az osztály külső ügyfeleinek, hogy új tagokat (metódusokat vagy attribútumokat) módosítsanak a különböző példányokban, amennyit csak akarnak. Ez egy nagyon dinamikus megközelítés, de talán nem olyan, amely ösztönözi az absztrakciókban való gondolkodást, a szinteket, az összevonásokat, szétválasztást stb. Még az is lehet, hogy ennek a módszernek a káros hatásáról cikkek születnének.

Konzervatívabb megközelítés, ha egy inicializálót helyezünk minden osztályba, az objektum példányosításának idején meghatározzuk, hogy milyen tagokat akarunk, és inicializáljuk a példányokat az osztályon belül. Tehát közeledtünk a C# / Java filozófiájához.

- Korábban több algoritmust kezdtünk el bevezetni a kurzusba. A Python egy hatékony tanítási nyelv – gyorsan haladhatunk. De az itt elért eredményeket szeretnénk az alapokba fektetni mélyebb problémamegoldásra és bonyolultabb algoritmusok tanítására, ahelyett, hogy „több Python-függvényt” vezetnénk be. Néhány ilyen változás utat tört ebben a verzióban, és biztos vagyok benne, hogy a jövőben még többet ilyet fogunk látni.
- Érdeklődünk az oktatással és a tanulással kapcsolatos kérdések iránt. Egyes kutatások szerint a „szellemi játékosság” nagyon fontos. A végén az apró-cseprő munkafüzetben hivatkozott tanulmány esetén úgy tűnt, hogy nem érdemes a könyvbe tenni, mégis azt akartam, hogy benne legyen. Nagyon valószínű, hogy több ilyen típusú témát engedélyezünk a könyvbe, hogy megpróbáljuk többé tenni, mint a Python programozás.

Közreműködői lista

A Free Software Foundation (Szabad Szoftver Alapítvány) filozófiájának parafrázisára: a könyv szabadon felhasználható. Szabad alatt gondolj az eszmére, ne egy ingyen pizzára, amit szabadon elvehetsz. Ez azért jött létre, mert az együttműködés nem lett volna lehetséges a GNU Szabad Dokumentációs Licenc nélkül. Ezért szeretnénk köszönetet mondani a Szabad Szoftver Alapítványnak (FSF) a licenc fejlesztéséért és természetesen az elérhetővé tételéért.

Ugyancsak szeretnénk köszönetet mondani a több mint 100 éles szemű és figyelmes olvasónak, akik az utóbbi években javaslatokat és korrekciókat küldtek számunkra. A szabad szoftverek szellemében úgy döntöttünk, hogy köszönetünket közreműködői listában fejezzük ki. Sajnos ez a lista nem teljes, de mindent megteszünk, hogy naprakész állapotban tartsuk. Túl hosszú a lista ahhoz, hogy mindazokat szerepeltessük, akik jeleztek 1-2 elírást. Hálásak vagyunk érte és a közreműködőket is elégedettséggel töltheti el, hogy jobbá tette a könyvet saját maga és mások számára is. A 2. kiadás listájához új kiegészítés lesz, azok listája, akik folyamatosan járulnak hozzá a könyv tökéletesítéséhez.

Ha van esély arra, hogy átnézd a listát, akkor tisztában kell lenni azzal, hogy mindenki, aki beküldött egy megjegyzést, megkímélt téged és minden további olvasót, a zavaró technikai hibáktól vagy egy kevésbé érthető magyarázattól.

Lehetetlennek tűnik a sok korrekció után, de még mindig vannak hibák ebben a könyvben. Ha találsz egyet, reméljük szívesen rá egy percet, hogy kapcsolatba lépj velünk. Az e-mail cím (a könyv Python 3 verziójához) p.wentworth@ru.ac.za. A végrehajtott lényeges módosításokat javaslatokat hozzá fogjuk adni a közreműködői lista következő verziójához (hacsak nem kéred, hogy hagyjuk ki). Köszönjük!

Második kiadás

- Mike MacHenry e-mailje elmagyarázta a jobb-rekurziót. Nem csak rámutatott a bemutatásban történt hibára, de azt is javasolta, hogyan javítsuk ki.
- Csak akkor jöttem rá, hogy mit akarok használni az objektum orientált programozással foglalkozó fejezetekben esettanulmányként, amikor egy 5. osztályos diák, Owen Davies egy szombat reggeli Python kurzuson odajött hozzám, és azt mondta, hogy meg akarja írni Python-ban a Gin Rummy kártyajátékot.
- Egy *különleges* köszönet az úttörő hallgatóknak, Jeff Python programozás osztályának **GCTAA** a 2009-2010-es tanév során: Safath Ahmed, Howard Batiste, Louis Elkner-Alfaro és Rachel Hancock. A folyamatos és átgondolt visszajelzések megváltoztatták a legtöbb fejezetet. Meghatározta az aktív és elkötelezett tanulók számára azt a normát, amely segíteni fog az új Kormányzó Akadémiájának létrehozásában. Nektek köszönhetően, ez valóban egy *diákok által tesztelt* szöveg lett.
- Köszönöm hasonlóan a HB-Woodlawn programban résztvevő Jeff informatika osztályában lévő diákoknak, a 2007-2008-as tanévben: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail és Iliana Vazuka.
- Ammar Nabulsi számos korrekciót küldött az 1. és a 2. fejezetből.
- Aldric Giacomoni hibát jelzett a Fibonacci-sorozat definíciójában az 5. fejezetben.
- Roger Sperberg több helyesírási hibát küldött, és rámutatott egy csavart logikára a 3. fejezetben.

- Adele Goldberg leült Jeff-el a PyCon 2007 konferencián, és odaadta neki a javaslatok és korrekciók listáját az egész könyvről.
- Ben Bruno küldött javítási javaslatokat a 4., 5., 6. és 7. fejezetből.
- Carl LaCombe rámutatott arra, hogy a 6. fejezetben helytelenül használtuk a kommutatív kifejezést, ahol a szimmetrikus volt a helyes.
- Alessandro Montanile a 3., 12., 15., 17., 18., 19. és 20. fejezetben szereplő példa kódok és a szövegek hibáinak javítását küldte.
- Emanuele Rusconi hibákat talált a 4., 8. és 15. fejezetben.
- Michael Vogt egy azonosítási hibát jelzett egy példában a 6. fejezetben, és javaslatot tett az 1. fejezet shell és szkript egyértelműségének javítására.

Első kiadás

- Lloyd Hugh Allen egy javítást küldött a 8.4. fejezethez.
- Yvon Boulianne az 5. fejezet szemantikai hibájának javítását küldte.
- Fred Bremmer egy korrekciót küldött a 2.1. fejezethez.
- Jonah Cohen írt egy Perl szkriptet, hogy átalakítsa a LaTeX forrást gyönyörű HTML-re ebben a könyvben.
- Michael Conlon egy nyelvtani korrekciót küldött a 2. fejezethez, és javította az 1. fejezet stílusát, és beszélgetést kezdeményezett a fordítók technikai aspektusairól.
- Benoit Girard egy humoros hibát küldött az 5.6. fejezetben.
- Courtney Gleason és Katherine Smith írta meg a *horsebet.py*-t, amelyet esettanulmányként használtunk a könyv korábbi változatában. Programjuk megtalálható a honlapon.
- Lee Harr korábban több korrekciót nyújtott be, mint amennyit itt felsorolunk, és valóban szerepelnie kell a könyv egyik legfontosabb szerkesztőjeként.
- James Kaylin egy hallgató, aki használja a könyvet. Számos korrekciót küldött.
- David Kershaw kijavította a törött *catTwice* függvényt a 3.10. fejezetben.
- Eddie Lam számos korrekciót küldött az 1., 2. és 3. fejezetbe. Beállította a Makefile-t is, hogy létrehozza az indexet az első futtatásnál, és segített nekünk egy verziókezelő-sémát is létrehozni.
- Man-Yong Lee korrekciót küldött a 2.4. fejezetben szereplő példa kódhoz.
- David Mayo rámutatott arra, hogy az 1. fejezetben a tudattalan (unconsciously) szót cserélni kell a szót tudat alatti (subconsciously) kifejezésre.
- Chris McAloon számos korrekciót küldött a 3.9. és a 3.10. fejezetekhez.
- Matthew J. Moelter régóta közreműködik, aki számos korrekciót és javaslatot küldött a könyvhöz.
- Simon Dicon Montford egy hiányzó függvény definíciót és több beírt hibát jelentett be a 3. fejezetben. A 13. fejezetben található *increment* függvényben is hibákat talált.
- John Ouzts javította a visszatérési érték meghatározását a 3. fejezetben.
- Kevin Parks értékes megjegyzéseket és javaslatokat küldött arra vonatkozóan, hogyan lehetne javítani a könyv terjesztését.
- David Pool egy gépelési hibát küldött az 1. fejezet szójegyzékében, valamint kedves bátorító szavakat.
- Michael Schmitt a fájlokról és kivételekről szóló fejezetben javított.

- Robin Shaw rámutatott egy hibára a 13.1. fejezetben, ahol a `printTime` függvényt használtuk egy példában meghatározás nélkül.
- Paul Sleight hibát talált a 7. fejezetben és egy program hibát Jonah Cohen Perl szkriptjében, amely HTML-t generál a LaTeX-ből.
- Craig T. Snydal tesztelte a könyvet egy kurzuson a Drew Egyetemen. Számos értékes javaslatot és korrekciót nyújtott be.
- Ian Thomas és tanítványai a könyvet egy programozási tanfolyamon használják. Ők az elsők, akik a könyv második felében is tesztelik a kódokat, és számos korrekciót és javaslatot tettek.
- Keith Verheyden korrekciót küldött a 3. fejezethez.
- Peter Winstanley értesített minket, hogy a 3. fejezetben régóta fennálló hiba van latinul.
- Chris Wrobel korrigálta a kódot az I/O fájllal és kivételekkel foglalkozó fejezetben.
- Moshe Zadka felbecsülhetetlen mértékben hozzájárult ehhez a projekthez. A szótárakról szóló fejezet első tervezetének elkészítése mellett folyamatos útmutatást nyújtott a könyv korai szakaszában.
- Christoph Zwerschke számos korrekciót és pedagógiai javaslatot küldött, és kifejtette a különbséget a *gleich* és a *selbe* között.
- James Mayer egy sornyi helyesírási és nyomdai hibát küldött nekünk, köztük kettőt a Közreműködői listán.
- Hayden McAfee rábukkant két példa közti lehetséges zavaró ellentmondásra.
- Angel Arnal egy nemzetközi fordítói csapat tagja, amely a szöveg spanyol változatán dolgozik. Számos hibát talált az angol változatban is.
- Tauhidul Hoque és Lex Berezhny elkészítették az 1. fejezet illusztrációit, és tökéletesítették sok más illusztrációt.
- Dr. Michele Alzetta hibát talált a 8. fejezetben és érdekes pedagógiai észrevételeket és javaslatokat küldött a Fibonacci-ról és az Old Maid-ről.
- Andy Mitchell az 1. fejezetben elírást és a 2. fejezetben egy hibás példát talált.
- Kalin Harvey a 7. fejezetben tisztázást, pontosítást javasolt, és néhány elírást is talált.
- Christopher P. Smith többféle elírást talált, és segít nekünk előkészíteni a Python 2.2. könyv frissítését.
- David Hutchins talált egy elírást az Előszóban.
- Gregor Lingl a Pythont egy középiskolában tanítja Bécsben, Ausztriában. A könyv német fordításán dolgozik, és néhány súlyos hibára bukkant az 5. fejezetben.
- Julie Peters talált egy hibát a Bevezetésben.

Magyar fordítás

- Kelemen Mátyás több javaslatot küldött a magyar fordítás pontosításához, javításához.

1. fejezet

A programozás mikéntje

Ennek a könyvnek az a célja, hogy megtanítson téged informatikusként gondolkodni. Ez a fajta gondolkodásmód kombinálja a matematika, a mérnöki- és a természettudományok legjavát. Mint ahogy a matematikusok úgy az informatikusok is egy formális nyelvet használnak a gondolatok lejegyzésére (különösen a számítások esetén). Mint ahogy a mérnökök úgy ők is terveznek dolgokat, összerakják a komponenseket egy rendszerré és kiértékelik a kompromisszumokat az alternatívák között. Mint ahogy a tudósok, ők is megfigyelik komplex rendszerek viselkedését, hipotéziseket állítanak fel és ellenőrzik a jóslataikat.

Az informatikusok számára a legfontosabb képesség a **problémamegoldás**. A problémamegoldás jelenti azt a képességet, hogy megfogalmazzuk a problémát, kreatívan gondolkodjunk a megoldás menetéről és tisztán, precízen fejezzük ki a megoldást. Ahogy ez ki fog derülni, a programozás megtanulásának folyamata egy kiváló lehetőség a problémamegoldási képesség fejlesztésére. Ez az amiért ez a fejezet *A programozás mikéntje* címet kapta.

Egy szinten programozást fogsz tanulni, ami önmagában is egy hasznos képesség. Egy másik szinten a programozást, mint eszközt fogod használni, hogy elérd a célkitűzésedet. Ahogy haladunk majd előre, ez a cél egyre tisztább lesz.

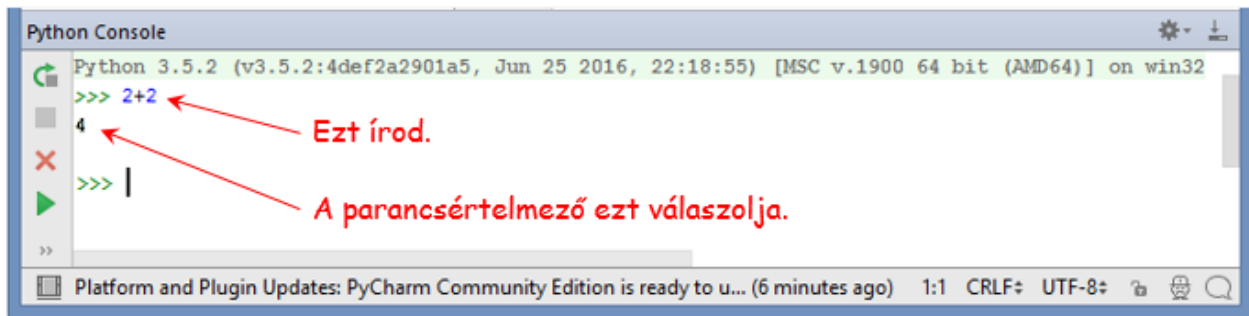
1.1. A Python programozási nyelv

A programozási nyelv, amit most megtanulsz a Python. A Python egy példa **magas szintű nyelvekre**. Talán már hallottál más magas szintű nyelvről is, mint például a C++, a PHP, a Pascal, a C# és a Java.

Ahogy a magas szintű nyelv névből következtethetsz, vannak **alacsony szintű nyelvek** is, amelyeket néha gépi nyelvként vagy assembly nyelvként emlegetünk. Pontatlanul mondván a számítógépek csak azokat a programokat tudják futtatni, amelyeket alacsony szintű nyelven írtak. Így tehát mielőtt futtatnánk egy magas szintű nyelven megírt programot, át kell alakítanunk valami sokkal megfelelőbbre.

Majdnem minden programot magas szintű nyelveken írnak ezek előnyei miatt. Sokkal egyszerűbb magas szintű nyelven programozni, így kevesebb időt vesz igénybe a kód megírása, illetve az rövidebb és olvashatóbb lesz, valamint sokkal nagyobb a valószínűsége, hogy hibátlan lesz. Másrészt a magas szintű nyelvek **hordozhatóak**, ami azt jelenti, hogy a programokat futtathatjuk különböző számítógépeken néhány kisebb módosítással, vagy változtatás nélkül.

A motort, amely átalakítja és futtatja a Pythont, **Python parancsértelmezőnek** hívjuk. Kétféleképpen használhatjuk: *interaktív módban* és *szkript módban*. Az interaktív módban Python kifejezéseket gépelsz a Python parancsértelmező ablakba és az azonnal mutatja az eredményt:



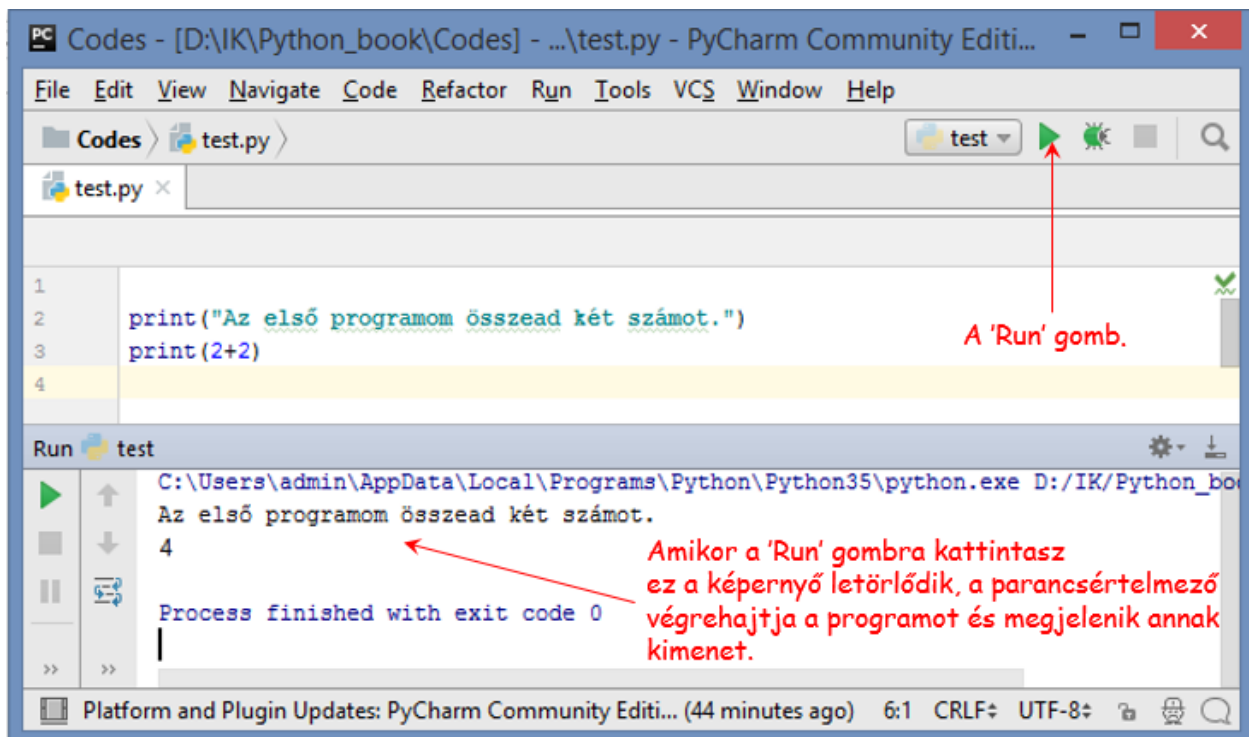
A `>>>` neve **Python prompt**. A parancsértelmező arra használja a promptot, hogy jelezze készen áll az utasítások fogadására. Mi azt gépeltük be, hogy `2+2` és a parancsértelmező kiértékelte a kifejezésünket, majd válaszolt a `4` eredménnyel és végül egy új sorban adott egy új promptot jelezve, hogy készen áll további bemenetre.

Emellett a programodat egy fájlba is írhatod és használhatod a parancsértelmezőt a fájl tartalmának végrehajtására. Az ilyen fájlok neve **szkript**. A szkriptek előnye, hogy el lehet menteni őket vagy akár kinyomtatni és így tovább.

A könyv ezen kiadásában mi a **PyCharm** nevű fejlesztői környezetet *Community Edition* kiadását használjuk. (Elérhető a <https://www.jetbrains.com/pycharm/> címen.) Van számos másik fejlesztői környezet is.

Például létrehozhatunk egy `firstprogram.py` nevű fájlt a PyCharm segítségével. Megállapodás szerint azok a fájloknak, amelyek Python programot tartalmaznak a nevük `.py` kiterjesztésre végződik.

Hogy végrehajtsuk a programot csak a PyCharm **Run** (Fuss) gombjára kell kattintanunk:



A legtöbb program érdekesebb, mint ez.

Közvetlenül a parancsértelmezőben dolgozni kényelmes, ha rövid kódokat tesztelünk, mert így azonnali visszajelzést kapunk. Gondolj úgy erre, mint egy vázlatpapírra gondoltál korábban, amely segít a probléma kidolgozásában. Minden, ami pár sornál hosszabb lehetőleg a szkript fájlba kerüljön.

1.2. Mi egy program?

A **program** az utasítások sorozata, amelyek azt határozzák meg, hogyan kell elvégezni egy számítást. A számítás lehet valami matematikai jellegű, mint például egyenletrendszerek megoldása vagy egy egyenlet gyökeinek megtalálása, de lehet szimbolikus számítás is, mint például megkeresni és kicserélni egy szöveget egy dokumentumban vagy (elégg különleges módon) fordítani egy programot.

A részletek különböző módon néznek ki különböző nyelveken, de néhány alapvető utasítás megjelenik szinte minden nyelvben:

bemenet (input) Adat beolvasása billentyűzetről, egy fájlból vagy valamilyen másik eszközzel.

kiment (output) Adat megjelenítése a képernyőn vagy adat küldése fájlba vagy más eszközre.

matematika Alap matematikai műveletek végrehajtása, mint például összeadás vagy szorzás.

feltételes végrehajtás Bizonyos feltételek ellenőrzése és ez alapján a megfelelő utasítássorozat végrehajtása.

ismétlés Néhány tevékenység végrehajtása újra meg újra, többnyire apró változásokkal.

Hiszed vagy nem, szinte ennyi az egész. Minden program, amit valaha használtál, függetlenül attól, mennyire komplikált, többé kevésbé olyan utasításokból épül fel, mint ezek. Így tehát leírhatjuk a programozást úgy, mint egy nagy, komplex feladat kisebb és kisebb részfeladatokra osztásának folyamatát, amíg a részfeladatok elég egyszerűek nem lesznek ahhoz, hogy ezeknek az alap utasításoknak a sorozatával megadjuk őket.

Ez egy kicsit még homályos lehet, de majd visszatérünk ehhez a témához, amikor az **algoritmus** fogalmáról fogunk beszélni.

1.3. Mi a nyomkövetés?

A programozás egy komplex feladat és mivel emberek végzik, gyakran hibához vezet. A programozás során fellépő rendellenesség a **program hiba** és ennek megkeresése és kijavítása a **nyomkövetés**. A program hibára használt angol *bug* (bogár) kifejezés, ami kis mérnöki nehézséget jelent, Thomas Edisontól származik 1889-ből.

Háromféle hiba jelenhet meg a programban: **szintaktikai hiba**, **futási idejű hiba** és **szemantikai hiba**. Fontos, hogy különbséget tegyünk köztük, azért, hogy gyorsabban lenyomozhassuk őket.

1.4. Szintaktikai hibák

A Python csak akkor tudja végrehajtani a programot, ha az szintaktikailag helyes, azaz a formai szabályoknak eleget tesz, különben a folyamat megakad és visszatér egy hibaüzenettel. A **szintaxis** a program szerkezetére és annak szabályaira vonatkozik. Mint például a magyar nyelvben az, hogy a mondatoknak nagybetűvel kell kezdődniük és írásjellel végződniük. tehát! ez a mondat **szintaktikai hibát** tartalmaz

A legtöbb olvasó számára néhány szintaktikai hiba nem mérvadó probléma. A Python nem ennyire megbocsájtó. Ha csak egyetlen szintaktikai hiba is van a programodban, a Python egy hibaüzenetet jelenít meg és kilép, így nem tudod lefuttatni a programodat. A programozói karriered első pár hetében bizonyára sok időt fogsz azzal tölteni, hogy keresed a szintaktikai hibákat. Ahogy tapasztalatot szerzel, kevesebb hibát fogsz ejteni és gyorsabban megtalálod őket.

1.5. Futási idejű hibák

A második hibacsoport a futási idejű hiba, amit azért hívunk így, mert nem jelennek meg addig, amíg nem futtatjuk a programot. Ezeket a hibákat **kivételek** néven is emlegetjük, mert gyakran azt jelzik valami kivételes (és rossz) dolog történt.

A futási idejű hibák ritkák az olyan egyszerű programokban, amelyeket az első fejezetekben fogsz látni, szóval beletelik egy kis időbe, míg végre összetalálkozol eggyel.

1.6. Szemantikai hibák

A harmadik hibatípus a **szemantikai hiba**. Ha csak szemantikai hiba van a programodban, akkor az sikeresen le fog futni, abban az értelemben, hogy nem generál egyetlen hibaiüzenetet sem, de nem azt fogja csinálni, amire szántad. Valami mást fog csinálni. Kimondottan azt teszi, amit mondtál neki, hogy tegyen.

A probléma az, hogy a program, amit írtál nem az a program, amit írni akartál. A program jelentése (szemantikája) más. A szemantikai hibák azonosítása trükkös, mert azt követeli meg tőled, hogy visszafelé dolgozz, nézd meg a program kimenetét és próbáld meg kitalálni mit és miért csinált.

1.7. Kísérleti nyomkövetés

Az egyik legfontosabb képesség, amit meg fogsz szerezni az a nyomkövetés, vagyis a hibakeresés és hibajavítás. Habár frusztráló lehet, a nyomkövetés az egyik intellektuálisan leggazdagabb, legnagyobb kihívást jelentő és érdekes része a programozásnak.

Bizonyos tekintetben a nyomkövetés olyan, mint a nyomozói munka. Szembekerülsz bűnjelkekkel és következtetned kell a folyamatokra és az eseményekre, amelyek az általad látott eredményekhez vezettek.

A nyomkövetés egyfajta kísérleti tudomány. Egyszer csak van egy ötleted azzal kapcsolatban mi zajlott rosszul módosítod a programodat és újra próbálkozol. Ha a feltevésed helyes volt, akkor megjósolhatod a módosításod eredményét és egy lépéssel közelebb jutsz a működőképes programhoz. Ha a hipotézised téves, akkor elő kell állnod egy újjal. Ahogy Sherlock Holmes is rámutatott: „Ha a lehetetlent kizártuk, ami marad, az az igazság, akármilyen valószínűtlen legyen is.” (A. Conan Doyle, *A Négyek jele*)

Néhány ember számára a programozás és a nyomkövetés ugyanaz a dolog. Azaz a programozás a program fokozatos nyomkövetésének a folyamata, mindaddig, amíg azt nem kapod, amit akartál. Az ötlet az, hogy kezdj egy programmal, ami csinál *valamit* és végezz kis módosításokat, végezz nyomkövetést, ahogy haladsz, így mindig egy működő programod lesz. Például a Linux egy operációs rendszer, ami kódsorok millióit tartalmazza, de egy egyszerű programként indult, amivel Linus Torvalds az Intel 80386-os chipjét vizsgálta. Larry Greenfield szerint Linus egyik korábbi projektjében volt egy program, amely képes volt az AAAA és a BBBB kijelzése között váltani. Később ez fejlődött a Linux operációs rendszerre (*A Linux felhasználói kézikönyv* béta verzió 1).

A későbbi fejezetek majd több javaslatot és egyéb programozói gyakorlatot fognak adni.

1.8. Formális és természetes nyelvek

A **természetes nyelvek** azok a nyelvek, amelyeket az emberek beszélnek, például angol, spanyol és francia. Ezeket nem emberek tervezték (habár próbálnak rendszert vinni bele), hanem természetesen fejlődtek.

A **formális nyelvek** azok a nyelvek, amelyeket az emberek terveztek tudományos alkalmazás céljából. Például a jelölések, amelyeket a matematikusok használnak, azok egy formális nyelvet alkotnak, ami különösen jó számok és szimbólumok közötti kapcsolatok jelölésére. A vegyészek egy formális nyelvet használnak a molekulák kémiai szerkezetének megjelenítésére. És a legfontosabb:

A programozási nyelvek is formális nyelvek, amelyeket arra fejlesztettek ki, hogy számításokat fejezzenek ki.

A formális nyelveknek többnyire szigorú szintaktikai szabályaik vannak. Például a $3+3=6$ kifejezés szintaktikailag helyes matematikai állítás, de a $3+=6\$$ nem az. H_2O egy szintaktikailag pontos kémiai jelölés, de a $_2Zz$ nem az.

A szintaktikai szabályok két csoportból állnak: a **szövegelemekre** és a szerkezetekre vonatkozóak. A szövegelemek a nyelv alap elemei, például szavak, számok, zárójelek, vesszők stb. Pythonban a `print("Boldog új évet ", 2018)` utasítás 6 szövegelemet tartalmaz: a függvény nevét, a nyitó zárójelet, a sztringet, a vesszőt, a számot és a bezáró zárójelet.

Ejthetünk hibát a szövegelem létrehozásakor. Az egyik probléma a $3+=6\$$ kifejezéssel az, hogy a $\$$ nem egy valós matematikai jelölés (legalábbis ahogy tudjuk). Hasonlóan a ${}_2Zz$ sem szabályos szövegelem a kémiai jelölésekben, mert nincs Zz vegyjelű elem.

A szintaktikai szabályok második csoportja az utasítások **szerkezetére** vonatkozik – azaz arra, hogyan rendezzük el a szövegelemeket. A $3+=6\$$ állítás szerkezetileg sem megfelelő, mivel nem tehetünk pluszjelet az egyenlőségjel elé közvetlenül. Hasonlóan a kémiai formulákban az alsó indexek mindig az elem vegyjele után kerülnek, nem elé. És a Python példánkban, ha kihagyjuk a vesszőt vagy ha felcseréljük a két zárójelet így `print) "Happy New Year for ", 2013 (`, akkor is 6 legális és érvényes szövegelemünk lesz, de a szerkezet nem elfogadható.

Amikor egy magyar szöveget olvasol vagy egy formális nyelv állítását, mindkét esetben neked kell kitalálni mi a mondat szerkezete (habár a természetes nyelv esetén ezt tudat alatt csinálod). Ez a folyamat a **nyelvtani elemzés**.

Például, amikor hallod a *Az ördög nem alszik* mondatot, akkor te érted, hogy az ördög az alany és az alszik szó az ige. Amikor elvégzed a nyelvtani elemzést, ki tudod találni, hogy mit jelent a mondat, vagyis mi a **szemantikája**. Feltéve, hogy tudod, mi az az ördög és mit jelent az alvás, meg fogod érteni a mondat általános jelentését.

Habár a formális és a természetes nyelveknek van sok közös tulajdonsága – szövegelemek, szerkezetek, szintaxis és szemantika – számos dologban különböznek.

félreérthetőség A természetes nyelvek tele vannak félreérthető dolgokkal, az embereknek a szöveggörnyezet vagy más információ alapján kell döntenie a jelentésről. A formális nyelveket úgy tervezték, hogy szinte teljesen félreérthetetlenek, ami azt jelenti, hogy minden állításnak pontosan egy jelentése van, tekintet nélkül a szöveggörnyezetre.

redundancia A félreértések csökkentése érdekében a természetes nyelvek sok redundanciát tartalmaznak (azaz többféleképpen is elmondhatjuk ugyanazt és egy kifejezésnek több jelentése is lehet). Ennek következtében bőbeszédűek. A formális nyelvek kevésbé redundánsak és sokkal velősebbek.

fantáziánélküliség A formális nyelvek esetén a jelentés pontosan az, amit mondunk. Másrészt a természetes nyelvek tele vannak kifejezésmódokkal és metaforákkal. Ha valaki azt mondja *Az ördög nem alszik* valószínűleg nem egy természetfeletti légy álmatlanságáról beszél. Ismerned kell a szólás jelentését, átvitt értelmét.

Az emberek, akik természetes nyelveken beszélve nőttek fel – azaz mindenki – gyakran nehézséget jelent a formális nyelvekhez való alkalmazkodás. Bizonyos értelemben a formális és a természetes nyelvek közti különbség hasonló a próza és a költészet közti különbséghez, de annál nagyobbak:

költészet A szavakat a hangzásuk és a jelentésük miatt is használjuk és a teljes versben együtt hoznak létre hatást vagy érzelmi reakciókat. A félreérthetőség nem szükséges, de gyakran szándékosan alkalmazott elem.

próza A szavak szó szerinti jelentése fontosabb és a szerkezet több jelentést ad. A próza sokkal alkalmasabb elemzésre, mint a költészet, de még ez is gyakran félreérthető.

program A számítógépes program jelentése félreérthetetlen, szó szerinti és maradéktalanul megérthető a szövegelemek és a szerkezet elemzésével.

Itt van néhány javaslat a programok (és más formális nyelvek) olvasásához. Először is emlékezz arra, hogy a formális nyelvek sokkal sűrűbbek a természetes nyelveknél, szóval hosszabb idő kell az olvasásukhoz. A szerkezet is nagyon fontos, általában nem jó fentről lefelé, balról jobbra olvasni őket. Helyette inkább fejben elemezd a programot, azonosítsd a szövegelemeket és értelmezd a struktúrákat. Végül az ördög a részletekben rejlik. Kis dolgok, mint a helyesírási hibák vagy a nem megfelelő szóhasználat, amelyeket természetes nyelvek esetén néha észre sem veszünk, jelentős különbségeket eredményezhetnek a formális nyelvekben.

1.9. Az első program

Hagyományosan az első programot, amelyet egy új nyelven megírunk *Helló, Világ!* programnak hívjuk, mert összesen annyit csinál, hogy megjeleníti a képernyőn a Helló, Világ szavakat. Pythonban a szkript így néz ki: (A szkriptek esetén be fogjuk számozni a sorokat a Python utasítások bal oldalán.)

```
1 print("Helló, Világ!")
```

Ez egy példa a **print függvény** használatára, amely igazából semmit sem nyomtat papírra. Ehelyett megjeleníti az értéket a képernyőn. Ebben az esetben az eredmény így néz ki

```
Helló, Világ!
```

Az idézőjelek a programban a speciális érték elejét és végét jelzik, nem jelennek meg az eredményben.

Néhány ember a Helló, Világ program egyszerűsége alapján mond ítéletet egy nyelvről. Ebben az értelemben a Python az egyik lehető legjobb.

1.10. Megjegyzések

Ahogy a programok egyre nagyobbak és bonyolultabbak lesznek, nehezebb lesz őket olvasni. A formális nyelvek információsűrűsége nagy, így nehéz egy részt megnézni és megmondani, mit miért csinál.

Emiatt jó ötlet feljegyzéseket adni a programunkhoz természetes nyelven, ami leírja, mit csinál a program.

A **megjegyzés** egy számítógép programban szándékosan elhelyezett szöveg kizárólag emberi olvasók számára – a parancsértelmező teljes egészében kihagyja ezeket.

Pythonban a *#* szövegelem kezdi a megjegyzéseket. A sor további része figyelmen kívül lesz hagyva. Itt egy új verziója a *Helló, Világ!* programnak.

```
1 #-----
2 # Ez a demó program megmutatja, milyen elegáns a Python.
3 # Írta Hát Izsák, 2017 szeptemberében.
4 # Bárki szabadon másolhatja és módosíthatja a programot.
5 #-----
6
7 print("Helló, Világ!")      # Hát nem egyszerű!
```

Felhívnanánk a figyelmedet, hogy maradt egy üres sor a programban. Az üres sorokat a parancsértelmező szintén figyelmen kívül hagyja, de a megjegyzések és az üres sorok az ember számára olvashatóbbá teszik a programodat.

1.11. Szójegyzék

alacsony szintű nyelv (low-level language) Egy programnyelv, amelyet úgy terveztek, hogy a számítógép könnyen végre tudja hajtani. Gépi nyelvnek vagy assembly nyelvnek is hívják.

algoritmus (algorithm) Sajátos lépések sorozata problémák egy kategóriájának megoldására.

formális nyelv (formal language) Bármelyik nyelv, amelyet emberek terveztek valamilyen speciális célból, mint például a matematikai ötletek megjelenítése vagy számítógépes programok írása. Minden programnyelv formális nyelv.

forráskód (source code) Egy fordítás előtt álló magas szintű program.

futási idejű hiba (runtime error) Egy hiba, ami addig nem jelenik meg, amíg a program végrehajtása el nem kezdődik, de meggátolja a program folytatását.

hordozhatóság (portability) A programnak az a tulajdonsága, amely révén több számítógépen is futtatható.

interaktív mód (immediate mode) A Python használatának az a módja, amikor a kifejezéseket parancssorba (prompt) gépeljük és azonnal láthatjuk az eredményeket, a **szkripttel** ellentétben. Lásd még a **Python shellt** is!

kivétel (exception) A futási idejű hibák másik neve.

magas szintű nyelvek (high-level language) Programozási nyelvek, mint a Python is, amelyeket úgy terveztek, hogy az ember számára könnyen olvasható és írható legyen.

megjegyzés (comment) Információ a programban más programozók számára (vagy bárki más számára, aki olvassa a kódot). Nincs hatása a program futtatására.

nyelvtani elemzés (parse) A program vizsgálata és a szintaktikai szerkezet elemzése.

nyomkövetés (debugging) A három fajta programhiba közül bármelyiknek a megkeresésére és eltávolítására irányuló folyamat.

parancsértelmező (interpreter) A motor, amely végrehajtja a Python szkriptjeidet és a kifejezéseket.

print függvény (print function) Egy programban vagy szkriptben használt függvény, amelynek hatására a Python parancsértelmező kijelez egy értéket a kimeneti eszközén.

problémamegoldás (problem solving) A probléma megfogalmazásának, a megoldás megtalálásának és a megoldás kifejezésének folyamata.

program (program) Utasítások sorozata, amely meghatározza a számítógép tevékenységeit és a számítás menetét.

program hiba (bug) Tévedés a programban.

Python shell A Python parancsértelmező interaktív felhasználói felülete. A shell felhasználója a prompt (`>>>`) után írja a parancsokat és megnyomja az Enter gombot, hogy a parancsértelmező azonnal feldolgozza azokat. A *shell* szó a Unix-tól származik. A PyCharmban a Python Console ablak teszi lehetővé az interaktív mód használatát.

szemantika (semantics) A program jelentése.

szemantikai hiba (semantic error) Egy hiba a programban, aminek a hatására az mást csinál, mint amit a programozó szeretett volna.

szintaxis (syntax) A program szerkezete.

szintaktikai hiba (syntax error) Egy hiba a programban, amely a nyelvtani elemzést megakadályozza – és így lehetetlenné teszi a futtatást.

szövegelem (token) A program szintaktikai szerkezetének egyik alapeleme, hasonlóan a természetes nyelvek szó fogalmához.

szkript (script) Egy fájlban tárolt program (általában az, amely futtatásra kerül).

tárgy kód (object code) A fordítóprogram kimenete azután, hogy a program le lett fordítva.

természetes nyelv (natural language) Természetesen kialakult nyelv, amelyet az emberek beszélnek.

1.12. Feladatok

1. Írj egy magyar mondatot, amely szemantikája érthető, szintaktikája helytelen! Írj egy mondatot, amelynek a szintaxisa helyes, de szemantikailag helytelen!

2. A Python parancsértelmezőt használva gépeld be a `1 + 2` szöveget és nyomd meg az Enter billentyűt! A Python *kiértékeli* ezt a *kifejezést*, megjeleníti az eredményt és ad egy új promptot. A `*` a *szorzás operátor* és a `**` a *hatványozás operátor*. Kísérletezz különböző kifejezésekkel és jegyezd fel a Python parancsértelmező által megjelenített értéket!
3. Gépeld be a `1 2` szöveget és üss Entert! A Python megpróbálja kiértékelni a kifejezést, de nem tudja, mert szintaktikailag nem szabályos. Emiatt hibaüzenetet jelenít meg:

```
File "<input>", line 1
  1 2
    ^
SyntaxError: invalid syntax
```

Sok esetben a Python jelzi, hogy a szintaktikai hiba hol található, de ez nem mindig pontos és nem ad arról több információt, mi nem jó.

Így a legtöbb esetben a teher rajtad van, meg kell tanulni a szintaktikai szabályokat.

A fenti esetben a Python azért panaszkodik, mert nincs műveleti jel a számok között.

Lássuk találsz-e további példát olyan dolgokra, amelyek hibaüzenetet eredményeznek, amikor beírod őket a parancssorba. Írd le a kifejezést, amit beírtál és az utolsó sorát annak, amit a Python kiírt!

4. Gépeld be a `print("helló")` szöveget! A Python végrehajtja ezt, aminek a hatása, a h-e-l-l-ó betűk kiírása. Jegyezd meg, hogy az idézőjel, ami közrefogja a sztringet, nem része a kimentnek! Most ezt írd: `"helló"` és írd le az eredményt! Készíts feljegyzést, mikor látod és mikor nem az idézőjeleket!
5. Írd be azt, hogy `Szia` idézőjelek nélkül! A kiment ilyesmi lesz:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'Szia' is not defined
```

Ez egy futási idejű hiba, konkrétan egy `NameError` (azaz név hiba) és még pontosabban ez egy hiba amiatt, hogy a `Szia` név nem definiált. Ha még nem tudod mit jelent ez, hamarosan megtudod.

6. Írd `6 + 4 * 9` kifejezést a Python prompt után és nyomj Entert! Jegyezd fel, mi történik!

Most készíts egy Python szkriptet az alábbi tartalommal:

```
1 6 + 4 * 9
```

Mi történik, ha futtatod a szkriptet? Most változtasd meg a szkript tartalmát erre:

```
1 print(6 + 4 * 9)
```

és futtasd újra!

Mi történik ekkor?

Bármikor, amikor egy *kifejezést* begépelünk a Python parancssorba, az kiértékelődik és az eredmény *automatikusan* látható lesz a következő sorban. (Mint egy számológépen, ha begépeled a fenti kifejezést 42-t fogsz kapni.)

A szkript ettől eltérő. A kifejezések kiértékelése nem jelenik meg automatikusan, az eredmény láthatóvá tételéhez a **print** függvényt kell használnunk.

A `print` függvény használata alig szükséges az interaktív módú parancssorban.

2. fejezet

Változók, kifejezések, utasítások

2.1. Értékek és típusok

Az értékek – számok, betűk, stb. – azon alapvető elemek közé tartoznak, amelyekkel a programok a működésük során dolgoznak. A korábbiakban a 4-es ($2 + 2$ eredménye) és a "Helló, Világ!" értékeket láthattuk. (Az ilyen önmagukat definiáló értékeket **konstansoknak** vagy **literáloknak** nevezzük.)

Az értékeket **osztályokba**, illetve **adattípusokba**, röviden típusokba sorolhatjuk. A 4-es egész szám, a "Hello, World!" pedig sztring, vagy másként mondva szöveges típusú. A sztring konstansok számunkra és az értelmező számára is könnyen felismerhetők arról, hogy idézőjelben szerepelnek.

Egy érték típusáról a **type** függvény segítségével bizonyosodhatunk meg. (Az interaktív mód a PyCharmon belül a **Tools** menü **Python Console** menüpontját választva nyílik meg. Az utasításokat a prompt (`>>>`) után kell begépelni.)

```
>>> type("Helló, Világ!")
<class 'str'>
>>> type(17)
<class 'int'>
```

A sztringek az **str** osztályba, az egész számok az **int** osztályba, a tizedespontot tartalmazó számok a **float** osztályba tartoznak. Utóbbi a számok ábrázolási formájára utal, ugyanis az ilyen számokat a gép *lebegőpontos* alakban tárolja. Az *osztály* és a *típus* fogalmakat eleinte szinonimáknak fogjuk tekinteni, majd egy későbbi fejezetben foglalkozunk mélyebben az osztályokkal.

```
>>> type(3.2)
<class 'float'>
```

Mi a helyzet az olyan értékekkel, mint a "17" vagy a "3.2"? Számoknak látszanak, de idézőjelek között állnak, akár a sztringek.

```
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

Ezek bizony sztringek!

A Python sztringek állhatnak aposztrófok (`'`), idézőjelek (`"`), tripla aposztrófok (`' '`) és tripla idézőjelek (`" "`) között is.

```
>>> type('Ez egy sztring.')
<class 'str'>
>>> type("Ez is egy sztring, ")
<class 'str'>
>>> type("""és ez is, """)
<class 'str'>
>>> type(''és még ez is...'')
<class 'str'>
```

Az idézőjelek által közrefogott szöveg aposztrófot is tartalmazhat ("Pista bá' mondta."), az aposztrófok által határolt szöveg pedig tartalmazhat idézőjelet is ("Csodás", megint elromlott!).

A tripla aposztróffal és a tripla idézőjellel körülvett sztringeket egyaránt háromszorosan idézőjelezett sztringeknek fogjuk nevezni. Az ilyen formában megadott sztringek tartalmazhatnak aposztrófot és idézőjelet is.

```
>>> print(''''Adjon Isten!" - mondta Pista bá' az embernek.'''')
"Pista bá' mondta."
>>>
```

A háromszorosan idézőjelezett szövegek több sort is felöllelhetnek:

```
>>> message = """Ez a szöveg
több sorban
jelenik meg."""
>>> print(message)
Ez a szöveg
több sorban
jelenik meg.
>>>
```

A Python számára teljesen mindegy, hogy a fentiek közül melyik jelölést használjuk a sztringek határainak jelzésére, a háttérben, a programszöveg elemzése után, már egységes tárolás valósul meg. Ráadásul a határoló jelek nem is tartoznak az értékhez, így nem is lesznek eltárolva. A megjelenítésnél az értelmező „csomagolja be” idézőjelek közé a szövegeket.

```
>>> 'Ez egy sztring.'
'Ez egy sztring.'
>>> """Ez is egy sztring."""
'Ez is egy sztring.'
```

Amint látható, a Python most aposztrófok közé tette a sztringeket, de vajon mi történik, ha a sztring aposztrófot tartalmaz?

Foglalkozzunk most egy kicsit a vessző használatával. Vegyük például a 3,12-es értéket. Pythonban ez nem érvényes szám, ettől függetlenül használhatjuk a programban, csak más jelentéssel bír.

```
>>> 3.12
3.12
>>> 3,12
(3, 12)
```

Nem erre számítottál? A Python értelmezése szerint egy *értékpárt* adtunk meg. Később majd tanulunk ezekről is, most azonban elegendő annyit megjegyezni, hogy a számok írásakor se vessző, se szóköz ne kerüljön a számjegyek közé. Tizedesjelként – az angol helyesírás szabályainak megfelelően – pontot kell használnunk.

Nem véletlenül említettük az előző fejezetben, hogy a formális nyelvekben szigorúak a szintaktikai szabályok, és még a legkisebb hiba is jelentős eltérést okozhat a kimenetben.

2.2. Változók

A programnyelvek egyik legfontosabb tulajdonsága, hogy képesek módosítani a változókat. A **változó** lényegében egy név, amely egy értékre utal.

A változókhoz értékadó kifejezés segítségével rendelhetünk értéket:

```
>>> uzenet = "Mi újság?"
>>> n = 17
>>> pi = 3.14159
```

Ebben a példában három értékadás történt. Az elsőnél az `uzenet` nevű változóhoz rendeltük hozzá a `"Mi újság?"` értéket. A második esetben az `n` nevű változó kapott 17-es, egész értéket, míg a harmadik értékadásnál egy lebegőpontos értéket, a `3.14159`-et rendeltük a `pi` nevezetű változóhoz.

Az **értékadás művelet** jele az egyenlőségjel (`=`), nem keverendő az **egyenlőségvizsgálat művelettel**, melynek jele két egyenlőségjel (`==`). Az értékadás során a műveleti jel jobb oldalán álló **értéket** a bal oldalán álló **névhez** rendeljük.

Az alábbi kifejezés éppen ezért hibás:

```
>>> 17 = n
File "<input>", line 1
SyntaxError: can't assign to literal
```

Javaslat: A program olvasása vagy írása közben sose mondd, hogy `n` egyenlő `17`-tel, mondd úgy, hogy `n` legyen egyenlő `17`-tel.

Papíron a változókat nevükkel, értékükkel, és egy a névtől az értékre mutató nyíllal szokás reprezentálni. Az ilyen ábrákat a változók állapotáról készült pillanatképek tekinthetjük, mivel azt mutatják, hogy egy adott időpillanatban milyen értéket hordoznak az egyes változók.

Az alábbi diagram az értékadó operátor hatását mutatja:

```
uzenet  —————> "Mi újság?"
n        —————> 17
pi       —————> 3.14159
```

Amikor az értelmezőt egy változó kiértékelésére utasítjuk, akkor azt az értéket határozza meg, amely aktuálisan a változóhoz tartozik.

```
>>> message
'Mi újság?'
>>> n
17
>>> pi
3.14159
```

A változókat különböző értékek megjegyzésére használjuk a programban, például tárolhatjuk egy focimeccs aktuális állását. A változók ugye változók, tehát időről, időre módosulhat az értékük, mint ahogy egy focimeccsen is változhat az eredményjelző táblán megjelenített állás. A változóhoz rendelt értékeket később felülírhatjuk, tehát új értéket rendelhetünk a változóhoz. *(Ez eltér a matematikában megszokottól. Ott, ha egy "x" 3-as értéket kapott, akkor nem változtathatjuk meg azt a számítás során.)*

```
>>> nap = "Csütörtök"
>>> nap
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
'Csütörtök'  
>>> nap = "Péntek"  
>>> nap  
'Péntek'  
>>> nap = 21  
>>> nap  
21
```

A fenti példában háromszor is megváltoztattuk a `nap` változó értékét, az utolsó esetben ráadásul egy eltérő típusú értéket rendeltünk a változóhoz.

A programok jelentős hányada különböző értékek megjegyzésén, és az értékek megfelelő módosításán alapul. Például tároljuk a nem fogadott hívások számát a telefonon, majd egy újabb nem fogadott hívásnál frissítjük az értéket.

2.3. Változónevek, kulcsszavak

A **változónevek** tetszőleges hosszúságúak lehetnek, tartalmazhatnak betűket és számjegyeket is, de mindenképpen betűvel vagy aláhúzás karakterrel kell kezdődniük. A változónevek nagybetűket is tartalmazhatnak, de ezzel ritkán élünk. Ha mégis használjuk a nagybetűket is, akkor tartsuk észben, hogy a kis és nagybetűk különbözőnek számítanak, tehát a „Pali” és „pali” két különböző változó.

Az aláhúzás karaktert (`_`), ami szintén megjelenhet a nevekben, gyakran használjuk a több szóból álló nevek tagolására, például `sajat_nev` vagy `csokolade_tipusa`.

Néhány esetben az aláhúzással kezdődő nevek különleges jelentést hordoznak, ezért kezdőként biztonságosabb betűvel kezdődő nevet választani.

A hibás névválasztás szintaktikai hibát eredményez:

```
>>> 76harsona = "nagy parádé"  
SyntaxError: invalid syntax  
>>> tobb$ = 1000000  
SyntaxError: invalid syntax  
>>> class = "Computer Science 101"  
SyntaxError: invalid syntax
```

A `76harsona` név azért hibás, mert nem betűvel kezdődik. A `tobb$` pedig azért, mert a dollár jel nem érvényes betű (illegális karakter). De mi a probléma a `class` szóval?

Nos, a `class` a Python nyelvben **kulcsszó**. A kulcsszavak határozzák meg a nyelv szintaktikai szabályait, struktúráit, ezért változónévként nem használhatóak. (A PyCharm interaktív módját használva a hibaüzenet nem is jelenik meg azonnal, ugyanis még vár a `class` kulcsszónak megfelelő folytatásra.)

A Pythonban harmincevalahány kulcsszó van, a konkrét érték a Python verziójától függően változhat:

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

Érdeemes ezt a listát kéznél tartani. Ha az értelmező „panaszodik” egy változónév miatt, de nem tudod miért, akkor nézd meg, rajta van-e a listán a választott név.

A programozók általában olyan nevet választanak, mely – az emberi olvasó számára – utal a változó szerepére, hogy később könnyebb legyen felidézni azt. Az ilyen „beszédes” nevek a program dokumentálását is segítik.

Figyelem: A kezdő programozók számára zavaró lehet az emberek számára hordozott jelentés és a gép számára hordozott jelentés megkülönböztetése. Tévesen azt gondolhatják, hogy csak azért, mert egy változó neve `pi` varázslatos módon kiszámítódik majd az átlagérték, illetve 3,14159 lesz a `pi` értéke. Ez nem így van! A számítógép nem érti a változónevek jelentését.

Biztosan találkozol majd olyan oktatókkal, akik a kezdők oktatásánál szándékosan nem választanak értelmes változóneveket. Nem mintha ez jó programozó stílus lenne, de tudatosítani akarjuk, hogy neked – a programozónak – kell megírni az átlagot kiszámoló kódrészletet, és a `pi` változó értékét is neked kell beállítani arra az értékre, amit tárolni akarsz benne.

2.4. Utasítások

Az **utasítás** olyan parancs, amelyet a Python értelmező képes végrehajtani. Eddig csak az értékadó utasítással ismerkedtünk meg, de hamarosan látni fogunk más fajta utasításokat is, többek közt a `while`, a `for`, az `if` és az `import` utasításokat is.

Amikor begépelünk egy utasítást a parancssorba, a Python végrehajtja azt. Az utasítások viszont nem mindig szolgáltatnak eredményt.

2.5. Kifejezések kiértékelése

A **kifejezések** értékekből (konstansokból), változókból, műveleti jelekből és függvényhívásokból épülhetnek fel. Amikor begépelünk egy kifejezést a Python prompt után, akkor az értelmező **kiértékeli**, majd megjeleníti az eredményt:

```
>>> 1 + 1
2
>>> len("helló")
5
```

Ebben a példában a `len` egy beépített Python függvény, mely egy sztring hosszát adja vissza. Korábban már láttuk a `print` és a `type` függvényeket, szóval ez már a harmadik példánk a függvényekre!

A **kifejezés kiértékelése** egy értéket határoz meg, ezért fordulhatnak elő kifejezések az értékadó utasítások jobb oldalán. Az értékek önmagukban állva egyszerű kifejezések csakúgy, mint a változók.

```
>>> 17
17
>>> y = 3.14
>>> x = len("helló")
>>> x
5
>>> y
3.14
```

2.6. Műveleti jelek és operandusok

A **műveleti jelek**, más szóval **operátorok**, különböző műveleteket (pl.: összeadás, szorzás, osztás) jelölő speciális nyelvi elemek. Az **operandusok** pedig azok az értékek, melyeken a műveleteket elvégezzük.

Az alábbi kifejezések mindegyike helyes Pythonban, jelentésük pedig többé-kevésbé világos:

```
20+32    ora-1    ora*60+perc    perc/60    5**2    (5+9)*(15-7)
```

A +, -, * és a zárójelek Pythonban is a matematikában megszokott jelentéssel bírnak. A csillag (*) a szorzás, a két csillag (**) a hatványozás jele.

```
>>> 2 ** 3
8
>>> 3 ** 2
9
```

A műveletek elvégzése előtt az operandusok helyén álló változónevek az értékükkel helyettesítődnek.

Az összeadás, a kivonás, a szorzás és a hatványozás is úgy működik, ahogy az várható.

Lássuk az osztást. Váltsunk át 645 percet órákra:

```
>>> percek = 645
>>> orak = percek / 60
>>> orak
10.75
```

Hoppá! Python 3-ban a / jellel végrehajtott osztás mindig lebegőpontos számot eredményez. Elképzelhető, hogy mi a teljes órák és a fennmaradó percek számát szeretnénk volna megtudni. A Python egy másik osztás operátort is biztosít számunkra, melynek jele a //. Ezt a fajta osztást **egész osztásnak** nevezzük, mert mindig egész értéket szolgáltat. Ha az osztás eredménye nem egész érték, akkor a képzeletbeli számegyenes bal oldala felé eső egész érték lesz az egész osztás eredménye. A 6 // 4 eredménye tehát 1. A -6 // 4 értéke talán meglepő lehet.

```
>>> 7 / 4
1.75
>>> 7 // 4
1
>>> percek = 645
>>> orak = percek // 60
>>> orak
10
```

Mindig gondosan válasszuk meg a megfelelő osztás műveletet! Ha szükség van a tizedesjegyek megőrzésére is, akkor a „sima” osztás jelet kell használnunk, mely pontosan adja vissza az osztás eredményét.

2.7. Típuskonverziós függvények

Ebben a részben újabb három Python függvénnyel, az `int`, `float` és az `str` függvényekkel ismerkedünk meg, melyek a nekik átadott paramétereket rendre `int`, `float` és `str` típusúvá alakítják át. Ezeket a függvényeket **típuskonverziós** függvénynek nevezzük.

Az `int` függvény valós számot, vagy sztringet vár bemeneti paraméterként, és egész értékke alakítja át. Ha tizedesjegyeket tartalmaz a konvertálandó érték, akkor a függvény a konvertálás során elhagyja azokat, vagyis *vágást* végez. Nézzük is néhány példát:

```
>>> int(3.14)
3
>>> int(3.9999)           # Nem a legközelebbi egészre kerekít!
3
>>> int(3.0)
3
>>> int(-3.999)           # Az eredmény 0-hoz esik közelebb.
-3
>>> int(percek / 60)
10
>>> int("2345")           # Egy sztringet alakít egész számmá.
2345
>>> int(17)                # Akkor is működik, ha a szám eredetileg is egész.
17
>>> int("23 uveg")
```

Az utolsó eset, nem úgy néz ki, mint egy szám. Mire számíthatunk?

```
Traceback (most recent call last):
File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 uveg'
```

A float típuskonverziós függvény egész és valós számot, valamint szintaktikailag megfelelő sztringet képes valós számmá alakítani:

```
>>> float(17)
17.0
>>> float("123.45")
123.45
```

Az str függvény a paramétereit karakterlánccá alakítja át:

```
>>> str(17)
'17'
>>> str(123.45)
'123.45'
```

2.8. Műveletek kiértékelési sorrendje

Ha egy kifejezésben több műveleti jel is szerepel, akkor a kiértékelés sorrendjét a **műveletek erőssége**, más szóval a **műveletek precedenciája** határozza meg. A Pythonban az aritmetikai műveletek erőssége megfelel a matematikában megszokottnak.

1. A zárójelnek van a legnagyobb precedenciája. Használhatjuk a műveleti sorrend megváltoztatására, ugyanis a zárójelben álló kifejezések lesznek először kiértékelve. Például a $2 * (3-1)$ az 4, $(1+1) ** (5-2)$ az 8. Alkalmazásukkal javíthatjuk az kifejezések olvashatóságát is: $(perc * 100) / 60$.
2. A hatványozás a második legerősebb művelet, szóval a $2**1+1$ az 3 és nem 4, a $3*1**3$ pedig 3 és nem 27.
3. Az szorzás és osztás azonos erősségű művelet. Magasabb precedenciával bírnak, mint a szintén egyforma erősségű összeadás és kivonás. A $2*3-1$ tehát 5 és nem 4, a $5-2*2$ pedig 1 és nem 6.
4. Az azonos erősségű operátorok kiértékelése balról jobbra haladva történik. Algebrai kifejezéssel élve: balasszociatívak. Vegyük például a $6-3+2$ kifejezést. A kiértékelés folyamatában a kivonást végezzük el először, ami 3-at eredményez, majd ehhez adunk 2-t, a végeredmény tehát 5. Ha jobbról balra értékelnénk ki, akkor igazából a $6-(3+2)$ kifejezés értékét határoznánk meg, ami 1.

- A hatványozás bizonyos történelmi oknál fogva kivételt jelent a balasszociativitás szabálya. Ha a `**` művelet előkerül, akkor jobb kitenni a zárójeleket, hogy biztosan abban a sorrendben menjen végbe a kiértékelés, ahogy azt elterveztük.

```
>>> 2 ** 3 ** 2      # A legjobboldalibb ** hajtódik végre először!  
512  
>>> (2 ** 3) ** 2    # Használjunk zárójeleket a megfelelő kiértékelési_  
↪ sorrend biztosítására!  
64
```

A Python interaktív módja nagyszerű lehetőség az ezekhez hasonló kifejezések felfedezésére, tapasztalatok szerzésére.

2.9. Sztringkezelő műveletek

A matematikai műveletek általában nem alkalmazhatók szövegekre, még akkor sem, ha történetesen számnak néznek ki. Az alábbi kifejezések hibásak (feltételezve, hogy az üzenet típusa sztring.)

```
>>> uzenet - 1      # Error  
>>> "Szia" / 123    # Error  
>>> uzenet * "Szia" # Error  
>>> "15" + 2        # Error
```

Érdekes módon a `+` jel sztringek esetében is működik, de olyankor nem az összegzés, hanem az **összefűzés** műveletet jelöli. Az összefűzéssel két sztringet kapcsolhatunk egymás után.

Például:

```
>>> izesites = "lekváros"  
>>> pekaru = "bukta"  
>>> izesites + pekaru  
'lekváros bukta'
```

Az összefűzés eredménye a lekváros bukta. A bukta előtt álló szóköz is a sztring része, azért kell, hogy az eredményben is legyen egy szóköz a két szó között.

A `*` szintén működik sztringekre, az **ismétlés** művelet jele. Például a `'Móka' * 3` eredménye a `'MókaMókaMóka'`. A művelet egyik operandusának sztringnek, a másiknak egész számnak kell lennie.

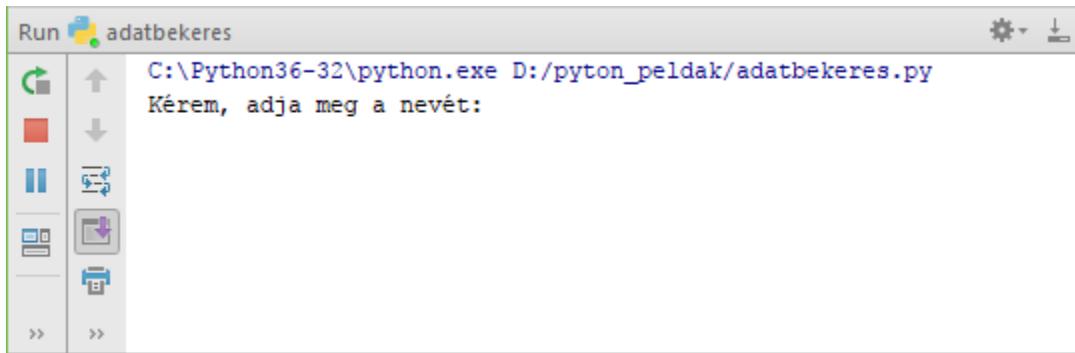
Bizonyos szempontból a `+` és `*` előbbi látott értelmezése analóg a matematikai értelmezéssel. A $4 \cdot 3$ egyenlő a $4+4+4$ -gyel, így a `'Móka' * 3` esetében is számíthatunk arra, hogy `'Móka' + 'Móka' + 'Móka'` lesz az eredmény, és az is. Másrésztől jelentős eltérések is vannak az összeadás és a szorzás, valamint az összefűzés és az ismétlés műveletek között. Tudsz-e olyan tulajdonságot találni, amely igaz az összeadásra és a szorzásra, de nem igaz az összefűzésre és az ismétlésre?

2.10. Adatbekérés

A Pythonban egy beépített függvény segítségével kérhetünk adatokat a felhasználóktól:

```
>>> n = input("Kérem, adj meg a nevét: ")
```

Amennyiben interaktív módban adjuk ki az utasítást a megjelenő prompt jel után lehet beírni a választ. Ha szkriptként futtatjuk, akkor az alábbihoz hasonló ablak nyílik meg a PyCharmban:



A felhasználó ebbe az ablakba gépelheti be a nevét. Az `input` függvény pedig – az Enter leütését követően – visszaadja a felhasználó által megadott szöveget, amelyet az `n` változóhoz rendelünk.

Ha a felhasználó korát kérnénk be, akkor is sztringet (pl.: "17") kapnánk vissza. A programozó feladata átalakítani egész vagy valós számmá a korábban látott `int` vagy `float` típuskonverziós függvények valamelyikével.

2.11. Függvények egymásba ágyazása

Az eddigiek során külön-külön néztük meg a programokat felépítő elemeket (változókat, kifejezéseket, utasításokat és függvényhívásokat), nem foglalkozva azzal, hogyan kell ezeket összekapcsolni.

A programnyelvekben az egyik leghasznosabb dolog, hogy kis „építőköveket” kombinálva nagyobb egységeket hozhatunk létre.

Például már tudunk adatot bekérni a felhasználótól, át tudunk alakítani egy sztringet valós számmá, tudunk összetett kifejezéseket készíteni és értékeket megjeleníteni. Gyűrjük most mindezt egybe! Írjunk egy programot, mely egy kör sugarát kéri be a felhasználótól, majd az alábbi képlet alapján kiszámolja a kör területét.

$$terület = r^2\pi$$

Először valósítsuk meg négy külön lépésben:

```
>>> bemenet = input("Mekkora a kör sugara? ")
>>> sugar = float(bemenet)
>>> terület = 3.14159 * sugar**2
>>> print("A terület ", terület)
```

Most vonjuk össze az első kettőt, illetve a második két sort is:

```
>>> sugar = float(input("Mekkora a kör sugara?"))
>>> print("A terület ", 3.14159 * sugar**2)
```

Ha nagyon trükkösek szeretnénk lenni, akár egyetlen sorban is megoldhatjuk:

```
>>> print("A terület ", 3.14159 * float(input("Mekkora a kör sugara? "))**2)
```

Az ennyire tömör kód a földi halandók számára nehezen olvasható, de jól mutatja hogyan építhetők nagyobb egységek a mi kis „építőköveinkből”.

Ha bármikor felmerülne benned a kérdés, hogy egymásba ágyazd-e a függvényeket vagy inkább külön lépésekben old-e meg a feladatot, mindig válaszd azt a megoldást, ami könnyebben érthető. Az előbbi példánál mi az első, a négy külön lépést tartalmazó megoldásra tennénk le a voksunkat.

2.12. A maradékos osztás művelet

A **maradékos osztás** művelete egész számokon végezhető el (illetve egész típusú kifejezéseken) és azt adja meg, hogy a műveleti jel bal oldalán álló számot a jobb oldalán álló számmal osztva mennyi lesz a maradék. Pythonban a maradékos osztás jele a százalék jele (%). A szintaktikája azonos a korábban látott matematikai operátorokéval, a szorzással azonos erősségű.

```
>>> e = 7 // 3      # Egész osztás
>>> e
2
>>> m = 7 % 3
>>> m
1
```

Tehát a 7-ben a 3 kétszer van meg, és 1 a maradék.

A maradékos osztás meglepően hasznos tud lenni. Ellenőrizhető, hogy egy szám osztható-e a másikkal. Ha $x \% y$ nullát ad eredményül, akkor az x osztható y -nal.

Meghatározhatjuk egy szám utolsó számjegyét vagy számjegyeit. Például $x \% 10$ az x utolsó számjegyét, az $x \% 100$ pedig az utolsó két számjegyét adja vissza (tíz-es számrendszerben).

Felettébb hasznos az átváltásoknál is, mondjuk másodpercről órákra, percekre és a fennmaradó másodpercekre. Írjunk is egy programot, mely bekéri a másodpercek számát a felhasználótól és elvégzi az átalakítást.

```
1 összes_masodperc = int(input("Összesen hány másodperc? "))
2 orak = összes_masodperc // 3600
3 megmaradt_masodpercek = összes_masodperc % 3600
4 percek = megmaradt_masodpercek // 60
5 megmaradt_masodpercek_a_vegen = megmaradt_masodpercek % 60
6
7 print("Órák=", orak, " Percek=", percek,
8       " Másodpercek=", megmaradt_masodpercek_a_vegen)
```

2.13. Szójegyzék

adattípus (data type) Egy értékhalmoz. Az értékek típusa határozza meg, hogy milyen műveletek végezhetők az értékeken, hogyan használhatók fel a kifejezésekben. A típusok közül eddig az egész (`int`), a valós (`float`) és a sztring (`str`) típusokkal találkoztunk.

egész osztás (floor division, integer division) Egy speciális osztás művelet, mely mindig egész értéket eredményez. Ha az osztás eredménye, a hányados egész, akkor az egész osztás eredménye maga a hányados, különben a hányadoshoz legközelebbi, nála kisebb egész szám. Jele: `//`.

érték (value) Egy szám vagy egy szöveg (vagy más egyéb, ami később kerül ismertetésre). Az értékek tárolhatók változóban, vagy szerepelhetnek kifejezésekben.

értékkadás jele (assignment token) Pythonban az egyenlőségjel (`=`) az értékkadás művelet jele. Nem keverendő az *egyenlőségvizsgálat* művelettel (`==`), mely két érték összehasonlítására szolgál.

értékkadó utasítás (assignment statement) Az értékkadó utasítás segítségével egy névhez (változóhoz) rendelhetünk értéket. Az értékkadás jel (`=`) bal oldalán kötelezően egy név áll, jobb oldalán egy kifejezés. A kifejezést a Python értelmező kiértékeli, és hozzárendeli a megadott névhez. A kezdő programozók számára gyakran problémát okoz a bal és jobb oldal közti különbség megértése. Az

```
n = n + 1
```

kifejezésben az n teljesen más szerepet játszik a műveleti jel két oldalán. A jobb oldalon az n egy *érték* mely az $n+1$ kifejezés része. A Python értelmező a kifejezést kiértékelése során keletkező értéket rendeli hozzá a bal oldalon álló névhez.

float Egy Python típus valós számok kezelésére. A valós számok a háttérben úgynevezett *lebegőpontos* alakban kerülnek tárolásra. A `float` típusú értékek használatánál ügyelni kell a kerekítésből adódó hibákra, tartasuk észben, hogy közelítő értékekkel dolgozunk.

int Pozitív és negatív egész számok tárolására szolgáló Python adattípus.

kiértékelés (evaluate) Egy kifejezés egyetlen értékre való egyszerűsítése a benne szereplő műveletek végrehajtásával.

kifejezés (expression) Változók, műveleti jelek és értékek kombinációja, mely egyetlen értéket reprezentál.

kulcsszó (keyword) Egy fenntartott szó melyet a fordító használ a program elemzésénél. A fenntartott szavak, mint például az `if`, `def`, vagy a `while` nem használható változónévként.

maradék oszta (modulus operator) Egy egész számokra alkalmazható művelet, mely két szám osztása során keletkezett maradékot adja meg. Jele a százalék (%).

műveleti jel (operator) Egy szimbólum, ami egy egyszerű számítási műveletet (pl. összeadást, szorzást, összefűzést) reprezentál.

operandus (operand) Egy olyan érték, melyen a művelet kifejti a hatását.

operátor (operator) Lásd: műveleti jel.

összefűzés (concatenate) Két sztring összekapcsolása.

precedenciarendszer (rules of precedence) Egy olyan szabályrendszer, amely meghatározza, hogy a több műveleti jelet és operátort is tartalmazó kifejezéseknél milyen sorrendben kell alkalmazni a műveleteket a kifejezés értékének meghatározásához.

str Szövegek (sztringek) tárolására szolgáló Python adattípus.

utasítás (statement) Olyan parancs, melyet a Python értelmezője képes végrehajtani. Eddig csak az értékadó utasítással ismerkedtünk meg, hamarosan látni fogjuk az `import` és a `for` utasítást.

változó (variable) Egy olyan név, amely egy értéket reprezentál.

változónév (variable name) Egy változónak adott név. Pythonban a név betűk és számjegyek, kötelezően betűvel kezdődő, sorozata. A legjobb programozói gyakorlat, ha a változó neve utal a programban betöltött szerepére, *öndokumentálóvá* téve a programot. Python 2.7-es változatban az angol ábécé kis- és nagybetűi és az aláhúzás karakter (a..z, A..Z, _) számítanak betűnek. A Python 3.0-s változatától kezdve ékezetes karakterek is használhatók.

2.14. Feladatok

1. Tárold el a *Lustaság fél egészség.* mondat minden szavát külön változóban, majd jelenítsd meg egy sorba a `print` függvény használatával.
2. Zárójelezd úgy a `6 * 1 - 2` kifejezést, hogy a 4 helyett -6 legyen az értéke.
3. Tegyél megjegyzés jelet egy olyan sor elé, amely korábban már működött. Figyeld meg, mi történik, amikor újra futtatod a programot!
4. Indítsd el a Python értelmezőt, és gépeled be a `Pista + 4` kifejezést, majd üss egy Entert. Az alábbi hiba jelenik meg:

```
NameError: name 'Pista' is not defined
```

Rendelj olyan értéket `Pista` változóhoz, hogy a `Pista + 4` kifejezés értéke 10 legyen.

5. Írj programot, amely meghatározza, mennyi lesz egy betét értéke a futamidő végén, ha 10000 Ft-t helyezünk betétbe 8%-os névleges kamatláb mellett. Az évközi kamatozások száma (m) 12. Az évek számát, vagyis a t értékét a felhasználótól kérje be a program. A futamidő végén nézett értéket (FV) az alábbi képlet alapján számold:

$$FV = C \cdot \left(1 + \frac{r}{m}\right)^{mt}$$

Ahol,

- C : alaptőke
- r : éves névleges kamatláb
- m : évközi kamatozások száma
- t : évek száma

6. Számold ki az alábbi kifejezések értékét fejben, majd ellenőrizd a Python értelmező segítségével:

(a) `>>> 5 % 2`

(b) `>>> 9 % 5`

(c) `>>> 15 % 12`

(d) `>>> 12 % 15`

(e) `>>> 6 % 6`

(f) `>>> 0 % 7`

(g) `>>> 7 % 0`

Mi történt az utolsó példánál, és miért? Ha mindegyikre helyesen válaszoltál az utolsó kivétellel, akkor ideje továbbhaladni. Ellenkező esetben írd fel saját példákat, szánj időt a maradékos osztás tökéletes megértésére.

7. Jelenleg pontosan 14 óra van. Beállítunk egy ébresztőórát úgy, hogy 51 órával később csörögjön. Hány órákor fog az ébresztőóra megszólalni? (Segítség: Ha túlzottan vonz a lehetőség, hogy az ujjaidon számold ki, akkor 51 helyett dolgozz 5100-zal.)
8. Írj egy Python programot az előző feladat általános megoldására. Kérd be a felhasználótól az aktuális időt (csak az órákat) és azt, hogy hány órával később szólaljon meg az ébresztőóra, majd jelenítsd meg a képernyőn, hogy hány órákor fog megszólalni az ébresztőóra.

3. fejezet

Helló, kis teknőcök!

Pythonban sok *modul* van, amelyek hatalmas mennyiségű kiegészítő lehetőséget nyújtanak saját programjainkhoz. Ezek között van például az e-mail küldés vagy akár a weblap letöltés. Amelyiket ebben a fejezetben megnézzük az lehetővé teszi, hogy teknőcöket hozzunk létre, amelyek alakzatokat és mintázatokat rajzolnak meg.

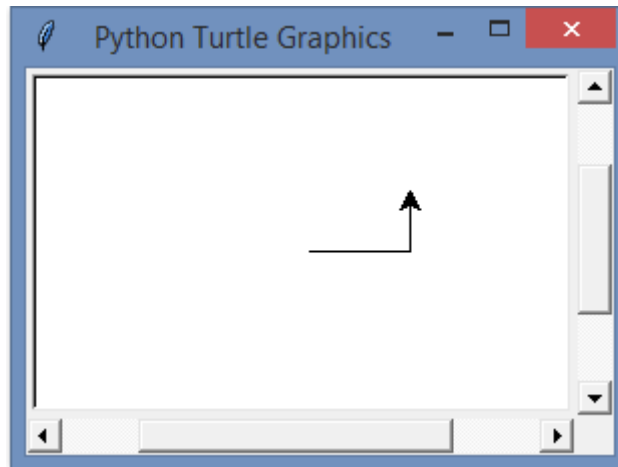
A teknőcök mókásak, de az igazi célja ennek a fejezetnek, hogy egy kicsivel több Pythont tanuljunk és fejlesszük az *algoritmikus gondolkodásunkat* vagyis hogy *gondolkozzunk úgy, mint egy informatikus*. Az itt bemutatott Python jelentős része később részletesen is ki lesz fejtve.

3.1. Az első teknőc programunk

Írjunk egy pár sornyi Python programot, hogy létrehozzunk egy új teknőcöt és kezdjünk el rajzolni vele egy téglalapot! (A változót, amely az első teknőcünkre hivatkozik, hívjuk Sanyi-nak, de más nevet is választhatunk, ha követjük az előző fejezet névadási szabályait.)

```
1  import turtle                # Lehetővé teszi a teknőc használatát
2  ablak = turtle.Screen()      # Hozz létre egy játszóteret a teknőcnek!
3  Sanyi = turtle.Turtle()      # Hozz létre egy teknőcöt Sanyi néven!
4
5  Sanyi.forward(50)            # Sanyi menjen 50 egységet előre!
6  Sanyi.left(90)               # Sanyi forduljon 90 fokot!
7  Sanyi.forward(30)            # Rajzold meg a téglalap második oldalát!
8
9  ablak.mainloop()             # Várj, amíg a felhasználó bezárja az ablakot!
```

Amikor futtatjuk a programot, egy új ablak ugrik fel:



Van itt pár dolog, amit meg kell értenünk a programmal kapcsolatban.

Az első sor megmondja a Pythonnak, hogy töltsé be a `turtle` nevű modult. Ez a modul két új típust hoz be a látótérbe, amelyeket ezután használhatunk: a `Turtle`, azaz teknőc típust és a `Screen`, azaz képernyő típust. A `turtle.Turtle` szövegben a pont jelölés azt jelenti, hogy „a *Turtle* típus, ami a *turtle* modulban van definiálva”. (Megjegyzés: a Python érzékeny a kis és nagy betűkre, így a modul neve *t*-vel írva különbözik a `Turtle` típus nevétől.)

Aztán létrehozuk és megnyitjuk azt, amit képernyőnek hívunk (talán lehetett volna ablaknak is hívni) és ezt hozzárendeljük az ablak változóhoz. Minden ablak tartalmaz egy **vászon** nevű részt, ami az a terület az ablakon belül, amire rajzolhatunk.

A 3. sorban létrehozuk a teknőcöt. A `Sanyi` nevű változó fog hivatkozni erre a teknőcre.

Így tehát az első három sor beállítja a szükséges dolgokat, tehát most készen állunk arra, hogy a teknőccel rajzoltassunk valamit a vászonra.

Az 5-7. sorokban arra utasítjuk a `Sanyi` objektumot, hogy mozduljon meg és forduljon el. Ezt `Sanyi` **metódusainak** aktiválásával, vagyis a **hívás** folyamatával tesszük meg – ezek utasítások, amelyekre minden teknőc tudja hogyan reagáljon.

Az utolsó sor is fontos szerepet játszik: az `ablak` változó hivatkozik az ablakra, ahogy fentebb bemutattuk. Ha meghívjuk a `mainloop` nevű metódusát, belép egy állapotba, ahol egy eseményre vár (mint például a billentyűleütés vagy egér mozgatás és kattintás).

Egy objektumnak számtalan metódusa lehet – ezek dolgok, amit meg tud tenni – és emellett **attribútum** halmazzal is rendelkezhet – (néha ezeket *tulajdonságoknak* hívjuk). Például minden egyes teknőcnek van egy *szín* tulajdonsága. A `Sanyi.color("red")` metódushívás pirossá teszi `Sanyi`-t, és amit ő rajzol majd az is piros lesz. (Megjegyzés a *color* vagyis *szín* szó az amerikai angol szabályai szerint van írva.)

A teknőc színe, a tollának vastagsága, a teknőc pozíciója az ablakon belül, az hogy merrefelé néz és így tovább ezek mind az ő aktuális **állapotának** részei. Ehhez hasonlóan, az `ablak` objektumnak van háttérszíne, és egy kis szöveget tartalmaz a címsorában, van mérete és pozíciója. Ezek mind az `ablak` objektum állapotának részei.

A metódusok mind azért léteznek, hogy lehetőségünk legyen a teknőc és az `ablak` objektumok módosítására. Mi csak egy párat fogunk bemutatni. A következő programban csak azokhoz a sorokhoz írtunk megjegyzést, amelyek különböznek az előző példától (és a teknőcnek most más nevet adtunk):

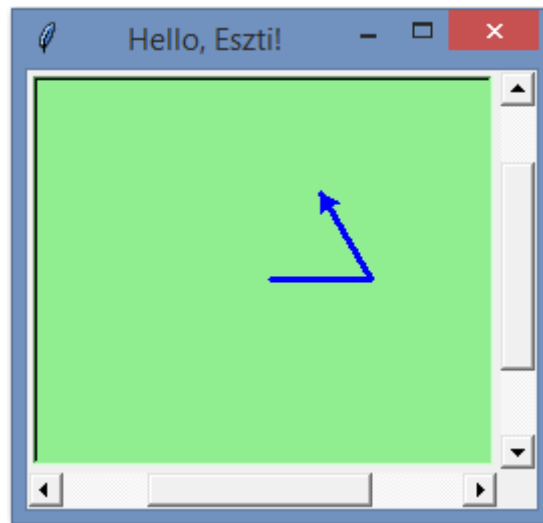
```
1 import turtle
2 ablak = turtle.Screen()
3 ablak.bgcolor("lightgreen")      # Állítsd be az ablak háttérszínét!
4 ablak.title("Hello, Eszti!")    # Állítsd be az ablak címét!
5
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6 Eszti = turtle.Turtle()
7 Eszti.color("blue")           # Mond meg Esztinek, hogy változtasson színt!
8 Eszti.pensize(3)             # Mond meg Esztinek, hogy változtassa meg a
  →tolla vastagságát!
9
10 Eszti.forward(50)
11 Eszti.left(120)
12 Eszti.forward(50)
13
14 ablak.mainloop()
```

Amikor futtatjuk ezt a programot, ez az új ablak felugrik és a képernyőn marad, amíg be nem zárjuk.



Terjeszd ki ezt a programot...

1. Módosítsd a programot úgy, hogy mielőtt létrehozod az ablakot, kérje meg a felhasználót, hogy adja meg a kívánt háttérszínt! El kell tárolnod a felhasználó választ egy változóban és módosítani az ablak színét a felhasználó kívánsága szerint. (Segítség: találsz egy listát az engedélyezett színekről a <http://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm> címen. Ez tartalmaz pár elég szokatlan színt is, mint például a „HotPink”, azaz forró rózsaszín.)
2. Végezz el hasonló változtatásokat, hogy a felhasználó futásidőben meg tudja adni Eszti tollának a színét!
3. Csináld meg ugyanezt a toll vastagsággal! *Segítség:* a felhasználóval folytatott párbeszéd során egy sztringet fogsz visszakapni, de Eszti pensize metódusa egész típusú értéket vár paraméterként. Szóval neked kell a sztringet int típusúvá konvertálnod, mielőtt átadnád a pensize metódusnak.

3.2. Példányok – teknőcök hada

Mint ahogy sok különböző egész változónk is lehet egy programban, úgy sok teknőcünk is lehet egyszerre. Mindegyik egy ún. **példány**. Minden egyes példánynak saját tulajdonságai vannak és saját metódusai – így Sanyi rajzolhat egy vékony fekete tollal egy bizonyos pozícióban, míg Eszti haladhat a saját útján egy vastag rózsaszín tollal.

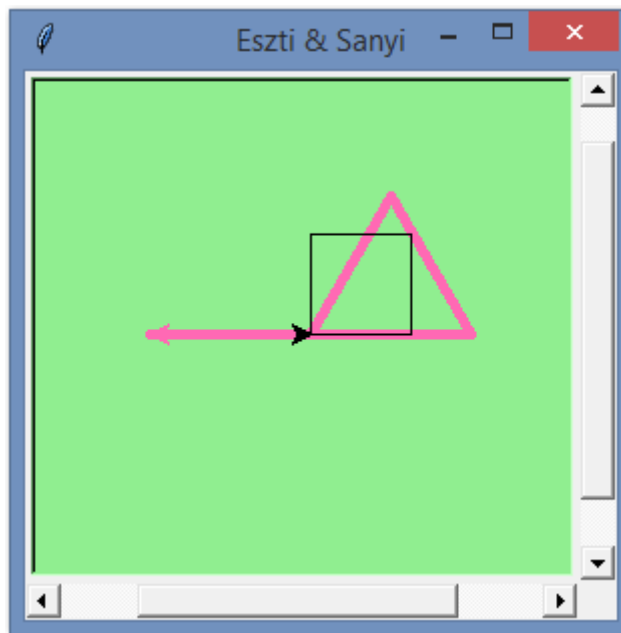
```
1 import turtle
2 ablak = turtle.Screen()           # Állítsd be az ablakot és tulajdonságait!
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3  ablak.bgcolor("lightgreen")
4  ablak.title("Eszti & Sanyi")
5
6  Eszti = turtle.Turtle()           # Hozd létre Eszti-t és add meg tulajdonságait!
7  Eszti.color("hotpink")
8  Eszti.pensize(5)
9
10 Sanyi = turtle.Turtle()           # Hozd létre Sanyit!
11
12 Eszti.forward(80)                  # Rajzoltass Eszti-vel egy egyenlő oldalú
   ↪háromszöget!
13 Eszti.left(120)
14 Eszti.forward(80)
15 Eszti.left(120)
16 Eszti.forward(80)
17 Eszti.left(120)                    # Fejezd be a háromszöget!
18
19 Eszti.right(180)                   # Eszti forduljon meg!
20 Eszti.forward(80)                  # Mozdítsd el őt a kiindulóponttól!
21
22 Sanyi.forward(50)                  # Rajzoltass Sanyival egy négyzetet!
23 Sanyi.left(90)
24 Sanyi.forward(50)
25 Sanyi.left(90)
26 Sanyi.forward(50)
27 Sanyi.left(90)
28 Sanyi.forward(50)
29 Sanyi.left(90)
30
31 ablak.mainloop()
```

Itt látható mi történik amikor Sanyi befejezi a négyzetet és Eszti is a háromszöget:



Néhány *Hogyan gondolkodhatsz úgy, mint egy informatikus* megállapítás:

- 360 fok van egy teljes körben. Ha összeadjuk egy teknőc összes fordulatát, *függetlenül attól, hogy a fordulatok*

között milyen lépések vannak, könnyen rájöhettünk arra, hogy az 360 fok többszöröse-e. Erről meggyőződhetünk úgy, hogy megnézzük pontosan ugyanabba az irányba néz-e a teknőc, mint amikor létrehoztuk. (Geometriai meggyőzés szerint a 0 fok keleti irányt jelent.)

- Kihagyhatjuk Sanyi utolsó fordulatát, de az nem lesz megfelelő számunkra. Ha arra kérnek meg, hogy rajzoljunk egy zárt alakzatot, mint a négyzet vagy a téglalap, az egy jó ötlet, ha befejezzük az összes fordulatot, hogy úgy hagyjuk ott a teknőcöt, ahogy találtuk, ugyanabba az irányba fordulva. Ennek akkor látjuk majd jelentőségét, ha egy terjedelmesebb kódrészletet készítünk egy nagyobb programban, ugyanis így könnyebb nekünk, embereknek.
- Ugyanezt tesszük Eszti esetén: aki háromszöget rajzolt és tett egy teljes 360 fokos fordulatot. Aztán elforgatjuk őt és félremozgatjuk. Még az üres 18. sor is arra céloz, hogyan működik a programozó *mentális blokkosítása*: Eszti mozdulatai egy „rajzolj háromszöget” blokkot (12-17 sorok) és aztán egy „mozdulj el a kiindulópontból” blokkot (19-20 sorok) tartalmaznak.
- Az egyik kulcsfontosságú felhasználása a megjegyzéseknek, ha lejegyezzük vele mentális blokkjainkat és nagy ötleteinket. Ezek nem mindig jelennek meg a kódban explicit módon.
- És végül, két teknőc még nem sereg. Azonban a lényeges gondolat az, hogy a turtle modul ad nekünk egyfajta gyárat, hogy annyi teknőcök készíthessünk, amennyit csak akarunk. Mindegyiknek saját állapota és viselkedése lesz.

3.3. A for ciklus

Amikor a négyzetet rajzoltuk, az elég fárasztó volt. Pontosán négyszer kellett megismételni a haladás-fordulás lépését. Ha hatszöget vagy nyolcszöget vagy egy 42 oldalú poligont rajzolunk, a helyzet még rosszabb lesz.

Emiatt minden program egyik építőeleme az kell legyen, hogy egy kódrészletet ismételni tudjunk újra és újra.

A Python **for** ciklusa megoldja ezt a problémát. Mondjuk azt, hogy van pár barátunk és mindegyikjüknek küldeni akarunk egy e-mailt, hogy meghívjuk őket a bulinkba. Még nem tudjuk, hogyan kell emailt küldeni, ezért egyelőre írunk ki egy üzenetet minden egyes barátunk:

```
1 for b in ["Misi", "Petra", "Botond", "Jani", "Csilla", "Peti", "Norbi"]:  
2     meghivas = "Szia, " + b + "! Kérlek gyere el a bulimba szombaton!"  
3     print(meghivas)  
4 # A többi utasítás ide kerülhet...
```

Amikor ezt futtatjuk, a kimenet így néz ki:

```
Szia, Misi! Kérlek gyere el a bulimba szombaton!  
Szia, Petra! Kérlek gyere el a bulimba szombaton!  
Szia, Botond! Kérlek gyere el a bulimba szombaton!  
Szia, Jani! Kérlek gyere el a bulimba szombaton!  
Szia, Csilla! Kérlek gyere el a bulimba szombaton!  
Szia, Peti! Kérlek gyere el a bulimba szombaton!  
Szia, Norbi! Kérlek gyere el a bulimba szombaton!
```

- A **b** változó az első sor **for** utasításában **ciklusváltozó** névre hallgat. Persze ehelyett más nevet is választhatasz.
- A 2. és 3. sor a **ciklus törzse**. A törzs sorai mindig bevannak húzva. Az indentálás határozza meg pontosan, hogy melyik utasítás van a ciklus törzsében.
- A ciklus minden egyes *ismétlésénél* először egy ellenőrzés hajtódik végre, hogy vár-e további elem feldolgozásra. Ha nem maradt több (ez a **befejezési feltétel**), akkor az ismétlés véget ér. A program végrehajtás a ciklus törzse utáni következő utasítással folytatódik (azaz esetünkben a következő utasításra a 4. sorban lévő megjegyzés alatt).

- Ha vannak további feldolgozásra váró elemek, akkor a ciklusváltozó frissül és a lista következő elemére hivatkozik. Ez azt jelenti, hogy ebben az esetben a ciklus magja 7-szer lesz végrehajtva és minden alkalommal `b` különböző barátra hivatkozik.
- A ciklusmag minden végrehajtása után a Python visszatér a `for` utasításra, hogy lássa van-e még kezelendő elem, és a következő elemet hozzárendeli az `b` változóhoz.

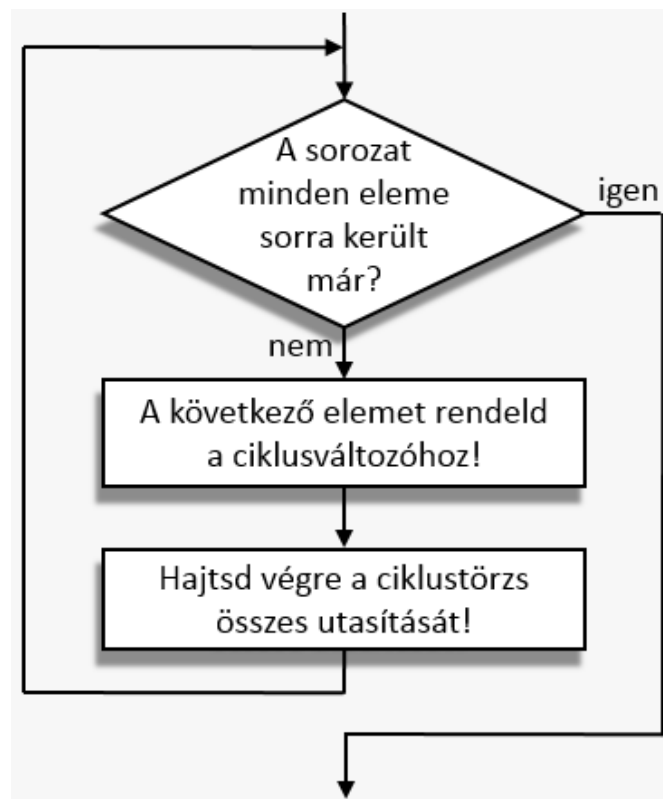
3.4. A for ciklus végrehajtási sorrendje

Egy program futása során a parancsértelmező nyomon követi, hogy melyik utasítás hajtódik éppen végre. Ezt hívhatjuk **programvezérlésnek** vagy az utasítások **végrehajtási sorrendjének**. Amikor az ember hajtja végre a programot, akkor az ujjával mutatja, melyik utasítás következik. Szóval úgy gondolhatunk a programvezérlésre, mint a „Python ujjának mozgása”.

A programvezérlés eddig mindig szigorúan fentről lefelé haladó volt, egy adott időpillanatban egy utasítás volt soron. A `for` ciklus megváltoztatja ezt.

A for ciklus folyamatábrája

A vezérlés menetét könnyű megjeleníteni és megérteni, ha rajzolunk egy folyamatábrát. Ez megmutatja a pontos lépéseket és azt a logikát, ahogyan a `for` utasítás végrehajtódik.



3.5. A ciklus egyszerűsíti a teknőc programunkat

Ahhoz, hogy egy négyzetet rajzoljunk, négyszer meg kell ismételni ugyanazt – mozgasd a teknőcöt és fordulj. Korábban 8 sornyi programrészletet használtunk, hogy Sanyival megrajzoltassuk a négyzet négy oldalát. Pontosan ugyanezt három sor használatával is elérhetjük:

```
1 for i in [0,1,2,3]:
2     Sanyi.forward(50)
3     Sanyi.left(90)
```

Néhány megjegyzés:

- „Néhány sornyi kód megspórolása” kényelmes lehet, de nem ez a nagydolog itt. Ami sokkal fontosabb az az, hogy találtunk „ismétlődő utasításmintákat” és újraszerveztük a programunkat, úgy hogy ismétlje a mintát. Megtalálni egy nagyobb egységet és akörül valahogy rendezettebbé tenni a programot, az nélkülözhetetlen képesség a számítógépes gondolkodásban.
- Az [0,1,2,3] értékeket arra használtuk, hogy a ciklus törzsét 4 alkalommal végrehajtsuk. Használhattuk volna bármelyik négy értéket, de konvencionálisan ezeket szoktuk. Valójában ezek olyan népszerűek, hogy a Python ad nekünk egy speciális beépített `range` (tartomány) objektumot:

```
1 for i in range(4):
2     # Hajtsd végre a törzset az i = 0, majd 1, aztán 2, végül 3 értékkel!
3 for x in range(10):
4     # Ez x értékét sorban beállítja a [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]-ra
    ↪ sorozat minden egyes elemére
```

- Az informatikusok 0-tól számolnak!
- A `range` előállíthatja az értékek egy sorozatát a `for` ciklus ciklusváltozója számára. Ezek 0-val kezdődnek és a fenti esetekben nem tartalmazzák a 4-et vagy a 10-et.
- A korábbi kis trükkünk, hogy biztosak legyünk abban, hogy Sanyi végül megtette a teljes 360 fokok fordulatot, kifizetődött: ha nem tettük volna meg ezt, akkor nem lettünk volna képesek használni a ciklust a négyzet négy oldalának megrajzolásához. Lett volna egy „speciális eset”, amely különbözött volna a többi oldaltól. Amikor csak lehetséges, sokkal jobban szeretünk általános mintához illeszkedő kódot készíteni, mint speciális helyzeteket kezelni.

Szóval ha négyszer kell megismételni valamit egy jó Python programozó így teszi:

```
1 for i in range(4):
2     Sanyi.forward(50)
3     Sanyi.left(90)
```

Mostanra bizonyára látod, hogyan kell megváltoztatni korábbi programunkat, hogy Eszti is használhasson `for` ciklust az egyenlő oldalú háromszöge megrajzolásához.

Mi történik azonban, ha az alábbi változtatást hajtuk végre?

```
1 for sz in ["yellow", "red", "purple", "blue"]:
2     Sanyi.color(sz)
3     Sanyi.forward(50)
4     Sanyi.left(90)
```

A változó szintén megkapja a lista értékeit. Szóval a listák sokkal általánosabb helyzetekben is használhatóak, nem csak a `for` ciklusban. A fenti kód átírható így is:

```
1 # Változónak listát adunk értékül
2 szinek = ["yellow", "red", "purple", "blue"]
3 for sz in szinek:
4     Sanyi.color(sz)
5     Sanyi.forward(50)
6     Sanyi.left(90)
```

3.6. További teknőc metódusok és trükkök

A teknőc metódusokban negatív szögeket és távolságokat is használhatunk. Így az `Eszti.forward(-100)` utasítás hatására `Eszti` hátrafelé fog mozogni és az `Eszti.left(90)` esetén jobbra fog fordulni. Továbbá mivel egy teljes kör 360 fok, a 30 fokkal balra fordulás esetén a teknőc ugyanabba az irányba néz, mintha 330 fokot jobbra fordult volna. (A képernyőn az animáció különbözni fog – mondhatod `Esztinek`, hogy az óra járásával megegyezően vagy ellentétesen forduljon.)

Ez azt sugallja, hogy nincs szükség jobbra és balra forduló metódusnak – lehetünk minimalisták, csak egy metódust használva. Van egy `backward` (hátrafelé) metódus is. (Ha elég kocka vagy, örömet leled abban, hogy azt mond `Sanyi.backward(-100)`, amikor azt akard, hogy előre felé haladjon.)

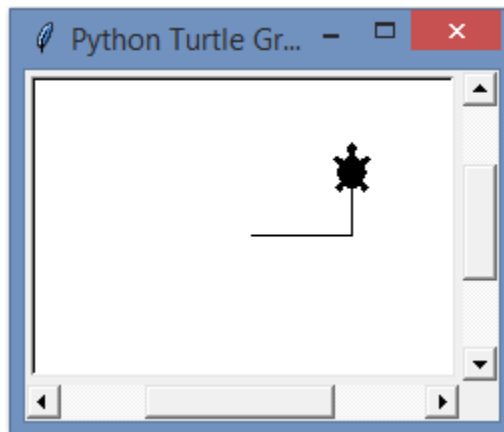
A *tudósként gondolkodás* része jobban megérteni a szerkezeteket és gazdagítani a tudományterületed kapcsolatait. Tehát néhány geometriai vagy számtani alap tény átgondolásához, a bal-jobb, az előre-hátra, a pozitív-negatív értékű távolságok és szögek közötti kapcsolat megértéséhez egy jó kiindulópont, ha játszol kicsit a teknőcökkel.

A teknőc tolla lehet felemelt vagy lehelyezett pozícióban. Ez lehetővé teszi számodra, hogy egy másik helyre mozgassd a teknőcot anélkül, hogy vonalat húznál oda. A metódusok a következők:

```
1 Sanyi.penup()
2 Sanyi.forward(100)      # Ez mozgatja Sanyit, de nem húz vonalat
3 Sanyi.pendown()
```

Minden teknőcnek saját alakja lehet. Néhány példa: `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
1 Sanyi.shape("turtle")
```



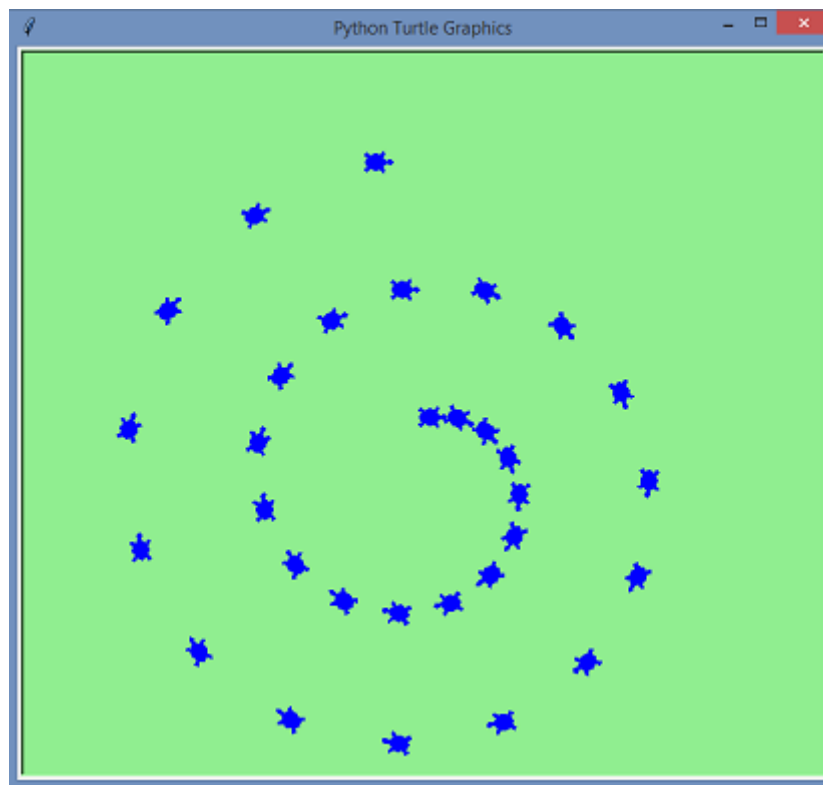
Felgyorsíthatjuk vagy lelassíthatjuk a teknőc animációját. (Az animáció vezérli, hogy milyen gyorsan fordul vagy halad a teknőc.) A sebességet 1 (leglassabb) és 10 (leggyorsabb) között állíthatjuk. Ha azonban 0-t állítunk be, annak speciális jelentése van – kapcsold ki az animációt, mozogj a lehető leggyorsabban.

```
1 Sanyi.speed(10)
```

A teknőc a vászonra „pecsételheti” a lábnyomát és az megmarad akkor is, ha a teknőc máshova megy. A pecsételés akkor is működik, ha a toll fel van emelve.

Csináljunk egy példát, hogy kicsit felvágjunk ezekkel az új tulajdonságokkal:

```
1 import turtle
2 ablak = turtle.Screen()
3 ablak.bgcolor("lightgreen")
4 Eszti = turtle.Turtle()
5 Eszti.shape("turtle")
6 Eszti.color("blue")
7
8 Eszti.penup()                # Ez új
9 meret = 20
10 for i in range(30):
11     Eszti.stamp()           # Hagyj egy lenyomatot a vásznon!
12     meret = meret + 3       # Növekd a méretet minden ismétlésnél!
13     Eszti.forward(meret)    # Mozgasd ...
14     Eszti.right(24)         # ... és fordítsd Esztit!
15
16 ablak.mainloop()
```



Légy óvatos! Hányszor lesz a törzs végrehajtva? Hány teknőc alakot látsz a képernyőn? Egy kivételével az összes alakzat a képernyőn lábnyom, amelyet a `stamp` hoz létre. Azonban a programban továbbra is csak *egy* teknőcpéldány van – kitaláld melyik az igazi *Eszti*? (Segítség: ha nem vagy biztos, írd egy új sort a kódhoz a `for` ciklus után, ami megváltoztatja *Eszti* színét, vagy tedd le a tollát és húzz egy vonalat, vagy esetleg változtasd meg az alakját, stb.)

3.7. Szójegyzék

attribútum (attribute) Néhány állapot vagy érték, amely egy bizonyos objektumhoz tartozik. Például: `Eszti.színe`.

befejezési feltétel (terminating condition) Egy feltétel, ami azt eredményezi, hogy a ciklus befejezi a törzsének ismétlését. A `for` ciklusokban, ahogy ebben a fejezetben láttuk, azt jelenti, hogy nincs több elem, amit a ciklusváltozóhoz rendeljünk.

ciklus törzs (loop body) Akárhány utasítást elhelyezhetünk a cikluson belül. Ezt azzal a ténnyel jelöljük, hogy a `for` ciklus alatti utasítások be vannak húzva (indentálva).

ciklusváltozó (loop variable) Egy a `for` ciklus részeként használt változó. Minden ismétlés során másik értéket kap.

for ciklus (for loop) Egy utasítás Pythonban, annak érdekében, hogy kényelmesen ismételhessünk utasításokat a ciklus *törzsében*.

hívás (invoke) Egy objektumnak vannak metódusai. Amikor a hívás igét használjuk, akkor a *metódus aktiválását* értjük alatta. A metódushívás során annak neve után kerek zárójeleket rakunk, néhány paraméterrel. Így a `tess.forward(20)` nem más, mint a `forward` metódus hívása.

metódus (method) Egy objektumhoz kötődő funkció. A metódus hívása vagy aktiválása az objektum valamilyen reakcióját, választát eredményezi, pl. amikor azt mondjuk `tess.forward(100)`, akkor a `forward` az egy metódus.

modul (module) Egy fájl, amely Python nyelvű definíciókat és utasításokat tartalmaz azért, hogy egy másik Python programban használjuk. A modul tartalma az `import` utasítással tehető elérhetővé egy másik programból.

objektum (object) Egy „dolog”, amire egy változó hivatkozhat. Ez lehet egy képernyő ablak vagy egy az általunk létrehozott teknőcök közül.

példány (instance) Egy bizonyos típus vagy osztály objektuma. `Eszti` és `Sanyi` a `Turtle` osztály eltérő példányai.

programvezérlés (control flow) Lásd a *végrehajtási sorrendet* a következő fejezetben!

tartomány (range) Egy beépített függvény Pythonban egész számok sorozatának generálására. Különösen fontos, ha írunk kell egy `for` ciklust, ami meghatározott számú ismétlést hajt végre.

vászon (canvas) A felület az ablakon belül ahol a rajzolás történik.

3.8. Feladatok

- Írj egy programot, amely 1000-szer kiírja a `Szeretjük a Python teknőcöket!` mondatot!
- Add meg három attribútumát a mobiltelefon objektumodnak! Add meg a mobilod három metódusát!
- Írj egy programot, ami a `for` ciklus használatával az alábbi szöveget írja ki**

```
Az év egyik hónapja január.  
Az év egyik hónapja február.  
...
```
- Tételezzük fel, hogy a teknőcünk `Eszti` a 0 irányban áll – kelet felé néz. Végrehajtjuk az `Eszti.left(3645)` utasítást. Mit csinál `Eszti` és merre néz?
- Tételezzük fel, hogy van egy `xs = [12, 10, 32, 3, 66, 17, 42, 99, 20]` értékadásunk.
 - Írj egy ciklust, amely mindegyik számot kiírja egy új sorba!
 - Írj egy ciklust, amely mindegyik számot és azok négyzetét is kiírja egy új sorba!

- (c) Írj egy ciklust, amely összeadja a listában szereplő összes számot egy *összeg* változóba! Az *összeg* változónak 0 értéket kell adnod az összegzés előtt, majd a ciklus befejezése után az *összeg* változó értékét írasd ki!
- (d) Írasd ki a listában szereplő összes szám szorzatát!
6. Használd a `for` ciklust, hogy egy teknőccel kirajzoltsd ezeket a szabályos sokszögeket (a szabályos azt jelenti, hogy minden oldala egyforma hosszú és minden szöge azonos):
- Egyenlő oldalú háromszög
 - Négyzet
 - Hexagon (hatszög)
 - Oktagon (nyolcszög)
7. Egy részeg kalóz véletlenszerűen fordul egyet majd megy 100 lépést, tesz még egy véletlen fordulatot és még 100 lépést, és így tovább. Egy bölcsész hallgató feljegyzi az összes fordulat szögét mielőtt a kalóz megtenné a következő 100 lépést. Az ő kísérletének adatai `[160, -43, 270, -97, -43, 200, -940, 17, -86]`. (A pozitív szögek az óra járásával ellentétes irányúak.) Használj egy teknőcöt, hogy kirajzold részeg barátunk útvonalt!
8. Fejleszd a programod, hogy a végén azt is megmondja, milyen irányba néz a pityókás kalóz a botorkálása végén! (Tételezzük fel, hogy a 0 irányból indul.)
9. Ha egy szabályos 18 oldalú sokszöget szeretnél rajzolni, hány fokkal kellene elfordulnia a teknőcnek minden csúcsnál?
10. Jóssold meg, mit csinál minden egyes sor az alábbi programban, aztán figyeld meg, mi történik! Értékelj magad, egy pontot adva minden pontos jóslatért!

```
>>> import turtle
>>> ablak = turtle.Screen()
>>> Eszti = turtle.Turtle()
>>> Eszti.right(90)
>>> Eszti.left(3600)
>>> Eszti.right(-90)
>>> Eszti.speed(10)
>>> Eszti.left(3600)
>>> Eszti.speed(0)
>>> Eszti.left(3645)
>>> Eszti.forward(-100)
```

11. Írj egy programot, amely egy olyan alakzatot rajzol, mint ez:

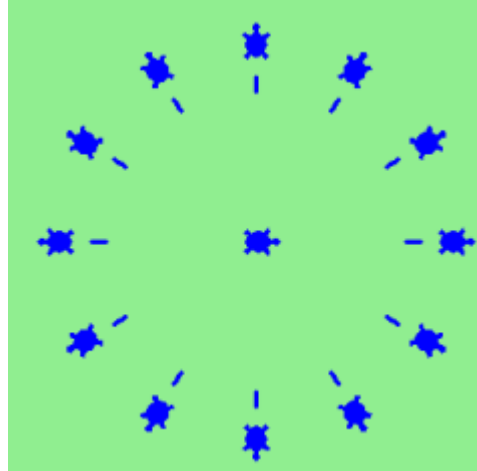


Segítség:

- Próbáld ki egy papírlapon, mozgatva a mobilodat, mint egy teknőst! Figyeld meg hány teljes forgást tesz a mobilod mielőtt befejezi a csillagot! Mivel minden teljes fordulat 360 fokkal, ki tudod találni hány fokot fordul a telefonod. Ha ezt 5-tel osztod, mivel a csillagnak 5 csúcsa van, megtudod a teknős hány fokot fordul egy csúcson.

- El tudod rejteni a teknőcöt a láthatatlanná tévő köpenyével, ha nem akarsz őt láttatni. Ő továbbra is rajzolja a vonalakat, ha a tolla lent van. Így hívható a szükséges módszer: `Eszti.hideturtle()`. Ha újra akarsz látni a teknőcöt, használd az `Eszti.showturtle()` utasítást!

12. Írj egy programot, ami rajzol egy olyan óralapot, mint ez:



13. Hozz létre egy teknőcöt és add értékül egy változónak! Amikor megkérdezed a típusát, mit kapsz?
14. Mi a teknőcök gyűjtőneve? (Segítség: ők nem alkotnak *hadat*.)
15. Mi a pitonok gyűjtőneve? Egy piton az egy vipera? A piton mérges kígyó?

4. fejezet

Függvények

4.1. Függvények

A **függvény** Pythonban nem más, mint összetartozó utasítások névvel ellátott sorozata. A függvényeknek fontos szerepük van a program rendezetté tételében, hiszen a segítségükkel olyan blokkokra bonthatjuk a program szövegét, melyek illeszkednek a probléma megoldása során követett gondolatmenetünkhöz.

A **függvény definíció** szintaktikája:

```
def NÉV( PARAMÉTEREK ) :  
    UTASÍTÁSOK
```

A függvények nevét szabadon választhatjuk, azonban a választott névnek eleget kell tennie az azonosítóra vonatkozó szabályoknak, és nem lehet Python kulcsszó.

A függvények belsejében egy vagy több utasítás is állhat, a `def` kulcsszóhoz képest jobbra igazítva. A könyv példái-
ban mindig 4 szóközzel állnak bentebb az utasítások, mint a kulcsszó első betűje. A függvény definíció a második a
számos **összetett utasítás** közül, amit látni fogunk.

Az összetett utasítások mindegyike hasonló mintát követ. Részei:

1. Egy **fejléc**, mely kulcsszóval kezdődik és kettősponttal záródik.
2. Egy **törzs**, mely egy vagy több Python utasítást tartalmaz. Az utasítások a fejléctől nézve azonos mértékű – a *Python kódolási szabványa* szerint 4 szóközny – behúzással állnak.

A `for` ciklussal, melynek szintaktikája megfelel a fenti leírásnak, már találkoztunk.

Na de térjünk vissza a függvény definícióhoz. A fejlécben álló kulcsszó a `def`, melyet a függvény neve és egy kerek zárójelek között álló *formális paraméterlista* követ. A paraméterlista lehet üres is, de akár több paraméter is állhat ott, egymástól vesszővel elválasztva. A zárójeleket minden esetben kötelező kitenni. Az esetleges paraméterek azt határozzák meg, hogy milyen információ megadása szükséges a függvény végrehajtásához.

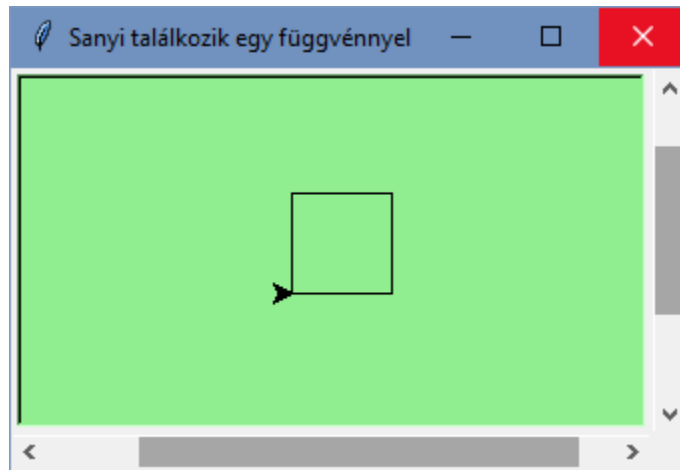
Tegyük fel, hogy a teknőcökkel dolgozva gyakran van szükségünk négyzetek rajzolására. A „négyzetek rajzolása” számos kisebb lépést tartalmazó *absztrakció*, egy részprobléma. Írjunk is egy függvényt, melyet „építőköckaként” is használhatunk majd a jövőben:

```
1 import turtle  
2  
3 def negyzet_rajzolas(t, h):  
4     """Egy h oldalhosszúságú négyzet rajzoltatása a t teknőccel"""  
5     for i in range(4):
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6         t.forward(h)
7         t.left(90)
8
9     # Egy ablak létrehozása és néhány tulajdonságának beállítása
10    a = turtle.Screen()
11    a.bgcolor("lightgreen")
12    a.title("Sanyi találkozik egy függvénynel")
13
14    Sanyi = turtle.Turtle()          # Sanyi létrehozása
15    negyzet_rajzolas(Sanyi, 50)      # Egy négyzet rajzolása a függvény_
    ↪meghívásával
16    a.mainloop()
```



A függvény neve `negyzet_rajzolas`. Két paramétere van: az első mondja meg a függvénynek, hogy melyik teknőcöt kell mozgatni, a második paraméter pedig megadja a rajzolandó négyzet oldalhosszúságát. Az utasítások indentálása határozza meg, hogy hol ér véget a függvény, az üres sorok nem játszanak szerepet.

Dokumentációs sztringek a dokumentációhoz

Közvetlenül a függvények fejléce alatt álló szövegeket a Python (és néhány más környezet is) **dokumentációs sztring-nek** tekinti, és a szokásostól eltérően kezeli. A PyCharm például egy felugró ablakban jeleníti meg a függvényhez tartozó sztringet, ha a függvény nevére állva lenyomjuk a „Ctrl+Q” billentyűkombinációt.

Ezek a sztringek kulcsszerepet töltenek be abban, hogy a Python nyelven írt függvényeinket megfelelő leírással láthassuk el, ami igen fontos része a programozásnak. Képzeljük csak el, hogy valaki fel szeretné használni az általunk írt egyik függvényt. Nem várhatjuk el tőle, hogy tisztában legyen a függvényünk működésével, belső felépítésével. Egy függvény meghívásához elegendőnek kellene lennie, ha ismeri a függvényünk által várt paramétereket, és a várható kimenetet. Ez a megállapítás vissza is juttat bennünket az absztrakció fogalmához, amelyről a későbbiekben még lesz szó.

A dokumentációs sztringeket három idézőjel között szokás megadni, mert ez a fajta jelölés teszi lehetővé, hogy később egyszerűen bővíthessük a leírást, akár több sort is írva.

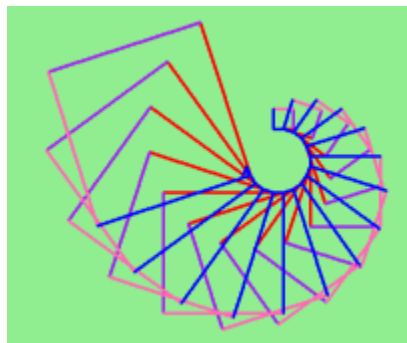
A megjegyzéseket és a dokumentációs sztringeket az is megkülönbözteti, hogy míg az előbbieket az elemző teljesen eltávolítja, utóbbiak a *futási idő* alatt is elérhetők a Python eszközök számára.

A függvények létrehozása nem jár együtt a függvény végrehajtásával, hiszen a függvények csak akkor kezdik meg a működésüket, ha meghívják őket. Korábban már láttunk néhány példát **függvényhívásra**, használtuk már a **print**, **range** és **int** beépített függvényeket is. A hívás a végrehajtandó függvény nevének és egy értéklistának a megadásával

tehető meg. A listában szereplő értékek, melyeket *argumentumoknak* vagy *aktuális paramétereknek* nevezzünk, a függvény definíciónál megadott (*formális*) *paraméterekhez* lesznek rendelve. A fenti program utolsó előtti sorában például a `negyzet_rajzolas` nevű függvényt hívtuk meg, és átadtuk `Sanyi`-t mint irányítandó teknőcöt, és egy 50-es értéket mint a rajzolandó négyzet oldalhosszúságát. Amikor a függvény végrehajtásra kerül, akkor a `h` változó 50-es értéket vesz fel, a `t` változó pedig ugyanarra a teknőc példányra hivatkozik majd, mint a `Sanyi` nevű változó.

A létrehozott függvények akárhányszor felhasználhatók, és minden egyes hívásnál lefutnak a függvényben szereplő utasítások. Ennélfogva bármelyik teknőcöt rávehetjük egy négyzet rajzolására. A következő példában egy kicsit változtatunk a `negyzet_rajzolas` függvényen, majd `Eszti`-vel rajzoltatunk 15 különböző négyzetet.

```
1 import turtle
2
3 def tobbszinu_negyzet_rajzolas(t, h):
4     """Egy h oldalhosszúságú, többszínű négyzet rajzoltatása a t teknőccel"""
5     ↪ "
6     for i in ["red", "purple", "hotpink", "blue"]:
7         t.color(i)
8         t.forward(h)
9         t.left(90)
10
11 # Egy ablak létrehozása és a tulajdonságainak beállítása
12 a = turtle.Screen()
13 a.bgcolor("lightgreen")
14
15 # Eszti létrehozása és tulajdonságainak beállítása
16 Eszti = turtle.Turtle()
17 Eszti.pensize(3)
18
19 meret = 20 # A legkisebb négyzet mérete
20 for i in range(15):
21     tobbszinu_negyzet_rajzolas(Eszti, meret)
22     meret = meret + 10 # Növeljük a következő négyzet méretét
23     Eszti.forward(10) # Kicsit arrébb léptetjük a teknőcöt
24     Eszti.right(18) # és kicsit elfordítjuk
25
26 a.mainloop()
```



4.2. A függvények is hívhatnak függvényeket

Tegyük fel, hogy egy olyan függvényt szeretnénk készíteni, amely egy téglalapot rajzol, méghozzá olyan szélességgel és magassággal, melyet a hívásnál argumentumként megadunk. Míg a négyzet rajzolásánál ugyanazt a tevékenységet ismételhettük négyszer, most ezt nem tehetjük, hiszen az oldalak hossza eltérhet egymástól.

Végül előállunk egy ilyen szépséggel:

```
1 def teglalap_rajzolas(t, sz, m):
2     """Egy sz szélességű, m magasságú téglalap rajzoltatása a t teknőccel. """
3     for i in range(2):
4         t.forward(sz)
5         t.left(90)
6         t.forward(m)
7         t.left(90)
```

A paraméternevek szándékosan ilyen rövidek. Később, amikor már kellő tapasztalat birtokában igazi programokat készítünk, ragaszkodni fogunk az értelmesebb nevekhez. Egyelőre igyekezzünk eloszlatni azt a képzetet, hogy a program értene a nevekből. A programnak nincs tudomása arról, mit szeretnénk rajzolni és a paraméterek jelentésével sincs tisztában. Az olyan fogalmak, mint a téglalap, a szélesség vagy a magasság csak az emberek számára bírnak értelemmel, a programok és gépek számára nem.

A tudományos gondolkodásmód meghatározó eleme a minták, összefüggések keresése. Bizonyos szinten a fenti kód is ezt mutatja, hiszen ahelyett, hogy oldalanként rajzoltuk volna meg a téglalapot, egy fél téglalapot rajzoltunk és megismételtük egy ciklus segítségével.

Ha már itt tartunk, az is eszünkbe juthat, hogy a négyzet a téglalap speciális esete, így akár a téglalaprajzoló függvényt is fel lehetne használni egy négyzetrajzoló függvény elkészítéséhez.

```
1 def negyzet_rajzolas(tk, h): # A négyzetrajzoló függvény új változata
2     teglalap_rajzolas(tk, h, h)
```

Van itt néhány említésre méltó pont:

- Egy függvény meghívhat egy másik függvényt.
- A `negyzet_rajzolas` függvény új változata már mutatja a négyzet és a téglalap közti összefüggést.
- Ha `negyzet_rajzolas(Eszti, 50)` formában meghívjuk a függvényt, akkor az `Eszti` objektum a `tk`, az `50`-es értéket a `h` paraméterhez kerül át.
- A függvény belsejében a paraméterek olyanok, mint a változók.
- Mire a `teglalap_rajzolas`-t meghívja a `negyzet_rajzolas` függvény, addigra a `tk` és `h` paraméterek már megkapták az értéküket. Az előbb látott hívás esetében a `teglalap_rajzolas` függvény `t` paramétere `Eszti`-t, `sz` és `m` paramétere egyaránt `50`-es értéket kap.

Az eddigiekből még nem feltétlenül látszik, hogy a függvények készítésével miért érdemes vesződni. Valójában rengeteg oka van ennek, nézzünk meg most kettőt:

1. Függvényeket írva névvel ellátott utasításcsoportokat hozhatunk létre, ennél fogva az összetett számításokat egyetlen, egyszerű utasítás mögé rejthetjük el. A függvények (a nevükkel együtt) képesek leírni egy feladat megoldása során meghatározott részproblémákat.
2. Rövidebbé tehetik a programjainkat, ha az ismétlődő programrészeket egy-egy függvénybe emeljük ki.

Amint az várható, egy függvényt csak akkor lehet végrehajtani, ha már létezik. Másként fogalmazva, a függvény definiálásának még a hívás előtt végbe kell mennie.

4.3. A programvezérlés

Csak akkor biztosíthatjuk, hogy egy függvény még a felhasználása előtt létrejöjjön, ha tisztában vagyunk az utasítások végrehajtási sorrendjével, vagyis a **programvezérlés menetével**. Az előző fejezetben érintőlegesen már volt róla szó.

A végrehajtás mindig a legelső utasítástól indul, majd fentről lefelé haladva, egyesével hajtódnak végre az utasítások.

A függvény definíció nem változtatja meg a végrehajtási sorrendet, de ugye emlékszünk még rá, hogy a benne szereplő utasítások csak a függvény meghívása esetén hajtódnak végre. Habár nem túl gyakori, akár a függvényen belül is létrehozhatunk függvényeket. Az ilyen függvények csak akkor futhatnak le, ha a külső függvény meghívásra került.

A függvényhívás lényegében egy kitérőt jelent a végrehajtási folyamatban. Ahelyett, hogy a következő utasításra kerülne a vezérlés, átkerül a meghívott függvény első sorára. Lefutnak a függvényen belül álló utasítások, majd visszakerül a vezérlés a hívás helyére és onnan megy tovább.

Elég egyszerűnek hangzik, egészen addig, ameddig eszünkbe nem jut, hogy a függvények is hívhatnak függvényeket. Egy függvény végrehajtása közben a programnak gyakran végre kell hajtania egy másik függvényhez tartozó utasításokat is. Ráadásul a másik függvény futtatása közben is szükség lehet egy újabb függvény végrehajtására!

A Python szerencsére mindig „észben tartja”, hogy hol jár. Valahányszor befejeződik egy függvény, a program vezérlése mindig pontosan oda tér vissza, ahonnan a függvényt meghívták. Ha pedig eléri a program végét, akkor a program is befejezi működését.

Mi a tanulság ebben a rusnya történetben? A programokat ne fentről lefele olvassuk, hanem a végrehajtás folyamatának megfelelően!

Kövessük élőben az utasítások végrehajtást

A PyCharm nyomkövető eszközével módunkban áll lépésről lépésre követni a program végrehajtásának menetét. A fejlesztőkörnyezet mindig kiemeli a soron következő utasítást, és a változók aktuális értékét is mutatja.

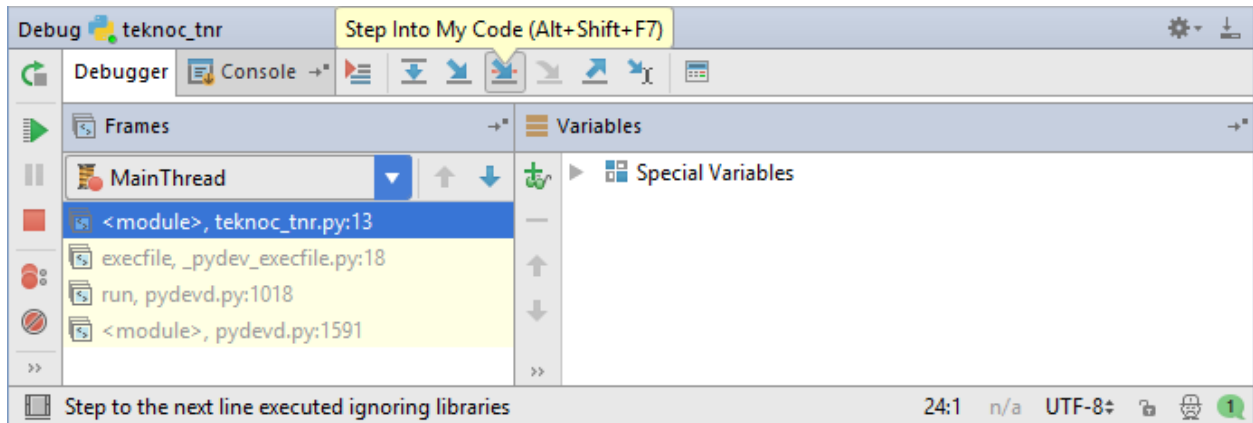
A nyomkövetésre szolgáló eszközök hatalmas segítséget jelenthetnek abban, hogy alaposan megértsük az egyes lépések során lezajló folyamatokat, ezért érdemes megtanulni a használatukat. Az utasításonkénti végrehajtás közben tesztelheted is magad, az alábbi kérdésekre keresve a választ:

1. Hogyan változnak majd a változók a következő sor végrehajtása során?
2. Hova kerül át a vezérlés, vagyis melyik utasítással folytatódik a program végrehajtása?

Nézzük is meg, hogyan működik a nyomkövetés a korábbi, 15 színes négyzetet rajzoló programunkat használva. Állj rá a szkript azon sorára, ahol a teknőcöt létrehoztuk, és nyomd le a *Ctrl+F8* billentyűkombinációt, vagy egyszerűen csak kattints az egérrel a sor elején álló szám mellé. Egy *töréspontot* hoztunk létre ezzel, melyet a PyCharm a sor előtt álló piros körrel jelöl.

```
1  import turtle
2
3
4  def tobbszinu_negyzet_rajzolas(t, h):
5      """Egy h oldalhosszúságú, többszínű négyzet rajzoltatása a t teknőccel"""
6      for i in ["red", "purple", "hotpink", "blue"]:
7          t.color(i)
8          t.forward(h)
9          t.left(90)
10
11
12  # Egy ablak létrehozása és a tulajdonságainak beállítása
13  a = turtle.Screen()
14  a.bgcolor("lightgreen")
15
```

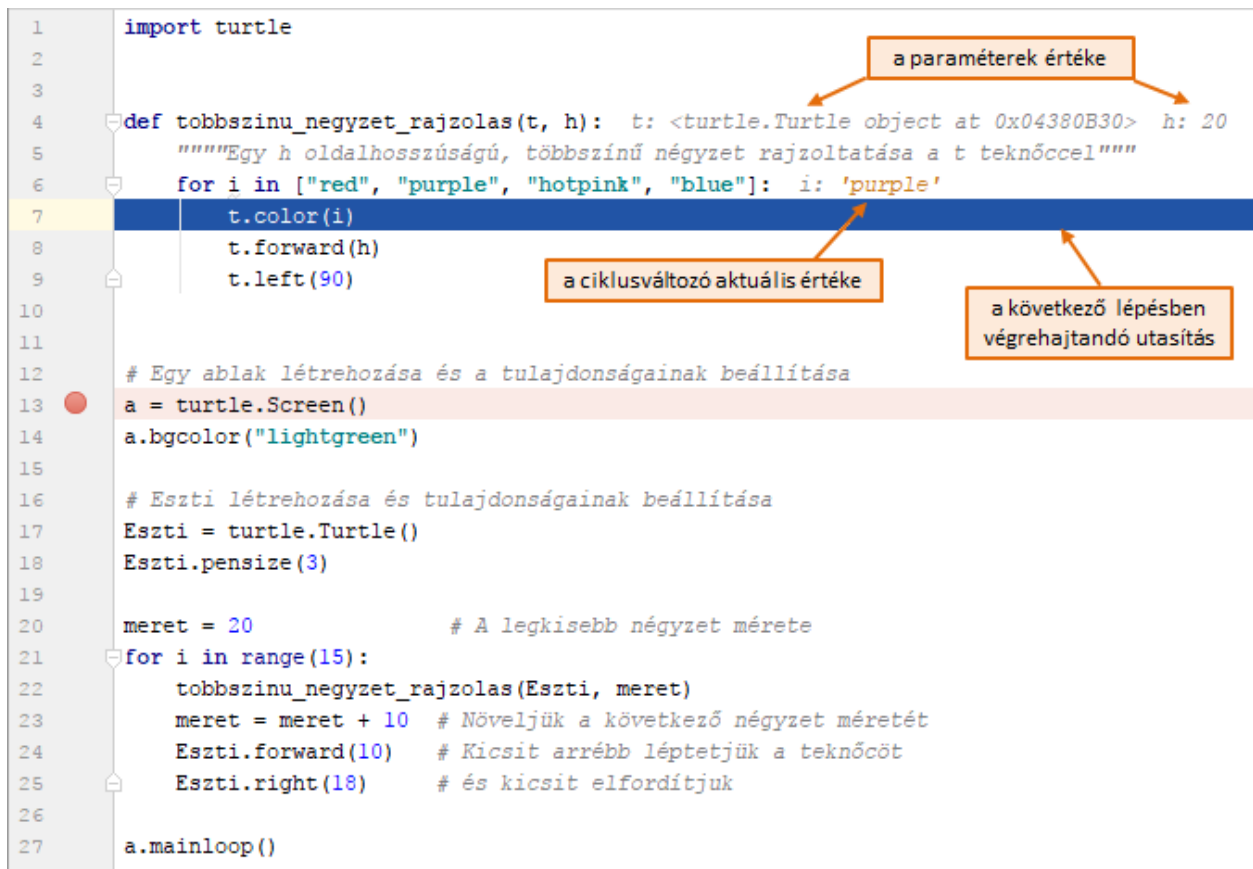
Ha készen vagy, akkor a szokásos Run helyett nyomd le a *Shift+F9*-et, vagy használd a bogár ikont. Elindul a program végrehajtása, de megszakad, amint a törésponthoz ér (a töréspont sorát már nem hajtja végre). A program indulásával párhuzamosan egy, az alábbihoz hasonló ablak is megjelenik:



Az ablak *Variables* részén belül a már létrehozott változók állapotát láthatjuk majd. A változók egy idő után eltűnnek, cserélődnek, újra megjelennek, később már tudni fogod miért.

Innentől kezdve egyesével hajthatjuk végre az utasításokat, ha újra és újra a képen látható *Step into My Code*... ikonra kattintunk. Figyeld meg, hogy a 13. és 14. sorok hatására létrejön és zölddé válik az ablak, majd a 17. sor végrehajtása után már a teknőc is az ablakba kerül. Lépdelj tovább, és nézd meg, hogyan kerül a vezérlés a ciklusba, majd onnan tovább a függvénybe, ahol egy másik ciklusba jut. Tovább folytatva az utasítások végrehajtását, a ciklus törzsében szereplő utasítások ismétlődnek meg újra és újra.

Ha itt tartasz, valószínűleg már azt is észrevetted, hogy a PyCharm ideiglenes megjegyzésekkel egészíti ki a kódot, mely a *Variables* ablakhoz hasonlóan az egyes változók aktuális értékét mutatja.



Pár négyzet megrajzolása után már unalmassá válhat a lépésenkénti végrehajtás. Ha a *Step Over* ikon (F8) billen-

tyű) használatára váltasz, akkor „átlépheted” a függvényhívásokat. Ilyenkor is végrehajtásra kerül az összes utasítás, de nem áll meg minden egyes lépésnél. Minden alkalommal eldönthetjük, hogy látni akarjuk-e a részleteket, vagy megelégszünk egy „magasabb szintű” nézettel, egyetlen egységként hajtva végre a függvényt.

Létezik néhány más lehetőség is, beleértve a program azonnali, vagyis a további utasítások végrehajtását mellőző *újrakezdést* is. A PyCharm *Run* menüjéből érhetők el.

4.4. Paraméterekkel rendelkező függvények

A legtöbb függvény rendelkezik paraméterekkel, ugyanis a paraméterek általánosabban felhasználhatóvá teszik a függvényeket. Például, ha egy szám abszolút értékét kívánjuk meghatározni, akkor meg kell adni, melyik számról van szó. Az abszolút érték számításához Pythonban egy beépített függvény áll rendelkezésünkre:

```
>>> abs(5)
5
>>> abs(-5)
5
```

A kódban az 5 és a -5 az `abs` függvénynek átadott argumentumok.

Néhány függvény több argumentumot is vár. A beépített `pow` függvény például kettőt: egy alapot, és egy kitevőt. A hívásnál átadott értékek a függvényen belül változókhoz lesznek rendelve. Az ilyen változókat **(formális) paramétereknek** nevezzük:

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

Egy másik beépített függvény, amely több paramétert is vár, a `max`:

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

A `max` függvénynek tetszőleges számú argumentumot átadhatunk, egymástól vesszővel elválasztva. A program az átadott számok legnagyobbikával tér vissza. Az argumentumok lehetnek egyszerű értékek vagy kifejezések is. Az utolsó példában a `max` függvény 503-as értéket ad vissza, ugyanis az nagyobb a 33-nál, 125-nél és az 1-nél is.

4.5. Visszatérési értékkel rendelkező függvények

Az előző fejezetben álló függvények mindegyike adott vissza értéket. A `range`-hez, `int`-hez vagy `abs`-hoz hasonló függvények által visszaadott értékeket összetettebb kifejezések építésére is használhatjuk.

A `negyzet_rajzolo` lényegesen eltér ezektől, hiszen nem egy érték előállítása miatt hívtuk meg, hanem azért, mert olyan lépések sorozatát kívántuk végrehajtani, melyek rajzolásra bírnak egy teknőcöt.

Azokat a függvényeket, amelyek értéket állítanak elő, *ebben a könyvben*, **produktív függvényeknek** nevezzük. A produktív függvények ellentétei a **void függvények**. Utóbbiak azok, amiket nem a visszatérési értékük miatt hívunk meg, hanem azért mert valami hasznosat visznek véghez. (Számos programnyelv, mint például a Java, C#, C és a C++

„void függvényeknek” nevezi ezeket, míg más nyelvekben, például Pascalban, az **eljárás** megnevezés a használatos.) Habár a void függvényeknél a visszatérési érték nem érdekes számunkra, a Python mindig vissza akar adni valamilyen értéket. Ha a programozó nem rendelkezik róla, akkor automatikusan `None` értéket juttat vissza a hívóhoz.

Hogyan írhatunk saját, produktív függvényt? A 2. fejezet végén található feladatoknál láttuk a kamatos kamatszámítás általános képletét. Most függvényként fogjuk megírni:

$$FV = C \cdot \left(1 + \frac{r}{m}\right)^{mt}$$

Ahol,

- **C:** alaptőke
- **r:** éves névleges kamatláb
- **m:** évközi kamatozások száma
- **t:** évek száma

```
1 def kamatos_kamat(c, r, m, t):
2     """A futamidő végén kapott érték számítása c befektetett összegre
3         a kamatos kamat képletének megfelelően."""
4
5     fv = c * (1 + r/m) ** (m*t)
6     return fv # Ez az újdonság tesz a függvényt *produktív* függvénné.
7
8 # Most, hogy van egy függvényünk, hívjuk is meg!
9 befektetettOsszeg = float(input("Mekkora összeget kíván befektetni?"))
10 vegOsszeg = kamatos_kamat(befektetettOsszeg, 0.08, 12, 5)
11 print("A futamidő végén Önnek ennyi pénze lesz: ", vegOsszeg)
```

- A **return** utasítást egy kifejezés követi (itt: `fv`). A kifejezés kiértékelésével kapott érték mint produktum kerül vissza a hívóhoz.
- A befektetni kívánt összeget a felhasználótól kérjük be az `input` függvénnyel. A visszakapott érték típusa sztring, nekünk viszont egy számra van szükségünk. Az esetleges tizedesjegyekre is szükségünk van a pontos számításhoz, ezért a `float` típuskonverziós függvénnyel alakítjuk át a sztringet valós számmá.
- A feladatban a kamatláb 8%, az évközi kamatozások száma 12, a futamidő pedig 5 év volt. Figyeld meg, hogyan adtuk meg a feladatnak megfelelő argumentumokat a függvényhívásnál.
- **Ha befektetett összegként 10000 Ft-ot adunk meg (input: 10000) akkor az alábbi kimenetet kapjuk:** *A futamidő végén Önnek ennyi pénze lesz: 14898.45708301605*

Ennyi tizedesjegy egy kicsit furcsának tűnhet, de hát a Python nincs tisztában azzal, hogy most pénzzel dolgozunk. Elvégzi a számítást a legjobb tudása szerint, kerekítés nélkül. Egy későbbi fejezetben majd látni fogjuk, hogyan jeleníthetjük meg szépen, két tizedesjegyre kerekített formában az üzenetben álló összeget.

- A `befektetettOsszeg = float(input("Mekkora összeget kíván befektetni?"))` sor újabb példa az egymásba ágyazható függvényekre. Meghívhatunk egy `float`-szerű függvényt úgy, hogy argumentumként egy másik függvény (jelen esetben az `input`) visszatérési értékét használjuk.

Van itt még egy nagyon fontos dolog! Az argumentumként átadott változó nevének (`befektetettOsszeg`) semmi köze nincs a paraméter nevéhez (`c`). Olyan, mintha a `kamatos_kamat` függvény hívásakor végbemenne egy `c = befektetettOsszeg` utasítás. Teljesen mindegy, hogy a hívásnál milyen névvel hivatkozunk az értéket, a `kamatos_kamat` függvényben a neve `c` lesz.

A rövid változónevek most már zavaróak lehetnek, talán jobban értékelnék az alábbi verziók valamelyikét:


```
1 def kamatos_kamat_v2(befektetettOsszeg, nevlegesKamatlab,  
2     evkoziKamatozasokSzama, evekSzama):  
3     vegOsszeg = befektetettOsszeg * (1 + nevlegesKamatlab /  
4         evkoziKamatozasokSzama) **  
5     ↪ (evkoziKamatozasokSzama*evkekSzama)  
6     return vegOsszeg  
7  
8 def kamatos_kamat_v3(betet, kamat, evkozi, evek):  
9     vo = betet * (1 + kamat/evkozi) ** (evkozi*evkek)  
10    return vo
```

Mindegyik változat ugyanazt csinálja. Válassz saját belátásod szerint olyan neveket, hogy a programjaid mások is könnyen megérthessék. A rövid változónevek használta „gazdaságos”, és javíthatják a kód olvashatóságát is. Az $E = mc^2$ -et sem lehetne olyan könnyen megjegyezni, ha Einstein hosszabb változóneveket adott volna! Amikor a rövid nevek mellett döntesz, mindenképpen tegyél megjegyzéseket a programodba a változók szerepének tisztázására.

4.6. A változók és a paraméterek lokálisak

A függvényeken belül létrehozott **lokális változók** csak a tartalmazó függvény belsejében léteznek, azokon kívül nem használhatóak. Példaként nézzük meg újra a `kamatos_kamat` függvényt.

```
1 def kamatos_kamat(c, r, m, t):  
2     fv = c * (1 + r/m) ** (m*t)  
3     return fv
```

Ha megpróbáljuk az `fv`-t a függvényen kívül használni, hibaüzenetet kapunk.

```
print(fv)  
NameError: name 'fv' is not defined
```

Az `fv` nevű változó a `kamatos_kamat` lokális változója, a függvényen kívül nem látható.

Ráadásul az `fv` csak a függvény végrehajtása alatt létezik, ez az **élettartama**. Amint egy függvény működése befejeződik, a lokális változói megsemmisülnek.

A (formális) paraméterek szintén lokálisak, és úgy is viselkednek, mint a lokális változók. A `c`, `r`, `m`, `t` paraméterek élettartama akkor kezdődik, amikor a `kamatos_kamat` függvény meghívásra kerül, és akkor ér véget, amikor a függvény befejezi a működését.

Nem történhet olyasmi, hogy egy függvény beállít egy értéket valamelyik lokális változójának, befejezi a működését, majd egy újabb hívásnál előszedi a korábban beállított értéket. A függvény minden egyes hívásakor új lokális változók jönnek létre, és az élettartamuk lejár, amikor a függvénytől visszatér a vezérlés a hívóhoz.

4.7. Teknőc revízió

Most, hogy már ismerjük a produktív függvényeket is, átalakíthatjuk a korábbi programjainkat úgy, hogy azok jobban illeszkedjenek meghatározható részfeladatokhoz. Az újraszervezés folyamatát nevezzük a kód **refaktorálásának**.

Két feladat mindig elő fog kerülni, amikor teknőcökkel dolgozunk: ablakot kell készítenünk a teknőcök számára, és létre kell hozni egy vagy több teknőcot. Írhatnánk is két függvényt, melyek a későbbiekben egyszerűbbé teszik ezeknek a lépéseknek a megvalósítását.

```
1 def ablak_keszites(szin, ablaknev):
2     """
3     Egy ablak elkészítése, és a háttérszín, valamint az ablaknév
4     ↪beállítása.
5     Visszatérési érték: az új ablak.
6     """
7     a = turtle.Screen()
8     a.bgcolor(szin)
9     a.title(ablaknev)
10    return a
11
12 def teknoc_keszites(szin, tm):
13     """
14     Létrehoz egy teknőcöt, és beállítja az általa használt toll
15     színét és méretét.
16     Visszatérési érték: az új teknőc.
17     """
18     t = turtle.Turtle()
19     t.color(szin)
20     t.pensize(tm)
21     return t
22
23
24 a = ablak_keszites("lightgreen", "Eszti és Sanyi táncol")
25 Eszti = teknoc_keszites("hotpink", 5)
26 Sanyi = teknoc_keszites("black", 1)
27 David = teknoc_keszites("yellow", 2)
```

Az újrászervezés titka abban áll, hogy előre kitaláljuk, hogy melyek azok a dolgok, amiket szinte minden függvényhívás alkalmával meg szeretnénk majd változtatni, ugyanis ezek lesznek a függvényünk paraméterei, megváltoztatható kódrészletei.

4.8. Szójegyzék

argumentum (argument) A függvények hívásakor a függvényeknek átadott értékeket argumentumoknak, más néven *aktuális paramétereknek* nevezzük. Az argumentumok a függvény definiálásánál megadott *formális paraméterek*hez rendelődnek hozzá. Argumentumként kifejezések használhatók, amelyek tartalmazhatnak operandusokat, műveleti jeleket, de akár visszatérési értékkel rendelkező függvények hívását is.

dokumentációs sztring (docstring) Egy speciális sztring, mely egy függvényhez kötődik annak `__doc__` attribútumaként. A különböző programozási környezetek, mint például a PyCharm, a dokumentációs sztring alapján tud leírást szolgáltatni a függvényekről a programozók számára. A modulok, osztályok, metódusok tárgyalásánál látni fogjuk, hogy ezek a speciális sztringek ott is használhatók.

élettartam (lifetime) A változóknak és objektumoknak van élettartama: a program futásának bizonyos pontján létrejönnek, majd később megsemmisülnek.

fejléc (header line) Az összetett utasítások első része. Kulcsszóval kezdődik és kettősponttal (:) zárul.

függvény (function) Egy névvel ellátott utasítássorozat, mely valamilyen hasznos tevékenységet végez. A függvényeknek lehetnek formális paraméterei, és adhatnak vissza értéket.

függvény definíció (function definition) Egy olyan utasítás, mely egy függvényt hoz létre a függvény nevének, paramétereinek és az általa végrehajtandó utasítások meghatározásával.

függvények egymásba ágyazása (function composition) Egy visszatéréssel rendelkező függvény argumentumként való szerepeltetése egy függvényhívásban.

függvényhívás (function call) Egy függvény végrehajtását végző utasítás. Az utasítás a meghívandó függvény nevével kezdődik, amit a kerek zárójelek közé tett argumentumlista (aktuális paraméterek listája) követ.

import utasítás (import statement) Egy olyan utasítás, mely lehetővé teszi, egy Python modulban definiált függvények és változók más szkriptben való felhasználását. Például a teknőcök eléréséhez is mindig importálnunk kellett a turtle modult.

keret (frame) Egy adott függvényhíváshoz tartozó rész a veremben. A függvény lokális változóit és paramétereit tartalmazza.

lokális változó (local variable) Egy függvény belsejében definiált változót az adott függvényre nézve lokálisnak nevezünk. A lokális változók csak a tartalmazó függvényen belül használhatók. A függvény paraméterei tekintetők speciális lokális változóknak.

összetett utasítás (compound statement) Egy olyan utasítás, mely az alábbi két részből áll: #. fejléc: kulcsszóval kezdődik és kettősponttal záródik, #. törzs: egy vagy több utasítást tartalmaz. Az utasítások a fejléctől nézve azonos mértékű behúzással állnak.

Az összetett utasítás szintaktikája az alábbi:

```
kulcsszó ... :  
    utasítás  
    utasítás ...
```

paraméter (parameter) Egy olyan név, mely a függvényen belül használható. A paraméterek arra az értékre hivatkoznak, amelyet a függvényhívásnál argumentumként megkapnak.

produktív függvény (fruitful function) Egy olyan függvény, mely értéket ad vissza, amikor meghívják. (Könyvbeli megnevezés.)

programvezérlés (flow of execution) Az a sorrend, ahogyan az utasítások végrehajtásra kerülnek a program futása során.

refaktorálás (refactor) Ez a fellengzős szó a programkód olyan átszervezésére utal, aminek célja a program átláthatóbbá tétele. Tipikusan akkor hajtjuk végre, amikor már működik a program. A folyamat gyakran tartalmazza a változónevek megfelelőbbre való cserélését, valamint az ismétlődő programrészek felismerését és függvényekbe történő kiemelését.

törzs (body) Az összetett utasítás második része. A törzs egy utasítássorozatot tartalmaz. Minden utasításnak a fejtől azonos mértékkel balra kell kezdődnie. A szabványos behúzás Pythonban 4 szóköz.

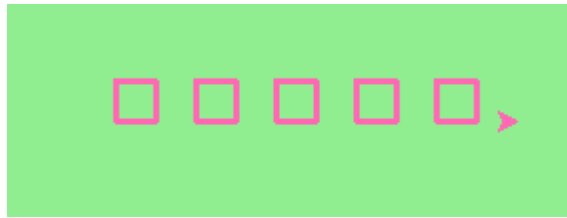
verem diagram (stack diagram) A (hívási) verem grafikus reprezentációja. A függvények hívása során verembe kerülő információkat, a függvények paramétereit, lokális változóit és azok értékeit jeleníti meg.

visszakövetés (traceback, stack trace) A végrehajtás során meghívott függvények listája, mely a futási idejű hibák-nál megjelenítésre kerül. A meghívott függvények a listában olyan sorrendben szerepelnek, ahogyan a **futási veremben** is, innen ered az angol *stack trace* elnevezés.

void függvény (void function) Egy olyan függvény, mely nem ad vissza értéket. Az általa végrehajtott tevékenység a lényeges. (Egyes programnyelvekben *eljárásnak* nevezik.)

4.9. Feladatok

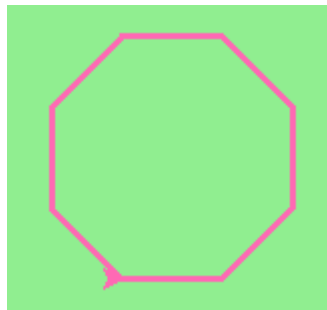
1. Készíts egy void függvényt, mely egy négyzetet rajzol. Használd fel egy olyan program elkészítéséhez, mely az alábbi ábrát hozza létre. Minden egyes négyzet legyen 20 egység. (Segítség: mire a program véget ér, a teknőc már elmozdul arról a helyről, ahová az utolsó négyzetet rajzolta.)



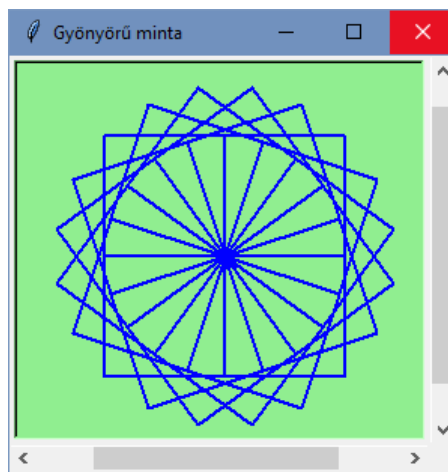
2. Írj egy programot, mely az alábbi ábrának megfelelő alakzatot rajzolja ki. A legbelső négyzet minden oldala 20 egység hosszú. A többi négyzet oldalhosszúsága minden esetben 20 egységgel nagyobb, mint az általa tartalmazott legnagyobb négyzet oldalhosszúsága.



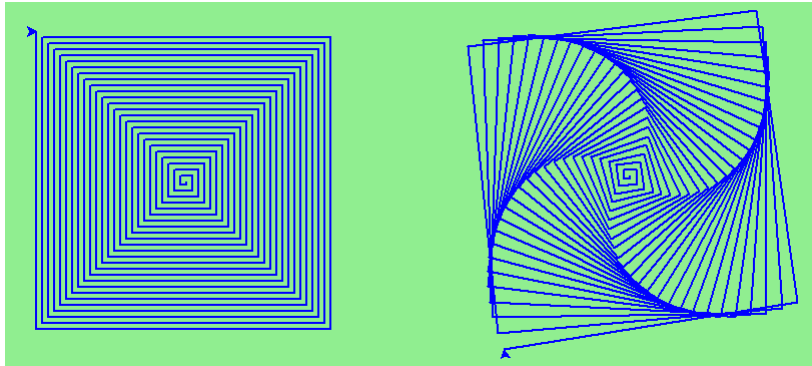
3. Írj egy `sokszog_rajzolas(t, n, sz)` fejlécű void függvényt, mely a teknőccel egy szabályos sokszöget rajzoltat. Ha majd meghívod a `sokszog_rajzolas(Eszti, 8, 50)` utasítással, akkor az alábbihoz hasonló ábrát kell kapnod:



4. Rajzold meg, ezt a gyönyörű ábrát:



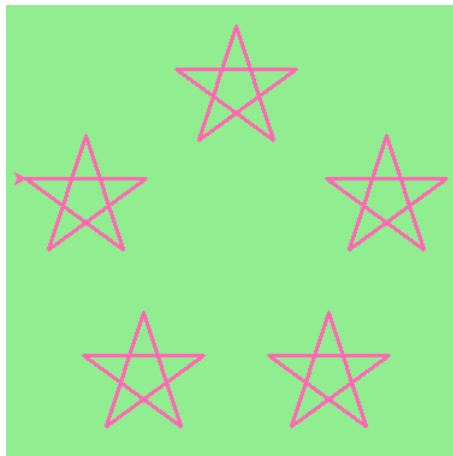
5. Az alábbi ábrán lévő két spirál csak a fordulási szögben tér el. Rajzold meg mind a kettőt.



6. Készíts egy `szabalyos_haromszog_rajzolas(t, sz)` fejlécű void függvényt, mely az előző feladatban szereplő `poligon_rajzolas` függvényt meghívva egy szabályos háromszöget rajzoltat a teknőccel.
7. Készíts egy `osszeg(n)` fejlécű produktív függvényt, amely összegzi az 1 és n közé eső egész számokat, a határokat is beleértve. Például `osszeg(10)` hívás esetében az $1+2+3+...+10$ eredményét, vagyis 55-öt kell visszaadni a függvénynek.
8. Írj egy `kor_terulet(r)` fejlécű produktív függvényt, amely egy r sugarú kör területét adja vissza.
9. Készíts egy void függvényt, mely egy olyan csillagot rajzol ki, melynek minden oldala pontosan 100 egység hosszúságú. (Segítség: 144 fokkal kell elforgatni a teknőcöt minden csúcsban.)



10. Bővítsd ki az előző feladatot programmá. Rajzolj öt csillagot, de minden egyes csillag rajzolása közt emeled fel a tollat, haladj előre 650 egységet és fordulj jobbra 144 fokkal, majd rakd le a tollat. Valami ilyesmit kell kapnod:



Hogyan nézne ki az ábra, ha nem emelnéd fel a tollat?

5. fejezet

Feltételes utasítások

A programok akkor válnak igazán érdekessé, ha feltételeket tesztelhetünk és megváltoztathatjuk a program viselkedését a teszt kimenetelétől függően. Erről szól ez a fejezet.

5.1. Boolean értékek és kifejezések

Egy *Boolean* (azaz logikai) érték vagy igaz, vagy hamis. A nevét a brit matematikusról, George Boole-ról kapta, aki először írta le a *Boole-algebrát* – néhány szabályt az érveléshez és ezeknek az értékeknek a kombinálásához. Ez az alapja minden modern számítógép logikájának.

Pythonban a két Boolean érték a `True` azaz igaz és a `False` azaz hamis (csak az első betű nagy), valamint a Python típus neve **bool**.

```
>>> type(True)
<class 'bool'>
>>> type(true)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'true' is not defined
```

Egy **Boolean kifejezés** egy olyan kifejezés, amelynek kiértékelése során Boolean értéket kapunk. Például az `==` operátor azt vizsgálja, hogy két érték egyenlő-e. Ez Boolean értéket eredményez:

```
>>> 5 == (3 + 2)    # Egyenlő-e vajon az 5 a 3+2 eredményével?
True
>>> 5 == 6
False
>>> j = "hel"
>>> j + "lő" == "hellő"
True
```

Az első állításban a két operandus kiértékelés során egyenlőnek bizonyul, így a kifejezés értéke `True` lesz. A második állításban mivel az 5 nem egyenlő a 6-tal így `False` értéket kapunk.

Az `==` operátor az egyike annak a hat közönséges **összehasonlító operátornak**, amelyek `bool` értéket eredményeznek. Itt van mind a hat:

```
x == y    # True (igaz) értéket ad ha ... x egyenlő y-nal
x != y    # ... x nem egyenlő y-nal
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
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
```

Habár ezek az operátorok valószínűleg ismerősek számodra, a Python jelölése különbözik a matematikai jelölésektől. Egy megszokott hiba az egy egyenlőségjel (=) használata a dupla egyenlőségjel (==) helyett. Jegyezd meg, hogy az = az értékadás operátora és az == az összehasonlítás operátora. Továbbá nincsenek =< vagy => jelölések.

Mint a többi típus esetén is, amiket később látni fogunk, a Boolean értékek is értékül adhatóak változóknak, kiírathatóak, stb.

```
>>> kor = 18
>>> eleg_idos_a_jogsihoz = kor >= 17
>>> print(eleg_idos_a_jogsihoz)
True
>>> type(eleg_idos_a_jogsihoz)
<class 'bool'>
```

5.2. Logikai operátorok

Van három **logikai operátor**, `and` (és), `or` (vagy) valamint a `not` (nem), amelyek lehetővé teszik számunkra, hogy bonyolultabb Boolean vagyis logikai kifejezéseket is létrehozzunk egyszerűbbekből. Ezen operátorok szemantikája (jelentése) hasonló az angol jelentéseikhez. Például az `x > 0 and x < 10` kifejezés `True` értéket kizárólag akkor eredményez, ha `x` egyszerre nagyobb, mint 0 és kisebb, mint 10.

Az `n % 2 == 0 or n % 3 == 0` kifejezés akkor ad `True` értéket, ha legalább az *egyik* feltétel igaz, azaz ha az `n` szám osztható 2-vel vagy 3-mal egy adott időpontban. (Mit gondolsz, mi történik, ha `n` egyszerre 2-vel és 3-mal is osztható? A kifejezés `True` vagy `False` értéket ad? Próbáld ki a Python parancsértelmezővel!)

Végül, a `not` operátor negálja (azaz tagadja) a Boolean értéket, így a `not (x > y)` kifejezés `True` értékű, ha az `(x > y)` kifejezés `False` (hamis), azaz ha `x` kisebb, vagy egyenlő `y`-nál.

Az `or` operátor bal oldala értékelődik ki először: ha az eredmény `True`, akkor a Python nem értékeli ki (mivel nem is szükséges kiértékelnie) a kifejezést a jobb oldalon – ez a *rövidzár kiértékelés*. Hasonlóan az `and` operátor esetén, ha a bal oldali kifejezés `False` értékű, a Python nem értékeli ki a jobb oldali kifejezést.

Így nincsenek felesleges kiértékelések.

5.3. Igazságtáblák

Az igazságtábla egy kis táblázat, amely lehetővé teszi, hogy listázzuk az összes lehetséges inputot és megadjuk a logikai operátorok eredményeit. Mivel az `and` és az `or` operátor is két operandussal rendelkezik, csak négy sor van az igazságtáblájukban. Az `and` szemantikáját a következő igazságtábla írja le:

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Az igazságtáblában gyakran csak T és F betűket írunk a Boolean értékek rövidítésének megfelelően. Az alábbi igazságtábla írja le az `or` logikáját:

a	b	a or b
F	F	F
F	T	T
T	F	T
T	T	T

A harmadik logikai operátor a `not`, csak egy operandussal rendelkezik, így az igazságtáblájában csak két sor van:

a	not a
F	T
T	F

5.4. Boolean kifejezések egyszerűsítése

A szabályok halmazát, amely segítségével egyszerűsíthetjük és átrendezhetjük a kifejezéseket, *algebrának* hívjuk. Például mindannyian járatosak vagyunk az iskolai algebra szabályaiban, mint például:

```
n * 0 == 0
```

Most mi egy ettől eltérő algebrát – *Boole* algebrát – használunk, amely szabályokat biztosít a Boolean értékekkel való munkához.

Először az `and` operátor:

```
x and False == False
False and x == False
y and x == x and y
x and True == x
True and x == x
x and x == x
```

Itt vannak az `or` operátor hasonló szabályai:

```
x or False == x
False or x == x
y or x == x or y
x or True == True
True or x == True
x or x == x
```

Két `not` operátor érvényteleníti egymást:

```
not (not x) == x
```

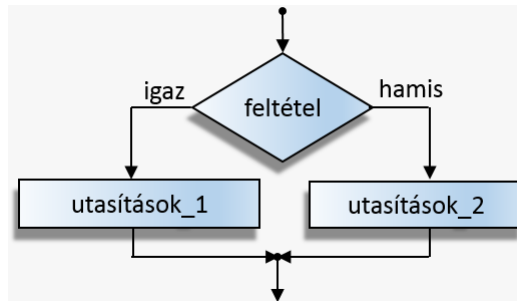
5.5. Feltételes végrehajtás

Hasznos programok írásához szinte mindig szükségünk van a feltételek ellenőrzésének képességére és arra, hogy ezek alapján megváltoztassuk a program viselkedését. A **feltételes utasítások** adják meg nekünk ezt a képességet. A legegyszerűbb formája ennek az `if` utasítás:


```
1 if x % 2 == 0:
2     print(x, " páros szám.")
3     print("Tudtad, hogy a 2 az egyetlen páros prímszám?")
4 else:
5     print(x, " páratlan szám.")
6     print("Tudtad, hogy két páratlan számot összeszorozva " +
7           "az eredmény mindig páratlan?")
```

Az `if` kulcsszó utáni logikai kifejezés a **feltétel**. Ha ez igaz, akkor az ezt követő mindegyik indented utasítás végrehajtódik. Ha nem igaz, akkor az `else` kulcsszó utáni behúzott utasítások hajtódnak végre.

Folyamatábra egy olyan `if` utasításról, amelynek `else` ága is van



Az `if` utasítás szintaktikája így néz ki:

```
1 if BOOLEAN_KIFEJEZÉS:
2     UTASÍTÁS_1           # Végrehajtódik, ha a feltétel kiértékelése igaz.
    ↳értéket ad
3 else:
4     UTASÍTÁS_2           # Végrehajtódik, ha a feltétel kiértékelése hamis.
    ↳értéket ad
```

Mint az előző fejezetben bemutatott függvénydefiníciók vagy bármely másik összetett utasítás esetén, mint például a `for` ciklus esetén, úgy az `if` utasítás is tartalmaz egy fejléc sort és egy törzset. A fejléc sor az `if` kulcsszóval kezdődik, amit egy *Boolean kifejezés* követ, majd kettőspont (`:`) karakterrel zárul.

Az ezt követő behúzott vagyis indented sorok alkotják a **törzset**. Az első nem indented sor jelenti a törzs végét.

Az első blokk összes utasítása sorban végre lesz hajtva, ha a Boolean kifejezés kiértékelése `True` értéket eredményez. Az első utasításblokk egésze ki lesz hagyva, ha a logikai kifejezés `False` értékű, és ezek helyett az `else` alatti összes indented sor hajtódnak végre.

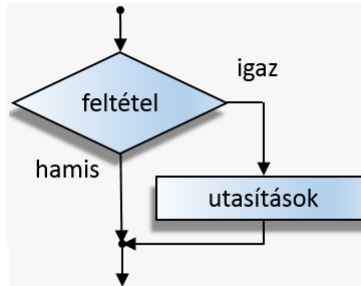
A két utasításblokk közül tehát pontosan az egyik hajtódnak végre és ezután a vezérlés a teljes `if` utasítás után következő utasításra ugrik.

Nincs limit arra vonatkozóan, hogy hány utasítás jelenhet meg az `if` utasítás két ágában, de minden blokkban legalább egy utasításnak kell szerepelnie. Néha hasznos lehet egy utasítások nélkül ág is (rendszerint amiatt, hogy fenntartsuk a helyet a kód számára, amit még nem írtunk meg). Ebben az esetben használhatjuk a `pass` utasítást, amely nem csinál semmit, csak helyőrzőként szerepel.

```
1 if True:           # Ez mindig True,
2     pass           # így ez hajtódnak végre, de nem csinál semmit.
3 else:
4     pass
```

5.6. Az `else` ág kihagyása

Folyamatábra egy `else` ág nélküli `if` utasításról



Az `if` utasítás másik alakja az, amelyben az `else` ágot teljes egészében kihagyjuk. Ebben az esetben, amikor a feltételkiértékelése `True` értéket ad, az indentált utasítások végrehajtódnak, ellenkező esetben a program végrehajtása az `if` utáni utasítással folytatódik.

```
1 if x < 0:
2     print("Negatív számnak, mint a", x, "itt nincs értelme.")
3     x = 42
4     print("Úgy döntöttem, a szám legyen inkább a 42.")
5
6 print("Kiszámoltam, hogy", x, "négyzetgyöke", math.sqrt(x))
```

Ebben az esetben a `print` függvény, amely kiírja a négyzetgyököket, az lesz az `if` utáni utasítás – nem azért, mert van előtte egy üres sor, hanem azért mert nincs indentálva. Azt is jegyezd meg, hogy a `math.sqrt(x)` függvényhívás hibát fog adni, ha csak nincs egy `import math` utasítás valahol a szkript elején.

Python terminológia

A Python dokumentációban a blokk kifejezésre gyakran más szót is használnak, de mivel a blokk a programozásban sokkal elfogadottabb, mi is ezt használjuk.

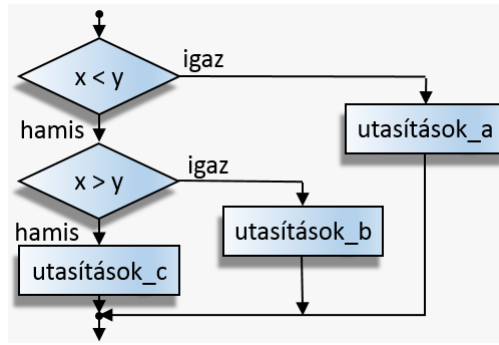
Vedd észre, hogy az `else` nem egy külön utasítás. Az `if` utasításnak van két ága, az egyik opcionális és az `else` kulcsszóval kezdődik.

5.7. Láncolt feltételes utasítások

Néha kettőnél több lehetőség van és szükségünk van kettőnél több ágra. Az egyik módja az ilyen számítások kifejezésére a **láncolt feltételes utasítás** használata:

```
1 if x < y:
2     UTASÍTÁSOK_A
3 elif x > y:
4     UTASÍTÁSOK_B
5 else:
6     UTASÍTÁSOK_C
```

A fenti láncolt feltételes utasítás folyamatábrája



Az `elif` kulcsszó az `else if` rövidítése. Ismét pontosan egy ág fog végrehajtódni. Nincs korlát az `elif` blokkok számára vonatkozólag, de csak egyetlen (opcionális) `else` ág megengedett az utasítás végén:

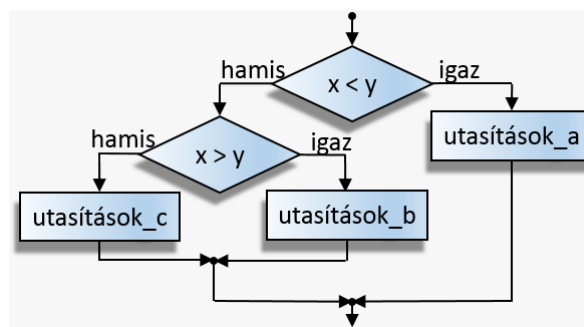
```
1 if valasztas == "a":
2     fuggveny_egy()
3 elif valasztas == "b":
4     fuggveny_ketto()
5 elif valasztas == "c":
6     fuggveny_harom()
7 else:
8     print("Érvénytelen választás.")
```

A feltételek sorrendben lesznek kiértékelve. Ha az első hamis, a következő lesz megvizsgálva, és így tovább. Ha egyikük igaz, akkor a hozzá tartozó blokk végrehajtódik és az összetett utasítás befejeződik. Ha több, mint egy feltétel igaz, akkor is csak az első ág lesz végrehajtva. Ha mindegyik feltétel hamis és van `else` ág, akkor az hajtódik végre.

5.8. Beágyazott feltételes utasítások

Egy feltételes utasítás **beágyazható** egy másikba. (Ez az összerakhatóság egyik fajtája.) Az előbbi példát így is írhattuk volna:

A beágyazott feltételes utasítás folyamatábrája



```
1 if x < y:
2     UTASÍTÁSOK_A
3 else:
4     if x > y:
5         UTASÍTÁSOK_B
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6     else:
7         UTASÍTÁSOK_C
```

A külső feltételes utasítás két ágat tartalmaz. A második ág (else) egy újabb `if` utasítást tartalmaz az ő két saját ágával. Ezek az ágak tartalmazhatnak további feltételes utasításokat is.

Habár az indentálások teszik a szerkezetet világossá, a beágyazott feltételes utasítások olvasása könnyen nehezzé válhat. Általában jó ötlet elkerülni őket, ha lehet.

A logikai operátorok gyakran lehetőséget nyújtanak arra, hogy egyszerűsítsük a beágyazott feltételes utasításokat. Például újraírhatjuk az alábbi kódot egyetlen feltételes utasítás segítségével:

```
1  if 0 <= x:                # Tegyük fel, hogy x itt egy int típusú érték
2      if x < 10:
3          print("x egy számjegy.")
```

A `print` függvényt csak akkor hívjuk meg, ha túljutunk mind a két feltételen, így a fenti kód helyett, amely két elágazást is használ egy-egy feltétellel, írhatunk egy komplexebb feltételt az `and` operátor használatával. Most csak egy `if` utasításra van szükségünk:

```
1  if 0 <= x and x < 10:
2      print("x egy számjegy.")
```

5.9. A `return` utasítás

A `return` utasítás értékkel vagy érték nélkül, attól függően, hogy a függvény produktív vagy void, lehetővé teszi, hogy befejezzük a függvény végrehajtását mielőtt (vagy amikor) elérjük a törzsének a végét. Az egyik ok, ami miatt *korai visszatérést* alkalmazunk, hogy hibafeltételt észlelünk:

```
1  def ird_ki_a_negyzetgyokot(x):
2      if x < 0:
3          print("Pozitív szám vagy nulla kell.")
4          return
5
6      eredmeny = x**0.5
7      print(x, "négyzetgyöke", eredmeny)
```

Az `ird_ki_a_negyzetgyokot` függvénynek van egy `x` nevű paramétere. Az első dolog ellenőrizni, hogy `x` kisebb-e, mint 0, amely esetben egy hibüzenet jelenik meg és `return` utasítást használunk a függvény befejezéséhez. A programvezérlés rögtön visszatér a hívóhoz, és a függvény további sorai nem lesznek végrehajtva.

5.10. Logikai ellentétek

Mind a hat relációs operátornak van egy logikai ellentéte: például ha feltételezzük, hogy akkor kaphatunk jogosítványt, ha a korunk nagyobb vagy egyenlő, mint 17, akkor nem kaphatunk jogosítványt, ha a korunk 17-nél kisebb.

Vedd észre, hogy a `>=` ellentéte a `<`.

operátor	logikai ellentét
==	!=
!=	==
<	>=
<=	>
>	<=
>=	<

Ezeknek a logikai ellentéteknek a megértésével néha megszabadulhatunk a `not` operátortól. A `not` operátort néha elég nehéz olvasni a számítógépes kódokban és a szándékunk rendszerint tisztább lesz, ha kiküszöböljük őket.

Például, ha ezt írjuk Pythonban:

```
1 if not (kor >= 17):
2     print("Hé, Te túl fiatal vagy a jogsihoz!")
```

talán tisztább lenne az egyszerűsítő szabályok használatával ezt írni helyette:

```
1 if kor < 17:
2     print("Hé, Te túl fiatal vagy a jogsihoz!")
```

Két hathatós egyszerűsítési szabály (amelyek *de Morgan azonosságok* névre hallgatnak) gyakran segíthet, ha komplikált logikai kifejezésekkel kell dolgoznunk.

```
not (x and y) == (not x) or (not y)
not (x or y)  == (not x) and (not y)
```

Ha például feltesszük azt, hogy csak akkor győzhetünk le egy sárkányt, ha a mágikus fénykardunk töltöttsége legalább 90%, és ha legalább 100 energiaegység van a védőpajzsunkban. A játék Python kódtörékében ezt találjuk:

```
1 if not ((kard_toltes >= 0.90) and (pajzs_energia >= 100)):
2     print("A támadásod hatástalan, a sárkány szénné éget!")
3 else:
4     print("A sárkány összeesik. Megmented a káprázatos hercegnőt!")
```

A logikai ellentétekkel együtt a *de Morgan azonosságok* lehetővé teszik, hogy újradolgozzuk a feltételt egy (talán) könnyebben érthető módon:

```
1 if (kard_toltes < 0.90) or (pajzs_energia < 100):
2     print("A támadásod hatástalan, a sárkány szénné éget!")
3 else:
4     print("A sárkány összeesik. Megmented a káprázatos hercegnőt!")
```

Megszabadulhatunk a `not` operátortól úgy is, ha felcseréljük az `if` és az `else` ágakat. Így a harmadik, szintén egyenértékű verzió ez:

```
1 if (kard_toltes >= 0.90) and (pajzs_energia >= 100):
2     print("A sárkány összeesik. Megmented a káprázatos hercegnőt!")
3 else:
4     print("A támadásod hatástalan, a sárkány szénné éget!")
```

A három közül talán ez a legjobb verzió, mivel elég hasonló az eredeti magyar mondathoz. Mindig előnyben kell részesítenünk a kódunk tisztaságát (mások számára) valamint azt, hogy láthatóvá tesszük, hogy a kód az, amit elvárunk.

Ahogy a programozási képességünk fejlődik, több mint egy megoldási módot fogunk találni a problémákra. A jó program *megtervezett*. Választásaink kedvezni fognak az átláthatóságnak, egyszerűségnek és eleganciának. A *program-*

tervező szakma neve sokat elárul arról, hogy mit csinálunk – egyfajta mérnökként tervezzük a terméket egyensúlyban tartva a szépséget, használhatóságot, egyszerűséget és tisztaságot az alkotásunkban.

Javaslat: Ha egyszer a programod működik, próbálj meg egy kicsit polírozni rajta! Írj jó megjegyzéseket! Gondolkodj el azon, hogy a kód nem lenne-e tisztább más változónevekkel! Meglehetne csinálni elegánsabban? Inkább függvényt kellene használni? Egyszerűsíthetők a feltételes utasítások?

Gondoljunk úgy a kódunkra, mint művészeti alkotásunkra! Tegyük egyszerűvé!

5.11. Típuskonverzió

Már vetettünk egy pillantást erre az egyik korábbi fejezetben. Nem árt azonban, ha újra átnézzük.

Sok Python típusnak van beépített függvénye, amelyek megpróbálnak különböző típusú értékeket a saját típusukra átalakítani. Az `int` függvény például mindenféle értéket egész típusúvá konvertál, ha lehetséges, különben panasz-kodik:

```
>>> int("32")
32
>>> int("Helló")
ValueError: invalid literal for int() with base 10: 'Helló'
```

Az `int` lebegőpontos (azaz valós) számokat is tud egészszé konvertálni, de emlékezz a tört rész csonkolására:

```
>>> int(-2.3)
-2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

A `float` függvény egészeket és sztringeket konvertál lebegőpontos számmá:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

Furcsa lehet, de a Python megkülönbözteti az egész típusú 1-et a valós 1.0-tól. Ezek ugyanazt a számot jelentik, de más típushoz tartoznak. Ennek oka az, hogy különbözőképpen vannak a számítógépben reprezentálva, eltárolva.

Az `str` függvény bármilyen típusú paraméterét sztringé alakítja:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'true' is not defined
```

Az `str` bármely típussal működni fog és azt sztringé alakítja. Ahogy korábban is említettük a `True` logikai érték, a `true` pedig csak egy közönséges változónév, ami itt nem definiált, így hibát kapunk.

5.12. Egy teknőc oszlopdiagram

A teknőcökben sokkal több rejlik, mint eddig láttuk. A teljes dokumentáció, amelyet a <http://docs.python.org/py3k/library/turtle.html> címen találhatunk, tartalmaz *Súgót* a teknőc modulról.

Itt van egy pár új trükk a teknőceinkkel kapcsolatban:

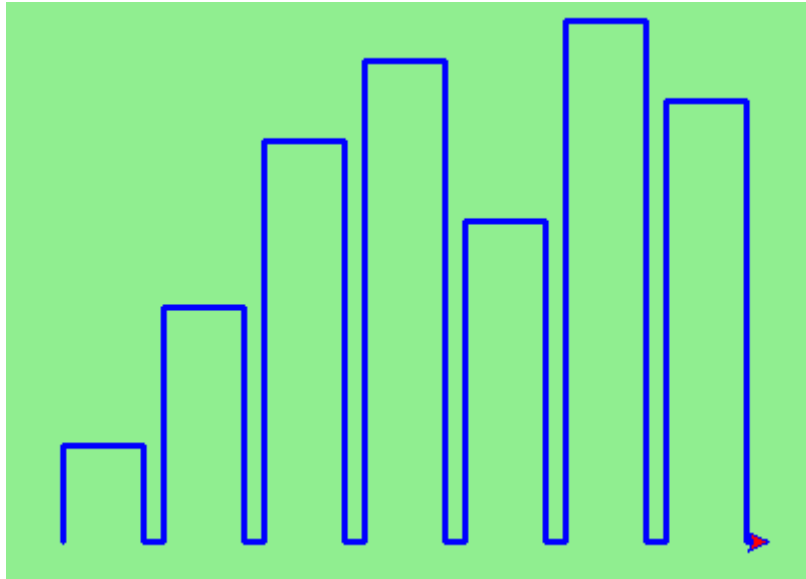
- Rávehetünk egy teknőcöt, hogy jelenítsen meg egy szöveget a vásznon. A teknőc aktuális pozíciójában. A metódus, amely ezt csinálja a `Sanyi.write("Helló")`.
- Ki tudunk tölteni egy alakzatot (kört, félkört, háromszöget, stb.) egy színnel. Ez egy több lépéses folyamat. Először meghívjuk a `Sanyi.begin_fill()` metódust, majd megrajzoljuk az alakzatot, végül meghívjuk a `Sanyi.end_fill()` metódust.
- Korábban beállítottuk a teknőc színét – most be tudjuk állítani a kitöltés színét is, amely nem kell, hogy azonos legyen a teknőc és a toll színével. A `Sanyi.color("blue", "red")` használata lehetővé teszi, hogy a teknőc kékkel rajzoljon, de pirossal töltsön ki.

Oké, így hogyan is tudjuk rávenni Esztit, hogy rajzoljon egy oszlopdiagramot? Először kezdjük néhány megjelenítendő adattal:

```
xs = [48, 117, 200, 240, 160, 260, 220]
```

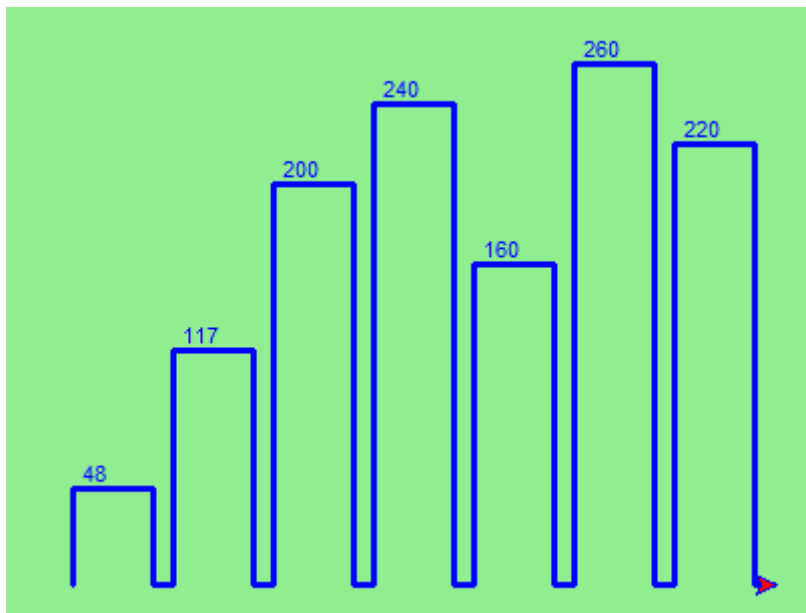
Az egyes mérési adatoknak megfelelően, rajolni fogunk egyszerű téglalapokat az adott magassággal és fix szélességgel.

```
1 def rajzolj_oszlopot(t, magassag):
2     """ A t teknőc oszlopot rajzol a megfelelő magassággal """
3     t.left(90)
4     t.forward(magassag) # Rajzold meg a bal oldalt!
5     t.right(90)
6     t.forward(40)       # Az oszlop szélessége a tetején.
7     t.right(90)
8     t.forward(magassag) # És ismét le.
9     t.left(90)          # Fordítsd a teknőcöt a megfelelő irányba!
10    t.forward(10)        # Hagyj egy kis rést minden oszlop után!
11
12    ...
13    for m in xs:          # Tegyük fel, Eszti és xs kész vannak!
14        rajzolj_oszlopot(Eszti, m)
```



Oké, nem fantasztikusan megkapó, de kezdetnek jó lesz. A lényeg itt a mentális blokkosítás vagyis az, hogy hogyan daraboljuk a problémát kisebb részekre. Az első egység egy oszlop rajzolása és írtunk egy függvényt ennek megtételére. Aztán az egész diagram a függvény ismételt meghívásával elkészíthető.

Következő lépésként, minden oszlop tetejére felírjuk az adat értékét. Ezt a `rajzolj_oszlopot` törzsében tesszük meg, kiegészítve a `t.write(' ' + str(magassag))` utasítással a törzs új harmadik sorában. Hagytunk egy szóközt a szám előtt és a számot a sztringbe helyeztük. A szóköz nélkül a szöveg esetenül kilógna az oszlop bal széléig. Az eredmény sokkal jobban néz ki:



És most két sort fogunk hozzáadni az oszlop kitöltéséhez. A programunk most így néz ki:

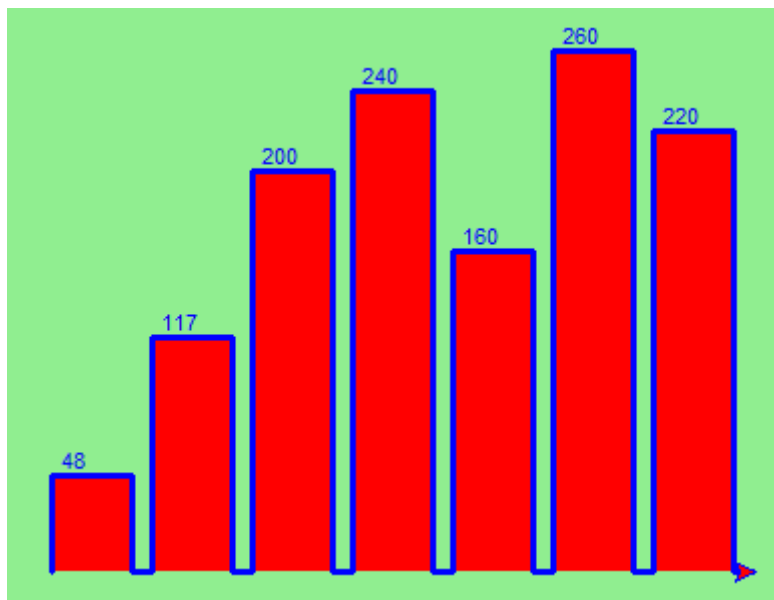
```
1 def rajzolj_oszlopot(t, magassag):  
2     """ A t teknőc oszlopot rajzol a megfelelő magassággal """  
3     t.begin_fill()           # Az új sor.  
4     t.left(90)  
5     t.forward(magassag)
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6     t.write("  "+ str(magassag))
7     t.right(90)
8     t.forward(40)
9     t.right(90)
10    t.forward(magassag)
11    t.left(90)
12    t.end_fill()           # A másik új sor.
13    t.forward(10)
14
15    ablak = turtle.Screen()   # Állítsd be az ablak tulajdonságait!
16    ablak.bgcolor("lightgreen")
17
18    Eszti = turtle.Turtle()   # Hozd létre Esztit, és állítsd be_
    ↪tulajdonságait!
19    Eszti.color("blue", "red")
20    Eszti.pensize(3)
21
22    xs = [48,117,200,240,160,260,220]
23
24    for m in xs:
25        rajzolj_oszlopot(Eszti, m)
26
27    ablak.mainloop()
```

Ez a következőt állítja elő, amely sokkal kielégítőbb:



Hmm. Talán az oszlopok alját nem kellene összekötni. Fel kell emelni a tollat az oszlopok közötti rés készítésekor. Ezt meghagyjuk a te feladatodnak.

5.13. Szójegyzék

ág (branch) A programvezérlés egyik lehetséges útvonala, amelyet egy feltétel határoz meg.

beágyazás (nesting) Egy programszerkezet egy másikon belül úgy, mint egy feltételes utasítás egy másik feltételes

utasítás egyik ágában.

blokk (block) Egymás utáni utasítások sorozata ugyanazzal az indentálással.

Boole algebra (Boolean algebra) Logikai kifejezések újrendezésére szolgáló néhány szabály.

Boolean érték (Boolean value) Pontosán két logikai érték: `True` vagyis igaz és `False` vagyis hamis. Ezek típusa `bool`. A Boolean kifejezés kiértékelése során a Python parancsértelmező Boolean értéket állít elő eredményként.

Boolean kifejezés (Boolean expression) Egy kifejezés, ami vagy igaz, vagy hamis.

feltétel (condition) A Boolean kifejezés egy feltételes utasítás fejrészében, amely meghatározza, hogy melyik ág legyen végrehajtva.

feltételes utasítás (conditional statement) Egy utasítás, amely befolyásolja a programvezérlést egy feltétel révén. Pythonban az ehhez használt kulcsszavak: `if`, `elif` és `else`.

igazságtábla (truth table) Logikai értékek tömör táblázata, amely leírja egy logikai operátor szemantikáját.

kód csomagolása függvénybe (wrapping code in a function) Gyakran így hívják azt a folyamatot, melynek során utasítások sorozatát egy fejrésszel látjuk el, hogy egy függvényt kapjunk. Ez nagyon hasznos, mert többszörösen tudjuk használni az utasításokat. Azért is fontos, mert megengedi a programozónak, hogy kifejezze mentális blokkjait és hogy hogyan darabolja fel a problémát kisebb részekre.

láncolt feltételes utasítás (chained conditional) Feltételes elágazás több, mint két lehetséges program-végrehajtási úttal. Pythonban a láncolt feltételes utasítás `if ... elif ... else` szerkezetként írható le.

logikai érték (logical value) A Boolean érték másik megnevezése.

logikai kifejezés (logical expression) Lásd: Boolean kifejezés.

logikai operátor (logical operator) Boolean kifejezéseket összekötő operátorok egyike: `and`, `or` és `not`.

összehasonlító operátor (comparison operator) Két érték összehasonlítására használt hat operátor egyike: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

prompt (prompt) Egy vizuális megoldás, ami azt mondja a felhasználónak, hogy a rendszer készen áll bemenet fogadására.

típuskonverzió (type conversion) Egy explicit függvény, amely vesz egy adott típusú értéket és előállítja a megfelelő másik típusú értéket.

törzs (body) Utasításblokk az összetett utasításokban a fejrész után.

5.14. Feladatok

1. Tételezzük fel, hogy a hét napjai hétfőtől vasárnapig be vannak számozva: 0,1,2,3,4,5,6. Írj egy függvényt, amely megkapja egy nap számát, és visszatér annak nevével!
2. Elmész egy gyönyörű nyaralásra (talán börtönbe, ha nem kedveled a mókás feladatokat) és a 2-es számú napon (tehát szerdán) indulsz. 137 ott alvás után térsz haza. Írj egy programot, amely megkérdezi, hogy hányas számú napon indultál és hány napig voltál távol, majd megmondja annak a napnak a nevét, amikor hazatérsz.
3. Add meg a logikai ellentétét ezeknek a kifejezéseknek!
 - (a) `a > b`
 - (b) `a >= b`
 - (c) `a >= 18 and nap == 3`
 - (d) `a >= 18 and nap != 3`

4. Mi az értéke ezeknek a kifejezéseknek?

- (a) `3 == 3`
- (b) `3 != 3`
- (c) `3 >= 4`
- (d) `not (3 < 4)`

5. Egészítsd ki ezt az igazságtáblát:

p	q	r	(not (p and q)) or r
F	F	F	?
F	F	T	?
F	T	F	?
F	T	T	?
T	F	F	?
T	F	T	?
T	T	F	?
T	T	T	?

6. Írj egy függvényt, amely kap egy vizsgapontszámot és visszaadja az érdemjegyed nevét – az alábbi séma szerint:

Pont	Jegy
<code>>= 90</code>	jeles
<code>[80-90)</code>	jó
<code>[70-80)</code>	közepes
<code>[60-70)</code>	elégséges
<code>< 60</code>	elégtelen

A szögletes- és kerek zárójelek zárt és nyílt intervallumot jelölnek. A zárt intervallum tartalmazza a számot, a nyílt nem. Így az 59.99999 elégtelent jelent, de a 60.0 már elégséges.

Teszteld a függvényed azzal, hogy kiíratod az összes jegyet az alábbi sorozat elemei (pontszámai) esetén:

`xs = [83, 75, 74.9, 70, 69.9, 65, 60, 59.9, 55, 50, 49.9, 45, 44.9, 40, 39.9, 2, 0]`

- 7. Módosítsd a teknőcös oszlopdiagram rajzoló programot, hogy az oszlopok közti résben a toll fel legyen emelve.
- 8. Módosítsd a teknőcös oszlopdiagram rajzoló programot, hogy a 200 és annál nagyobb értékű oszlopok kitöltése piros legyen, amelyek értéke a `[100 és 200)` intervallumban vannak, legyenek sárgák és a 100 alattiak zöldek.
- 9. A teknőcös oszlopdiagram rajzoló programban mit gondolsz, mi történik, ha egy vagy több érték a listán negatív? Próbáld ki! Változtasd meg a programot úgy, hogy a negatív értékű oszlopok felirata az oszlop alá essen.
- 10. Írj egy `atfogo` függvényt, amely megkapja egy derékszögű háromszög két befogójának a hosszát és visszaadja az átfogó hosszát! (Segítség: az `x ** 0.5` a négyzetgyököt adja vissza.)
- 11. Írj egy `derekszogu_e` függvényt, amely megkapja egy háromszög három oldalának a hosszát és meghatározza, hogy derékszögű háromszögről van-e szó! Tételezzük fel, hogy a harmadik megadott oldal mindig a leghosszabb. A függvény `True` értékkel térjen vissza, ha a háromszög derékszögű, `False` értékkel különben!

Segítség: A lebegőpontos aritmetika (azaz a valós számok használata) nem mindig pontos, így nem biztonságos a valós számok egyenlőségével való tesztelés. Ha a jó programozó azt akarja tudni, hogy `x` értéke egyenlő-e vagy nagyon közeli-e `y` értékéhez, valószínűleg ezt a kódot írják:

```
if abs(x-y) < 0.000001:      # Ha x megközelítően egyenlő y-nal
    ...
```

12. Egészítsd ki a fenti programot úgy, hogy tetszőleges sorrendű adatok megadása esetén is működjön!
13. Ha kíváncsivá tett, hogy a lebegőpontos számok aritmetikája miért pontatlan néha, akkor egy darab papíron oszd el a 10-et 3-mal és írd le a decimális eredményt! Azt találod, hogy az osztás sohasem fejeződik be és végtelen hosszú papírra lesz szükséged. A számítógépek memóriájában a számok *ábrázolásának* ugyanez a problémája: a memória véges és lesznek helyiértékek, amelyek eldobásra kerülnek. Így egy kis pontatlanság kerül a dologba. Próbáld ki ezt a szkriptet:

```
1  import math
2  a = math.sqrt(2.0)
3  print(a, a*a)
4  print(a*a == 2.0)
```

6. fejezet

Produktív függvények

6.1. Visszatérési érték

A korábban látott beépített függvények közül az `abs`, `pow`, `int`, `max`, és a `range` is előállított valamilyen eredményt. Ha meghívjuk ezek egyikét, akkor a függvény egy értéket fog generálni, amelyet rendszerint egy változóhoz rendelünk hozzá, vagy egy kifejezésbe építünk be.

```
1 legnagyobb = max(3, 7, 2, 5)
2 x = abs(3 - 11) + 10
```

Írtunk már egy saját függvényt is, mely kamatos kamatot számított.

Ebben a fejezetben még több visszatérési értékkel rendelkező függvényt fogunk készíteni. Jobb név híján *produktív függvényeknek* nevezzük majd ezeket.

Az első példánk egy `terulet` függvény, mely egy adott sugarú kör területét határozza meg:

```
1 def terulet(sugar):
2     b = 3.14159 * sugar**2
3     return b
```

A `return` utasítást már láttuk. Egy produktív függvényben a `return` után mindig áll egy **visszatérési érték**. Az utasítás jelentése: értékeld ki a `return` után álló kifejezést, és a kapott értéket azonnal add vissza a függvény eredményeként. A kifejezés tetszőleges bonyolultságú lehet, akár az alábbi formában is megadhattuk volna a függvényt:

```
1 def terulet(sugar):
2     return 3.14159 * sugar * sugar
```

Azonban a `b`-hez hasonló **ideiglenes változók** gyakran egyszerűbbé teszik a hibakeresés folyamatát.

Alkalmanként több `return` utasítást is szerepeltetünk a függvényekben, például egy elágaztató utasítás különböző ágaiba írva.

A beépített `abs` függvénnyel már találkoztunk, most nézzük meg, hogyan írhatjuk meg a sajátunkat:

```
1 def abszolut_ertek(x):
2     if x < 0:
3         return -x
4     else:
5         return x
```

Egy másik lehetőség a fenti függvény elkészítésére, ha kihagyjuk az `else` kulcsszót és az `if` utasítást egy `return` követi.

```
1 def abszolut_ertek(x):
2     if x < 0:
3         return -x
4     return x
```

Gondold át ezt a változatot, győződj meg róla, hogy ugyanúgy működik, mint az első.

Azon kódrészleteket, amelyek egy `return` után, vele azonos szinten állnak, vagy bárhol máshol, ahová a vezérlés soha nem kerül át, **halott kódnak** vagy **elérhetetlen kódnak** nevezzük.

Egy produktív függvényt érdemes úgy megírni, hogy minden lehetséges útvonalon legyen egy `return` utasítás. Az `abszolut_ertek` függvény alábbi változata ezt nem teszi meg:

```
1 def rossz_abszolut_ertek(x):
2     if x < 0:
3         return -x
4     elif x > 0:
5         return x
6
7     #a függvény meghívása és eredményének megjelenítése
8     print(rossz_abszolut_ertek(0))
```

Azért hibás ez a változat, mert ha az `x` a 0 értéket veszi fel, akkor egyik ágba álló feltétel sem teljesül, és a függvény `return` utasítás érintése nélkül ér véget. A szkriptet futtatva láthatod, hogy ebben az esetben a visszaadott érték **None**.

Amennyiben nincs más visszaadott érték, akkor minden Python függvény `None`-nal tér vissza.

A `return` utasítás `for` cikluson belül is szerepelhet, a vezérlés ebben az esetben is azonnal visszatér a hívóhoz. Tegyük fel, hogy egy olyan függvényt kell írunk, amely megkeresi és visszaadja a paraméterként kapott szólista első, két betűből álló szavát. Ha nincs ilyen szó, akkor üres sztringet kell visszaadnia.

```
1 def ketbetus_szo_keresese(szolista):
2     for szo in szolista:
3         if len(szo) == 2:
4             return szo
5     return ""
6
7     #a függvény tesztelése
8     print(ketbetus_szo_keresese(["ez", "egy", "halott", "papagáj"]))
9     print(ketbetus_szo_keresese(["szeretem", "a", "sajtot"]))
```

Egyesével hajtsd végre a sorokat. Győződj meg arról, hogy az általunk megadott első tesztenél a függvény nem nézi végig a listát, hiszen már az első eleménél visszatér az ott álló szóval. A második esetben a kimenet üres sztring lesz.

6.2. Programfejlesztés

Ezen a ponton már elvárható, hogy egy egyszerű függvény kódját látva meg tudd mondani, hogy az mire szolgál. Ha a feladatokat is elkészítetted, akkor néhány rövid függvényt is írtál már. A nagyobb, bonyolultabb függvények készítése során azonban több nehézségbe fogsz ütközni, különösen a futási idejű és a szemantikai hibákkal gyűlhet meg a bajod.

Az egyre összetettebb programok készítéséhez az **inkrementális fejlesztést** javasoljuk. A módszer igyekszik a hosszú hibakeresési fázisokat kiküszöbölni úgy, hogy a programhoz egyszerre mindig csak kevés új kódrészletet ad hozzá,

így a tesztelés is kevés sort érint.

Tegyük fel, hogy két pont közötti távolságot akarjuk megtudni. A pontok (x_1, y_1) , (x_2, y_2) formában adottak. A Pitagorasz-tétel alapján a távolság:

$$tavolsag = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Az első lépés annak átgondolása, hogy a `tavolsag` függvénynek hogyan kellene Pythonban kinéznie, tehát, milyen bemenetre (paraméterekre) van szükség és mi lesz a kimenet (a visszatérési érték)?

Ebben az esetben a bemenő adat a két pont, melyet négy paraméterrel reprezentálunk. A visszatérési érték egy valós szám lesz, a távolság.

Most már írhatunk egy olyan függvény vázat, amely az eddigi megállapításainkat rögzíti:

```
1 def tavolsag(x1, y1, x2, y2):  
2     return 0.0
```

Ez a változat nyilvánvalóan nem számolja még ki a távolságot, hiszen mindig nullával tér vissza. De szintaktikailag helyes és futtatható, tehát tesztelhetjük, mielőtt bonyolítani kezdenénk.

A teszteléshez egészítsük ki egy olyan sorral, mely meghívja függvényt, és megjeleníti a kapott eredményt:

```
1 def tavolsag(x1, y1, x2, y2):  
2     return 0.0  
3  
4 #teszt  
5 print(tavolsag(1, 2, 4, 6))
```

A teszteléshez úgy választottuk meg az értékeket, hogy a vízszintes távolság 3, a függőleges 4 legyen. Az eredmény 5 (egy olyan háromszög átfogója, melynek oldalhosszúságai: 3, 4, 5). Amikor tesztelünk egy függvényt, jól jöhet a helyes válasz ismerete. Most még persze 0.0-at fogunk kapni.

Azt tehát már megállapítottuk, hogy a függvény szintaktikailag helyes, kezdhetünk új sorokat adni hozzá. Minden egyes változtatás után, újra futtatjuk majd a programot. Ha hiba lépne fel, akkor egyértelmű hol van: az utoljára hozzáadott sorokban.

Logikailag a számítás első lépése az $x_2 - x_1$ és az $y_2 - y_1$ különbségek meghatározása. Az értékeket két ideiglenes változóba (`dx` és `dy`) mentjük.

```
1 def tavolsag(x1, y1, x2, y2):  
2     dx = x2 - x1  
3     dy = y2 - y1  
4     return 0.0  
5  
6 #a függvény meghívása, és eredményének megjelenítése  
7 print(tavolsag(1, 2, 4, 6))
```

Ha a korábban már látott argumentumokkal hívjuk a függvényt, a `dx` változó a 3-as, a `dy` változó a 4-es értéket fogja tartalmazni, mire a vezérlés a `return` utasításra kerül. A **PyCharm**ban könnyen ellenőrizheted is, csak tégy egy töréspontot a `return` sorába, és indítsd el nyomkövető módban (Debug) a programot. Ellenőrizd, hogy helyes paramétereket kap-e a függvény, és jók-e a számítások! Ha netán félresiklott valami, akkor is elegendő néhány sort átnézned.

A következő lépésben számoljuk ki `dx` és `dy` négyzetösszegét:

```
1 def tavolsag(x1, y1, x2, y2):  
2     dx = x2 - x1  
3     dy = y2 - y1
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4     negyzetosszeg = dx*dx + dy*dy
5     return 0.0
6
7     #teszt
8     print(tavolsag(1, 2, 4, 6))
```

Ebben az állapotában is futtatható a program, ellenőrizhető a `negyzetosszeg` változó értéke (25-nek kellene lennie).

Végül határozzuk meg a gyököt a `negyzetosszeg` változó tört hatványra (0.5) való emelésével, és adjuk vissza az eredményt:

```
1     def tavolsag(x1, y1, x2, y2):
2         dx = x2 - x1
3         dy = y2 - y1
4         negyzetosszeg = dx*dx + dy*dy
5         eredmeny = negyzetosszeg**0.5
6         return eredmeny
7
8     #teszt
9     print(tavolsag(1, 2, 4, 6))
```

Ha jól működik, vagyis az 5.0 értéket kaptad, akkor készen is vagy. Ha nem, akkor érdemes megnézned az `eredmeny` változó értékét a `return` előtt.

Kezdetben egyszerre csak egy vagy két sort adj a kódhoz. Ha már gyakorlottabb leszel, könnyen azon kaphatod magad, hogy nagyobb összetartozó egységeket írsz és tesztelsz. Akárhogy is, ha lépésről lépésre futtatod a programot és ellenőrzöd, hogy minden az elvárás szerint alakul-e, azzal jelentősen csökkentheted a hibakeresésre fordítandó időt. Minél jobban programozol már, annál nagyobb és nagyobb egységekkel érdemes próbálkoznod. Olyan ez, mint amikor megtanultuk olvasni a betűket, szótagokat, szavakat, kifejezéseket, mondatokat, bekezdéseket, stb., vagy ahogyan a kottával ismerkedtünk, az egyes hangoktól az akkordokig, ütemig, frázisokig, és így tovább.

Az alábbiakat a legfontosabb szem előtt tartanod:

1. Egy működő program vázzal kezdj, és csak kis lépésekben bővítsd. Ha bármely ponton hiba lépne fel, pontosan tudni fogod hol a gubanc.
2. Használj ideiglenes változókat a közbenső értékek tárolására, így könnyen ellenőrizheted majd a számításokat.
3. Ha végre működik a program, dőlj hátra, pihenj, és játszadozz egy kicsit a különböző lehetőségekkel. (Egy érdekes kutatás a „játékosságot” a hatékonyabb tanulással, az ismeretanyag jobb megértésével, nagyobb élvezettel és a pozitívabb jövőképpel hozta kapcsolatba, szóval szánj egy kis időt a játszadozásra!) Például összevonhatsz egy-két utasítást egyetlen összetett kifejezésbe, átnevezhetsz változókat, vagy kipróbálhatod sikerül-e lerövidíteni a függvényt. Jó irány lehet, ha azt tűzöd ki célul, hogy a programodat a lehető legkönnyebben megértse mások is.

Itt láthatjuk a `tavolsag` egy másik változatát, amely egy a `math` modulban lévő függvényt használ a gyökvonáshoz (hamarosan tanulni fogunk a modulokról is). Melyik tetszik jobban? Melyik áll közelebb a kiindulópontként használt Pitagorasz-tételt leíró képlethez?

```
1     import math
2
3     def tavolsag(x1, y1, x2, y2):
4         return math.sqrt( (x2-x1)**2 + (y2-y1)**2 )
5
6
7     #teszt
8     print(tavolsag(1, 2, 4, 6))
```


A kimenet ezúttal is 5.0.

6.3. Nyomkövetés a `print` utasítással

Egy másik kiváló módszer a nyomkövetésre és hibakeresésre (a változók állapotának lépésenkénti végrehajtás közben történő ellenőrzése mellett), ha a program jól megválasztott pontjaihoz `print` függvényeket szúrunk be. A kimenetet vizsgálva megállapítható, hogy az algoritmus az elvárásnak megfelelően működik-e. Az alábbiakat azonban tartsd észben:

- Már a nyomkövetés előtt ismerned kell a probléma megoldását, és tudnod kell mi fog történni. Old meg a problémát papíron (esetleg készíts folyamatábrát az egyes lépések leírására) *mielőtt* a programírással foglalkoznál. A programozás nem oldja meg a problémát, csak *automatizálja* azokat lépéseket, amelyeket egyébként kézzel hajtánál végre. Első lépésként feltétlenül legyen egy papírra vetett, helyes megoldásod. Utána írd meg a programot, amely automatizálja a már kigondolt lépéseket.
- Ne írd **csevegő** függvényeket. Azokat a produktív függvényeket nevezzük így ebben a könyvben, amelyek az elsődleges feladatukon túl bemenetet kérnek a felhasználótól, vagy értékeket jelenítenek meg, pedig sokkal hasznosabb lenne, ha „befognák a szájuk” és csendben dolgoznának.

Gondoljunk a korábban látott `range`, `max` vagy `abs` függvényekre. Egyik sem lenne hasznos építőelem más programoknak, ha a felhasználótól kérnék a bemenetet, vagy a képernyőn jelenítenék meg az eredményt a feladatukat elvégzése közben.

Kerüld a `print` és az `input` függvényeket a produktív függvényeken belül, *kivéve, ha a függvény elsődleges célja az adatbekérés vagy a megjelenítés*. Az egyetlen kivétel a szabály alól az lehet, amikor ideiglenesen teletűzdeled `print` függvényekkel a kódot, hogy segítsen a hibakeresésben, a program futása alatt történő események megértésében. Az ilyen célból hozzáadott sorok természetesen eltávolítandók, amikor már működik a vizsgált rész.

6.4. Függvények egymásba ágyazása

Ahogy arra számíthattunk, a függvények belsejéből is lehet hívni függvényeket. Ezt hívjuk **egymásba ágyazásnak**.

Példaként egy olyan függvényt fogunk írni, amely egy kör területét határozza meg, a kör középpontja és egy a körvonalra eső pontja alapján.

Tegyük fel, hogy a kör középpontját a `kx` és a `ky`, a körvonalra eső pontot pedig az `x` és `y` változók tárolják. Az első lépés a sugár hosszának, vagyis a két pont közötti távolságnak a kiszámítása. Erre szerencsére már írtunk korábban egy függvényt (`tavolsag`), így most egyszerűen csak meg kell hívnunk:

```
1 sugar = tavolsag(kx, ky, x, y)
```

A második lépés a kör területének meghatározása és visszaadása.

Ismét használhatjuk a korábban megírt függvényeink egyikét:

```
1 eredmeny = terület(sugar)
2 return eredmeny
```

A két részt egy függvénybe gyúrva az alábbiit kapjuk:

```
1 def terület2(kx, ky, x, y):
2     sugar = tavolsag(kx, ky, x, y)
3     eredmeny = terület(sugar)
4     return eredmeny
```

A `terulet2` nevet azért kapta, hogy megkülönböztessük a korábban definiált `terulet` függvénytől.

Az `sugar` és az `eredmeny` ideiglenes változók. Jól jönnek a fejlesztés alatt és a nyomkövetésnél, amikor a sorokat egyesével végrehajtva vizsgáljuk, mi folyik a háttérben; de ha a program már működik, a függvények egymásba ágyazásával tömörebbé tehetjük a kódot:

```
1 def terulet2(kx, ky, x, y):
2     return terulet(tavolsag(kx, ky, x, y))
```

6.5. Logikai függvények

A Boolean értékekkel visszatérő függvények felettébb jól alkalmazhatók a bonyolult vizsgálatok elrejtésére. Lássunk egy példát:

```
1 def oszthato_e(x, y):
2     """ Annak vizsgálata, hogy x osztható-e y-nal. """
3     if x % y == 0:
4         return True
5     else:
6         return False
```

A **logikai függvények**, vagy más néven **Boolean függvények**, gyakran kapnak olyan nevet, mely egy eldöntendő kérdésre hasonlít. (Az ilyen függvények angol neve általában az `is_` taggal indul, a fenti függvényt például `is_divisible`-nek nevezhetnénk.) Az `oszthato_e` függvény `True` vagy `False` értékkel tér vissza, megadván, hogy az `x` osztható-e az `y`-nal vagy sem.

Tömörebbé tehetjük a függvényt, ha kihasználjuk, hogy az `if` utasításban szereplő feltétel önmagában is egy Boolean kifejezés. A feltételes utasítást elhagyva, egyből a kifejezéssel is visszatérhetünk:

```
1 def oszthato_e(x, y):
2     return x % y == 0
```

A teszteléshez fűzzünk két olyan sort a szkript végére, amelyek meghívják az `oszthato_e` függvényt, és megjelenítik a kapott eredményt. Az első esetben hamis (`False`), a második esetben igaz (`True`) értéket kell visszakapnunk.

```
1 def oszthato_e(x, y):
2     return x % y == 0
3
4 #teszt
5 print(oszthato_e(6, 4))
6 print(oszthato_e(6, 3))
```

A logikai függvényeket gyakran alkalmazzuk feltételes utasításokban:

```
1 if oszthato_e(x, y):
2     ... # Csinálj valamit ...
3 else:
4     ... # Csinálj valami mást ...
```

Csábító lehet az alábbihoz hasonló kifejezést írni, de ez egy felesleges összehasonlítást eredményez.

```
1 if oszthato_e(x, y) == True:
```

6.6. Stílusos programozás

Az olvashatóság igen fontos a programozók számára, hiszen a gyakorlatban a programokat sokkal gyakrabban olvassák és módosítják, mint ahányszor megírják őket. Persze, ahogy a legtöbb szabállyal is tesszük, „Az írd könnyen olvasható programot!” íratlan szabályát is áthágjuk néha. A könyvben szereplő példaprogramok nagy része megfelel a *Python Enhancement Proposal* 8-nak (PEP 8), amely egy a Python közösség által fejlesztett programozói stíluskalauz.

Részletesebben is ki fogunk térni a megfelelő programozói stílusra, amikor már összetettebbek lesznek a programjaink, addig is álljon itt néhány pont útmutatásul:

- Használj 4 szóközt az indentálásnál (tabulátorok helyett).
- Minden sor legfeljebb 78 karakterből álljon.
- Az osztályoknak (hamarosan sorra kerülnek) adj nagy kezdőbetűs neveket. Ha több tagból áll össze a név, akkor írd egybe őket, és minden egyes tagot kezdj nagy kezdőbetűvel (TeveHat). A változók és a függvények azonosítására szolgáló neveket írd csupa kisbetűvel, ha több tag is van, akkor aláhúzással válaszd el őket (kisbetuk_alahuzasjellel_elvalasztva).
- Az import utasításokat a fájl legelején helyezd el.
- A függvénydefiníciók egymás után álljanak a szkriptben.
- Használj dokumentációs sztringeket a függvények dokumentálásához.
- Két üres sorral válaszd el egymástól a függvénydefiníciókat.
- A legfelső szintű utasításokat, beleértve a függvényhívásokat is, egy helyen, a program alján szerepeltesd.

Bizonyos pontok betartására a *PyCharm* is figyelmeztet majd.

6.7. Egységteszt

A szoftverfejlesztés során jó gyakorlat és bevett szokás **egységteszteket** szűrni a programba. Az egységteszt a különböző kódrészletek, mint például a függvények, megfelelő működésének automatikus ellenőrzését teszi lehetővé. Ha később módosítanunk kell egy függvény implementációját (a függvény törzsében lévő utasításokat), akkor az egységteszt segítségével gyorsan megállapíthatjuk, hogy még mindig eleget tesz-e az elvárásainknak a függvény.

Néhány éve a vállalatok még csak a programkódot és a dokumentációt tekintették igazi értéknek, ma már a fejlesztésre szánt keret jelentős részét a tesztesetek készítésére és tárolására fordítják.

Az egységtesztek a programozókat arra kényszerítik, hogy alaposan átgondolják a különböző eshetőségeket, amelyeket a függvényeiknek kezelniük kell. A teszteseteket csak egyszer szükséges beírunk a szkriptbe, nincs szükség arra, hogy újra és újra megadjuk ugyanazokat a tesztadatokat a programfejlesztés során.

A programba épített plusz kódrészleteket, amelyek kimondottan a nyomkövetést és a tesztelést hivatottak egyszerűbbé tenni **scaffolding** megnevezéssel szokás illetni.

Az egy kódrészlet ellenőrzésére szolgáló tesztesetek **tesztkészletet** alkotnak.

Pythonban több különböző mód is kínálkozik egységtesztek megvalósítására, azonban a Python közösség által javasolt módszerekkel ezen a ponton még nem foglalkozunk. Két saját készítésű függvénnyel kezdünk bele egy egységteszt megvalósításába.

Kezdjük a munkát a fejezet korábbi részében megírt `abszolut_ertek` függvénnyel. Több verziót is készítettük, az utolsó pedig hibás volt. Vajon megtalálják-e majd a tesztek a hibát?

Először a tesztek tervezését meg. Tudni szeretnénk, hogy helyes értéket ad-e vissza a függvény, amikor az argumentum negatív, amikor az argumentum pozitív, vagy ha nulla. A tesztesetek tervezésekor mindig alaposan át kell gondolnunk a szélsőséges eseteket. Az `abszolut_ertek` függvényben a 0 szélsőséges esetnek számít, hiszen a

függvény viselkedése ezen a ponton megváltozik, és ahogy azt a fejezet elején láthattuk, a programozók itt könnyen hibázhatnak! Érdemes tehát a nullát felvenni a tesztkészletünkbe.

Most írjunk egy segédfüggvényt, amely egy teszt eredményét jeleníti majd meg. Boolean argumentumot vár majd inputként, és egy képernyőre írt üzenetben értesít minket arról, hogy sikeres vagy sikertelen volt-e a teszt. A függvénytörzs első sora (a dokumentációs sztring után) „misztikus módon” meghatározza annak a sornak a szkripten belüli sorszámát, ahonnan a függvényt meghívják. A sorszám kiírásakor gyakorlatilag egy teszteset szkriptbeli helyét jelenítjük meg, így megtudhatjuk, hogy mely tesztesetek voltak sikeresek és melyek nem.

```
1 import sys
2
3 def teszt(sikeres_teszt):
4     """ Egy teszt eredményének megjelenítése. """
5     sorszam = sys._getframe(1).f_lineno # A hívó sorának száma
6     if sikeres_teszt:
7         msg = "A(z) {0}. sorban álló teszt sikeres.".format(sorszam)
8     else:
9         msg = ("A(z) {0}. sorban álló teszt SIKERTELEN.".format(sorszam))
10    print(msg)
```

Tartalmaz a függvény egy kicsit cseles, a `format` metódust használó sztringformázást is, de ezt most felejtjük el egy pillanatra, majd egy későbbi fejezetben részletesen megnézzük. A lényeg az, hogy a `teszt` függvény felhasználásával felépíthetjük a tesztkészletet:

```
1 def tesztkészlet():
2     """ Az ehhez a modulhoz (fájlhoz) tartozó tesztkészlet futtatása. """
3
4     teszt(abszolot_ertek(17) == 17)
5     teszt(abszolot_ertek(-17) == 17)
6     teszt(abszolot_ertek(0) == 0)
7     teszt(abszolot_ertek(3.14) == 3.14)
8     teszt(abszolot_ertek(-3.14) == 3.14)
9
10 tesztkészlet()
```

Amint az látható, öt esetet vettünk fel a tesztkészletbe, melyekkel az `abszolot_ertek` függvény első két változatának valamelyikét, tehát az egyik helyes változatot, ellenőrizhetjük. A kimenet az alábbihoz hasonló lesz (ha a szkriptedben más sorokban állnak a tesztesetek, akkor a sorszámok természetesen eltérnek majd):

```
A(z) 22. sorban álló teszt sikeres.
A(z) 23. sorban álló teszt sikeres.
A(z) 24. sorban álló teszt sikeres.
A(z) 25. sorban álló teszt sikeres.
A(z) 26. sorban álló teszt sikeres.
```

Rontsuk most el a függvényt valahogy így:

```
1 def abszolot_ertek(n): # Hibás változat
2     """ Az n abszolút értékének kiszámítása. """
3     if n < 0:
4         return 1
5     elif n > 0:
6         return n
```

Találsz-e legalább két hibát a kódban? A tesztkészletünk talált! A szkriptünk az alábbi kimenetet adta:

```
A(z) 23. sorban álló teszt sikeres.
A(z) 24. sorban álló teszt SIKERTELEN.
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
A(z) 25. sorban álló teszt SIKERTELEN.  
A(z) 26. sorban álló teszt sikeres.  
A(z) 27. sorban álló teszt SIKERTELEN.
```

Láthatjuk, három példánk is van a *sikertelen tesztre*.

A Pythonban van egy beépített utasítás is, az **assert**, amely majdnem ugyanazt csinálja, mint a mi **teszt** függvényünk. Az eltérés az, hogy az assert alkalmazása esetén a program leáll az első olyan esetenél, amikor az elvárt feltétel nem teljesül. Érdemes többet megtudnod róla, és akkor alkalmazhatod majd a saját **teszt** függvényünk helyett.

6.8. Szójegyzék

Boolean függvény (Boolean function) Egy olyan függvény, mely Boolean értéket ad vissza. A `bool` típuson belül csak kétféle érték van: `True` (igaz) és `False` (hamis).

csevegő függvény (chatterbox function) Az olyan függvények könyvbeli megnevezése, amelyek kapcsolatba lépnek a felhasználóval (`input` és `print` utasításokat használva), pedig nem kellene nekik. A leghasznosabb függvények általában azok, amelyek egyszerűen csak előállítják az eredményt a bemenetként kapott argumentumokból. (Hivatalosabb elnevezése: mellékhatással rendelkező függvény.)

egységteszt (unit testing) Egy olyan automatikus folyamat, amely az egyes programegységek validálását végzi, vagyis ellenőrzi, hogy megfelelő-e a működésük. Az egységteszthez használt tesztkészletek rendkívül hasznosak a kód módosításánál és bővítésénél. Egyfajta biztonsági kötélként funkcionálnak, nehogy visszaessünk, egy korábban már működő kódrészletbe hibát téve. A *regressziós* tesztelés megnevezés is ezt a Nem akarunk hátralepni! elvet tükrözi.

elérhetetlen kód (unreachable code) Egy olyan kódrészlet, mely soha nem kerülhet végrehajtásra. Gyakran fordul elő a `return` utasítás után.

függvények egymásba ágyazása (composition of functions) Egy függvény meghívása egy másik függvény törzsén belül, vagy egy függvény (visszatérési értékének) argumentumként való használata egy másik függvény meghívásához.

halott kód (dead code) Egy másik elnevezés az elérhetetlen kódrészletekre.

ideiglenes változó (temporary variable) Az összetett számítások közbenső lépéseiben előálló értékek tárolására használt változók.

inkrementális fejlesztés (incremental development) Egy a hibakeresést egyszerűsítő programfejlesztési módszer. A lényege, hogy egyszerre mindig csak kevés kóddal bővítjük a programot, kevés kódot tesztelünk.

logikai függvény (Boolean function) A Boolean függvények másik megnevezése.

None Egy speciális Python érték. Például ezt az értéket adja vissza a Python, ha egy függvényen belül nem kerül végrehajtásra olyan `return` utasítás, amely mögött valamilyen kifejezés áll.

produktív függvény (fruitful function) Olyan függvény, mely ad vissza értéket (az alapértelmezett visszatérési érték, a `None`, helyett).

scaffolding Olyan kódrészletek, amelyek a programfejlesztés során segítik a fejlesztést és a nyomkövetést, mint például a jelen fejezetben szereplő egységteszt.

tesztkészlet (test suite) Egy meghatározott kódrészlet ellenőrzésére szolgáló tesztesetek gyűjteménye.

visszatérési érték (return value) A függvényhívás eredményeként előálló érték.

6.9. Feladatok

Az összes feladatot egyetlen szkriptben írd meg, és a szkripthez add hozzá a korábban látott `teszt` és `tesztkeszlet` függvényeket is. Amint megoldasz egy feladatot, adj új, a feladatnak megfelelő teszteseteket a tesztkészletedhez.

Az összes feladat megoldása után ellenőrizd, hogy minden teszt sikeres-e.

1. A négy tájegységet rövidítse: „K”, „Ny”, „É”, „D”. Írj egy `fordulj_orajarasi_iranyba` függvényt, amely egy tájegységet leíró karakter rövidítését várja, és visszaadja az órajárási irányban nézve szomszédos égtáj rövidítését. Itt van néhány teszt, melyre működni kell a függvényednek:

```
teszt(fordulj_orajarasi_iranyba("É") == "K")
teszt(fordulj_orajarasi_iranyba("Ny") == "É")
```

Talán most azt kérdezed magadban: „Mi legyen, ha az argumentum nem is egy égtáj rövidítése?” Minden más esetben `None` értékkel térjen vissza a függvény:

```
teszt(fordulj_orajarasi_iranyba(42) == None)
teszt(fordulj_orajarasi_iranyba("ostobaság") == None)
```

2. Írj egy `nap_nev` függvényt, amely a `[0, 6]` tartományba eső egész számot vár paraméterként, és visszaadja az adott sorszámu nap nevét. A 0. nap a hétfő. Még egyszer leírjuk, ha nem várt érték érkezik, akkor `None` értékkel térj vissza. Néhány teszt, melyen át kell mennie a függvényednek:

```
teszt(nap_nev(3) == "csütörtök")
teszt(nap_nev(6) == "vasárnap")
teszt(nap_nev(42) == None)
```

3. Írd meg az előző függvény fordítottját, amely egy nap neve alapján adja meg annak sorszámát:

```
teszt(nap_sorszam("péntek") == 4)
teszt(nap_sorszam("hétfő") == 0)
teszt(nap_sorszam(nap_nev(3)) == 3)
teszt(nap_nev(nap_sorszam("csütörtök"))) == "csütörtök")
```

Még egyszer: ha a függvény érvénytelen argumentumot kap, akkor `None` értéket adj vissza:

```
teszt(nap_sorszam("Halloween") == None)
```

4. Írj egy függvényt, amely segít megválaszolni az ehhez hasonló kérdéseket: „Szerda van. 19 nap múlva nyaralni megyek. Milyen napra fog esni?” A függvénynek tehát egy nap nevét és egy „hány nap múlva” értéket vár argumentumként, és egy nap nevét adja vissza:

```
teszt(napok_hozzaadasa("hétfő", 4) == "péntek")
teszt(napok_hozzaadasa("kedd", 0) == "kedd")
teszt(napok_hozzaadasa("kedd", 14) == "kedd")
teszt(napok_hozzaadasa("vasárnap", 100) == "kedd")
```

(Segítség: Érdemes felhasználni az előző két feladatban megírt függvényt a `napok_hozzaadasa` függvény megírásához.)

5. Működik a `napok_hozzaadasa` függvényed negatív értékekre, „hány nappal korábban” kérdésekre is? Például a `-1` a tegnapi napot, a `-7` az egy héttel ezelőttit jelenti:

```
teszt(napok_hozzaadasa("vasárnap", -1) == "szombat")
teszt(napok_hozzaadasa("vasárnap", -7) == "vasárnap")
teszt(napok_hozzaadasa("kedd", -100) == "vasárnap")
```

Ha már működik a függvényed, akkor magyarázd meg miért. Ha még nem, akkor old meg, hogy működjön.

Segítség: Játsszozz néhány esettel a maradékos osztás (%) műveletet használva. (Az előző fejezet elején mutattuk be.) Különösen azt érdemes kiderítened, hogy mi történik, amikor az $x \% 7$ kifejezésben az x negatív.

6. Írj egy `honap_napja` függvényt, mely egy hónap neve alapján megadja, hogy hány nap van a hónapban. (A szökőévekkel ne foglalkozz.):

```
teszt(honap_napja("február") == 28)
teszt(honap_napja("november") == 30)
teszt(honap_napja("december") == 31)
```

Érvénytelen argumentum esetén `None` értéket kell visszaadnia a függvénynek.

7. Írj egy `masodpercre_valtas` függvényt, mely órákat, perceket és másodperceket kap meg argumentumként, és kiszámolja hány másodpercnek felelnek meg összesen. Néhány tesztet:

```
teszt(masodpercre_valtas(2, 30, 10) == 9010)
teszt(masodpercre_valtas(2, 0, 0) == 7200)
teszt(masodpercre_valtas(0, 2, 0) == 120)
teszt(masodpercre_valtas(0, 0, 42) == 42)
teszt(masodpercre_valtas(0, -10, 10) == -590)
```

8. Egészítsd ki a `masodpercre_valtas` függvényt úgy, hogy valós számok érkezése esetén is helyesen működjön. A végeredményt egészre *vágva* (ne kerekítve) add meg:

```
teszt(masodpercre_valtas(2.5, 0, 10.71) == 9010)
teszt(masodpercre_valtas(2.433, 0, 0) == 8758)
```

9. Írj három függvényt, melyek a `masodpercre_valtas` fordítottját valósítják meg:

- (a) `orakra_valtas`: az argumentumként átadott másodperceket órákra váltja. A teljes órák számával tér vissza.
- (b) `percekre_valtas`: az argumentumként átadott másodpercekből leszámítja a teljes órákat, a maradékot pedig percekbe váltja. A teljes percek számával tér vissza.
- (c) `masodpercekre_valtas`: visszatér az argumentumként kapott másodpercekből fennmaradó másodpercekkel.

Feltételezheted, hogy az átadott másodpercek száma egész érték. Néhány teszt:

```
teszt(orakra_valtas(9010) == 2)
teszt(percekre_valtas(9010) == 30)
teszt(masodpercekre_valtas(9010) == 10)
```

Nem lesz mindig nyilvánvaló, hogy mire van szükség ...

Az előző feladat harmadik része félreérthetően és homályosan van megfogalmazva, a tesztesetek azonban tisztázzák, hogy mire is van szükség valójában.

Az egységtesztek járulékos haszna, hogy pontosítani, tisztázni tudják a követelményeket. A tesztkészlet összeállítását is tekintsd a problémamegoldás részének. Kérdezd meg magadtól milyen eshetőségekre számíthatsz, és hogy gondoltál-e minden lehetséges forgatókönyvre.

Mivel a könyv címe *Hogyan gondolkodj ...*, talán szeretnél legalább egy utalást arra, hol olvashatsz a gondolkodásról, és olyan szórakoztató elméletekről, mint a *folyékony intelligencia*, amely a problémamegoldás egyik fontos összetevője. Ha tudsz angolul, akkor a <http://psychology.about.com/od/cognitivepsychology/a/fluid-crystal.htm> oldalon elérhető cikket ajánljuk.

A számítástudományok tanuláshoz a folyékony és a kristályosodott intelligencia megfelelő elegye szükséges.

10. Melyik teszt lesz sikertelen és miért?:

```
teszt(3 % 4 == 0)
teszt(3 % 4 == 3)
teszt(3 / 4 == 0)
teszt(3 // 4 == 0)
teszt(3+4 * 2 == 14)
teszt(4-2+2 == 0)
teszt(len("helló, világ!") == 13)
```

11. Írj egy összehasonlítás függvényt, amely 1-et ad vissza, ha $a > b$, 0-t ad vissza, ha $a == b$, és -1-t, ha $a < b$

```
teszt(osszehasonlitas(5, 4) == 1)
teszt(osszehasonlitas(7, 7) == 0)
teszt(osszehasonlitas(2, 3) == -1)
teszt(osszehasonlitas(42, 1) == 1)
```

12. Írj egy `atfogo` nevű függvényt, amely egy derékszögű háromszög két befogójának hossza alapján visszaadja az átfogó hosszát:

```
teszt(atfogo(3, 4) == 5.0)
teszt(atfogo(12, 5) == 13.0)
teszt(atfogo(24, 7) == 25.0)
teszt(atfogo(9, 12) == 15.0)
```

13. Implementáld a `meredekseg(x1, y1, x2, y2)` függvényt, úgy, hogy az $(x1, y1)$ és $(x2, y2)$ pontokon átmenő egyenes meredekségét határozza meg:

```
teszt(meredekseg(5, 3, 4, 2) == 1.0)
teszt(meredekseg(1, 2, 3, 2) == 0.0)
teszt(meredekseg(1, 2, 3, 3) == 0.5)
teszt(meredekseg(2, 4, 1, 2) == 2.0)
teszt(meredekseg(2, 4, 2, 5) == None)
```

Utána használd fel egy új, `meteszespont(x1, y1, x2, y2)` függvényben, amely visszaadja, hogy az $(x1, y1)$, $(x2, y2)$ pontokon átmenő egyenes milyen y értéknél metszi a derékszögű koordináta-rendszer függőleges tengelyét. (Feltételezheted, hogy az $x1$ és $x2$ értéke különböző.):

```
teszt(metszespont(1, 6, 3, 12) == 3.0)
teszt(metszespont(6, 1, 1, 6) == 7.0)
teszt(metszespont(4, 6, 12, 8) == 5.0)
```

14. Írj egy `paros_e(n)` függvényt, amely egy egészet vár argumentumként, és `True`-t ad vissza, ha az argumentum **páros szám**, és `False`-t, ha **páratlan**.

Adj a tesztkészlethez saját teszt eseteket.

15. Most írd meg egy `paratlan_e(n)` függvényt is, amely akkor tér vissza `True` értékkel, ha n páratlan, és akkor `False` értékkel, ha páros. Tesztet is készíts!

Végül, módosítsd úgy a függvényt, hogy az a `paros_e` függvényt használja annak eldöntésére, hogy az argumentum páros-e. Ellenőrizd, hogy még mindig átmegy-e a tesztek a függvény.

16. Készíts egy `tenyezo_e(t, n)` fejlécű függvényt, amely átmegy az alábbi teszteken. (Ne csak a prímtényezőkre adjon vissza igazat a függvényed.):


```
teszt(tenyezo_e(3, 12))
teszt(not tenyezo_e(5, 12))
teszt(tenyezo_e(7, 14))
teszt(not tenyezo_e(7, 15))
teszt(tenyezo_e(1, 15))
teszt(tenyezo_e(15, 15))
teszt(not tenyezo_e(25, 15))
```

Az egységteszt fontos szerepe, hogy „félreérhetetlen” követelmény megadásként viselkedjen. A tesztesetek megadják a választ arra a kérdésre, hogy magát az 1-et és a 15-öt is a 15 tényezőjének tekintsük-e.

17. Írj egy többszorose_e fejlécű függvényt, mely kielégíti az alábbi egységtesztet:

```
teszt(tobbszorose_e(12, 3))
teszt(tobbszorose_e(12, 4))
teszt(not tobbzorose_e(12, 5))
teszt(tobbszorose_e(12, 6))
teszt(not tobbzorose_e(12, 7))
```

Fel tudnád-e használni a tenyezo_e függvényt a tobbszorose_e függvény megírásánál?

18. Írj egy celsiusra_valtas(f) függvényt, mely egy Fahrenheitben megadott értéket Celsius fokra vált át. A függvény a legközelebbi egész értéket adja vissza. (Segítség: Ha a beépített round függvényt szeretnéd használni, próbáld kiírni a round.__doc__ -et a Python konzolban, vagy a függvény nevén állva nyomd le a *Ctrl+Q* billentyűkombinációt. Kísérletezz, ameddig rá nem jössz, hogyan működik.):

```
teszt(celsiusra_valtas(212) == 100)      # A víz forráspontja
teszt(celsiusra_valtas(32) == 0)         # A víz fagyáspontja
teszt(celsiusra_valtas(-40) == -40)      # Ó, micsoda érdekes eset!
teszt(celsiusra_valtas(36) == 2)
teszt(celsiusra_valtas(37) == 3)
teszt(celsiusra_valtas(38) == 3)
teszt(celsiusra_valtas(39) == 4)
```

19. Most tedd az ellenkezőjét: írd egy celsiusra_valtas függvényt, mely egy Celsius-fokban megadott értéket Fahrenheit skálára vált át:

```
teszt(fahrenheitre_valtas(0) == 32)
teszt(fahrenheitre_valtas(100) == 212)
teszt(fahrenheitre_valtas(-40) == -40)
teszt(fahrenheitre_valtas(12) == 54)
teszt(fahrenheitre_valtas(18) == 64)
teszt(fahrenheitre_valtas(-48) == -54)
```

7. fejezet

Iteráció

A számítógépek gyakran használnak automatikusan ismétlődő feladatokat. Az egyedi vagy nagyon hasonló feladatok hiba nélküli ismétlése olyan dolog, amiben a számítógépek nagyon jók, az emberek viszont nem igazán.

Egy utasításhalmaz végrehajtásának ismétlése az **iteráció**. Mivel az iteráció elég hétköznapi dolog, a Python számos nyelvi lehetőséget biztosít ezek egyszerűvé tételéhez. Már láttuk a `for` ciklust a 3. fejezetben. Ezt az iterációformát fogod valószínűleg a legtöbbször használni. Azonban ebben a fejezetben a `while` ciklust fogjuk megnézni – ami egy másik módja az ismétlésnek, amely picit eltérő körülmények között hasznos.

Mielőtt ezt megnéznénk, ismételjünk át néhány ötletet...

7.1. Értékadás

Ahogy már korábban is említettük, szabályos dolog egy változónak többször is értéket adni. Az új értékadás hatására a változó az új értékre fog hivatkozni (és elfelejti a régi értéket).

```
1 hatralevo_ido = 15
2 print(hatralevo_ido)
3 hatralevo_ido = 7
4 print(hatralevo_ido)
```

A program kimenete ez lesz:

```
15
7
```

mert az első kiíratásnál a `hatralevo_ido` változó értéke 15, a másodiknál 7.

Különösen fontos, hogy különbséget tegyünk az értékadás és az egyenlőséget vizsgáló logikai kifejezés között. Mivel a Python az egyenlőségjelet (`=`) használja az értékadásra, könnyen elcsúbulhatunk az értelmezés során azt gondolva, hogy az `a=b` egy logikai teszt. Eltérően a matematikától ez nem az! Emlékezz, a Python egyenlőségvizsgálat operátora `a ==` operátor!

Figyelj arra is, hogy az egyenlőségvizsgálat szimmetrikus, de az értékadás nem. Például, ha `a==7`, akkor `7==a` is fennáll. Azonban az `a=7` egy szabályos utasítás, míg a `7=a` nem az.

Pythonban az értékadás két változót egyenlővé tehet, de mivel a későbbi értékadások megváltoztathatják az egyiket, így nem maradnak azok.

```
1 a = 5
2 b = a      # Miután ezt a sort végrehajtottuk a és b egyenlők lesznek
3 a = 3      # A sor végrehajtása után a és b többé már nem egyenlők
```

A harmadik sor megváltoztatja az `a` értékét, de nem változtatja meg `b` értékét, így többé nem egyenlők. (Néhány programnyelvben az értékadásra más szimbólumot használnak, mint a `<-` vagy az `:=`, így elkerülhetőek a félreértések. A Python a megszokott jelölést választotta és követi, amely a C, C++, Java és C# nyelvekben is megtalálható, szimpla egyenlőségjel (`=`) az értékadáshoz, dupla egyenlőségjel (`==`) az egyenlőségvizsgálathoz.

7.2. A változók frissítése

Amikor egy értékadó utasítás végrehajtott, a kifejezés jobb oldala (vagyis az egyenlőségjel utáni rész) értékelődik ki először. Ez előállítja az értéket. Aztán megtörténik maga az értékadás, tehát a baloldalon lévő változó most már az új értékre hivatkozik.

Az értékadás egyik legközönségesebb formája a frissítés, ahol a változó új értéke függ a régitől. Vonj le 10 forintot az árból, vagy adj hozzá egy pontot az eredményjelzőn lévő értékhez.

```
1 n = 5
2 n = 3 * n + 1
```

A második sor azt jelenti *Vedd az aktuális értékét `n`-nek, szorozd meg hárommal, és adj hozzá egyet!* Szóval a fenti két sor végrehajtása után az `n` a 16 értéket fogja jelenteni.

Ha egy olyan változó értékét próbálsz meg felhasználni, amelynek eddig nem is adtál értéket, akkor hibaüzenetet kapsz:

```
>>> w = x + 1
Traceback (most recent call last):
  File "<input>", line 1, in <modul>
NameError: name 'x' is not defined
```

Mielőtt frissítesz egy változót, **inicializálnod** kell őt valamilyen kezdőértékkel, többnyire egy egyszerű értékadásban:

```
1 gol_szam = 0
2 ...
3 gol_szam = gol_szam + 1
```

A 3. sor – ami aktualizálja a változó értékét azzal, hogy egyet hozzáad – nagyon hétköznapi utasítás. Ezt a változó **inkrementálásának** hívjuk, az egyel való csökkentést pedig **dekrementálásnak**.

7.3. A for ciklus újra

Emlékezz rá, hogy a `for` ciklus egy lista minden elemét feldolgozza. Turnusonként minden egyes elem értékül lesz adva a ciklusváltozónak és a ciklus törzse végre lesz hajtva. Láttunk már ilyet egy korábbi fejezetben:

```
1 for b in ["Misi", "Petra", "Botond", "Jani", "Csilla", "Peti", "Norbi"]:
2     meghivas = "Szia, " + b + "! Kérlek gyere el a bulimba szombaton!"
3     print(meghivas)
```

A lista összes elemén való egyszeri végigfutást a lista **bejárásának** hívjuk.

Írjunk most egy függvényt, amely megszámlálja a lista elemeit. Csináld kézzel először, és próbáld pontosan meghatározni a szükséges lépéseket! Rájössz majd, hogy kell egy „futó összeg”, ami a részösszegeket fogja tárolni, mint

ahogy papíron, fejben vagy számológépen is. Emlékezz, pontosan azért vannak változóink, hogy emlékezzünk dolgokra miközben egyikről a másik lépésre haladunk: így tehát a „futó összeg” azért kell, hogy emlékezzünk néhány értékre. Ezt nulla értékkel kell inicializálnunk, és aztán be kell járni a listát. Minden elem esetén frissíteni fogjuk a részösszeget hozzáadva a következő számot.

```
1 def saját_osszeg(xs):
2     """ Az xs lista elemeinek összegzése és az eredmény visszaadása """
3     futo_osszeg = 0
4     for x in xs:
5         futo_osszeg = futo_osszeg + x
6     return futo_osszeg
7
8 # Adj ilyen teszteket a tesztkészletedhez...
9 teszt(saját_osszeg([1, 2, 3, 4]) == 10)
10 teszt(saját_osszeg([1.25, 2.5, 1.75]) == 5.5)
11 teszt(saját_osszeg([1, -2, 3]) == 2)
12 teszt(saját_osszeg([ ]) == 0)
13 teszt(saját_osszeg(range(11)) == 55) # A 11 nem eleme a listának.
```

7.4. A while utasítás

Itt egy kódtörredék, amely a while utasítás használatát demonstrálja:

```
1 def osszeg_eddig(n):
2     """ Számsorozat összege 1+2+3+...+n """
3     ro = 0
4     e = 1
5     while e <= n:
6         ro = ro + e
7         e = e + 1
8     return ro
9
10 # A tesztkészlethez
11 teszt(osszeg_eddig(4) == 10)
12 teszt(osszeg_eddig(1000) == 500500)
```

A while utasítást majdnem úgy olvashatjuk, mintha egy (angol) mondat lenne. A fenti példa azt jelenti, hogy amíg az e értéke kisebb vagy egyenlő n értékénél, addig ismételd a ciklustörzs végrehajtását. A törzsben minden alkalommal inkrementáld e -t! Amikor e értéke meghaladja az n értékét térj vissza az összegzett értékkel!

Itt a while utasítás végrehajtásának menete formálisan:

- Értékelj ki a feltételt az 5. sorban, megadva az értékét, amely vagy False, vagy True.
- Ha az érték False (hamis feltétel), lépj ki a while ciklusból és folytasd a végrehajtást a következő utasítással!
- Ha az érték True, akkor hajtsd végre az összes utasítást a törzsben (6. és 7. sor) és aztán menj vissza a while utasítás elejére, az 5. sorba.

A ciklus törzsét a while kulcsszó alatti indentált sorok alkotják.

Vedd észre, hogy ha a ciklusfeltétel hamis az első alkalommal is, akkor a ciklus törzse sohasem lesz végrehajtva!

A ciklus törzse egy vagy több változó értékét is megváltoztathatja úgy, hogy végül is a feltétel hamissá válik és a ciklus véget ér. Ellenkező esetben a ciklus örökké ismétlődni fog, azaz **végtelen ciklust** kapunk.

A fenti esetben be tudjuk bizonyítani, hogy a ciklus véget ér, mert tudjuk, hogy n értéke véges és e értéke minden lépésben növekszik, így végül is el fogja érni n értékét. Más esetekben nem ilyen egyszerű a dolgunk, néha lehetetlen

megmondani, hogy valaha véget fog-e érni a ciklus.

Amit észre fogsz venni, az hogy a `while` ciklus több munka neked – programozónak – mint az egyenértékű `for` ciklus. Amikor `while` ciklust használsz, neked magadnak kell kezelned a ciklusváltozót: kezdőértéket adni, ellenőrizni a feltételt, gondoskodni arról, hogy olyan változás történjen a törzsben, aminek hatására a ciklus véget ér. Összehasonlításként itt egy egyenértékű függvény `for` használatával:

```
1 def osszeg_eddig(n):
2     """ Számsorozat összege 1+2+3+...+n """
3     ro = 0
4     for e in range(n+1):
5         ro = ro + e
6     return ro
```

Figyeld meg a kicsit trükkös `range` függvényhívást – egyet hozzá kell adnunk `n`-hez, különben a `range` által előállított lista nem tartalmazná a megadott értéket. Könnyű úgy programozási hibát ejteni, hogy ez elkerüli a figyelmünket. Azonban mivel már megírtuk az egységesíteteket, a tesztkészletünk észlelné ezt a hibát.

Miért van akkor kétféle ciklus, ha a `for` egyszerűbbnek néz ki? A következő példa megmutat egy esetet, amikor olyan extra képességekre van szükségünk, amit a `while` tud csak megadni.

7.5. A Collatz-sorozat

Tegyük egy pillantást egy egyszerű sorozatra, amely elbűvölte a matematikusokat sok évig. Még mindig nem tudnak megválaszolni egy elég egyszerű kérdést ezzel kapcsolatban.

A „számítási szabály” amivel előállítjuk a sorozatot egy adott `n` számmal kezdődik és ahhoz, hogy előállítsuk a következő elemet az `n`-ből, azt kell tennünk, hogy megfelezzük `n`-et, ha az páros, különben pedig egyel növelni kell az `n` háromszorosát. A sorozat véget ér, ha `n` értéke eléri az 1-et.

Ez a Python függvény megragadja az algoritmus lényegét:

```
1 def sor3np1(n):
2     """ Kiírja a 3n+1 sorozatot n-től amíg el nem éri az 1-et. """
3     while n != 1:
4         print(n, end=" ", " )
5         if n % 2 == 0:          # n páros
6             n = n // 2
7         else:                   # n páratlan
8             n = n * 3 + 1
9     print(n, end=".\n")
```

Figyeld meg, hogy a `print` függvénynek a 6. sorban van egy extra paramétere `end=" ", " !` Ez azt mondja a `print` függvénynek, hogy folytassa a kiíratott szöveget azzal, amit a programozó akar (ebben az esetben egy vesszővel és egy szóközzel), ahelyett, hogy befejezné a sort. Szóval valahányszor kiírnunk valamit a ciklusban, az mindig ugyanabban a sorban jelenik meg, vesszővel elválasztott számok formájában. A `print(n, end=".\n")` hívása a 11. sorban a ciklus befejeződése után kiíratja az utolsó `n` értéket, majd egy pont és egy új sor karaktert. (A `\n` (új sor) karakterrel a következő fejezetben foglalkozunk.)

A ciklus ismétlésének feltétele most `n!=1`, így a ciklus addig folytatódik, amíg el nem éri a befejezési feltételt (azaz `n==1`).

Minden egyes cikluslépésben a program kiírja az `n` értékét, aztán megvizsgálja, hogy páros-e vagy páratlan. Ha páros, akkor `n`-et osztja 2-vel egész osztás segítségével. Ha a szám páratlan, akkor a változó értékét `n * 3 + 1`-re cseréli. Itt van pár példa:

```
1 sor3npl(3)
2 sor3npl(19)
3 sor3npl(21)
4 sor3npl(16)
```

Az első hívás kimenete: 3, 10, 5, 16, 8, 4, 2, 1. A második hívás kimenete: 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. A harmadik hívás kimenete: 21, 64, 32, 16, 8, 4, 2, 1. A negyedik hívás kimenete: 16, 8, 4, 2, 1.

Mivel n értéke néha nő, néha csökken, nincs nyilvánvaló bizonyíték, hogy n valaha eléri az 1-et vagyis, hogy a programunk befejeződik. Néhány különös n érték esetén be tudjuk bizonyítani a befejeződést. Például, ha a kezdőérték 2 valamelyik hatványa, akkor n értéke mindig páros lesz és csökken 1-ig. A fenti példa egy ilyen esettel zárult, 16-tal kezdve.

Lássuk találsz-e olyan kis kezdőértéket, mely esetén több, mint száz ismétlés szükséges a befejezéshez!

Az igazán érdekes kérdést először a német matematikus Lothar Collatz tette fel: ez a *Collatz-sejtés* (vagy más néven a *$3n+1$ sejtés*), miszerint ez a sorozat befejeződik *minden* pozitív n kezdőérték esetén. Eddig senki sem tudta bebizonyítani vagy megcáfolni ezt. (A sejtés egy olyan állítás, amely mindig igaz lehet, de senki sem tudja biztosan.)

Gondolkozz el azon, hogy mi szükséges a bizonyításhoz vagy a cáfolathoz „*minden pozitív egész szám végül is 1-hez konvergál a Collatz-szabályokat alkalmazva*”. Gyors számítógépekkel minden egész számot tesztelni tudunk egy nagyon nagy számig, mostanáig mind az 1-gyel végződött. De ki tudja? Talán van egy eddig nem vizsgált érték, ami nem 1-hez vezet.

Észre fogod venni, hogy ha nem állsz meg az 1-nél, akkor a sorozat egy ciklusba vezet: 1, 4, 2, 1, 4, 2, 1, 4, ... Így előfordulhat, hogy akár olyan esetek is léteznek, amikor egy másik ciklust kapunk, amit eddig még nem találtunk meg.

A Wikipédiának van egy tájékoztató cikke a Collatz sejtésről. A sorozatot más nevekkel is szokták illetni (Hailstone-sorozat, Wonderous-számok, stb.) és megtalálhatod azt is, hány számot vizsgáltak már meg eddig és kaptak konvergenciát.

Választás `for` és `while` közül

Használj `for` ciklust, ha már az ismétlés előtt tudod, hogy maximum hányszor kell végrehajtani a ciklustörzset! Ha például elemek egy listáját járod be, akkor tudhatod, hogy maximum hány ismétlés szükséges a „lista összes eleméhez”. Vagy ha ki kell íratnod a 12-es szorzótáblát, tudhatod hányszor kell ismétlni a ciklust.

A `for` ciklus alkalmazását javasolják az olyan jellegű problémák, mint az „ismételd az időjárási modellt 1000 ciklusra” vagy „keress ezen szavak listájában” vagy „találd meg az összes 10000-nél kisebb prímszámot”.

Ezzel ellentétben, ha egy feltétel bekövetkeztéig kell ismételned egy számítást és nem tudod előre kiszámolni, hogy ez mikor következhet be, ahogy ez a $3n+1$ probléma esetén is volt, akkor a `while` ciklust kell használnod.

Az első esetet **előírt lépésszámú ciklusnak** hívjuk – időben előre ismerjük azt a határozott határt, ami szükséges. Az utóbbi esetet **feltételes ciklusnak** hívjuk – nem vagyunk biztosak az ismétlésszámban – itt nem tudunk felső limitet mondani az ismétlésre.

7.6. A program nyomkövetése

A hatékony programíráshoz és fogalmilag helyes program-végrehajtási modell felállításához a programozónak arra van szüksége, hogy fejlessze a programvégrehajtáshoz kapcsolódó **nyomkövető** képességét. A nyomkövetés magába foglalja a számítógéppé válást és az utasítások végrehajtásának követését egy példaprogramon keresztül, minden változó állapotának és az összes generált kimenetnek a rögzítését.

A folyamat megértéséhez kövessük nyomon az előző alfejezetbeli `sor3np1(3)` függvényhívást. A nyomkövetés kezdetén van egy `n` változónk (a paraméter) 3 kezdőértékkel. Mivel a 3 nem egyenlő 1-gyel, a `while` ciklus törzse végrehajtódik. A 3 kiíratódik és a `3 % 2 == 0` kiértékelődik. Mivel az értéke `False` az `else` ág hajtódik végre és a `3 * 3 + 1` kiértékelése után az eredmény az `n` változóban kerül tárolásra.

Ennek az egésznek a kézi nyomon követéséhez készítünk egy fejléctet egy darab papíron a program összes változója és az kimenete számára. A nyomkövetésünk eddig így nézne ki:

n	Eddigi kimenet
--	-----
3	3,
10	

Mivel a `10 != 1` kiértékelés `True` értéket ad, a ciklus törzs ismét végrehajtódik és kiírja a 10-et. A `10 % 2 == 0` igaz, így az `if` ág hajtódik végre és `n` értéke 5 lesz. Ennek a nyomkövetésnek a végére azt kapjuk:

n	Eddigi kimenet
--	-----
3	3,
10	3, 10,
5	3, 10, 5,
16	3, 10, 5, 16,
8	3, 10, 5, 16, 8,
4	3, 10, 5, 16, 8, 4,
2	3, 10, 5, 16, 8, 4, 2,
1	3, 10, 5, 16, 8, 4, 2, 1.

Ez a nyomkövetés egy kicsit unalmas és hiba gyanús (ez az, amiért elsősorban a számítógéppel végeztetjük az ilyen dolgokat), de ez egy alapvető programozói képesség. Ezzel a nyomkövetéssel sokat tanulhatunk arról, hogyan működik a kódunk. Megfigyelhetjük, hogy amint `n` értéke 2 egyik hatványa lesz, a programnak $\log_2(n)$ alkalommal kell végrehajtania a ciklustörzset a befejezéshez. Azt is láthatjuk, hogy a végső 1 érték nem lesz kiíratva a ciklustörzs kimeneteként, emiatt egy speciális `print` függvényhívást tettünk a program végére.

A program nyomkövetése természetesen a **lépésenkénti** kódvégrehajtáshoz kapcsolódik, és képes a változók ellenőrzésére. A számítógép használata az **egyesével történő léptetéshez** sokkal kevésbé tévesztés veszélyes, és sokkal kényelmesebb. Valamint ahogy a program egyre komplexebb lesz, akár több milliónyi utasítást is végrehajthat, mire a bennünket érdeklő kódrészlethez jutunk, így a manuális nyomkövetés lehetetlenné válik. A **töréspont** használatának képessége, ahol szükség van rá, az sokkal hathatósabb. Így arra bátorítunk, hogy fektess időt abba, hogy megtanuld a fejlesztési környezeted használatát (itt ez a PyCharm) a teljes hatás eléréséért.

Van néhány nagyszerű vizualizációs eszköz, amely elérhető számodra is a nyomkövetés segítéséhez és kis Python kódtörödékek megértéséhez.

Figyelmeztettünk téged a csevegő függvények elkerülésére, de itt használhatod őket. Ahogy egy kicsit többet tanulunk a Pythonról, képesek leszünk megmutatni neked, hogyan generálhatsz értéklistát a sorrend megtartásához, ahelyett, hogy a `print` függvényt használnád. Ennek használatával szükségtelen lesz az összes bosszantó `print` függvény a program közepén, és így a függvényed sokkal hasznosabb lesz.

7.7. Számjegyek számlálása

A következő függvény egy pozitív számot alkotó decimális számjegyek számát számolja meg:

```
1 def szamjegy_szam(n):
2     szamlalo = 0
3     while n != 0:
4         szamlalo = szamlalo + 1
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5     n = n // 10
6     return szamlalo
```

A `print(szamjegy_szam(710))` hívás eredménye a 3 kiírása. Kövesd nyomon ezt a függvényhívást (használhatsz lépésenkénti függvényvégrehajtást a PyCharmban vagy egy Python vizualizációt esetleg egy darab papírt) számodra kényelmes módon.

Ez a függvény egy fontos számítási mintát demonstrál, a **számlálót**. A `szamlalo` változó 0-val van inicializálva, és inkrementálva van a ciklustörzs minden egyes végrehajtásánál. Amikor a ciklusból kilépünk a `szamlalo` tartalmazza az eredményt – a ciklustörzs végrehajtásának maximális számát, ami megegyezik a számjegyek számával.

Ha csak a 0 és 5 számjegyeket akarjuk megszámolni, akkor a trükkhöz egy feltételes utasítás kell a számláló inkrementálás elé:

```
1 def nulla_es_ot_szamjegy_szam(n):
2     szamlalo = 0
3     while n > 0:
4         szamjegy = n % 10
5         if szamjegy == 0 or szamjegy == 5:
6             szamlalo = szamlalo + 1
7         n = n // 10
8     return szamlalo
```

Ellenőrizd a helyességet egy teszt `nulla_es_ot_szamjegy_szam(1055030250) == 7)` tesztel.

Vedd észre, hogy a teszt `(szamjegy_szam(0) == 1)` teszt elbukik! Magyarázd meg miért! Szerinted ez egy programhiba a kódban vagy hiba a specifikációban vagy ez az, amit elvárunk?

7.8. Rövidített értékadás

Egy változó inkrementálása olyan hétköznapi dolog, hogy a Python egy lerövidített szintaxist is szolgáltat hozzá:

```
>>> szamlalo = 0
>>> szamlalo += 1
>>> szamlalo
1
>>> szamlalo += 1
>>> szamlalo
2
```

a `szamlalo += 1` egy rövidítése a `szamlalo = szamlalo + 1` kifejezésnek. Az operátor kiejtése: „*plusz-egyenlő*”. A hozzáadandó érték nem kell, hogy 1 legyen:

```
>>> n = 2
>>> n += 5
>>> n
7
```

Vannak más hasonló rövidítések is `-=`, `*=`, `/=`, `//=` and `%=`:

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
```

(folytatás a következő oldalon)

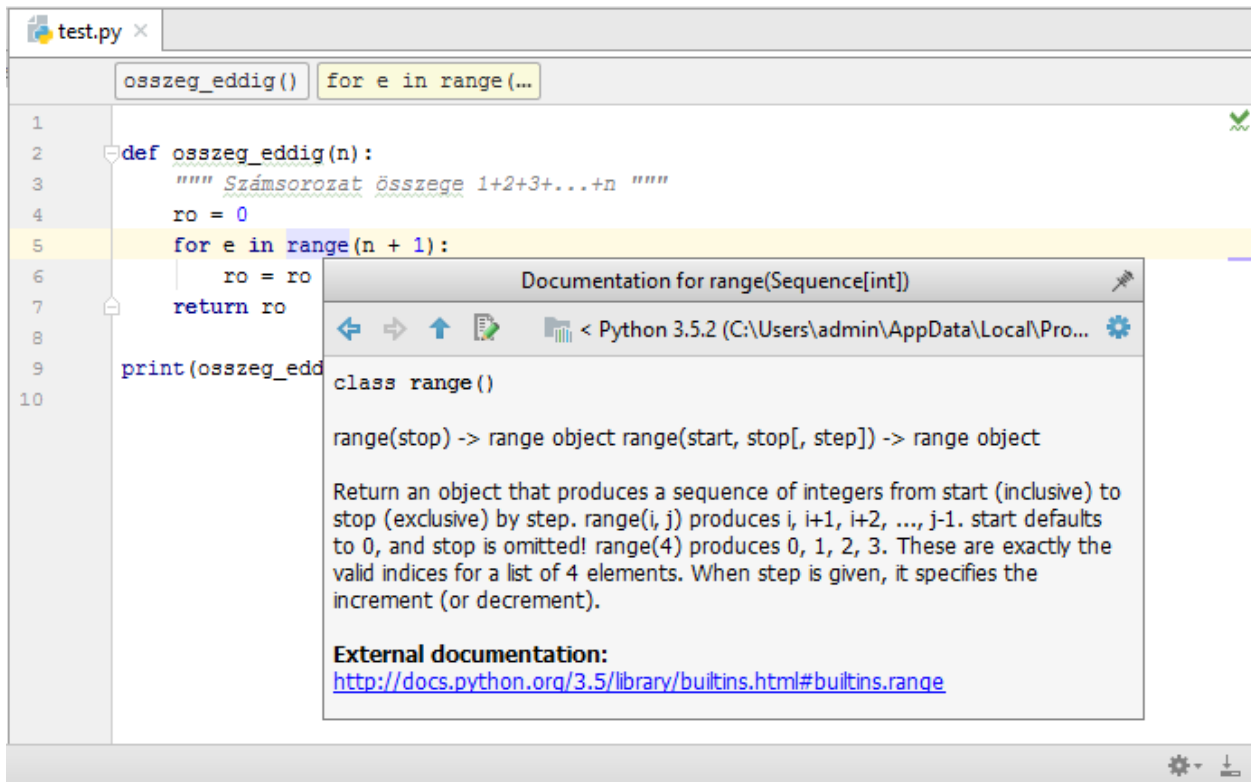
(folytatás az előző oldalról)

```
>>> n
6
>>> n //= 2
>>> n
3
>>> n %= 2
>>> n
1
```

7.9. Súly és meta-jelölés

A Python kiterjedt dokumentációval van ellátva az összes beépített függvényről és könyvtárról. A különböző rendszerekben különbözőképpen férhetünk hozzá ehhez a súlyhoz. A PyCharmban egy függvény vagy modul néven állva nyomhatunk **SHIFT + F1** billentyűkombinációt, aminek hatására a böngészőben megnyílik egy külső, angol nyelvű dokumentáció (cím: <https://docs.python.org/3.6/>). Itt a beépített konstansok, típusok, függvények és egyebek részleteiről olvashatsz.

Egy másik lehetőség információszerzésre, hogy a kiválasztott programozói eszköz néven állva **CTRL + q** billentyűkombináció segítségével megjelenik egy kis felugró ablak az adott eszköz leírásával. A `range` esetén ezt láthatod:



Figyeld meg a szögletes zárójeleket a paraméterek leírásában! Ezek a **meta-jelölések** példák – a jelölések írják le a Python szintaxisát, de ezek nem részei annak. A szögletes zárójelek a dokumentációkban azt jelentik, hogy a paraméter *opcionális* – a programozó kihagyhatja. Mint látható a `stop` paraméter kötelező, viszont lehetnek más paraméterei is a `range`-nek (vesszővel elválasztva). Egy kezdőértéktől (`start`) indulhatunk a végértékig (`stop`) felfelé vagy lefelé inkrementálva egy lépésközzel (`step`). A dokumentáció azt is mutatja, hogy a paraméterek `int` típusúak kell, hogy legyenek. Gyakran félkövérrel jelölik azokat a szövegelemeket – kulcsszavakat és szimbólumokat

– amelyeket pontosan a megadott módon kell írni Pythonban, és dőlt stílussal írják a helyettesíthető szövegelemeket (például változók nevei):

for variable in list :

Egyes esetekben – például a `print` függvény esetén – láthatunk a dokumentációban egy `...` meta-jelölést, ami azt fejezi ki, hogy tetszőleges számú paraméter alkalmazható.

print([object, ...])

A meta-jelölés egy tömör és hatékony módja a szintaxisminták leírásának.

7.10. Táblázatok

Az egyik dolog, amiben a ciklusok jók, az a táblázatok generálása. Mielőtt a számítógépek elérhetőek lettek, az embereknek kézzel kellett logaritmust, szinuszt, koszinuszt és egyéb matematikai függvényeket számolniuk. Ennek egyszerűbbé tételére a matematikai könyvek hosszú táblázatokat tartalmaztak, amelyekben e függvények értékeit listázták. Egy táblázat készítése lassú és unalmas volt, és ezek tele voltak hibával.

Amikor a számítógépek megjelentek a színen, akkor ez első reakciók egyike ez volt: „*Ez nagyszerű! Használhatjuk a számítógépeket táblázatok generálására, így azok hibamentesek lesznek.*” Ez igaz volt, de rövidlátó. Hamarosan a számítógépek olyan elterjedtek lettek, hogy a táblázatok elavulttá váltak.

Nos, majdnem. Néhány művelet esetén a számítógépek értékek táblázatait használják, hogy kapjanak egy közelítő választ, és aztán számításokat végeznek, hogy javítsák a közelítést. Néhány esetben vannak hibák a háttérben megbúvó táblázatokban, a legismertebb az Intel Pentium processzorchipen által lebegőpontos osztás során használt táblázat hibája.

Habár a log táblázat már nem annyira hasznos, mint régebben volt, még mindig jó példa az iterációra. A következő program kimenete értékek egy sorozata a bal oldali oszlopban, és a 2 ennyiedik hatványa a jobb oldali oszlopban:

```
1 for x in range(13):    # Generálj számokat 0 és 12 között
2     print(x, "\t", 2**x)
```

A `"\t"` sztring jelenti a **tabulátor karaktert**. A visszafelé-per jel a `"\t"` sztringben egy speciális **escape karakter** kezdetét jelzi. Az escape karakterek segítségével jelenítjük meg a nem nyomtatható karaktereket, mint a tabulátor (tab) vagy az új sor karakter. A `\n` jelenti az **új sor karaktert**.

Egy escape karakter bárhol megjelenhet a sztringben, a fenti példában e tabulátor karakter ez egyetlen dolog a sztringben. Mít gondolsz, hogyan kell megjeleníteni egy visszafelé-per jelet egy sztringben?

Ahogy a karakterek és a sztringek megjelenítődnek a képernyőn egy láthatatlan jelölő az ún. **kurzor** tartja számon hova kerül a következő karakter. A `print` függvény után normális esetben a kurzor a következő sor elejére kerül. A tabulátor karakter eltolja a kurzort jobbra a következő kurzorstop egységhez. A tabulátorok hasznosak az oszlopok készítésekor, mint ahogy az előző példában is:

0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

(folytatás a következő oldalon)

(folytatás az előző oldalról)

11	2048
12	4096

Mivel egy tabulátor karakter van a két oszlop között, így a második oszlop pozíciója nem függ az elsőben lévő számjegyek számától.

7.11. Kétdimenziós táblázat

Egy kétdimenziós táblázat, olyan táblázat, ahol az értéket egy sor és egy oszlop kereszteződéséből kell kiolvasni. A szorzótábla egy jó példa. Mondjuk, ki szeretnéd írni a 6x6-os szorzótáblát.

Egy jó módja a kezdésnek, ha írsz egy ciklust a 2 többszöröseinek egy sorba történő kiírásához:

```
1 for i in range(1, 7):
2     print(2 * i, end=" ")
3 print()
```

Itt a `range` függvényt használtuk, de az 1-gyel kezdi a sorozatát. Ahogy a ciklus előrehalad az `i` értéke 1-ről 6-ra változik. Amikor már a sorozat összes eleme hozzá lett rendelve az `i` változóhoz, akkor a ciklus befejeződik. Minden egyes cikluslépés során a `2 * i` értéke jelenik meg három szóközzel elválasztva.

Ismét egy extra `end=" "` paraméter van a `print` függvényben, ami elnyomja az új sor karakter megjelenését, és helyette inkább 3 szóközt használ. Miután a ciklus befejeződött a 3. sor `print` hívása befejezi az aktuális sort, és kezd egy újat.

A program kimenete ez:

2	4	6	8	10	12
---	---	---	---	----	----

Eddig jó. A következő lépés az **enkapszuláció** (vagyis beágyazás) és az **általánosítás**.

7.12. Enkapszuláció és általánosítás

Az enkapszuláció a kód egy darabjának becsomagolása egy függvényben, lehetővé téve számodra, hogy kihasználd az előnyét az összes olyan dolognak, amiben a függvények jók. Láttál már pár példát az enkapszulációra, beleértve az előző fejezet `osztathato_e` függvényét.

Az általánosítás azt jelenti, hogy veszünk valami specifikus dolgot, mint például a 2 többszöröseinek kiírás és általánosabbá tesszük, mint például bármely egész többszöröseinek kiírása.

Ez a függvény az enkapszuláció révén tartalmazza az előző ciklust és általánosítja `n` többszöröseinek kiírására:

```
1 def tobbszorosok_kiirasa(n):
2     for i in range(1, 7):
3         print(n * i, end=" ")
4     print()
```

Ahhoz, hogy egységbe gyűrjünk mindent, amit tennünk kell, először deklaráljuk a függvény nevét és a paraméterlistáját. Az általánosításhoz, ki kell cserélnünk a 2 értéket az `n` paraméterre.

Ha meghívjuk ezt a függvényt a 2 paraméterrel, akkor az előzőhöz hasonló kimenetet kapunk. A 3, mint paraméter esetén ez a kimenet:

3	6	9	12	15	18
---	---	---	----	----	----

Ha az aktuális paraméter 4, a kiment a következő:

4	8	12	16	20	24
---	---	----	----	----	----

Mostanra bizonyára sejtet, hogy íratjuk ki az szorzótáblát – a `tobbszorosok_kiirasa` függvény különböző paraméterekkel történő hívásával. Valójában egy másik ciklust kell alkalmazni:

```
1 for i in range(1, 7):  
2     tobbszorosok_kiirasa(i)
```

Figyeld meg, milyen hasonló ez a ciklus a `tobbszorosok_kiirasa` függvényen belül lévőhöz. Amit tettünk, az csak annyi, hogy kicseréltük a `print` függvényt egy másik függvényhívásra.

Ennek a programnak a kimenete a szorzótábla:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

7.13. Még több enkapszuláció

Az enkapszuláció újbóli demonstrációjához vegyük a kódot a legutóbbi alfejezetből és csomagoljuk be egy függvénybe:

```
1 def szorzotabla_kiiras():  
2     for i in range(1, 7):  
3         tobbszorosok_kiirasa(i)
```

Ez a folyamat egy közösleges **fejlesztési terv**. A program fejlesztés során az új kódrészleteket a függvényeken kívül hozzuk létre, vagy parancsértelmezőben próbáljuk ki. Amikor működőképpessé tettük a kódot, akkor becsomagoljuk egy függvénybe.

A fejlesztési terv különösen hasznos, ha az írás kezdetekor még nem tudjuk, hogyan osszuk fel a programot függvényekre. Ez a megközelítés lehetővé teszi számodra, hogy menet közben tervezz.

7.14. Lokális változó

Talán csodálkozol, hogyan tudjuk ugyanazt az `i` változót használni mind a `tobbszorosok_kiirasa`, mind a `szorzotabla_kiiras` függvényben. Nem jelent ez gondot, amikor az egyik függvény módosítja a változó értékét?

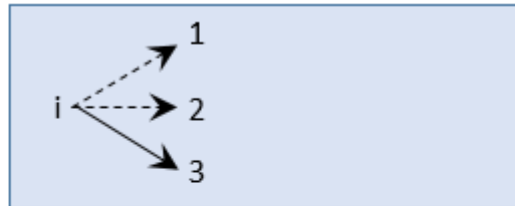
A válasz nem, mert az `i` változó a `tobbszorosok_kiirasa` függvényben és az `i` változó a `szorzotabla_kiiras` függvényben *nem* ugyanaz a változó.

Egy függvényen belül létrehozott változó az lokális, nem tudsz egy lokális változót elérni az őt deklaráló függvényen kívül. Ez azt jelenti, hogy szabadon adhatjuk ugyanazt a nevet több változónak, amíg azok nem ugyanabban a függvényben vannak.

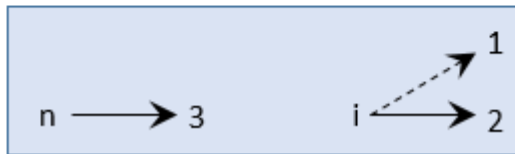
Python megvizsgálja az összes utasítást a függvényben – ha bármelyik ezek közül értéket ad egy változónak, az annak jele, hogy a Python lokális változóként használja.

Ennek a programnak a verem diagramja azt mutatja, a két `i` nevű változó nem ugyanaz. Ők különböző értékekre hivatkoznak, és az egyik megváltoztatásának nincs hatása a másikra.

szorzotabla_kiiras



tobbszorosek_kiirasa



Az `i` értéke a `szorzotabla_kiiras` függvényben 1-től 6-ig megy. A diagramban ez éppen 3. A következő alkalommal ez 4 lesz a ciklusban. Minden egyes cikluslépés során a `szorzotabla_kiiras` meghívja a `tobbszorosek_kiirasa` függvényt az aktuális `i` értékével, mint aktuális paraméterrel. Ez az érték kerül átadásra az `n` formális paraméterbe.

A `tobbszorosek_kiirasa` függvényben az `i` értéke 1-től 6-ig megy. A diagramban ez éppen 2. Ennek az értéknek a megváltoztatása nincs hatással a `szorzotabla_kiiras` függvény `i` változójára.

Hétköznapi és teljesen legális, ha több különböző lokális változónk van ugyanazzal a névvel. Különösen az `i` és a `j` neveket használjuk gyakran ciklusváltozóként. Ha elkerülöd a használatukat egy függvényben csak azért, mert máshol használod a nevet, akkor valószínűleg nehezebben olvashatóvá teszed a programot.

7.15. A `break` utasítás

A `break` utasítás arra való, hogy azonnal elhagyjuk a ciklus törzsét. A következő végrehajtandó utasítás a törzs utáni első utasítás:

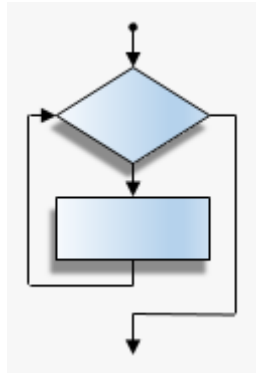
```
1 for i in [12, 16, 17, 24, 29]:
2     if i % 2 == 1: # Ha a szám páratlan ...
3         break    # ... azonnal hagyd el a ciklust
4     print(i)
5 print("Kész.")
```

Ezt írja ki:

```
12
16
Kész.
```

Az előtesztelő ciklus – normál ciklus viselkedés

A `for` és `while` ciklusok először végrehajtják a tesztjeiket mielőtt a törzs bármelyik részét végrehajtanák. Ezeket gyakran **előtesztelő ciklusoknak** hívják, mert teszt a törzs előtt történik. A `break` és a `return` eszközök ehhez a normál viselkedéshez való alkalmazkodáshoz.

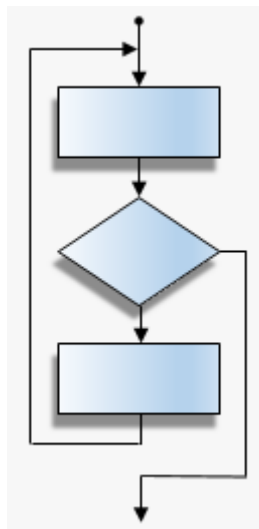


7.16. Más típusú ciklusok

Néha szükségünk lenne egy **középen tesztelő** ciklusra, egy kilépési tesztel inkább a törzs közepén, mint az elején. Vagy máskor **hátultesztelő** ciklus lenne jó, ahol a kilépési feltétel a ciklus legvégén van. Más nyelveknek különböző szintaxisa és kulcsszava van ezekre, de a Python csak egyszerűen kombinálja a `while` és az `if` feltételt: `break` szerkezeteket.

Egy tipikus példa, amikor a felhasználó által bemenetként adott számokat kell összegezni. A felhasználó, hogy jelezze nincs több adat, egy speciális értéket ad meg, gyakran a -1 értéket vagy egy üres sztringet. Ehhez egy középen kilépő ciklusminta kell: olvasd be a következő számot, ellenőrizd ki kell-e lépni, ha nem, akkor dolgozd fel a számot.

A középen tesztelő ciklus folyamatábrája



```
1 összeg = 0
2 while True:
3     bemenet = input("Add meg a következő számot! (Hagyd üresen a_
    ↳befejezéshez) ")
4     if bemenet == "":
5         break
6     összeg += int(bemenet)
7     print("Az általad megadott számok összege: ", összeg)
```

Győződj meg arról, hogy ez illeszkedik a középen tesztelő ciklus folyamatábrájához: a 3. sor valami hasznosat csinál, a 4. és 5. sor kilép a ciklusból, ha kell, ha viszont nem lép ki, akkor a 6. sor ismét valami hasznosat csinál, mielőtt a ciklus újrakezdődne.

A `while` logikai-kifejezés: szerkezet egy Boolean kifejezést használ, hogy döntsön az ismétlésről. A `True` egy triviális logikai kifejezés, így a `while True:` azt jelenti *mindig ismételd meg a ciklustörzset újra és újra*. Ez egy nyelvi fordulat – egy konvenció, amit a programozónak azonnal fel kell ismernie. Mivel a 2. sor kifejezése sohasem fejezti be a ciklust (ez egy kamu teszt), a programozónak kell gondoskodnia a ciklus elhagyásáról valahol máshol, más módon (pl.: a fenti program 4. és 5. sorában). Egy okos fordító vagy parancsértelmező megérti, hogy a 2. sor egy álesztst, ami mindig igaz így nem fog tesztet generálni, és a folyamatábrán sem lesz soha feltételt jelentő paralelogramma a ciklus tetején.

Hasonlóan ehhez, ha az `if` feltétel: `break` szerkezetet a ciklustörzs végére tesszük, megkapjuk a hátultesztelő ciklus mintáját. A hátultesztelő ciklus akkor használatos, ha biztos akarsz lenni abban, hogy a ciklustörzs legalább egyszer lefut (mivel az első teszt csak akkor történik meg, amikor a ciklustörzs első végrehajtása befejeződött). Ez hasznos például akkor, amikor egy interaktív játékot akarsz játszani a felhasználó ellen – mindig legalább egy játékot akarunk játszani:

```
1 while True:
2     jatek_egyszer()
3     felelet = input("Játszunk megint? (igen vagy nem)")
4     if felelet != "igen":
5         break
6 print("Viszlát!")
```

Segítség: Gondolkodj azon, hová tennéd a kilépési feltételt?

Ha már egyszer rájöttél arra, hogy szükséged van egy ciklusra valami ismétléséhez, akkor gondolkodj a befejezési feltételen – mikor akarod megállítani az ismétlést? Aztán találd ki vajon szükséged van arra, hogy a teszt az első (és minden további) iteráció elején legyen vagy az első (és minden további) iteráció végén, esetleg talán az egyes iterációk közepén. Az interaktív programok esetén, amelyek bemenetet igényelnek a felhasználótól vagy fájlból olvasnak gyakran a ciklus közepén vagy végén van a kilépési feltétel, ekkor válik világossá, hogy nincs több adat, vagy a felhasználó nem akar többet játszani.

7.17. Egy példa

A következő program egy kitalálós játékot implementál:

```
1 import random # Beszélni fogunk a véletlen számokról...
2 vel = random.Random() # ...a modulok fejezetbe, szóval fel a
   ↳ fejjel.
3 szam = vel.randrange(1, 1000) # véletlen szám [1 és 1000) intervallumban.
4
5 tippszam = 0
6 uzenet = ""
7
8 while True:
9     tipp = int(input(uzenet + "\nTaláld ki az 1 és 1000 közötti számot,
   ↳ amire gondoltam: "))
10    tippszam += 1
11    if tipp > szam:
12        uzenet += str(tipp) + " túl nagy.\n"
13    elif tipp < szam:
```

(folytatás a következő oldalon)

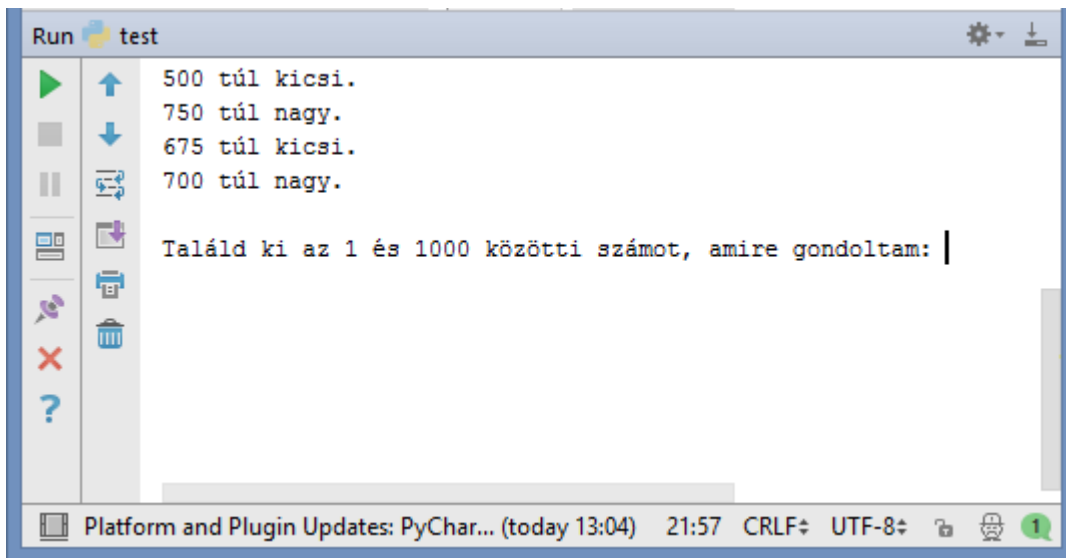
(folytatás az előző oldalról)

```
14     uzenet += str(tipp) + " túl kicsi.\n"
15     else:
16         break
17
18 input("\n\nNagyszerű, kitaláltad {0} tipp segítségével!\n\n".
    ↪format(tippszam))
```

Ez a program a matematikai **trichotómia** szabályát alkalmazza (adott a és b valós számok esetén az alábbi esetek pontosan egyike igaz: $a > b$, $a < b$ vagy $a == b$).

A 18. sorban van egy input függvény hívás, de nem csinálunk semmit az eredményével, még változónak sem adjuk értékül. Ez szabályos Pythonban. Itt ennek az a hatása, hogy megnyílik egy párbeszéd ablak várva a felhasználó válaszára, mielőtt befejeződne a program. A programozók gyakran használják ezt a trükköt a szkript végén, csak azért, hogy nyitva tartsák az ablakot.

Figyeld meg az uzenet változó használatát is! Kezdetben ez egy üres sztring. Minden egyes cikluslépésben kiterjesztjük a megjelenítendő üzenetet: ez lehetővé teszi, hogy a program visszajelzést adjon a megfelelő helyen.



7.18. A continue utasítás

Ez egy vezérlésátadó utasítás, amely a ciklustörzs hátralévő utasításainak kihagyását eredményezi az adott ismétlésre vonatkozóan. A ciklus azonban tovább folytatódik a hátralévő ismétlésekkel:

```
1 for i in [12, 16, 17, 24, 29, 30]:
2     if i % 2 == 1:      # Ha a szám páratlan ...
3         continue      # ... ne dolgozd fel!
4     print(i)
5 print("Kész.")
```

Ez ezt írja ki:

```
12
16
24
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

30
Kész.

7.19. Még több általánosítás

Az általánosítás másik példaként képzeljük el, hogy írni akarunk egy programot tetszőleges méretű szorzótábla kiírására, nem csak a 6x6-os esetre korlátozódva. Adhatsz egy paramétert a `szorzotabla_kiiras` függvényhez:

```
1 def szorzotabla_kiiras(magassag):  
2     for i in range(1, magassag+1):  
3         tobbszorosok_kiirasa(i)
```

Kicséréltük a 7-es értéket a `magassag+1` kifejezésre. Amikor a `szorzotabla_kiiras` hívásra kerül a 7 aktuális paraméter értékkel, akkor ezt látjuk:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

Ez jó, kivéve, ha azt várjuk el, hogy a táblázat négyzet alakú legyen – azonos sor- és oszlop számmal. Ekkor egy másik paramétert is adnunk kell a `szorzotabla_kiiras` függvénynek az oszlopok számának megadásához.

Szörszálhasogatásként megjegyeznénk, hogy ezt a paramétert `magassag` névvel láttuk el, azt demonstrálva, hogy különböző függvényeknek lehetnek azonos nevű paraméterei (mivel lokális változók). Itt van a teljes program:

```
1 def tobbszorosok_kiirasa(n, magassag):  
2     for i in range(1, magassag+1):  
3         print(n * i, end="    ")  
4         print()  
5  
6 def szorzotabla_kiiras(magassag):  
7     for i in range(1, magassag+1):  
8         tobbszorosok_kiirasa(i, magassag)
```

Jegyezd meg, ha adunk egy új paramétert a függvényhez, akkor meg kell változtatni az első sort (függvény fejléc), és a hívás helyén is változtatást kell eszközölnünk.

Most ha végrehajtjuk a `szorzotabla_kiiras(7)` függvényhívást, ezt látjuk:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

Amikor megfelelően általánosítasz egy függvényt, gyakran kapsz olyan programot, ami nem tervezett dolgokra is képes. Például, bizonyára észrevetted, hogy mivel $ab = ba$ így minden elem kétszer jelenik meg a táblázatban. Tintát spórolhatsz meg, ha csak a táblázat felét nyomtatod ki. Ehhez csak a `szorzotabla_kiiras` egyik sorát kell megváltoztatni. Írd ehelyett

```
1 tobbszorosok_kiirasa(i, magassag+1)
```

ezt

```
1 tobbszorosok_kiirasa(i, i+1)
```

és ezt kapod:

1						
2	4					
3	6	9				
4	8	12	16			
5	10	15	20	25		
6	12	18	24	30	36	
7	14	21	28	35	42	49

7.20. Függvények

Egy párszor említettük már mire jók a függvények. Mostanra bizonyára csodálkozol, melyek is azok a dolgok. Itt van néhányuk:

1. A mentális blokkosítás eszköze. A komplex feladatod részfeladatokra tördelésével és ezeknek jelentésteli névadásával egy hatásos technikát kapunk. Tekints vissza a hátultesztelési ciklust illusztráló példára: azt feltételeztük, hogy van egy `jatek_egyszer` függvényünk. Ez a kis blokk lehetővé teszi számunkra, hogy félretegyük a konkrét játék részleteit – ami lehetne kártyajáték, sakk vagy szerepjáték – így a program logikájának egy izolált részére fókuszálhatunk – a játékos választhasson, hogy akar-e újra játszani.
2. Egy hosszú program függvényekre osztásával lehetőségünk van elszeparálni a program részeit, izoláltan tesztelni őket, és egy nagy egészet alkotni belőlük.
3. A függvények megkönnyítik a ciklusok használatát.
4. A jól megtervezett függvények hasznosak lehetnek több programban is. Egyszer megírod és debugolod, aztán máshol újra felhasználod.

7.21. Értékpár

Láttunk már neveket listájában és számok listájában Pythonban. Most egy kicsit előre fogunk tekinteni a könyvben és megmutatjuk az adattárolás egy fejlettebb módját. Párokat csinálni dolgokból Pythonban egyszerűen csak annyi, hogy zárójelekbe tesszük őket így:

```
1 szuletesi_ev = ("Paris Hilton", 1981)
```

Több párt be tudunk tenni egy párokat tartalmazó listába.

```
1 celebek = [("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", ↵  
↪1994)]
```

Itt egy kis példa azokra a dolgokra, amelyeket strukturált adatokkal tudunk csinálni. Először írassuk ki az összes celebet:

```
1 print(celebek)  
2 print(len(celebek))
```

```
[("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", 1994)]  
3
```

Figyeld meg, hogy a celebek lista csak 3 elemű, mindegyik elem egy pár!

Most írassuk ki azoknak a celebeknek a nevét, akik 1980 előtt születtek:

```
1 for (nev, ev) in celebek:  
2     if ev < 1980:  
3         print(nev)
```

```
Brad Pitt  
Jack Nicholson
```

Ez egy olyan dolgot demonstrál, amit eddig még nem láttunk a `for` ciklusban: egyszerű ciklusváltozó helyett egy `(nev, ev)` változónév párt használunk inkább. A ciklus háromszor fut le – egyszer minden listaelemre és mindkét változó kap egy értéket az éppen kezelt adatpárból.

7.22. Beágyazott ciklus beágyazott adatokhoz

Most egy még haladóbb strukturált adatlistát mutatunk. Ebben az esetben egy hallgatólistánk van. Minden egyes hallgatónak van egy neve, amihez egy másik listát párosítunk a felvett tantárgyairól:

```
1 hallgatok = [  
2     ("Jani", ["Informatika", "Fizika"]),  
3     ("Kata", ["Matematika", "Informatika", "Statisztika"]),  
4     ("Peti", ["Informatika", "Könyvelés", "Közgazdaságtan", "Menedzsment"]),  
5     ("Andi", ["Információs rendszerek", "Könyvelés", "Közgazdaságtan",  
6     ↪ "Vállalkozási jog"]),  
7     ("Linda", ["Szociológia", "Közgazdaságtan", "Jogi ismeretek",  
8     ↪ "Statisztika", "Zene"])]
```

Ezt az öt elemű listát értékül adjuk a `hallgatok` nevű változóhoz. Írassuk ki a hallgatók nevét és a tárgyaiknak a számát:

```
1 # Kiíratni a hallgatóneveket és a kurzusszámokat  
2 for (nev, targyak) in hallgatok:  
3     print(nev, "felvett", len(targyak), "kurzust.")
```

A Python az alábbi kimenettel válaszol:

```
Jani felvett 2 kurzust.  
Kata felvett 3 kurzust.  
Peti felvett 4 kurzust.  
Andi felvett 4 kurzust.  
Linda felvett 5 kurzust.
```

Most azt szeretnénk megkérdezni, hogy hány hallgató vette fel az Informatika tárgyat. Ehhez egy számlálóra van szükségünk és minden egyes hallgató esetén kell egy második ciklus, amely a tárgyakat ellenőrzi:

```
1 # Számold meg hány hallgató vette fel az Informatikát  
2 szamlalo = 0  
3 for (nev, targyak) in hallgatok:  
4     for t in targyak: # Beágyazott ciklus!
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5     if t == "Informatika":
6         szamlalo += 1
7
8     print("Az Informatikát felvett hallgatók száma:", szamlalo)
```

```
Az Informatikát felvett hallgatók száma: 3
```

Elő kellene állítanod azokat a listákat, amelyek érdekelnek – talán a CD-id listáját, az ezeken lévő dalok címeivel vagy a mozifilmek címeit a főszereplőikkel. Aztán tehetsz fel ezekkel kapcsolatos kérdéseket, mint például „Melyik filmben játszik Angelina Jolie?”

7.23. Newton módszer a négyzetgyök megtalálásához

A ciklusokat gyakran használjuk olyan programokban, amelyek numerikus számításokat végeznek, kiindulva egy közelítő értékből, majd azt közelítő értéket lépésről lépésre javítva.

Például, mielőtt számológépek és számítógépek lettek az embereknek, a négyzetgyök értékeket kézzel kellett számolniuk. Newton egy különösen jó módszert használt (van pár bizonyíték, miszerint a módszer már évekkel korábban ismert volt). Tegyük fel, hogy tudni akarod az n négyzetgyökét. Bármilyen közelítéssel is indulsz, kaphatsz egy jobbat (közelebb kerülhetsz az aktuális válaszhoz) az alábbi formulával:

```
1 jobb = (kozelites + n/kozelites)/2
```

Ismételd meg azt a számítást párszor egy számológéppel! Látod, miért kerül a becslésed mindig kicsit közelebb a megoldáshoz? Az egyik csodálatos tulajdonsága ennek a különleges algoritmusnak, hogy milyen gyorsan konvergál a pontos válaszhoz – ami nagyszerű dolog, ha kézzel számolsz.

Ennek a formulának az ismétlésével egyre jobb közelítést kaphatunk, amíg elég közel nem jutunk az előző értékhez, megírhatjuk ezt egy négyzetgyökszámoló függvényben. (Valóban ez a módszer, amit a számítógéped használ, talán egy kicsivel másabb a formula, de az is ismétléssel javítja a becslést.)

Ez egy példa a *határozatlan* iteráció problémájára: nem tudjuk előre megbecsülni, hányszor akarjuk majd ismételni a becslés javítását – mi csak egyre közelebb akarunk jutni. A ciklus megállási feltételünk azt jelenti, mikor lesz az előzőleg kapott értékünk „elég közel” az éppen most kapott értékhez.

Ideális esetben azt szeretnénk, hogy az új és a régi érték pontosan megegyezzen, amikor megállunk. Azonban a pontos egyenlőség trükkös dolog a számítógép aritmetikában, ha valós számokkal dolgozunk, mivel a valós számok nem abszolút pontosan vannak tárolva (mindezen túl az olyan számok, mint a π vagy a 2 négyzetgyöke, végtelen számú tizedesjegyet tartalmaznak), meg kell fogalmaznunk a megállási feltételt ezzel a kérdéssel: „ a elég közel van b -hez?” Ezt a feltételt így kódolhatjuk le:

```
1 if abs(a-b) < 0.001: # Csökkentheted az értéket a nagyobb pontossághoz
2     break
```

Vedd észre, hogy az a és b különbségének abszolút értékét használtuk!

Ez a probléma arra is jó példa, mikor alkalmazzuk a középben kilépő ciklusokat:

```
1 def gyok(n):
2     kozelites = n/2.0 # Kezdjük egy alap sejtéssel
3     while True:
4         jobb = (kozelites + n/kozelites)/2.0
5         if abs(kozelites - jobb) < 0.001:
6             return jobb
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
7         kozelites = jobb
8
9     # Teszt esetek
10    print(gyok(25.0))
11    print(gyok(49.0))
12    print(gyok(81.0))
```

A kimenet ez:

```
5.00000000002
7.0
9.0
```

Lásd, hogy a kilépési feltétel változtatásával tudod javítani a közelítést! Menj végig az algoritmuson (akár kézzel vagy számológéppel is), hogy meglásd, hány iteráció szükséges mielőtt eléred a megfelelő pontosságot a `gyok(25)` számolása során!

7.24. Algoritmusok

A Newton módszer egy példa **algoritmusokra**: ez egy mechanikus folyamat problémák egy kategóriájának (ebben az esetben négyzetgyök számolás) megoldásához.

Némely tudás nem algoritmikus. Például a történelmi dátumok megjegyzése vagy a szorzótábla speciális megoldásainak memorizálása.

De a technikák, amelyeket tanultál az átviteles összeadáshoz, a kivonáshoz vagy az osztáshoz azok is mind algoritmusok. Netán ha egy gyakorlott Sudoku megoldó vagy, akkor biztosan van pár speciális lépéssorod, amit követsz.

Az algoritmusok egyik sajátossága, hogy nem igényelnek semmilyen intelligenciát a kivitelezéshez. Mechanikus folyamatok, amelyekben minden lépés a megelőzőt követi egy egyszerű szabályhalmaznak megfelelően. Ezeket az algoritmusokat arra tervezték, hogy problémák egy általános osztályát vagy kategóriáját oldják meg, nem csak egy egyedülálló problémát.

Meg kell érteni, hogy a nehéz problémák lépésenként megoldhatóak algoritmikus folyamatokkal (és az ezek végrehajtásához szükséges technológiával), amely az egyik fő áttörés, ami óriási előnyt jelent. Szóval, míg az algoritmus futtatása unalmas lehet és nem igényel intelligenciát, addig az algoritmikus vagy számítógépes gondolkodás – azaz algoritmusok és automaták használata a problémák megközelítésének alapjaként – gyorsan megváltoztatja a társadalmunkat. Jó néhány dolog visz bennünket az algoritmikus gondolkodás irányába, és a folyamatok arra felé haladnak, hogy ennek nagyobb hatása lesz a társadalmunkra, mint a nyomtatás feltalálásának. Az algoritmusok megtervezésének folyamata érdekes, intellektuális kihívást jelent és központi része annak, amit programozásnak hívunk.

Néhány dolog, amit az emberek természetesen csinálnak nehézségek és tudatos gondolatok nélkül, a legnehezebben kifejezhető algoritmikusan. A természetes nyelvek megértése egy jó példa. Mindannyian megértjük, de eddig senki nem volt képes megmagyarázni *hogyan* csináljuk, legalábbis nem lépésenkénti mechanikus algoritmus formájában.

7.25. Szójegyzék

algoritmus (algorithm) Egy lépésenkénti folyamat problémák egy kategóriájának megoldásához.

általánosítás (generalization) Valami szükségtelenül specifikus dolog (mint egy konstans érték) lecserélése valami megfelelően általánosra (mint egy változó vagy egy paraméter). Az általánosítás sokoldalúvá teszi a programot a valószínű újrahajthatósághoz és a még könnyebb megíráshoz.

beágyazás (encapsulate) Egy nagy komplex program komponensekre (például függvényekre) bontása és a komponensek elszeparálása (például lokális változók használatával).

beágyazott ciklus (nested loop) Egy ciklus egy másik ciklus törzsén belül.

ciklus (loop) A konstrukció, ami lehetővé teszi, egy utasításcsoport ismételt végrehajtását egy feltétel teljesüléséig.

ciklusváltozó (loop variable) Egy változó, amit egy ciklus befejezési feltételében használunk.

continue utasítás (continue statement) Egy utasítás, mely azt eredményezi, hogy az aktuális ciklusismétlés hátralévő része ki lesz hagyva. A programvezérlés visszamegy a ciklus tetejére, a kifejezés kiértékeléséhez, és ha ez igaz, akkor elkezdődik a ciklus újbóli ismétlése.

dekrementálás (decrementation) Csökkentés 1-gyel.

előírt lépésszámú ciklus (definite iteration) Egy ciklus, ahol van egy felső határunk a ciklustörzs végrehajtásának számára vonatkozóan. Ez rendszerint jól kódolható a `for` ciklussal.

előltesztelő ciklus (pre-test loop) Egy ciklus, amely a törzs kiértékelése előtt hajtja végre a feltétel ellenőrzést. A `for` és a `while` ciklus is ilyen.

escape karakter (escape sequence) Egy `\` jel és az azt követő nyomtatható karakter együttese egy nem nyomtatható karakter megjelenítésére.

fejlesztési terv (development plan) Egy folyamat egy program fejlesztéséhez. Ebben a fejezetben bemutattunk egy fejlesztési stílust, ami egyszerű, specifikus dolog végrehajtására szolgáló kód fejlesztésén alapul, majd azt beágyazzuk másokba és általánosítjuk.

hátultesztelő ciklus (post-test loop) Egy ciklus, ami végrehajtja a törzset, aztán értékeli ki a kilépési feltételt. Pythonban ilyen célra a `while` és `break` utasítások együttesét használhatjuk.

inicializáció (initialization) Kezdeti érték adása egy változónak. Mivel a Python változók nem léteznek addig, amíg nem adunk nekik értéket, így ezek akkor, amikor létrejönnek inicializálódnak. Más programozási nyelvek esetén ez nem biztos, hogy igaz és a változók inicializálatlanul jönnek létre, amikor is vagy alapértelmezett értékük van vagy az értékük valami *memória szemét*.

inkrementálás (incrementation) Eggyel való növelés.

iteráció (iteration) Programozási utasítások sorozatának ismételt végrehajtása.

határozatlan ismétlés (indefinite iteration) Egy ciklus, ahol addig ismétlünk, amíg egy feltétel nem teljesül. A `while` utasítást használjuk ilyen helyzetben.

középen tesztelő ciklus (middle-test loop) Egy ciklus, amely végrehajtja a ciklus egy részét, aztán ellenőrzi a kilépési feltételt, majd ha kell, végrehajtja a törzs további részeit. Erre nincs speciális Python konstrukció, de a `while` és a `break` együttes használatával megoldhatjuk.

kurzor (cursor) Egy láthatatlan jelölő, ami nyomon követi hova lesz írva a következő karakter.

lépésenkénti végrehajtás (single-step) A parancsértelmező futásának olyan módja, ahol képes vagy az egyes utasítások külön-külön történő végrehajtása között a végrehajtás következményeinek vizsgálatára. Hasznos a debugoláshoz és egy mentális modell felállításához arról, mi is folyik éppen.

meta-jelölés (meta-notation) Extra szimbólumok vagy jelölések, amelyek segítenek leírni más jelöléseket. Mi bevezettük a szögletes zárójelet, a tripla-pont, a félkövér vagy a dőlt jelölést, hogy segítsen leírni az opcionálitást, az ismételhetőséget, a helyettesíthetőséget és a fix szintaxisrészeket Pythonban.

nyomkövetés (trace) A program végrehajtásának követése manuálisan, feljegyezve a változók állapotainak változását és minden előállított kimenetet.

számláló (counter) Egy változó valaminek a megszámlálásához, gyakran nullával van inicializálva és inkrementálva van a ciklus törzsben.

tabulátor (tab) Egy speciális karakter, ami a kurzort néhány pozícióval jobbra (a tab stopig) mozgatja az aktuális sorban.

törzs (body) Utasítások a cikluson belül.

töréspont (breakpoint) Egy hely a programkódban, ahol a programvégrehajtás szünetel (vagy megtörik), lehetővé téve számodra, hogy megvizsgáld a programváltozók állapotát, egyenként hajtsd végre az utasításokat.

trichotómia (trichotomy) Adott a és b valós számok esetén, a következő relációk közül pontosan az egyik áll fenn: $a < b$, $a > b$ vagy $a == b$. Így, amikor rájössz, hogy két reláció hamis, feltételezheted, hogy a harmadik igaz lesz.

új sor karakter (newline) Egy speciális karakter, ami a kurzort a következő sor elejére viszi.

végtelen ciklus (infinite loop) Egy ciklus, amelyben a befejezési feltétel sohasem teljesül.

7.26. Feladatok

Ez a fejezet megmutatta, hogyan összegezhettük egy lista elemeit és hogyan tudjuk megszámlolni őket. A számlási példában egy `if` utasítás is volt, hogy csak a kiválasztott elemekkel dolgozzunk. Az előző fejezetben volt egy `ketbetus_szo_keresese` függvény, ami lehetővé tette a „korai kilépést” a ciklusból a `return` használatával, amikor valamilyen feltétel teljesült. Használhatjuk a `break` utasítást is ciklusból kilépésre (de nem kilépve a függvényből) és `continue` utasítást a ciklus hátralévő részének elhagyásához a ciklusból való kilépés nélkül.

A listák bejárásának, összegzésének, számlálásának, tesztelésének lehetősége és a korai kilépés építőkövek egy gazdag kollekcióját biztosítják, amelyek hatékonyan kombinálhatóak sok különböző függvény létrehozásához.

Az első hat feladat tipikus függvény, amelyet képes kell legyél megírni csak ezeknek az építőelemeknek a használatával.

1. Írj egy függvényt, ami megszámlolja hány páratlan szám van egy listában!
2. Add össze az összes páros számot a listában!
3. Összegezd az összes negatív számot a listában!
4. Számold meg hány darab 5 betűs szó van egy listában!
5. Összegezd egy lista első páros száma előtti számokat! (Írd meg az egységtesztet! Mi van, ha nincs egyáltalán páros szám?)
6. Számold meg, hány szó szerepel egy listában az első „nem” szóig (beleértve magát a „nem” szót is! (Írd meg itt is az egységtesztet! Mi van, ha a „nem” szó egyszer sem jelenik meg a listában?)
7. Adj egy `print` függvényt a Newton-féle `gyok` függvényhez, amely kiírja a `jobb` változó értékét minden cikluslépésben! Hívd meg a módosított függvényt a 25 aktuális paraméterrel, és jegyezd fel az eredményt!
8. Kövesd nyomon a `szorzotabla_kiiras` függvény legutóbbi változatát és találd ki, hogyan működik!
9. Írj egy `haromszogszamok` nevű függvényt, amely kiírja az első n darab háromszögszámot! A `haromszogszamok(5)` hívás ezt a kimenetet eredményezi:

1	1
2	3
3	6
4	10
5	15

(Segítség: keress rá a neten, mik azok a háromszögszámok!)

10. Írj egy `prim_e` függvényt, amely kap egy egészet paraméterként és `True` értéket ad vissza, ha a paramétere egy *prímszám* és `False` értéket különben! Adj hozzá ilyen teszteket:

```
teszt(prim_e(11))
teszt(not prim_e(35))
teszt(prim_e(19981121))
```

Az utolsó szám jelentheti a születési dátumodat. Prím napon születted? 100 hallgató évfolyamában, mit gondolsz hány prím napon született hallgató van?

11. Emlékezz a részeg kalóz problémára a 3. fejezet feladataiból! Most a részeg kalóz tesz egy fordulatot és pár lépést előre, majd ezt ismételi. A bölcsészhallgatónk feljegyzi a mozgás adatpárjait: az elfordulás szöge és az ezt követő lépések száma. A kísérleti adatai ezek: [(160, 20), (-43, 10), (270, 8), (-43, 12)]. Használj egy teknőcöt a pityókás barátunk útvonalának megjelenítéséhez!
12. Sok érdekes alakzat kirajzolható a teknőcökkel, ha a fentihez hasonló adatpárokat adunk nekik, ahol az első érték egy szög a második pedig egy távolság. Készítsd el az értékpár listát, és rajzold ki a teknőccel az alább bemutatott házat! Ez elkészíthető anélkül, hogy egyszer is felemelnénk a tollat vagy egy vonalat duplán rajzolnánk.



13. Nem minden alakzat olyan, mint a fenti, azaz nem rajtolható meg tollfelemelés vagy dupla vonal nélkül. Melyek rajzolhatóak meg?



Most olvasd el Wikipédia cikkét (<https://hu.wikipedia.org/wiki/Euler-k%C3%B6r>) az Euler-köréről. Tanuld meg, hogyan lehet megmondani azonnal, hogy vajon van-e megoldás vagy nincs. Ha létezik útvonal, akkor azt is tudni fogod, hol kell elkezdni a rajzot és hol ér véget.

14. Mit fog a `szamjegy_szam(0)` függvényhívás visszaadni? Módosítsd, hogy 1-et adjon vissza ebben az esetben! Miért okoz a `szamjegy_szam(-24)` hívás végtelen ciklust? (Segítség: *-1//10 eredménye -1*) Módosítsd a `szamjegy_szam` függvényt, hogy jól működjön bármely egész szám esetén! Add hozzá ezeket a teszteket:

```
teszt(szamjegy_szam(0) == 1)
teszt(szamjegy_szam(-12345) == 5)
```

15. Írj egy `paros_szamjegy_szam(n)` függvényt, amely megszámolja a páros számjegyeket az `n` számban. Ezeken a teszteken át kell mennie:


```
teszt(paros_szamjegy_szam(123456) == 3)
teszt(paros_szamjegy_szam(2468) == 4)
teszt(paros_szamjegy_szam(1357) == 0)
teszt(paros_szamjegy_szam(0) == 1)
```

16. Írj egy `negyzetosszeg(xs)` függvényt, amely visszaadja a paraméterként kapott listában szereplő számok négyzetének összegét! Például a `negyzetosszeg([2, 3, 4])` hívás eredménye $4+9+16$, azaz 29 kell legyen.

```
teszt(negyzetosszeg([2, 3, 4]) == 29)
teszt(negyzetosszeg([ ]) == 0)
teszt(negyzetosszeg([2, -3, 4]) == 29)
```

17. Te és a barátod csapatként írtatok egy kétszemélyes játékot, melyben a játékos a gép ellen játszhat például Tic-Tac-Toe játékot! A barátod írja meg az egyetlen játszma logikáját, míg te megírod a többször játszásért, a pontok tárolásáért, a kezdőjátékos eldöntéséért felelős kódrészletet. Ketten beszéljétek meg, hogy fog a két programrész összeilleni:

```
1  # A barátod befejezi ezt a függvényt
2  def egy_jatszma(felhasznalo_kezd):
3      """
4          A játék egy játszmája. Ha a paraméter True, akkor az ember
5          kezdi a játszmát, különben a számítógép kezd.
6          A játék végén az alábbi értékek egyikével tér
7      vissza
8          -1 (felhasználó nyert), 0 (döntetlen), 1 (a gép
9      nyert).
10     """
11     # Ez egy kamu váz ebben a pillanatban...
12     import random # Lásd a Modulok fejezetet...
13     veletlen = random.Random()
14     # Véletlen szám -1 és 1 között
15     eredmeny = veletlen.randrange(-1,2)
16     print("Felhasználó kezd={0}, Nyertes={1} ".format(felhasznalo_kezd, eredmeny))
17     return eredmeny
```

- (a) Írj egy főprogramot, amely ismételten meghívja ezt a függvényt egy játszma lejátszásához és utána mindig bejelenti a kimenetet: „Én nyertem!”, „Te nyertél!” vagy „Döntetlen!”. Aztán a program megkérdezi: „Akarsz még egyet játszani?” Majd vagy újra indítja a játékot vagy elköszön és befejeződik.
- (b) Tárold hányszor nyertek az egyes játékosok (a felhasználó vagy a gép) valamint hányszor volt döntetlen! Minden játszma után írasd ki a pontokat!
- (c) Adj a programhoz olyan részt, amely biztosítja, hogy a játékosok felváltva kezdjenek!
- (d) Számoldtasd ki, hány százalékban nyerte meg a játszmákat a felhasználó! Ezt minden kör végén írasd ki!
- (e) Rajzold meg a logikád folyamatábráját!

8. fejezet

Sztringek

8.1. Összetett adattípusok

A könyv korábbi részében megismerkedtünk az `int`, `float`, `bool` és az `str` típussal, illetve foglalkoztunk a listákkal és az értékpárokkal is. A sztringek, listák és párok jellegi eltérést mutatnak a többi típushoz képest, ezek ugyanis kisebb részekből épülnek fel. Például a sztringek építőelemei egyetlen **karaktert** tartalmazó sztringek.

A több, kisebb részből összeálló típusokat **összetett (adat)típusoknak** nevezzük. Az összetett értékeket kezelhetjük egyetlen egységként is, de a részeihez is hozzáférhetünk, attól függően, hogy mi a célunk. Ez a kettősség igen praktikus.

8.2. Sztringek kezelése egy egységként

Az előző fejezetekben láthattuk, hogy minden egyes teknőcpéldány saját attribútumokkal és rá alkalmazható metódusokkal rendelkezik. Beállíthattuk például a teknőcök színét, és írhattunk olyat, hogy `Eszter.left(90)`.

A sztringek is objektumok, akár a teknőcök, tehát minden egyes sztring példánynak vannak saját attribútumai és metódusai.

Az alábbi kódrészletben az `upper` például egy metódus:

```
1 ss = "Helló, Világ!"
2 tt = ss.upper()
3 print(tt)
```

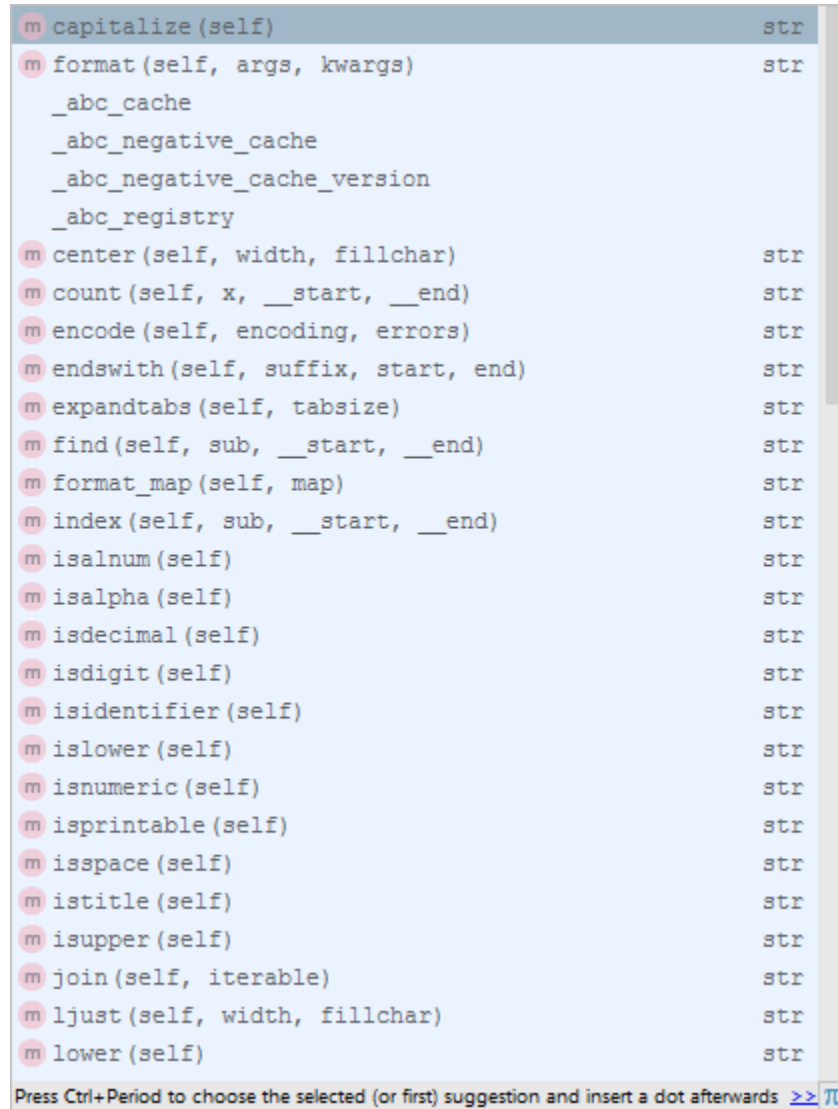
Az `upper` bármely sztring objektumra meghívva egy új sztring objektumot állít elő, amelyben már minden karakter nagybetűs, tehát a kódrészlet a `HELLÓ, VILÁG!` üzenetet jeleníti meg. (Az eredeti `ss` változatlan marad.)

Létezik sok más érdekes függvény is. A `lower` metódus például a sztring kisbetűs, a `capitalize` a sztring nagy kezdőbetűs, de egyébként kisbetűs változatát hozza létre, míg a `swapcase` egy olyan sztring példányt ad vissza, melyben az eredeti sztring kisbetűiből nagybetűk, a nagybetűiből kisbetűk lesznek.

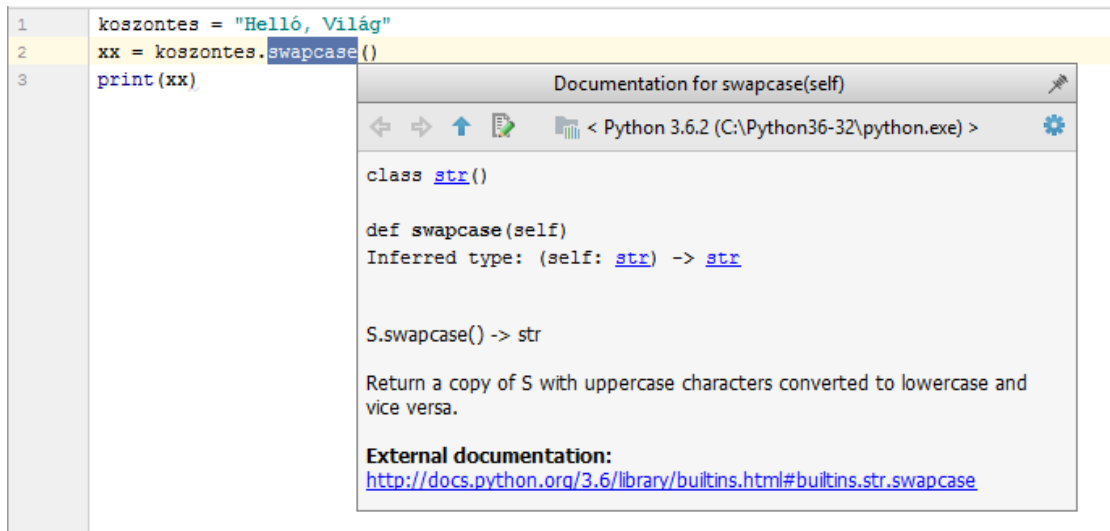
Ha szeretnéd megtudni, milyen metódusok érhetők el, akkor olvasd el a Python dokumentáció `string` modulról szóló részét, vagy – lustább megoldásként – gépeled be egy PyCharm szkriptbe a következőt:

```
1 ss = "Helló, Világ!"
2 xx = ss.
```

Amint a pontot kiteszed a PyCharm megnyit egy felugró ablakot, amelyben az `ss` sztring összes olyan metódusa látszik, amelyet rá alkalmazhatsz (a metódusok neve előtt egy kis «m» áll). Körülbelül 70 van. Hála az égnek, mi csak néhányat fogunk használni!



A felugró ablakban az is látszik, hogy az egyes metódusok milyen paramétereket várnak. Amennyiben további segítségre van szükséged, akkor a metódus nevére állva nyomd le a `Ctrl+Q` billentyűkombinációt a függvényhez tartozó leírás megjelenítéséhez. Ez egy jó példa arra, hogy egy fejlesztői eszköz, a PyCharm, hogyan használja fel a modul készítői által nyújtott meta-adatokat, vagyis a dokumentációs sztringeket.



8.3. Sztringek kezelése részenként

Az **indexelő operátor** egyetlen karakterből álló részsstringet jelöl ki egy sztringből. Pythonban az index mindig szögletes zárójelek között áll:

```
1 gyumolcs = "banán"
2 m = gyumolcs[1]
3 print(m)
```

A `gyumolcs[1]` kifejezés a `gyumolcs` változóban álló sztring 1-es indexű karakterét jelöli ki. Készít egy új sztringet, amely csak a kiválasztott karaktert tartalmazza. Az eredményt az `m` változóba mentjük.

Az `m` megjelenítésnél érhet bennünket némi meglepetés:

```
a
```

Az informatikusok mindig nullától számolnak! Mivel a "banán" sztring 0. pozícióján a `b` betű áll, az 1. pozíción az `a` betűt találjuk.

Ha a nulladik betűt kívánjuk elérni, csak írjunk egy 0-t, vagy bármilyen kifejezést, ami kiértékelve 0-át ad, a szögletes zárójelek közé:

```
1 m = gyumolcs[0]
2 print(m)
```

Most már az alábbi kimenetet kapjuk:

```
b
```

A zárójelben lévő kifejezést **indexnek** nevezzük. Az index adja meg, hogy egy rendezett gyűjtemény elemei – jelen esetben a sztring karakterei – közül melyik elemet kívánjuk elérni. Tetszőleges egész kifejezés lehet.

Az indexeket megjeleníthetjük az `enumerate` függvény segítségével:

```
1 gyumolcs = "banán"
2 lista = list(enumerate(gyumolcs))
3 print(lista)
```

Az eredményül kapott lista (index, karakter) értékpárokat tartalmaz:

```
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'á'), (4, 'n')]
```

Az `enumerate` függvény miatt ne aggódj, a listáknál majd foglalkozunk vele.

Az index operátor egy *sztringet* ad vissza. Pythonban nincs külön típus a karakterek tárolására, ezek 1 hosszúságú sztringek.

Az előző fejezetekben listákkal is találkoztunk már. Az indexelő operátor segítségével a listák egyes elemeit is elérhetjük. A jelölés ugyanaz, mint a sztringeknél:

```
1 primek = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
2 p4 = primek[4]
3 print(p4)
4
5 barátok = ["Misi", "Petra", "Botond", "Jani", "Csilla", "Peti", "Norbi"]
6 b3 = barátok[3]
7 print(b3)
```

A szkript kimenete az alábbi:

```
11
Jani
```

8.4. Hossz

A `len` függvénnyel meghatározható a sztringben álló karakterek száma, amely az alábbi példában 5:

```
1 gyumolcs = "banán"
2 hossz = len(gyumolcs)
3 print(hossz)
```

Ha az utolsó betűt szeretnénk elérni, ígéretesnek tűnhet az alábbi kódrészlet:

```
1 sz = len(gyumolcs)
2 utolso = gyumolcs[sz]           # HIBA!
```

Nos, ez nem fog működni. Futási hibát okoz (`IndexError: string index out of range`), ugyanis a "banán" sztring 5. pozícióján nem áll semmilyen karakter. A számozás 0-tól indul, tehát az indexek 0-tól 4-ig vannak számozva. Az utolsó karakter eléréséhez le kell vonnunk 1-et a `gyumolcs` változóban tárolt szöveg hosszából:

```
1 sz = len(gyumolcs)
2 utolso = gyumolcs[sz-1]
3 print(utolso)
```

Egy másik megoldási lehetőség, ha **negatív indexet** alkalmazunk. A negatív indexek számozása a sztring végétől indul -1-es értékkel. A `gyumolcs[-1]` az utolsó karaktert, a `gyumolcs[-2]` az utolsó előtti (hátról a másodikat) adja meg, és így tovább.

Valószínűleg már kitaláltad, hogy a negatív indexelés a listák esetében is hasonlóan működik.

A könyv további részében nem fogjuk alkalmazni a negatív indexeket. Feltehetően jobban jársz, ha kerülöd a használatát, ugyanis nem sok programozási nyelv enged meg ilyesmit. Az interneten viszont rengeteg olyan Python kód van, ami használja ezt a trükköt, ezért nem árt tudni a létezéséről.

8.5. Bejárás és a `for`

Igen sok olyan számítási probléma van, ahol a sztring karaktereit egyesével dolgozzuk fel. Általában a sztring elejétől indul a folyamat: vesszük a soron következő karaktert, csinálunk vele valamit, és ezt a két lépést ismétljük a sztring végéig. Az ilyen feldolgozási módot **bejárásnak** hívjuk.

A bejárást megvalósíthatjuk például `while` utasítás segítségével:

```
1 i = 0
2 while i < len(gyumolcs):
3     karakter = gyumolcs[i]
4     print(karakter)
5     i += 1
```

Ez a ciklus a sztringet járja be, miközben minden egyes karakterét külön-külön sorba megjeleníti. A ciklusfeltétel az `i < len(gyumolcs)`. Amikor az `i` értéke a sztring hosszával azonos, akkor a feltétel hamis, tehát a ciklus törzs nem hajtódik végre. A legutoljára feldolgozott karakter a `len(gyumolcs) - 1` pozíción áll, vagyis a sztring utolsó karaktere.

Na de korábban már láttuk, hogy a `for` ciklus milyen könnyen végig tudja járni egy lista elemeit. Sztringekre is működik:

```
1 for c in gyumolcs:
2     print(c)
```

A `c` változóhoz minden egyes cikluslépésnél hozzárendelődik a sztring következő karaktere, egészen addig, ameddig el nem fogynak a karakterek. Itt láthatjuk a `for` ciklus kifejező erejét a `while` ciklussal történő sztring bejárással szemben.

A következő kódrészlet az összefűzésre ad példát, és megmutatja, hogyan használható a `for` ciklus egy ábécérendben álló sorozat generálásához. Mindehhez Peyo (Pierre Culliford) híres rajzfilmsorozatának, a Hupikék törpikéknek a szereplőit, Törpapát, Törperőst, Törpicurt, Törpköltőt, Törpmorgót, Törpöltöt és Törpszakállt hívjuk segítségül. Az alábbi ciklus az ő neveiket listázza ki betűrendben:

```
1 elotag = "Törp"
2 utotagok_listaja = ["erős", "költő", "morgó", "öltő", "papa", "picur",
3     ↪ "szakáll" ]
4
5 for utotag in utotagok_listaja:
6     print(elotag + utotag)
```

A program kimenete az alábbi:

```
Törperős
Törpköltő
Törpmorgó
Törpöltő
Törppapa
Törppicur
Törpszakáll
```

Természetesen ez nem teljesen jó, ugyanis Törpapa és Törpicur neve hibásan szerepel. Majd a fejezet végén szereplő feladatok megoldása során kijavítod.

8.6. Szeletelés

Szeletnek vagy *részsztringnek* nevezzük a sztring azon részét, melyet annak **szeletelésével** kaptunk. A szeletelést listákra is alkalmazhatjuk, hogy megkapjuk az elemek egy *részlistáját*. A könnyebb követhetőség érdekében ezúttal a sorok mellett álló kommentben adjuk meg a kimenetet:

```
1 s = "A Karib-tenger kalózzai"
2 print(s[0:1])          # A
3 print(s[2:14])         # Karib-tenger
4 print(s[15:22])        # kalózzai
5 barátok = ["Misi", "Petra", "Botond", "Jani", "Csilla", "Peti", "Norbi"]
6 print(barátok[2:4])    # ['Botond', 'Jani']
```

Az operátor `[n:m]` visszaadja a sztring egy részét az `n`. karakterétől kezdve az `m`. karakterig, az `n`. karaktert beleértve, az `m`-et azonban nem. Ha az indexeket a karakterek *közé* képzeled, ahogy azt az ábrán látod, akkor logikusnak fogod találni:

gyumolcs →	"	b	a	n	á	n	"
indexek:	0	1	2	3	4	5	

Képzeld el, hogy ez egy papírdarab. Az `[n:m]` szeletelés az `n`. és az `m`. vonal közti papírdarabot másolja ki. Ha a megadott `n` és `m` is a sztringen belül van, akkor az új sztring hossza $(m-n)$ lesz.

Trükkök a szeleteléshez: Ha elhagyjuk az első indexet (a kettőspont előtt), akkor a sztring elejétől induló részletet másolunk ki. Ha elhagyjuk a második indexet, illetve ha a sztring hosszával egyenlő vagy annál nagyobb értéket adunk az `m`-nek, akkor a szeletelés a sztring végéig tartó részt adja meg. (A szeletelés, szemben az indexelő operátorral, nem ad `index out of range` hibaüzenetet, ha túlmegyünk a tartományon.) Tehát:

```
1 gyumolcs = "banán"
2 gy = gyumolcs[:3]
3 print(gy)          # ban
4 gy = gyumolcs[3:]
5 print(gy)          # án
6 gy = gyumolcs[3:999]
7 print(gy)          # án
```

Mit gondolsz, mi lesz az `s[:]` és a `barátok[4:]` eredménye?

8.7. Sztringek összehasonlítása

Lássuk a sztringeken dolgozó összehasonlító operátorokat. Két sztring egyenlőségét az alábbi módon ellenőrizhetjük:

```
1 if szo == "banán":
2     print("Nem, nincs banánunk!")
```

Más összehasonlító operátorok a szavak *lexikografikus* sorrendbe való rendezésére is alkalmasak:

```
1 if szo < "banán":
2     print("A szavad, a(z) " + szo + ", a banán elé jön.")
3 elif szo > "banán":
4     print("A szavad, a(z) " + szo + ", a banán után jön.")
5 else:
6     print("Nem, nincs banánunk!")
```

A lexikografikus sorrend a szótárakban lévő *alfabetikus* sorrendhez hasonlít, azonban az összes nagybetű a kisbetűk előtt áll. Valahogy így:

```
A szavad, a(z) Zebra, a banán elé jön.
```

A probléma kezelésére bevett szokás a sztringek egységes formára való átalakítása, például kisbetűsítése, az összehasonlítás előtt. Nagyobb kihívás lesz annak megoldása, hogy a program rájöjjön, a zebra nem is gyümölcs.

Ha alaposan teszteljük a fenti kódrészletet, más meglepetés is érhet minket:

```
A szavad, a(z) áfonya, a banán után jön.
```

A probléma orvosolásához állítsuk be a, hogy milyen nyelvet és milyen karakterkódolást kívánunk használni, majd hasonlítsuk össze a sztringeket egy olyan függvény, az `strcoll(s1, s2)` segítségével, amely figyelembe veszi ezt a beállítást. A függvény nullát ad eredményül, ha a hasonlítandó sztringek egyformák, -1 értéket ad, ha az `s1` paraméternek átadott sztring betűrendben előrébb áll, mint az `s2`-nek átadott sztring, különben pedig +1 értéket ad. Az előző példánkat átírva, az alábbi kódrészletet kapjuk:

```
1 import locale
2
3 szo = input("A szavad: ")
4 locale.setlocale(locale.LC_ALL, "HU_hu.UTF8") # a nyelv és a kódolás_
   ↳ beállítása
5
6 k = locale.strcoll(szo, "banán") # a két sztring összehasonlítása
7 if k < 0:
8     print("A szavad, a(z) " + szo + ", a banán elé jön.")
9 elif k > 0:
10    print("A szavad, a(z) " + szo + ", a banán után jön.")
11 else:
12    print("Nem, nincs banánunk!")
```

8.8. A sztringek módosíthatatlanok

Ha egy sztring egy karakterét szeretnénk megváltoztatni, kézenfekvőnek tűnhet az indexelő operátort (`[]`) egy értékadás bal oldalára írni. Valahogy így:

```
1 koszontes = "Helló, Világ!"
2 koszontes[0] = 'J' # HIBA!
3 print(koszontes)
```

Az eredmény a `Jelló, Világ!` helyett egy futási hiba (`TypeError: 'str' object does not support item assignment`), amely jelzi számunkra, hogy az `str` objektumok elemeihez nem rendelhető érték.

A sztringek **módosíthatatlanok**, vagyis a meglévő sztringet nem változtathatjuk meg. Annyit tehetünk, hogy létrehozunk a módosítani kívánt sztring egy új változatát:

```
1 koszontes = "Helló, Világ!"
2 uj_koszontes = 'J'+ koszontes[1:]
3 print(uj_koszontes)
```

Az itt álló megoldás az új első betűt és a `koszontes` változóban lévő sztring egy szeletét fűzi össze. Az eredeti sztringre nincs hatással a művelet.

8.9. Az `in` és a `not in` operátor

Az `in` operátorral tartalmazást ellenőrizhetünk. Ha az operátor mindkét operandusa sztring, akkor azt adja meg, hogy az `in` jobb oldalán álló sztring tartalmazza-e a bal oldalán álló sztringet.

```
1 szerepel_e = "m" in "alma"
2 print(szerepel_e)           # True
3 szerepel_e = "i" in "alma"
4 print(szerepel_e)           # False
5 szerepel_e = "al" in "alma"
6 print(szerepel_e)           # True
7 szerepel_e = "la" in "alma"
8 print(szerepel_e)           # False
```

Fontos megjegyezni, hogy egy sztring részsstringjeinek halmazába saját maga és az üres sztring is beletartozik. (Mint ahogy azt is, hogy a programozók mindig szeretik alaposan átgondolni a szélsőséges eseteket!) Az alábbi kifejezések mindegyikének `True` az értéke:

```
1 "a" in "a"
2 "alma" in "alma"
3 "" in "a"
4 "" in "alma"
```

A `not in` operátor az `in` operátor logikai ellentétét adja meg, ezért a következő kifejezés is `True` értékű.

```
1 "x" not in "alma"
```

Az `in` és az összerakás (+) operátorok alkalmazásával már egy olyan függvényt is el tudunk készíteni, amely az összes magánhangzót eltávolítja egy szövegből:

```
1 def maganhangzo_torles(s):
2     maganhangzok = "aáééííóóőőuúüüAÁÉÉÍÍÓÓŐŐUÚÜÜ"
3     massalhangzos_s = ""
4     for k in s:
5         if k not in maganhangzok:
6             massalhangzos_s += k
7     return massalhangzos_s
8
9 teszt(maganhangzo_torles("informatika") == "nfrmtk")
10 teszt(maganhangzo_torles("aábeéíífoóőőuúüüPÁÁÉÉÍÍÓÓŐŐUÚÜÜs") == "bfjps")
```

8.10. Egy kereses függvény

Mit csinál az alábbi függvény?

```
1 def kereses(szoveg, k):
2     """
3     Megkeresi a k karaktert a szövegben (szoveg) és visszatér annak_
4     ↪ indexével.
5     A visszatérési érték -1, ha a k karakter nem szerepel a szövegben.
6     """
7     i = 0
8     while i < len(szoveg):
9         if szoveg[i] == k:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
9         return i
10        i += 1
11    return -1
12
13    teszt(kereses("Informatika", "o") == 3)
14    teszt(kereses("Informatika", "I") == 0)
15    teszt(kereses("Informatika", "a") == 6)
16    teszt(kereses("Informatika", "x") == -1)
```

A kereses függvény bizonyos értelemben az indexelő operátor ellentéte. Míg az indexelő operátorral a sztring egy adott indexén álló karakterét érhetjük el, a kereses függvény egy megadott karakter alapján adja meg, hogy az melyik indexen áll a sztringen belül. Ha a karakter nem található, akkor `-1`-et ad vissza a függvény.

Láthatunk egy újabb példát a `return` utasítás cikluson belül való alkalmazására is. Ha a `szoveg[i] == k`, akkor a függvény azonnal befejezi működését, idő előtt megszakítva a ciklus végrehajtását.

Amennyiben a karakter nem szerepel a sztringben, a program a normális módon lép ki a ciklusból, és `-1`-et ad vissza.

A fenti programozási minta hasonlóságot mutat a *rövidzár kiértékeléssel*, hiszen azonnal befejezzük a munkát, amint megismerjük az eredményt, az esetleges hátralévő részek feldolgozása nélkül. Találó név lehetne erre az algoritmusra a *Heuréka bejárás*, mivel ha ráleltünk a keresett elemre, már kiálthatjuk is: „Heuréka!”. Leggyakrabban azonban **teljes keresés** néven fogsz találkozni vele.

8.11. Számlálás ciklussal

A következő program az a betűk előfordulásának számát határozza meg egy sztringenben. Ez lesz a második példánk, ahol a *Számjegyek számlálása* részben ismertetett számlálás algoritmusát használjuk.

```
1 def a_betuk_szama(szoveg):
2     darab = 0
3     for k in szoveg:
4         if k == "a":
5             darab += 1
6     return darab
7
8 teszt(a_betuk_szama("banán") == 1)
```

8.12. Opcionális paraméterek

A kereses függvény könnyen módosítható úgy, hogy egy karakter második, harmadik, stb. előfordulását is megtalálhassuk egy sztringben. Egészítsük ki a függvény fejlécét egy harmadik paraméterrel, amely a keresés sztringen belüli kezdőpontját határozza meg:

```
1 def kereses2(szoveg, k, kezdet):
2     i = kezdet
3     while i < len(szoveg):
4         if szoveg[i] == k:
5             return i
6         i += 1
7     return -1
8
9 teszt(kereses2("banán", "n", 2) == 2)
```

A `kereses2("banán", "n", 2)` függvényhívás 2-t ad vissza, ugyanis az "n" karakter a 2. pozíción fordul elő először a "banán" sztringben. Mi lenne a `kereses2("banán", "n", 3)` hívás eredménye? Ha a válaszdod 4, akkor valószínűleg sikeresen megértetted a `kereses2` működését.

Még jobb, ha a `kereses` és a `kereses2` függvényeket egybeépítjük egy **opcionális paramétert** alkalmazva:

```
1 def kereses(szoveg, k, kezdet = 0):
2     i = kezdet
3     while i < len(szoveg):
4         if szoveg[i] == k:
5             return i
6         i += 1
7     return -1
```

Az opcionális paraméternek a függvény hívója adhat át argumentumot, de nem kötelező megtennie. Ha adott át 3. argumentumot a hívó, akkor az a szokásos módon hozzárendelődik a `kereses` függvény `kezdet` paraméteréhez. Máskülönbén a `kezdet` paraméter **alapértelmezett értéket** kap, jelen esetben 0-át a függvénydefinícióban szereplő, `kereses=0` hatására.

Szóval a `kereses("banán", "n", 2)` hívás esetében a `kereses` függvény úgy működik, mint a `kereses2`, a `kereses("banán", "n")` hívásnál viszont 0 **alapértelmezett értéket** kap a `kezdet` paraméter.

Egy újabb opcionális paraméterrel elérhetjük, hogy a keresés az első paraméterben megadott pozícióról induljon, és érjen véget egy második paraméterben megadott pozíció előtt:

```
1 def kereses(szoveg, k, kezdet = 0, veg = None):
2     i = kezdet
3     if veg is None:
4         veg = len(szoveg)
5
6     while i < veg:
7         if szoveg[i] == k:
8             return i
9         i += 1
10    return -1
```

A `veg` opcionális paraméter egy érdekes eset. Amennyiben a hívó nem ad meg számára argumentumot, akkor a `None` rendelődik hozzá, mint alapértelmezett érték. A függvény törzsében a paraméter értéke alapján megállapítjuk, hogy a hívó megadta-e hol érjen véget a keresés. Ha nem adta meg, akkor a `veg` változó értékét felülírjuk a sztring hosszával, különben az argumentumként kapott pozíciót használjuk a ciklusfelvételnél.

A `kezdet` és a `veg` paraméterek ebben a függvényben pontosan azzal a jelentéssel bírnak, mint a beépített `range` függvény `start` és `stop` paraméterei.

Néhány tesztet, amelyen a függvénynek át kell mennie:

```
1 ss = "Érdekes metódusai vannak a Python sztringeknek."
2 teszt(kereses(ss, "e") == 3)
3 teszt(kereses(ss, "e", 5) == 5)
4 teszt(kereses(ss, "e", 6) == 9)
5 teszt(kereses(ss, "e", 18, 34) == -1)
6 teszt(kereses(ss, ".") == len(ss) - 1)
```

8.13. A beépített `find` módszer

Most, hogy már túl vagyunk egy hatékony kereső függvény, a `kereses` megírásának kemény munkáján, elárulhatjuk, hogy a sztringeknek van egy beépített módszere erre a célra, a `find`. Mindenre képes, amire a saját függvényünk, sőt

még többre is!

```
1  teszt(ss.find("e") == 3)
2  teszt(ss.find("e", 5) == 5)
3  teszt(ss.find("e", 6) == 9)
4  teszt(ss.find("e", 18, 34) == -1)
5  teszt(ss.find(".") == len(ss) - 1)
```

A beépített `find` metódus általánosabb, mint a mi verziónk, ugyanis sztringet is kereshetünk vele egy sztringben, nem csak karaktert:

```
1  hol_van = "banán".find("nán")
2  print(hol_van)
3  hol_van = "papaja".find("pa", 1)
4  print(hol_van)
```

Mindkét esetben 2-es indexet ad vissza a `find` metódus. (Továbbra is 0-tól számozzunk.)

Többnyire a Python által biztosított beépített függvények használatát javasoljuk a saját változataink helyett. Ugyanakkor számos beépített függvény és metódus van, amelyek újrírásával nagyszerűen lehet tanulni. A mögöttes megoldások, technikák elsajátításával olyan „építőkövek” kerülnek a kezédbe, amelyek segítenek majd képzett programozóvá válni.

8.14. A `split` metódus

A `split` a sztringek egyik leghasznosabb metódusa, ugyanis a több szóból álló sztringeket szavak listájává alakítja át. A szavak közt álló *whitespace* karaktereket (szóközöket, tabulátorokat, újsor karaktereket) eltávolítja. Ez a függvény lehetővé teszi, hogy egyetlen sztringként olvassunk be egy inputot, és utólag bontsuk szavakra.

```
1  ss = "Nos én sose csináltam mondtá Alice"
2  szavak = ss.split()
3  print(szavak)
```

A kódrészlet hatására az alábbi lista jelenik meg:

```
['Nos', 'én', 'sose', 'csináltam', 'mondtá', 'Alice']
```

8.15. A sztringek tisztítása

Gyakran dolgozunk olyan sztringekkel, amelyek különböző írásjeleket, tabulátorokat, vagy újsor karaktert tartalmaznak. Egy későbbi fejezetben tapasztalni is fogjuk ezt, amikor már internetes honlapokról szedjük le, vagy fájlokból olvassuk fel a feldolgozni kívánt szövegeket. Ha azonban olyan programot írunk, ami a szavak gyakoriságát határozza meg, vagy az egyes szavak helyesírását ellenőrzi, akkor előnyösebb megszabadulni ezektől a nemkívánatos karakterektől.

Az alábbiakban mutatunk egy példát arra, hogyan távolíthatók el a különféle írásjelek a sztringekből. A sztringek ugye módosíthatatlanok, ezért nem változtathatjuk meg az eredeti sztringet. Bejárjuk a sztringet, és egy új sztringet hozunk létre a karakterekből az írásjeleket kihagyva:

```
1  irasjelek = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
2
3  def irasjel_eltavolitas(szoveg):
4      irasjel_nelkuli = ""
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5     for karakter in szoveg:
6         if karakter not in irasjelek:
7             irasjel_nelkuli += karakter
8     return irasjel_nelkuli
```

Az első értékadás kissé kaotikus, könnyen hibához vezethet. Szerencsére a Python `string` modulja definiál egy `sztring` konstanst, mely tartalmazza az írásjeleket. A programunk javított változatának elkészítéséhez importáljuk a `sztring` modult, és használjuk fel az ott megadott definíciót.

```
1 import string
2
3 def irasjel_eltavolitas(szoveg):
4     irasjel_nelkuli = ""
5     for karakter in szoveg:
6         if karakter not in string.punctuation:
7             irasjel_nelkuli += karakter
8     return irasjel_nelkuli
9
10
11 teszt(irasjel_eltavolitas('"Nos, én sose csináltam!" - mondta Alice') ==
12      "Nos én sose csináltam mondta Alice")
13 teszt(irasjel_eltavolitas("Teljesen, de teljesen biztos vagy benne?") ==
14      "Teljesen de teljesen biztos vagy benne")
```

Az előző részben látott `split` metódus és ennek a függvénynek az egymásba ágyazásával, egy felettebb hatásos kombinációt kapunk. Először eltávolíthatjuk az írásjeleket, majd a `split` segítségével a szöveget szavak listájára bontjuk, megszabadulva egyúttal az újsor karaktertől és a tabulátoroktól is:

```
1 a_tortenetem = """
2 A pitonok nem méreggel ölnek, hanem kiszorítják a szuszt az áldozatukból.
3 A prédájuk köré tekerik magukat, minden egyes lélegzeténél egy kicsit
4 szorosabban, egészen addig, amíg a légzése abba nem marad. Amint megáll
5 a zsákmány szíve, lenyelik az egészet. A bunda és a tollazat kivételével
6 az egész állat a kígyó gyomrába lesz temetve. Mit gondolsz, mi történik
7 a lenyelt bundával, tollakkal, csőrökkel és tojásbélyekkel? A felesleges
8 'dolgozók' távoznak, -- jól gondoltod -- kígyó ÜRÜLÉK lesz belőlük!"""
9
10 szavak = irasjel_eltavolitas(a_tortenetem).split()
11 print(szavak)
```

A kimenet:

```
['A', 'pitonok', 'nem', '...', 'kígyó', 'ÜRÜLÉK', 'lesz', 'belőlük']
```

Sok más hasznos `sztring` metódus létezik, azonban ez a könyv nem szándékozik referenciakönyv lenni. A *Python Library Reference* viszont az, elérhető a [Python honlapján](#) más dokumentációk társaságában.

8.16. A `sztring` `format` metódusa

Python 3-ban a legkönnyebb és leghatásosabb módja a `sztring`ek formázásának a `format` metódus alkalmazása. Nézzük is meg néhány példán keresztül, hogyan működik:

```
1 s1 = "A nevem {0}!".format("Artúr")
2 print(s1)
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3
4  nev = "Alice"
5  kor = 10
6  s2 = "A nevem {1}, és {0} éves vagyok.".format(kor, nev)
7  print(s2)
8
9  n1 = 4
10 n2 = 5
11 s3 = "2**10 = {0} és {1} * {2} = {3:f}".format(2 ** 10, n1, n2, n1 * n2)
12 print(s3)
```

A szkript futtatása az alábbi kimenetet eredményezi:

```
A nevem Artúr!
A nevem Alice, és 10 éves vagyok.
2**10 = 1024 és 4 * 5 = 20.000000
```

A formázandó sztringbe *helyettesítő mezőket* ágyazhatunk: ... {0} ... {1} ... {2} ..., stb. A `format` metódus a mezőket kicseréli az argumentumként átadott értékekre. A helyettesítő mezőben álló szám, az arra a helyre behelyettesítendő argumentum sorszámát adja meg. Ne menj tovább, ameddig a 6-os sort meg nem érted a fenti kódban.

Van tovább is! A helyettesítő mezőket *formázó mezőknek* is szokás nevezni, utalva arra, hogy mindegyik mező tartalmazhat **formátum leíró**t. A formátum leírókat mindig „:” szimbólum vezeti be, mint a fenti példa 11. sorában. Az argumentumok sztringbe való helyettesítésére vannak hatással. Az alábbiakhoz hasonló formázási lehetőségekkel élhetünk:

- Balra (<), jobbra (>) vagy középre (^) legyen-e igazítva a mező?
- Milyen széles mezőt kell lefoglalni az argumentum számára a formázandó sztringen belül? (pl. 10)
- Történjen-e konverzió? (Eleinte csak float típusú konverziót (f) fogunk alkalmazni, ahogy azt a fenti példa 11. sorában is tettük, esetleg egy egész értéket fogunk hexadecimális alakra hozatni az x-szel).
- Ha a konverzió típusa float, akkor megadható a megjelenítendő tizedesjegyek száma is. (A .2f megfelelő lehet a pénznemek kiírásánál, ami általában két tizedesjegyre érdekes.)

Lássunk most néhány szokványos, egyszerű példát, amikben szinte minden benne van, amire szükségünk lehet. Ha netán valami különlegesebb formázásra van igényed, akkor olvasd el a dokumentációt, amely részletekbe menően tárgyal minden lehetőséget.

```
1  n1 = "Paris"
2  n2 = "Whitney"
3  n3 = "Hilton"
4
5  print("A pi értéke három tizedesjegyre: {0:.3f}".format(3.1415926))
6  print("123456789 123456789 123456789 123456789 123456789 123456789")
7  print("|||{0:<12}|||{1:^12}|||{2:>12}|||Születési év: {3}|||".
8      .format(n1, n2, n3, 1981))
9  print("A {0} decimális érték {0:x} hexadecimális értékékké konvertálódik."
10      .format(123456))
```

A szkript kimenete:

```
A pi értéke három tizedesjegyre: 3.142
123456789 123456789 123456789 123456789 123456789 123456789
|||Paris      |||  Whitney  |||      Hilton|||Születési év: 1981|||
A 123456 decimális érték 1e240 hexadecimális értékékké konvertálódik.
```

Több helyettesítő mező is hivatkozhat ugyanarra az argumentum sorszáma, illetve lehetnek olyan argumentumok is, amelyekre egyetlen mező sem hivatkozik:

```
1 level = """
2 Kedves {0} {2}!
3
4 {0}, van egy rendkívüli üzleti ajánlatom az Ön számára.
5 Amennyiben küld 10 millió dollárt a bankszámlámra, megduplázom a pénzét...
6 """
7
8 print(level.format("Paris", "Whitney", "Hilton"))
9 print(level.format("Bill", "Henry", "Gates"))
```

A program két levelet eredményez:

```
Kedves Paris Hilton!

Paris, van egy rendkívüli üzleti ajánlatom az Ön számára.
Amennyiben küld 10 millió dollárt a bankszámlámra, megduplázom a pénzét...

Kedves Bill Gates!

Bill, van egy rendkívüli üzleti ajánlatom az Ön számára.
Amennyiben küld 10 millió dollárt a bankszámlámra, megduplázom a pénzét...
```

Ha egy helyettesítő mező nem létező argumentum sorszámot tartalmaz, akkor a várakozásoknak megfelelően, index-hibát kapunk:

```
1 "helló {3}".format("Dávid")
2 Traceback (most recent call last):
3   File "<input>", line 1, in <module>
4   IndexError: tuple index out of range
```

A következő példa a sztring formázás igazi értelmét mutatja meg. Először próbáljunk egy táblázatot sztring formázás nélkül kiírni:

```
1 print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
2 for i in range(1, 11):
3     print(i, "\t", i**2, "\t", i**3, "\t", i**5, "\t",
4           i**10, "\t", i**20, sep='')

```

A program az [1; 10] tartományba eső egész számok különböző hatványait jeleníti meg. (A kimenet megadásánál feltételeztük, hogy a tabulátor szélessége 8-ra van állítva. PyCharmon belül az alapértelmezett tabulátorszélesség 4 szököznyi, ezért még ennél is rosszabb kimenetre számíthatsz. A print utasításban a sep=' ' kifejezéssel érhető el, hogy ne kerüljön szököz a kimenetben vesszővel elválasztott argumentumok közé.)

A program e változatában tabulátor karakterekkel (\t) rendeztük oszlopokba az értékeket, de a formázás elcsúszik azoknál a táblázatbeli értékeknél, amelyek több számjegyből állnak, mint amennyi a tabulátor szélessége:

i	i**2	i**3	i**5	i**10	i**20
1	1	1	1	1	1
2	4	8	32	1024	1048576
3	9	27	243	59049	3486784401
4	16	64	1024	1048576	1099511627776
5	25	125	3125	9765625	95367431640625
6	36	216	7776	60466176	3656158440062976
7	49	343	16807	282475249	79792266297612001

(folytatás a következő oldalon)

(folytatás az előző oldalról)

8	64	512	32768	1073741824	1152921504606846976
9	81	729	59049	3486784401	12157665459056928801
10	100	1000	100000	10000000000	10000000000000000000

Egy lehetséges megoldás a tabulátor szélességének átállítása, de az első oszlop már most is szélesebb a kellenénél. A legjobb megoldás a problémára, ha oszloponként állítjuk be a megfelelő szélességet. Valószínűleg nem ér meglepetés-ként, hogy a sztring formázással szebb eredményt érhetünk el. Még jobbra is igazíthatjuk a mezőket:

```
1 elrendezes = "{0:>4}{1:>6}{2:>6}{3:>8}{4:>13}{5:>24}"
2
3 print(elrendezes.format("i", "i**2", "i**3", "i**5", "i**10", "i**20"))
4 for i in range(1, 11):
5     print(elrendezes.format(i, i ** 2, i ** 3, i ** 5, i ** 10, i ** 20))
```

A futtatás után az alábbi, jóval tetszetősebb kimenet jelenik meg:

i	i**2	i**3	i**5	i**10	i**20
1	1	1	1	1	1
2	4	8	32	1024	1048576
3	9	27	243	59049	3486784401
4	16	64	1024	1048576	1099511627776
5	25	125	3125	9765625	95367431640625
6	36	216	7776	60466176	3656158440062976
7	49	343	16807	282475249	79792266297612001
8	64	512	32768	1073741824	1152921504606846976
9	81	729	59049	3486784401	12157665459056928801
10	100	1000	100000	10000000000	10000000000000000000

8.17. Összefoglalás

Ebben a fejezetben nagyon sok új gondolat jött elő. Az alábbi összefoglaló segíthet felidézni a tanultakat. (A fejezet címének megfelelően, most a sztringekre koncentrálnunk.)

indexelés ([]) Egy sztring adott pozícióján álló karakter elérésére használható. Az indexek számozása 0-tól indul. Például "Ennek" [4] az eredménye "k".

A len függvény Egy sztring hosszát, vagyis a benne szereplő karakterek számát állapíthatjuk meg vele. Például a len("boldog") eredménye 6.

bejárás for ciklussal (for) Egy sztring *bejárása* azt jelenti, hogy minden egyes karakterét pontosan egyszer érintjük. Például a

```
for k in "Példa":
    ...
```

ciklus végrehajtása alatt a törzs 5 alkalommal fut le. A k változó minden egyes alkalommal más-más értéket vesz fel.

szeletelés ([:]) A sztringek valamely részének, egy részsstringnek az elérésére szolgál. Például: 'papaja tejszínnel' [0:2] eredménye pa (a 'papaja tejszínnel' [2:4] eredménye is az).

sztringek hasonlítása (>, <, >=, <=, ==, !=) A 6 megszokott hasonlító operátor sztringekre is működik, a *lexikografikus sorrendnek* megfelelően. Példák: Az "alma" < "banán" eredménye igaz (True). A "Zebra" < "Alma" eredménye hamis (False). A "Zebra" <= "kulonos" eredménye is True, ugyanis a lexikografikus sorrend alapján minden nagybetű a kisbetűk előtt áll.

Az in és a not in operátor (in, not in) Az in operátor tartalmazás ellenőrzésére szolgál. Sztringekre alkalmazva megadja, hogy egy sztring szerepel-e egy másik sztringben. Például a "sajt" in "kisajtolom belőle" kifejezés értéke igaz, míg az "ementáli" in "kisajtolom belőle" hamis.

8.18. Szójegyzék

alapértelmezett érték (default value) Az az érték, amit az opcionális paraméter megkap, ha a függvény hívója nem adja meg a hozzá tartozó argumentumot.

bejárás (traverse) Egy kollekció elemein való végighaladás, miközben minden elemre hasonló műveletet hajtunk végre. A bejárás során minden elemet pontosan egyszer érintünk.

dokumentációs sztring (docstring) Egy, a függvény első sorában, vagy a modulok definíciójában (és ahogy később látni fogjuk az osztályok és metódusok definíciójában) álló sztring konstans. Kényelmes megoldást nyújtanak a kód és a dokumentáció összekapcsolására. A dokumentációs sztringeket különböző programozási eszközök is használják interaktív súgó szolgáltatásához.

index Egy változó vagy egy konstans érték, amely egy rendezett sorozat egy elemét jelöli ki, például egy sztring valamely karakterét, vagy egy lista valamely elemét.

minősítés (dot notation) A pont (.) a **minősítés operátora**. Egy objektum attribútumait és metódusait érhetjük el vele.

módosíthatatlan érték (immutable data value) Olyan érték, amely nem változtatható meg. A módosíthatatlan összetett adatoknál, az elemek vagy részek (pl. részsstring) felülírására irányuló kísérlet futási hibát eredményez.

módosítható érték (mutable data value) Olyan érték, amely módosítása lehetséges. Minden módosítható adat összetett típusú. A listák és a szótárak megváltoztathatóak, a sztringek és a rendezett n-esek (tuples) nem.

opcionális paraméter (optional parameter) Olyan, a függvény fejlécében szereplő paraméter, amelyhez egy kezdőérték tartozik. Ha a függvény hívója nem ad át a paraméternek argumentumot, akkor az alapértelmezett érték rendelődik a paraméterhez.

összetett adattípus (compound data type) Olyan típus, amely több komponensből épül fel. A komponensek maguk is típusok. Az összetett típusú értékeket *összetett értékeknek* nevezzük.

rövidzár-kiértékelés (short-circuit evaluation) Egy olyan kifejezés kiértékelési módszer, amely csak addig értékeli ki a kifejezést, ameddig az eredmény el nem dől. A *rövidzár* szóval az olyan programozói stílust is jellemezhetjük, amely megakadályozza az eredmény megismerése utáni felesleges munkavégzést. Például a kereses függvény azonnal visszaadta az eredményt a hívónak, ahogy a keresett karaktert megtaláltuk, nem járta be a sztring még hátra lévő részét.

szelet (slice) A sztring egy adott indextartománnyal meghatározott részletét szeletnek nevezzük. Általánosabban fogalmazva, a szelet egy sorozat olyan részsorozata, amelyet a szeletelő operátor alkalmazásával kaphatunk (sequence[start:stop]).

whitespace Minden olyan karakter, amely arrébb viszi a kurzort anélkül, hogy látható karakter jelenne meg. A string.whitespace konstans tartalmazza az összes white-space karaktert.

8.19. Feladatok

Javasoljuk, hogy egy fájlban készítsd el az alábbi feladatokat, az előző feladatokban látott tesztelő függvényeket is bemásolva a fájlba.

1. Milyen eredményt adnak az alábbi kifejezések? (Ellenőrizd a válaszaid a print függvény segítségével.)

```
"Python"[1]
"A sztringek karaktersorozatok."[5]
len("csodálatos")
"Rejtély"[:4]
"k" in "Körte"
"barack" in "sárgabarack"
"körte" not in "Ananász"
"barack" > "sárgabarack"
"ananász" < "Barack"
```

2. Javítsd ki úgy az alábbi programot, hogy Törpapa és Törpicur neve is helyesen jelenjen meg:

```
1 elotag = "Törp"
2 utotagok_listaja = [erős", "költő", "morgó", "öltő", "papa", "picur",
3   ↪ "szakáll"]
4
5 for utotag in utotagok_listaja:
6     print(elotag + utotag)
```

3. Ágyazd be az alábbi kódrészletet egy karakter_szamlalas nevű függvénybe, majd általánosítsd úgy, hogy a sztringet és a számlálandó karaktert is paraméterként várja. A függvény adja vissza a karakter sztringbeli előfordulásainak számát, ne írassa ki. Az érték megjelenítése a függvény hívójának feladata.

```
1 gyumolcs = "banán"
2 darab = 0
3 for karakter in gyumolcs:
4     if karakter == "a":
5         darab += 1
6 print(darab)
```

4. Most írd át úgy a karakter_szamlalas függvényt, hogy a sztring bejárása helyett a beépített find metódusát hívja meg újra és újra. A második paraméternek átadott értékkel biztosíthatod, hogy a metódus mindig új előfordulását találja meg a számlálandó karakternek.
5. Adj értékül egy bekezdést a kedvenc szövegedből – egy beszédből, egy süteményes receptkönyvből, vagy egy inspiráló versből, stb. – egy változónak. A szöveget tripla idézőjelek közé zárd.

Írj egy függvényt, amely eltávolítja az összes írásjelet a sztringből, és a szöveget szavak listájára bontja. Számold meg, hány olyan szó van a szövegben, melyben szerepel az „e” betű. Jeleníts meg egy alábbihoz hasonló elemzést a szövegedről:

```
A szövegben 243 szó áll, melyből 109 (44.8%) tartalmaz "e" betűt.
```

6. Jeleníts meg egy ilyen szorzótáblát:

	1	2	3	4	5	6	7	8	9	10	11	12
:	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
1:	1	2	3	4	5	6	7	8	9	10	11	12
2:	2	4	6	8	10	12	14	16	18	20	22	24
3:	3	6	9	12	15	18	21	24	27	30	33	36
4:	4	8	12	16	20	24	28	32	36	40	44	48
5:	5	10	15	20	25	30	35	40	45	50	55	60
6:	6	12	18	24	30	36	42	48	54	60	66	72
7:	7	14	21	28	35	42	49	56	63	70	77	84
8:	8	16	24	32	40	48	56	64	72	80	88	96
9:	9	18	27	36	45	54	63	72	81	90	99	108
10:	10	20	30	40	50	60	70	80	90	100	110	120

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
11:    11  22  33  44  55  66  77  88  99 110 121 132
12:    12  24  36  48  60  72  84  96 108 120 132 144
```

7. Írj egy függvényt, amely meghatározza egy paraméterként kapott sztring fordítottját. A függvénynek át kell mennie ezeken a teszteken:

```
1  teszt(sztring_forditas("boldog") == "godlob")
2  teszt(sztring_forditas("Python") == "nohtyP")
3  teszt(sztring_forditas("") == "")
4  teszt(sztring_forditas("a") == "a")
```

8. Írj egy függvényt, amely összefűzi argumentumát annak tükörképével:

```
1  teszt(tukor("jo") == "jooj")
2  teszt(tukor("Python") == "PythohtyP")
3  teszt(tukor("") == "")
4  teszt(tukor("a") == "aa")
```

9. Írj függvényt, amely eltávolítja egy karakter összes előfordulását egy sztringből. A függvény a karaktert és a sztringet is argumentumként várja.

```
1  teszt(betu_eltuntetes("a", "alma") == "lm")
2  teszt(betu_eltuntetes("a", "banán") == "bnán")
3  teszt(betu_eltuntetes("z", "banán") == "banán")
4  teszt(betu_eltuntetes("e", "Kerepes") == "Krps")
5  teszt(betu_eltuntetes("b", "") == "")
6  teszt(betu_eltuntetes("b", "c") == "c")
```

10. Írj függvényt, mely képes a palindromok felismerésére. (Segítség: a korábban megírt sztring_forditas függvény felhasználása megkönnyíti a dolgod!):

```
1  teszt(palindrom_e("abba"))
2  teszt(not palindrom_e("abab"))
3  teszt(palindrom_e("teret"))
4  teszt(not palindrom_e("banán"))
5  teszt(palindrom_e("mesék késem"))
6  teszt(palindrom_e("a"))
7  # teszt(palindrom_e("")) # Egy üres sztring palindrom-e?
```

11. Írj egy függvényt, amely meghatározza, hányszor szerepel egy sztringben egy másik sztring:

```
1  teszt(szamlalas("gö", "görögös") == 2)
2  teszt(szamlalas("pa", "papaja") == 2)
3  teszt(szamlalas("apa", "papaja") == 1)
4  teszt(szamlalas("papa", "papaja") == 1)
5  teszt(szamlalas("apap", "papaja") == 0)
6  teszt(szamlalas("aaa", "aaaaaa") == 4)
```

12. Írj függvényt, amely eltávolítja egy sztringből egy másik sztring első előfordulását:

```
1  teszt(torles("alma", "almafa") == "fa")
2  teszt(torles("an", "banán") == "bán")
3  teszt(torles("pa", "papaja") == "paja")
4  teszt(torles("pa", "Papaja") == "Paja")
5  teszt(torles("alma", "kerékpár") == "kerékpár")
```

13. Írj függvényt, amely eltávolítja egy sztringből egy másik sztring minden előfordulását. (A törlés hatására új előfordulások is keletkezhetnek. Rád bízunk, hogy ezeket eltünteted-e.):

```
1  teszt(alapos_torles("an", "banán") == "bán")
2  teszt(alapos_torles("pa", "papaja") == "ja")
3  teszt(alapos_torles("pa", "Papaja") == "Paja")
4  teszt(alapos_torles("alma", "kerékpár") == "kerékpár")
5  # A megoldástól függően: "pa" vagy ""
6  # teszt(alapos_torles("pa", "ppapaa") == "")
```

9. fejezet

Rendezett n-esek

9.1. Adatcsoportosításra használt rendezett n-esek

Korábban láttuk, hogy összecsoportosíthatunk értékpárokat, ha zárójelekkel vesszük körül őket. Emlékezz erre a példára:

```
1 szuletesi_ev = ("Paris Hilton", 1981)
2 print(szuletesi_ev)
```

Ez egy példa **strukturált adatokra** – egy mechanizmusra az adatok csoportosításához és szervezéséhez az egyszerűbb használat céljából.

A pár egy példa rendezett n-esekre. Ezt általánosítva, a rendezett n-es tetszőleges számú elem csoportosítására használható, hogy egy összetett értéket hozzunk létre. Szintaktikailag ez egy vesszővel elválasztott értéksorozat. Habár nem szükséges, megegyezés szerint kerek zárójelek közé tesszük őket:

```
1 julia = ("Julia", "Roberts", 1967, "Kettős játék", 2009, "színésznő",
2 ↪ "Atlanta, Georgia")
3 print(julia)
```

A rendezett n-esek hasznosak a más nyelveken többnyire *rekordnak* nevezett dolog reprezentálására – ezek összetartozó, egymással kapcsoltban lévő értékek, mint a korábbi hallgató rekord. Nincs leírás arra vonatkozólag, hogy mit jelentenek az egyes mezők, de sejtethjük. Lehetővé teszi, hogy összetartozó dolgokat egyetlen egységként kezeljük.

A rendezett n-esek lehetővé teszik a sztringeknél használt néhány speciális operátor használatát. Az index operátor kiválasztja az egyik elemet.

```
1 print(julia[2]) # 1967
```

Azonban, ha egy elemet fel akarunk használni egy értékadás bal oldalán, hogy megváltoztassuk az értékét, akkor hibát kapunk:

```
1 julia[0] = "X"
```

A hibaüzenet ez:

```
TypeError: 'tuple' object does not support item assignment
```

Mint a sztringek, úgy a rendezett n-esek is megváltoztathatatlanok. Ha egyszer a Python létrehozott egyet a memóriában, nem lehet megváltoztatni.

Természetesen, még ha nem is tudjuk megváltoztatni a rendezett n-es egy elemét, mindig használhatunk egy `julia` nevű változót, amely új információkat tartalmazó rendezett n-esre fog hivatkozni. Egy új rendezett n-es létrehozásához kényelmes módon feldarabolhatunk régebbieket és összefűzhetjük a szükséges darabokat. Így ha `julia` új filmje megjelenik, megváltoztathatjuk a változót, hogy hivatkozzon egy új rendezett n-esre, ami a régiből származó információkat is tartalmaz:

```
1 julia = julia[:3] + ("Pénzes cápa", 2016) + julia[5:]
2 print(julia)
```

A kimenet:

```
("Julia", "Roberts", 1967, "Pénzes cápa", 2016, "színésznő", "Atlanta, ↵
↪Georgia")
```

Egy elemű rendezett n-es létrehozásakor (bár ilyen kis valószínűséggel teszünk) meg kell adnunk a végén egy vesszőt is, mert a záró vessző nélkül a Python a lentebbi (5) kifejezést egy zárójelben lévő egészként kezeli:

```
>>> tup = (5,)
>>> type(tup)
<class 'tuple'>
>>> x = (5)
>>> type(x)
<class 'int'>
```

9.2. Értékadás rendezett n-esel

A Pythonnak van egy nagyon hatékony **rendezett n-es értékadás** tulajdonsága, amely megengedi a változók rendezett n-esét az értékadás bal oldalán, értéket adva nekik a jobb oldalon lévő rendezett n-esnek megfelelően. (Láttunk már ilyet a pároknál, de ez általánosítható.)

```
(k_nev, v_nev, szul_ev, film, film_ev, foglalkozas, szul_hely) = julia
```

Ez egyenértékű hét értékadó utasítással, mindegyik egyszerűen külön sorban. Az egyetlen követelmény az, hogy a bal oldali változók száma megegyezzen a jobb oldali rendezett n-es elemszámával.

Az ilyen értékadásra gondolhatunk úgy is, mint rendezett n-esek be- és kicsomagolása.

Becsomagolás esetén, a bal oldalon rendezett n-esbe fogjuk össze a dolgokat:

```
>>> b = ("Tibi", 19, "PTI")      # becsomagolás
```

A kicsomagolás során a jobb oldali rendezett n-es értékei bekerülnek a bal oldali változókba:

```
>>> b = ("Tibi", 19, "PTI")
>>> (nev, kor, szak) = b      # kicsomagolás
>>> nev
'Tibi'
>>> kor
19
>>> szak
'PTI'
```

Időnként hasznos felcserélni két változó értékét. A hagyományos értékadó utasításokat használva szükségünk lesz egy átmeneti változóra. Például cseréljük meg az `a` és `b` értékét:

```
1 temp = a
2 a = b
3 b = temp
```

Rendezett n-es értékadással így oldható meg ez a probléma:

```
1 (a, b) = (b, a)
```

A jobb és a bal oldal is egy-egy rendezett n-est tartalmaz. Mindegyik érték a megfelelő helyre kerül. A jobb oldal összes kifejezése kiértékelődik, mielőtt bármelyik értékadás megtörténik. Ez nagyon sokoldalúvá teszi a rendezett n-es értékadást.

Természetesen a jobb és a bal oldalon is azonos számú értéknek kell lennie:

```
>>> (a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

9.3. Rendezett n-es visszatérési értéként

A függvények mindig csak egy értékkel térhetnek vissza, de ha ez az érték egy rendezett n-es, akkor hatékonyan csoportosíthatunk több értéket is a `return` utasításban. Ez nagyon hatékony – gyakran tudni akarjuk néhány ütőjátékos legmagasabb és legalacsonyabb pontszámát is, vagy meg akarjuk találni értékek átlagát és szórását egyszerre, esetleg visszaadhatunk évet, hónapot és napot, esetleg ökológiai modellekben szeretnénk tudni egy adott időpontban a nyulak és farkasok számát egy szigeten.

Például írhatunk egy függvényt, ami visszaadja mind a kerületét, mind a területét egy r sugarú körnek:

```
1 def f(r):
2     """ Visszatér a (kerület, terület) értékekkel egy r sugarú kör esetén """
3     k = 2 * math.pi * r
4     t = math.pi * r * r
5     return (k, t)
```

9.4. Adatszerkezetek alakíthatósága

Láttuk egy korábbi fejezetben, hogy tudunk értékpárokból listát csinálni, és volt már olyan példánk is, ahol a rendezett n-es egyik eleme maga is lista:

```
hallgatok = [
    ("Jani", ["Informatika", "Fizika"]),
    ("Kata", ["Matematika", "Informatika", "Statisztika"]),
    ("Peti", ["Informatika", "Könyvelés", "Közgazdaságtan", "Menedzsment"]),
    ("Andi", ["Információs rendszerek", "Könyvelés", "Közgazdaságtan",
↪ "Vállalkozási jog"]),
    ("Linda", ["Szociológia", "Közgazdaságtan", "Jogi ismeretek", "Statisztika
↪ ", "Zene"])]
```

Egy rendezett n-es előfordulhat egy másikon belül. Például finomíthatjuk a mozicsillagokról tárolt információkat úgy, hogy a teljes születési időt használjuk inkább egyszerű születési év helyett, valamint lehet egy listánk néhány filmjéről és annak megjelenési évéről és így tovább:

```
julia_more_info = ( ("Julia", "Roberts"), (1967, "október", 8),  
                    "színésznő", ("Atlanta", "Georgia"),  
                    [ ("Sztárom a párom", 1999),  
                      ("Micsoda nő", 1990),  
                      ("Ízek, imák, szerelmek", 2010),  
                      ("Erin Brockovich", 2000),  
                      ("Álljon meg a nászmenet", 1997),  
                      ("Egy veszedelmes elme vallomásai", 2002),  
                      ("Oceans Twelve", 2004) ])
```

Figyeld meg, hogy ebben az esetben a rendezett n-esnek csak 5 eleme van – de mindegyik lehet egy másik rendezett n-es, lista, sztring vagy más Python típus. Ezt a tulajdonságot **heterogenitásnak** hívjuk, ami azt jelenti, hogy különböző típusú elemekből épül fel.

9.5. Szójegyzék

adatszerkezet (data structure) Adatok összeszervezése a könnyebb használat céljából.

változtathatatlan adatérték (immutable data value) Egy adatérték, amelyet nem lehet módosítani. Ezek elemeire vagy szeleteire (részeire) vonatkozó értékadás hibát eredményez.

változtatható érték (mutable data value) Egy adatérték, amely módosítható. Minden ilyen értéknek a típusa összetett. A listák és könyvtárak változtathatóak, a sztringek és a rendezett n-esek nem.

rendezett n-es (tuple) Egy változtathatatlan adatérték, amely összetartozó elemeket tartalmaz. A rendezett n-eseket adatok csoportosítására használjuk, például az egy személyhez tartozó adatok (név, kor és nem) együttes tárolására.

rendezett n-es értékadás (tuple assignment) Rendezett n-esek összes eleméhez érték rendelés egyetlen értékadó utasítással. Ez *szimultán* módon történik meg soros helyett, lehetővé téve a hatékony értékcserét.

9.6. Feladatok

1. Nem mondtunk semmit ebben a fejezetben arról, hogy vajon a rendezett n-esek átadhatóak-e függvénynek paraméterként. Hozz létre egy kis Python példakódot ennek kiderítésére!
2. Az értékpár a rendezett n-es általánosítása vagy ez fordítva van?
3. Az értékpár egyfajta rendezett n-es vagy a rendezett n-es egyfajta értékpár?

10. fejezet

Eseményvezérelt programozás

A legtöbb program vagy elektronikus eszköz, mint például a mobiltelefon, reagál különböző *eseményekre* (megtörténő dolgokra). Például, ha megmozdítjuk az egeret, a számítógép érzékeli és reagál; ha egy gombra kattintunk, a program csinál valami érdekeset. Ebben a fejezetben vázlatosan bemutatjuk, hogyan működik az eseményvezérelt programozás.

10.1. Billentyű leütés események

Az alábbi program több újdonságot tartalmaz. Másold be egy szkriptbe és futtasd. A teknőc ablak megjelenése után a kurzormozgató billentyűkkel (a nyilakkal) irányíthatod Esztit.

```
1 import turtle
2
3 turtle.setup(400, 500) # Az ablak méretének beállítása
4 ablak = turtle.Screen() # Az ablak referenciájának lekérése
5 ablak.title("Billentyű leütés kezelése!") # Az ablaknév módosítása
6 ablak.bgcolor("lightgreen") # Háttér színének beállítása
7 Eszti = turtle.Turtle() # A kedvenc teknőcünk elkészítése
8
9 # A következő függvények az eseménykezelőink
10 def ek1():
11     Eszti.forward(30)
12
13 def ek2():
14     Eszti.left(45)
15
16 def ek3():
17     Eszti.right(45)
18
19 def ek4():
20     ablak.bye() # A teknőc ablak bezárása
21
22 # Ezek a sorok rendelik össze a billentyű leütés eseményeket
23 # az általunk definiált eseménykezelő függvényekkel
24 ablak.onkey(ek1, "Up")
25 ablak.onkey(ek2, "Left")
26 ablak.onkey(ek3, "Right")
27 ablak.onkey(ek4, "q")
28
29 # Most megkérjük az ablakot, hogy kezdje el figyelni az eseményeket.
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
30 # Ha bármelyik általunk figyelt billentyűt lenyomja valaki, akkor
31 # a hozzá tartozó eseménykezelő meghívásra kerül.
32 ablak.listen()
33 ablak.mainloop()
```

Néhány megjegyzés a programhoz:

- Az ablak `listen` metódusának meghívása (32. sor) szükséges ahhoz, hogy a program észlelje a billentyűk leütését.
- Az eseménykezelőknek ezúttal az `ek1`, `ek2`, stb. nevet adtuk, de választhatnánk jobb neveket is. Az eseménykezelő függvények tetszőleges komplexitásúak lehetnek, hívhatnak más függvényeket is, stb.
- A `q` billentyű lenyomása az `ek4` függvényt hívja meg (mert a 27. sorban egymáshoz rendeltük a `q` billentyűt és az `ek4` függvényt). A `ek4` függvény végrehajtása alatt az ablak `bye` metódusa (20. sor) bezárja a teknőc ablakot, és befejezteti a `mainloop` metódus hívása által indított folyamatokat. A 33. sor után már nem áll utasítás, tehát a program mindennel elkészült, befejezi működését.
- Egy billentyűre vagy a hozzá tartozó karakterrel (ld.: 27. sor) vagy szimbolikus névvel hivatkozhatunk.
- Néhány szimbolikus név, amit kipróbálhatsz: `Cancel` (a `Break` billentyű), `BackSpace`, `Tab`, `Return` (az `Enter` billentyű), `Shift_L` (bármelyik `Shift` billentyű), `Control_L` (bármelyik `Control` billentyű), `Alt_L` (bármelyik `Alt` billentyű), `Pause`, `Caps_Lock`, `Escape`, `Prior` (Page Up), `Next` (Page Down), `End`, `Home`, `Left`, `Up`, `Right`, `Down`, `Print`, `Insert`, `Delete`, `F1`, `F2`, `F3`, `F4`, `F5`, `F6`, `F7`, `F8`, `F9`, `F10`, `F11`, `F12`, `Num_Lock` és `Scroll_Lock`.

10.2. Egér események

Az egér események egy kicsit eltérnek a billentyűzet eseményektől. Az eseménykezelőnek két paraméterre van szüksége az `x` és `y` koordináták fogadásához. A koordináták adják meg, hogy hol volt az egér az esemény bekövetkeztekor.

```
1 import turtle
2
3 turtle.setup(400,500)
4 ablak = turtle.Screen()
5 ablak.title("Ablakon belüli kattintások kezelése")
6 ablak.bgcolor("lightgreen")
7
8 Eszti = turtle.Turtle()
9 Eszti.color("purple")
10 Eszti.pensize(3)
11 Eszti.shape("circle")
12
13 def ek1(x, y):
14     Eszti.goto(x, y)
15
16 ablak.onclick(ek1) # Összerendeljük a kattintás eseményt az eseménykezelővel
17 ablak.mainloop()
```

A 14. sorban egy új teknőc metódust használtunk, amely lehetővé teszi, hogy a teknőcöt egy *abszolút* módon megadott koordinátára mozgassuk. (A korábbi példáinknál szinte mindig azt adtuk meg, hogy a teknőc az aktuális pozíciójához képest merre menjen, vagyis *relatív* elmozdulást használtunk.) A program oda mozgatja a teknőst, ahová kattintunk az egérrel, miközben a teknőc vonalat rajzol. Próbáld ki!

Ha a 14. sor elé beszurjuk az alábbi sort, akkor egy igen hasznos nyomkövetési trükköt tanulhatunk:

```
ablak.title("Kattintás koordinátái: {0}, {1}".format(x, y))
```

Az ablak fejlécében álló cím könnyen módosítható, ezért néha jó itt megjeleníteni a nyomkövetési és a státusz információkat. (Természetesen nem erre lett kitalálva!)

Van még más is!

Nemcsak az ablak képes az egér események fogadására, a teknőcöknek is lehet saját eseménykezelőjük, amellyel reagálhatnak a kattintásokra. Az a teknőc „kapja meg” az eseményt, amelyik az kurzor alatt áll. Két teknőcöt fogunk készíteni, és mindkét teknőc eseménykezelőjét hozzá rendeljük az onclick eseményhez. Az eseménykezelők eltérő dolgokat tehetnek a hozzájuk tartozó teknőccel.

```
1 import turtle
2
3 turtle.setup(400, 500)      # Az ablak méretének beállítása
4 ablak = turtle.Screen()    # Az ablak referenciájának lekérése
5 ablak.title("Kattintások kezelése!") # Az ablaknév módosítása
6 ablak.bgcolor("lightgreen") # Háttér színének beállítása
7 Eszti = turtle.Turtle()    # Két teknőc készítése
8 Eszti.color("purple")
9 Sanyi = turtle.Turtle()
10 Sanyi.color("blue")
11 Sanyi.forward(100)        # Teknőcök szétválasztása
12
13 def Eszti_esemenykezeloje(x, y):
14     ablak.title("Eszti kattintásának koordinátái: {0}, {1}".format(x, y))
15     Eszti.left(42)
16     Eszti.forward(30)
17
18 def Sanyi_esemenykezeloje(x, y):
19     ablak.title("Sanyi kattintásának koordinátái: {0}, {1}".format(x, y))
20     Sanyi.right(84)
21     Sanyi.forward(50)
22
23 Eszti.onclick(Eszti_esemenykezeloje)
24 Sanyi.onclick(Sanyi_esemenykezeloje)
25
26 ablak.mainloop()
```

Futtasd, és kattintgass a teknőcökre. Figyeld meg, hogy mi történik!

10.3. Időzített, automatikus események

Az ébresztőórák, a konyhai időzítők, vagy a James Bond filmek termonukleáris bombái mind-mind „automatikus” eseményeket generálnak egy bizonyos idő letelte után. A Python teknőc modulja is rendelkezik időzítővel, amely a beállított idő lejártakor egy eseményt hoz létre.

```
1 import turtle
2
3 turtle.setup(400, 500)
4 ablak = turtle.Screen()
5 ablak.title("Időzítő használata")
6 ablak.bgcolor("lightgreen")
7
8 Eszti = turtle.Turtle()
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
9 Eszti.color("purple")
10 Eszti.pensize(3)
11
12 def ek1():
13     Eszti.forward(100)
14     Eszti.left(56)
15
16 ablak.ontimer(ek1, 2000)
17 ablak.mainloop()
```

A 16. sorban az időzítő elindul, 2000 milliszekundum (2 másodperc) múlva „robban”. Amikor az esemény bekövetkezik (*kiváltódik*), az eseménykezelő meghívásra kerül, és Eszti akcióba lép.

Sajnos a beállított időzítő csak egyszer jár le, ezért a szokásos eljárás az, hogy az időzítőt újraindítjuk az eseménykezelőn belül. Ha így járunk el, akkor az időzítő újabb és újabb eseményeket fog generálni. Próbáld ki az alábbi programot:

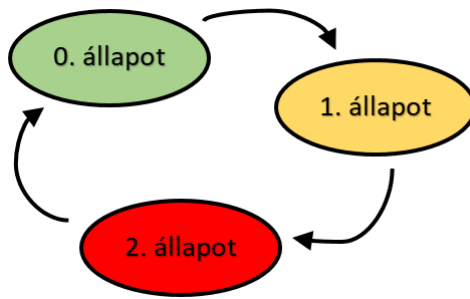
```
1 import turtle
2
3 turtle.setup(400,500)
4 ablak = turtle.Screen()
5 ablak.title("Időzítő használata")
6 ablak.bgcolor("lightgreen")
7
8 Eszti = turtle.Turtle()
9 Eszti.color("purple")
10
11 def ek1():
12     Eszti.forward(100)
13     Eszti.left(56)
14     ablak.ontimer(ek1, 60)
15
16 ek1()
17 ablak.mainloop()
```

10.4. Egy példa: állapotautomata

Az állapotautomaták olyan rendszerek, amelyeknek különböző *állapotaik* lehetnek. Az állapotautomatákat állapotdiagrammal fogjuk leírni, minden egyes állapotot egy-egy körrel vagy ellipszissel reprezentálva. Bizonyos események hatására az automata kiléphet egy állapotból, és átmehet egy másik állapotba, ezeket az *állapotátmeneteket* általában nyilakkal jelöljük a diagramon.

Az ötlet nem új: amikor bekapcsoljuk a mobiltelefonunkat, akkor az egy olyan állapotba kerül, amit „PIN kódra vár” állapotnak nevezhetnénk. A helyes kód beütése átviszi egy másik, „Indulásra kész” állapotba. Ha ezután lezárnánk a telefonunkat, akkor a „Lezárt” állapotba kerülne, és így tovább.

Az egyszerű állapotautomaták közül gyakran találkozunk a közlekedési lámpákkal. Tegyük fel, hogy a jelzőlámpa csak zöld, sárga és piros jelzéseket ad. Ugyan ez eltér a Magyarországon megszokottól, hiszen az együttes piros-sárga jelzés hiányzik, de akad néhány ország, ahol ezt a jelzést alkalmazzák. Az alábbi állapotdiagramon láthatjuk, hogy a készülékben három állapot váltja egymást ciklikusan. Az állapotokat sorszámokkal jelöltük: 0., 1. és 2.



Most egy olyan programot készítünk, amely egy teknőccel szimulálja a közlekedési lámpákat. Három „leckéből” áll majd össze a program. Az első a teknőcök eddigiektől eltérő használatát mutatja be. A második azt demonstrálja, hogyan készíthetünk állapotautomatát Pythonban. A rendszer állapotát egy változóval fogjuk nyomon követni, és számos `if` utasítás szolgál majd az aktuális állapot ellenőrzésére, és azon műveletek elvégzésére, amivel a rendszert egy másik állapotba vihetjük át. A harmadik rész arról szól majd, hogyan lehet a billentyűzet eseményeit felhasználni az állapotátmenetek előidézéséhez.

Másold be a programot a saját környezetedbe és futtasd. Minden egyes sor szerepét meg kell értened. Ha szükséges, akkor használd a dokumentációt.

```
1 import turtle                # Eszti közlekedési lámpává válik.
2
3 turtle.setup(400,500)
4 ablak = turtle.Screen()
5 ablak.title("Eszti közlekedési lámpává válik.")
6 ablak.bgcolor("lightgreen")
7 Eszti = turtle.Turtle()
8
9
10 def doboz_rajzolas():
11     """ Egy csinos doboz rajzolása a közlekedési lámpa számára """
12     Eszti.pensize(3)
13     Eszti.color("black", "darkgrey")
14     Eszti.begin_fill()
15     Eszti.forward(80)
16     Eszti.left(90)
17     Eszti.forward(200)
18     Eszti.circle(40, 180)
19     Eszti.forward(200)
20     Eszti.left(90)
21     Eszti.end_fill()
22
23
24 doboz_rajzolas()
25
26 Eszti.penup()
27 # Eszti pozicionálása oda, ahol a zöld lámpának kell lennie
28 Eszti.forward(40)
29 Eszti.left(90)
30 Eszti.forward(50)
31 # Eszti egy nagy zöld körre alakítjuk át
32 Eszti.shape("circle")
33 Eszti.shapesize(3)
34 Eszti.fillcolor("green")
35
36 # A közlekedési lámpa egyfajta állapotautomata, három állapottal:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
37 # zölddel, sárgával és pirossal. Az állapotokat rendre
38 # 0, 1, 2 számokkal írjuk le.
39 # Az állapotváltásnál Eszti helyzetét és színét változtatjuk meg.
40
41 # Ez a változó hordozza az aktuális állapotot
42 állapot_sorszam = 0
43
44
45 def állapot_automata_esemenykezeloe():
46     global állapot_sorszam
47     if állapot_sorszam == 0:      # Átmenet a 0. állapotból az 1. állapotba
48         Eszti.forward(70)
49         Eszti.fillcolor("orange")
50         állapot_sorszam = 1
51     elif állapot_sorszam == 1:    # Átmenet az 1. állapotból a 2. állapotba
52         Eszti.forward(70)
53         Eszti.fillcolor("red")
54         állapot_sorszam = 2
55     else:                         # Átmenet a 2. állapotból az 0. állapotba
56         Eszti.back(140)
57         Eszti.fillcolor("green")
58         állapot_sorszam = 0
59
60 # Az eseménykezelőt a space billentyűhöz kötjük
61 ablak.onkey(állapot_automata_esemenykezeloe, "space")
62
63 ablak.listen()                  # Események figyelése
64 ablak.mainloop()
```

A 46. sorban egy, még ismeretlen utasítást láthatunk. A `global` kulcsszó azt mondja a Pythonnak, hogy ne hozzon létre új lokális változót az `állapot_sorszam` számára (annak ellenére, hogy a függvény 50., 54. és 58. sora is használja), így a függvényen belüli `állapot_sorszam` név mindig a 42. sorban létrehozott változóra utal.

Akármelyik állapotban is van az automata, az `állapot_automata_esemenykezeloe` függvényen belüli utasítások átviszik az automatát a következő állapotba. Az állapotváltás közben Eszti új helyre kerül és megváltozik a színe. Természetesen az `állapot_sorszam` is változik, annak az állapotnak a száma lesz hozzárendelve, amelybe éppen most került át az automata.

A közlekedési lámpa automata, a space billentyű minden egyes lenyomásakor, új állapotba kerül át.

10.5. Szójegyzék

esemény (event) Olyasvalami, ami a normális programvezérlésen kívül történik. Gyakran felhasználói tevékenység okozza. A tipikus események közé tartoznak az egérműveletek és a billentyűk leütései is. Láthattuk, hogy az időzítők is válhatnak ki eseményt.

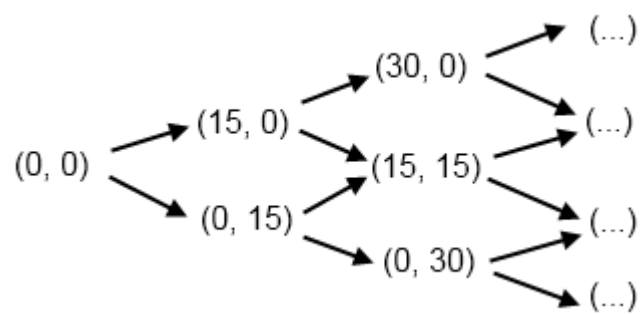
eseménykezelő (handler) Egy olyan függvény, amely egy esemény bekövetkeztekor kerül meghívásra, mintegy válaszul arra.

hozzárendelés (bind) Egy függvény eseményhez rendelése azt jelenti, hogy az esemény bekövetkeztekor a függvény kerül meghívásra. Az eseményhez rendelt függvény feladata az esemény kezelése.

kötés (bind) A hozzárendelés szinonimája.

10.6. Feladatok

1. Társíts még néhány billentyűt az első példaprogramhoz:
 - Az R, G és B billentyűk leütése változtassa meg Eszti színét pirosra (Red), zöldre (Green) és kékre (Blue).
 - A + és - billentyűk leütése növelje, illetve csökkentse Eszti tollának méretét. Biztosítsd, hogy a toll mérete 1 és 20 között maradjon (a határokat is beleértve).
 - Néhány más billentyűt is vezess be Eszti vagy az ablak különböző tulajdonságainak állítására, vagy adj Esztihez új, billentyűzettel vezérelhető viselkedést.
2. Változtasd meg úgy a közlekedési lámpa vezérlő programot, hogy automatikusan váltson, egy időzítő hatására.
3. Az egyik korábbi fejezetben találkoztunk a `hideturtle` és a `showturtle` metódusokkal, amelyekkel elrejtethetők, illetve megjeleníthetők a teknőcök. A két metódus lehetővé teszi, hogy egy másik megközelítést alkalmazzunk a közlekedési lámpa vezérlésére készített program fejlesztésénél. Egészítsd ki a programod a következőkkel. Rajzolj egy második dobozt az újabb lámpák tárolására. Készíts három különböző teknőcöt a zöld, sárga és a zöld lámpák reprezentálásához, és tedd őket az új lámpadobozba. Az állapotváltáskor csak tegyél egy teknőcöt láthatóvá a háromból. Amikor készen vagy dőlj hátra, és mélyedj el a gondolataidban. Két különböző megoldásod van, mindkettő működőképesnek látszik. Jobb-e valamilyen szempontból az egyik megoldás, mint a másik? Melyik áll közelebb a valósághoz? Melyik hasonlít jobban a városodban lévő közlekedési lámpák működéséhez?
4. A közlekedési lámpák fényeit most már különböző teknőcök jelenítik meg a programban. A látható/nem látható trükk nem volt túl jó ötlet. Ha megnézzük egy közlekedési lámpát, akkor azt látjuk, hogy a különböző színű lámpák bekapcsolnak majd lekapcsolódnak, de akkor is látszanak, amikor éppen nem világítanak, csak egy kicsit sötétebb a színük. Módosítsd úgy a programot, hogy ne tűnjenek el a lámpák sem kikapcsolt, sem bekapcsolt állapotban. Kikapcsolt állapotban is lehessen valamennyire látni a lámpákat.
5. A közlekedési lámpa vezérlő programod szabadalmaztatva lett, így arra számítasz, hogy nagyon gazdag leszel. Egy új ügyfeled azonban változtatást igényel. Négy állapotot akar az automatában: zöldet, zöldet és sárgát együtt, csak sárgát és csak pirosat. Ráadásul azt is szeretné, ha a különböző állapotokhoz különböző időtartam társulna. Az állapotautomatának 3 másodpercet kell a zöld állapotban töltenie, amit 1 másodperc zöld+sárga állapot követ. Utána 1 másodperc sárga állapot jön, majd 2 másodperc piros következik. Változtasd meg az automata működési logikáját!
6. Ha nem tudod, hogyan történik a pontozás a teniszben, kérdezd meg egy barátodat, vagy nézd meg a Wikipédián. Az egyszemélyes tenisz játszmákat (melyek A és B játékos között folynak) mindig pontra játsszák. A játék állására gondoljunk állapotautomataként. A játék a (0, 0) állapotból indul, ami azt jelenti, hogy az egyik játékosnak sincs még pontja. Feltételezzük, hogy az értékpár első tagja az A játékos pontszáma. Ha az A játékos nyeri az első pontot, akkor az állás (1, 0), ha a B játékos nyeri, akkor (0, 1) lesz. Az alábbi ábrán látható az első néhány állapot és állapotátmenet. Az összes diagramon látható állapotban két lehetséges kimenet van (vagy A nyeri a következő pontot, vagy B). A felső nyíl által jelzett átvitel mindig akkor lép életbe, ha az A játékos nyeri a pontot. Egészítsd ki a diagramot úgy, hogy az összes állapot, és az összes állapotátmenet szerepeljen rajta. (Segítség: összesen 20 állapot van, beleszámítva az előny állapotokat, a döntetlent és az „A nyert” és a „B nyert” állapot is.)



11. fejezet

Listák

A **lista** értékek rendezett gyűjteménye. Azokat az értékeket, amelyek a listát alkotják **elemeknek** nevezzük. A listák hasonlóak a sztringekhez, amelyek a karakterek rendezett gyűjteményei, kivéve, hogy a lista elemei bármilyen típusúak lehetnek. A listákat és a sztringeket – és más gyűjteményeket, amelyek megőrzik az elemek sorrendjét – **sorozatnak** nevezzük.

11.1. A lista értékei

Többféle módon lehetséges egy új lista létrehozása; legegyszerűbb az elemek szógletes zárójelbe való felsorolása ([és]):

```
1 ps = [10, 20, 30, 40]
2 qs = ["alma", "eper", "barack"]
```

Az első példa egy lista, amely négy egész számot tartalmaz. A második lista pedig három sztringet tartalmaz. A lista elemeinek nem kell azonos típusúnak lennie. A következő lista tartalmaz egy sztringet, egy valós számot, egy egész számot és (érdekességgéppen) egy másik listát.

```
1 zs = ["hello", 2.0, 5, [10, 20]]
```

A listában szereplő másik listáról azt mondjuk, hogy **beágyazott**.

Végül azt a listát, amely nem tartalmaz elemeket, üres listának nevezzük, és [] jelöljük.

Ahogy már korábban láthattuk, a változókhoz vagy a listákhoz tartozó listaértékeket paraméterként hozzárendelhetjük a függvényekhez:

```
1 szotar = ["alma", "sajt", "kutya"]
2 szamok = [17, 123]
3 ures_lista = []
4 print(szotar, szamok, ures_lista)
```

```
["alma", "sajt", "kutya"] [17, 123] []
```

11.2. Elemek elérése

A listaelemek elérésének szintaktikája hasonló, mint a sztringek esetében – az index operátort használjuk: `[]` (ne tévesszük össze az üres listával). A zárójelben lévő kifejezés adja meg az indexet. Emlékezzünk arra, hogy az indexek 0-tól kezdődnek:

```
1 print(szamok[0])
```

```
17
```

Bármilyen egész értéket visszaadó kifejezés használható indexként:

```
1 print(szamok[9-8])
```

```
123
```

```
1 print(szamok[1.0])
```

```
Traceback (most recent call last):
  File "<input>", so 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Ha egy olyan elemet akarunk elérni, amely nem létezik, futási idejű hibát kapunk:

```
1 print(szamok[2])
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Használhatjuk a ciklusváltozót lista indexként.

```
1 lovasok = ["háború", "éhínség", "pestis", "halál"]
2
3 for i in [0, 1, 2, 3]:
4     print(lovasok[i])
```

A cikluson belül minden alkalommal az `i` változót használjuk a lista `i`. elemének kiírtatására. Ezt az algoritmust nevezzük **lista bejárásnak**.

A fenti példa esetén nem szükséges vagy nem használja az `i` indexet semmire, csak az elemek elérésére, így ez a direktebb verzió – ahol a `for` ciklus megkapja az elemeket – kedveltebb lehet:

```
1 lovasok = ["háború", "éhínség", "pestis", "halál"]
2
3 for h in lovasok:
4     print(h)
```

11.3. A lista hossza

A `len` függvény visszatér a lista hosszával, amely egyenlő a lista elemeinek számával. Amennyiben egy egész indexet használunk a lista eléréséhez, célszerűbb, ha a lista hosszát használjuk a ciklus felső értékeként egy konstans helyett. Így, ha a lista mérete megváltozik, nem szükséges végig követni a teljes programot és módosítani az összes ciklust, mivel bármilyen méretű lista esetén megfelelően fog működni:

```
1 lovasok = ["háború", "éhínség", "pestis", "halál"]
2
3 for i in range(len(lovasok)):
4     print(lovasok[i])
```

Az ciklus utolsó végrehajtása esetén, `i` értéke a `len(lovasok)-1`, amely az utolsó elem indexe. (De az index nélküli változat jobban néz ki!).

Habár a lista egy másik listát is tartalmazhat, a beágyazott lista egyetlen elemként szerepel a szülői listában.

A lista hossza 4:

```
1 hossz = len(["autó gyártók", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
2 print(hossz)
```

```
4
```

11.4. Lista tagság

Az `in` és a `not in` Boolean típusú operátorok, amelyek megvizsgálják egy elem tagságát a sorozatban. Korábban a sztringeknél már használtuk, de listákkal és más sorozatokkal is működnek:

```
1 lovasok = ["háború", "éhínség", "pestis", "halál"]
2
3 print("pestis" in lovasok)
4 print("dezertálás" in lovasok)
5 print("dezertálás" not in lovasok)
```

Az eredmény a következő:

```
True
False
True
```

Ez a módszer sokkal elegánsabb a beágyazott ciklusoknál, amit korábban az Informatikára jelentkező hallgatók számának meghatározásához használtunk a *Beágyazott ciklus beágyazott adatokhoz* fejezetben:

```
1 hallgatok = [
2     ("Jani", ["Informatika", "Fizika"]),
3     ("Kata", ["Matematika", "Informatika", "Statisztika"]),
4     ("Peti", ["Informatika", "Könyvelés", "Közgazdaságtan", "Menedzsment"]),
5     ("Andi", ["Információs Rendszerek", "Könyvelés", "Közgazdaságtan", "
↪ Vállalkozási Jog"]),
6     ("Linda", ["Szociológia", "Közgazdaságtan", "Jogi ismeretek",
↪ "Statisztika", "Zene"])]
7
8 # Számold meg, hány hallgató vette fel az Informatikát.
9 szamlalo = 0
10 for (nev, targyak) in hallgatok:
11     if "Informatika" in targyak:
12         szamlalo += 1
13
14 print("Az Informatikát felvett hallgatók száma:", szamlalo)
```

11.5. Lista műveletek

A + operátor összefűzi a listákat:

```
1 a = [1, 2, 3]
2 b = [4, 5, 6]
3 c = a + b
4 print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

Hasonlóképpen, a * operátor megismétli a listát egy megadott számszor:

```
1 d = [0] * 4
2 print(d)
3 e = [1, 2, 3] * 3
4 print(e)
```

```
[0, 0, 0, 0]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Az első példában [0]-t négyszer ismétli. A második példában az [1, 2, 3] listát háromszor ismétli meg.

11.6. Lista szeletek

A szeletelő operátorok, ahogyan korábban a sztringeknél is láthattuk, működnek részlisták esetében is:

```
1 a_list = ["a", "b", "c", "d", "e", "f"]
2 print(a_list[1:3])
3 print(a_list[:4])
4 print(a_list[3:])
5 print(a_list[:])
```

Az eredmény a következő:

```
['b', 'c']
['a', 'b', 'c', 'd']
['d', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

11.7. A listák módosíthatók

A sztringektől eltérően a listák **módosíthatók**, ami azt jelenti, hogy megváltoztathatjuk az elemeiket. Az értékadás bal oldalán az index operátor használatával az egyik elemet módosíthatjuk.

```
1 gyumolcs = ["banán", "alma", "eper"]
2 gyumolcs[0] = "körte"
3 gyumolcs[2] = "narancs"
4 print(gyumolcs)
```

```
['körte', 'alma', 'narancs']
```

A zárójel operátor a listákra alkalmazva bárhol megjelenhet egy kifejezésben. Ha a kifejezés bal oldalán jelenik meg, akkor megváltoztatja a lista egyik elemét, így a gyumolcs lista első eleme fog cserélődni "banán"-ról "körte"-re, és az utolsó eleme pedig "eper"-ről "narancs"-ra. Az elem listához való hozzárendelését **indexelt értékadásnak** nevezzük. Az indexelt értékadás nem működik a sztringek esetében:

```
1 saját_sztring = "ADAT"
2 saját_sztring[3] = "G"
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

de a listák esetében igen:

```
1 saját_lista = ["A", "D", "A", "T"]
2 saját_lista[3] = "G"
3 print(saját_lista)
```

```
['A', 'D', 'A', 'G']
```

A szeletelő operátor használatával módosíthatjuk a teljes részlistát:

```
1 a_list = ["a", "b", "c", "d", "e", "f"]
2 a_list[1:3] = ["x", "y"]
3 print(a_list)
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```

Az elemeket a listából eltávolíthatjuk úgy, hogy hozzárendelünk egy üres listát:

```
1 a_list = ["a", "b", "c", "d", "e", "f"]
2 a_list[1:3] = []
3 print(a_list)
```

```
['a', 'd', 'e', 'f']
```

Hozzáadhatunk a listához elemeket úgy, hogy beszúrjuk őket egy üres szeletre a kívánt helyen:

```
1 a_list = ["a", "d", "f"]
2 a_list[1:1] = ["b", "c"]
3 print(a_list)
```

```
['a', 'b', 'c', 'd', 'f']
```

```
1 a_list[4:4] = ["e"]
2 print(a_list)
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

11.8. Lista törlése

A szeletek használata a lista törlésére hibát adhat. A Python egy jobban olvasható alternatívát is kínál. A `del` utasítás eltávolít egy elemet a listából:

```
1 a = ["egy", "kettő", "három"]
2 del a[1]
3 print(a)
```

```
['egy', 'három']
```

A `del` utasítás futási idejű hibát ad vissza, amennyiben az index kívül esik a tartományon.

A `del`-t használhatjuk egy szelettel, hogy kitöröljünk egy részhalmazt:

```
1 a_list = ["a", "b", "c", "d", "e", "f"]
2 del a_list[1:5]
3 print(a_list)
```

```
['a', 'f']
```

A szokásos módon a szelet által választott részhalmaz tartalmazza az összes elemet az első indextől kezdődően, de már nem tartalmazza a második indexű elemet.

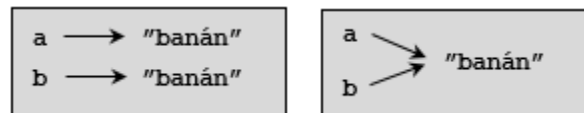
11.9. Objektumok és hivatkozások

Miután végrehajtjuk az értékadó utasításokat:

```
1 a = "banán"
2 b = "banán"
```

látjuk, hogy az `a` és `b` a `"banán"` sztring objektumra utal. De még nem tudjuk, hogy *ugyanarra* a sztring objektumra mutatnak-e.

Két lehetséges módja van annak, hogy a Python kezelje a memóriát:



Az első esetben `a` és `b` két különböző objektumra hivatkozik, amelyek azonos értékűek. A második esetben *ugyanarra* az objektumra hivatkoznak.

Az `is` operátor segítségével megvizsgálhatjuk, hogy a két név *ugyanarra* az objektumra hivatkozik-e:

```
1 print(a is b)
```

```
True
```

Azt mutatja, hogy az `a` és `b` *ugyanarra* az objektumra hivatkozik, továbbá, hogy a második a két pillanatnyi állapot közül az, amely pontosan leírja a kapcsolatot.

Mivel a sztringek *megváltoztathatatlanok*, a Python úgy optimalizálja az erőforrásokat, hogy létrehoz két nevet, amely *ugyanarra* a sztringre, *ugyanarra* az objektumra hivatkozik.

Ez nem áll fenn a listák esetében:

```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
```

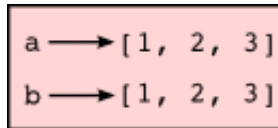
(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3 print(a == b)
4 print(a is b)
```

```
True
False
```

Az aktuális állapot a következőképpen néz ki:



a és b-nek ugyanaz az értéke, de nem ugyanarra az objektumra hivatkoznak.

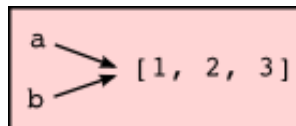
11.10. Fedőnevek

Mivel a változók objektumokra hivatkoznak, ha egy változót hozzárendelünk egy másikhoz, mindkét változó ugyanarra az objektumra fog hivatkozni:

```
1 a = [1, 2, 3]
2 b = a
3 print(a is b)
```

```
True
```

Ebben az esetben a pillanatnyi állapot a következőképpen néz ki:



Mivel ugyanazon a listára két különböző névvel hivatkozunk, a-val és b-vel, azt mondjuk, hogy ők **fedőnevek**. A fedőneveken végrehajtott változtatások hatással vannak egymásra.

```
1 b[0] = 5
2 print(a)
```

```
[5, 2, 3]
```

Habár ez a tulajdonság hasznos, néha kiszámíthatatlan és nemkívánatos. Általában biztonságosabb elkerülni a fedőnevek használatát, amikor módosítható objektumokkal dolgozunk (például: a tankönyvünk ezen pontján lévő felsorolások, továbbá több módosítható objektummal is fogunk találkozni, osztályokkal és objektumokkal, szótárakkal és halmazokkal). Természetesen a megváltozhatatlan objektumok (például: sztringek, rendezett n-esek) esetén nincs probléma – tehát a fedőneveket nem lehetséges csak úgy megváltoztatni. Ezért a Python szabadon ad fedőnevet a sztringeknek (és bármilyen más megváltozhatatlan adat típusoknak), amikor lehetőséget lát a takarékoskodásra.

11.11. Listák klónozása

Ha módosítani szeretnénk egy listát, és az eredeti példányát is meg szeretnénk őrizni, szükséges egy másolatot készíteni a listáról, nem csak a hivatkozásról. Ezt a folyamatot **klónozásnak** nevezzük, hogy elkerüljük a másolás szó

kétértelműségét.

A lista klónozásának legegyszerűbb módja a szelet operátor használata:

```
1 a = [1, 2, 3]
2 b = a[:]
3 print(b)
```

```
[1, 2, 3]
```

Az a bármelyik szeletével egy új listát hozhatunk létre. Ebben az esetben a szelet tartalmazza a teljes listát. Tehát most a kapcsolat a következőképpen néz ki:

```
a → [1, 2, 3]
b → [1, 2, 3]
```

Most szabadon megváltoztathatjuk `b`-t anélkül, hogy aggódnánk attól, hogy véletlenül megváltoztatjuk az `a`-t:

```
1 b[0] = 5
2 print(a)
```

```
[1, 2, 3]
```

11.12. Listák és a `for` ciklus

A `for` ciklus működik a listákkal is, ahogyan már korábban láthattuk. A `for` ciklus általánosított szintaxisa:

```
for VÁLTOZÓ in LISTA:
    TÖRZS
```

Tehát, ahogy láttuk:

```
1 barátok = ["Péter", "Zoli", "Kata", "Zsuzsa", "Tamás", "József", "Sándor"]
2 for barát in barátok:
3     print(barát)
```

Bármely lista kifejezés használható egy `for` ciklusban:

```
1 for szam in range(20):
2     if szam % 3 == 0:
3         print(szam)
4
5 for filmek in ["vígjáték", "animációs", "romantikus"]:
6     print("Én szeretem a " + filmek + "et!")
```

Az első példa kiírja a 3 szám összes többszörösét 0 és 19 között. A második példa a különféle filmek iránti rajongást fejezi ki.

Mivel a listák módosíthatók, gyakran a listát szeretnénk bejárni, megváltoztatva minden elemét. A következő példában az `xs` lista összes elemét négyzetre emeljük:


```
1 xs = [1, 2, 3, 4, 5]
2
3 for i in range(len(xs)):
4     xs[i] = xs[i]**2
```

Vessünk egy pillantást a `range(len(xs))` utasításra, amíg meg nem értjük, hogy működik!

Ebben a példában mind az elem értéke (négyzetre akarjuk emelni az értékeket), mind pedig az `indexe` (az új értéket hozzárendeljük a pozícióhoz) érdekel bennünket. Ez a minta elég gyakori, a Python szebb módot ajánl ennek megvalósítására.

```
1 xs = [1, 2, 3, 4, 5]
2
3 for (i, ert) in enumerate(xs):
4     xs[i] = ert**2
```

Az `enumerate` (index, érték) párokat generál a lista bejárás során. Próbáld ki a következő példát, hogy jobban megértsd az `enumerate` működését!

```
1 for (i, v) in enumerate(["banán", "alma", "körte", "citrom"]):
2     print(i, v)
```

```
0 banán
1 alma
2 körte
3 citrom
```

11.13. Lista paraméterek

Ha egy listát argumentumként átadunk, akkor hivatkozni fog a listára, nem egy másolatot vagy klónt készít a listáról. Tehát a paraméterátadásra egy fedőnevet hoz létre: a hívónak van egy változója, mely a listára hivatkozik, és a hívott függvénynek van egy fedőneve, de alapvetően csak egy lista objektum van. Például az alábbi függvény argumentuma egy lista, mely a listának minden elemét megszorozza 2-vel:

```
1 def megduplaz(a_list):
2     """ Átírjuk a lista minden elemét a kétszeresére. """
3     for (idx, ert) in enumerate(a_list):
4         a_list[idx] = 2 * ert
```

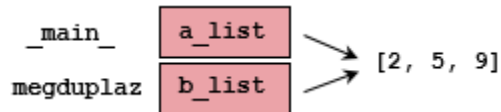
Ha a szkripthez hozzáadjuk a következőket:

```
1 b_list = [2, 5, 9]
2 megduplaz(b_list)
3 print(b_list)
```

Amikor futtatjuk a következő eredményt kapjuk:

```
[4, 10, 18]
```

A fenti függvényben az `a_list` paraméter és a `b_list` változó ugyanazon objektum fedőnevei. Tehát a listában szereplő elemek módosítása előtt a lista állapota a következőképpen néz ki:



Mivel a lista objektum meg van osztva két keretre, a listát ezek közé írtuk. Ha egy függvény módosítja a lista paramétereinek elemeit, akkor a hívó látja a módosítást.

Használjuk a Python megjelenítőt!

A Python megjelenítő egy nagyon hasznos eszköz, mely segítséget nyújt a hivatkozások, fedőnevek, értékadások és a függvény argumentumok átadásának megértéséhez. Különös figyelmet kell fordítani azokra az esetekre, amikor egy listát klónozzunk vagy két külön listánk van, valamint amikor csak egy alapvető lista szerepel, de egynél több fedőneves változó hivatkozik a listára.

11.14. Lista metódusok

A pont operátor is használható a lista objektumok beépített metódusainak elérésére. Kezdjük a leghasznosabb metódussal, amellyel hozzáadhatunk valamit a lista végéhez:

```
1 saját_lista = []
2 saját_lista.append(5)
3 saját_lista.append(27)
4 saját_lista.append(3)
5 saját_lista.append(12)
6 print(saját_lista)
```

```
[5, 27, 3, 12]
```

Az `append` lista metódus hozzáfűzi a megadott argumentumot a lista végéhez. Gyakran használjuk új lista készítésénél. A következő példával bemutatjuk néhány további lista metódus használatát.

Szúrjuk be a 12-t az 1-es pozícióra, eltolva a többi elemet!

```
1 saját_lista.insert(1, 12)
2 print(saját_lista)
```

```
[5, 12, 27, 3, 12]
```

Hány 12-es érték szerepel a listában?

```
1 print(saját_lista.count(12))
```

```
2
```

Szúrjuk be a teljes listát a saját_lista végére!

```
1 saját_lista.extend([5, 9, 5, 11])
2 print(saját_lista)
```

```
[5, 12, 27, 3, 12, 5, 9, 5, 11]
```

Keressük meg az első 9-es érték indexét a listában!

```
1 print(sajat_lista.index(9))
```

```
6
```

Fordítsuk meg a listát!

```
1 saját_lista.reverse()
2 print(sajat_lista)
```

```
[11, 5, 9, 5, 12, 3, 27, 12, 5]
```

Rendezzük a listát!

```
1 saját_lista.sort()
2 print(sajat_lista)
```

```
[3, 5, 5, 5, 9, 11, 12, 12, 27]
```

Rendezzünk egy szöveges adatokat tartalmazó listát!

```
1 szoveg_lista = ["barack", "alma", "mandarin"]
2 szoveg_lista.sort()
3 print(szoveg_lista)
4 szoveg_lista2 = ["én", "te", "ő", "mi", "ti", 'ők']
5 szoveg_lista2.sort()
6 print(szoveg_lista2)
```

```
['alma', 'barack', 'mandarin']
['mi', 'te', 'ti', 'én', 'ő', 'ők']
```

A második listára kapott eredménnyel valószínűleg nem vagyunk elégedettek. Mivel a rendezés az elemek összehasonlításán alapul, ezért azt kell megoldanunk, hogy a hasonlítás az általunk kívánt módon történjen meg:

```
1 import locale
2 import functools
3
4 locale.setlocale(locale.LC_ALL, "HU_hu.UTF8") # a nyelv és a kódolás_
   ↳ beállítása
5 szoveg_lista2 = ["én", "te", "ő", "mi", "ti", 'ők']
6 szoveg_lista2.sort(key = functools.cmp_to_key(locale.strcoll))
7 print(szoveg_lista2)
```

A korábbiakban a `sort` metódust paraméter nélkül használtuk. Most a `key` paraméter segítségével megadjuk, hogy az `strcoll` hasonlítsa össze az elemeket, mely figyelembe veszi a 4. sorban beállított környezetet. Most már nem okoznak problémát az ékezetes karakterek.

```
['én', 'mi', 'ő', 'ők', 'te', 'ti']
```

Távolítsuk el az első 12-es értéket a listából!

```
1 saját_lista.remove(12)
2 print(sajat_lista)
```

```
[3, 5, 5, 5, 9, 11, 12, 27]
```

Kísérletezz és játssz az itt bemutatott lista metódusokkal, és olvasd el a rájuk vonatkozó dokumentációkat, addig, amíg nem vagy biztos benne, hogy megértetted hogyan működnek.

11.15. Tiszta függvények és módosítók

Azok a függvények, amelyek argumentumként egy listát kapnak, és módosítják a listát a végrehajtás során **módosítónak**, és az általuk végrehajtott változtatásokat pedig **mellékhatásnak** nevezzük.

A **tiszta függvény** nem eredményez mellékhatásokat. A tiszta függvény a hívó programmal csak a paramétereken keresztül kommunikál, amelyeket nem módosít, és visszaad egy értéket. Itt a `megduplaz` egy tiszta függvényként van megírva:

```
1 def megduplaz(a_list):
2     """ Visszaad egy listát, mely az a_list elemeinek kétszeresét_
    ↪ tartalmazza. """
3     uj_list = []
4     for ertekek in a_list:
5         uj_elem = 2 * ertekek
6         uj_list.append(uj_elem)
7
8     return uj_list
```

Ez a `megduplaz` változat nem változtatja meg a függvény argumentumait:

```
1 b_list = [2, 5, 9]
2 xs = megduplaz(b_list)
3 print(b_list)
4 print(xs)
```

```
[2, 5, 9]
[4, 10, 18]
```

Egy korábbi szabály szerint, amely az értékadásra vonatkozott „először kiértékeljük a jobb oldalt, majd hozzárendeljük az eredményt a változóhoz”. Tehát elég biztonságos a függvény eredményét ugyanahhoz a változóhoz rendelni, melyet átadtunk a függvénynek:

```
1 b_list = [2, 5, 9]
2 b_list = megduplaz(b_list)
3 print(b_list)
```

```
[4, 10, 18]
```

Melyik stílus a jobb?

Bármilyen, amit a módosítókkal meg lehet tenni az tiszta függvényekkel is elvégezhető. Valójában egyes programozási nyelvek csak tiszta függvényeket engedélyeznek. Van néhány bizonyíték arra, hogy azok a programok, melyek tiszta függvényeket használnak gyorsabbak, és kevesebb hibalehetőséget tartalmaznak, mint a módosítókat használók. Mindazonáltal a módosítók néha kényelmesebbek, és egyes esetekben a funkcionális programok kevésbé hatékonyak.

Általánosságban azt javasolják, hogy tiszta függvényeket írjunk, és csak akkor alkalmazzuk a módosítókat, ha nyomós okunk van rá, és előnyünk származik belőle. Ezt a megközelítést *funkcionális programozási stílusnak* nevezzük.

11.16. Listákat előállító függvények

A fent említett megdupláz tiszta verziója egy fontos **mintát** használ az eszköztárából. Amikor egy listát létrehozó és visszaadó függvényt kell írni, a minta általában:

```
1 inicializálja az eredmény változót, legyen egy üres lista
2 ciklus
3     hozzon létre egy új elemet
4     fűzze hozzá az eredményhez
5 return eredmény
```

Mutassuk be egy másik használati módját ennek a mintának! Tegyük fel, hogy már van egy `primszam(x)` függvényünk, amely teszteli, hogy az `x` prímszám-e. Írj egy függvényt, amely visszaadja az összes `n`-nél kisebb prímszámot:

```
1 def prim_kisebbmint(n):
2     """ Visszaadja az összes n-nél kisebb prímszámot. """
3     eredmény = []
4     for i in range(2, n):
5         if primszam(i):
6             eredmény.append(i)
7     return eredmény
```

11.17. Sztringek és listák

A két leghasznosabb módszer a sztringek esetében a részsstringek listájának (oda és vissza) konverziója. A `split` módszer (melyet már korábban láthattunk) szétválasztja a sztringet szavak listájába. Alapértelmezés szerint bármilyen számú whitespace karakter tekinthető szóhatárnak:

```
1 nota = "Esik eső, szép csendesén csepereg..."
2 szavak = nota.split()
3 print(szavak)
```

```
['Esik', 'eső,', 'szép', 'csendesén', 'csepereg...']
```

Az opcionálisként megadott argumentumot **határolónak** nevezzük, amely meghatározza, hogy mely karakterlánc legyen a határ a részsstringek között. A következő példában az `se` sztring határolót használjuk:

```
1 print(nota.split("se"))
```

```
['Esik eső, szép c', 'nde', 'n c', 'pereg...']
```

Figyeljünk meg, hogy a határoló nem jelenik meg az eredményben.

A `split` módszer inverze a `join` módszer. Kiválaszthatjuk a kívánt határoló sztringet (gyakran *ragasztónak* nevezik), és összefűzhetjük a lista minden egyes elemét a ragasztóval.

```
1 ragasztó = ";"
2 s = ragasztó.join(szavak)
3 print(s)
```

```
'Esik;eső,;szép;csendesén;csepereg...'
```

Az összeillesztett lista (a példában szereplő `szavak`) nem módosul. Továbbá, amint ez a következő példa is mutatja, használhatunk üres vagy több karakterből álló sztringet ragasztóként:

```
1 print(" -- ".join(szavak))
```

```
'Esik -- eső, -- szép -- csendesen -- csepereg...'
```

```
1 print("".join(szavak))
```

```
'Esikeső,szépcsendesencsepereg...'
```

11.18. A list és a range

A Python egy `list` nevezetű beépített konverziós függvénnyel rendelkezik, amely megpróbál bármit listává alakítani.

```
1 xs = list("Mocsári Béka")
2 print(xs)
3 print("".join(xs))
```

```
['M', 'o', 'c', 's', 'á', 'r', 'i', ' ', 'B', 'é', 'k', 'a']
'Mocsári Béka'
```

A `range` egyik tulajdonsága az, hogy nem számolja ki rögtön az összes értéket: „félre teszi” a számolást, és csak kérésre végzi el, azaz „lustán”. Mondhatni ígéretet ad rá, hogy amikor szükségünk lesz egy elemre, akkor elő fogja azt állítani. Ez nagyon kényelmes, ha a számításunk rövidzáras keresés, és korábban visszatér az értékkel, mint a következő esetben:

```
1 def f(n):
2     """ Keresse meg az első pozitív egész számot 101 és n között, amely osztható
    ↳ 21-el. """
3     for i in range(101, n):
4         if (i % 21 == 0):
5             return i
6
7     teszt(f(110) == 105)
8     teszt(f(1000000000) == 105)
```

A második teszt-ben, ha a `range` fel lenne töltve a lista összes elemével, gyorsan kihasználná a számítógép memóriáját, és a program összeomlana. De ennél okosabb! Ez a számítás jól működik, mert a `range` objektum csak ígéret ad az elemek előállítására, amikor szükséges. Amint a ha feltétel igazgá válik, nem generál további elemeket, és a függvény visszatér. (Megjegyzés: A Python 3 előtt a `range` nem volt lusta. Ha a Python korábbi verzióját használja, YMMV!)

YMMV: Your Mileage May Vary (A kilométer teljesítményed változhat)

A YMMV rövidítés azt jelenti, hogy a kilométer teljesítményed változhat. Az amerikai autós hirdetések gyakran megemlítették az autók üzemanyag-felhasználási adatait, például, hogy az autó gallonként 28 mérföldet tehet meg. De ezt mindig egy apró-betűs jogi résznek kell kísérnie, figyelmeztetve az olvasót, hogy lehet nem fognak ugyanannyit kapni. Az YMMV kifejezést a köznyelvben úgy használjuk, hogy „az eredmények eltérhetnek”, például *A telefon akkumulátorának élettartama 3 nap, de YMMV.*

Néha találkozhattuk a lusta `range`-el, amely egy `list` hívásába van beágyazva. Ez arra kényszeríti Pythont, hogy a lusta ígéretét egy listává változtassa:

```
1 print(range(10))           # Hozzon létre egy lusta ígéretet
2 print(list(range(10)))     # Hívja meg az ígéretet, mely létrehozza a listát
```

Az eredmény a következő:

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

11.19. Beágyazott listák

A **beágyazott lista** olyan lista, amely egy másik listában elemként jelenik meg. Ebben a listában a 3. indexű elem egy beágyazott lista:

```
1 beagyazott = ["hello", 2.0, 5, [10, 20]]
```

Ha a lista 3. elemét kiírjuk a következőt kapjuk:

```
1 print(beagyazott[3])
```

```
[10, 20]
```

Ha a beágyazott listának egy elemét ki akarjuk írni, ezt két lépésben tehetjük meg:

```
1 elem = beagyazott[3]
2 print(elem[0])
```

```
10
```

Vagy kombinálhatjuk őket:

```
1 print(beagyazott[3][1])
```

```
20
```

A zárójel operátor kiértékelése balról jobbra történik, tehát a kifejezés megkapja a beagyazott lista 3. elemének első elemét.

11.20. Mátrixok

A beágyazott listákat gyakran használják a mátrixok ábrázolásánál. Például legyen a következő mátrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

amit ábrázolni lehet, mint:

```
1 mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`m` egy három elemű lista, mely mindegyik eleme a mátrix egy sora.

A mátrixból egy teljes sort kiválaszthatunk a szokásos módon:

```
1 print(m[1])
```

```
[4, 5, 6]
```

Vagy kiválaszthatunk egy elemet a mátrixból, a kettős-indexet használva:

```
1 print(m[1][2])
```

```
6
```

Az első index kiválasztja a sort, a második pedig az oszlopot. Habár a mátrixok ábrázolásának ezen módja a gyakoribb, ez nem az egyetlen lehetőség. Kevesebb változatot használhatunk az oszlopok listájára, mint a sorokra. Később további radiálisabb alternatívát fogunk látni a szótár használatával.

11.21. Szójegyzék

beágyazott lista (nested list) Egy lista, amelynek egy eleme egy másik lista.

elem (element, item) Egy érték a listából. A szögletes zárójel operátor segítségével választjuk ki az elemet a listából.

fedőnevek (aliases) Több változó, amelyek ugyanazon objektumra hivatkoznak.

határoló (delimiter) Olyan karakter vagy karakterlánc, amely jelzi, hol kell szétválasztani egy sztringet.

index (index) Egy egész szám, amely jelöli egy elem listán belüli pozícióját. Az indexek 0-tól kezdődnek.

ígéret (promise) Egy olyan objektum, amely megígéri, hogy valamilyen munkát elvégez vagy valamilyen értéket kiszámol, ha szükség van rá, de lustán végzi el a munkát (nem azonnal). A `range` hívása ígéretet eredményez.

klónozás (clone) Új objektum létrehozása, melynek ugyanaz az értéke, mint egy meglévő objektumnak. Egy objektumra mutató hivatkozás másolása fedőnevet hoz létre, de nem klónozza az objektumot.

lépésköz (step size) A lineáris sorozatban az egymást követő elemek közötti intervallum. A `range` függvénynek a harmadik (és opcionális) argumentumát lépés méretnek nevezzük. Ha nincs megadva, az alapértelmezett értéke 1.

lista (list) Értékek gyűjteménye. Mindegyik értéknek meghatározott helye van a listában. Más típusokhoz hasonlóan `str`, `int`, `float`, `stb.` van egy `list` típus-átalakító függvény is, amely bármely argumentumát listává próbálja alakítani.

lista bejárás (list traversal) A lista minden egyes elemének sorrendben történő elérése.

mellékhatás (side effect) Egy program állapotának megváltoztatása a hívó függvény által. A mellékhatásokat csak módosítókkal lehet előállítani.

minta (pattern) Utasítások sorozata vagy olyan kódolási stílus, amely általánosan alkalmazható számos különböző helyzetben. Érett informatikussá válik az, aki megtanulja, létrehozza az eszközkészletet alkotó mintákat és algoritmusokat. A minták gyakran megfelelnek a mentális blokkosításnak.

módosítható adat típusok (mutable data value) Olyan adat értékek, amelyek módosíthatók. Minden módosítható értéktípus összetett. A listák és szótárok módosíthatók, a sztringek és a rendezett `n`-esek nem.

módosító (modifier) Olyan függvény, amely megváltoztatja az argumentumokat a függvény törzsében. Csak a módosítható típusok változtathatók meg.

objektum (object) Egy „dolog”, amelyre egy változó hivatkozhat.

sorozat (sequence) Bármilyen olyan adat típus, mely rendezett elemeket tartalmaz, és minden elemet egy index-el azonosítunk.

tiszta függvény (pure function) Olyan függvény, mely nem okoz mellékhatásokat. A tiszta függvények csak a visszatérítési értékekkel okozhatnak változást a hívó függvényben.

változtathatatlan adat érték (immutable data value) Olyan adatérték, amelyet nem lehet módosítani. Az értékadások a megváltoztathatatlan elemek vagy szeletek esetén futási idejű hibát okoznak.

11.22. Feladatok

1. Mi lesz a Python kód eredménye a következő utasítás esetén?

```
list(range(10, 0, -2))
```

A range függvény három argumentuma a *start*, *stop* és *step*. Ebben a példában a *start* nagyobb, mint a *stop*. Mi történik, ha a *start* < *stop* és a *step* < 0? Írj egy szabályt a *start*, a *stop* és a *step* közötti kapcsolatokra.

2. Tekintsük a következő kódrészletet:

```
1 import turtle
2
3 Eszti = turtle.Turtle()
4 Sanyi = Eszti
5 Sanyi.color("hotpink")
```

Ez a kódrészlet egy vagy két teknőc példányt hoz létre? A Sanyi színének megváltoztatása Eszti színét is meg fogja változtatni? Magyarázd el részletesen!

3. Rajzolj az a és b számára egy pillanatképet, a következő Python kód 3. sorának végrehajtása előtti és utáni állapotában:

```
1 a = [1, 2, 3]
2 b = a[:]
3 b[0] = 5
```

4. Mi lesz a következő programrészlet kimenete?

```
1 ez = ["Én", "nem", "vagyok", "egy", "csodabogár"]
2 az = ["Én", "nem", "vagyok", "egy", "csodabogár"]
3 print("Test 1: {0}".format(ez is az))
4 ez = az
5 print("Test 2: {0}".format(ez is az))
```

Adj *részletes* magyarázatot az eredményekről.

5. A listákat használhatjuk matematikai *vektorok* ábrázolására. Ebben és az ezt követő néhány gyakorlatban olyan függvényeket írunk le, amelyek végrehajtják a vektorok alapvető műveleteit. Hozz létre egy `vektorok.py` szkriptet, és írd bele az alábbi Python kódot, hogy mindegyiket letesztelhesd!

Írj egy `vektorok_osszege(u, v)` függvényt, amely paraméterként két azonos hosszúságú listát kap, és adjon vissza egy új listát, mely tartalmazza a megfelelő elemek összegét:

```
1 teszt(vektorok_osszege([1, 1], [1, 1]) == [2, 2])
2 teszt(vektorok_osszege([1, 2], [1, 4]) == [2, 6])
3 teszt(vektorok_osszege([1, 2, 1], [1, 4, 3]) == [2, 6, 4])
```

6. Írj egy `szorzas_skalarral(s, v)` függvényt, amely paraméterként egy `s` számot, és egy `v` listát kap, és visszatér a függvény a `v` lista `s` skalárral való szorzatával.

```
1 teszt(szorzas_skalarral(5, [1, 2])) == [5, 10]
2 teszt(szorzas_skalarral(3, [1, 0, -1])) == [3, 0, -3]
3 teszt(szorzas_skalarral(7, [3, 0, 5, 11, 2])) == [21, 0, 35, 77, 14]
```

7. Írj egy `skalaris_szorzat(u, v)` függvényt, amely paraméterként megkap két azonos hosszúságú számokat tartalmazó listát, és visszaadja a megfelelő elemek skaláris szorzatát.

```
1 teszt(skalaris_szorzat([1, 1], [1, 1])) == 2
2 teszt(skalaris_szorzat([1, 2], [1, 4])) == 9
3 teszt(skalaris_szorzat([1, 2, 1], [1, 4, 3])) == 12
```

8. *Extra matematikai kihívások:* Írj egy `vektorialis_szorzat(u, v)` függvényt, amely paraméterként megkap két 3 hosszúságú számokból álló listát, és visszatér a vektoriális szorzatukkal. Írd meg a saját tesztjeid!

9. Írd le a `" ".join(nota.split())` és `nota` közötti kapcsolatot az alábbi kódrészletben. Ugyanazok a sztringek vannak hozzárendelve a `nota`-hoz? Mikor lennének különbözőek?

```
1 nota = "Esik eső, szép csendesen csepereg..."
```

10. Írj egy `cserel(s, regi, uj)` függvényt, amely kicseréli a `regi` összes előfordulását a `uj`-ra az `s` sztringben.

```
1 teszt(cserel("Mississippi", "i", "I")) == "MIssIssIppI"
2
3 s = "Kerek a gömb, gömbszerű!"
4 teszt(cserel(s, "öm", "om")) ==
5     "Kerek a gomb, gombszerű!"
6
7 teszt(cserel(s, "o", "ö")) ==
8     "Kerek a gömb, gömbszerű!"
```

Tipp: Használd a `split` és `join` metódusokat.

11. Tegyük fel, hogy két változó értékét akarjuk felcserélni. Újra felhasználható függvényt hozz létre, írd bele az alábbi kódot:

```
1 def csere(x, y):          # Hibás változat
2     print("csere utasítás előtt: x:", x, "y:", y)
3     (x, y) = (y, x)
4     print("csere utasítás után: x:", x, "y:", y)
5
6 a = ["Ez", "nagyon", "érdekes"]
7 b = [2, 3, 4]
8 print("csere függvény hívása előtt: a:", a, "b:", b)
9 csere(a, b)
10 print("csere függvény hívása után: a:", a, "b:", b)
```

Futtasd a fenti programot, és írd le az eredményeket. Hoppá! Nem azt tette, amit szerettünk volna! Magyarázd el miért nem. Használd a Python megjelenítőt, amely segítségével építs egy működő koncepcionális modellt! Mi lesz az `a` és `b` értéke a `csere` függvény hívása után?

12. fejezet

Modulok

A **modul** egy olyan fájl, amely Python definíciókat és utasításokat tartalmaz, más Python programokba is felhasználható. Számos Python modul létezik, amely a **standard könyvtár** része. Korábban ezek közül már legalább kettőt láthattunk, a `turtle` és a `string` modult.

A Python dokumentációban megtalálhatod az elérhető standard modulok listáját, és további információkat olvashatsz a használatukról. Kísérletezz, böngéssz a modulokról szóló dokumentációt!

12.1. Véletlen számok

Gyakran szeretnénk véletlen számokat használni a programokban, itt láthatjuk néhány tipikus felhasználását:

- A szerencsejátékok játszásánál, ahol a számítógépnek kell dobókockát dobni, vagy egy számot választani, vagy egy érmet feldobni,
- A játékkártyák véletlenszerű kiosztásánál,
- Egy ellenséges űrhajó véletlenszerű helyen való megjelenítéséhez, amely elkezd löni a játékosra,
- Az esetleges esőzések szimulálásánál, amikor egy számítógépes modellt készítünk a gátak építése során a környezeti hatások megbecsüléséhez,
- Internetes banki munkamenetek titkosításánál.

A Python egy `random` modult biztosít, amely segít az ilyen típusú feladatok megoldásánál. A Python dokumentáció segítségével rákereshetsz, de itt vannak a legfontosabb dolgok, amelyeket elvégezhetünk vele:

```
1 import random
2
3 # Létrehoz egy fekete doboz objektumot, amely véletlen számokat generál
4 rng = random.Random()
5
6 kocka_dobas = rng.randrange(1,7) # Vissza ad egy egész értéket, az 1, 2, 3, 4, 5, 6 számok egyikét
7 kesleltetes_masodpercben = rng.random() * 5.0
```

A `randrange` metódus hívása egy egész számot generál a megadott alsó és felső argumentum között, ugyanazt a szemantikát használja, mint a `range` – tehát az alsó korlátot tartalmazza, de a felső korlátot nem. Valamennyi érték azonos valószínűséggel jelenik meg, tehát az eredményként kapott értékek *egyenletes* eloszlást követnek. A `range`-hez hasonlóan a `randrange` is felvehet egy opcionális lépésköz argumentumot. Tegyük fel, hogy 100-nál kevesebb véletlenszerű páratlan számra van szükségünk:

```
1 r_paratlan = rng.randrange(1, 100, 2)
```

Más módszerek is képesek eloszlások generálására, például: egy harang görbe vagy a „normális” eloszlás alkalmasabb lehet az évszakos csapadék becsléséhez, vagy a gyógyszer bevétele után, a szervezetben egy vegyület koncentrációjának kiszámítására.

A `random` metódus egy valós számot ad vissza a $[0,0; 1,0)$ intervallumban – a bal oldalon szereplő szögletes zárójel „zárt intervallumot” jelent, és a kerek zárójel a jobb oldalon pedig „nyílt intervallumot”. Más szavakkal a 0,0 érték generálása lehetséges, de az összes visszaadott szám szigorúan kevesebb, mint 1,0. Megszokott dolog az, hogy *skalázzuk* az eredményt a metódushívása után, hogy az alkalmazásának megfelelő intervallumot kapjunk. Ebben az esetben a metódus hívás eredményét egy $[0,0; 5,0)$ intervallumra konvertáltuk. Tehát ezek egyenletesen eloszlott számok – a 0-hoz közelállóak ugyanolyan valószínűek lesznek, mint a 0,5-höz vagy 1,0-hez közeliek.

Ez a példa azt mutatja, hogyan keverhető össze egy lista. (A `shuffle` nem működhet közvetlenül egy lusta ígérettel, ezért vegyük figyelembe, hogy először a `list` típuskonverziós függvénnyel kell átalakítani a `range` objektumot.)

```
1 kartyak = list(range(52)) # Egész számokat generál [0 .. 51] között,  
2                               # amely egy kártyacsomagot szimbolizál.  
3 rng.shuffle(kartyak)     # Véletlenszerűen összekeveri a kártyákat.
```

12.1.1. Ismételhetőség és tesztelés

A véletlenszám generátorok **determinisztikus** algoritmuson alapulnak – ismételhetők és megjósolhatók. Tehát **pseudo-véletlen** generátoroknak hívják őket – nem igazán véletlenszerűek. Egy *kezdeti* értékkel indulnak. Minden alkalommal, amikor egy másik véletlen számot kérünk, az aktuális kezdőérték (ami a generátor egyik attribútuma) alapján kerül meghatározásra.

A hibakeresés és az egységtesztek esetén célszerű ismételhető programot írni, amelyek ugyanazt csinálják minden egyes futáskor. Ezt úgy valósítjuk meg, hogy a véletlenszám generátort minden alkalommal egy ismert kezdőértékkel indítjuk. (Gyakran csak tesztelés alatt akarjuk ezt – a kártyajátékok esetén, ha a kiosztott kártyák mindig ugyanolyan sorrendben lennének, mint a legutóbb lejátszott játék során, ez nagyon gyorsan unalmassá válna!)

```
1 drng = random.Random(123) # Létrehozza az ismert indítási állapotot
```

A véletlenszám generátor létrehozásának ezen alternatív módja explicit kezdőértéket ad az objektumnak. Ennek az argumentumnak a hiányában a rendszer valószínűleg valamilyen alapértéket használ az aktuális rendszeridő alapján. Tehát a véletlenszerű számok generálása a `drng`-nal ma pontosan ugyanazt a véletlen sorozatot adja, mint holnap!

12.1.2. Golyóhúzás, kockadobás, kártyakeverés

Íme egy példa egy olyan lista létrehozására, amely n véletlenszerűen generált egészet tartalmaz az alsó és a felső határ között:

```
1 import random  
2  
3 def random_egeszek(szam, also_hatar, felso_hatar):  
4     """ Véletlenszerűen generáljunk egy megadott számú egészeket tartalmazó  
5     ↪ listát az alsó és felső határ között. A felső határ nyitott. """  
6     rng = random.Random() # Hozzunk létre egy véletlenszám generátort.  
7     eredmeny = []  
8     for i in range(szam):  
9         eredmeny.append(rng.randrange(also_hatar, felso_hatar))  
10    return eredmeny
```

```
1 print(random_egeszek(5, 1, 13)) # Válasszunk 5 egész számot véletlenszerűen,  
   ↪ a hónapok sorszámaiból
```

```
[8, 1, 8, 5, 6]
```

Láthatjuk, hogy az eredményben duplikátumokat kaptunk. Gyakran szeretnénk ezt, például: ha ötször dobunk egy kockával, elvárjuk, hogy legyenek duplikátumok.

De mi van akkor, ha nem akarunk duplikátumokat? Ha 5 különböző hónapot szeretnénk, akkor ez az algoritmus nem megfelelő. Ebben az esetben a jó algoritmus az lesz, ha generálunk egy listát ami, a különböző lehetőségeket tartalmazza, majd összekeverjük a lista elemeket, és szeleteléssel kivágunk annyi elemet, amennyire szükségünk van.

```
1 xs = list(range(1,13)) # Létrehozunk egy listát 1..12-ig (itt nincsenek  
   ↪ ismétlődések)  
2 rng = random.Random() # Készítünk egy véletlenszám generátort  
3 rng.shuffle(xs)       # Összekeverjük a listát  
4 eredmény = xs[:5]     # Vesszük az első öt elemet
```

A statisztikai kurzusokon, az első esetet – amely lehetővé teszi a duplikátumokat – általában úgy írják le, mint a *visszatevéses* húzást – minden húzás alkalmával visszatesszük a golyókat, tehát így újra kihúzhatóak. Az utóbbi esetet, amikor nincsenek duplikátumok általában a *visszatevés nélküli* húzással jellemzik. Miután egy golyót kihúztunk, nem tesszük vissza, hogy újra előforduljon. A TV-lottós játékok is így működnek.

A második „kever és szeletel” algoritmus nem megfelelő abban az esetben, ha csak néhány elemet szeretnénk egy nagyon nagy tartományból. Tegyük fel, hogy öt számot szeretnénk egy és tízmillió között, duplikátum nélkül. A lista generálása tíz millió elemmel, majd összekeverése, és az első öt kivágása a teljesítmény szempontjából katasztrofális lenne. Tegyük egy másik próbát:

```
1 import random  
2  
3 def random_egeszek_duplikatum_nelkul(szam, also_hatar, felső_hatar):  
4     """ Véletlenszerűen generáljunk egy megadott számú egészeket tartalmazó,  
   ↪ listát  
5     az alsó és felső határ között. A felső határ nyitott. Az eredménylista,  
   ↪ nem  
6     tartalmazhat duplikátumokat. """  
7     eredmény = []  
8     rng = random.Random()  
9     for i in range(szam):  
10        while True:  
11            választott = rng.randrange(also_hatar, felső_hatar)  
12            if választott not in eredmény:  
13                break  
14            eredmény.append(választott)  
15        return eredmény  
16  
17 xs = random_egeszek_duplikatum_nelkul(5, 1, 10000000)  
18 print(xs)
```

Ez az algoritmus 5 véletlenszámot állít elő duplikátumok nélkül:

```
[3344629, 1735163, 9433892, 1081511, 4923270]
```

Habár ez a függvény is okozhat buktatókat. Mi történik az alábbi esetben?

```
1 xs = random_egeszek_duplikatum_nelkul(10, 1, 6)
```

12.2. A time modul

Ahogy egyre kifinomultabb algoritmusokkal és nagyobb programokkal kezdünk dolgozni, természetesen feltevődik a kérdés, hogy: „*hatékony a kódunk?*” A kísérlet egyik módja az, hogyha megnézzük, mennyi ideig tartanak a különböző műveletek. A `time` modul `clock` függvényét erre a célra ajánlják. Amikor meghívjuk az `clock` függvényt, egy valós számmal fog visszatérni, mely meghatározza, hogy hány másodperc telt el a program futása óta.

A használat módja, hogy meghívja a `clock` függvényt, és hozzárendeli a visszaadott értéket a `t0` változóhoz, mielőtt még elkezdenénk a kódot futtatni, melyet meg szeretnénk mérni. A végrehajtás után ismét meghívja a `clock` függvényt (ezt az időt a `t1` változóba fogja menteni.) A `t1-t0` különbsége lesz az eltelt idő és annak mértéke, hogy a program milyen gyorsan fut.

Nézzünk egy kis példát. A Python tartalmaz egy beépített `sum` függvényt, amely összegezi az elemeket a listában. Mi is írhatunk egy saját összegző függvényt. Mit gondolsz, hogyan lehetne összehasonlítani a sebességeket? Mindkét esetben végezd el a `[0,1,2,...]` lista összegzését, és hasonlítsd össze az eredményeket:

```
1  import time
2
3  def saját_szum(xs):
4      szum = 0
5      for v in xs:
6          szum += v
7      return szum
8
9  sz = 10000000          # Legyen 10 millió eleme a listának
10 testadat = range(sz)
11
12 t0 = time.clock()
13 saját_eredmeny = saját_szum(testadat)
14 t1 = time.clock()
15 print("saját_eredmény = {0} (eltelt ido = {1:.4f} másodperc)"
16       .format(saját_eredmeny, t1-t0))
17
18 t2 = time.clock()
19 gepi_eredmeny = sum(testadat)
20 t3 = time.clock()
21 print("gépi_eredmény = {0} (eltelt idő = {1:.4f} másodperc)"
22       .format(gepi_eredmeny, t3-t2))
```

Egy átlagos laptopon a következő eredményeket kapjuk:

```
saját_eredmény = 49999995000000 (eltelt idő = 1.5567 másodperc)
gépi_eredmény  = 49999995000000 (eltelt idő = 0.9897 másodperc)
```

Tehát a mi függvényünk 57%-al lassabb, mint a beépített. 10 millió elem létrehozása és összegzése esetén egy másodperc alatti eredmény nem túl rossz!

12.3. A math modul

A `math` modul olyan matematikai függvényeket tartalmaz, amelyeket általában a számológépeken találhatsz (`sin`, `cos`, `sqrt`, `asin`, `log`, `log10`), és néhány matematikai állandót, mint a `pi` és `e` konstansok:

```
1  import math
2  print(math.pi)          # A pi konstans
3  print(math.e)           # A természetes logaritmus alap, Euler_
  konstans
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4 print(math.sqrt(2.0))          # A négyzetgyök függvény
5 print(math.radians(90))        # 90 fok konvertálása radiánra
6 print(math.sin(math.radians(90))) # A sin(90) fok
7 print(math.asin(1.0) * 2)      # Az arcsin(1.0) kétszerese, megadja a pi-t
```

Az eredmények a következők:

```
3.141592653589793
2.718281828459045
1.4142135623730951
1.5707963267948966
1.0
3.141592653589793
```

Mint minden más programozási nyelvhez hasonlóan a szögek inkább *radiánban*, mint fokban vannak kifejezve. Két függvény áll a rendelkezésünkre, a `radians` és a `degrees`, hogy konvertáljuk a szögeket a két mérési mód között.

Figyeljünk meg egy másik különbséget ezen modul, és a `random`, `turtle` modulok használati módja között: létrehozunk a `random` és `turtle` objektumokat, és meghívjuk a metódusokat az objektumokra. Ennek az oka az, hogy objektumoknak van *állapota* – a teknőcnek van színe, pozíciója, címsora, stb., és minden véletlenszám generátornak van egy kezdeti értéke, mely meghatározza a következő értéket.

A matematikai függvények „tiszták”, és nincs semmilyen állapotuk – 2.0 négyzetének kiszámítása nem függ semmilyen állapottól vagy előzménytől, mely a múltban történt. Tehát ezen függvények nem metódusai egy objektumnak – egyszerűen olyan függvények melyeket a `math` modulban csoportosítottak.

12.4. Saját modul létrehozása

Saját modul létrehozása esetén mindössze csak annyit kell tennünk, hogy elmentjük a szkriptet `.py` kiterjesztésű fájlként. Feltételezzük például, hogy ezt a szkriptet `resztorles.py` néven mentettük:

```
1 def torol(poz, sor):
2     return sor[:poz] + sor[poz+1:]
```

Mostantól használhatjuk a modulunkat a szkriptekben vagy az interaktív Python parancsértelmezővel. Viszont ehhez először be kell importálni a modult.

```
1 import resztorles
2 s = "Egy asztalt!"
3 resztorles.torol(7, s)
```

```
'Egy aszalt!'
```

A `.py` fájlkiterjesztést nem írjuk ki az importálásnál. A Python elvárja, hogy a modulok nevei `.py`-al végződjenek, így a fájlkiterjesztést nem kell hozzátenni az **import** utasításnál.

A modulok használata lehetővé teszi, hogy a nagyméretű programokat könnyen kezelhető részekre bontsuk, de a kapcsolódó részeket együtt kezeljük.

12.5. Névterek

A **névterek** olyan azonosítók gyűjteményei, amelyek vagy egy modulhoz vagy egy függvényhez tartoznak (hamarosan látni fogjuk az osztályoknál is). Általában olyan névtereket kedvelünk, melyek egymáshoz „kapcsolódnak”, mint

például az összes matematikai függvény vagy az összes olyan tipikus dolog, amiket véletlenszerű számokkal végzünk.

Minden modul saját névtérrel rendelkezik, így ugyanazt az azonosítónevet használhatjuk több modulban anélkül, hogy az azonosítók problémát okoznának.

```
1 # Modul1.py
2
3 kerdes = "Mi a jelentése az Életnek, a Világegyetemnek és a Mindenségnek?"
4 valasz = 42
```

```
1 # Modul2.py
2
3 kerdes = "Mi a küldetésed?"
4 valasz = "Keresni a Szent Grált!"
```

Most importálhatjuk mindkét modult, és hozzáférhetünk a kerdes-ekhez és valasz-okhoz:

```
1 import Modul1
2 import Modul2
3
4 print(Modul1.kerdes)
5 print(Modul2.kerdes)
6 print(Modul1.valasz)
7 print(Modul2.valasz)
```

a következőket kapjuk:

```
Mi a jelentése az Életnek, a Világegyetemnek és a Mindenségnek?
Mi a küldetésed?
42
Keresni a Szent Grált!
```

A függvények saját névtérrel is rendelkeznek:

```
1 def f():
2     n = 7
3     print("n kiírása az f-ben:", n)
4
5 def g():
6     n = 42
7     print("n kiírása a g-ben:", n)
8
9 n = 11
10 print("n kiírása az f hívása előtt:", n)
11 f()
12 print("n kiírása az f hívása után:", n)
13 g()
14 print("n kiírása a g hívása után:", n)
```

A program futtatása a következő kimenetet eredményezi:

```
n kiírása az f hívása előtt: 11
n kiírása az f-ben: 7
n kiírása az f hívása után: 11
n kiírása a g-ben: 42
n kiírása a g hívása után: 11
```

A három `n` itt nem ütközik, mivel mindegyikük egy másik névtérben van – három különböző változónak ugyanaz a neve, ugyanúgy, mintha három különböző embert „Péter”-nek hívnak.

A névterek lehetővé teszik, hogy több programozó ugyanazon projekten dolgozhasson ütközés nélkül.

Mi a kapcsolat a névterek, a fájlok és a modulok között?

A Pythonnak egy kényelmes és egyszerűsítő „egy-az-egyhez” rendszere van, ami azt jelenti, hogy egy modulhoz egy fájl tartozik, amelyhez egy névtér kapcsolódik. A Python a fájl névből határozza meg a modul nevét, amely a névtér nevévé is válik. A `math.py` a fájlnev, a modul neve `math` és a névtér szintén `math`. Így Pythonban a fogalmak többé-kevésbé felcserélhetők.

De találkozni fogsz más nyelvekkel (például: C#), ahol engedélyezve vannak olyan modulok, amelyek több fájlt is tartalmazhatnak, vagy egy fájlnak lehet több névtére, vagy több fájl ugyanazt a névtérrel használja. Vagyis a fájl neve nem feltétlenül egyezik meg a névtérrel.

Tehát jó ötlet, ha megpróbáljuk fejben elkülöníteni a fogalmakat.

A fájlok és könyvtárak rendszerezik a számítógépünkben tárolt adatok helyét. Másrésről a névterek és modulok olyan programozási koncepciók, melyek segítenek nekünk megszervezni, hogyan szeretnénk csoportosítani a kapcsolódó függvényeket és attribútumokat. Tehát nem arról szól, hogy hol tároljuk az adatokat, és nem kell egybe esnie a fájl és könyvtár szerkezeteknek.

Pythonban ha átnevezzük a `math.py` fájlt, módosul a modul neve is, az `import` utasításokat meg kell változtatni, és a kódot, amely a névtéren belüli függvényekre vagy attribútumokra utal, szintén módosítani kell.

Más nyelvekben nem feltétlenül ez a helyzet. Ne ködösrítsük el a fogalmakat, csak azért, mert a Python ezt teszi!

12.6. Hatókör és keresési szabályok

Az azonosító hatóköre a programkód olyan része, amelyben az azonosító elérhető vagy használható.

Három fontos **hatókört** különböztetünk meg a Pythonban:

- A **lokális hatókör** egy függvényben deklarált azonosítókra hivatkozik. Ezek az azonosítók a függvényhez tartozó névtérben vannak tárolva, és minden függvénynek van saját névtére.
- A **globális hatókör** az aktuális modulon vagy fájlban belül deklarált összes azonosítóra vonatkozik.
- A **beépített hatókör** a Pythonban deklarált összes azonosítóra vonatkozik – olyan, mint a `range` és a `min`, melyeket bármikor használhatunk, anélkül, hogy importálnánk és (szinte) mindig elérhetők.

A Python (mint a legtöbb egyéb számítógépes nyelv) a precedencia szabályait használja: ugyanaz a név több, mint egy hatókörön belül is előfordulhat, de a legbelső vagy a lokális hatókör mindig elsőbbséget élvez a globális hatókör felett, és a globális hatókört mindig előnyben részesítjük a beépített hatókörrel szemben. Kezdjük egy egyszerű példával:

```
1 def range(n):  
2     return 123*n  
3  
4 print(range(10))
```

Mit fog kiírni? Definiáltuk a saját `range` függvényünket, tehát most kétértelműség lehetséges. Amikor a `range`-t használjuk, a sajátunkat vagy a beépítettet értjük ezalatt? A hatókör keresési szabályok határozzák meg a választ, mely alapján: a saját `range` függvény van meghívva, nem a beépített, mivel a saját `range` függvény a globális névtérben van, mely precedenciája magasabb, mint a beépítetté.

Habár az olyan nevek mint a `range` és `min` beépítettek, de azok a használat szempontjából „elrejtethők”, ha a saját változóinkat vagy függvényeinket definiáljuk, amelyek újra felhasználják azokat a neveket. (Összeshazárhat bennünket

az újra definiált beépített nevek használata – tehát nagyon jó programozónak kell lenni, hogy megértsük a hatókör szabályait, és megértsük azokat a zavaró tényezőket, amelyek a kétértelműséget okozzák, tehát kerüljük ezek használatát!)

Most nézzünk egy kissé összetettebb példát:

```
1 n = 10
2 m = 3
3 def f(n):
4     m = 7
5     return 2*n+m
6
7 print(f(5), n, m)
```

Kiírja a 17 10 3-at. Ennek az az oka, hogy a `m` és `n` két változó az 1. és 2. sorokban a függvényen kívül a globális névtérben vannak deklarálva. A függvényben az `n` és `m` új változóként az *f* függvény végrehajtásának időtartamára vannak létrehozva. Ezeket az *f* függvény helyi névtérében hozzuk létre. A *f* függvény törzsében a hatókör keresési szabályok határozzák meg a helyi `m` és `n` változók használatát. Ezzel szemben, miután az *f* függvény visszatér, az `n` és `m` argumentumok a `print` függvény során az eredeti változókra utalnak, az 1. és 2. sorokra, és ezek a változók nem módosultak az *f* végrehajtása során.

Figyeljük meg, hogy itt a `def` helyezi el az *f*-et a globális névtérben! Tehát a 7. sorban tudjuk meghívni.

Mi lesz az 1. sorban található `n` változó hatóköre? Hatókör – az a terület, amelyben láthatók – az 1., 2., 6., 7. sor. Mivel az `n` lokális változó, el van rejtve a 3., 4., 5. sorokban.

12.7. Attribútumok és a pont operátor

A modulban definiált változókat a modul **attribútumainak** nevezzük. Ahogyan már láttuk az objektumoknak is vannak attribútumai: például a legtöbb objektum rendelkezik a `__doc__` attribútummal, és néhány függvénynek van `__annotation__` attribútuma. Az attribútumok a **pont operátorral** (`.`) érhetők el. A `modul1` és `modul2` nevű modulok `kerdes` attribútuma elérhető, használva a `modul1.kerdes` és `modul2.kerdes`-t.

A modulok függvényeket és attribútumokat tartalmaznak, a pont operátort használva tudjuk elérni őket. A `resztorles.torol` hivatkozik a `torol` függvényre a `resztorles` modulban.

Amikor pontozott nevet használunk, gyakran hivatkozunk rá, úgy mint **teljesen minősített névre**, mert pontosan megmondjuk, hogy melyik `kerdes` attribútumot értjük alatta.

12.8. Az import utasítás három változata

Három különböző módon lehet neveket importálni az aktuális névtérbe és használni őket:

```
1 import math
2 x = math.sqrt(10)
```

Itt csak ez egyszerű `math` azonosítót adtuk hozzá az aktuális névtérhez. Ha egy modul függvényeit szeretnénk elérni, akkor a pont jelzést kell használnunk:

Itt látható egy másik elrendezés:

```
1 from math import cos, sin, sqrt
2 x = sqrt(10)
```

A nevek közvetlenül az aktuális névtérbe kerülnek, és minősítés nélkül felhasználhatók. A `math` nevet nem importáljuk, ezért ha használjuk a minősített `math.sqrt` formát, hibát kapunk.

Egy egyszerű rövidítéssel:

```
1 from math import *    # Importáljuk az összes azonosítót a math-ból,  
2                       # és hozzáadjuk az aktuális névtérhez.  
3 x = sqrt(10)          # Használjuk minősítés nélkül
```

A három módszer közül az első általában a kedveltebb, még akkor is, ha egy kicsivel több gépelést jelent. Habár lerövidíthetjük a modulokat úgy, hogy másik néven importáljuk őket:

```
1 import math as m  
2 print(m.pi)
```

```
3.141592653589793
```

De egy jó kis szerkesztővel, amelynek van automatikus kiegészítője, és gyors ujjainkkal ez kifizetődő.

Végül figyeljük meg az alábbi esetet:

```
1 def terület(sugar):  
2     import math  
3     return math.pi * sugar * sugar  
4  
5 x = math.sqrt(10)    # Hibát eredményez
```

Itt a `math` modult importáltuk, viszont az importálás a `terület` lokális névtérében történt. Tehát a név használható a függvény törzsén belül, de az 5. sorban nem, mivel a `math` importálása nem a globális névtérben van.

12.9. Az egységtesztelődöt alakítsd modullá

A 6. fejezet (Produktív függvények) vége fele bevezettük az egységtesztet, és a megírtuk saját `teszt` függvényeinket, melyeket be kellett másolni minden egyes modulba, amelyekre a teszteket írtuk. Most ezt saját modullá alakítjuk, legyen a neve `egyseg_teszt.py` és inkább használjunk a következő sort, minden egyes új szkriptben:

```
1 from egyseg_teszt import teszt
```

12.10. Szójegyzék

attribútum (attribute) Egy modulon belül definiált változó (vagy osztály, vagy példány – lásd később). A modulok attribútumai minősítéssel érhetőek el, a **pont operátort** (`.`) használva.

import utasítás (import statement) Egy olyan utasítás, amely a modulban lévő objektumokat egy másik modulon belül elérhetővé teszi. Az import utasítás használatának két formája van. A `sajatmod1` és `sajatmod2` elnevezésű hipotetikus modulokat használva, amelyeknek mindegyike tartalmazza az `f1` és `f2` függvényeket, valamint a `v1` és `v2` változókat.

```
1 import sajátmod1  
2 from sajátmod2 import f1, f2, v1, v2
```

A második forma az importált objektumokat az importáló modul névtérébe viszi, míg az első forma megőrzi egy külön névtérrel az importált modul számára, ezért ebben az esetben kötelező a `sajatmod1.v1` forma használata, ha hozzá akarunk férni a modul `v1` változójához.

metódus (method) Az objektum függvénytípusú attribútuma. A metódusok meghívása egy objektumra a pont operátor segítségével történik. Például:

```
1 s = "ez egy sztring."  
2 print(s.upper())
```

```
'EZ EGY SZTRING.'
```

Azt mondjuk, hogy az `upper` metódus meghívjuk az `s` sztringre, `s` implicit módon az első argumentuma az `upper`-nek.

modul (module) Python definíciókat és utasításokat tartalmazó fájl, melyet más Python programokban is használhatunk. A modul tartalma elérhetővé válik másik program számára az `import` utasítás által.

névtér (namespace) Szintaktikai konténer, amely kontextusba helyezze a neveket, hogy ugyanaz a név különböző névterekben is létezhesen kétértelműség nélkül. Pythonban a modulok, osztályok, függvények és metódusok mind névtérre alkotnak.

névütközések (naming collision) Olyan szituáció, amelyben két vagy több név ugyanabban az adott névtérben nem határozható meg egyértelműen. Használva az

```
1 import string
```

utasítást a

```
1 from string import *
```

helyett, megelőzhetjük a névütközéseket.

pont operátor (dot operator) A pont operátor (`.`), vagyis a minősítés, lehetővé teszi a modul attribútumainak és függvényeinek (vagy az osztály netán példány attribútumainak és metódusainak – ahogyan már korábban láthattuk) elérését.

standard könyvtár (standard library) A könyvtár egy olyan szoftver gyűjtemény, amelyeket eszközként használnak más szoftverek fejlesztésénél. A programozási nyelvek standard könyvtárai olyan eszközkészletet tartalmaznak, amelyeket az alap programozási nyelvekkel együtt adnak. Python egy kiterjesztett standard könyvtárral rendelkezik.

teljesen minősített név (fully qualified name) Olyan név, amelynek prefixe lehet néhány névtér azonosító és a pont operátor, vagy egy objektum példány, például: `math.sqrt` vagy `Eszti.forward(10)`.

12.11. Feladatok

1. Olvasd el a `calendar` modul dokumentációját.

(a) Próbáld ki a következőket:

```
1 import calendar  
2 naptar = calendar.TextCalendar()      # Hozd létre egy példányát!  
3 naptar.pryear(2017)                  # Mi történik itt?
```

(b) Figyeld meg, hogy a hét hétfőn kezdődik! A kalandvágó informatikus hallgató azt gondolja, hogy jobb felosztás lenne, ha a hét csütörtökön kezdődne, mert csak két munkanap lenne a hétvégeig, és minden hét közepén szünetet tarthatnának.

(c) Keress egy olyan függvényt, amelynek segítségével kiírhatod ebben az évben a születésnapodnak megfelelő hónapot!

(d) Próbáld ki ezt:

```
1 d = calendar.LocaleTextCalendar(6, "HUNGARIAN")
2 d.pryear(2017)
```

Próbáld ki néhány más nyelvet, beleértve egyet, amelyen nem működik, és figyeld meg, mi történik!

(e) Kísérletezz a `calendar.isleap`-el! Milyen argumentumok kér? Mi lesz a visszatérési értéke? Milyen függvény ez?

Készíts részletes jegyzetet arról, hogy mit tanultál ezekből a feladatokból!

2. Nyisd meg a `math` modul dokumentációját.

(a) Hány függvényt tartalmaz a `math` modul?

(b) Mit csinál a `math.ceil`? Mit a `math.floor`? (Tipp: mindkettő a `floor` és a `ceil` valós értéket vár argumentumként.)

(c) Írd le, hogyan számoltuk ki ugyanazt az értéket, mint a `math.sqrt`, a `math` modul használata nélkül.

(d) Mi a `math` modul két adat konstansa?

Készíts részletes jegyzetet a fenti feladatban elvégzett vizsgálatokról!

3. Vizsgáld meg a `copy` modult! Mit csinál a `deepcopy`? A legutóbbi fejezet mely feladataiban lenne hasznos a `deepcopy` használata?

4. Hozd létre a `sajatmodul1.py`-t! Az `sajatev` attribútumhoz rendeld hozzá az életkorod, és az `ev`-hez az aktuális évet! Hozd létre egy másik `sajatmodul2.py`-t! A `sajatev` attribútumot állítsd 0-ra, és az `ev` attribútumot arra az évre, amikor születted! Most hozd létre a `nevter_teszt.py` fájlt! Importáld mindkét fent megadott modult, és futtasd a következő utasítást:

```
1 print( (sajatmodul2.sajatev - sajatmodul1.sajatev) ==
2        (sajatmodul2.ev - sajatmodul1.ev) )
```

A `nevter_teszt.py` futtatásakor `True` vagy `False` kimenet jelenik meg, attól függően, hogy az időn volt-e már születésnapod.

Ez a példa szemlélteti, hogy a különböző modulok egyaránt rendelkezhetnek `sajatev` és `ev` nevű attribútumokkal. Mivel különböző névtérben vannak, ezért nem ütköznek egymással. Amikor a `nevter_teszt.py`-t megírjuk, pontosan meghatározzuk, hogy melyik `ev` és `sajatev` változókra hivatkozunk.

5. Írd a következő utasításokat a `sajatmodul1.py`, a `sajatmodul2.py` és a `nevter_teszt.py` fájlokhoz az előző gyakorlatból:

```
1 print("Az én nevem", __name__)
```

Futtassuk a `nevter_teszt.py`-t! Mi történik? Miért? Most adjuk hozzá a következőket a `sajatmodul1.py` végére:

```
1 if __name__ == "__main__":
2     print("Ez nem fog futni, ha importálok.")
```

Futtasd újra a `sajatmodul1.py` és `nevter_teszt.py`-t! Mely esetben látod az új kiíratást?

6. Próbáld ki a következőt:

```
import this
```

Tim Peters mit mond a névterekről?

7. Add meg a következő Python kód választ az utasítások mindegyikére, folyamatosan a végrehajtás során:

```
1 s = "Esik eső csendesesen, leperreg az ereszen..."
2 print(s.split())
3 print(type(s.split()))
4 print(s.split("e"))
5 print(s.split("s"))
6 print("0".join(s.split("e")))
```

Győződj meg arról, hogy megértetted miért kaptad az eredményeket! Ezután alkalmazd a tanultakat, és töltsd ki az alábbi függvény törzsét, használva a `split`, `join` metódusokat és az `str` objektumotokat:

```
1 def cserel(regi, uj, s):
2     """ Cseréld az s-ben a regi paraméter összes előfordulását az uj-ra. """
3     ...
4
5 teszt(cserel(",", ";", "ez, az, és valami más dolog") ==
6       "ez; az; és valami más dolog")
7 teszt(cserel(" ", "*", "A szavak most csillaggal vannak elválasztva.") ==
8       "A**szavak**most**csillaggal**vannak**elválasztva.")
```

A megoldásoknak át kell mennie a teszteken.

8. Készíts egy `wordtools.py` modult, mely lehetővé teszi az egységteszt használatát helyben.

Most add hozzá a függvényeket ezekhez a tesztekhez.

```
teszt(szo_tisztitas("hogyan?") == "hogyan")
teszt(szo_tisztitas("'most!'") == "most")
teszt(szo_tisztitas("?+='s-z-a-v!a,k@$( )'"') == "szavak")

teszt(van_duplavonal("kicsi--nagy") == True)
teszt(van_duplavonal("") == False)
teszt(van_duplavonal("magas--") == True)
teszt(van_duplavonal("piros--fekete") == True)
teszt(van_duplavonal("-igen-nem-") == False)

teszt(szavakra_bontas(" Most van itt az idő? Igen, most.") ==
      ['most', 'van', 'itt', 'az', 'idő', 'igen', 'most'])
teszt(szavakra_bontas("Ő megpróbált udariasan viselkedni!") ==
      ['ő', 'megpróbált', 'udvariasan', 'viselkedni'])

teszt(szavak_szama("most", ["most", "később", "soha", "most"]) == 2)
teszt(szavak_szama("itt", ["itt", "ott", "amott", "itt", "ott", "amott", "itt",
↪ ""]) == 3)
teszt(szavak_szama("tél", ["tavasz", "nyár", "ősz", "tél", "tavasz", "nyár",
↪ "ősz"]) == 1)
teszt(szavak_szama("kakukk", ["cinege", "fecske", "gólya", "sas", "veréb",
↪ "páva", "rigó"]) == 0)

teszt(szo_halmaz(["most", "van", "itt", "most", "van", "itt"]) ==
      ["itt", "most", "van"])
teszt(szo_halmaz(["én", "te", "ő", "én", "te", "ő", "mi", "én"]) ==
      ["én", "mi", "ő", "te"])
teszt(szo_halmaz(["egy", "kettő", "három", "négy", "öt", "hat", "hét",
↪ "nyolc"]) ==
      ["egy", "három", "hat", "hét", "kettő", "négy", "nyolc", "öt"]])
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
teszt(leghosszabb_szo(["alma", "eper", "körte", "szőlő"]) == 5)
teszt(leghosszabb_szo(["én", "te", "ő", "mi"]) == 2)
teszt(leghosszabb_szo(["ez", "szórakoztatóelektronikai"]) == 24)
teszt(leghosszabb_szo([ ]) == 0)
```

Mentsd el ezt a modult, hogy használhasd az eszközeit a jövőbeni programjaiban!

13. fejezet

Fájlok

13.1. Fájlokról

A program futása alatt, az adatok a *Random Access Memory*-ban (RAM) vannak tárolva. A RAM gyors és olcsó, de **felejtő** memória, ami azt jelenti, hogy amikor a számítógépet kikapcsoljuk, a RAM-ból az adatok elvesznek. Ahhoz, hogy a számítógép bekapcsolásakor és a program indításakor az adatok elérhetőek legyenek, az adatokat egy **nem felejtő** adattárolóra, például merevlemezre, USB meghajtóra vagy CD-RW-re kell kiírni.

A nem felejtő adattárolókon tárolt adatokat **fájloknak** nevezzük. A fájlok olvasásával és írásával, a programok el tudják menteni az információkat a programfutások között.

A fájlok kezelése hasonlít a jegyzetfüzet kezeléséhez. Ha szeretnénk használni, először ki kell nyitnunk. Ha végeztünk, akkor be kell zárnunk. Amíg a jegyzetfüzet nyitva van, olvashatunk és írhatunk. A jegyzetfüzet tulajdonosa bármelyik esetben tudja, hogy hol tart benne. A teljes jegyzetfüzetet egy meghatározott sorrendben tudja olvasni, vagy át is ugorhat bizonyos részeket.

Mindezt alkalmazhatjuk a fájlokra is. A fájlok megnyitásához meg kell adnunk a fájl nevét, és azt, hogy olvasni vagy írni szeretnénk.

13.2. Első fájlunk írása

Kezdjünk egy egyszerű programmal, amely három sornyi szöveget ír a fájlba:

```
1 saját_fajl = open("első.txt", "w")
2 saját_fajl.write("Az első Python fájlom!\n")
3 saját_fajl.write("-----\n")
4 saját_fajl.write("Helló, világ!\n")
5 saját_fajl.close()
```

A fájl megnyitásával létrehozunk egy **kezelőt**, ebben az esetben egy fájlkezelőt. A fenti példában a `saját_fajl` változó hivatkozik az új kezelő objektumra. A programunk meghívja a fájlkezelő metódusait, amelyek módosítják az adott fájlt, és általában elhelyezik a lemezünkön.

Az első sorban az `open` függvény két argumentumot tartalmaz. Az első a fájl neve, és a második a megnyitás módja. A `"w"` **mód** azt jelenti, hogy megnyitja a fájlt írásra. A `"w"` móddal, ha nincs az `első.txt` fájl a lemezünkön, akkor létrehozza azt. Ha már van ilyen fájl, akkor kicseréli a fájlt arra, amit éppen írunk.

A fájlba történő adatbevitelnél meghívjuk a `write` metódust a kezelő objektumon, lásd 2-4. sorok. A nagyobb programoknál ezen sorok általában egy ciklussal vannak megvalósítva, mely segítségével több sort is írhatnak a fájlba.

A fájlkezelő bezárása (5. sor) jelzi a rendszernek, hogy elkészültünk az írással, és tegye a fájlt olvasásra elérhetővé más programok (vagy a saját programunk) számára.

A kezelő hasonlít a TV távirányítójára

Mindannyian ismerjük a TV távirányítóját. A távirányító műveletei: csatornaváltás, hangerő szabályozás, stb. De az igazi tevékenység a TV belsejében történik. Így egy egyszerű analógiával, a távirányítót nevezzük a TV kezelőjének.

Néha szeretnénk kihangsúlyozni a különbséget, a fájlkezelő nem ugyanaz, mint a fájl és a távirányító sem ugyanaz, mint a TV. De máskor szívesen kezeljük őket egyetlen mentális blokként vagy absztrakcióként, és azt mondjuk, hogy „zárd be a fájlt” vagy „váltás TV csatornát”.

13.3. Fájl soronkénti olvasása

Most, hogy a fájl létezik a háttérlemezünkön, meg tudjuk nyitni, és ki tudjuk olvasni a fájl minden sorát egyenként. A fájl megnyitásának módja az olvasásra: "r".

```
1 uj_kezelo = open("elso.txt", "r") # Nyisd meg a fájlt
2 while True:
3     sor = uj_kezelo.readline()      # Próbáld beolvasni a következő sort
4     if len(sor) == 0:               # Ha nincs több sor
5         break                      # Hagyd el a ciklust
6     # Most dolgozd fel az éppen aktuálisan beolvasott sort
7     print(sor, end=" ")
8
9 uj_kezelo.close()                  # Zárd be a fájlt
```

Ez egy praktikus minta az eszköztárunk számára. Nagyobb programok esetén, a ciklus törzsében lévő 8. sorban, bonyolultabb utasításokat is elhelyezhetünk – például, ha a fájl minden sora egy barátunk nevét és e-mail címét tartalmazza, szétdarabolhatnánk a sorokat kisebb részekre, és meghívhatnánk a függvényt, hogy küldjön egy buli meghívást a barátainknak.

A 8. sorban kihagytuk az újsor karaktert, amelyet a kiírás során általában hozzákapcsolunk a sztringünk végéhez. Miért? Ennek az az oka, hogy a sztringnek már van egy saját újsor karaktere, mivel a 3. sorban szereplő `readline` metódus egy olyan sorral tér vissza, amely már tartalmazza az újsor karaktert. Ez megmagyarázza a fájl végét jelző logikát is, ha már nincs több olvasandó sor, a `readline` egy üres sztringgel tér vissza, amelynek a végén nem szerepel az új sor karakter, tehát a hossza 0.

Elsőre kudarcs...

A mintánkban három sor van, mégis *négy*szer lépünk be a ciklusba. A Pythonban csak akkor tudjuk, hogy nincs több sor a fájlban, ha nem tudjuk beolvasni a következő sort. Néhány más programozási nyelv esetében (pl. Pascal) ezek különbözőek: ott 3 sort olvasunk, és meghívjuk az *előre néz* függvényt – tehát 3 sor beolvasása után már pontosan tudjuk, hogy nincs több sor a fájlban. Ebben az esetben nincs engedélyezve a negyedik sor olvasása.

Tehát Pascalban és Pythonban a soronkénti beolvasás különböző.

Ha a Python utasításokat egy másik számítógépes nyelvre fordítjuk, győződjünk meg arról, hogy hogyan értelmezi fájl végét: vagy „megpróbálja beolvasni a következő sort, és miután nem sikerül, tudja” vagy „előre tekint”?

Ha megpróbálunk megnyitni egy nem létező fájlt hibüzenetet kapunk:

```
1 uj_kezelo = open("nincs_ilyen.txt", "r")
```

```
FileNotFoundError: [Errno 2] No such file or directory: "nincs_ilyen.txt"
```

13.4. Fájl átalakítása sorok listájává

Gyakran hasznos lehet a fájl adatainak lekérése, és a sorok listává való átalakítása. Tegyük fel, hogy van egy fájlunk, amely soronként a barátaink nevét és e-mail címét tartalmazza. De azt szeretnénk, hogyha a sorok alfabetikus sorrendben lennének rendezve. A terv az, hogy soronként olvassunk, és a sorokat listába mentjük, majd rendezzük a listát, és a rendezett listát beírjuk egy másik fájlba:

```
1 f = open("baratok.txt", "r")
2 xs = f.readlines()
3 f.close()
4
5 xs.sort()
6
7 g = open("rendezett.txt", "w")
8 for v in xs:
9     g.write(v)
10 g.close()
```

A 2. sorban a `readlines` metódus beolvassa az összes sort, és visszatér a sztringek listájával.

Használhatjuk az előző fejezetben használt beolvasási sablont, hogy egy sort olvasunk egyszerre, és felépítjük a saját listánkat, de sokkal könnyebb használni ezt a módszert, amelyet a Python implementálók ajánlanak.

13.5. A teljes fájl beolvasása

A szöveges fájlok kezelésének másik módja az, hogy a fájl teljes tartalmát egy sztringbe mentjük, és használjuk a sztring-feldolgozási algoritmusokat.

Általában ezt a fájl-feldolgozó módszert használnánk, ha nem érdekelne bennünket a fájl sorainak szerkezete. Például, láthattuk a `split` metódust a sztringek esetében, amelyek feldarabolják a sztringeket szavakra. Tehát így meghatározhatnánk a fájlban szereplő szavak számát.

```
1 f = open("szoveg.txt")
2 tartalom = f.read()
3 f.close()
4
5 szavak = tartalom.split()
6 print("A fájl szavainak száma: {}".format(len(szavak)))
```

Vegyük észre, hogy az első sorban kihagytuk az `"r"` módot. Amennyiben nem adjuk meg Pythonban a megnyitás módját, alapértelmezetten mindig olvasásra nyitja meg a fájlt.

Előfordulhat, hogy a fájl útvonalát explicit módon szükséges megadni.

A fenti példában azt feltételeztük, hogy a `szoveg.txt` ugyanabban a könyvtárban van, mint a Python forráskód. Ha nem ez a helyzet, előfordulhat, hogy a fájl teljes vagy relatív útvonalát meg kell adni.

Windowsban a teljes elérési útvonalat a "C:\\temp\\szoveg.txt", míg Unixban "/home/peti/szoveg.txt" módon adhatjuk meg.

Ebben a fejezetben erre még később visszatérünk.

13.6. Bináris fájlok kezelése

Azon fájlokat, melyek fényképeket, videókat, zip fájlokat, végrehajtható programokat, stb. tartalmaznak, nevezhetjük **bináris** fájloknak: nem sorokban szervezettek, és nem tudjuk megnyitni egy normál szövegszerkesztővel. Pythonban hasonlóan könnyen kezelhetjük a bináris fájlokat, de amikor olvasunk a fájlból, nem sztringeket hanem bájtokat használunk. Itt egy bináris fájl tartalmát átmásoljuk egy másik fájlba:

```
1 f = open("szoveg.zip", "rb")
2 g = open("masolat.zip", "wb")
3
4 while True:
5     buf = f.read(1024)
6     if len(buf) == 0:
7         break
8     g.write(buf)
9
10 f.close()
11 g.close()
```

A fenti kódrészletben láthatunk néhány új dolgot. Az 1. és 2. sorban a megnyitási módhoz hozzáadtunk egy "b" betűt, mely jelzi a Python számára, hogy bináris fájlokról van szó. Az 5. sorban látható, hogy a `read` kaphat egy argumentumot, amely megadja, hogy hány bájtot olvasson a fájlból. Itt mindegyik ciklus lépésben legfeljebb 1024 bájtot olvassunk és írunk. Amikor egy üres buffert kapunk vissza, tudjuk, hogy megszakíthatjuk a ciklust, és bezárhatjuk mindkét fájlt.

Ha a 6. sorba teszünk egy töréspontot, és Debug módban futtatjuk a szkriptet (vagy kiírjuk a `type(buf)` hívás eredményét), akkor látni fogjuk, hogy a `buf` típusa a `bájt`. Ebben a könyvben a `byte` objektumokat nem részletezzük.

13.7. Egy példa

Sok hasznos sor-feldolgozó program sorról sorra olvassa be a szöveges állományokat, és kisebb feldolgozást végez, miközben az eredményt soronként egy kimeneti fájlba írja. A kimeneti fájl sorai sorszámozhatók, vagy minden 60-ik sor után üres sort szűrhatunk be, a papírra történő nyomtatás megkönnyítése végett, vagy csak néhány speciális oszlopát írjuk ki a forrás fájl soraiból vagy csak olyan sorokat íratunk ki, melyek egy bizonyos részsstringet tartalmaznak. Ezt a típusú programot **szűrőnek** nevezzük.

Itt láthatunk egy szűrő programot, amely egy fájlt átmásol egy másikba, és kihagyja azokat a sorokat, melyek `#` jellel kezdődnek.

```
1 def szuro(regifajl, ujfajl):
2     bemenet = open(regifajl, "r")
3     kimenet = open(ujfajl, "w")
4     while True:
5         szoveg = bemenet.readline()
6         if len(szoveg) == 0:
7             break
8         if szoveg[0] == "#":
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
9         continue
10
11         # Szűrj be további sorfeldolgozási módszereket!
12         kimenet.write(szoveg)
13
14     bemenet.close()
15     kimenet.close()
```

A 9. sorban szereplő `continue` utasítás kihagyja a fennmaradó sorokat a ciklus aktuális iterációjában, de a ciklus folytatódik a következő iterációs lépéssel. Ez a stílus kissé bonyolultnak tűnik, de gyakran hasznos azt mondani: „vegyük ki azokat a sorokat, amelyekkel nem foglalkozunk azért, hogy áttekinthetőbb legyen a ciklus fő része, amit a 11. sor környékére írhatunk.”

Így, ha a `szoveg` egy üres sztring, a ciklus leáll. Ha a `szoveg` első karaktere a `#`, a végrehajtás során a ciklus elejére ugrik, és újra kezdi a feldolgozást a következő sortól. Csak akkor dolgozzuk fel a 11. sort, ha mindkét feltétel hamis. Ebben a példában azt a sort az új fájlba írjuk.

Tekintsünk egy újabb esetet: feltételezzük, hogy az eredeti fájl üres sorokat tartalmazott. A fenti 6. sorban, amikor a program megtalálja az első üres sort, azonnal befejezi a végrehajtást? Nem! Emlékezzünk vissza, hogy a `readline` mindig tartalmazza az újsor karaktert a visszatérő karakterláncban. Amikor megpróbáljuk a fájl végét beolvasni, akkor egy 0 hosszúságú sztringet kapunk vissza.

13.8. Könyvtárak

A nem-felejtő tároló eszközökön lévő fájlokat az ismert szabályok alapján **fájlrendszerbe** csoportosítják. A fájlrendszerek fájlokból és **könyvtárakból** állnak, amelyek tárolók, mind a fájlok és egyéb könyvtárak számára.

Amikor létrehozunk egy új fájlt írásra, az új fájl az aktuális könyvtárba kerül (bárhol is voltunk, amikor a programot futtattuk). Hasonlóképpen, amikor megnyitunk egy fájlt az olvasásra, a Python az aktuális könyvtárban keresi.

Ha valahol máshol szeretnénk megnyitni egy fájlt, meg kell adnunk a fájl **elérési útját**, amely a könyvtár neve (vagy mappa), ahol a fájl található:

```
1 fajl = open("/usr/share/dict/szavak", "r")
2 lista = fajl.readlines()
3 print(lista[:5])
```

```
['\n', 'egy\n', 'kettő\n', 'három\n', 'négy\n']
```

A fenti (Unix) példa megnyitja a `dict` nevű könyvtárban található `szavak` nevű fájlt, ez a `share` és ezen felül az `usr` nevű könyvtárban van, a legfelsőbb szint pedig a gyökérkönyvtár, amit perjellel / jelölünk. Ez után minden egyes sort a `readline`-nal olvasunk egy listába, és kiírjuk a lista első 5 elemét.

A Windows elérési útja lehet a `"c:/temp/szavak.txt"` vagy `"c:\\temp\\szavak.txt"`. Mivel a backslash-t használjuk, hogy elkerüljük az új sort és a tabulátort, ezért két backslash karaktert kell írni egy sztring literálba, hogy egyet kapjunk. Tehát a két sztring hossza ugyanaz.

Nem használhatjuk a `/` vagy `\\` karaktereket a fájlnev részeként, ezek a könyvtár és fájlnevek **határolóiként** vannak fenntartva.

A `/usr/share/dict/szavak` fájlnak léteznie kell a Unix-alapú rendszerekben, és egy olyan szólistát tartalmaz, ahol a szavak alfabetikus sorrendben vannak.

13.9. Mi a helyzet az internetről való letöltéssel?

A Python könyvtárak helyenként elég rendetlenek. De itt van egy nagyon egyszerű példa, amely a webes URL-cím tartalmát egy helyi fájlba másolja.

```
1 import urllib.request
2
3 url = "http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/thinkcspy3.pdf"
4 cel_fajlnev = "thinkcspy3.pdf"
5
6 urllib.request.urlretrieve(url, cel_fajlnev)
```

Az `urlretrieve` függvény – csak egy hívás – felhasználható bármilyen tartalom internetről való letöltésére.

Néhány dolgot ellenőriznünk kell, mielőtt ezt kipróbálnánk:

- A forrásnak, amelyet megpróbálunk letölteni, léteznie kell! Ellenőrizd a böngészővel!
- Írási jogosultságra van szükségünk a cél fájl megírásához, és a fájlt az „aktuális könyvtárban” fogjuk létrehozni – például ugyanabban a könyvtárban, amelyben a Python program mentésre került.
- Ha hitelesítést igénylő proxy szerveret használunk (mint néhány hallgató), szükség lehet néhány speciális proxy beállításra. Használjunk helyi forrást ezen példa bemutatására!

Itt egy kicsit eltérőbb példa. Ahelyett, hogy a webes erőforrást a helyi lemezünkre mentenénk, közvetlenül egy sztring-be olvassuk és visszatérünk vele:

```
1 import urllib.request
2
3 def weboldal(url):
4     """ Visszatér a weboldal tartalmával.
5     A tartalmat sztringgé alakítja, mielőtt visszatérne.
6     """
7     csatlakozo = urllib.request.urlopen(url)
8     adat = str(csatlakozo.readall())
9     csatlakozo.close()
10    return adat
11
12 szoveg = weboldal("http://www.gnu.org/software/make/manual/make.txt")
13 print(szoveg)
```

A távoli URL megnyitásának visszatérési értékét **csatlakozónak** nevezzük. Ez egy kezelő a kapcsolat végén, mely a programunk és a távoli webszerver között jön létre. Meghívhatjuk az olvasás, írás és bezárás metódusokat a csatlakozó objektumra, ugyanúgy ahogy egy fájlkezelővel tennénk.

13.10. Szójegyzék

csatlakozó (socket) Egy olyan kapcsolat, mely lehetővé teszi, hogy információkat olvasson és írjon egy másik számítógépről.

fájl (file) Megnevezett entitás, általában egy merevlemezen, hajlékonylemezen vagy CD-ROM-on van tárolva, amely egy karaktersorozatot tartalmaz.

fájl rendszer (file system) A fájlok és az általuk tárolt adatok elnevezése, elérése és rendszerezése.

felejtő memória (volatile memory) Olyan memória, amely elektromos áramot igényel az állapotának fenntartásához. A számítógép fő memóriája a RAM, felejtő memória. A RAM-ban tárolt adatok elvesznek, amikor kikapcsoljuk a számítógépet.

határoló (delimiter) Egy vagy több karakter szekvenciája, a szöveg különálló részei közötti határ meghatározásához.

kezelő (handle) Olyan objektum a programunkban, amely egy mögöttes erőforráshoz (például egy fájlhoz) kapcsolódik. A fájlkezelő lehetővé teszi programunk számára a lemezünkön található aktuális fájl manipulálását / beolvasását / kiírását / bezárását.

könyvtár (directory) Fájlgyűjtemény, melyet más néven mappának neveznek. A könyvtárak fájlokat és egyéb könyvtárakat tartalmazhatnak, amelyekre úgy utalunk, mint a könyvtár *alkönyvtárai*.

mód (mode) Megkülönböztetett működési mód egy számítógépes programon belül. A Python fájlok négyféleképpen nyithatók meg: olvasásra („r”), írásra („w”), hozzáfűzésre („a”), olvasásra és írásra („+”).

nem felejtő memória (non-volatile memory) Olyan memória, amely fenn tudja tartani az állapotát áram nélkül is. Néhány példa nem felejtő memóriára: a merevlemez, a flash meghajtók, újraírható CD-k, stb.

szöveges fájl (text file) Olyan fájl, amely nyomtatható karaktereket tartalmaz, és a sorok az új sor karakterrel vannak elválasztva.

útvonal (path) A könyvtárnevek sorozata, mely meghatározza a fájl pontos helyét.

13.11. Feladatok

1. Írj egy olyan programot, amely beolvas egy fájlt, és a sorait fordított sorrendben írja be egy új fájlba (például az első sor a régi fájlban az utolsó, és az utolsó sor a régi fájlban az első).
2. Írj egy olyan programot, amely beolvas egy fájlt, és csak azokat a sorait írja ki, melyek tartalmazzák az `info` részszerínget.
3. Írj egy olyan programot, amely beolvas egy szöveges fájlt, és egy kimeneti fájlt hoz létre, amely az eredeti fájl másolata, kivéve, hogy minden egyes sor első öt oszlopa tartalmaz egy négyjegyű sorszámot, amelyet egy szóköz követ. A kimeneti fájl sorszámozását 1-től kezd. Győződj meg arról, hogy minden egyes sorszám ugyanolyan széles a kimeneti fájlban. Használd az egyik Python programot, teszt adatként ehhez a feladathoz: a kimenet a Python program kiírt és sorszámozott listája kell legyen.
4. Írj egy olyan programot, amely megszünteti az előző gyakorlat számozását: ennek egy beszámozott sorokat tartalmazó fájlt kellene beolvasnia, és egy másik fájlt előállítani a sorszámozás nélkül.

14. fejezet

Lista algoritmusok

Ez a fejezet kissé különbözik az előzőktől: ahelyett, hogy további új Python szintaktikát és funkciókat vezetnénk be, a programfejlesztés folyamatára és listákkal foglalkozó algoritmusokra fókuszálunk.

Úgy, mint a könyv összes részében, azt várjuk el, hogy kipróbáld a kódot Python környezetben, játssz, kísérletezz és dolgozz velünk együtt.

Ebben a fejezetben az *Alice Csodaországban* című könyvvel és a *szókincs* nevű fájlal dolgozunk. A böngésző segítségével mentsd le ezeket a fájlokat a megadott linkekről.

14.1. Tesztvezérelt fejlesztés

A korábbi *Produktív függvények* című fejezetben bevezettük az *inkrementális fejlesztés* ötletét, ahol kisebb kódrészeket adtunk hozzá programunkhoz, hogy lassan felépítsük az egészet, ezáltal könnyebben és korábban megtalálhatjuk a problémákat. Később ugyanabban a fejezetben bevezettük az egységtesztet, és megadtunk egy kódot a teszt keretrendszerünknek azért, hogy kód formában kaphassuk meg a függvényekre megírt teszteket.

A **tesztvezérelt fejlesztés (TDD)** olyan szoftverfejlesztési gyakorlat, amelyik egy lépéssel tovább viszi ezeket a gyakorlatokat. Az alapötlet az, hogy *először* az automatizált teszteket kell megírni. Ezt a technikát nevezzük *tesztvezéreltnek*, mivel – ha hiszünk a szélsőségeknek – nem-tesztelő kódot csak akkor kellene írni, amikor nem sikerül a tesztet végrehajtani.

Maradva a tanulási munkamódszerünkénél, kis növekvő lépésekben fogunk haladni, de most ezeket a lépéseket egyre kifinomultabban, egységtesztekkel támasztjuk alá, többet várva el a kódunktól minden egyes szinten.

Felhívjuk a figyelmet néhány alap algoritmusra, amelyek feldolgozzák a listákat, de ahogy haladunk előre ebben a fejezetben, megpróbáljuk ezt a TDD szellemiségében tenni.

14.2. A teljes keresés algoritmus

Szeretnénk tudni azt az indexet, ahol egy adott elem szerepel a listában. Pontosabban, visszatérünk az elem indexével amennyiben megtaláltuk, vagy -1 -el, ha az elem nem szerepel a listában. Kezdjük néhány teszttel:

```
1 barátok = ["Péter", "Zoltán", "János", "Kata", "Zita", "Sándor", "Panni"]
2 teszt(teljes_kereses(barátok, "Zoltán") == 1)
3 teszt(teljes_kereses(barátok, "Péter") == 0)
4 teszt(teljes_kereses(barátok, "Panni") == 6)
5 teszt(teljes_kereses(barátok, "Béla") == -1)
```

Mivel motivál bennünket az a tény, hogy a tesztjeink még nem futnak, most írjuk meg a függvényt:

```
1 def teljes_kereses(xs, ertekek):
2     """ Keresse meg és térjen vissza az érték indexével az xs sorozatban. """
3     for (i, v) in enumerate(xs):
4         if v == ertekek:
5             return i
6     return -1
```

Itt van néhány pont, amit megtanulhatunk: Hasonló az algoritmus, mint amit a 8.10 fejezetben láttunk, amikor egy karaktert kerestünk a sztringben. Ott a `while` ciklust használtuk, itt pedig a `for` ciklust az `enumerate`-tel együtt, hogy kiválasszuk az `(i, v)` párokat minden iterációnál. Vannak további változatok – például, használhattuk volna a `range`-t, akkor a ciklus csak az indexeket használta volna, vagy használhattuk volna a `None`-t is visszatérési értéként, ha nem találtuk meg a keresett elemet a listában. Az alapvető hasonlóság mindezen változatok között az, hogy a listában szereplő minden elemet egymás után, az elejétől a vége felé haladva teszteljük, a korábban bemutatott rövidzár elvet, az általunk *Heuréka bejárásnak* is nevezett mintát használva, és visszatérünk a függvényből, amint megtalálta az elemet, amelyet kerestünk.

Azt a keresést, mely a sorozat első elemétől halad a végéig, **teljes keresésnek** nevezzük. Minden alkalommal ellenőrizzük, hogy `v == ertekek`, ezt **összehasonlításnak** nevezzük. Szeretnénk megszámolni az összehasonlítások számát, hogy megmérjük az algoritmusunk hatékonyságát, és ez egy jó visszajelzés arra, hogy milyen hosszú lesz az algoritmus végrehajtásának időtartama.

A teljes keresés jellemzője, hogy hány összehasonlítás szükséges, hogy megtaláljuk a keresett elemet, ez függ a lista hosszától. Tehát ha a lista 10-szer nagyobb, mi 10-szer hosszabb időt várunk a keresés során. Figyeljük meg, ha a keresési érték nincs a listában, ellenőrizni kell a lista összes elemét, mielőtt negatív értékkel térnénk vissza. Ezért ebben az esetben N összehasonlításra van szükség, ahol N a lista hossza. Ha azonban egy olyan értéket keresünk, amely benne van a listában, szerencsések lehetünk, ha azonnal megtaláljuk a 0-s pozíción, de az is lehet, hogy tovább kell keressük, előfordulhat, hogy csak az utolsó helyen lesz. Átlagosan, ha a keresett érték a listában van, el kell menünk a lista feléig, vagyis $N/2$ összehasonlítás lesz.

Azt mondjuk, hogy ezen keresési algoritmusnak **lineáris a teljesítménye** (a lineáris azt jelenti *egyenes vonalú*), ha a különböző méretű listák (N) átlagkeresési idejét mérjük, majd az N függvényében a keresési időt ábrázoljuk egy grafikonon, többé-kevésbé egy egyenes vonalat kapunk.

Az ilyen elemzés értelmetlen kis méretű listák esetén – a számítógép elég gyors ahhoz, hogy ne terhelje meg, ha a lista csak néhány elemből áll. Általában érdekel bennünket az algoritmusok **skalázhatósága**, hogyan működnek, ha nagyobb problémát kell megoldaniuk. Vajon ezt a keresési algoritmust ésszerű lenne használni, ha már millió vagy tíz millió elemet (talán a helyi könyvtár könyveit) találunk a listánkban? Mi történik az igazán nagy adathalmazokkal, például hogyan keres ilyen briliánsan a Google?

14.3. Egy valós probléma

Ahogy a gyerekek megtanulnak olvasni, elvárjuk, hogy a szókincsük növekedjen. Tehát egy 14 éves gyerek várhatóan több szót ismer, mint egy 8 éves. Amikor egy évfolyam számára könyvet javasolunk, egy fontos kérdés merülhet fel: *a könyv mely szavai nem ismertek ezen a szinten?*

Tegyük fel, hogy a programunk be tudja olvasni egy szókincs szavait és a könyv szövegét, melyet szétválogathatunk szavakra. Írj néhány tesztet arra, hogy mi legyen a következő lépés. A tesztadatok általában nagyon kicsik, még akkor is, ha szándékunkban áll a programunkat egyre nagyobb esetekben használni:

```
1 szokincs = ["alma", "esett", "fűre", "fáról", "alá", "a", "fel"]
2 konyv_szavai = "az alma a fáról le esett a fűre".split()
3 teszt(ismeretlen_szavak_keresese(szokincs, konyv_szavai) == ["az", "le"])
4 teszt(ismeretlen_szavak_keresese([], konyv_szavai) == konyv_szavai)
5 teszt(ismeretlen_szavak_keresese(szokincs, ["alma", "alá", "esett"]) == [])
```


Figyeld meg, hogy egy kicsit lusták voltunk és a `split`-et használtuk, hogy egy szavakból álló listát hozzunk létre – ez könnyebb, mint a lista begépelése és nagyon kényelmes, hogyha a bemenetként megadott mondatot szavak listájává szeretnénk alakítani.

Most implementálni kell a függvényt, amelyre megírtuk a teszteket, és használjuk a saját teljes keresésünket. Az alapvető stratégia az, hogy végig megyünk a könyv minden egyes szaván és megkeressük, hogy benne van-e a szókincsben, ha nincs, akkor egy új eredmény listába mentjük azokat, és ezzel az új listával tér vissza a függvényünk.

```
1 def ismeretlen_szavak_keresese(szokincs, szavak):
2     """ Visszatérünk a könyv azon szavainak listájával, amelyek nincsenek
    ↪ benne a szókincsben. """
3     eredmeny = []
4     for w in szavak:
5         if (linearis_kereses(szokincs, w) < 0):
6             eredmeny.append(w)
7     return eredmeny
```

Boldogan jelentjük, hogy a tesztjeink mind sikeresek.

Most nézzük meg a skálázhatóságát. A szövegfájlban egy valósabb szókincs van, melyet a fejezet elején töltöttünk le, tehát olvassuk be a fájlt (mint egy egyszerű sztringet), és daraboljuk szét szavak listájává. A kényelem érdekében létrehozunk egy függvényt, hogy ezt végrehajtsa és tesztelje egy elérhető fájlon:

```
1 def szavak_betoltese_fajlbol(fajlnev):
2     """ Szavak olvasása a megadott fájlból, visszatér a szavak listájával. """
    ↪ ""
3     f = open(fajlnev, "r")
4     tartalom = f.read()
5     f.close()
6     szavak = tartalom.split()
7     return szavak
8
9 nagyobb_szokincs = szavak_betoltese_fajlbol("vocab.txt")
10 print("A szókincsben {0} szó található, kezdve: \n {1} "
11       .format(len(nagyobb_szokincs), nagyobb_szokincs[:6]))
```

A Python válasza:

```
1 A szókincsben 19455 szó található, kezdve:
2 ['a', 'aback', 'abacus', 'abandon', 'abandoned', 'abandonment']
```

Így most van egy értelmesebb méretű szókincsünk. Töltsük be a könyvet, újra azt a fájlt használjuk, amit letöltöttünk a fejezet elején. A könyv betöltése olyan, mint a fájl betöltése, de egy kis extra fekete mágia fogunk alkalmazni. A könyvek tele vannak írásjelekkel, kis- és nagybetűk keverékével. Meg kell tisztítanunk a könyv tartalmát. Ez magában foglalja az írásjelek eltávolítását, és minden karakter ugyanolyan típusúvá való konvertálását (kisbetűssé, mivel a szókincsünk kisbetűs). Tehát egy kifinomultabb módon szeretnénk a szöveget szavakra konvertálni.

```
1 teszt(szovegbol_szavak("Az én nevem Alice!") == ["az", "én", "nevem", "alice
    ↪ "])
2 teszt(szovegbol_szavak('"Nem, Én soha!", mondta Alice.') ==
3       ["nem", "én", "soha", "mondta", "alice"])
```

Van egy hathatós `translate` metódus, mely elérhető a sztringeknél. Az ötlet az, hogy állítsuk be a kívánt helyettesítéseket – minden karakterhez megfelelő helyettesítési karaktert adhatunk meg. A `translate` metódust alkalmazzuk a cserénél a teljes sztringen. Így a következőket kapjuk:

```
1 def szovegbol_szavak(szoveg):
2     """ Visszaadja a szavak listáját, eltávolítva az összes írásjelt és
    ↪ minden szót kisbetűssé alakít. """
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3
4     helyettesites = szoveg.maketrans(
5         # Ha bármelyikükkel találkozol
6         "AÁBCDEÉFGHIÍJKLMNOÓÖPQRSTUÚÚVWXYZ0123456789!\\"#$%&()*+,-./:;<=>?@[ ]^_`
        ↳{|}~'\\"",
7         # Cseréld őket ezekre
8         "aábcdeéfgghiíjklmnoóöőpqrstuúúüvwxyz
        ↳")
9
10    # Most alakítsd át a szöveget
11    tisztított_szoveg = szoveg.translate(helyettesites)
12    szavak = tisztított_szoveg.split()
13    return szavak
```

Az átalakítás során az összes nagybetűs karaktert kisbetűssé konvertálta, az írásjel karaktereket és a számokat pedig szóközzé. Tehát a `split` meg fog szabadulni a szóközőktől, és szétválasztja a szöveget szavak listájává. A tesztek átmennek.

Most készen állunk, hogy olvassunk a könyvből:

```
1 def szavak_a_konyvbol(fajlnev):
2     """ Olvassa be a könyvet a megadott fájlból, és adja vissza a szavak_
        ↳listáját. """
3     f = open(fajlnev, "r")
4     tartalom = f.read()
5     f.close()
6     szavak = szovegbol_szavak(tartalom)
7     return szavak
8
9 konyv_szavai = szavak_a_konyvbol("alice_in_wonderland.txt")
10 print("A könyvben {0} szó található, az első 100 a következő:\n{1}".
11       format(len(konyv_szavai), konyv_szavai[:100]))
```

A Python kiírja a következőket: (minden szót egy sorban, a könyv miatt itt kicsit csaltunk a kiírásnál):

```
A könyvben 27336 szó található,
az első 100 a következő:
['alice', 's', 'adventures', 'in', 'wonderland', 'lewis', 'carroll',
'chapter', 'i', 'down', 'the', 'rabbit', 'hole', 'alice', 'was',
'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by',
'her', 'sister', 'on', 'the', 'bank', 'and', 'of', 'having',
'nothing', 'to', 'do', 'once', 'or', 'twice', 'she', 'had',
'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading',
'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in',
'it', 'and', 'what', 'is', 'the', 'use', 'of', 'a', 'book',
'thought', 'alice', 'without', 'pictures', 'or', 'conversation',
'so', 'she', 'was', 'considering', 'in', 'her', 'own', 'mind',
'as', 'well', 'as', 'she', 'could', 'for', 'the', 'hot', 'day',
'made', 'her', 'feel', 'very', 'sleepy', 'and', 'stupid',
'whether', 'the', 'pleasure', 'of', 'making', 'a']
```

Nos, most már készen vagyunk az összes alprogrammal. Lássuk, milyen szavak vannak a könyvben, melyek nem szerepelnek a szókinszben:

```
1 hianyzo_szavak = ismeretlen_szavak_keresese(nagyobb_szokincs, konyv_szavai)
2 print(hianyzo_szavak)
```

Jelentős időt várunk, körülbelül egy percet, mielőtt a Python befejezi a munkát, és kiírja a könyv 3396 szavát, amelyek nem szerepelnek a szókincsben. Hmmm... Ez gyakorlatilag nem skálázható. Ha hússzor nagyobb lenne a szókincs (például gyakran találhatsz iskolai szótárt 300 000 szóval), és hosszabb könyvet, akkor ez az algoritmus nagyon lassú lesz. Készítsünk néhány időmérést, miközben elgondolkodunk arról, hogyan fejleszthetnénk a következő részben.

```
1 import time
2
3 t0 = time.clock()
4 hanyzo_szavak = ismeretlen_szavak_keresese(nagyobb_szokincs, konyv_szavai)
5 t1 = time.clock()
6 print("{0} ismeretlen szó van.".format(len(hanyzo_szavak)))
7 print("Ez {0:.4f} másodpercet vett igénybe.".format(t1-t0))
```

Megkaptuk az eredményeket és az időintervallumot, melyekre később majd visszatérünk:

```
3396 ismeretlen szó van.
Ez 49,8014 másodpercet vett igénybe.
```

14.4. Bináris keresés

Ha arra gondolsz, amit az előbb végrehajtottunk, az nem így működik a valós életben. Hogyha adott egy szókincs és azt kérdezik, hogy valamilyen szó benne van-e, valószínűleg a közepéről indulnánk. Ezt megteheted, mivel a szókincs szavai rendezettek – így megvizsgálhatsz egy szót a közepén, és rögtön el tudod dönteni, hogy a keresett szó előtte van (vagy talán utána) amivel összehasonlítottad. Ha ezt az elvet ismét alkalmazod, akkor egy sokkal jobb keresési algoritmust kapsz egy olyan listára, ahol az elemek már rendezettek. (Ne feledd, amennyiben az elemek nem rendezettek, nincs más választás, mint, hogy mindegyiket sorban megvizsgáljuk. De ha tudjuk, hogy az elemek rendezettek, javíthatjuk a keresési technikánkat).

Kezdjünk néhány teszttel. Ne feledd, a lista rendezett kell, hogy legyen:

```
xs = [2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, 43, 47, 53]
teszt(binaris_kereses(xs, 20) == -1)
teszt(binaris_kereses(xs, 99) == -1)
teszt(binaris_kereses(xs, 1) == -1)
for (i, v) in enumerate(xs):
    teszt(binaris_kereses(xs, v) == i)
```

Ezúttal még a tesztesetek is érdekesek: figyelj meg, hogy a listán nem szereplő elemekkel és a határ feltételek vizsgálatával kezdünk, tehát a lista középső elemével, egy kisebb elemmel, mely a lista összes eleménél kisebb és egy nagyobb, mint a lista legnagyobb eleme. Ezután egy ciklus segítségével, mely a lista minden elemét keresési érték-ként használja, azt láthatjuk, hogy a bináris keresés a listában szereplő elemek megfelelő indexét adja vissza.

Hasznos lehet, ha arra gondolunk, hogy a listán belül meg kell keresnünk a *vizsgált területet* (ROI – region-of-interest). Ez a ROI a lista azon része, amelyben lehetséges, hogy megtalálható a keresett elem. Az algoritmusunk azzal a ROI-val kezdődik, amely a ciklus összes elemére van beállítva. Az első összehasonlítás a ROI közepére vonatkozik, három eset lehetséges: vagy megtaláltuk a keresett elemet, vagy megtudjuk, hogy a keresett elem ROI alsó felében vagy felső felében található, ezáltal felezhetjük a keresési intervallumot. És ezt ismételtelen folytatjuk, amíg megtaláljuk a keresett értéket, vagy addig, ameddig elfogynak az elemek a keresési intervallumban. A következőképpen kódolhatjuk:

```
1 def binaris_kereses(xs, ertek):
2     """ Keressük meg és térjünk vissza az érték indexével az xs sorozatban. """
3     ah = 0
4     fh = len(xs)
5     while True:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6     if ah == fh:      # Ha a vizsgált terület üres
7         return -1
8
9     # A következő összehasonlítás a ROI közepén kell legyen
10    kozep_index = (ah + fh) // 2
11
12    # Fogjuk középső indexen lévő elemet
13    kozep_elem = xs[kozep_index]
14
15    #print("ROI[{0}:{1}](méret={2}), próba='{3}', érték='{4}'")
16    #      .format(ah, fh, fh-ah, kozep_elem, ertekek))
17
18    # Hasonlítsuk össze az elemet az adott pozícióban lévővel
19
20    if kozep_elem == ertekek:
21        return kozep_index      # Megtaláltuk!
22    if kozep_elem < ertekek:
23        ah = kozep_index + 1    # Használjuk a felső ROI-t
24    else:
25        fh = kozep_index        # Használjuk az alsó ROI-t
```

A vizsgált területet két változó reprezentálja, az alsó határ `ah` és a felső határ `fh`. Fontos pontosan meghatározni, hogy az indexeknek milyen értékeik vannak. `ah` az első elem indexe a ROI-ban, és az `fh` a legutolsó utáni elem indexe a ROI-ban, tehát ez a szemantika hasonlít a Python szeletek szemantikájára: a vizsgált terület pontosan egy szelet `xs[ah:fh]`. (Az algoritmus sosem vesz ki tömb szeleteket!)

Ezzel a kóddal a tesztünk sikeres. Nagyszerű! Most, ha helyettesítünk egy hívást erre a keresési algoritmusra a `teljes_kereses` helyett az `ismeretlen_szavak_keresese` függvényben, javíthatjuk a teljesítményünket? Tegyük meg, és futtassuk újra ezt a tesztet:

```
1  t0 = time.clock()
2  hanyzo_szavak = ismeretlen_szavak_keresese(nagyobb_szokincs, konyv_szavai)
3  t1 = time.clock()
4  print("{0} ismeretlen szó van.".format(len(hanyzo_szavak)))
5  print("Ez {0:.4f} másodpercet vett igénybe.".format(t1-t0))
```

Milyen látványos a különbség! Több mint 200-szor gyorsabb!

```
3396 ismeretlen szó van.
Ez 0,2262 másodpercet vett igénybe.
```

Miért sokkal gyorsabb a bináris keresés, mint a teljes? Ha a 15. és 16. sorban elhelyezünk egy kiíró utasítást, nyomon követhetjük a keresés során végzett összehasonlításokat. Nos rajta, próbáljuk ki:

```
1  print(binaris_kereses(nagyobb_szokincs, "magic"))
2  ROI[0:19455](méret=19455), próba='knowing', érték='magic'
3  ROI[9728:19455](méret=9727), próba='resurgence', érték='magic'
4  ROI[9728:14591](méret=4863), próba='overslept', érték='magic'
5  ROI[9728:12159](méret=2431), próba='misreading', érték='magic'
6  ROI[9728:10943](méret=1215), próba='magnet', érték='magic'
7  ROI[9728:10335](méret=607), próba='lightning', érték='magic'
8  ROI[10032:10335](méret=303), próba='longitudinal', érték='magic'
9  ROI[10184:10335](méret=151), próba='lumber', érték='magic'
10 ROI[10260:10335](méret=75), próba='lyrical', érték='magic'
11 ROI[10298:10335](méret=37), próba='made', érték='magic'
12 ROI[10317:10335](méret=18), próba='magic', érték='magic'
13 10326
```

Itt láthatjuk, hogy a „magic” szó keresése során összesen 11 összehasonlításra volt szükség, mielőtt a 10326-os indexen megtalálta. A lényeg az, hogy minden egyes összehasonlítás során felezi a fennmaradó vizsgált területet. Ezzel szemben a teljes keresésnél 10327 összehasonlításra volt szükség, ameddig megtalálta a szót.

A *bináris* szó jelentése kettő. A bináris keresés a nevét abból kapta, hogy minden alkalommal két részre bontjuk a listát, és elhagyjuk a vizsgált terület felét.

Az algoritmus szépsége, hogy meg tudjuk duplázni a szókincs méretét, és csak egy újabb összehasonlításra van szükség. Egy további duplázás esetén szintén csak egy újabb összehasonlítás kellene. Így a szókincsünk egyre nagyobb, és az algoritmus teljesítménye egyre hatásosabbá válik.

Megadhatunk egy képletet erre? Ha a listánk mérete N , akkor mennyi lesz a legnagyobb számú összehasonlítások száma, amely szükséges? Matematikailag egy kicsivel egyszerűbb, ha körül járjuk a kérdést: mekkora nagy N elemű listát tudnánk kezelni, ha csak k számú összehasonlítást végezhetnénk?

1 összehasonlítással csak 1 elemű listában kereshetünk. Két összehasonlítással már három elemű listával is meg tudunk birkózni – (vizsgáljuk az középső elemet az első összehasonlítással, majd vagy a bal vagy a jobb oldali listát a második összehasonlítással.) Eggyel több összehasonlítással 7 elemmel tudunk megbirkózni (vizsgáljuk a középső elemet, majd a hármas méretű részlistát). Négy összehasonlítással 15 elemű listában kereshetnénk, 5 összehasonlítással pedig 31 elemű listában. Tehát az általánosított összefüggést a következő képlet adja:

$$N = 2^k - 1$$

ahol k az összehasonlítások száma, és N a lista maximális mérete, amelyben keresünk. Ez a függvény *exponenciálisan* nő k -ban (mivel k jelenik meg a kitevőben). Ha meg akarjuk változtatni a képletet, és szeretnénk megadni a k -t N elem esetén, akkor az 1-es konstanszt átvisszük az egyenlőség másik oldalára, és minkét oldalnak vesszük a 2-es alapú logaritmusát. (A logaritmus az exponenciális inverze.) Tehát a k -ra vonatkozó képlet N függvényben a következő:

$$k = \lceil \log_2(N + 1) \rceil$$

A fent-szögletes-zárójelt úgy nevezhetjük, mint *felső egészrész* zárójel, azt jelenti, hogy a számot felfele kerekítjük a következő egész számra.

Próbáljuk ki ezt egy számológépen, vagy Pythonban, amely az összes számológép őse: tegyük fel, hogy 1000 elem között keresünk, mekkora lesz a legnagyobb összehasonlítási szám, amely a keresés során szükséges. (A képletben kissé bosszantó a +1, tehát ne felejtjük el hozzáadni...):

```
1 from math import log
2 print(log(1000 + 1, 2))
```

```
9.967226258835993
```

Tehát maximum 9,96 összehasonlításra van szükség, 1000 elem esetén, nem azt kaptuk, amit akartunk. Elfeledkeztünk a felső egészre való kerekítésről. A `math` modul `ceil` függvénye pontosan ezt teszi. Tehát még pontosabban:

```
1 from math import log, ceil
2 print(ceil(log(1000 + 1, 2)))
3 print(ceil(log(1000000 + 1, 2)))
4 print(ceil(log(1000000000 + 1, 2)))
```

```
10
20
30
```

1000 elem esetén a kereséshez 10 összehasonlításra van szükség. (Technikailag 10 összehasonlítással 1023 elem között kereshetünk, de egyszerűbb megjegyezni, hogy „1000 elemhez 10 összehasonlításra, 1 millió elemhez 20 és 1 milliárd elem esetén 30 összehasonlításra van szükség”).

Ritkán találkozhatunk olyan algoritmusokkal, amelyek nagyméretű adathalmazok esetén hasonlóan nagyszerűek, mint a bináris keresés!

14.5. A szomszédos duplikátumok eltávolítása

Gyakran szeretnénk, hogy csak egyedi elemekből álló listánk legyen, például létrehozunk egy új listát, amelyben minden elem csak egyszer szerepel, tehát különbözőek. Figyeljük meg azt az esetet, hogy mi történik akkor, amikor olyan szavakat keressünk az Alice Csodaországban szövegében, amelyek nincsenek a szókincsben. A kódunk 3396 szót eredményezett, viszont ezen listában szerepelnek duplikátumok. Valójában az „alice” szó 389 alkalommal szerepelt a könyvben és nincs a szókincsben! Hogyan tudnánk eltávolítani a duplikátumokat?

Egy jó megközelítés, ha először rendezzük a listát, majd eltávolítjuk az összes duplikátumot. Kezdjük a szomszédos duplikátumok eltávolításával:

```
1  teszt(szomszedos_dupl_eltovolit([1,2,3,3,3,3,5,6,9,9]) == [1,2,3,5,6,9])
2  teszt(szomszedos_dupl_eltovolit([]) == [])
3  teszt(szomszedos_dupl_eltovolit(["egy", "kis", "kis", "kölyök", "kutya"]) ==
4  ["egy", "kis", "kölyök", "kutya"])
```

Az algoritmus könnyű és hatékony. Megjegyezzük a legutóbbi elemet, melyet beszűrtünk az eredménybe és nem szűrjük be ismét:

```
1  def szomszedos_dupl_eltovolit(xs):
2      """ Visszatér egy új listával, amelyben a szomszédos
3          duplikátumok el vannak távolítva az xs listából.
4      """
5      eredmeny = []
6      aktualis_elem = None
7      for e in xs:
8          if e != aktualis_elem:
9              eredmeny.append(e)
10             aktualis_elem = e
11
12     return eredmeny
```

Az algoritmus lineáris – minden elem az `xs` lista a ciklus egy végrehajtását eredményezi, és nincs más egymásba ágyazott ciklus. Így az `xs` elemszámának megduplázása során a függvény kétszer olyan hosszú lenne: a lista mérete és a futási idő közötti kapcsolat szintén lineáris, egyenes vonalként ábrázolható.

Most térjünk vissza az *Alice Csodaországban* című példánk elemzéséhez. Mielőtt ellenőriznénk a könyvben szereplő szavakat, rendezni kell, majd eltávolítani a duplikátumokat. Tehát az új kódunk a következőképpen néz ki:

```
1  osszes_szo = szavak_a_konyvbol("alice_in_wonderland.txt")
2  osszes_szo.sort()
3  konyv_szavai = szomszedos_dupl_eltovolit(osszes_szo)
4  print("A könyvben {0} szó van. Csak {1} egyedi.".
5        format(len(osszes_szo), len(konyv_szavai)))
6  print("Az első 100 szó\n{0}".
7        format(konyv_szavai[:100]))
```

Varázslatszerűen a következő kimenetet kapjuk:

```
A könyvben 27336 szó van. Csak 2569 egyedi.
Az első 100 szó
['a', 'abide', 'able', 'about', 'above', 'absence', 'absurd',
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
'acceptance', 'accident', 'accidentally', 'account', 'accounting',  
'accounts', 'accusation', 'accustomed', 'ache', 'across', 'act',  
'actually', 'ada', 'added', 'adding', 'addressed', 'addressing',  
'adjourn', 'adoption', 'advance', 'advantage', 'adventures',  
'advice', 'advisable', 'advise', 'affair', 'affectionately',  
'afford', 'afore', 'afraid', 'after', 'afterwards', 'again',  
'against', 'age', 'ago', 'agony', 'agree', 'ah', 'ahem', 'air',  
'airs', 'alarm', 'alarmed', 'alas', 'alice', 'alive', 'all',  
'allow', 'almost', 'alone', 'along', 'aloud', 'already', 'also',  
'altered', 'alternately', 'altogether', 'always', 'am', 'ambition',  
'among', 'an', 'ancient', 'and', 'anger', 'angrily', 'angry',  
'animal', 'animals', 'ann', 'annoy', 'annoyed', 'another',  
'answer', 'answered', 'answers', 'antipathies', 'anxious',  
'anxiously', 'any', 'anything', 'anywhere', 'appealed', 'appear',  
'appearance', 'appeared', 'appearing', 'applause', 'apple', 'apples', 'arch']
```

Lewis Carroll egy klasszikus irodalmi alkotást csupán 2569 különböző szóval alkotott meg.

14.6. Sorbarendeztet listák összefésülése

Feltételezzük, hogy két rendezett listánk van. Írjunk egy algoritmust, mely összefésüli őket egyetlen rendezett listába.

Egy egyszerű, de nem hatékony algoritmus lehet, hogy egyszerűen összevonja a két listát, majd rendezi az eredményt:

```
1 ujlista = (xs + ys)  
2 ujlista.sort()
```

De a fenti algoritmus nem használja ki listák azon tulajdonságát, hogy rendezve vannak, és egy nagyon nagy lista esetén rossz a skálázhatósága és a teljesítménye.

Először készítsünk néhány tesztet:

```
1 xs = [1,3,5,7,9,11,13,15,17,19]  
2 ys = [4,8,12,16,20,24]  
3 zs = xs+ys  
4 zs.sort()  
5 teszt(osszefesul(xs, []) == xs)  
6 teszt(osszefesul([], ys) == ys)  
7 teszt(osszefesul([], []) == [])  
8 teszt(osszefesul(xs, ys) == zs)  
9 teszt(osszefesul([1,2,3], [3,4,5]) == [1,2,3,3,4,5])  
10 teszt(osszefesul(["cica", "egér", "kutya"], ["cica", "kakas", "medve"])) ==  
11 ["cica", "cica", "egér", "kakas", "kutya", "medve"])
```

Itt van az összefésülési algoritmusunk:

```
1 def osszefesul(xs, ys):  
2     """ Összefésüli a rendezett xs és ys listákat. Visszatér a rendezett_  
    ↪eredménnyel. """  
3     eredmeny = []  
4     xi = 0  
5     yi = 0  
6  
7     while True:  
8         if xi >= len(xs): # Ha az xs lista végére értünk
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
9         eredmeny.extend(ys[yi:]) # Még vannak elemek az ys listában
10     return eredmeny           # Készen vagyunk
11
12     if yi >= len(ys):          # Ugyanaz, csak fordítva
13         eredmeny.extend(xs[xi:])
14     return eredmeny
15
16     # Ha mindkét listában vannak még elemek, akkor a kisebbik elemet
17     ↪ másoljuk az eredmény listába
18     if xs[xi] <= ys[yi]:
19         eredmeny.append(xs[xi])
20         xi += 1
21     else:
22         eredmeny.append(ys[yi])
23         yi += 1
```

Az algoritmus a következőképpen működik: létrehozuk az eredmény listát, és két indexet használunk egyet-egyét a listák számára (3-5. sorok). Mindegyik ciklus lépésnél, attól függően, hogy melyik elem a kisebb, bemásoljuk az eredménylistába, és a lista indexét megnöveljük. Amint bármelyik index eléri a lista végét, az összes többi elemet a másik listáról bemásoljuk az eredménybe, és visszatérünk az új rendezett listával.

14.7. Alice Csodaországban, ismét!

Az algoritmus alapja a rendezett listák összefésülése, mely egy összetettebb számolási minta, és széleskörűen újra-hasznosítható. A minta lényege: „Járja be a listát, mindig a legkevesebb hátralévő elemet dolgozza fel, és vegye figyelembe a következő eseteket.”

- Mit tegyünk, ha nincs több eleme a listának?
- Mit tegyünk, ha a listák esetében a legkisebb elemek megegyeznek?
- Mit tegyünk, ha az első lista legkisebb eleme kisebb, mint a második lista legkisebb eleme?
- Mit tegyünk a fennmaradó esetekben?

Feltételezzük, hogy két rendezett listánk van. Használjuk az algoritmikus készségünket, és alkalmazzuk az összefűzés algoritmust mindegyik esetben.

- Csak azokkal az elemekkel térjünk vissza, melyek mindkét listában megtalálhatók.
- Csak azokkal az elemekkel térjünk vissza, melyek benne vannak az első listában, de a másodikban nem.
- Csak azokkal az elemekkel térjünk vissza, melyek benne vannak a második listában, de az elsőben nem.
- Csak azokkal az elemekkel térjünk vissza, melyek vagy az egyikben vagy a másik listában vannak benne.
- Csak azokkal az elemekkel térjünk vissza az első listából, amelyeket a második lista egy megegyező eleme nem távolít el. Ebben az esetben a második lista egyik eleme „kiüti” az első listában szereplő elemet. Például `kivonas([5, 7, 11, 11, 11, 12, 13], [7, 8, 11])` visszaadja következő listát: `[5, 11, 11, 12, 13]`.

Az előző alfejezetben rendeztük a könyv szavait és kihagytuk a duplikátumokat. A szókincs szintén rendezett. Tehát, fent a harmadik esetben – megkerestük a második listában szereplő elemeket, melyek nem voltak benne az elsőben, és másképp implementáltuk az `ismeretlen_szavak_keresese` algoritmust. Ahelyett, hogy minden szót a szótárban keresnénk (vagy teljes vagy bináris kereséssel), miért nem használjuk az összefésülést, és térünk vissza azokkal a szavakkal, melyek benne vannak a könyvben, de nincsenek a szókincsben.


```
1 def ismeretlen_szavak_osszefesulessel(szokincs, szavak):
2     """ Mind a szókincs és könyv szavai rendezettek kell, legyenek.
3         Visszatérünk egy új szólistával, mely szavak benne vannak a könyvben,
4         de nincsenek a szókincsben.
5     """
6
7     eredmeny = []
8     xi = 0
9     yi = 0
10
11     while True:
12         if xi >= len(szokincs):
13             eredmeny.extend(szavak[yi:])
14             return eredmeny
15
16         if yi >= len(szavak):
17             return eredmeny
18
19         if szokincs[xi] == szavak[yi]: # A szó benne van a szókincsben
20             yi += 1
21
22         elif szokincs[xi] < szavak[yi]: # Haladjon tovább
23             xi += 1
24
25         else: # Találtunk olyan szót, mely nincs a
26             ↪szókincsben
27             eredmeny.append(szavak[yi])
28             yi += 1
```

Most mindent összerakunk:

```
1 osszes_szo = szavak_a_konyvbol("alice_in_wonderland.txt")
2 t0 = time.clock()
3 osszes_szo.sort()
4 konyv_szavai = szomszedos_dupl_eltovolit(osszes_szo)
5 hanyzo_szavak = ismeretlen_szavak_osszefesulessel(nagyobb_szokincs, konyv_
6     ↪szavai)
7 t1 = time.clock()
8 print("{0} ismeretlen szó van.".format(len(hanyzo_szavak)))
9 print("Ez {0:.4f} másodpercet vett igénybe.".format(t1-t0))
```

Sokkal lenyűgözőbb teljesítményt kaptunk:

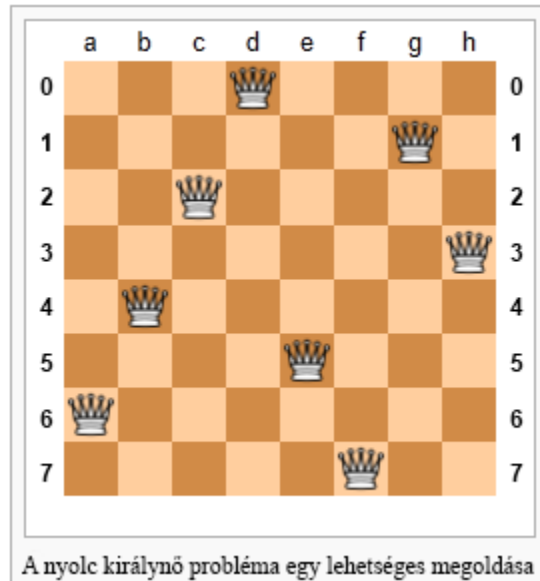
```
827 ismeretlen szó van.
Ez 0,0410 másodpercet vett igénybe.
```

Tekintsük át, hogyan dolgoztunk. Egy szóról szóra teljes kereséssel kezdtünk a szókincsben, mely körülbelül 50 másodpercig tartott. Majd implementáltunk egy okosabb bináris keresést, mely 0,22 másodpercig tartott, mely 200-szor gyorsabb volt. Aztán még valamit jobban csináltunk: rendeztük a könyv szavait, kihagytuk a duplikátumokat, majd használtuk az összefésülést, hogy megtaláljuk azokat a szavakat, melyek benne vannak a könyvben, de nincsenek a szókincsben. Ez ötször gyorsabb volt, mint a bináris keresési algoritmus. A fejezet végén az algoritmusunk 1000-szer gyorsabb, mint az első kísérletünk.

Ezt már egy nagyon jó napnak nevezhetjük!

14.8. Nyolc királynő probléma, első rész

Ahogy a Wikipedián olvashatjuk: „A nyolckirálynő-probléma egy sakktáblán, lényege a következő: hogyan, illetve hányféleképpen lehet 8 királynőt (vezért) úgy elhelyezni egy 8×8-as sakktáblán, hogy a sakk szabályai szerint ne üssék egymást. Ehhez a királynő lépési lehetőségeinek ismeretében az kell, hogy ne legyen két királynő azonos sorban, oszlopban vagy átlóban.”



Próbáld ki magad és keress néhány kézi megoldást.

Szeretnénk egy olyan programot írni, mely megoldást talál a fenti problémára. Valójában a probléma általánosan N királynő $N \times N$ -es sakktáblán való elhelyezéséről szól, így az általános esetre gondolunk, nem csak a 8×8 -as esetre. Talán találhatunk megoldásokat a 12 királynő egy 12×12 -es sakktáblán, vagy 20 királynő egy 20×20 -as sakktáblán való elhelyezésére.

Hogyan közelítsük meg ezt a komplex problémát? Egy jó kiindulópont, ha az *adatszerkezetre* gondolnánk – tehát pontosan, hogyan ábrázoljuk a sakktáblát, és hogyan a királynők állapotát a programunkban? Miután elkezdünk dolgozni a problémán, érdemes átgondolni, hogy hogyan fog kinézni a memória, elkezdhetünk gondolkodni a függvényekről és a részfeladatokról, amivel meg tudjuk oldani a problémát, például hogyan helyezhetünk el egy királynőt a sakktáblán, anélkül, hogy egy másik királynő üsse.

Egy megfelelő reprezentáció megtalálása, és aztán egy jó algoritmus létrehozása, mely az adatokon működik, nem mindig lehetséges egymástól függetlenül. Ha úgy gondolsz, hogy műveletekre van szükség, akkor érdemes változtatni vagy újra szervezni az adatokat, hogy megkönnyítsd a műveletek elvégzését.

Ezt a kapcsolatot az algoritmusok és adatszerkezetek között elegánsan egy könyv címmel *Algoritmusok + Adatszerkezetek = Programok* fejezhetjük ki, melyet az Informatika egyik úttörője, Niklaus Wirth írt, a Pascal fejlesztője.

Ötleteljünk, hogy hogyan lehet a sakktáblát és a királynőket reprezentálni a memóriában.

- Egy kétdimenziós mátrix (8 listából álló lista, mely 8 négyzetet tartalmaz) az egyik lehetőség. A sakktábla minden négyzetében szeretnénk tudni, hogy tartalmaz királynőt vagy sem – két lehetséges állapot lehet minden négyzet esetén – így tehát minden eleme a listának True vagy False vagy egyszerűbben 0 vagy 1.

A fenti megoldásra vonatkozó állapot és az adatok reprezentációja lehet:

```
1 d1 = [0, 0, 0, 1, 0, 0, 0, 0],
2     [0, 0, 0, 0, 0, 0, 1, 0],
3     [0, 0, 1, 0, 0, 0, 0, 0],
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4      [0, 0, 0, 0, 0, 0, 0, 1],
5      [0, 1, 0, 0, 0, 0, 0, 0],
6      [0, 0, 0, 0, 1, 0, 0, 0],
7      [1, 0, 0, 0, 0, 0, 0, 0],
8      [0, 0, 0, 0, 0, 1, 0, 0]]
```

Tudnunk kell egy üres sakktáblát is reprezentálni, és elképzelni, hogy az adatokkal kapcsolatosan milyen műveletekre és változtatásokra van szükség ahhoz, hogy egy királynőt el tudjunk helyezni a táblán.

- Egy másik ötlet lehet, hogy megtartjuk a királynők koordinátáit a listában. Az ábrán látható jelölés segítségével például a királynők állapotát a következőképpen reprezentálhatjuk:

```
1      d2 = [ "a6", "b4", "c2", "d0", "e5", "f7", "g1", "h3" ]
```

- Ehhez más trükköt is használhatnánk – talán a lista minden egyes eleme lehetne egy rendezett n-es, mindkét tengelyen egész koordinátákkal. És mint jó informatikusok, valószínűleg a tengelyek számozását 0-tól számoznánk 1. helyett. Ebben az esetben a reprezentáció a következő lenne:

```
1      d3 = [(0, 6), (1, 4), (2, 2), (3, 0), (4, 5), (5, 7), (6, 1), (7, 3)]
```

- Ezt a reprezentációt figyelembe véve láthatjuk, hogy az első koordináták: 0, 1, 2, 3, 4, 5, 6, 7, és ezek pontosan megfelelnek a párok indexével a listában. Tehát elhagyhatnánk őket és megkapnánk a megoldásnak egy nagyon kompakt alternatív ábrázolását:

```
1      d4 = [6, 4, 2, 0, 5, 7, 1, 3]
```

Ez lesz az, amit használni fogunk, lássuk, hogy hova vezet.

Ez a reprezentáció nem általános

Egy egyszerű reprezentációt hoztunk létre. De működni fog további problémák esetén is? A lista ábrázolásnak van egy megszorítása, hogy mindegyik oszlopban csak egy királynőt helyezhetünk. Mindenesetre ez egy megszorítás – két királynő nem oszthat ugyanazon az oszlopon. Tehát a probléma és az adatok reprezentációja jól illeszkedik.

De megpróbálhatnánk megoldani egy másik problémát a sakktáblán, játszhatunk a sakkfigurákkal, ahol több figura ugyanazt az oszlopot foglalja el, a mi reprezentációnk esetén ez nem működne.

Kicsit mélyebben gondolkozzunk el a problémán. Szerinted véletlen, hogy nincsenek ismétlődő számok a megoldásban? A [6, 4, 2, 0, 5, 7, 1, 3] megoldás tartalmazza a 0, 1, 2, 3, 4, 5, 6, 7 számokat, melyek nem duplikátumok. Más megoldások tartalmazhatnak duplikátumokat vagy nem?

Ha végiggondoljuk, rá kell jönnünk, hogy nem lehetnek duplikátumok a megoldásban: a számok melyek a sorokat jelölik, amelyre a királynőt helyeztünk, nem engedélyezett hogy azon a soron két királynő legyen, tehát nem lehet megoldás, ha duplikátumokat találunk a sor számaiban.

Kulcs pont

A mi reprezentációk során, az N királynő problémának a megoldása a [0 .. N-1] számok permutációja kell legyen.

Nem minden egyes permutáció lesz megoldása a problémának. Például, ebben az esetben [0, 1, 2, 3, 4, 5, 6, 7] a királynők ugyanazon átlón helyezkednek el.

Remek, most úgy tűnik, hogy előreléphetünk, és a gondolkodás helyett elkezdhetjük a kódolást.

Az algoritmusunkat elkezdjük felépíteni. Kezdetünk a $[0..N-1]$ listával, létrehozuk a lista különböző permutációit, és megvizsgáljuk minden egyes permutáció esetében, hogy van-e ütközés (a királynők ugyanazon az átlón vannak-e).

Amennyiben nincs ütközés, az egy megoldása a problémának és kiírjuk. Pontosabban: ha csak a sorok permutációit és a mi kompakt reprezentációinkat használjuk, akkor egy királynő sem ütközhet, sem a sorokban, sem az oszlopokban, és még aggódnunk sem kell ezekért az esetekért. Tehát az ütközések, melyre tesztelnünk kell, az átlókon történhetnek.

Úgy hangzik, mint egy hasznos függvény, amely megvizsgálja, hogy két királynő egy átlón helyezkedik-e el. Minden királynő valamilyen (x, y) pozícióban van. Tehát az $(5,2)$ -es királynő ugyanazon átlón van mint a $(2,0)$? Az $(5,2)$ ütközik a $(3,0)$ -val?

```
1 teszt(ugyanazon_az_atlon(5,2,2,0) == False)
2 teszt(ugyanazon_az_atlon(5,2,3,0) == True)
3 teszt(ugyanazon_az_atlon(5,2,4,3) == True)
4 teszt(ugyanazon_az_atlon(5,2,4,1) == True)
```

Egy kis geometria segít nekünk. Az átlónak 1 vagy -1-es lejtése van. A kérdés, amire szeretnénk a választ, hogy *ugyanakkora-e a távolságuk egymástól az x és az y irányban?* Ha így van, akkor ők egy átlón vannak. Mivel az átló lehet balra és jobbra, ennek a programnak a lényege, hogy kiszámoljuk az abszolút távolságot minden irányba:

```
1 def ugyanazon_az_atlon(x0, y0, x1, y1):
2     """ Az (x0, y0) királynő ugyanazon az átlón van-e (x1, y1) királynővel? """
3     dy = abs(y1 - y0)      # Kiszámoljuk y távolságának abszolút értékét
4     dx = abs(x1 - x0)      # Kiszámoljuk x távolságának abszolút értékét
5     return dx == dy        # Ütköznek, ha dx == dy
```

Ha kimásolod és futtatod, akkor örömmel láthatod, hogy a tesztek sikeresek.

Most nézzük meg, hogy hogyan tudjuk megszerkeszteni a megoldást kézzel. Az egyik királynőt az első oszlopba helyezzük, majd a másodikat a második oszlopba, csak akkor, ha nem ütközik a már sakktáblán lévővel. Aztán egy harmadikat elhelyezünk, és ellenőrizzük a két már balra lévő királynővel szemben. Ha a királynőt a 6. oszlopba tesszük, ellenőrizni kell, hogy van-e ütközés a baloldali oszlopban lévőekkel, azaz például a 0,1,2,3,4,5 oszlopokon lévőekkel.

Tehát a következő elem egy olyan függvény, amely egyrészt egy részben befejezett probléma, mely alapján ellenőrizni tudja, hogy a c oszlopban lévő királynő ütközik-e a bal oldalon lévő királynők egyikével, vagyis a 0,1,2,3,..., $c-1$: oszlopokon.

```
1 # Olyan megoldási esetek, amikor nincsenek ütközések
2 teszt(oszlop_utkozes([6,4,2,0,5], 4) == False)
3 teszt(oszlop_utkozes([6,4,2,0,5,7,1,3], 7) == False)
4
5 # További tesztesetek, amikor többnyire ütközések vannak
6 teszt(oszlop_utkozes([0,1], 1) == True)
7 teszt(oszlop_utkozes([5,6], 1) == True)
8 teszt(oszlop_utkozes([6,5], 1) == True)
9 teszt(oszlop_utkozes([0,6,4,3], 3) == True)
10 teszt(oszlop_utkozes([5,0,7], 2) == True)
11 teszt(oszlop_utkozes([2,0,1,3], 1) == False)
12 teszt(oszlop_utkozes([2,0,1,3], 2) == True)
```

Itt van az a függvény, mely mindegyik esetben sikeres:

```
1 def oszlop_utkozes(bs, c):
2     """ True-val tér vissza, hogyha a c oszlopban lévő királynő ütközik a
3     →tőle balra levőkkel. """
4     for i in range(c): # Nézd meg az összes oszlopot a c-től balra
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4         if ugyanazon_az_atlon(i, bs[i], c, bs[c]):
5             return True
6
7     return False          # Nincs ütközés, a c oszlopban biztonságos helyen
↪ van
```

Végül a programunk által adott eredményt permutáljuk – például minden királynőt elhelyezünk valahol, minden sorban egyet, és minden oszlopban egyet. De van-e a permutáció során átlós ütközés?

```
1 teszt(oszlop_utkozes([6,4,2,0,5,7,1,3]) == False) # A fenti megoldás
2 teszt(oszlop_utkozes([4,6,2,0,5,7,1,3]) == True)  # Felcseréljük az első két
↪ sort
3 teszt(oszlop_utkozes([0,1,2,3]) == True)          # Kipróbáljuk egy 4x4
↪ sakktáblán
4 teszt(oszlop_utkozes([2,0,3,1]) == False)         # Megoldás 4x4-es esetben
```

És a kód, hogy a tesztek sikeresek legyenek:

```
1 def van_utkozes(sakktabla):
2     """ Meghatározzuk, hogy van-e rivális az átlóban.
3         Feltételezzük, hogy a sakktábla egy permutációja az oszlop számoknak,
4         ezért nem kifejezetten ellenőrizzük a sor vagy oszlop ütközéseket.
5     """
6     for col in range(1, len(sakktabla)):
7         if oszlop_utkozes(sakktabla, col):
8             return True
9     return False
```

Összefoglalva, ahogy eddig is tettük, van egy erőteljes függvényünk, amely neve a `van_utkozes`, ez a függvény meg tudja mondani, hogy ez a konfiguráció megoldása-e a 8 királynő problémára. Menjünk tovább, generáljuk több permutációt és találunk további megoldásokat.

14.9. Nyolc királynő probléma, második rész

Ez egy szórakoztató, könnyű rész. Megpróbáljuk megtalálni a `[0, 1, 2, 3, 4, 5, 6, 7]` összes permutációját, amely algoritmikusan egy kihívást jelent, és a *Brute force* módszerrel kezeljük a problémát. Mindent megpróbálunk, hogy megtaláljuk az összes lehetséges megoldást.

Természetesen tudjuk, hogy $N!$ az N elem a permutációinak száma, így a korábbi ötlet alapján tudhatjuk, hogy mennyi időbe telik az összes megoldás megtalálása. Nem túl sokáig, valójában – $8!$ csak 40320 különböző esetet kell ellenőriznünk. Ez sokkal jobb, mintha 64 helyre tennénk a 8 királynőt. Hogyha összeadjuk mennyi lehetőségünk van 8 királynő 64 négyzetre való helyezésére, a képlet (úgynevezett *N elem k-ad osztályú kombinációi*, kiválasztunk $k=8$ négyzetet az elérhető $N=64$ -ből) amely egy elképesztően nagy értéket jelent 4426165368 ($64!/(8! \times 56!)$).

Így egy korábbi kulcsfontosságú betekintés által – csak a permutációkat kell figyelembe venni – csökkenteni tudjuk, azt amit a *probléma térnek* hívunk, 4,4 milliárd esetről 40320-ra!

Azonban nem is fogjuk tudni mindet felfedezni. Amikor bevezettük a véletlenszámok modult, megtanultuk, hogy van egy `shuffle` metódusa, mely véletlenszerűen permutálja a lista elemeket. Így írni fogunk egy „véletlenszerű” algoritmust, hogy megoldásokat találjunk a 8 királynő problémára. Kezdjük a `[0,1,2,3,4,5,6,7]` permutációjával, és ismételten összekeverjük a listát, majd kipróbáljuk ezekre a tesztekre, hogy működik-e! Közben számolni fogjuk, hogy hány próbálkozásra van szükségünk, mielőtt megtaláljuk a megoldásokat, és találunk 10 megoldást (egyszerre több hasonló megoldást is találhatunk, mivel véletlenszerűen keverjük a listát!):

```
1 def main():
2     import random
3     rng = random.Random()    # A generátor létrehozása
4
5     bd = list(range(8))      # Generálja a kezdeti permutációt
6     talalat_szama = 0
7     proba = 0
8     while talalat_szama < 10:
9         rng.shuffle(bd)
10        proba += 1
11        if not van_utkozes(bd):
12            print("Megoldás: {0}, próbálkozás: {1}.".format(bd, proba))
13            proba = 0
14            talalat_szama += 1
15
16 main()
```

Szinte varázslatszerűen és nagyon gyorsan a következőt kapjuk:

```
Megoldás: [3, 6, 2, 7, 1, 4, 0, 5], próbálkozás: 693.
Megoldás: [5, 7, 1, 3, 0, 6, 4, 2], próbálkozás: 82.
Megoldás: [3, 0, 4, 7, 1, 6, 2, 5], próbálkozás: 747.
Megoldás: [1, 6, 4, 7, 0, 3, 5, 2], próbálkozás: 428.
Megoldás: [6, 1, 3, 0, 7, 4, 2, 5], próbálkozás: 376.
Megoldás: [3, 0, 4, 7, 5, 2, 6, 1], próbálkozás: 204.
Megoldás: [4, 1, 7, 0, 3, 6, 2, 5], próbálkozás: 98.
Megoldás: [3, 5, 0, 4, 1, 7, 2, 6], próbálkozás: 64.
Megoldás: [5, 1, 6, 0, 3, 7, 4, 2], próbálkozás: 177.
Megoldás: [1, 6, 2, 5, 7, 4, 0, 3], próbálkozás: 478.
```

Láthatunk egy érdekességet. A 8x8-as sakktáblán 92 különböző megoldás létezik. Véletlenszerűen választjuk ki a 40320 lehetséges permutációk egyikét a reprezentációnkból. Tehát minden egyes megoldás kiválasztása 92/40320 próbálkozással jár. Másképp, átlagosan 40320/92 próbálkozás szükséges – azaz körülbelül 438,26 – mielőtt rátalálnánk a megoldásra. A kiírt próbálkozások száma és a kísérleti adataink nagyon jól illeszkednek az elméletünkhöz!

Mentsük el ezt a kódot későbbre.

A PyGame fejezetben azt tervezzük, hogy olyan modult írunk, amellyel a királynőket rajzolhatunk a sakktáblára, és integráljuk a modult ezzel a kóddal.

14.10. Szójegyzék

bináris keresés (binary search) Egy híres algoritmus, mely megkeres egy értéket a rendezett listában. Mindegyik próbálkozás során felezzük az elemeket, tehát az algoritmus nagyon hatékony.

lineáris (linear) Kapcsolódik egy egyenes vonalhoz. Itt arról beszélünk, hogy hogyan ábrázoljuk grafikusán, hogy hogyan függ az algoritmus számára szükséges idő a feldolgozott adatok méretétől. A lineáris algoritmusok egyenes vonalú grafikokként ábrázolhatók, melyek leírják ezt a kapcsolatot.

teljes keresés (linear search) Olyan keresés, amely minden elemet a listában sorozatosan megvizsgál, ameddig meg nem találja, amit keres. Használjuk egy elem keresésére egy rendezetlen listában.

Összefésülés algoritmus (Merge algorithm) Hatékony algoritmus, amely két már rendezett listát fésül össze, és szintén egy rendezett listát eredményez. Az összefésülés algoritmus egy olyan számítási minta, amelyet kü-

lönböző esetekben lehet adaptálni és újrahasznosítani, például olyan szavak megtalálása esetén, melyek benne vannak a könyvben, de nincsenek a szókinszben.

összehasonlítás (probe) Minden alkalommal, mikor keresünk egy elemet és megvizsgáljuk, ezt összehasonlításnak nevezzük. Az *Iterációk* fejezetben szintén játszottunk egy kitalálós játékot, ahol a felhasználó próbálta kitalálni a számítógép titkos számát. Minden egyes próbálkozást szintén összehasonlításnak nevezzük.

tesztvezérelt fejlesztés (test-driven development) (TDD) Olyan szoftverfejlesztési gyakorlat, amely sok kis iteratív lépésen keresztül ad megoldást, amelyeket automatikus tesztek támasztanak alá, ezeket írjuk meg először, hogy hatékonyabbá tegyék az algoritmus funkcionalitását. (további információt olvashatunk a [Tesztvezérelt fejlesztésről](#) a Wikipédiás cikkben.)

14.11. Feladatok

- Ez a rész az *Alice Csodaországban, ismét!* fejezetről szól, azzal az észrevétellel kezdődött, hogy az összefésülés algoritmus egy olyan mintát használ, melyet más helyzetben újra használhatunk. Módosítsd az összefésülés algoritmusát, és írd meg az alábbi függvényeket, ahogyan itt javasoljuk:
 - Csak azokat az elemeket adja vissza, melyek mindkét listába benne vannak.
 - Csak azokat az elemeket adja vissza, melyek benne vannak az első listában, de nincsenek benne a másodikban.
 - Csak azokat az elemeket adja vissza, melyek benne vannak a második listában, de nincsenek az elsőben.
 - Csak azokat az elemeket adja vissza, melyek vagy az elsőben vagy a másodikban vannak benne.
 - Azokat az elemeket adja vissza az első listából, amelyeket a második lista egy megegyező eleme nem távolít el. Ebben az esetben a második lista egyik eleme „kiüti” az első listában szereplő elemet. Például `kivonas([5, 7, 11, 11, 11, 12, 13], [7, 8, 11])` visszaadja következő listát: `[5, 11, 11, 12, 13]`.
- Módosítsd a királynő programot 4, 12 és 16-os méretű sakktáblák megoldására. Mekkora a legnagyobb méretű sakktábla, melyet az algoritmus egy perc alatt meg tud oldani?
- Módosítsd a királynő programot, hogy megtartsd a megoldások listáját, azért, hogy ugyanazt a megoldást csak egyszer írja ki.
- A sakktáblák szimmetrikusak: ha megoldást keresünk a királynő problémára, akkor a tükörképe is megoldás lesz – vagy**
A Wikipédián néhány lenyűgöző dolgot találhatunk ezzel kapcsolatosan.
 - Írj egy függvényt, amely egy megoldást tükröz az Y tengelyre.
 - Írj egy függvényt, amely egy megoldást tükröz az X tengelyre.
 - Írj egy függvényt, amely a megoldást elforgatja 90 fokkal az óra járásának ellentétesen, és használja a 180 és 270 fokos forgatásokat is.
 - Írj egy függvényt, amely kap egy megoldást, és generál egy szimmetriacsaládot a megoldásnak megfelelően. Például, a `[0,4,7,5,2,6,1,3]` megoldás szimmetriái:

```
[ [0, 4, 7, 5, 2, 6, 1, 3], [7, 1, 3, 0, 6, 4, 2, 5],  
  [4, 6, 1, 5, 2, 0, 3, 7], [2, 5, 3, 1, 7, 4, 6, 0],  
  [3, 1, 6, 2, 5, 7, 4, 0], [0, 6, 4, 7, 1, 3, 5, 2],  
  [7, 3, 0, 2, 5, 1, 6, 4], [5, 2, 4, 6, 0, 3, 1, 7] ]
```
- Most módosítsd a programot, hogy ne sorolja fel azokat a megoldásokat, amelyek ugyanahhoz a családnak tartoznak. Csak egyedi családokból származó megoldásokat írd ki.

5. Egy informatikus minden héten négy lottószelvényt vásárol. Ő mindig ugyanazokat a prímszámokat választja, abban reménykedve, hogyha valaha megnyeri a jackpotot, a TV-ben illetve a Facebookon elmondja a titkát. Ez hirtelen egy széleskörű érdeklődést válthat ki a prímszámok iránt, és ez lesz az az esemény, amely beharangoz egy új felvilágosodást. A heti szelvényeit Pythonba egy beágyazott listával ábrázoljuk:

```
szelvenyek = [ [ 7, 17, 37, 19, 23, 43],  
               [ 7, 2, 13, 41, 31, 43],  
               [ 2, 5, 7, 11, 13, 17],  
               [13, 17, 37, 19, 23, 43] ]
```

Végezd el ezeket a feladatokat!

- (a) Minden lottószelvény húzáshoz hat véletlenszerű golyó tartozik, sorszámozva 1-től 49-ig. Írj egy függvényt, mely visszatér egy lottóhúzással.
- (b) Írj egy függvényt, amely összehasonlít egy egyszerű szelvényt és egy lottóhúzást, visszaadja a találatok számát:

```
teszt(lotto_talalat([42,4,7,11,1,13], [2,5,7,11,13,17]) == 3)
```

- (c) Írj egy függvényt, amely megkapja a szelvények és húzások listáját, és visszatér egy olyan listával, mely megadja, hogy minden egyes szelvényen hány találat volt:

```
teszt(lotto_talalatok([42,4,7,11,1,13], my_tickets) == [1,2,3,1])
```

- (d) Írj egy függvényt, amely megkapja az egész számok listáját, és visszatér a listában szereplő prímszámok számával:

```
teszt(primek_szama([42, 4, 7, 11, 1, 13]) == 3)
```

- (e) Írj egy függvényt, amely felderíti, hogy vajon az informatikus elhibázott-e prímszámokat a négy szelvényén. Térjen vissza a prímszámok listájával, amelyeket elhibázott:

```
teszt(hianyzo_primek(my_tickets) == [3, 29, 47])
```

- (f) Írj egy függvényt, amely ismételten új húzást generál, és összehasonlítja a húzásokat a négy szelvénnel!
- Számold meg hány húzás szükséges addig, amíg az informatikus szelvényein legkevesebb három találatot kap! Próbáld ki a kísérletet hússzor, és átlagold a szükséges húzások számát!
 - Hány húzásra van szükség átlagosan, mielőtt legalább 4 találatot kapna?
 - Hány húzásra van szükség átlagosan az 5 találathoz? (Tipp: ez eltarthat egy ideig. Jó lenne néhány pontot kiíratni, mint az előrehaladás folyamatát, mely megjelenik minden 20 kísérlet befejezése után.)

Figyeld meg, hogy itt nehézségeket okoz a vizsgálati esetek létrehozása, mert a véletlenszámok nem determinisztikusak. Az automatizált tesztelés csak akkor működik, ha már tudod a választ!

6. Olvasd el az *Alice Csodaországban*-t. Elolvashatod a könyv szöveges verzióját, vagy ha van e-book olvasó szoftver a számítógépeden vagy egy Kindle, iPhone, Android, stb. megtalálhatod az eszközödnek megfelelő verziót a következő <http://www.gutenberg.org/> weboldalon. Megtalálható html és pdf változatban is, képekkel és több ezer más klasszikus könyvvel!

15. fejezet

Osztályok és objektumok – alapok

15.1. Objektumorientált programozás

A Python **objektumorientált programozási nyelv**, ami azt jelenti, hogy az **objektumorientált programozást (OOP)** támogató eszközkészletet nyújt a programozók számára.

Az objektumorientált programozás gyökerei egészen az 1960-as évekig nyúlnak vissza, azonban csak az 1980-as évek közepére vált az új szoftverek létrehozásánál használt vezető **programozási paradigmává**. Úgy alkották meg, hogy képes legyen kezelni a szoftveres rendszerek méretének és komplexitásának gyors növekedését, és megkönnyítse a nagy, bonyolult rendszerek karbantartását, az idővel szükségessé váló módosítások elvégzését.

A korábban elkészített programjainak többségénél a **procedurális programozási** paradigmát használtuk. A procedurális programozásnál a hangsúly az adatokat feldolgozó függvényeken és eljárásokon van. Az objektumorientált programozásnál viszont olyan **objektumok** létrehozására törekszünk, amelyek az adatot és a hozzájuk kötődő funkcionalitást foglalják egybe. (Már találkoztunk teknőc, sztring és véletlen számokat generáló objektumokkal – csak hogy megnevezzünk néhány esetet, amikor objektumokkal dolgoztunk.)

Az objektumok definíciója általában megfelel valamilyen valós világbeli objektumnak vagy fogalomnak, az objektumokon operáló metódusok pedig az objektumok valós világban történő egymásra hatását írják le.

15.2. Saját, összetett adattípusok

Az eddigiekben az `str`, `int`, `float` és a `Turtle` osztályokkal találkoztunk. Immár készen állunk egy saját osztály, a `Pont` definiálására.

Vegyük alapul a matematikai pont fogalmat. Két dimenzióban a pont két szám (két koordináta), amelyet együtt, egyetlen objektumként kezelünk. A pontok megadásánál a koordinátákat gyakran zárójelek közé írjuk, egymástól vesszővel elválasztva. Például a $(0, 0)$ az origót reprezentálja, az (x, y) pedig egy olyan pontot, amely az origótól x egységgel jobbra, és y egységgel felfele van.

A tipikus pont műveletek közé tartozik egy pont origótól, vagy egy másik ponttól mért távolságának meghatározása, két pontot összekötő szakasz felezőpontjának számítása, vagy annak eldöntése, hogy egy pont egy téglalapon vagy körön belülre esik-e. Hamarosan látni fogjuk, miként szervezhetjük egybe ezeket a műveleteket az adatokkal.

A Pythonban természetes, hogy a pontokat két számmal reprezentáljuk, a kérdés csak az, hogyan szervezzük ezeket egy összetett objektumba. Az értékpárok használta gyors, de nem elegáns megoldás. Néhány alkalmazásnál viszont jó választás lehet.

Alternatív megoldásként definiálhatunk egy új **osztályt**, ami ugyan több erőfeszítést igényel, de hamarosan nyilvánvalóak lesznek az előnyei is. Azt szeretnénk, hogy minden pontunknak legyen egy *x* és egy *y* attribútuma, ezért az első osztály definíción az alábbi módon néz ki:

```
1 class Pont:
2     """A Pont osztály (x, y) koordinátáinak reprezentálására és
   ↪manipulálására. """
3
4     def __init__(self):
5         """ Egy új, origóban álló pont létrehozása. """
6         self.x = 0
7         self.y = 0
```

Az osztály definíciók bárhol állhatnak a programokon belül. Általában a szkriptek elejére tesszük őket (az `import` utasítások után), de a programozók és a programnyelvek egy része is az osztályok külön modulba való elhelyezését támogatja inkább. (A könyvbeli példánknál nem fogunk külön modult használni.) Az osztályokra vonatkozó szintaktikai szabályok ugyanazok, mint a többi összetett utasításnál. Van egy, a `class` kulcsszóval kezdődő fejléc, amelyet az osztály neve követ, és kettősponttal zárul. Az utasítások behúzása, vagyis az indentálási szintek határozzák meg, hol ér véget az osztály.

Ha az osztály fejlécét követő első sor egy sztring konstans, akkor dokumentációs megjegyzéseként lesz kezelve, számos eszköz fel fogja ismerni. (A függvényeknél is így működik a dokumentációs sztring.)

Minden osztályban kötelező egy `__init__` nevű, **inicializáló metódus** szerepeltetése, amely automatikusan meghívásra kerül, minden alkalommal, amikor egy új példány jön létre az osztályból (most a `Pont`-ból). Az inicializáló metódusban a programozók kezdőértékek / kezdőállapotok megadásával beállíthatják az új példánynál szükséges attribútumokat. A `self` (igazából bármilyen nevet választhatnánk, de ez a konvenció) paraméter értéke automatikusan az újonnan létrehozott, inicializálandó példány referenciája lesz.

Használjuk is fel az új `Pont` osztályunkat:

```
1 p = Pont() # A Pont osztály egy objektumának létrehozása (példányosítás)
2 q = Pont() # Egy második Pont objektum készítése
3
4 # Minden Pont objektum saját x és y attribútumokkal rendelkezik
5 print(p.x, p.y, q.x, q.y)
```

A program kimenete

```
0 0 0 0
```

ugyanis az inicializálás során két attribútumot hoztunk létre *x*-et és *y*-t, mindkettőt 0 értékkel.

Ennek ismerősnek kell lennie, hiszen már használtunk korábban osztályokat több teknőcpéldány létrehozásához is:

```
1 from turtle import Turtle
2
3 Eszti = Turtle()      # Teknőc objektumok példányosítása
4 Sanyi = Turtle()
```

A *p* és *q* változók egy-egy új `Pont` objektum referenciáját tartalmazzák. A `Turtle`-höz vagy `Pont`-hoz hasonló, új objektum példányt előállító függvényeket **konstruktoroknak** nevezzük. Az osztályok automatikusan biztosítanak egy, az osztályával azonos nevű, konstruktort.

Talán segíthet, ha az osztályt úgy képzeljük el, mint egy objektum készítő *gyárat*. Az osztály maga nem egy `Pont` példány, de tartalmazza a `Pont` példányok előállításához szükséges eszközöket. Minden egyes konstruktor hívással arra kérjük a gyárat, hogy készítsen nekünk egy új objektumot. Amint legördül az objektum a gyártósorról, végrehajtódik az inicializáló metódusa, ami beállítja az objektum tulajdonságait a gyári alapbeállításoknak megfelelően.

Az „új objektum készítése” és a „gyári beállítások elvégzése” tevékenységek kombinálásával nyert folyamatot **példányosításnak** nevezzük.

15.3. Attribútumok

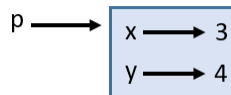
Az objektumoknak, akár a valós világbeli objektumoknak, vannak **attribútumai** (tulajdonságai) és metódusai is.

Az attribútumokat a pont operátor segítségével módosíthatjuk:

```
1 p.x = 3
2 p.y = 4
```

A modulok és az osztályok is saját névtérre alkotnak. A bennük álló nevek, az úgynevezett **attribútumok**, azonos szintaktikával érhetők el. Ebben az esetben a kiválasztott attribútum a példány adata eleme.

A következő állapotdiagram mutatja az értékadások eredményét:



A `p` változó egy 2 attribútumot tartalmazó `Pont` objektumra hivatkozik. Az objektum attribútumai egy-egy számot tartalmaznak.

Az attribútumok értékeit ugyanazzal a szintaktikával érhetjük el:

```
1 print(p.y)    # 4-et ír ki
2 x = p.x
3 print(x)      # 3-at ír ki
```

A `p.x` kifejezés jelentése: „Menj el a `p` által hivatkozott objektumhoz, és kérd le az `x` értéket.” A fenti példában az `x` változóhoz rendeltük hozzá az értéket. Az `x` változó (itt a globális névtérben áll), és az `x` attribútum (a példány névtéréhez tartozik) közt nem lép fel névütközés. A minősített nevek használatának célja éppen az, hogy egyértelműen meghatározhassuk melyik változóra, melyik programozási eszközre hivatkozunk.

A pont operátor kifejezésekben is alkalmazható, így a következő kifejezések is szabályosak:

```
1 print("(x={0}, y={1})".format(p.x, p.y))
2 origotol_mert_tavolsag_negyzete = p.x * p.x + p.y * p.y
```

Az első sor kimenete `(x=3, y=4)`. A második sor 25-ös értéket számít ki.

15.4. Az inicializáló metódus továbbfejlesztése

Egy a (7, 6) koordinátán álló pont létrehozásához most három kódsorra van szükségünk:

```
1 p = Pont()
2 p.x = 7
3 p.y = 6
```

Általánosabbá tehetjük a konstruktort, ha újabb paramétereket adunk az `__init__` metódushoz, ahogy azt az alábbi példa is mutatja:

```
1 class Pont:
2     """A Pont osztály (x, y) koordinátáinak reprezentálására és
   ↪manipulálására. """
3
4     def __init__(self, x=0, y=0):
5         """ Egy új, (x, y) koordinátán álló pont készítése. """
6         self.x = x
7         self.y = y
8
9     # További, osztályon kívül álló utasítások
```

Az `x` és `y` paraméter is opcionális, ha a hívó nem ad át argumentumokat, akkor 0 alapértelmezett értéket kapnak. Lássuk most működés közben a továbbfejlesztett osztályunkat. Szúrjuk be az alábbi utasításokat a Pont osztály alá, az osztályon kívülre.

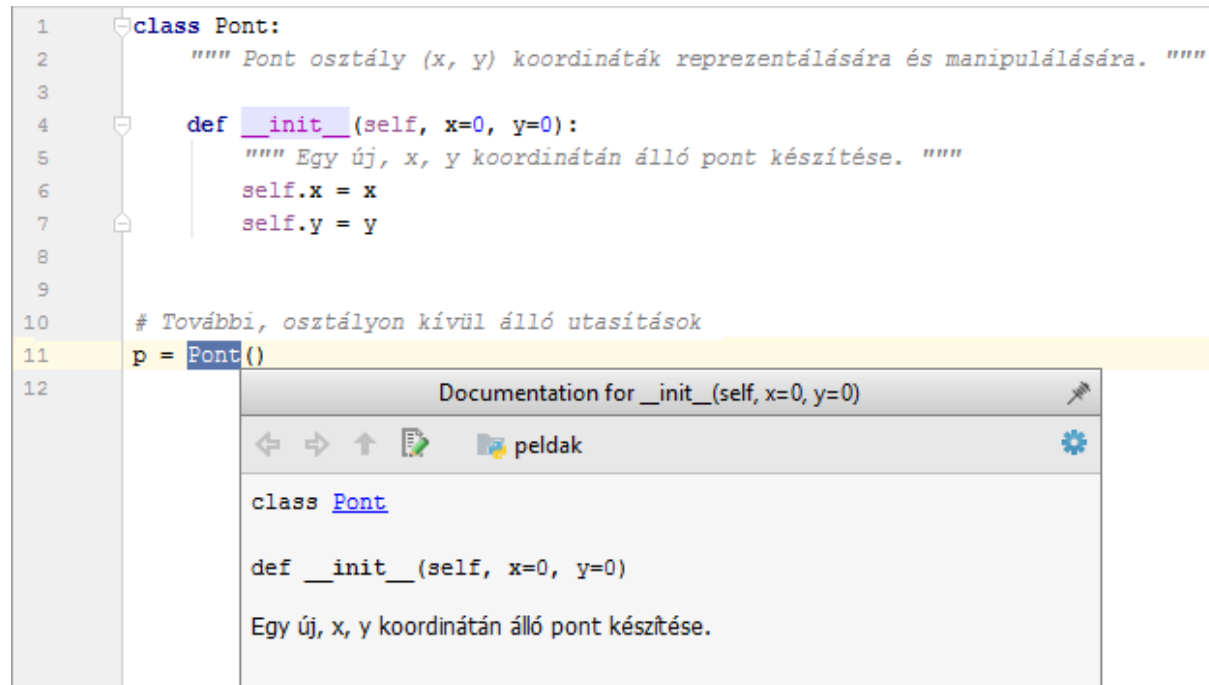
```
1 p = Pont(4, 2)
2 q = Pont(6, 3)
3 r = Pont()      # r az origót (0, 0) reprezentálja
4 print(p.x, q.y, r.x)
```

A program a 4 3 0 értékeket jeleníti meg.

Technikai részletek ...

Ha nagyon szeretnénk akadékoskodni, akkor mondhatjuk, hogy az `__init__` metódus dokumentációs sztringje pontatlan. Az `__init__` ugyanis nem hoz létre objektumot (nem foglal memóriát a számára), csak beállítja a már létrejött objektum tulajdonságait a gyári beállításoknak megfelelően.

A PyCharm szerű eszközök viszont tudják, hogy a példányosítás – létrehozás és inicializáció – együtt megy végbe, ezért az *inicializálóhoz* tartozó súgót (dokumentációs sztringet) jelenítik meg a konstruktorokhoz. A dokumentációs sztringet tehát úgy írtuk meg, hogy a konstruktor hívásakor segítse a Pont osztályunkat felhasználó programozót.



The screenshot shows the PyCharm IDE with the `Pont` class definition. The `__init__` method is highlighted, and a pop-up window titled "Documentation for __init__(self, x=0, y=0)" is displayed. The pop-up window shows the class definition and the docstring for the `__init__` method.

```
1 class Pont:
2     """ Pont osztály (x, y) koordináták reprezentálására és manipulálására. """
3
4     def __init__(self, x=0, y=0):
5         """ Egy új, x, y koordinátán álló pont készítése. """
6         self.x = x
7         self.y = y
8
9
10    # További, osztályon kívül álló utasítások
11    p = Pont()
12
```

Documentation for `__init__(self, x=0, y=0)`

```
class Pont

def __init__(self, x=0, y=0)

Egy új, x, y koordinátán álló pont készítése.
```

15.5. Újabb metódusok hozzáadása az osztályunkhoz

Itt jön elő igazán, miért előnyösebb egy `Pont`-szerű osztályt használni, egy egyszerű $(6, 7)$ értékpár helyett. Olyan metódusokkal bővíthetjük a `Pont` osztályt, amelyek értelmes pontműveleteket takarnak. Lehet, hogy más értékpárok esetében nem is lenne értelmük, hiszen egy $(12, 25)$ értékpár reprezentálhat akár egy hónapot és napot is (például karácsony egyik napját). Pontok esetében értelmezhető az origótól való távolság számítása, de a $(\text{hónap}, \text{nap})$ párok esetében nincs semmi értelme. A $(\text{hónap}, \text{nap})$ adatokhoz más műveleteket szeretnénk, talán egy olyat, amelyik megadja, hogy a hét mely napjára esnének 2020-ban.

A `Pont`-hoz hasonló osztályok készítése kivételes mértékű „rendszerzési erővel” ruházza fel a programjainkat és a gondolkodásunkat. Egy csoportba foglalhatjuk a szóba jöhető műveleteket és azokat az adattípusokat, amelyekre alkalmazhatók, ráadásul az osztály minden példánya saját állapottal rendelkezik.

A **metódusok** függvényként viselkednek, de mindig egy adott példányra vannak meghívva, gondoljunk csak az `Eszti.right(90)` kifejezésre. A metódusokhoz, akárcsak az adatokhoz, a pont operátort alkalmazva férhetünk hozzá.

Adjunk az osztályhoz egy újabb metódust, hogy jobban megértsük a metódusok működését. Legyen ez az `origotol_mert_tavolsag`. A szkript végén – az osztályon kívül – készítünk néhány pont példányt is, megjelöljük az attribútumaikat, és meghívjuk rájuk az új metódust.

```
1 class Pont:
2     """ Pont osztály (x, y) koordináták reprezentálására és manipulálására. """
3     ↪ ""
4
5     def __init__(self, x=0, y=0):
6         """ Egy új, x, y koordinátán álló pont készítése. """
7         self.x = x
8         self.y = y
9
10    def origotol_mert_tavolsag(self):
11        """ Az origótól mért távolság számítása. """
12        return ((self.x ** 2) + (self.y ** 2)) ** 0.5
13
14    #tesztek
15    p = Pont(3, 4)
16    print(p.x)
17    print(p.y)
18    print(p.origotol_mert_tavolsag())
19
20    q = Pont(5, 12)
21    print(q.x)
22    print(q.y)
23    print(q.origotol_mert_tavolsag())
24
25    r = Pont()
26    print(r.x)
27    print(r.y)
28    print(r.origotol_mert_tavolsag())
```

A szkript futtatása után az alábbi kimenet jelenik meg:

```
3
4
5.0
5
12
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
13.0
0
0
0.0
```

A metódusok definiálásakor az első paraméter a manipulálandó példányra hivatkozik. Korábban már jeleztük, hogy ennek a paraméternek, megállapodás szerint, `self` a neve. Figyeld meg, hogy az `origotol_mert_tavolsag` metódus hívója nem ad át explicit módon argumentumot a `self` paraméter számára, ez az átadás a háttérben megy végbe.

15.6. Példányok felhasználása argumentumként és paraméterként

Az objektumokat a megszokott módon adhatjuk át argumentumként. Néhány teknőcös példában már láttunk is ilyet. Amikor a `Feltételes` utasítások fejezetben a `rajzolj_oszlopot`-hoz hasonló függvényeknek adtuk át a teknőcöket, megfigyelhettük, hogy bármelyik teknőcpéldányt adtuk is át, a függvények képesek voltak irányítani.

Tartsd észben, hogy a változóink csak az objektum referenciáját tartalmazzák. Ha `Eszt`-t átadjuk egy függvénynek, akkor egy fedőnév keletkezik, tehát a hívó és a hívott függvény is rendelkezik egy-egy referenciával, de teknőcből csak egy van!

Itt egy egyszerű függvény, amely `Pont` objektumokat fogad:

```
1 def pont_kiiras(pt):
2     print("{0}, {1}".format(pt.x, pt.y))
```

A `pont_kiiras` egy `Pont` objektumot vár argumentumként, és formázva megjeleníti. Ha meghívjuk a `pont_kiiras` függvényt a korábban definiált `p` pontra, akkor a kimenet `(3, 4)` lesz.

15.7. Egy példány átalakítása sztringgé

Az objektumorientált nyelvekben jártas programozók többsége valószínűleg nem tenne olyat, amit mi tettünk az előbb a `pont_kiiras` függvényen belül. Ha osztályokkal és objektumokkal dolgozunk, akkor jobb megoldás egy új metódust adni az osztályhoz. Viszont nem akarunk csevegni, a `print` függvényt hívó, metódust írni. Előnyösebb megközelítés, ha minden egyes példánynak van egy olyan metódusa, amellyel képes egy saját magát reprezentáló sztring előállítására. Hívjuk először `sztringge_alakitas`-nak:

```
1 class Pont:
2     # ...
3
4     def sztringge_alakitas(self):
5         return "{0}, {1}".format(self.x, self.y)
6
7     #Most már írhatunk ilyesmit is:
8     p = Pont(3, 4)
9     print(p.sztringge_alakitas())
```

A szkriptet futtatva a `(3, 4)` kimenet jelenik meg.

De hát van nekünk egy `str` típuskonverterünk, amely az objektumokat sztringgé alakítja, vagy nem? De igen! Nem hívja meg automatikusan a `print` függvény, amikor meg kell jelenítenie valamit? De bizony meghívja! Csakhogy nem pont azt teszi, amit várnánk tőle. A

```
1 p = Pont(3, 4)
2 print(str(p))
3 print(p)
```

kódrészlet az alábbihoz hasonló eredményt szolgáltat:

```
<__main__.Pont object at 0x02CF35F0>
<__main__.Pont object at 0x02CF35F0>
```

A Pythonnak van egy okos trükkje a helyzet megoldására. Ha az új metódusunkat `sztringge_alakitas` helyett `__str__`-nek nevezzük, akkor a Python értelmező mindig az általunk írt metódust fogja meghívni, ha szükség van egy `Pont` objektum sztringgé alakítására.

Alakítsuk át az osztályt:

```
1 class Pont:
2     # ...
3
4     def __str__(self):    # A metódus átnevezése az egyetlen feladatunk
5         return "{0}, {1}".format(self.x, self.y)
6
7 #teszt
8 p = Pont(3, 4)
9 # Az str(p) kifejezés kiértékelésénél a Python az általunk írt
10 # __str__ metódust hívja meg.
11 print(str(p))
12 print(p)
```

Most már nagyszerűen néz ki!

```
(3, 4)
(3, 4)
```

15.8. Példányok, mint visszatérési értékek

A függvények és metódusok képesek objektum példányok visszaadására. Például legyen adott két `Pont` objektum, és határozzuk meg a súlypontjukat (a két pontot összekötő szakasz felezőpontját). Először szokványos függvényként írjuk meg, amit rögtön fel is használunk majd.

```
1 def súlypont_szamitas(p1, p2):
2     """ Visszatér a p1 és p2 pontok súlypontjával. """
3     mx = (p1.x + p2.x)/2
4     my = (p1.y + p2.y)/2
5     return Pont(mx, my)
6
7 #teszt
8 p = Pont(3, 4)
9 q = Pont(5, 12)
10 r = súlypont_szamitas(p, q)
11 print(r)
```

A `súlypont_szamitas` függvény egy új `Pont` objektummal tér vissza. A szkript a `(4.0, 8.0)` kimenetet adja.

Most tegyük meg ugyanezt egy metódussal. Tegyük fel, hogy van egy `Pont` objektumunk, és egy olyan metódust kívánunk írni, amely meghatározza a pont és egy argumentumként kapott másik pont súlypontját:

```
1 class Pont:
2     # ...
3
4     def súlypont_szamitas(self, másik_pont):
5         """ A súlypontom a másik ponttal. """
6         mx = (self.x + másik_pont.x)/2
7         my = (self.y + másik_pont.y)/2
8         return Pont(mx, my)
9
10
11 #Példa a metódus felhasználása:
12 p = Pont(3, 4)           # Az adott pont objektum
13 q = Pont(5, 12)          # Egy másik pont objektum
14 r = p.súlypont_szamitas(q) # A súlypont számítása
15 print(r)                 # és megjelenítése
```

A metódus megfelel a korábbi függvénynek. A kimenet ezúttal is (4.0, 8.0).

Habár a fenti példában minden egyes pontot egy változóhoz rendelünk hozzá, erre nincs szükség. Ahogy a függvényhívások, úgy a metódushívások és a konstruktorok is egymásba ágyazhatók. Mindez egy alternatív, változók nélküli, megoldáshoz vezethet:

```
1 print(Pont(3, 4).súlypont_szamitas(Pont(5, 12)))
```

A kimenet természetesen a korábbival azonos.

15.9. Szemléletváltás

A `súlypont_szamitas(p, q)` függvényhívás szintaktikája azt sugallja, hogy a pontok elszenvedői, és nem végrehajtói a műveletnek. Valami ilyesmit mond: „Itt van két pont, amelyeknek most meghatározzuk a súlypontját.” A függvény a cselekvő fél.

Az objektumorientált programozás világában az objektumokat tekintjük cselekvő félnek. Egy `p.súlypont_szamitas(q)`-hoz hasonló hívás azt sugallja: „Hé, *p* pont, nesze itt egy *q* pont. Számítsd ki a súlypontotokat!”

A teknőcsök korábbi bemutatásakor is objektumorientált stílust használtunk, amikor az `Eszti.forward(100)` kifejezéssel megkértük a teknőcot, hogy tegyen meg előrefele adott számú lépést.

Ez a szemléletváltás lehetne udvariasabb is, de kezdetben nem biztos, hogy nyilvánvalók az előnyei. Időnként rugalmasabb, könnyebben újrafelhasználható és karbantartható függvényeket írhatunk, ha a felelősséget a függvényekről az objektumokra ruházzuk át.

Az objektumorientált programozás legfontosabb előnye, hogy jobban illeszkedik a valós világhoz, és a problémamegoldás során meghatározott lépésekhez. A valóságban a főzés metódus a mikrohullámú sütő része. Nem ül egy főzés függvény a konyha sarkában, amelybe belerakhatnánk a mikrohullámú sütőt! Hasonlóképpen, a mobiltelefon saját metódusait használjuk egy SMS elküldésére vagy csendes üzemmódra váltásra. A valós világban az objektumokhoz szorosan kötődnek a hozzájuk tartozó funkcionalitások, az OOP lehetőséget ad arra, hogy a programszervezés pontosan tükrözze ezt.

15.10. Az objektumoknak lehetnek állapotai

Az objektumok akkor a leghasznosabbak, amikor valamilyen, időről-időre frissítendő állapot tárolására is szükségünk van. Tekintsünk egy teknőc objektumot. Az állapota tartalmazza a pozícióját, az irányt, amerre néz, a színét és

az alakját. A `left(90)`-szerű metódushívások a teknőc irányát, a `forward` metódus a pozícióját frissíti, és így tovább.

Egy bankszámla objektum legfontosabb komponense az aktuális egyenleg, és talán az összes tranzakciót tartalmazó napló. A metódusok megengednék az aktuális egyenleg lekérdezését, új pénz elhelyezését a számlán, és pénzek kifizetését. A kifizetés tartalmazná az összeget, valamint egy leírást, így hozzá lehetne adni a tranzakciós naplóhoz. Szeretnénk, ha létezne egy olyan metódus is, amely a tranzakciókat képes megjeleníteni.

15.11. Szójegyzék

attribútum (attribute) Egy névvel rendelkező adatelem. A példány egy alkotóeleme.

inicializáló metódus (initializer method) Egy speciális, `__init__` nevű metódus Pythonban, amely automatikusan meghívásra kerül, ha egy új objektum jön létre. Az objektumok kezdeti állapotát állítja be (a gyári alapbeállításokat).

konstruktor (constructor) Minden osztály rendelkezik egy „gyárral”, amely új példányokat képes előállítani. A neve azonos az osztály nevével. Ha az osztálynak van *inicializáló metódusa*, akkor az állítja be a frissen létrehozott objektumok attribútumaihoz a megfelelő kezdőértéket, vagyis beállítja az objektumok állapotát.

metódus (method) Egy osztályon belül definiált függvény. Az osztály példányaira hívható meg.

objektum (object) Összetett adattípus, amelyet gyakran használnak a valós világbeli dolgok és fogalmak modellezésére. Egymáshoz köti az adatot és az adott adattípus esetében releváns műveleteket. A példány és az objektum fogalmakra szinonimaként tekinthetünk.

objektumorientált nyelv (object-oriented language) Olyan programozási nyelv, amely az objektumorientált programozást támogató eszközöket biztosít, például lehetőséget ad saját osztály definiálására, öröklődés megvalósítására.

objektumorientált programozás (object-oriented programming) Egy hatékony programozási stílus, amelyben az adatok és a rajtuk dolgozó műveletek objektumokba vannak szervezve.

osztály (class) Egy felhasználó által definiált, összetett típus. Az osztályokra tekinthetünk úgy is, mint az osztályba tartozó objektumok sablonjára. (Az iPhone egy osztály. A becslések szerint 2010 decemberéig 50 millió példány kelt el.)

példány (instance) Egy objektum, amelynek típusa valamilyen osztály. A példány és az objektum fogalmakra szinonimaként tekinthetünk.

példányosítás (instantiate) Egy osztály egy új példányának létrehozása, és az inicializálójának futtatása.

15.12. Feladatok

- Írd át a *Produktív függvények* fejezetben található `tavolsag` függvényt úgy, hogy négy szám típusú paraméter helyett két `Pont` típusú paramétere legyen!
- Bővítsd egy `tukrozes_x_tengelyre` nevű metódussal a `Pont` osztályt! A metódus térjen vissza egy új `Pont` példánnyal, mely az aktuális pont `x`-tengelyre vett tükörképe. Például a `Pont(3, 5).tukrozes_x_tengelyre()` eredménye `(3, -5)`.
- Adj hozzá az osztályhoz egy `origotol_mert_meredekseg` nevű metódust, amely az origó és a pont közötti egyenes szakasz meredekségét határozza meg! A `Pont(4, 10).origotol_mert_meredekseg()` eredménye például 2.5.

Milyen esetben nem működik helyesen a metódus?

4. Az egyenes egyenlete $y = ax + b$ (vagy másképpen $y = mx + c$). Az a és b együtthatók egyértelműen meghatározzák az egyenest. Írj egy metódust a `Pont` osztályon belül, amely az aktuális objektum és egy másik, argumentumként kapott pont alapján meghatározza a két ponton átmenő egyenest! A metódusnak az egyenes együtthatóival, mint értékpárral, kell visszatérniük:

```
print(Pont(4, 11).egyenes(Pont(6, 15))) # kimenet: (2, 3)
```

A kimenet azt mutatja, hogy a két ponton átmenő egyenes az $y = 2x + 3$. Milyen esetben működik majd rosszul a metódus?

5. Határozd meg egy kör középpontját négy, a kör területére eső pont alapján! Milyen esetben nem fog működni a függvény?

Segítség: Tudnod kell, hogyan old meg a geometriai problémát, *mielőtt* a gondolataid a programozás körül kezdenének forogni. Nem tudod leprogramozni a megoldást, ameddig nem érted, hogy mit akarsz a géppel megcsináltatni.

6. Készíts egy új `SMS_tarolo` osztályt! Az osztály olyan objektumokat példányosít majd, amelyek hasonlítanak a telefonon lévő bejövő / kimenő üzenet tárolókra:

```
bejovo_uzenetek = SMS_tarolo()
```

Ez a tároló több SMS üzenetet tárol (tehát a belső állapota az üzenetek listája lesz). Minden üzenetet egy rendezett 4-es reprezentáljon:

```
(olvasott_e, kuldo_szama, erkezesi_ido, SMS_szovege)
```

A bejövő üzenetek tárolójának az alábbi metódusokat kell biztosítania:

```
bejovo_uzenetek.beerkezo_uzenet_hozzaadasa(kuldo_szama, erkezesi_ido, SMS_szovege)
# Készít egy új rendezett 4-est az SMS számára,
# és beszúrja őket a tárolóba a többi üzenet után.
# Az üzenet készítésénél az olvasott_e állapotát
# hamisra (False) állítja.

bejovo_uzenetek.uzenetek_szama()
# Visszatér a bejovo_uzenetek tárolóban lévő SMS-ek számával

bejovo_uzenetek.olvasatlan_uzenetek_indexeinek_lekerese()
# Visszatér az összes olvasatlan SMS indexét tartalmazó listával.

bejovo_uzenetek.uzenet_lekerese(i)
# Visszatér az uzenet[i]-hez tartozó (kuldo_szama, erkezesi_ido, SMS_szovege) 4-
→essel.
# Az üzenet státuszát olvasottra állítja.
# Ha nincs üzenet az i. indexen, akkor a visszatérési érték None.

bejovo_uzenetek.torol(i) # Kitörli az i. pozícióban álló üzenetet.
bejovo_uzenetek.mindent_torol() # Kitörli az összes üzenetet a bejövő SMS-ek_
→tárolójából.
```

Írd meg az osztályt, készíts egy SMS tároló objektumot, írd teszteket a metódusokhoz és implementáld őket!

16. fejezet

Osztályok és objektumok – ássunk egy kicsit mélyebbre

16.1. Téglalapok

Tegyük fel, hogy létre akarunk hozni egy osztályt egy XY síkon elhelyezkedő téglalap reprezentálására. A kérdés az, hogy milyen információkat kell megadnunk egy ilyen téglalap leírásához. Nem szeretnénk elbonyolítani a dolgot, ezért feltételezzük, hogy a téglalap függőleges vagy vízszintes orientációjú, soha nem áll eltérő szögben.

Több lehetőség is adódik. Megadhatjuk a téglalapot a középpontjával (két koordináta) és a méretével (magasság, szélesség), vagy az egyik csúcspontjával és a méretével, vagy két ellentétes csúcspontjával is. A konvenció az, hogy a bal felső csúcspontot és a téglalap méretét használjuk.

Ismét definiálunk egy új osztályt, egy inicializáló és egy az objektumot sztringgé alakító metódussal:

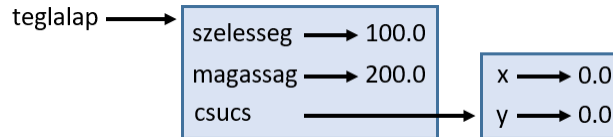
```
1 class Téglalap:
2     """ Egy osztály a téglalapok előállításához. """
3
4     def __init__(self, poz, sz, m):
5         """ Inicializálja a téglalapot a poz pozícióra
6             sz szélességgel m magassággal. """
7         self.csucs = poz
8         self.szelesseg = sz
9         self.magassag = m
10
11     def __str__(self):
12         return "({0}, {1}, {2})"
13             .format(self.csucs, self.szelesseg, self.magassag)
14
15 teglalap = Téglalap(Pont(0, 0), 100, 200)
16 bomba = Téglalap(Pont(100, 80), 5, 10)    # A videojátékomban.
17 print("teglalap: ", teglalap)
18 print("bomba: ", bomba)
```

A bal-felső csúcs megadásához egy Pont objektumot ágyazunk be az új Téglalap objektumunkba (mivel ezt használtuk az előző fejezetben). Készítünk két Téglalap példányt, majd megjelenítjük őket:

```
teglalap: ((0, 0), 100, 200)
bomba: ((100, 80), 5, 10)
```

A pont operátorok halmozódhatnak. A `teglalap.csucs.x` kifejezés jelentése: „Menj el a `teglalap` által hivatkozott objektumhoz, válaszd ki a `csucs` nevű attribútumát, majd menj el ahhoz az objektumhoz és válaszd ki az `x` nevű attribútumát.”

Az ábra mutatja az objektum állapotát:



16.2. Az objektumok módosíthatók

Az objektumok állapotát módosíthatjuk, ha valamelyik attribútumához új értéket rendelünk. Ha például növelni szeretnénk a téglalap méretét anélkül, hogy a pozícióját megváltoztatnánk, módosíthatjuk a `szelesseg` és / vagy a `magassag` attribútumokat:

```
1 teglalap.szelesseg += 50
2 teglalap.magassag += 100
```

Erre a célra szinte biztosan be szeretnénk vezetni egy metódust, hogy az osztályba foglaljuk ezt a műveletet. Egy olyan metódust is biztosítunk majd, amellyel más helyre mozgatható a téglalap:

```
1 class Teglalap:
2     # ...
3
4     def noveles(self, delta_szelesseg, delta_magassag):
5         """ Növeli (vagy csökkenti) ezt az objektumot a delta értékekkel. """
6         self.szelesseg += delta_szelesseg
7         self.magassag += delta_magassag
8
9     def mozgatas(self, dx, dy):
10        """ Elmozdítja ezt az objektumot a delta értékekkel. """
11        self.csucs.x += dx
12        self.csucs.y += dy
```

Az új metódusok kipróbálásához írjuk az alábbi utasításokat az osztályok létrehozása után, az osztályon kívülre:

```
1 r = Teglalap(Pont(10,5), 100, 50)
2 print(r)
3
4 r.noveles(25, -10)
5 print(r)
6
7 r.mozgatas(-10, 10)
8 print(r)
```

A kimenet az alábbi lesz:

```
((10, 5), 100, 50)
((10, 5), 125, 40)
((0, 15), 125, 40)
```

16.3. Azonosság

Az „azonos” szó jelentése tökéletesen világosnak tűnik egészen addig, ameddig el nem kezdünk egy kicsit gondolkodni, és rá nem jövünk, hogy több van mögötte, mint ahogy azt kezdetben hittük. Például, ha azt mondjuk, hogy „Az Indycarban azonos autóval indulnak a versenyzők.”, azt úgy értjük, hogy egyforma autóval lépnek pályára, de nem ugyanazzal az autóval. Ha viszont azt mondjuk, hogy „A két tigriskölyök azonos anyától származik.”, akkor arra gondolunk, hogy kölykök anyja egy és ugyanaz.

Az objektumoknál is hasonló kétértelműség áll fenn. Mit jelent például az, hogy két `Pont` azonos? Azt jelenti-e, hogy a tartalmazott adatok (a koordinátáik) egyformák, vagy azt, hogy ténylegesen ugyanarról az objektumról van szó?

A listák fejezetben, amikor a fedőnevektől beszéltünk, már láttuk az `is` operátort. Segítségével megvizsgálhatjuk, hogy két objektum ugyanarra az objektumra hivatkozik-e. Írjuk az osztálydefiníciók után az alábbi sorokat:

```
1 p1 = Pont(3, 4)
2 p2 = Pont(3, 4)
3 print(p1 is p2) # a kimenet False
4 p3 = p1
5 print(p1 is p3) # a kimenet True
```

Bár a `p1` és a `p2` azonos értékű koordinátákat tartalmaz, nem azonos objektumok. Ha viszont a `p1`-et hozzárendeljük a `p3`-hoz, akkor ez a két változó már ugyanannak az objektumnak a fedőneve.

Ezt a fajta egyenlőségvizsgálatot **referencia szerinti egyenlőségvizsgálatnak** nevezzük, mert nem az objektumok tartalmát, hanem a referenciákat hasonlítja össze.

Az objektumok tartalmának összehasonlításához, vagyis az **érték szerinti egyenlőségvizsgálathoz**, írhatunk egy függvényt. Legyen a neve: `azonos_koordinatak`:

```
1 def azonos_koordinatak(p1, p2):
2     return (p1.x == p2.x) and (p1.y == p2.y)
```

Ha most készítünk két különböző, de ugyanolyan adatairésszel rendelkező objektumot, akkor az `azonos_koordinatak` függvény segítségével kideríthetjük, hogy az objektumok által reprezentált pontok koordinátái megegyeznek-e.

```
1 p1 = Pont(3, 4)
2 p2 = Pont(3, 4)
3 egy_pont = azonos_koordinatak(p1, p2)
4 print(egy_pont)
```

A kimenet ezúttal `True` lesz.

Ha két változó ugyanarra az objektumra hivatkozik, akkor természetesen referencia szerint, és érték szerint is egyenlők.

Óvakodj az `==`-től

„Ha én használok egy szót – mondta Dingidungi megrovó hangsúllyal –, akkor az azt jelenti, amit én akarok, sem többet, sem kevesebbet!” (Lewis Carroll: *Alice Tükkörországban* (Révbíró Tamás fordításában))

A Python hatékony eszközt biztosít az osztályok tervezőinek annak meghatározására, hogy milyen jelentése legyen az olyan operátoroknak, mint például az `==` vagy a `<`. (Épp az előbb mutattuk be, hogyan kontrollálhatjuk az objektumaink sztringgé alakítását, tehát a kezdőlépéseket már megtettük!) Később több részletre is kitérünk majd. Az implementálók néha a referencia, néha az érték szerinti egyenlőség használata mellett döntenek, ahogy ezt az alábbi kis példa is mutatja:

```
1 p = Pont(4, 2)
2 s = Pont(4, 2)
3 print("Az == eredménye Pontokra alkalmazva:", p == s)
4 # Alapértelmezés szerint az == a Pont objektumoknál
5 # referencia szerinti egyenlőséget néz.
6
7 a = [2, 3]
8 b = [2, 3]
9 print("Az == eredménye listákra alkalmazva:", a == b)
10 # A listáknál viszont az érték szerinti egyenlőségvizsgálat
11 # az alapértelmezett.
```

A kimenet:

```
Az == eredménye Pontokra alkalmazva: False
Az == eredménye listákra alkalmazva: True
```

Levonhatjuk hát a következtetést, hogy még ha két külön lista (vagy rendezett n-es, stb.) objektumunk van is, eltérő memóriacímen, az == akkor is érték szerint fogja vizsgálni az egyenlőségüket, míg a Pontok esetében a referenciák alapján dönt.

16.4. Másolás

A fedőnevek megnehezítik a program olvasását, mert az egyik helyen történő módosítás nem várt hatást okozhat egy másik helyen. Nehéz követni az összes olyan változót, amelyek egy adott objektumra utalhatnak.

Az objektumok másolása gyakori megoldás a fedőnevek kiváltására. A `copy` modul tartalmaz egy `copy` nevű függvényt, amellyel bármilyen objektumot duplikálhatunk:

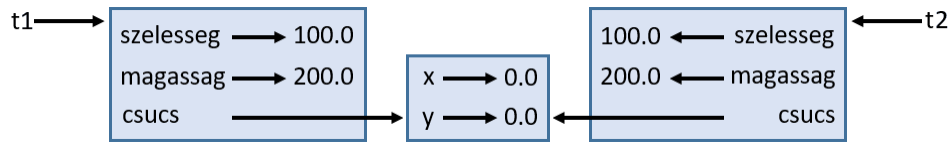
```
1 import copy          # Ezt a sort a szkript elejére szokás írni.
2 #...
3 # Ezek a sorok az osztály- és függvénydefiníciók alá kerüljenek.
4 p1 = Pont(3, 4)
5 p2 = copy.copy(p1)
6 azonos = p1 is p2
7 print(azonos)
8 azonos = azonos_koordinatak(p1, p2)
9 print(azonos)
```

A `copy` modul importálása után már a `copy` függvényt is használhatjuk egy új `Pont` objektum létrehozására. A `p1` és `p2` nem ugyanaz a `Pont` példány, de egyforma adatrésszel rendelkeznek, ezért az első kiírás `False` a második viszont `True` értéket jelenít meg.

Az egyszerű, beágyazott objektumot nem tartalmazó objektumok másolására, mint amilyen a `Pont` objektum is, megfelelő a `copy` függvény, amely **sekély másolást** valósít meg.

A Teglalap-szerű, más objektum referenciáját is tartalmazó objektumok másolására a `copy` nem ad teljesen jó eredményt. A Teglalap egy `Pont` objektum referenciáját tartalmazza, a sekély másolás ezt a referenciát másolja át, ezért a régi és az új Teglalap objektum is ugyanarra a `Pont` objektumra hivatkozik majd.

Ha készítünk egy `t1` nevű teglalatot a szokásos módon, majd készítünk egy másolatot `t2` néven a `copy` függvénnyel, akkor az alábbi állapotdiagrammal írhatjuk le az eredményt:



Szinte biztos, hogy nem ezt szeretnénk. Ha most meghívánk a `noveles` metódust az egyik `Teglalap` objektumra, akkor nem lenne hatással a másikra, ellenben, ha a `mozgatas-t` hívánk meg, az mindkettőre hatna! Az ilyen viselkedés zavaró, könnyen hibákhoz vezethet. A sekély másoló csak egy fedőnevet készített a `Pont`-ot reprezentáló csúcsához.

Szerencsére a `copy` modul tartalmaz egy `deepcopy` nevezetű függvényt is, amely nem csak az objektumot, de a beágyazott objektumokat is másolja. Nem meglepő módon, ezt a műveletet **mély másolásnak** hívják.

```
1 t2 = copy.deepcopy(t1)
```

A `t1` és a `t2` most két teljesen különálló objektum.

16.5. Szójegyzék

érték szerinti egyenlőség (deep equality) Két egyenlő érték, vagy két ugyanolyan értékű (állapotú) objektumra hivatkozó referencia *érték szerint egyenlő*.

mély másolás (deep copy) Egy objektum tartalmának másolása, beleértve a beágyazott objektumok másolását is, és azokba beágyazott objektumok másolását is, és így tovább. A `copy` modul `deepcopy` függvénye implementálja.

referencia szerinti egyenlőség (shallow equality) Két azonos referencia, vagy két, ugyanarra az objektumra hivatkozó referencia *referencia szerint egyenlő*.

sekély másolás (shallow copy) Egy objektum tartalmának másolása, beleértve a beágyazott objektumokra hivatkozó referenciák másolását is. A `copy` modul `copy` függvénye implementálja.

16.6. Feladatok

- Adj hozzá egy `terulet` metódust a `Teglalap` osztályhoz, amelyet ha meghívunk egy példányra, akkor annak területét adja vissza:

```
r = Teglalap(Pont(0, 0), 10, 5)
teszt(r.terulet() == 50)
```

- Írj egy `kerulet` metódust a `Teglalap` osztályon belül, amely segítségével meghatározhatjuk a `Teglalap` példányok kerületét:

```
r = Teglalap(Pont(0, 0), 10, 5)
teszt(r.kerulet() == 30)
```

- Írj egy `forditas` metódust a `Teglalap` osztályon belül, amellyel felcserélhetjük a `Teglalap` példányok magasságát és szélességét:

```
r = Teglalap(Pont(100, 50), 10, 5)
teszt(r.szelesseg == 10 and r.magassag == 5)
r.forditas()
teszt(r.szelesseg == 5 and r.magassag == 10)
```

4. Készíts egy új metódust a `Teglalap` osztályon belül annak ellenőrzésére, hogy egy `Pont` objektum a téglalapon belülre esik-e! Ennél a feladatnál feltételezzük, hogy a téglalap a (0, 0) koordinátán van, a szélessége 10, a magassága pedig 5. A téglalap felső határai *nyíltak*, tehát a téglalap a [0; 10) tartományt foglalja el az x tengelyen, ahol a 0 a tartomány része, de a 10 nem; y irányban pedig a [0; 5) tartományban áll. Szóval a (10, 2) pontot nem tartalmazza. Ezeken a teszteken át kell, hogy menjen:

```
r = Teglalap(Pont(0, 0), 10, 5)
teszt(r.tartalmazza_e(Pont(0, 0)))
teszt(r.tartalmazza_e(Pont(3, 3)))
teszt(not r.tartalmazza_e(Pont(3, 7)))
teszt(not r.tartalmazza_e(Pont(3, 5)))
teszt(r.tartalmazza_e(Pont(3, 4.99999)))
teszt(not r.tartalmazza_e(Pont(-3, -3)))
```

5. A játékokban gyakran vesszük körül a sprite-okat befoglaló téglalapokkal. (A sprite-ok olyan objektumok, amelyek mozoghatnak a játékban. Hamarosan látni fogjuk.) Utána már végezhetünk *ütközésfigyelést*, például bombák és űrhajók között, azt vizsgálva, hogy átfednek-e valahol a téglalapjaik.

Írj egy függvényt, mely meghatározza, hogy két téglalap összeér-e! *Segítség: Ez kemény dió! Gondolj át alaposan minden lehetőséget, mielőtt kódolni kezdesz.*

17. fejezet

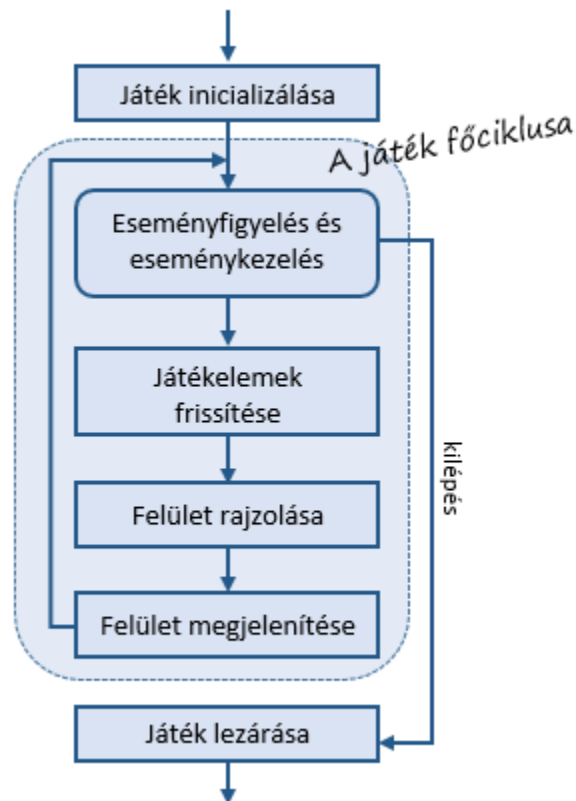
PyGame

A PyGame csomag nem része a standard Python disztribúciónak. Ha még nincs a gépeden (azaz az `import pygame` hibát okoz), akkor töltsd le a megfelelő verziót a <http://pygame.org/download.shtml> címről, és telepítsd fel. A fejezetben álló példák a PyGame 1.9.1-es verzióján alapulnak, ugyanis ez volt a legfrissebb verzió a könyv írásának pillanatában.

A PyGame csomaghoz rengeteg segédanyag, példaprogram és dokumentáció létezik, ami kiváló lehetőség ad arra, hogy beleásd magad a kódokba. A források megtalálása igényelhet egy kis keresgélést. Ha például Windowsos gépre telepítjük a PyGame csomagot, akkor a `C:\\Python32-36\\Lib\\site-packages\\pygame\\`-hez hasonló útvonalra kerülnek az anyagok, ott lehet megtalálni a *docs* és az *examples* mappákat is.

17.1. A játék főciklusa

Feltételezzük, hogy a játék felépítése minden esetben a következő mintát követi:



Az *inicializálás* során, minden egyes játéknál, készítünk egy ablakot, betöltünk és előkészítünk bizonyos tartalmakat, majd belépünk a **főciklusba**. A főciklus (más néven *szimulációs hurok*) 4 fő lépést hajt végre folyamatosan:

- Figyeli az eseményeket, azaz lekérdezi a rendszertől, hogy történt-e valamilyen esemény, és ha igen, akkor megfelelően reagál is rájuk.
- Frissíti azokat a belső adatszerkezeteket vagy objektumokat, amelyeknek változniuk kell.
- Kirajzolja a játék aktuális állapotát egy (nem látható) felületre.
- Megjeleníti a frissen kirajzolt felületet a képernyőn.

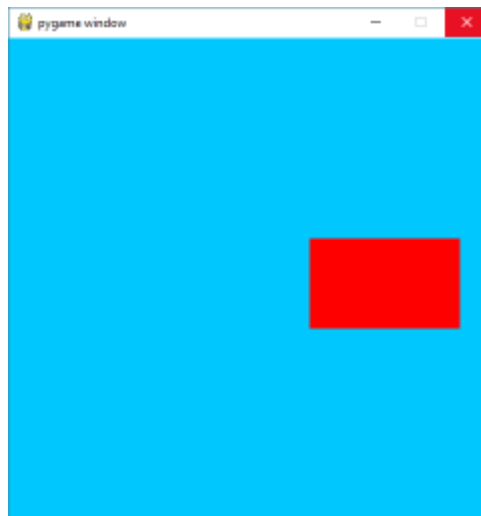
```
1 import pygame
2
3 def main():
4     """ A játék inicializálása és a főciklus futtatása. """
5     pygame.init() # Előkészíti a pygame modult a használatra.
6     felulet_meret = 480 # A felület kívánt fizikai mérete pixelben.
7
8     # Egy felület és a hozzá tartozó ablak elkészítése.
9     # A set_mode (szélesség, magasság) értékpárt vár.
10    fo_felulet = pygame.display.set_mode((felulet_meret, felulet_meret))
11
12    # Egy kis téglalap adatai, beleértve a színét is.
13    kis_teglalap = (300, 200, 150, 90)
14    valamilyen_szin = (255, 0, 0) # A szín a (piros, zöld, kék) színekből áll.
15    ↪elő.
16
17    while True:
18        esemeny = pygame.event.poll() # Események lekérdezése.
19        if esemeny.type == pygame.QUIT: # Az ablakbezárása gombra kattintottak?
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
19         break # ... a játékciklus elhagyása.
20
21     # A játék objektumaid, adatszerkezeteid frissítése ide jön.
22
23     # Minden egyes képkockánál teljesen előről kezdve rajzoljuk újra a képet:
24     # Először mindent háttérszínnel töltünk ki.
25     fo_felulet.fill((0, 200, 255))
26
27     # Majd átfestjük a főfelületen lévő kis téglalapot.
28     fo_felulet.fill(valamilyen_szin, kis_teglalap)
29
30     # A felület most már kész, megkérjük a pygame-et, hogy jelenítse meg!
31     pygame.display.flip()
32
33     pygame.quit() # Ha kilépünk a főciklusból, akkor az ablakot is bezárjuk.
34
35 main()
```

A program feldob egy ablakot, ami egészen addig marad ott, amíg be nem zárjuk:



A PyGame mindent egy téglalap alakú *felületre* rajzol rá. Miután az 5. sorban inicializáltuk a PyGame-et készítünk egy ablakot, amely a főfelületet tartalmazza. A főciklushoz tartozó sorok (16-31.) leglényegesebb elemei az alábbiak:

- Először (a 17. sorban) lekérdezzük a következő eseményt, hátha már rendelkezésünkre áll. Ezt a lépést mindig feltételes utasítások követik, amelyek azt hivatottak eldönteni, hogy történt-e számunkra érdekes esemény. A lekért eseményeket a Python feldolgozottként tekinti és megsemmisíti, tehát minden egyes eseményt csak egyetlen alkalommal kérhetünk le és használhatunk fel. A 18. sorban megvizsgáljuk, hogy az esemény megegyezik-e az előre definiált `pygame.QUIT` konstanssal. Ez az esemény akkor váltódik ki, ha a felhasználó a PyGame ablak bezárás gombjára kattint. Ha az esemény bekövetkezik, akkor kilépünk a ciklusból.
- Ha kilépünk a ciklusból, akkor a 33. sorban álló kód bezárja az ablakot, és a `main` függvény is befejezi a működését, visszatér a hívás helyére. A program ugyan folytatódhat más tevékenységekkel, akár újra is inicializálhatja a `pygame`-et, és létrehozhat egy újabb ablakot, de a legtöbbször a program is véget ér a `main`-ből való kilépés után.
- Különböző típusú események léteznek, mint például a billentyűleütés, egér mozgás, kattintás, joystick mozdulat, stb. Általában ezeket vizsgáljuk a 20. sor elé beszúrt új kódrészletekkel. Az általános elv: „Először kezeld az eseményeket, minden mással csak utána törődj!”
- A 21. sorhoz jöhet az objektumok, adatok frissítése. Ha például megváltoztatnánk a kirajzolandó téglalap

színét, pozícióját vagy méretét, akkor itt módosíthatjuk a `valamilyen_szin` és a `kis_teglalap` változók értékét.

- A játékirás modern módja (most, hogy ehhez már elég gyorsak a számítógépek és a videokártyák), hogy minden egyes cikluslépében teljesen újrarajzoljuk a felületet. Szóval a 25. sorban az egész felületet háttérszínnel töltjük ki. A felület `fill` metódusa két argumentumot vár: a kitöltés színét, és a kitöltendő téglalapot. Az utóbbi opcionális paraméter, ha nem adjuk meg, akkor az egész felületre vonatkozik a kitöltés.
- A 28. sorban a 2. téglalapot is kitöltjük `valamilyen_szin`-nel. A téglalap elhelyezését és méretét a `kis_teglalap` változóban tárolt rendezett 4-es adja meg: (`x`, `y`, szélesség, magasság).
- Fontos megérteni, hogy a PyGame felületeinél a bal felső sarok az origó (a turtle modullal szemben, ahol az origó az ablak közepe), így ha az ablak tetejéhez közelebb szeretnénk vinni a téglalapot, akkor csökkentenünk kell az `y` koordinátáját.
- Ha a grafikus megjelenítő ugyanakkor próbálja olvasni a memóriát, mint amikor a program írja azt, akkor megzavarhatják egymást, ami zajos, villogó képekhez vezethet. A PyGame úgy kerüli meg a problémát, hogy két puffert tart fenn a főfelülethez: a *háttér pufferbe* rajzol a program, míg az *előtér puffer* jelenik meg a képernyőn. Ha a program teljesen elkészült a hátsó pufferrel, akkor a két puffer szerepet cserél. A 25-28. sor tehát egészen addig nem változtatja meg a képernyőn látottakat, ameddig a `flip` metódussal meg nem cseréljük a két puffer tartalmát.

17.2. Képek és szövegek megjelenítése

Mivel egy képet szeretnénk rajzolni a főfelületre, ezért betöltünk egyet, mondjuk egy strandlabdát. A betöltött kép egy új felületre kerül. A főfelületnek van egy saját `blit` metódusa, amely átmásolja a strandlabda képét a saját felületére. A metódus hívásakor megadhatjuk azt is, hogy a főfelület mely részére kerüljön a strandlabda. A **képvitel** (`blit`) művelete gyakran használt a számítógépes grafikában, *a pixelek gyors, egyik területről a másikra történő másolását* valósítja meg.

Szóval betöltünk egy képet az inicializálás alatt, mielőtt még belépünk a főciklusba. Valahogy így:

```
1 labda = pygame.image.load("labda.png")
```

Majd a fenti kód 28. sora után beszurjuk ezt a kódrészletet, amely megjeleníti a képünket a (100, 120) pozícióban:

```
1 fo_felulet.blit(labda, (100, 120))
```

Egy szöveg megjelenítéséhez három dologra van szükségünk. Inicializálnunk kell egy `font` objektumot még a főciklusba való belépés előtt:

```
1 # Példányosítunk egy 16-os méretű, Courier típusú
2 # betűkészletet a szöveg rajzolásához.
3 font = pygame.font.SysFont("Courier", 16)
```

A 28. sor után a `font.render` metódusának hívásával létrehozunk egy új, a rajzolt szöveg pixeleit tartalmazó felületet, majd ahogyan azt a kép esetében is tettük, átmásoljuk a felületet a főfelületre. Figyeljük meg, hogy a `render` még két paramétert vár a szövegen kívül. A 2. paraméter mondja meg, hogy a szöveg éleit finoman elsimítsuk-e a rajzolás során (ezt az eljárást *anti-aliasing*-nak hívják), a harmadik paraméter pedig a szöveg színét adja meg. Az általunk használt (0, 0, 0) a fekete szín.

```
1 szoveg = betukeszlet.render("Hello, world!", True, (0,0,0))
2 fo_felulet.blit(szoveg, (10, 10))
```

Az új funkcionalitást a képkockák (vagyis a cikluslépések) számlálásával fogjuk bemutatni. Az eltelt időt is mérni fogjuk, hogy minden egyes képkockán megjeleníthessük majd a képkocka sorszámát és a képfreállítás sebességét

is. Utóbbit csak minden 500. képkocka után frissítjük: megnézzük mennyi idő telt el, és elvégezzük a szükséges számítást.

```
1 import pygame
2 import time
3
4 def main():
5     pygame.init() # Előkészíti a pygame modult a használatra.
6     fo_felulet = pygame.display.set_mode((480, 240))
7
8     # Betölti a rajzolandó képet. Cseréld ki a sajátodra.
9     # A PyGame képes kezelni a gif, jpg, png, stb. képformátumokat is.
10    labda=pygame.image.load("labda.png")
11
12    # Egy font objektum készítése a szöveg rendereléséhez.
13    betukeszlet=pygame.font.SysFont("Courier", 16)
14
15    kepcockak_szama = 0
16    fps = 0
17    t0 = time.clock()
18
19    while True:
20
21        # Billentyűzet, egér, joystick, stb. események figyelése.
22        esemeny = pygame.event.poll()
23        if esemeny.type == pygame.QUIT: # Rákattintottak az ablakbezárás_
24            ↪gombra? break # Kilépés a ciklusból
25
26        # A játéklógika egy másik darabja
27        kepcockak_szama += 1
28        if kepcockak_szama % 500 == 0:
29            t1 = time.clock()
30            fps = 500 / (t1 - t0)
31            t0 = t1
32
33        # Teljesen újrarajzoljuk a felületet, a háttérszínnel kezdjük.
34        fo_felulet.fill((0, 200, 255))
35
36        # Elhelyezünk egy piros téglalapot valahol a felületen.
37        fo_felulet.fill((255, 0, 0), (300, 100, 150, 90))
38
39        # Átmásoljuk a képünket a felület (x, y) pontjára.
40        fo_felulet.blit(labda, (100, 120))
41
42        # Egy új felületet készítünk, mely a szöveg képét tartalmazza.
43        szoveg = betukeszlet.render("Képkocka: {0}, sebesség: {1:.2f} fps"
44                                     .format(kepcockak_szama, fps), True, (0, 0, 0))
45        # Átmásoljuk a szöveg felületét a főfelületre.
46        fo_felulet.blit(szoveg, (10, 10))
47
48        # Most, hogy mindent megrajzoltunk, kirakjuk a képernyőre!
49        pygame.display.flip()
50
51    pygame.quit()
52
53    main()
```

A képfrissítés sebessége szinte már nevetséges. Sokkal gyorsabban jönnek a képkockák, mint ahogyan az emberi

szem képes feldolgozni. (A kereskedelmi videojátékoknál általában úgy tervezik a cselekményt, hogy 60 képkocka per másodperc (fps) legyen a képfrissítés sebessége. Természetesen nálunk is zuhanni fog a tempó, ha valami számításigényesebb dolgot hajtunk végre a főcíkluson belül.



17.3. Tábla rajzolása az N királynő problémához

Korábban már megoldottuk az N királynő problémát. 8x8-as tábla esetén, a [6, 4, 2, 0, 5, 7, 1, 3] lista írta le az egyik lehetséges megoldást. Most rajzoljunk egy olyan táblát a PyGame-mel, amely a királynőket is tartalmazza. A korábbi megoldásunkat használjuk tesztadatként.

A rajzolást végző kódot egy új modulban fogjuk elkészíteni. Legyen a fájl neve `kiralynok_rajzolasa.py`. Ha a tesztelésünkre (eseteinkre) már működik a program, akkor visszatérhetünk az N királynő probléma megoldóhoz. Az elkészült modul importálását követően minden egyes megoldás megtalálásakor meghívhatjuk majd az új táblarajzoló függvényünket.

A háttérrel, a táblát alkotó fekete és piros négyzetekkel kezdünk. Akár készíthetnénk is egy képet, amit csak be kellene tölteni és kirajzolni, de akkor minden egyes táblamérethez külön képre lenne szükségünk. Sokkal szórakoztatóbbnak érzékelik, ha mi magunk rajzoljuk meg a megfelelő méretű piros és fekete téglalapokat.

```
1 def tabla_rajzolas(kiralynok):
2     """ Egy sakktábla rajzolása a kiralynok listával adott királynőkkel_
   ↪együtt. """
3
4     pygame.init()
5     szinek = [(255,0,0), (0,0,0)] # A színek beállítása [piros, fekete].
6
7     n = len(kiralynok) # A tábla mérete: nxn.
8     felulet_meret = 480 # A felület javasolt fizikai mérete.
9     mezo_meret = felulet_meret // n # A négyzetek oldalhosszúsága.
10    felulet_meret = n * mezo_meret # Az n négyzet méretéhez igazítjuk a_
   ↪felületet.
11
12    # Elkészítjük a felületet (szélesség, magasság) és a hozzá tartozó_
   ↪ablakot.
13    felulet = pygame.display.set_mode((felulet_meret, felulet_meret))
```

A fenti kódban meghatározzuk a `mezo_meret` értékét, egy egész számot. Minden négyzet alakú mezőnek ez lesz

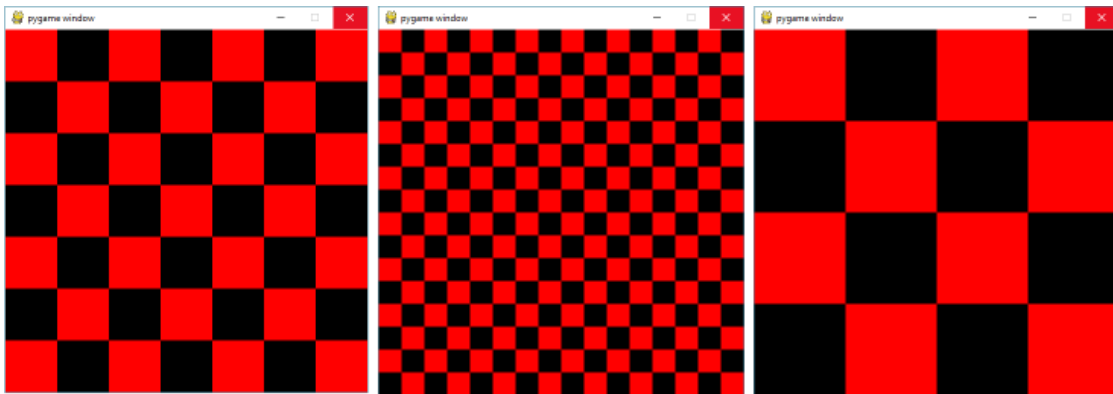
a mérete, hogy szépen kitöltésük majd a rendelkezésre álló ablakot. Tegyük fel, hogy a tábla méretét (a felületet) 480x480-ra állítjuk, és egy 8x8-as sakktáblát rajzolunk rá. Ekkor minden egyes négyzet oldalhosszúsága 60 egység lesz. Megfigyelhetjük azt is, hogy egy 7x7-es sakktábla nem passzol a 480-as értékhez. Csúnya keret kapnánk, mert a mezők nem töltik ki rendesen a táblát. Éppen ezért újraszámítjuk a felület méretét még az ablak elkészítése előtt, hogy az pontosan illeszkedjen majd a négyzetekhez.

Most rajzoljuk ki a négyzeteket a főcikluson belül. Egy beágyazott ciklusra is szükség lesz: a külső ciklus a sakktábla sorain, a belső a sakktábla oszlopain fut majd végig:

```
1 # Új háttér rajzolása (egy üres sakktábla).
2 for sor in range(n):          # Az összes sor megrajzolása.
3     szín_index = sor % 2      # A kezdőszín megváltoztatása minden sorban.
4     for oszlop in range(n):   # Az oszlopokat bejárva kirajzoljuk a mezőket.
5         mezo = (oszlop*mezo_meret, sor*mezo_meret, mezo_meret, mezo_meret)
6         felulet.fill(szinek[szín_index], mezo)
7         # A következő mező rajzolása előtt megváltoztatjuk a szín indexét.
8         szín_index = (szín_index + 1) % 2
```

Két lényeges ötlet található ebben a kódban. Az egyik az, hogy a felfestendő négyzeteket a `sor` és az `oszlop` ciklusváltozók alapján számoljuk. A ciklusváltozókat a négyzet méretével szorozva megkapjuk az egyes mezők pozícióit. A szélesség és a magasság természetesen minden mezőnél azonos. A `mezo` tehát minden egyes cikluslépésben az éppen kitöltendő téglalapot reprezentálja. A második ötlet, hogy minden egyes négyzet rajzolásánál színt váltunk, vagyis felváltva használjuk a színeket. A korábbi inicializálás során készítettünk egy 2 színt tartalmazó listát. Itt csak a `szín_index` értékét változtatjuk. Úgy állítjuk be az értékét (ami mindig 0 vagy 1), hogy minden egyes sor az előzőtől eltérő színnel induljon, és minden egyes kitöltés után megcseréljük a kitöltés színét.

A kód (az itt nem szereplő kódrészlettel együtt, amelyek megjelenítik a felületet a kijelzőn) különböző táblaméretek mellett is az alábbiakhoz hasonló, tetszetős háttereket eredményez:

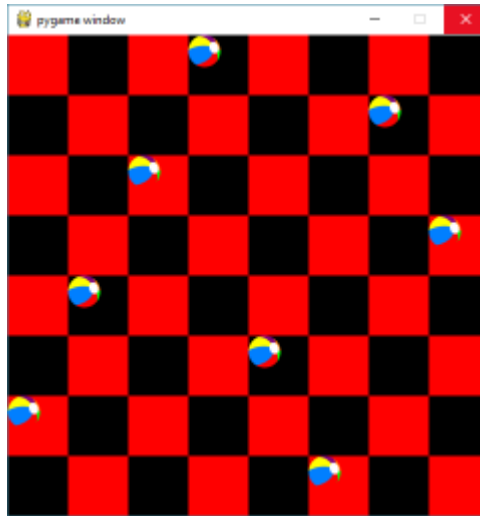


Most térjünk át a királynők rajzolására! Talán még emlékszünk rá, hogy a `[6, 4, 2, 0, 5, 7, 1, 3]` megoldás azt jelenti, hogy a 0. oszlop 6. sorába, az 1. oszlop 4. sorába, stb. akarjuk elhelyezni a királynőket. Szükségünk lesz tehát egy ciklusra, amely bejárja a királynők listáját:

```
1 for (oszlop, sor) in enumerate(kiralynok):
2     # egy királynő rajzolása az oszlop, sor pozícióra...
```

Ebben a fejezetben már dolgoztunk egy strandlabda képpel, jó lesz az a királynőkhöz is. A főciklus előtti inicializáló részben betöltjük a labda képet, (ahogy azt korábban is tettük) és hozzáadjuk az alábbi sort a ciklus törzséhez:

```
1 felulet.blit(labda, (oszlop * mezo_meret, sor * mezo_meret))
```



Közeledünk a célhoz, de még középre kellene igazítani a királynőket a mezőkben. A problémának abból ered, hogy a labdánál és a téglalapnál is a bal felső sarok a referencia pont. Ha középre akarjuk igazítani a labdát, egy kicsit el kell tolnunk x és y irányban is. (Mivel a labda kör a mező pedig négyzet alakú, az eltolás mindkét irányban azonos lesz, ezért elég az egyik irányhoz kiszámolni az értékét, és azt használni mindkét iránynál.)

A szükséges eltolás a négyzet és a kör közti méretkülönbség fele. Előre kiszámoljuk majd a játék inicializáló részében, miután a labdás képet betöltöttük és meghatároztuk a mezők méretét:

```
1 labda_eltolas = (mezo_meret - labda.get_width()) // 2
```

Most már kijavíthatjuk a labda rajzolásnál lévő kisebb hibát, és készen is leszünk:

```
1 felulet.blit(labda, (oszlop * mezo_meret + labda_eltolas, sor * q_sz + labda_
    ↳eltolas))
```

Érdekes még átgondolni, hogy mi történne akkor, ha a labda mérete meghaladná a négyzetét. A `labda_eltolas` ebben az esetben egy negatív érték lenne, de ilyenkor is a mező középre kerülne a labda, csak átlógna a mező határán. Talán teljesen el is takarná a négyzetet.

Itt a teljes program:

```
1 import pygame
2
3 def tabla_rajzolas(kiralynok):
4     """ Egy sakktábla rajzolása a kiralynok listával adott királynőekkel_
    ↳együtt. """
5
6     pygame.init()
7     szinek=[(255, 0, 0), (0, 0, 0)] # A színek beállítása [piros, fekete]
8
9     n=len(kiralynok) # A tábla mérete: nxn.
10    felulet_meret=480 # A felület javasolt fizikai mérete.
11    mezo_meret=felulet_meret // n # A négyzetek oldalhosszúsága.
12    felulet_meret=n * mezo_meret # Az n négyzet méretéhez igazítjuk a_
    ↳felületet.
13
14    # Elkészítjük a felületet (szélesség, magasság) és a hozzá tartozó_
    ↳ablakot.
15    felulet=pygame.display.set_mode((felulet_meret, felulet_meret))
16
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```

17 labda=pygame.image.load("labda.png")
18
19 # A labda négyzeten belüli középére igazításhoz szükséges eltolás.
20 # Ha a négyzet túl kicsi, az eltolás negatív lesz,
21 # de akkor is középre kerül a labda :-).
22 labda_eltolas=(mezo_meret - labda.get_width()) // 2
23
24 while True:
25
26     # Billentyűzet, egér, stb. események figyelése.
27     esemény=pygame.event.poll()
28     if esemény.type == pygame.QUIT:
29         break;
30
31     # Új háttér rajzolása (egy üres sakktábla).
32     for sor in range(n):          # Az összes sor megrajzolása.
33         szin_index = sor % 2      # A kezdőszín megváltoztatása minden_
↪ sorban.
34         for oszlop in range(n):   # Az oszlopokat bejárva kirajzoljuk a_
↪ mezőket.
35             mezo = (oszlop*mezo_meret, sor*mezo_meret, mezo_meret, mezo_
↪ meret)
36             felulet.fill(szinek[szin_index], mezo)
37             # A következő mező rajzolása előtt megváltoztatjuk a szín_
↪ indexét.
38             szin_index = (szin_index + 1) % 2
39
40     # A négyzetek rajzolása után a királynőket is megrajzoljuk.
41     for (oszlop, sor) in enumerate(kiralynok):
42         felulet.blit(labda, (oszlop * mezo_meret + labda_eltolas,
43                               sor * mezo_meret + labda_eltolas))
44
45     pygame.display.flip()
46
47     pygame.quit()
48
49 if __name__ == "__main__":
50     tabla_rajzolas([0, 5, 3, 1, 6, 4, 2]) # Az ablak méret tesztelése 7x7-
↪ es táblára.
51     tabla_rajzolas([6, 4, 2, 0, 5, 7, 1, 3])
52     tabla_rajzolas([9, 6, 0, 3, 10, 7, 2, 4, 12, 8, 11, 5, 1]) # 13 x 13
53     tabla_rajzolas([11, 4, 8, 12, 2, 7, 3, 15, 0, 14, 10, 6, 13, 1, 5, 9])

```

Van még itt egy említésre méltó pont. A 49. sorban lévő feltételes utasítás megvizsgálja, hogy az aktuálisan végrehajtás alatt álló program a `__main__`-e. Ennek a sornak a segítségével megkülönböztethető az az eset, amikor a modul főprogramként futtatjuk, és az, amikor más modulból importáljuk, és ott használjuk fel. Ha ezt a modult futtatjuk a Pythonnal, akkor az 50-53. sorban álló tesztesetek lefutnak majd. Ha egy másik programba importáljuk (például a korábbi N királynő megoldónkba), akkor a 49. sorban álló feltétel hamis lesz, és az 50-53. sorok nem hajtódnak végre.

A *Nyolc királynő probléma, második rész* fejezetben a főprogramunk így nézett ki:

```

1 def main():
2     ...
3     bd = list(range(8))      # A kezdeti permutáció generálása.
4     talalat_szama = 0
5     proba = 0
6     while talalat_szama < 10:

```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
7     rng.shuffle(bd)
8     proba += 1
9     if not van_utkozes(bd):
10        print("Megoldás: {0}, próbálkozás: {1}.".format(bd, proba))
11        proba = 0
12        talalat_szama += 1
13
14 main()
```

Két változtatásra van szükség. A program elején importáljuk be a modult, amin dolgoztunk. (Tegyük fel, hogy `kiralynok_rajzolas` a neve. Gondoskodjunk róla, hogy a két fájl ugyanabban a mappába legyen elmentve.) Majd a 11. sor után hívjuk meg a megoldás rajzoló függvényt az éppen megtalált megoldásra:

```
1 kiralynok_rajzolas.tabla_rajzolas(bd)
```

Ez egy nagyszerű kombinációhoz vezet, amely képes megkeresni az N királynő probléma megoldásait, és amikor talál egyet, akkor egy táblát feldobva meg is jeleníti azt.

17.4. Sprite-ok

A sprite-ok olyan saját állapottal és viselkedéssel rendelkező objektumok, melyek képesek mozogni a játékon belül. Sprite lenne például egy űrhajó, egy játékos, a lövedékek, vagy a bombák.

Az objektumorientált programozás (OOP) ideális az ilyen szituációkban: minden objektumnak saját tulajdonságai, saját belső állapota és metódusai vannak. Szórakozzunk egy kicsit az N királynő táblával. Ahelyett, hogy a királynőt a végleges helyére tennénk, csak ledobjuk a tábla tetejére, és hagyjuk, hogy onnan a helyére zuhanjon, esetleg pattogjon.

Először is minden királynőt egy objektummá kell alakítanunk. Tárolunk majd egy listát is, amely az összes aktív sprite-ot tartalmazza (ez tehát a királynők listája). A főciklusban két új dolgot fogunk elrendezni:

- Az események kezelése után, de még a rajzolás előtt, meghívjuk a `frissites` metódust az összes sprite-ra, ami minden sprite-nak megadja a lehetőséget arra, hogy módosítsa a belső állapotát, például cserélje a képét, helyzetét, forgassa el magát, vagy változtassa a méretét.
- Miután az összes sprite betöltötte magát, a főciklus elkezd a rajzolást. Először a háttérrel rajzoljuk ki, majd minden egyes sprite-ra sorra meghívjuk a `rajzolas` metódust, amivel a rajzolás munkáját – az OOP elveivel összhangban – az objektumokra hárítjuk át. Nem azt mondjuk, hogy „Hé, *rajzolas*, jelenítsd meg a királynőt!”, hanem azt, hogy „Hé, *királynő*, rajzold meg magad!”

Egy egyszerű objektummal kezdünk, se mozgás, se animáció. Ez csak egy vázlat, hogy lássuk hogyan illeszkednek össze a darabok:

```
1 class KiralynoSprite:
2
3     def __init__(self, kep, cel_pozicio):
4         """ Létrehoz és inicializál egy királynőt
5             a tábla cél pozíciójában.
6         """
7         self.kep = kep
8         self.cel_pozicio = cel_pozicio
9         self.pozicio = cel_pozicio
10
11     def frissites(self):
12         return # Most még semmit nem csinál.
13
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
14 def rajzolas(self, cel_felulet):
15     cel_felulet.blit(self.kep, self.pozicio)
```

Három attribútummal ruháztuk fel a sprite-ot: egy kép attribútummal, egy cél pozícióval, ahová rajzoljuk majd a képet, valamint egy aktuális pozícióval. Ha már mozgatni is fogjuk a sprite-ot, akkor az aktuális pozíció nem lehet mindig azonos azzal a hellyel, ahol szeretnénk, ha végül megjelenne a királynő. A fenti kód `frissites` metódusában jelenleg semmit nem csinálunk. A `rajzolas` metódus pedig csak rárajzolja a hívó által megadott felületre a képet, az aktuális pozíciónak megfelelően. (Ez a metódus valószínűleg a későbbiekben is ilyen egyszerű marad.)

Most, hogy az osztály definíció a helyére került, példányosíthatjuk a királynőket, betehetjük őket a sprite-ok listájába, és gondoskodhatunk arról, hogy a főciklus minden egyes képkockára meghívja a `frissites` és a `rajzolas` metódust. Így néz ki az új kódrészlet és az átalakított főciklus:

```
1  osszes_sprite = []      # Lista a játék összes sprite-ja részére.
2
3  # Egy-egy sprite készítése minden királynőhöz,
4  # és a spirte hozzáadása a listához.
5  for (oszlop, sor) in enumerate(kiralynok):
6      kiralyno = KiralynoSprite(labda,
7                                (oszlop*mezo_meret+labda_eltolas, sor*mezo_meret+labda_
8                                ↪eltolas))
9      osszes_sprite.append(kiralyno)
10
11 while True:
12     # Egér, billentyűzet, stb. események figyelése.
13     esemeny = pygame.event.poll()
14     if esemeny.type == pygame.QUIT:
15         break;
16
17     # Minden sprite-ot megkérünk, hogy frissítse magát.
18     for sprite in osszes_sprite:
19         sprite.frissites()
20
21     # Új háttér rajzolása (üres sakktábla)
22     # ... ugyanaz, mint korábban ...
23
24     # Minden sprite-ot megkérünk, hogy rajzolja ki magát.
25     for sprite in osszes_sprite:
26         sprite.rajzolas(felulet)
27
28     pygame.display.flip()
```

A program pontosan ugyanúgy működik, mint korábban, viszont az a pluszmunka, amivel minden királynőhöz objektumot készítettünk, megnyitja az utat néhány ambiciózus kiterjesztés felé.

Kezdjünk egy zuhanó királynő objektummal, amely minden pillanatban rendelkezik valamilyen irányú sebességgel. (Mi csak az *y* irányú elmozdulással foglalkozunk, de használd a képzeleted!) A `frissites` metódusban a királynő aktuális pozícióját a sebességvektornak megfelelően kívánjuk módosítani. Ha az *N* királynő táblánk az űrben lebegne, akkor a sebesség állandó volna, de hát itt a Földön gravitáció is van! A gravitáció minden időpillanatban megváltoztatja a sebességet, ezért egy olyan labdára van szükségünk, amely az esés közben egyre gyorsul. A gravitáció minden egyes királynőre nézve azonos, ezért nem a példányokban fogjuk tárolni az értékét, csak a modulon belül hozunk létre egy változót a számára. Egy másik változtatást is teszünk: minden királynőt a tábla tetejétől indítunk, hogy onnan zuhanhassanak a céljuk felé. A változtatások után az alábbi kódrészletet kapjuk:

```
1  gravitacio = 0.0001
2
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3 class KiralynoSprite:
4
5     def __init__(self, kep, cel_pozicio):
6         self.kep = kep
7         self.cel_pozicio = cel_pozicio
8         (x, y) = cel_pozicio
9         self.pozicio = (x, 0)          # Az oszlop tetejétől indul a labda,
10        self.y_sebesseg = 0            # 0 kezdősebességgel.
11
12    def frissites(self):
13        self.y_sebesseg += gravitacio    # A gravitáció módosítja a
14        ↪ sebességet.
15        (x, y) = self.pozicio
16        uj_y_poz = y + self.y_sebesseg  # A sebesség új pozícióba
17        self.pozicio = (x, uj_y_poz)    # mozgatja a labdát.
18
19    def rajzolas(self, cel_felulet):     # Ugyanaz, mint korábban.
20        cel_felulet.blit(self.kep, self.pozicio)
```

A változtatások eredményeként egy új sakktáblához jutunk, amelyen minden királynő a saját oszlopának tetejétől indul és egyre gyorsul, ameddig a pálya aljánál leesve el nem tűnik örökre. Jó kezdés: van mozgás!

A következő lépés annak megvalósítása, hogy a labda visszapattanjon, amikor eléri a cél pozíciót. Nagyon könnyen elérhető, hogy visszapattanjon valami, hiszen ha a sebesség előjelét megváltoztatjuk, akkor az ellenkező irányba fog ugyanazzal a sebességgel haladni. Ha a tábla teteje felé halad az objektum, akkor a gravitáció természetesen csökkenteni fogja a sebességét. (A gravitáció mindig lehúz!) Látni fogjuk, hogy a labda visszapattan oda, ahonnan indult, miközben a sebessége 0-ra csökken, majd újra elkezd zuhanni. Szóval lesz egy pattogó labdánk, amely soha nem áll le.

Az objektum úgy kerülhet életszerű módon nyugalmi helyzetbe, ha minden egyes ütközéskor veszít valamennyi energiát (talán a súrlódás hatására), ezért ahelyett, hogy egyszerűen az ellenkezőjére váltanánk a sebesség előjelét, egy törttel, mondjuk -0.65-tel szorozzuk meg azt. Ez azt jelenti, hogy a labda minden egyes visszapattanáskor csak az energiájának 65%-át őrzi majd meg, ezért rövid idő után abbamarad a pattogás, és megáll majd a labda a „földön”, ahogy az a való életben is történne.

Csak a `frissites` metóduson belül lesz változás, amely most már így néz ki:

```
1 def frissites(self):
2     self.y_sebesseg += gravitacio
3     (x, y) = self.pozicio
4     uj_y_poz = y + self.y_sebesseg
5     (cel_x, cel_y) = self.cel_pozicio    # A cél pozíció kicsomagolása.
6     tav = cel_y - uj_y_poz               # Milyen messze van a padló?
7
8     if tav < 0:                          # A padló alatt vagyunk?
9         self.y_sebesseg = -0.65 * self.y_sebesseg    # visszapattanás
10        uj_y_poz = cel_y + tav            # Visszatérés a padló fölé.
11
12    self.pozicio = (x, uj_y_poz)          # Az új pozíciónk beállítása.
```

He, he, he! Nem fogunk animált képernyőképet mutatni, szóval másold át a kódot a saját Python környezetbe és nézd meg magadnak!

17.5. Események

Mindeddig csak a `QUIT`, vagyis a kilépés eseményt kezeltük, holott a billentyűleütés, felengedés, egérmozgás, egérgombok lenyomása és felengedése eseményeket is tudjuk érzékelni. Nézz utána a PyGame dokumentációban, kattints az [event](#) linkre.

Amikor a program lekérdezi, illetve fogadja a PyGame eseményeket, az esemény típusa adja meg, hogy milyen másodlagos információk állnak rendelkezésre. Minden esemény objektum magával „hordoz” egy *szótárat* (később fogjunk érinteni ebben a könyvben), amelyben olyan *kulcsok* és *értékek* szerepelnek, amelyeknek van értelmük az adott típusú eseményekre nézve.

Például a `MOUSEMOTION` típusú eseményeknél a kurzor pozíciója és az egérgombok állapota érhető el az eseményhez tartozó szótárban. A `KEYDOWN` eseménynél pedig azt tudhatjuk meg a szótárból, hogy mely billentyű lett leütve, valamint azt, hogy nyomva van-e tartva valamelyik módosító billentyű (`Shift`, `Ctrl`, `Alt`, stb.). Akkor is érkezik esemény, ha a játék ablak aktívvá vagy inaktívvá válik, azaz megkapja vagy elveszíti a fókuszot.

Ha egyetlen esemény sem vár feldolgozásra, akkor egy `NOEVENT` típusú esemény érkezik vissza. Az eseményeket ki lehet írni, így bátran kísérletezhetünk, játszadózhatunk velük. Szűrjük be a következő sorokat a játék főciklusába közvetlenül az esemény lekérdezése alá, felettebb informatív lesz:

```
1 if esemeny.type != pygame.NOEVENT:    # Csak akkor írjuk ki, ha érdekes!
2     print(esemeny)
```

Ha a helyére került a kódrészlet, akkor üsd le a `Space`, majd az `Escape` billentyűt, és figyeld meg, milyen eseményeket kapsz. Kattints a három egérgombbal, mozgasd az egeret az ablak felett. (Az utóbbi rengeteg eseményt generál, ezért érdemes lehet kihagyni ezeket az eseményeket a kiíratásnál.) Az alábbihoz hasonló kimenetet fogsz kapni:

```
<Event(17-VideoExpose {})>
<Event(1-ActiveEvent {'state': 1, 'gain': 0})>
<Event(2-KeyDown {'scancode': 57, 'key': 32, 'unicode': ' ', 'mod': 0})>
<Event(3-KeyUp {'scancode': 57, 'key': 32, 'mod': 0})>
<Event(2-KeyDown {'scancode': 1, 'key': 27, 'unicode': '\x1b', 'mod': 0})>
<Event(3-KeyUp {'scancode': 1, 'key': 27, 'mod': 0})>
...
<Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (323, 194), 'rel': (-3, -1)})>
<Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (322, 193), 'rel': (-1, -1)})>
<Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (321, 192), 'rel': (-1, -1)})>
<Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (319, 192), 'rel': (-2, 0)})>
<Event(5-MouseButtonDown {'button': 1, 'pos': (319, 192)})>
<Event(6-MouseButtonUp {'button': 1, 'pos': (319, 192)})>
<Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (319, 191), 'rel': (0, -1)})>
<Event(5-MouseButtonDown {'button': 2, 'pos': (319, 191)})>
<Event(5-MouseButtonDown {'button': 5, 'pos': (319, 191)})>
<Event(6-MouseButtonUp {'button': 5, 'pos': (319, 191)})>
<Event(6-MouseButtonUp {'button': 2, 'pos': (319, 191)})>
<Event(5-MouseButtonDown {'button': 3, 'pos': (319, 191)})>
<Event(6-MouseButtonUp {'button': 3, 'pos': (319, 191)})>
...
<Event(1-ActiveEvent {'state': 1, 'gain': 0})>
<Event(12-Quit {})>
```

Szűrjük most be ezeket a változtatásokat a főciklus elejéhez:

```
1 while True:
2
3     # Egér, billentyűzet, stb. események figyelése.
4     esemeny = pygame.event.poll()
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5     if esemeny.type == pygame.QUIT:
6         break;
7     if esemeny.type == pygame.KEYDOWN:
8         key = esemeny.dict["key"]
9         if key == 27:                                # Escape billentyűre ...
10            break                                     #   lépünk ki a játékból.
11         if key == ord("r"):
12             szinek[0] = (255, 0, 0)                  # Váltás piros-feketére.
13         elif key == ord("g"):
14             szinek[0] = (0, 255, 0)                  # Váltás zöld-feketére.
15         elif key == ord("b"):
16             szinek[0] = (0, 0, 255)                  # Váltás kék-feketére.
17
18     if esemeny.type == pygame.MOUSEBUTTONDOWN: # Egérgomb lenyomva?
19         kattintas_helye = esemeny.dict["pos"]        # Lekérjük a koordinátákat,
20         print(kattintas_helye)                       # és kiírjuk őket.
```

A 7-16. sorok dolgozzák fel a KEYDOWN eseményt. Ha billentyűleütés történt, akkor megvizsgáljuk, hogy mely billentyű lett leütve, és attól függően csinálunk valamit. A kód beépítését követően már az eddigittől eltérő módon, az Esc billentyű lenyomásával is ki tudunk lépni a játékból, és a rajzolandó tábla színét is változtathatjuk különböző billentyűk leütésével.

Végül, a 20. sorban az egér gombjának lenyomására reagálunk, nem túl kifinomult módon.

Az alfejezet utolsó feladatáért az egér kattintáshoz fogunk egy jobb eseménykezelőt készíteni. Ellenőrizni fogjuk, hogy a felhasználó valamelyik sprite-ra kattintott-e. Ha a kurzor alatt egy sprite van a kattintás pillanatában, akkor átadjuk majd a kattintás eseményt a sprite-nak, hogy az reagáljon rá megfelelő módon.

Egy olyan kódreszlettel kezdünk, ami meghatározza, hogy melyik sprite áll a kattintás pozíciójában. Az is lehet, hogy semelyik! Adjunk egy `tartalmazza_a_pontot` metódust az osztályhoz, mely True értékkel tér vissza, ha a paraméterként kapott pont a sprite-hoz tartozó téglalapon belül van:

```
1     def tartalmazza_a_pontot(self, pt):
2         """ True-t ad vissza, ha a sprite téglalapja tartalmazza a pt pontot. """
3         ↪
4         (sajat_x, sajat_y) = self.pozicio
5         sajat_szelesseg = self.kep.get_width()
6         sajat_magassag = self.kep.get_height()
7         (x, y) = pt
8         return ( x >= sajat_x and x < sajat_x + sajat_szelesseg and
9                 y >= sajat_y and y < sajat_y + sajat_magassag)
```

Ha a főcikluson belül egéreseményt érzékelünk, akkor megnézzük, hogy melyik királynőre kell bízni a válaszadást (ha egyáltalán rá kell bízni valamelyikre):

```
1     if esemeny.type == pygame.MOUSEBUTTONDOWN:
2         kattintas_helye = esemeny.dict["pos"]
3         for sprite in osszes_sprite:
4             if sprite.tartalmazza_a_pontot(kattintas_helye):
5                 sprite.kattintas_kezelo()
6                 break
```

Utolsó lépésként egy új metódust kell írunk `KiralynoSprite` osztályon belül `kattintas_kezelo` néven. Ha egy sprite-on történt a kattintás, akkor hozzáadunk majd valamennyi felfele irányuló sebességet, azaz visszaütjük a levegőbe.

```
1 def kattintas_kezelo(self):  
2     self.y_sebesseg += -0.3 # Felfele ütjük.
```

Ezekkel a változtatásokkal már egy játszható játékunk van! Próbáld ki, hogy sikerül-e az összes labdát mozgásban tartanod. Ne engedd, hogy egy is megálljon!

17.6. Egy integetős animáció

Nagyon sok játék tartalmaz animált sprite-okat: guggolnak, ugranak, lőnek. Hogyan csinálják?

Tekintsük ezt a 10 képből álló sorozatot: ha elég gyorsan játsszuk le a képeket, akkor Duke integetni fog nekünk. (Duke egy barátságos látogató Javaland királyságából.)



A kisebb *mintákat* tartalmazó, animációs célra készített összetett képeket **sprite lapoknak** nevezzük. Töltsd le ezt a sprite lapot. Kattints a jobb oldali egérgombbal a böngészőbeli képre, és mentsd el a munkakönyvtárdba `duke_spritesheet.png` néven.

A sprite lap nagyon gondosan tervezett: mind a 10 minta pontosan 50 pixelnyire van egymástól. Tegyük fel, hogy a (nullától számozott) 4. mintát akarjuk kirajzolni, ekkor a sprite lap 200-as x koordinátától kezdődő, 50 pixel szélességű téglalap rajzolására van csak szükségünk. Itt látható kiemelve a rajzolendő minta:



A `blit` metódus, amit a pixelek egyik felületről a másikra való átvitelére használunk, azt is megengedi, hogy csak egy téglalap alakú részt másoljunk át. A nagy ötlet tehát az, hogy amikor Duke-ot rajzoljuk, akkor soha nem fogjuk az egész képet másolni. Átadunk majd egy plusz információt, egy téglalap argumentumot, amely meghatározza a sprite másolandó részét.

Új kódrészletet fogunk adni a már létező N királynő rajzoló játékunkhoz, ugyanis szeretnénk elhelyezni Duke néhány példányát valahol a sakktáblán. Ha a felhasználó rákattint valamelyikre, akkor egy animációs ciklus segítségével elérjük, hogy Duke visszaintegessen neki.

A kezdés előtt még egy másik változtatásra is sort kell kerítenünk. A főciklusunk mindaddig nagy és kiszámíthatatlan képfrissítési sebességgel futott, ezért a gravitációnál, a labda visszapattanásánál és ütésénél is próba-hiba alapon választottunk egy-egy *bűvös számot*. Ha több sprite animálásába kezdünk, akkor rá kell vennünk a főciklust, hogy egy ismert, előre rögzített képfrissítési sebességgel működjön, ami tervezhetőbbé teszi az animálást.

A PyGame eszközeivel két sorban megtehetjük mindezt. A játék inicializálásánál példányosítottunk egy `Clock` objektumot:

```
1 ora = pygame.time.Clock()
```

majd a főciklus legvégén meghívjuk ennek egy metódusát, mely az általunk megadott érték szerint szabályozza a képfrissítés sebességét. Tervezzük például a játékot és az animációt 60 képkockával másodpercenként, ehhez az alábbi sort kell a ciklus aljához adnunk:


```
1 # Pazarol egy kis időt, hogy a képfrissítés sebessége 60 fps legyen.  
2 ora.tick(60)
```

Láthatni fogjuk, hogy vissza kell térni a gravitáció és a labda ütés sebességének beállításához, hogy ehhez, a jóval lassabb tempóhoz igazítsuk az értékeket. Amikor az animációt úgy tervezzük, hogy az csak egy rögzített képfrissítési sebesség mellett működik jól, akkor *beégetett* animációról beszélünk. Ebben az esetben 60 képkocka per másodperchez igazítjuk az animációt.

A már meglévő, a királynős táblához készített keretrendszerhez való illeszkedés érdekében készíteni fogunk egy DukeSprite osztályt, amelynek pont olyan metódusai lesznek, mint a KiralynoSprite osztálynak. Utána hozzáfűzhetünk egy vagy több Duke példányt az `osszes_sprite` listához, a már létező főciklusunk pedig meghívja majd a Duke példány(ok) metódusait. Kezdjük az új osztály vázlatával:

```
1 class DukeSprite:  
2  
3     def __init__(self, kep, cel_pozicio):  
4         self.kep = kep  
5         self.pozicio = cel_pozicio  
6  
7     def frissites(self):  
8         return  
9  
10    def rajzolas(self, cel_felulet):  
11        return  
12  
13    def kattintas_kezelo(self):  
14        return  
15  
16    def tartalmazza_a_pontot(self, pt):  
17        # Használd a KiralynoSprite-ban lévő kódot.  
18        return
```

Egyetlen ponton kell módosítanunk a már létező játékot, az inicializálásnál. Betöltjük az új sprite lapot, majd példányosítunk néhány Duke objektumot a sakktábla kívánt pozícióira. Az alábbi kód tehát a főciklusba való belépés elé kerül:

```
1 # A sprite lap betöltése.  
2 duke_sprite_lap = pygame.image.load("duke_spritesheet.png")  
3  
4 # Két Duke példány létrehozása és elhelyezése a táblán.  
5 duke1 = DukeSprite(duke_sprite_lap, (mezo_meret*2, 0))  
6 duke2 = DukeSprite(duke_sprite_lap, (mezo_meret*5, mezo_meret))  
7  
8 # A példányok hozzáadása a főciklus által kezelt sprite listához.  
9 osszes_sprite.append(duke1)  
10 osszes_sprite.append(duke2)
```

A főciklus minden példány esetén megvizsgálja, hogy rákattintott-e a felhasználó, és szükség esetén meghívja az adott példány kattintás kezelő metódusát. Ezenkívül minden példányra meghívja a `frissites` és a `rajzolas` metódusokat. Most már csak a DukeSprite osztály metódusait kell módosítanunk.

Kezdjük a minták rajzolásával. Felveszünk egy új `aktualis_minta_sorszam` attribútumot az osztályba, mely a rajzolandó minta sorszámát, vagyis 0-9 közti értékeket fog tárolni. A `rajzolas` metódus feladata a mintát tartozó téglalap meghatározása, és a sprite lap ezen részének másolása:

```
1 def rajzolas(self, cel_felulet):  
2     minta_teglalap = (self.aktualis_minta_sorszam * 50, 0,
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3         50, self.kep.get_height())
4     cel_felulet.blit(self.kep, self.pozicio, minta_teglalap)
```

Most koncentráljunk az animáció munkára bírására, amihez rendeznünk kell a `frissites` metódus logikáját. Az animáció közben sűrűn változtatjuk majd az `aktualis_minta_sorszam`-ot, és arról is dönteni fogunk, hogy mikor juttassuk vissza Duke-ot a nyugalmi helyzetbe az animáció leállításával. Fontos megjegyezni, hogy a főciklus képfrissítési sebessége – esetünkben 60 fps – nem azonos az **animáció sebességével**, amely azt adja meg, hogy milyen gyakran változtatjuk Duke animációs mintáját. Úgy tervezzük, hogy Duke integetős animációs ciklusa 1 másodpercig tartson, vagyis a `frissites` metódus 60 hívása alatt 10 animációs mintát szeretnénk lejátszani. (Így zajlik az animáció beégetése.) A megvalósításhoz szükség van egy másik animációs képkocka-számlálóra is az osztályon belül, amely 0 értéket vesz fel, amikor nem animálunk. A `frissites` minden hívása eggyel növeli majd a számlálót. Az 59-es érték elérését követően a számlálót 0-ról újra indítjuk. A számláló értékét 6-tal osztva megkapjuk az `aktualis_minta_sorszam` értékét, vagyis a megjelenítendő minta sorszámát.

```
1     def frissites(self):
2         if self.anim_kepkockak_szama > 0:
3             self.anim_kepkockak_szama = (self.anim_kepkockak_szama + 1) % 60
4             self.aktualis_minta_sorszam = self.anim_kepkockak_szama // 6
```

Figyeljük meg, hogy amikor az `anim_kepkockak_szama` értéke nulla, azaz Duke pihen, semmi nem történik a metóduson belül. Ellenben, ha elindítjuk a számlálót, akkor az elszámol egészen 59-ig, mielőtt újra 0 értéket venne fel. Vegyük észre azt is, hogy az `aktualis_minta_sorszam` értéke mindig a 0-9 egész számok valamelyike, hiszen az `anim_kepkockak_szama` mindig a [0;59] tartományba esik. Pont úgy, ahogy akartuk!

Hogyan váltjuk ki, hogyan indítjuk el az animációt? Egy kattintással.

```
1     def kattintas_kezelo(self):
2         if self.anim_kepkockak_szama == 0:
3             self.anim_kepkockak_szama = 5
```

A kódban két figyelemre méltó pont van. Csak akkor indítjuk el az animációt, ha Duke nyugalmi helyzetben van, ha Duke éppen integet, amikor rákattintanak, akkor figyelmen kívül hagyjuk a kattintást. Az animáció indításakor 5-re állítjuk a számlálót, ami azt jelenti, hogy már a `frissites` következő hívásánál 6 lesz az értéke és megváltozik a kép. Ha 1-re állítanánk a számlálót, akkor még 5 hívást kellene kivárnunk, mire végre történne valami. Ez nem túl nagy idő, de pont elég ahhoz, hogy lassúnak érezzük a reakciót.

A legutolsó teendő, hogy a két új attribútumot inicializáljuk az osztály példányosító metódusában. Itt az egész osztály kódja:

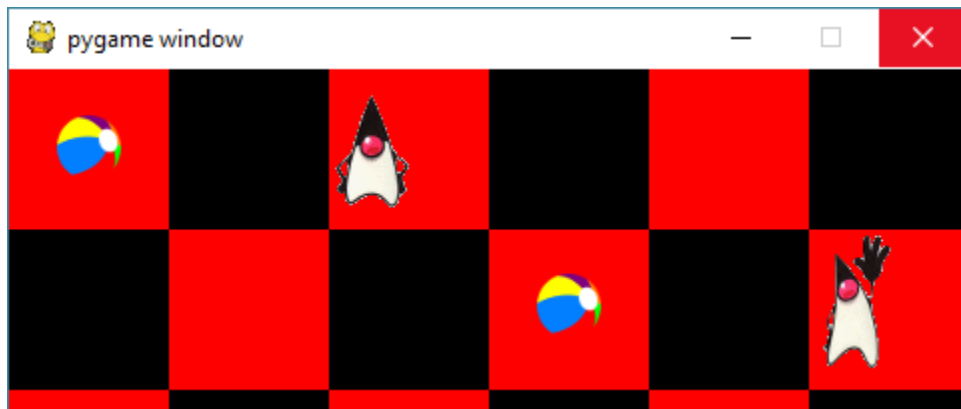
```
1     class DukeSprite:
2
3         def __init__(self, kep, cel_pozicio):
4             self.kep = kep
5             self.pozicio = cel_pozicio
6             self.anim_kepkockak_szama = 0
7             self.aktualis_minta_sorszam = 0
8
9         def frissites(self):
10             if self.anim_kepkockak_szama > 0:
11                 self.anim_kepkockak_szama = (self.anim_kepkockak_szama + 1) % 60
12                 self.aktualis_minta_sorszam = self.anim_kepkockak_szama // 6
13
14         def rajzolas(self, cel_felulet):
15             minta_teglalap = (self.aktualis_minta_sorszam * 50, 0,
16                               50, self.kep.get_height())
17             cel_felulet.blit(self.kep, self.pozicio, minta_teglalap)
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
18
19     def tartalmazza_a_pontot(self, pt):
20         """ True-t ad vissza, ha a sprite téglalapja tartalmazza a pt pontot.
21         """
22         (sajat_x, sajat_y)=self.pozicio
23         sajat_szelesseg=self.kep.get_width()
24         sajat_magassag=self.kep.get_height()
25         (x, y)=pt
26         return (x >= sajat_x and x < sajat_x + sajat_szelesseg
27                 and y >= sajat_y and y < sajat_y + sajat_magassag)
28
29     def kattintas_kezelo(self):
30         if self.anim_kepkockak_szama == 0:
31             self.anim_kepkockak_szama = 5
```

Most már van két Duke objektumunk is a sakktáblán, és ha rákattintunk valamelyikre, akkor az a példány integetni fog.



17.7. Alienek – esettanulmány

Keresd meg a PyGame csomaghoz tartozó példajátékokat (Windows operációs rendszer alatt valami ilyesmi az útvonal: C:\Python36-32\Lib\site-packages\pygame\examples), és játssz az Aliens játékkal. Utána töltsd be a kódot egy szövegszerkesztőbe, vagy a Python fejlesztő környezetbe, ahol látszik a sorok számozása.

Ez a játék sok olyan dolgot tartalmaz, amelyek bonyolultabbak az általunk végzett tevékenységeknél, és a PyGame keretrendszerét is jobban kihasználja a játéklógika megvalósításához. Itt egy lista a figyelemre méltó pontokról:

- A képfrissítés sebessége szándékosan korlátozva van a főciklus végén, a 313. sorban. Az érték megváltoztatásával lelassíthatjuk a játékot, vagy akár játszhatatlanul gyorsá is tehetjük!
- Többféle sprite is van: robbanások, lövések, bombák, földönkívüliek és a játékos. Néhányhoz több kép is tartozik, ilyenkor a képek cserélgetésével valósul meg az animáció, például a 115. sor hatására megváltoznak az idegenek űrhajóinak fényei.
- A különböző típusú objektumok különböző sprite csoportokba vannak szervezve, melyek kezelésében a PyGame segít. Ez lehetőséget ad arra, hogy a program ütközésvizsgálatot hajtson végre, mondjuk a játékos által kilőtt töltények listája és a támadó űrhajók listája között. A PyGame keményen dolgozik helyettünk.
- Az Aliens játékban – szemben a mi játékunkkal – az objektumok élettartama korlátozott, meg kell ölni azokat. A lövésnél például egy Shot objektum jön létre. Ha ütközés (vagyis robbanás) nélkül eléri a képernyő tetejét,

akkor ki kell venni a játékból. Ezt a 146-147. sorok intézik. Hasonlóan, ha egy bomba megközelíti a földet (161. sor), akkor egy Explosion sprite példány keletkezik, és a bomba megsemmisíti magát.

- A játékot véletlenszerű időzítések teszik szórakoztatóbbá: mikor induljon el a következő idegen, mikor dobja le a következő bombát, stb.
- A játék hangokat is hallat: egy nem túl pihentető, folyamatosan ismétlődő zenét, valamint a lövések és a robbanások hangjait.

17.8. Mérlegelés

Az objektumorientált programozás jó szervezési eszköz a szoftverek készítésénél. A fejezetben található példákban elkezdjük kihasználni (és remélhetőleg értékelni is) az OOP előnyeit. Van N királynőnk, mindegyiknek van saját állapota, a saját szintjükre esnek le, visszapattannak, visszaüthetők, stb. Lehet, hogy az objektumok szervezési ereje nélkül is meg tudtuk volna oldani mindezt. Tárolhattuk volna a királynők sebességét egy listában, a cél pozícióikat egy másik listában, és így tovább. De a kódunk valószínűleg sokkal bonyolultabb, rondább és rosszabb lett volna!

17.9. Szójegyzék

animáció sebessége (animation rate) Az a sebesség, amellyel az egymást követő mintákat lejátszunk a mozgás illúzióját keltve. A fejezetben szereplő példában 1 másodperc alatt 10 Duke mintát játszottunk le. Nem keverendő a képfriállítás sebességével.

beégetett animáció (baked animation) Olyan animáció, melyet úgy terveznek, hogy egy előre meghatározott képfriállítási sebesség mellett jól nézzen ki. Csökkentheti a játék futása közben elvégzendő számítások mennyiségét. A felső kategóriás kereskedelmi játékok általában beégetett animációkat tartalmaznak.

eseményfigyelés (poll) Annak figyelése, hogy történt-e billentyűleütés, egér mozgás, vagy más hasonló esemény. Általában a játék főciklusa kérdezi le az eseményeket, hogy kiderítse milyen esemény történt. Ez különbözik az esemény-vezérelt programoktól, mint amik például az Események fejezetben láthatók. Azoknál a kattintás vagy a billentyűleütés esemény kiváltja a programban lévő eseménykezelő meghívását, ez viszont a háttérben történik.

felület (surface) A Turtle modul *vászon* kifejezésének PyGame-beli megfelelője. A felület alakzatok és képek megjelenítéséhez használt képpontokból álló téglalap.

főciklus (game loop) A játéklógikát vezérlő ciklus. Általában figyeli az eseményeket, frissíti a játékbeli objektumokat, majd mindent kirajzoltat, és kiteszi az újonnan készített képkockát a megjelenítőre. Szimulációs hurok néven is találkozhatasz vele.

képatvitel (blitting) A számítógépes grafika világából származó kifejezés. Egy kép vagy egy felület, vagy ezek egy téglalap alakú részének egy másik képre, vagy felületre történő gyors átmásolását jelenti.

képfriállítás sebessége (frame rate) Az a sebesség, amellyel a főciklus frissíti és megjeleníti a kimenetet.

képpont (pixel) Egy képet felépítő elemek (pontok) egyike.

sprite Egy játék aktív, önálló állapottal, pozícióval és viselkedéssel rendelkező szereplője, vagy eleme.

17.10. Feladatok

1. Szórakozz a Pythonnal és a PyGame-mel!

2. Szándékosan hagyunk egy hibát a Duke animációban. Duke akkor is integetni fog, ha valamelyik tőle jobbra lévő mezőre kattintasz. Miért? Találj egy 1 soros javítást a hibára!
3. Keress egy kártyalapokat tartalmazó képgyűjteményt a kedvenc keresőmotorod segítségével (ha angol nyelven keresel, akkor több találat várható: „sprite sheet playing cards”). Készíts egy $[0, 1, \dots, 51]$ listát a pakliban lévő 52 kártya reprezentálására! Keverd össze a kártyákat és vedd fel az első öt lapot a kezedbe, mint a pókerben az osztásnál! Jelenítsd meg a kezedben lévő lapokat!
4. Az Aliens játék az űrben játszódik, gravitáció nélküli térben: a lövések elszállnak örökre, a bombák nem gyorsulnak zuhanás közben. Adj egy kis gravitációt a játékhoz! Döntsd el, hogy a saját lövéseid is visszahullhatnak-e rád, megölhetnek-e!
5. Úgy tűnik, hogy azok a bosszantó földönkívüliek keresztül tudnak menni egymáson! Változtasd meg úgy a játékot, hogy ütközhessenek, és az ütközéskor elpusztítsák egymást egy hatalmas robbanás kíséretében!

18. fejezet

Rekurzió

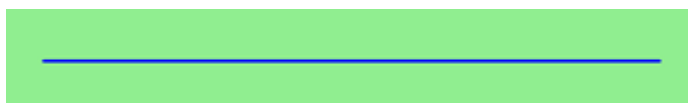
A **rekurzió** azt jelenti, hogy „önmagával definiálunk valamit”, általában kisebb mértékben, talán többször is a cél eléréséért. Például azt mondhatjuk, hogy „az ember olyan valaki, akinek az édesanya is ember” vagy „egy könyvtár egy olyan struktúra, amely fájlokat és kisebb könyvtárakat tartalmaz” vagy „egy családfa egy olyan párral kezdődik, akiknek vannak gyerekeik, és mindegyik gyerek rendelkezik saját alcsaládfával.”

A programozási nyelvek általában támogatják a rekurziót, ami azt jelenti, hogy egy probléma megoldása érdekében a függvények *önmagukat hívhatják* kisebb alproblémák megoldására.

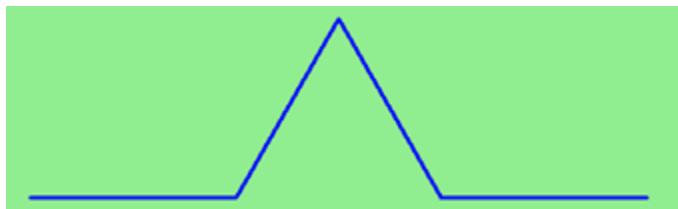
18.1. Fraktálok rajzolása

A célunk egy olyan **fraktál** rajzolása, mely szintén egy *önhasználó* szerkezettel rendelkezik, melyet önmagával tudunk definiálni.

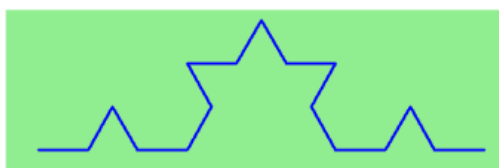
Kezdjük a híres Koch fraktál áttekintésével. A 0. rendű Koch fraktál egy adott méretű egyenes.



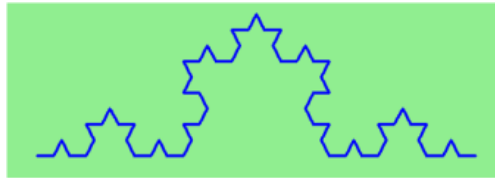
Az 1. rendű Koch fraktál: ahelyett, hogy csak egy vonalat rajzolnánk, helyette rajzolunk 4 kisebb szegmenst, mint az itt bemutatott mintában:



Mi fog történni, ha megismételjük ezt a Koch mintát ismét, minden 1-es szintű szegmensre? Megkapnánk a 2-od rendű Koch fraktált:



A minta ismétlésével megkapjuk a 3-ad rendű Koch fraktált:



Most gondolkozzunk egy kicsit másképp. A 3-ad rendű Koch fraktál rajzolásához, egyszerűen csak négy másodrendű Koch fraktált rajzolunk. De ezek mindegyikéhez szükség van négy 1. rendű Koch fraktálra, és mindegyikhez még négy 0. rendű fraktálra. Végülis az egyetlen rajz, amelyre sor kerül, az a 0. rendű. Ez egyszerűen kódolható Pythonban:

```
1 def koch(t, rend, meret):
2     """
3     Készíts egy t teknőst és rajzolj egy megadott 'rendű' és 'méretű' Koch
4     ↪fraktált.
5     Hagyjuk a teknőst ugyanabban az irányban.
6     """
7     if rend == 0:          # Alapesetben csak egy egyenes
8         t.forward(meret)
9     else:
10        koch(t, rend-1, meret/3)    # Menj az út 1/3-ig
11        t.left(60)
12        koch(t, rend-1, meret/3)
13        t.right(120)
14        koch(t, rend-1, meret/3)
15        t.left(60)
16        koch(t, rend-1, meret/3)
```

A kulcsponthoz és az újdonsághoz az, hogy ha a rend nem nulla, akkor a koch függvény rekurzívan meghívja önmagát azért, hogy elvégezze a feladatát.

Figyeld meg és rövidítsd le ezt a kódot. Emlékezz, hogy a 120 fokos jobbra fordulás ugyanolyan, mint a -120 fokos balra fordulás. Így egy kicsit ügyesebben szervezve a kódot, használhatunk egy ciklust a 10-16. sorok helyett:

```
1 def koch(t, rend, meret):
2     if rend == 0:
3         t.forward(meret)
4     else:
5         for szog in [60, -120, 60, 0]:
6             koch(t, rend-1, meret/3)
7             t.left(szog)
```

Az utolsó forgás 0 fokos – így nincs hatása. De lehetővé tette számunkra, hogy találjuk egy mintát és hét sornyi kódot háromra csökkentünk, amely elősegítette a következő észrevételeinket.

Rekurzió, a magas szintű nézet

Egy lehetséges útja, hogy megbizonyosodj arról, hogy a függvény megfelelően fog működni, ha a 0. rendű fraktált hívod. Tegyél gondolatban ugrást, mondván: „a tündér keresztanyja (vagy Python, ha úgy tekintesz a Pythonra, mint a tündér keresztanyára) tudja, hogyan kell meghívni a 0. rekurzív szintet a 11, 13, 15 és 17 sorokra, ezért nem szükséges ezen gondolkodni!” Mindössze arra kell fókuszálnod, hogy hogyan kell kirajzolni az 1. rendű fraktált, ha feltételezzük, hogy a 0. rendű már elkészült.

Ha gyakorlod a mentális absztrakciót – figyelmen kívül hagyhatod az alproblémát, amíg meg nem oldottad a nagyobbbat.

Ha ez a gondolkodásmód működik (ezt gyakorolni kell!), akkor léphetsz a következő szintre. Aha! Most látom, hogy akkor fog megfelelően működni, amikor a másodrendű hívás van, feltéve, hogy az 1. szint már működik.

És általában, ha feltételezzük, hogy az $n-1$ -es eset működik, meg tudjuk oldani az n -es szintű problémát is?

A matematikus hallgatók, akik már játszottak az indukciós bizonyítással, itt látniuk kell a hasonlóságot.

Rekurzió, az alacsony szintű operatív nézet

A rekurzió megértésének másik módja, ha megszabadulunk tőle. Ha külön függvényünk van a 3. szintű fraktálra, a 2., 1. és 0. szintű fraktálokra, akkor egyszerűen leegyszerűsíthetnénk ezt a kódot, mechanikusan, egy olyan helyzetre, ahol már nincs rekurzió, mint itt:

```
1 def koch_0(t, meret):
2     t.forward(meret)
3
4 def koch_1(t, meret):
5     for szog in [60, -120, 60, 0]:
6         koch_0(t, meret/3)
7         t.left(meret)
8
9 def koch_2(t, meret):
10    for szog in [60, -120, 60, 0]:
11        koch_1(t, meret/3)
12        t.left(szog)
13
14 def koch_3(t, meret):
15    for szog in [60, -120, 60, 0]:
16        koch_2(t, meret/3)
17        t.left(szog)
```

Ez a trükk a „visszatekerés”, egy áttekinthető nézetet ad a rekurzió működéséről. Nyomon követhetjük a programot a koch_3-ra, majd onnan a koch_2-re és a koch_1-re, stb. Végigvehetjük a rekurzió különböző rétegeit.

Ez hasznos lehet a megértés szempontjából. A cél azonban az, hogy képesek legyünk az absztrakció megvalósítására!

18.2. Rekurzív adatszerkezetek

Az eddig látott Python adattípusokat különféle módon tudjuk listákba és rendezett n -esekbe csoportosítani. A listák és a rendezett n -esek szintén beágyazhatók, így számos lehetőséget biztosítanak az adatok rendszerezésére. Az adatok szervezésének az a célja, hogy megkönnyítsék a felhasználásukat, ezt nevezzük **adatszerkezetnek**.

Választási időszak van és mi segítünk megszámolni a szavazatokat, ahogyan beérkeznek. Az egyes egységekből, körzetekből, önkormányzatokból, megyékből és államokból érkező szavazatokat néha összesítve, esetenként a szavazatok részarányának listájaként jelentik. Miután megvizsgáltuk, hogy miként lehet a legjobban tárolni az adatokat, úgy döntünk, hogy egy *beágyazott listát* használunk, melyet az alábbiak szerint definiálunk:

A *beágyazott lista* egy olyan lista, amelynek elemei:

1. számok
2. beágyazott listák

Figyeljük meg, hogy a *beágyazott lista* szintén szerepel a saját definíciójában. Az ilyen **rekurzív definíciók** meglehetősen gyakoriak a matematikában és informatikában. Ezek tömör és hatékony módot nyújtanak a **rekurzív adatszerkezetek** leírására, amelyek részben kisebb és egyszerűbb példái önmaguknak. A definíció nem körkörös, mivel egy bizonyos ponton olyan listához jutunk, amely nem tartalmaz további listaelemeket.

Most feltételezzük, hogy egy olyan függvényt kell létrehoznunk, amely összeadja a beágyazott lista összes elemét. A Pythonnak van egy beépített függvénye, amely megadja egy számsorozat összegét:

```
1 print(sum([1, 2, 8]))
```

```
11
```

A mi *beágyazott listánk* esetében azonban a `sum` nem fog működni:

```
1 print(sum([1, 2, [11, 13], 8]))
```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

A probléma a harmadik elemmel van, amely szintén egy lista `[11, 13]`, ezért nem lehet hozzáadni az 1, 2 és 8-at.

18.3. Listák rekurzív feldolgozása

A beágyazott lista összes elemének rekurzív összegzéséhez be kell járnunk a listát, érintve a beágyazott struktúra minden egyes elemét, hozzáadjuk minden elemét az összeghez, és *rekurzív módon ismételjük az összegzést* azokra az elemekre is, amelyek allisták.

A rekurciónak köszönhetően a beágyazott listák értékeinek összegzéséhez szükséges Python kód meglepően rövid:

```
1 def rek_szum(beagyazott_lista):
2     ossz = 0
3     for elem in beagyazott_lista:
4         if type(elem) == type([]):
5             ossz += rek_szum(elem)
6         else:
7             ossz += elem
8     return ossz
```

A `rek_szum` törzse tartalmaz egy olyan `for` ciklust, amely bejárja a `beagyazott_lista`-t. Ha az `elem` egy numerikus érték (az `else` ágon), egyszerűen csak hozzáadja a `ossz`-höz. Ha az `elem` egy lista, akkor ismét meghívjuk a `rek_szum`-ot, az elemre, mint egy argumentum. Azt az utasítást a függvény definícióban belül, mely meghívja önmagát, **rekurzív hívásnak** nevezzük.

A fenti példában van egy **alapeset** (a 13. sorban), amely nem vezet rekurzív híváshoz: abban az esetben, ha az `elem` nem egy (rész-) lista. Alapeset nélkül egy **végtelen rekurziót** kapunk, tehát a program nem fog működni.

A rekurzió valóban az egyik legszebb és legelegánsabb informatikai eszköz.

Egy kicsit bonyolultabb probléma a legnagyobb érték megtalálása a beágyazott listánkban:

```
1 def rek_max(nxs):
2     """
3     Keresd meg a maximumot rekurzív módon
4     egy beágyazott listában.
5     Előfeltétel: A listák vagy részlisták nem üresek.
6     """
7     legnagyobb = None
8     elso_alk = True
9     for e in nxs:
10        if type(e) == type([]):
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
11     ert = rek_max(e)
12     else:
13         ert = e
14
15     if elso_alk or ert > legnagyobb:
16         legnagyobb = ert
17         elso_alk = False
18
19     return legnagyobb
20
21 teszt(rek_max([2, 9, [1, 13], 8, 6]) == 13)
22 teszt(rek_max([2, [[100, 7], 90], [1, 13], 8, 6]) == 100)
23 teszt(rek_max([[13, 7], 90], 2, [1, 100], 8, 6) == 100)
24 teszt(rek_max(["jancsi", ["sanyi", "bence"]]) == "sanyi")
```

A tesztek példát adnak a `rek_max` működésére.

A csavar ebben a problémában, hogy megtaláljuk a legnagyobb változó kezdőértékét. Nem használhatjuk csak a `nxs[0]`-t, mivel ez lehet egy elem vagy egy lista. A probléma megoldásához (minden rekurzív hívásnál) inicializálunk egy kétállapotú jelzőt (8. sor). Amikor megtaláltuk a keresett értéket, ellenőrizzük (a 15. sorban), hogy vajon ez a kezdeti értéke a legnagyobb-nak vagy a legnagyobb értéket meg kell változtatni.

A 13. sorban ismét van egy alapeset. Ha nem adjuk meg az alapesetet, a Python megáll, miután eléri a maximális rekurziós mélységet és futási idejű hibát ad vissza. Figyeld meg, mi történik a következő szkript futtatása során, melyet *vegtelen_rekurzio.py*-nak neveztünk:

```
1 def rekurzio_melysege(szam):
2     print("{0}, ".format(szam), end="")
3     rekurzio_melysege(szam + 1)
4
5 print(rekurzio_melysege(0))
```

Az üzenetek villogása után megjelenik egy hosszú nyomkövetés, melyet a következő üzenet zár le:

```
RuntimeError: maximum recursion depth exceeded ...
```

Nem szeretnénk, hogy valami hasonló történjen a programjaink felhasználóival, ezért a következő fejezetben látni fogjuk hogyan kezelhetjük a hibákat és bármilyen hibát a Pythonban.

18.4. Esettanulmány: Fibonacci-számok

A híres **Fibonacci sorozat** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, ... melyet Fibonacci (1170-1250) fedezett fel, aki ezzel modellezte a nyulak (párok) tenyésztését. Ha 7 generációban összesen 21 pár van, ebből 13 felnőtt, a következő generációban a felnőtteknek lesznek gyerekeik, és az előző gyerekek pedig felnőtté válnak. Tehát a 8. generációban van $13+21=34$ nyúl párunk, amelyből 21 felnőtt.

Ez a *modell* a nyúl tenyésztésre vonatkozott, egy egyszerű feltétellel, hogy a nyulak sosem haltak meg. A tudósok gyakran (nem reális) egyszerűsítő feltételezéseket és korlátozásokat tesznek annak érdekében, hogy némi előrehaladást érjenek el a problémával.

Ha a sorozatba bevesszük a 0-t is, akkor minden egyes kifejezést rekurzívan írhatunk le az előző két kifejezés összegeként:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)   for n >= 2
```

Ezt lefordítva Pythonra:

```
1 def fib(n):
2     if n <= 1:
3         return n
4     t = fib(n-1) + fib(n-2)
5     return t
```

Ez egy kevésbé hatékony algoritmus, a javítására majd mutatunk egy módszert, amikor a szótárakról tanulunk:

```
1 import time
2 t0 = time.clock()
3 n = 35
4 eredmeny = fib(n)
5 t1 = time.clock()
6
7 print("fib({0}) = {1}, ({2:.2f} masodperc)".format(n, eredmeny, t1-t0))
```

Helyes eredményt kapunk, de ez nagyon sok időt vesz igénybe!

```
fib(35) = 9227465, (10.54 masodperc)
```

18.5. Példa a rekurzív könyvtárakra és fájlokra

A következő program kilistázza az adott könyvtár és összes alkönyvtárának tartalmát.

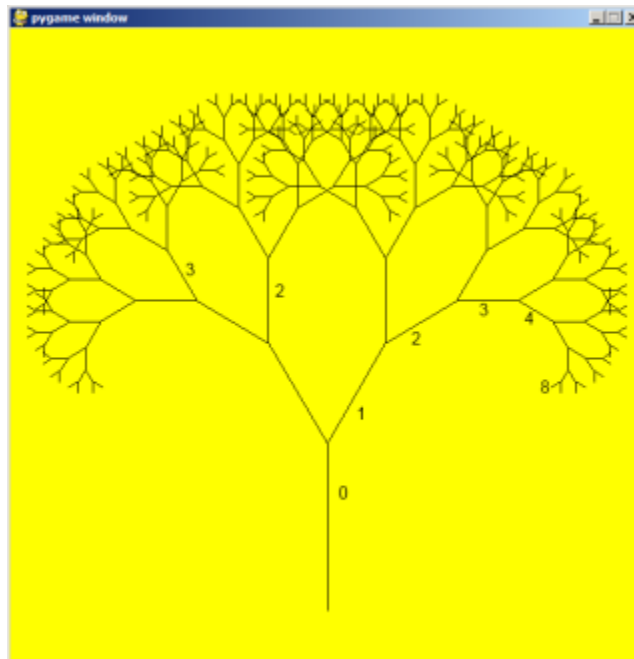
```
1 import os
2
3 def mappa_listazas(utvonal):
4     """
5     Visszaadja összes elem rendezett listáját az útvonalon.
6     Ez csak a neveket adja vissza, nem pedig a teljes elérési utat.
7     """
8     mappalista = os.listdir(utvonal)
9     mappalista.sort()
10    return mappalista
11
12 def fajok_kiiratas(utvonal, prefix = ""):
13     """ Az útvonalak tartalmának rekurzív kiírása. """
14     if prefix == "": # Észleli a legkülső hívást és kiírja a címsorát
15         print("A mappa kilistázása", utvonal)
16         prefix = "| "
17
18     mappalista = mappa_listazas(utvonal)
19     for f in mappalista:
20         print(prefix+f) # Sor kiírása
21         teljesnev = os.path.join(utvonal, f) # A név átváltása a teljes elérési útra
22         if os.path.isdir(teljesnev): # Ha könyvtár, újraindul
23             fajok_kiiratas(teljesnev, prefix + "| ")
```

A fajok_kiiratas függvényhívás az egyes mappák nevével a következőhöz hasonló kimenetet eredményez:

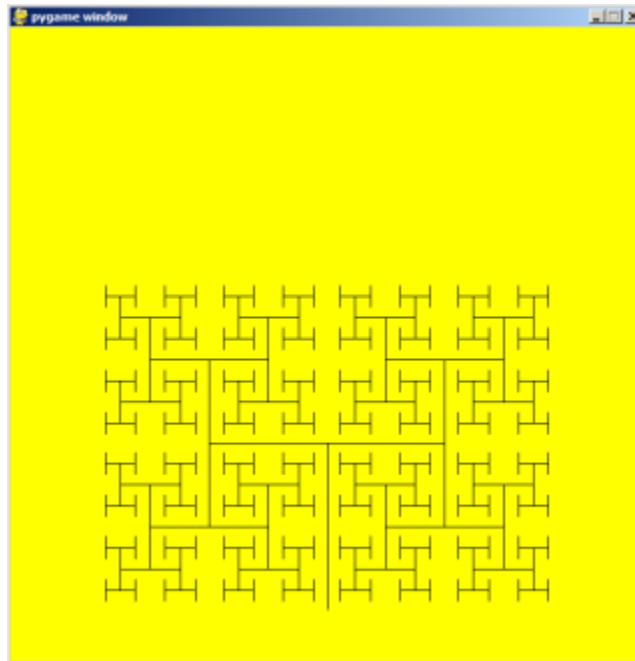
```
A mappa kilistázása c:\python31\Lib\site-packages\pygame\examples
| __init__.py
| aacircle.py
| aliens.py
| arraydemo.py
| blend_fill.py
| blit_blends.py
| camera.py
| chimp.py
| cursors.py
| data
| | alien1.png
| | alien2.png
| | alien3.png
...
```

18.6. Animált fraktál, PyGame használatával

Itt van egy 8-ad rendű fa fraktál mintázata. Címkével láttuk el néhány élt, amely megmutatja a rekurzió mélységét, ahol már minden él kirajzolásra került.



A fenti fában a törzstől való eltérés szöge 30 fokok. Ennek a szögnek a változtatása más érdekes alakokat ad, például 90 fokok esetén ezt kapjuk:



Érdekes animáció jön létre, ha nagyon gyorsan hozzuk létre és rajzoljuk ki a fákat, és minden egyes pillanatban kicsit megváltoztatjuk a szöveget. Habár a Turtle modul nagyon elegánsan képes ilyen fákat rajzolni, küzdhetünk a jó képfrissítési frekvenciáért. Ezért inkább PyGame-t használunk, néhány díszítéssel és megfigyeléssel. (Még egyszer azt javasoljuk, hogy vágd ki és illeszd ezt a kódot a Python környezetbe.)

```
1 import pygame, math
2 pygame.init()           # Előkészíti a pygame modult a használatra
3
4 # Hozz létre egy új felületet és ablakot.
5 felulet_meret = 1024
6 fo_felulet = pygame.display.set_mode((felulet_meret, felulet_meret))
7 my_clock = pygame.time.Clock()
8
9
10 def fa_rajzolasa(rend, szog, sz, poz, irany, szin=(0,0,0), melyseg=0):
11
12     torzs_arany = 0.29      # Milyen nagy a fa törzse az egész fához viszonyítva?
13     torzs = sz * torzs_arany # törzs hossza
14     delta_x = torzs * math.cos(irany)
15     delta_y = torzs * math.sin(irany)
16     (u, v) = poz
17     ujpoz = (u + delta_x, v + delta_y)
18     pygame.draw.line(fo_felulet, szin, poz, ujpoz)
19
20     if rend > 0:      # Rajzolj egy szintet
21
22         # A következő hat sor egyszerű megoldás nyújt arra, hogy a rekurzió a
23         # két nagyobb felét eltérő színűvé változtassa. Csaljunk itt egy kicsit, hogy
24         # megváltoztassuk a színeket a mélységekben, amikor a mélység páros vagy
25         # páratlan, stb.
26         if melyseg == 0:
27             szin1 = (255, 0, 0)
28             szin2 = (0, 0, 255)
29         else:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```

29     szin1 = szin
30     szin2 = szin
31
32     # hív meg rekurzívan, hogy kirajzolja a két részfát
33     ujsz = sz*(1 - torzs_arany)
34     fa_rajzolasa(rend-1, szog, ujsz, ujpoz, irany-szog, szin1, melyseg+1)
35     fa_rajzolasa(rend-1, szog, ujsz, ujpoz, irany+szog, szin2, melyseg+1)
36
37
38 def gameloop():
39
40     szog = 0
41     while True:
42
43         # Kezeld az eseményeket a billentyűzettel, egérrel stb.
44         esemeny = pygame.event.poll()
45         if esemeny.type == pygame.QUIT:
46             break;
47
48         # Aktualizálás - változtasd meg a szöget
49         szog += 0.01
50
51         # Rajzolj ki mindent
52         fo_felulet.fill((255, 255, 0))
53         fa_rajzolasa(9, szog, felulet_meret*0.9, (felulet_meret//2, felulet_meret-50),
54         ↪ -math.pi/2)
55
56         pygame.display.flip()
57         my_clock.tick(120)
58
59 gameloop()
60 pygame.quit()

```

- A “math”könyvtár radiánban és nem fokban mért szögekkel dolgozik.
- A 14. és a 15. sor középiskolai trigonometriai fogalmakat használ. A kívánt vonal hosszából (t_{runk}) és a kívánt szögből a \cos és \sin segít nekünk kiszámítani a x és y távolságokat, amiket mozgatni kell.
- A 22-30. sorok feleslegesek, kivéve, ha színes fát akarunk.
- A ciklus 49. sorában megváltoztatjuk a szöget minden képkockánál, és kirajzoljuk az új fát.
- A 18. sor azt mutatja, hogy a PyGame vonalakat is rajzolhat, és még sok mást. Nézd meg a dokumentációt. Például rajzolj egy kis kört az ágak minden egyes pontjában úgy, hogy ezt a sort közvetlenül a 18-as sor alá írod:

```

1 pygame.draw.circle(fo_felulet, szin, (int(pozn[0]), int(pozn[1])), 3)

```

Egy másik érdekes eredmény – tanulságos is, ha szeretnéd megerősíteni azt az ötletet, mely a függvény különböző példányait hívja a rekurzió különböző mélységeinél – hozd létre a színek listáját, és hagyd, hogy minden rekurzív mélység más színt használjon a rajzoláshoz. (Használd a rekurzió mélységét a színek listájának indexeléséhez.)

18.7. Szójegyzék

alapeset (base case) A rekurzív függvényben feltételes utasításának azon ága, amely nem vezet további rekurzív hívásokhoz.

rekurzió (recursion) Egy olyan függvény meghívása, amely már végrehajtás alatt áll.

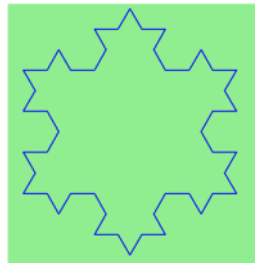
rekurzív definíció (recursive definition) Olyan definíció, amely önmagával definiál valamit. Ahhoz, hogy hasznos legyen tartalmaznia kell *alapesetek*-et, amelyek nem rekurzívak. Ily módon eltér a *körkörös definíciótól*. A rekurzív definíciók gyakran elegáns módot nyújtanak az összetett adatstruktúrák kifejezésére. Például egy könyvtár, amely más alkönyvtárakat vagy egy menü, amely más almenüket tartalmazhat.

rekurzív hívás (recursive call) Egy utasítás, amely egy már végrehajthatott függvényt hív. A rekurzió lehet közvetett is – az f függvény hívja a g -t, amelyik meghívja a h -t, és h visszahívhatja az f -et.

végtelen rekurzió (infinite recursion) Olyan függvény, amely rekurzív módon hívja önmagát anélkül, hogy bármilyen alapesetet elérne. Végül, a végtelen rekurzió egy futási idejű hibát okoz.

18.8. Feladatok

1. Módosítsd a Koch fraktál programot úgy, hogy egy Koch hópelyhet rajzoljon ki, így:



2. (a) Rajzolj egy Cesaro-fraktált, a felhasználó által megadott rendben. Megmutatjuk a vonalak négy különböző rendjét a 0, 1, 2, 3-at. Ebben a példában a törés szöge 10 fokos.



- (b) Négy vonal alkotja a négyzetet. Használd a kódot az a) részben a Cesaro négyzetek létrehozásához. A szög változtatása érdekes hatásokat eredményez – kísérletezz egy kicsit, vagy hagyd, hogy a felhasználó adhassa meg a törés szögét.

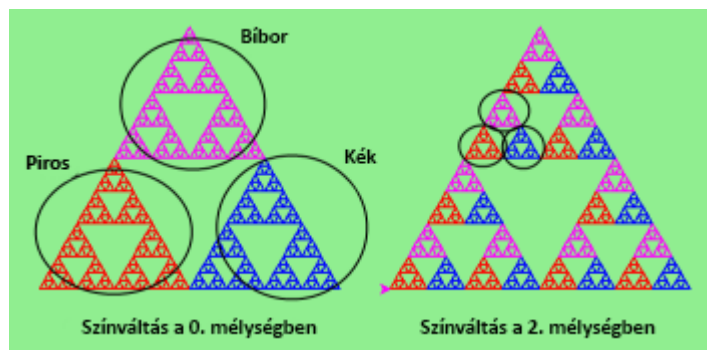


- (a) (A matematikai vénával megáldott hallgatóknak.) Az itt bemutatott négyzeteknél a magasabb rendű rajzok kicsit nagyobbak lesznek. (Tekintsd az egyes négyzetek legalsó vonalát - nincsenek igazítva.) Ennek az oka, hogy éppen most feleztük meg a vonalat minden egyes rekurzív alproblémára. Tehát a „teljes” négyzetet a szakadások szélessége növelte. Meg tudod oldani az a geometriai problémát, ha az alprobléma esetének teljes mérete (beleértve a szakadást is) pontosan ugyanolyan méretű legyen, mint az eredeti?

3. A 0. rendű Sierpinski-háromszög egy egyenlő oldalú háromszög. Az 1. rendűt le tudjuk rajzolni 3 kisebb háromszöggént (itt kissé szétválasztva, azért, hogy segítsen a megértésben.) A 2. és 3. rendű háromszög szintén látható. Rajzolj a felhasználó bemenetének megfelelő Sierpinski-háromszögeket.



4. Módosítsd a fenti programot úgy, hogy a három háromszög színei megváltozzanak, a rekurzió valamely mélységben. Az alábbi ábra két különböző esetet mutat be: a bal oldali képen, a szín a 0. mélységben változik (a rekurzió legmagasabb szintje), a jobb oldalinál pedig a 2. mélységben. Ha a felhasználó negatív mélységet ad meg, a szín ne változzon. (Tipp: adj hozzá egy új, opcionális `szinValtoMelyseg` paramétert (amely alapértelmezés szerint -1), és változtasd ezt kisebbre minden egyes rekurzív hívásnál. Ezután a kód ezen szakaszában, mielőtt újramezned, teszteld, hogy a paraméter nulla-e, és megváltoztatja-e a színt.)



5. Írj egy `rekurziv_min` függvényt, amely a visszaadja a beágyazott lista legkisebb elemét. Feltételezzük, hogy a lista vagy a részlista nem üres:

```
teszt(rekurziv_min([2, 9, [1, 13], 8, 6]) == 1)
teszt(rekurziv_min([2, [[100, 1], 90], [10, 13], 8, 6]) == 1)
teszt(rekurziv_min([2, [[13, -7], 90], [1, 100], 8, 6]) == -7)
teszt(rekurziv_min([[[-13, 7], 90], 2, [1, 100], 8, 6]) == -13)
```

6. Írj egy `szamol` függvényt, amely visszaadja egy `cel` elem előfordulásának számát a beágyazott listában:

```
teszt(szamol(2, []), 0)
teszt(szamol(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]]) == 4)
teszt(szamol(7, [[9, [7, 1, 13, 2], 8], [7, 6]]) == 2)
teszt(szamol(15, [[9, [7, 1, 13, 2], 8], [2, 6]]) == 0)
teszt(szamol(5, [[5, [5, [1, 5], 5], 5], [5, 6]]) == 6)
teszt(szamol("a",
    [{"ez", ["a", ["keres", "a"], "a"], "nagyon"], ["a", "konnyu"]}] == 4)
```

7. Írjon egy olyan `kisimit` függvényt, amely egy egyszerű listát ad vissza, amely tartalmazza az összes, a beágyazott listán szereplő értéket:

```
teszt(kisimit([2, 9, [2, 1, 13, 2], 8, [2, 6]]) == [2, 9, 2, 1, 13, 2, 8, 2, 6])
teszt(kisimit([[9, [7, 1, 13, 2], 8], [7, 6]]) == [9, 7, 1, 13, 2, 8, 7, 6])
teszt(kisimit([[9, [7, 1, 13, 2], 8], [2, 6]]) == [9, 7, 1, 13, 2, 8, 2, 6])
teszt(kisimit([{"ez", ["a", ["keres"], "a"], "nagyon"}, {"a", "konnyu"}]) ==
    ["ez", "a", "keres", "a", "nagyon", "a", "konnyu"])
teszt(kisimit([]) == [])
```

8. Írd újra a Fibonacci algoritmust rekurzió nélkül. Találsz nagyobb elemet a sorozatnak? Megtalálsz a `fib(200)`?
9. Használd a Python dokumentációt, hogy megismerd a `sys.getrecursionlimit()` és a `sys.setrecursionlimit(n)`-t. Végezz számos kísérletet, hasonlóan ahhoz, mint amit az `infinite_recursion.py` program során végeztél, hogy megértsd ezen modulok, függvények működését.
10. Írj egy olyan programot, amely könyvtárstruktúrát jár be (mint a fejezet utolsó részében), de a fájlnevek kiírása helyett a könyvtárban vagy az alkönyvtárakban lévő fájlok teljes elérési útját add vissza. (Ne szerepeljenek a könyvtárak ezen a listán – csak fájlok.) Például a kimeneti listának ilyen elemei lehetnek:

```
["C:\\Python31\\Lib\\site-packages\\pygame\\docs\\ref\\mask.html",
 "C:\\Python31\\Lib\\site-packages\\pygame\\docs\\ref\\midi.html",
 ...
 "C:\\Python31\\Lib\\site-packages\\pygame\\examples\\aliens.py",
 ...
 "C:\\Python31\\Lib\\site-packages\\pygame\\examples\\data\\boom.wav",
 ... ]
```

11. Írj egy `szemetel.py` nevű programot, amely létrehoz egy `lomtar.txt` nevű fájlt a könyvtárfa minden egyes alkönyvtárába, argumentumként add meg a fájlgyökerét (vagy az aktuális könyvtárat alapértelmezettként). Most írd meg a `tisztit.py` nevű programot, amely eltávolítja ezeket a fájlokat.

Tipp #1: Használd a fejezet utolsó részében található példa programot a két rekurzív program alapjaként. Mivel azt tervezted, hogy a lemezeden lévő fájlokat fogsz megsemmisíteni, ezt nagyon jól kell megcsinálnod, különben azt kockáztatod, hogy elveszíted a fájljaidat. Egy hasznos tanács: tegyél úgy, mintha kitörölnéd a fájlokat – de csak írasd ki a törölni kívánt fájlok teljes elérési útját. Ha elégedett vagy az eredménnyel, azt látod, hogy helyes és nem töröl a rossz dolgokat, akkor helyettesítheted a kiíratást az igazi utasítással.

Tipp #2: Keress az `os` modulban, egy fájlok törlésére szolgáló függvényt.

19. fejezet

Kivételek

19.1. Kivételek elkapása

Valahányszor egy futási idejű hiba lép fel, létrejön egy **kivétel** objektumot. A program ezen a ponton leáll, és a Python kiírja a visszakövetési információkat, amely egy olyan üzenettel végződik, mely leírja a bekövetkezett kivételt:

Például a nullával való osztáskor létrehozott kivétel:

```
1 print(55/0)
```

```
Traceback (most recent call last):  
  File "<interactive input>", line 1, in <module>  
ZeroDivisionError: integer division or modulo by zero
```

Egy nem létező listaelemhez való hozzáférés esetén:

```
1 a = []  
2 print(a[5])
```

```
Traceback (most recent call last):  
  File "<interactive input>", line 1, in <module>  
IndexError: list index out of range
```

Vagy ha megpróbálunk egy elemet hozzárendelni egy rendezett n-eshez:

```
1 tup = ("a", "b", "d", "d")  
2 tup[2] = "c"
```

```
Traceback (most recent call last):  
  File "<interactive input>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

Az utolsó sorban lévő hibaüzenet minden esetben két részből áll: a hiba típusából, mely a kettőspont előtt van, és a hiba leírásából a kettőspont után.

Néha szeretnénk, hogy egy művelet végrehajtása kivételhez vezessen, de nem akarjuk, hogy a program leálljon. A `try` utasításba „csomagolt” kódrészlettel **kezelhetjük a kivételt**.

Például, bekérhetjük a felhasználótól a fájl nevét, majd megpróbáljuk megnyitni. Amennyiben a fájl nem létezik, nem akarjuk, hogy a program összeomoljon; szeretnénk kezelni a kivételt:

```
1 fajlnev = input("Add meg a fájl nevét: ")
2 try:
3     f = open(fajlnev, "r")
4 except:
5     print("Nincs ilyen nevű fájl!", fajlnev)
```

A try utasítás három különálló ágból vagy részből áll, a következő kulcsszavakkal kezdve try ... except ... finally. Vagy az except vagy a finally ág elhagyható, ezért a fenti tekinthető a try utasítás leggyakoribb formájának.

A try utasítás végrehajtja és ellenőrzi az utasításokat az első blokkban. Ha nem lép fel kivétel, akkor átugorja az except alatti blokkot. Ha valamilyen kivétel kiváltódik, végrehajtja az utasításokat az except ágban.

A kivételkezelést függvénybe is ágyazhatjuk: a letezik olyan függvény, mely kap egy fájlnevet és igazat ad vissza, ha a fájl létezik, és hamisat, ha nem:

```
1 def letezik(fajlnev):
2     try:
3         f = open(fajlnev)
4         f.close()
5         return True
6     except:
7         return False
```

Egy sablon a fájl létezésének tesztelésére, kivételek használata nélkül

A függvény, melyet most bemutatunk nem ajánlott. Megnyitja és bezárja a fájlt, amely szemantikailag különbözik attól, hogy megkérdezze, hogy létezik-e? Hogyan? Először is aktualizálhat néhány, a fájlhoz tartozó időbélyeget. Másodsorban azt mondhatja, hogy nincs ilyen fájl, ha más program már megnyitotta a fájlt, és nem engedélyezi, hogy mi is megnyissuk.

A Python egy `os.path` nevű függvényt ajánl az `os` modulban. Számos hasznos függvénnyel rendelkezik az elérési útvonalak, fájlok és könyvtárak kezeléséhez, ezért érdemes lenne megnézni a Python dokumentációt.

```
1 import os
2
3 # Ez egy kedvelt módja a fájl létezésének ellenőrzésére
4 if os.path.isfile("c:/temp/testdata.txt"):
5     ...
```

Használhatunk többszörös except ágot a különböző típusú kivételeket kezelésére (lásd. [Hibák és Kivételek...](#) példák Guido van Rossum-tól, a Python alkotójától, [Python Tananyag](#) a kivételek sokkal részletesebb bemutatásának érdekében). Tehát a program mást tehet, ha a fájl nem létezik, és mást, ha a fájlt egy másik program használja.

19.2. Saját kivételek létrehozása

Tud-e a programunk szándékosan saját kivételeket létrehozni? Ha a programunk egy hibát észlel, akkor kivétel lép fel. Íme egy példa, amelyik a bemenetet a felhasználótól kapja és ellenőrzi, hogy a kapott szám negatív-e?

```
1 def ev_keres():
2     ev = int(input("Írd be az életkorodat: "))
3     if ev < 0:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4         # Hozd létre a kivétel új példányát
5         saját_hiba = ValueError("{0} érvénytelen életkor.".format(ev))
6         raise saját_hiba
7     return ev
```

Az 5. sor létrehoz egy kivétel objektumot, ebben az esetben a `ValueError` objektumot, amely összefoglalja a hibára vonatkozó speciális információkat. Feltételezzük, hogy ebben az esetben az A függvény hívja a B-t, amely hívja a C-t, amely hívja a D-t, amely hívja az `ev_keres()`-t. A 6. sorban szereplő `raise` utasítás ezt az objektumot egyfajta „visszatérési értékként” adja meg, és azonnal kilép az `ev_keres()` függvényből és visszatér D-be, a hívó függvénybe. Ezután a D is befejeződik és visszatér a hívójába C-be, és C kilép a B-be és így tovább, mindegyik visszatéríti a kivétel objektumot a hívójukhoz, amíg meg nem találja a `try ... except` utasítást, amely kezeli a kivételt. Ezt úgy hívjuk, mint: „felgöngyölíti a hívási vermet”.

A `ValueError` az egyik olyan beépített kivétel típus, amely a legközelebb van ahhoz a hibatípushoz, melyet szeretnénk kiváltani. A beépített kivételek teljes felsorolása megtalálható a [Beépített kivételek](#) című fejezetben a [Python Library Reference](#)-ben, melyet szintén a Python alkotója, Guido van Rossum írt.

Ha a függvény, melyet `ev_keres`-nek hívunk (vagy a függvény hívója(i)) kezeli a hibát, akkor a program folytatja a futást, egyébként a Python kiírja a visszakövetési információkat és kilép:

```
1     print(ev_keres())
```

```
Írd be az életkorodat: 42
42
```

```
1     print(ev_keres())
```

```
Írd be az életkorodat: -2

Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "learn_exceptions.py", line 4, in ev_keres
    raise ValueError("{0} érvénytelen életkor.".format(ev))
ValueError: -2 érvénytelen életkor.
```

A hibaüzenet tartalmazza a kivétel típusát és a kiegészítő információkat, amelyekről akkor gondoskodtak, amikor a kivétel objektumot először létrehozták.

Gyakran előfordul, hogy az 5-ös és a 6-os sorok (a kivétel objektum létrehozása, majd a kivétel kiváltása) egyetlen utasítás, de valójában két különböző és egymástól független dolog történik, így talán érdemes a két lépést különválasztani, amikor először tanulunk a kivételekről. Itt mindent egyetlen utasításban mutatunk be:

```
1     raise ValueError("{0} érvénytelen életkor.".format(ev))
```

19.3. Egy korábbi példa áttekintése

A kivételkezelést használva most már megváltoztathatjuk az előző fejezet `rekurzio_melysege` nevű példáját, így megáll a maximális rekurzív mélység elérésekor:

```
1     def rekurzio_melysege(szam):
2         print("Rekurziós mélység száma:", szam)
3         try:
4             rekurzio_melysege(szam + 1)
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5     except:
6         print("Nem lehetséges a mélyebb szint.")
7
8     print(rekurzio_melysege(0))
```

Futtasd ezt a verziót és figyeld meg az eredményeket.

19.4. A `finally` ág és a `try` utasítás

Egy gyakori programozási minta, hogy lefoglalunk bizonyos erőforrásokat, pl. létrehozunk egy ablakot a teknősök számára, hogy rajzoljanak, vagy csatlakozzunk az internetszolgáltatóhoz, vagy megnyitunk egy fájlt írásra. Ezután elvégezzük néhány számítást, amely kivételt okozhat, vagy problémamentesen működhet.

Bármi is történik, „fel kell szabadítanunk” az általunk lefoglalt erőforrásokat – például zárjuk be az ablakot, szakítsuk meg az internet kapcsolatot, vagy zárjuk be a fájlt. A `try` utasítás `finally` ágával ezt megtehetjük. Tekintsük ezt a (kissé erőltetett) példát:

```
1  import turtle
2  import time
3
4  def poly_rajz():
5      try:
6          ablak = turtle.Screen()    # Hozz létre egy ablakot
7          Eszti = turtle.Turtle()
8
9          # Ez a párbeszéd törölhető,
10         # vagy ha az int-re való konverzió nem sikerül, vagy ha az_
11         ↪ n nulla.
12         n = int(input("Hány oldalú sokszöget szeretnél?"))
13         szog = 360 / n
14         for i in range(n):          # Rajzold le a sokszöget
15             Eszti.forward(10)
16             Eszti.left(angle)
17             time.sleep(3)            # A program vár néhány másodpercet
18         finally:
19             ablak.bye()              # Zárd be a teknőc ablakot
20
21     print(poly_rajz())
22     print(poly_rajz())
23     print(poly_rajz())
```

A 20-22. sorokban a `poly_rajz`-ot háromszor hívjuk meg. Mindegyik új ablakot hoz létre a teknősnek, és egy sokszöget rajzol a felhasználó által megadott oldalak számával. De mi van akkor, ha a felhasználó beír egy olyan karakterláncot, amelyet nem lehet `int`-re konvertálni? Mi van, ha bezárjuk a párbeszédablakot? Kivételt kapunk *de annak ellenére, hogy kivétel lépett fel, még mindig szeretnénk bezárni a teknős ablakát.* A 17-18. sorok ezt megteszik számunkra. Függetlenül attól, hogy sikerült-e befejezni vagy sem a `try` utasítást, a `finally` blokk mindig végrehajtásra kerül.

Vegyük észre, hogy a kivétel még mindig nincs kezelve – csak az `except` ágak kezelik a kivételeket, így programunk még mindig összeomlik. De legalább a teknős ablak zárva lesz, mielőtt összeomlik!

19.5. Szójegyzék

kivált (raise) Egy kivétel szándékos létrehozása az `raise` utasítás használatával.

kivétel (exception) Egy futási idejű hiba.

kivételkezelés (handle an exception) Egy kódrészlet `try ... except` blokkba csomagolásával megelőzi, hogy a program összeomoljon egy kivétel kiváltódása miatt.

19.6. Feladatok

1. Írj egy `olvas_pozint` nevű függvényt, amely az `input` függvénnyel bekér a felhasználótól egy pozitív egész számot, majd ellenőrizd a bevitelt, hogy megfelel-e a követelményeknek. Képesnek kell lennie arra, hogy olyan bemeneteket kezeljen, amelyeket nem lehet `int`-re konvertálni vagy negatív `int`-re, és képesnek kell lennie a szélsőséges esetek kezelésére is (pl. Amikor a felhasználó egyáltalán nem ad meg semmit.)

20. fejezet

Szótárak

Az eddig részletesen tanulmányozott összetett adattípusok – sztringek, listák és rendezett n-esek – olyan szekvencia-típusok, amelyek egész számokat használnak indexként a bennük tárolt értékek eléréséhez.

A **szótárak** összetett adattípusok. Ezek a Python beépített **leképezési típusai** (mapping type). Leképezik a **kulcsokat** az értékekre. A kulcsok bármilyen megváltozhatatlan típusúak lehetnek. Az értékek, csak úgy mint egy listánál és a rendezett n-eseknél, bármilyen (akár különböző) típusúak is lehetnek. Más nyelvekben asszociatív tömböknek nevezik, mivel egy kulcsot rendel hozzá egy értékhez.

Például hozzunk létre egy szótárt, amely lefordítja a magyar szavakat spanyolra. Ezért ebben a szótárban a kulcsok sztringek.

A szótár létrehozásának egyik módja, ha egy üres szótár létrehozásával kezdünk, és hozzáadunk **kulcs:érték párokat**. Az üres szótárat `{ }` jelöljük:

```
1 hun2esp = {}  
2 hun2esp["egy"] = "uno"  
3 hun2esp["kettő"] = "dos"
```

Az első hozzárendelés létrehoz egy `hun2esp` nevű szótárt; a többi hozzárendelés új kulcs:érték párokat rendel a szótárhoz. A szótár aktuális értékét a szokásos módon írhatjuk ki:

```
1 print(hun2esp)  
  
{ "kettő": "dos", "egy": "uno" }
```

A szótárban lévő kulcs:érték párok vesszővel vannak elválasztva egymástól. Minden pár tartalmaz egy kulcsot és egy értéket kettősponttal elválasztva.

Hasítás (Hashelés)

A párok sorrendje talán nem olyan, mint amire számítottunk. A Python komplex algoritmusokat használ, amelyeket nagyon gyors elérésre terveztek, hogy meghatározzák, hogy a kulcs:érték párok a szótárban vannak-e tárolva. A mi céljainknak megfelel, ha az elrendezést kiszámíthatatlannak tekintjük.

Csodálkozhat az, hogy vajon miért használunk szótárakat, amikor rendezett n-esek listájával is implementálhatnánk ugyanezt a koncepciót, mely a kulcsokat értékekre képezi le.

```
1 {"alma": 430, "banán": 312, "narancs": 525, "körte": 217}  
2 {"körte": 217, 'alma': 430, 'narancs': 525, 'banán': 312}
```

```
[('alma', 430), ('banán', 312), ('narancs', 525), ('körte', 217)]  
[('alma', 430), ('banán', 312), ('narancs', 525), ('körte', 217)]
```

Ennek az oka az, hogy a szótárak nagyon gyorsak, ugyanis egy hasítás nevezetű technikával vannak megvalósítva, mely lehetővé teszi számunkra, hogy nagyon gyorsan elérjük az értékeket. Ezzel ellentétben a rendezett n-esek listájával történő megvalósítás lassú. Ha szeretnénk megtalálni egy kulcshoz rendelt értéket, a lista minden rendezett n-esében meg kell vizsgálni a 0. elemet (a kulcsot). Mi történik akkor, ha a kulcs nem szerepel a listán? Szintén be kell járnunk ahhoz, hogy ezt kiderítsük.

A szótár létrehozásának másik módja, hogy megadjuk a kulcsok listáját a kulcs:érték párokhoz, ugyanazt a szintaxist használva, mint amit az előző kimenetnél láttunk:

```
1 hun2esp = {"egy": "uno", "kettő": "dos", "három": "tres"}
```

Nem számít, hogy milyen sorrendben írjuk a párokat. A szótárban szereplő értékek a kulcsokkal érhetők el, nem indexekkel, így nem kell törődni az elrendezéssel.

Tehát használhatjuk a kulcsot a megfelelő érték megkereséséhez:

```
1 print(hun2esp["kettő"])
```

```
'dos'
```

A "kettő" leképeződik a "dos" értékre.

A listákat, rendezett n-eseket és sztringeket *szekvenciáknak* nevezzük, mert az elemeik rendezettek. Az általunk látott összetett típusok közül a szótár az első, amely nem szekvenciális, tehát nem tudjuk sem indexelni, sem szeletelni.

20.1. Szótár műveletek

A `del` utasítás eltávolít egy kulcs:érték párt a szótárból. Például a következő szótár különböző gyümölcsök nevét és a készleten lévő gyümölcsök számát tartalmazza:

```
1 keszlet = {"alma": 430, "banán": 312, "narancs": 525, "körte": 217}  
2 print(keszlet)
```

```
{'körte': 217, 'alma': 430, 'narancs': 525, 'banán': 312}
```

Ha valaki megveszi az összes körtét, eltávolíthatjuk a bejegyzést a szótárból:

```
1 del keszlet["körte"]  
2 print(keszlet)
```

```
{'alma': 430, 'narancs': 525, 'banán': 312}
```

Vagy ha hamarosan még körtére számítunk, akkor csak a körtéhez rendelt értéket változtatjuk meg:

```
1 keszlet["körte"] = 0  
2 print(keszlet)
```

```
{'körte': 0, 'alma': 430, 'narancs': 525, 'banán': 312}
```

A beérkező új banán szállítmánya így kezelhető:

```
1 keszlet["banán"] += 200
2 print(keszlet)
```

```
{'körte': 0, 'alma': 430, 'narancs': 525, 'banán': 512}
```

A `len` függvény a szótárakkal is működik; visszaadja a kulcs:érték párok számát:

```
1 print(len(keszlet))
```

```
4
```

20.2. Szótár metódusok

A szótáraknak számos hasznos beépített metódusa van.

A `keys` metódus visszaadja a szótárban álló kulcsok listáját, amit a Python 3 **nézet**-nek nevez. A nézet objektumnak van néhány hasonló tulajdonsága a korábban látott `range` objektumhoz – ez is egy lusta ígéret, akkor adja át az elemeit, amikor szükség van rá a program hátralévő részében. Bejárhatjuk a nézetet, vagy átalakíthatjuk egy listává, például így:

```
1 for k in hun2esp.keys(): # A k rendje nem definiált
2     print("A(z) ", k, " kulcs a leképezi a(z) ", hun2esp[k], " értéket.")
3
4 ks = list(hun2esp.keys())
5 print(ks)
```

Ezt a kimenetet eredményezi:

```
A(z) három kulcs leképezi a(z) tres értéket.
A(z) kettő kulcs leképezi a(z) dos értéket.
A(z) egy kulcs leképezi a(z) uno értéket.

['három', 'kettő', 'egy']
```

Annyira gyakori, hogy egy szótárban bejárjuk a kulcsokat, hogy elhagyhatjuk a `keys` metódushívást a `for` ciklusban – a szótár bejárása implicit módon a kulcsokat járja be:

```
1 for k in hun2esp:
2     print("A kulcs", k)
```

A `values` metódus hasonló; visszaad egy olyan nézet objektumot, amely listává alakítható:

```
1 print(list(hun2esp.values()))
```

```
['tres', 'dos', 'uno']
```

A `items` metódus szintén visszaad egy nézetet, amely egy rendezett n-es listát eredményez – a rendezett n-es minden eleme egy kulcs:érték pár:

```
1 print(list(hun2esp.items()))
```

```
[('három', 'tres'), ('kettő', 'dos'), ('egy', 'uno')]
```

A ciklusok során a rendezett n-esek gyakran hasznosak a kulcs és az érték egy időben történő eléréséhez:


```
1 for (k,v) in hun2esp.items():  
2     print("A(z)", k, "leképezése a(z) ", v, ".")
```

Ez a következőket eredményezi:

```
A(z) három leképezése a(z) tres.  
A(z) kettő leképezése a(z) dos.  
A(z) egy leképezése a(z) uno.
```

Az `in` és `not in` operátorok megvizsgálják, hogy egy kulcs benne van-e a szótárban:

```
1 print("egy" in hun2esp)
```

```
True
```

```
1 print("hat" in hun2esp)
```

```
False
```

```
1 print("tres" in hun2esp)  
2 # Jegyezd meg, hogy az 'in' a kulcsokat vizsgálja nem az értékeket.
```

```
False
```

Ez a módszer nagyon hasznos lehet, mert ha a szótárban nem-létező kulcsra hivatkozunk, az futási idejű hibát okoz:

```
1 print(hun2esp["kutya"])
```

```
Traceback (most recent call last):  
...  
KeyError: 'kutya'
```

20.3. Fedőnevek és másolás

Mivel a szótárak megváltoztathatók, úgy mint a listák esetében, szükséges ismernünk a fedőnév fogalmát. Ha két változó azonos objektumra utal, akkor az egyik változó módosításai hatással vannak a másikra.

Ha módosítani akarunk egy szótárt, és szeretnénk megtartani az eredeti példányát, használjuk a `copy` metódust. Például, az ellentetek egy olyan szótár, amely ellentét párokat tartalmaz:

```
1 ellentetek = {"fel": "le", "jó": "rossz", "igen": "nem"}  
2 alnev = ellentetek  
3 masolat = ellentetek.copy() # Felszínes másolás
```

Az `alnev` és az `ellentetek` ugyanazon objektumra hivatkoznak; a `masolat` ugyanazon szótár frissített másolata utal. Ha módosítjuk az `alnev`-et, az `ellentetek` is megváltozik:

```
1 alnev["jó"] = "hibás"  
2 print(ellentetek["jó"])
```

```
'hibás'
```

Ha módosítjuk a `masolat`-ot, az `ellentetek` nem változik meg:

```
1 masolat["jó"] = "téves"
2 print(ellentetek["jó"])
```

```
'hibás'
```

20.4. Ritka mátrixok

Korábban egy mátrixot listák listájával ábrázoltunk. Ez egy jó választás egy olyan mátrix ábrázolására, amelynek főként nem nulla értékei vannak, de most tekintsünk egy *ritka mátrixot* mint ezt:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

A lista reprezentációja sok nullát tartalmaz:

```
1 matrix = [[0, 0, 0, 1, 0],
2           [0, 0, 0, 0, 0],
3           [0, 2, 0, 0, 0],
4           [0, 0, 0, 0, 0],
5           [0, 0, 0, 3, 0]]
```

Egy alternatíva a szótár használata. A kulcsok esetében használhatunk rendezett n-eseket, melyek sor- és oszlopszámokat tartalmaznak. Itt van ugyanannak a mátrixnak az szótár segítségével történő ábrázolása:

```
1 matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

Mindössze három kulcs:érték párra van szükségünk, egy a mátrix minden nem nulla elemére. Minden kulcs egy rendezett n-es, és minden érték egy egész szám.

A mátrix egy elemének eléréséhez a `[]` operátort használhatjuk:

```
1 print(matrix[(0, 3)])
```

```
1
```

Figyeljük meg, hogy a szótár ábrázolásának szintaxisa nem ugyanaz, mint a beágyazott lista reprezentációjának szintaxisa. Két egész index helyett egy indexet használunk, amely egy egészekből álló rendezett n-es.

Van egy kis probléma. Ha egy olyan elemet adunk meg, amelyik nulla, akkor hibaüzenetet kapunk, mivel a szótárban nincs bejegyzés ezzel a kulccsal:

```
1 print(matrix[(1, 3)])
```

```
KeyError: (1, 3)
```

A `get` metódus megoldja ezt a problémát:

```
1 print(matrix.get((0, 3), 0))
```

```
1
```

Az első argumentum a kulcs; a második argumentum a `get` értékével tér vissza, ha a kulcs nincs a szótárban:

```
1 print(matrix.get((1, 3), 0))
```

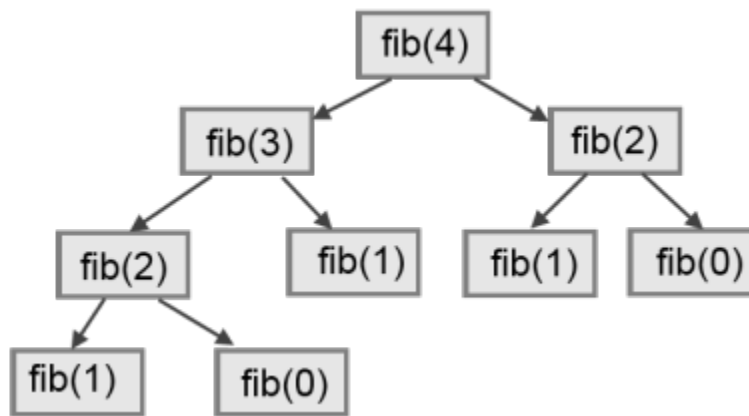
```
0
```

A `get` határozottan javítja a ritka mátrixok elérésének szemantikáját. Megszégyenítve a szintaxist.

20.5. Memoizálás (a feljegyzéses módszer)

Ha játszottál a rekurzióról szóló fejezetben a `fib` függvénnyel, akkor észreveheted, hogy minél nagyobb az argumentum, annál hosszabb ideig fut a függvény. Ráadásul a futási idő is nagyon gyorsan nő. Az egyik gépünkön a `fib(20)` azonnal befejeződik, a `fib(30)` körülbelül egy másodpercet vesz igénybe, és a `fib(40)` durván „örökké” tart.

Hogy megértsük, miért van ez, tekintsük az alábbi **hívási gráfot** a `fib` függvény esetén az $n = 4$ -re:



A hívási gráf bemutat néhány függvény keretet (példányokat, amikor a függvényt meghívták), olyan vonalakkal, amelyek összekötnék minden egyes keretet a meghívott függvények kereteivel. A grafikon tetején `fib` függvény az $n = 4$ -el meghívja `fib`-et az $n = 3$ -mal és az $n = 2$ -vel. Tovább a `fib` az $n = 3$ -mal meghívja a `fib` $n = 2$ -t és $n = 1$ -et. És így tovább.

Számolja meg, hányszor történik a `fib(0)` és a `fib(1)` hívása. Ez nem túl hatékony megoldása a problémának, és sokkal rosszabbá válik, ahogy az argumentum egyre nagyobb lesz.

Az a jó megoldás, hogyha nyomonkövetjük azokat az értékeket, amelyek már kiszámításra kerültek egy szótárban tárolva őket. Egy későbbi felhasználás érdekében tárolt, korábban kiszámított értéket **memo**-nak, vagyis emlékeztetőnek nevezik. Itt van a `fib` végrehajtása a memo-val:

```
1 mar_ismert = {0: 0, 1: 1}
2
3 def fib(n):
4     if n not in mar_ismert:
5         uj_ertek = fib(n-1) + fib(n-2)
6         mar_ismert[n] = uj_ertek
7     return mar_ismert[n]
```

A `marismert` szótár a már kiszámolt Fibonacci-számokat követi nyomon. Csak két párral kezdjük: 0-t leképezi 1-re; és az 1-et 1-re.

Amikor a `fib` meghívásra kerül, ellenőrzi a szótárt, hogy tartalmazza-e az eredményt. Ha igen, akkor a függvény azonnal visszatérhet anélkül, hogy rekurzív hívásokat kellene megtennie. Ha nem, akkor ki kell számolnia az új értéket. Az új érték hozzáadódik a szótárhoz, mielőtt a függvény visszatér.

A `fib` függvény ezen verziójának használatával a számítógépeink a `fib(100)`-at egy szempillantás alatt kiszámítják.

```
1 print(fib(100))
```

```
354224848179261915075
```

20.6. Betűk számlálása

A 8. fejezet (Sztringek) gyakorlataiban egy olyan függvényt írtunk, amely megszámlálta a betűk előfordulásának számát. A probléma általánosabb verziója a sztringben lévő betűk gyakorisági táblázata, tehát az, hogy az egyes betűk hányszor fordulnak elő.

Egy ilyen gyakorisági táblázat hasznos lehet egy szövegfájl tömörítéséhez. Mivel a különböző betűk különböző gyakorisággal jelennek meg, tömöríthetjük a fájlt rövidebb kódokat használva a közös betűkhöz és hosszabb kódokat a kevésbé gyakran megjelenő betűkhöz.

A szótárak elegáns módon generálnak egy gyakorisági táblát:

```
1 betu_szamlalo = {}
2 for betu in "Mississippi":
3     betu_szamlalo[betu] = betu_szamlalo.get(betu, 0) + 1
4 print(betu_szamlalo)
```

```
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Egy üres szótárral kezdünk. A sztring minden egyes betűjére megkeressük az aktuális számlálót (esetleg nullát) és növeljük azt. Végül a szótár a betűk és azok gyakoriságait tartalmazza.

Sokkal szebb, hogyha a gyakorisági táblázatot betűrendben jelenítjük meg. Ezt a `items` és `sort` metódusokkal tehetjük meg:

```
1 betuk = list(betu_szamlalo.items())
2 betuk.sort()
3 print(betuk)
```

```
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Figyeld meg, hogy az első sorban meg kellett hívni a `list` típus átalakító függvényt. Ez a `items`-ből származó elemeket egy listává konvertálja, ez a lépés szükséges ahhoz, hogy a listán a `sort` metódust használhassuk.

20.7. Szójegyzék

hívási gráf (call graph) Olyan gráf, amely csomópontokat tartalmaz, melyek függvényeket (vagy hívásokat), és irányított éleket (nyilakat) tartalmaznak, amelyekből kiderül, hogy mely függvények hoztak létre más függvényeket.

kulcs (key) Egy olyan adatelem, amely egy szótárbeli értékre lesz *leképezve*. A kulcsok segítségével meglehetősen könnyű keresni az értékeket egy szótárban. Minden kulcsnak egyedinek kell lennie a szótárban.

kulcs:érték pár (key:value pair) A szótár egyik elempárja. Az értékeket kulcs alapján keresik a szótárban.

leképezés típus (mapping type) A leképezés típus a kulcsok és a kapcsolódó értékek gyűjteményéből álló adattípus. A Python egyetlen beépített leképezési típusa a szótár. A szótárak implementálhatják az *asszociatív tömbök* absztrakt adattípust.

megváltoztathatatlan adatérték (immutable data value) Egy olyan adatérték, amelyet nem lehet módosítani. Elem hozzárendelése vagy a szeletelés (alrész képzés) megváltoztathatatlan érték esetén futási idejű hibát okoz.

memo (memo) Az előre kiszámított értékek ideiglenes tárolása, hogy elkerüljük az azonos számítások ismétlését.

módosítható adatérték (mutable data value) Olyan adatérték, mely módosítható. Az összes módosítható érték típusa összetett. A listák és a szótárak változtathatóak; a sztringek és a rendezett n-esek nem.

szótár (dictionary) A kulcs:érték párok gyűjteménye, mely a kulcsokat értékekre képezi le. A kulcsok megváltoztathatatlan értékek, és a hozzájuk tartozó érték bármilyen típusú lehet.

változtatható adatérték (mutable data value) Lásd módosítható adatérték.

20.8. Feladatok

- Írj egy olyan programot, amely beolvasson egy karakterláncot, és ábécé sorrendben visszaadja a karakterláncban előforduló betűk táblázatát, valamint az egyes betűk számát. A kis- és nagybetűket tekintsd egyformának. Amikor a felhasználó beírja a következő mondatot „Ez a Sztring Kis es Nagy Betuket tartalmaz” a program kimenete a következő:

```
a 5
b 1
e 4
g 2
i 2
k 2
l 1
m 1
n 2
r 2
s 3
t 4
u 1
y 1
z 3
```

- Add meg a Python értelmező válaszát az alábbi kódrészek mindegyikére:

(a)

```
d = {"alma": 15, "banán": 35, "szőlő": 12}
2 print(d["banán"])
```

(b)

```
d["narancs"] = 20
2 print(len(d))
```

(c)

```
print("szőlő" in d)
```

(d)

```
print(d["körte"])
```

(e)

```
print(d.get("körte", 0))
```

```
(f) gyumolcs = list(d.keys())
2     gyumolcs.sort()
3     print(gyumolcs)
```

```
(g)     del d["alma"]
2     print("alma" in d)
```

Győződj meg róla, hogy megértetted, miért kaptad ezeket az eredményeket. Ezután alkalmazd a megtanultakat az alábbi függvény testének megírására:

```
1     def plusz_gyumolcs(keszlet, gyumolcs, mennyiseg=0):
2         return
3
4     # Futasd ezeket a teszteket...
5     uj_keszlet = {}
6     plusz_gyumolcs(uj_keszlet, "eper", 10)
7     teszt("eper" in uj_keszlet)
8     teszt(uj_keszlet["eper"] == 10)
9     plusz_gyumolcs(uj_keszlet, "eper", 25)
10    teszt(uj_keszlet["eper"] == 35)
```

3. Írj egy `alice_words.py` nevű programot, amely egy `alice_words.txt` nevű szöveges fájlt hoz létre, mely tartalmazza az összes előforduló szó betűrendes felsorolását és darabszámát, az *Alice's Adventures in Wonderland* könyv szöveges verziójában. (A könyv ingyenes szöveges változata, valamint sok más szöveg elérhető a <http://www.gutenberg.org> címen.) A kimeneti fájl első 10 sorában ilyesmit kell látnod:

Szavak	Száma
=====	
a	631
a-piece	1
abide	1
able	1
about	94
above	3
absence	1
absurd	2

Hányszor fordul elő az `alice` szó a könyvben?

4. Mi a leghosszabb szó az *Alice in Wonderland*-ban? Hány karaktere van?

21. fejezet

Esettanulmány: A fájlok indexelése

Bemutatunk egy kis esettanulmányt, amely összekapcsolja a modulokat, rekurziót, fájlokat, szótárakat, és bevezetjük az egyszerű serializációt és deserializációt.

Ebben a fejezetben egy szótár használatával segítünk gyorsan megtalálni egy fájlt.

Az esettanulmánynak két komponense van:

- A kereső (*crawler*) program, amely átvizsgálja a lemezt (vagy mappát), és szerkeszti és elmenti a szótárt a lemezre.
- A lekérdező (*query*) program, amely betölti a szótárt, és gyorsan válaszol az olyan felhasználói kérdésekre, hogy hol található a fájl.

21.1. A kereső program

A rekurzióról szóló fejezet vége fele mutattunk egy példát arról, hogy hogyan lehet rekurzívan kilistázni a fájlokat a fájlrendszerünk egy adott útvonalán.

Ezt a kódot fogjuk használni a kereső programunk alapjaként, ezt fogjuk átírni. Ez a függvény rekurzívan bejárja a fájlokat egy megadott útvonalon. (Hamarosan meg fogjuk tudni, hogy mit csinálunk a fájlal: itt csak a rövid nevét és a teljes útját írjuk ki.)

```
1  # A kereső (Crawler) feltérképezi a fájlrendszert, és létrehoz egy szótárt
2  import os
3
4  def fajn_kereso(ut):
5      """ Rekurzívan járd be az összes fájlt a megadott útvonalon. """
6
7      # Add meg az aktuális mappában lévő összes bejegyzést.
8      mappa_lista = os.listdir(ut)
9      for f in mappa_lista:
10         # Alakítsd az egyes neveket elérési úttá.
11         teljes_nev = os.path.join(ut, f)
12
13         # Ha ez egy könyvtár, folytasd.
14         if os.path.isdir(teljes_nev):
15             fajn_kereso(teljes_nev)
16         else: # Csinálj valami hasznosat a fájlal.
17             print("{0:30} {1}".format(f, teljes_nev))
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
18
19 fajl_kereso("C:\\Python32")
```

Hasonló kimenetet kapunk ehhez:

CherryPy-wininst.log	C:\\Python32\\CherryPy-wininst.log
bz2.pyd	C:\\Python32\\DLLs\\bz2.pyd
py.ico	C:\\Python32\\DLLs\\py.ico
pyc.ico	C:\\Python32\\DLLs\\pyc.ico
pyexpat.pyd	C:\\Python32\\DLLs\\pyexpat.pyd
python3.dll	C:\\Python32\\DLLs\\python3.dll
select.pyd	C:\\Python32\\DLLs\\select.pyd
sqlite3.dll	C:\\Python32\\DLLs\\sqlite3.dll
tc185.dll	C:\\Python32\\DLLs\\tc185.dll
tcpip85.dll	C:\\Python32\\DLLs\\tcpip85.dll
tk85.dll	C:\\Python32\\DLLs\\tk85.dll
...	

Most arra fogjuk használni ezt a függvényt, hogy a rövid fájlneveket és a hozzájuk tartozó teljes elérési utat eltároljuk egy szótárban. De először két észrevétel:

- Számos azonos nevű fájl létezhet (a különböző útvonalakon). Például az *index.html* név meglehetősen gyakori. A szótár kulcsainak azonban egyedinek kell lenniük. A megoldásunk során a szótárunkban szereplő kulcsokat az útvonalak *listájára* képezzük le.
- A fájlnevek nem kis- és nagybetű érzékenyek. (Mármint a Windows felhasználók számára!) Tehát egy jó módszer, hogyha *normalizáljuk* a kulcsokat a tárolás előtt. Itt csak arról kell gondoskodunk, hogy az összes kulcsot kisbetűssé alakítsuk. Természetesen ugyanezt tesszük később is, amikor megírjuk a lekérdező programot.

A fenti kódot megváltoztatjuk egy globális szótár beállításával, amely eredetileg üres. A 3. sorba beillesztett `szotar = {}` utasítás fogja ezt megtenni. Ezután a 17. sorban lévő információk kiírása helyett hozzáadjuk a fájlnevet és az útvonalat a szótárhoz. Ellenőrizni kell, hogy a kulcs már létezik-e:

```
1 kulcs = f.lower() # A fájlnev normalizálása (kisbetűsítése).
2 if kulcs in szotar:
3     szotar[kulcs].append(teljes_nev)
4 else: # Szúrd be a kulcsot és az elérési útvonal listáját.
5     szotar[kulcs] = [teljes_nev]
```

A függvény hívása után ellenőrizhetjük, hogy a szótár helyesen lett-e felépítve:

```
1 print(len(szotar))
2 print(szotar["python.exe"])
3 print(szotar["logo.png"])
```

A kimenet:

```
14861

['C:\\Python32\\python.exe']

['C:\\Python32\\Lib\\site-packages\\PyQt4\\doc\\html\\_static\\logo.png',
 'C:\\Python32\\Lib\\site-packages\\PyQt4\\doc\\sphinx\\static\\logo.png',
 'C:\\Python32\\Lib\\site-packages\\PyQt4\\examples\\demos\\textedit\\images\\logo.png'
↪ ',
 'C:\\Python32\\Lib\\site-packages\\sphinx-1.1.3-py3.2.
↪ egg\\sphinx\\themes\\scrolls\\static\\logo.png']
```


Hasznos lenne egy állapotjelző, amely mutatja hol tart a program a futás során: egy tipikus módszer a pontok kiírása, hogy mutassa az előrehaladást. Bevezetünk egy számlálót a már indexelt fájlok számának nyilvántartására (ez lehet egy globális változó),). Szúrjuk be ezt a kódot az aktuális fájl feldolgozása után:

```
1 fajl_szamlalo += 1
2 if fajl_szamlalo % 100 == 0:
3     print(".", end="")
4     if fajl_szamlalo % 5000 == 0:
5         print()
```

Minden 100. fájl befejezése után kiírunk egy pontot. Minden 50 pont után új sort kezdünk. Létrehozunk egy globális változót is, inicializálni kell nullával, és ne felejtsük el a változót globálisan deklarálni a keresőben.

A fő program meghívja a kódot és kiír néhány statisztikát számunkra. A következőképpen:

```
1 fajl_kereso("C:\\Python32")
2 print() # A pontokat tartalmaó sor vége
3 print("{0} indexelt fájl, {1} bejegyzés a szótárban.".
4         format(fajl_szamlalo, len(szotar)))
```

Valami hasonlót kapunk:

```
.....
.....
.....
.....
18635 indexelt fájl, 14861 bejegyzés a szótárban.
```

Ellenőrzésként nézz rá az operációs rendszer mappájának tulajdonságaira, és észreveheted, hogy pontosan ugyanannyi fájlt számolt, mint a mi programunk!

21.2. A szótár lemezre mentése

A szótár, amit felépítettünk egy objektum. Ha el akarjuk menteni, akkor egy sztringbe fogjuk konvertálni és kiírjuk a sztringet a lemezünkre. A sztring olyan formátumú kell legyen, amely lehetővé teszi egy másik program számára, hogy egyértelműen rekonstruáljon egy másik szótárt ugyanolyan kulcs-érték elemekkel. Az objektum sztringként való ábrázolásának folyamatát **szerializációnak** nevezzük, és az inverz műveletet – egy objektum sztringből való rekonstruálását pedig – **deszerializációnak** nevezzük.

Van ennek néhány módja: egyesek bináris formátumokat használnak, mások pedig szövegformátumokat, és a különböző típusú adatok kódolása is különbözik. Egy népszerű, könnyű technika, melyet széles körben használnak a webszerverek és weboldalak, a JSON (JavaScript Object Notation) kódolás.

Meglepően csak négy új sor szükséges a szótárunk lemezünkre való mentéséhez:

```
1 import json
2
3 f = open("C:\\temp\\sajat_szotar.txt", "w")
4 json.dump(szotar, f)
5 f.close()
```

Megkeresheted a fájlt a lemezen, és megnyithatod egy szövegszerkesztővel annak érdekében, hogy lásd hogyan néz ki a JSON kódolás.

21.3. A lekérdező (Query) program

Ehhez rekonstruálni kell a szótárt a fájlból, majd biztosítani kell egy kereső függvényt:

```
1 import json
2
3 f = open("C:\\temp\\sajat_szotar.txt", "r")
4 szotar = json.load(f)
5 f.close()
6 print("{0} betöltött fájlnev a lekérdezésnél.".format(len(szotar)))
7
8 def lekerdezo(fajlnev):
9     f = fajlnev.lower()
10    if f not in szotar:
11        print("Nem található a {0}.".format(fajlnev))
12    else:
13        print("{0} itt található ".format(fajlnev))
14        for p in szotar[f]:
15            print("...", p)
```

És itt egy minta a futásra:

```
14861 betöltött fájlnev a lekérdezésnél.
```

További példák a futásra:

```
1 print(lekerdezo('python.exe'))
2 print(lekerdezo('java.exe'))
3 print(lekerdezo('INDEX.Html'))
```

A kimenet:

```
python.exe itt található
... C:\Python32\python.exe

Nem található a java.exe

INDEX.Html itt található
... C:\Python32\Lib\site-packages\cherrypy\test\static\index.html
... C:\Python32\Lib\site-packages\eric5\Documentation\Source\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\css\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\htmlmixed\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\javascript\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\markdown\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\python\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\rst\index.html
... C:\Python32\Lib\site-
→packages\IPython\frontend\html\notebook\static\codemirror\mode\xml\index.html
... C:\Python32\Lib\site-packages\pygame\docs\index.html
... C:\Python32\Lib\site-packages\pygame\docs\ref\index.html
... C:\Python32\Lib\site-packages\PyQt4\doc\html\index.html
```

21.4. A szerializált szótár tömörítése

A JSON fájl nagyon nagy is lehet. A Gzip tömörítési módszer elérhető a Pythonban, használjuk ki ezt az előnyét. . .

Amikor a szótárt a lemezre mentettük, megnyitottunk egy szöveges fájlt írására. Egyszerűen meg kell változtatnunk a program egy sorát (és be kell importálnunk a megfelelő modulokat), hogy létrehozzunk egy gzip fájlt a normál szöveges fájl helyett. Cseréljük a kódot erre

```
1 import json, gzip, io
2
3 ## f = open("C:\\temp\\sajat_szotar.txt", "w")
4 f = io.TextIOWrapper(gzip.open("C:\\temp\\sajat_szotar.gz", mode="wb"))
5 json.dump(szotar, f)
6 f.close()
```

Varázslatos módon most kaptunk egy tömörített fájlt, amely körülbelül 7-szer kisebb a szöveges változatnál. (Az ilyen tömörítő / kitömörítő műveleteket gyakran a webszerverek és a böngészők lehetővé teszik a gyorsabb letöltésekhez.)

Most természetesen a lekérdező programunknak ki kell tömörítenie az adatokat:

```
1 import json, gzip, io
2
3 ## f = open("C:\\temp\\sajat_szotar.txt", "r")
4 f = io.TextIOWrapper(gzip.open("C:\\temp\\sajat_szotar.gz", mode="r"))
5 szotar = json.load(f)
6 f.close()
7 print("{0} betöltött fájlnev a lekérdezésnél.".format(len(szotar)))
```

A komponálhatóság a kulcs...

A könyv korábbi fejezeteiben már beszéltünk a komponálhatóságról: mely az a képesség, hogy össze tudunk kapcsolni, vagy *kombinálni* a különböző kódrészeket és funkcionalitásokat, hogy egy erősebb konstrukciókat alkossunk.

Ez az esettanulmány kiváló példát mutatott erre. A JSON szerializáló és a deszerializáló kapcsolódhat a mi fájl mechanizmusunkkal. A gzip tömörítő / kitömörítő is megjelenhet a programunkban, mintha csak egy speciális adatfolyam lenne, amely egy fájl olvasásából származhat. A végeredmény egy nagyon elegánsan komponálható erőteljes eszköz. Ahelyett, hogy külön lépéseket kellene megtenni a szótár sztringé való szerializációjához, a sztring tömörítéséhez, az eredmény bájtok fájlba való írásához stb., a komponálhatóság lehetővé teszi számunkra, hogy mindezt nagyon egyszerűen megtehessek!

21.5. Szójegyzék

deszerializáció (deserialization) Valamilyen külső szöveg reprezentációjából származó memóriaobjektum rekonstrukciója.

gzip Veszteségmentes tömörítési eljárás, amely csökkenti az adat tárolásának méretét. (Veszteségmentes azt jelenti, hogy pontosan visszaállíthatja az eredeti adatokat.)

JSON A JavaScript Object Notation olyan objektumok szerializációja és szállítása, amelyet gyakran alkalmaznak a webszerverek és a JavaScript futtató webböngészők között. A Python tartalmaz egy `json` modult, amely ezt a lehetőséget biztosítja.

szerializáció (serialization) Egy objektum karakterláncba (vagy bájt sorozatba) történő mentése, hogy az interneten keresztül küldhető legyen, vagy el lehessen menteni egy fájlba. A címzett tudja rekonstruálni az objektumot az adatokból.

22. fejezet

Még több OOP

22.1. Az Ido osztály

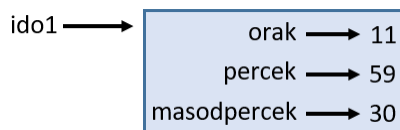
A saját típusok készítésének újabb példaként egy időpont, illetve időtartam tárolására alkalmas `Ido` osztályt fogunk létrehozni. Egy `__init__` metódus megadásával biztosítjuk, hogy minden elkészült objektumpéldány megfelelő attribútumokkal rendelkezzen, és megfelelően legyen inicializálva. Az osztály definíciója az alábbi:

```
1 class Ido:
2
3     def __init__(self, orak=0, percek=0, masodpercek=0):
4         """ Egy Ido objektum inicializálása az orak, percek, masodpercek
5         ↪ értékekre. """
6         self.orak = orak
7         self.percek = percek
8         self.masodpercek = masodpercek
```

Egy új `Ido` objektumot így példányosíthatunk:

```
1 ido1 = Ido(11, 59, 30)
```

Az objektumhoz tartozó állapotdiagram a következő:



Az `__str__` metódus elkészítését, ami lehetővé teszi, hogy az `Ido` objektumok megfelelően jeleníthessék meg magukat, az olvasókra hagyjuk.

22.2. Tiszta függvények

Az elkövetkező néhány alfejezetben két `Ido` objektum összegét meghatározó `ido_osszeadas` két változatát készítjük el, melyek egy-egy függvénytípust fognak demonstrálni: a tiszta függvényt és a módosítót.

Az alábbi kód az `ido_osszeadas` függvény nyers változata:

```
1 def ido_osszeadas(i1, i2):
2     orak = i1.orak + i2.orak
3     percek = i1.percek + i2.percek
4     masodpercek = i1.masodpercek + i2.masodpercek
5     ossz_ido = Ido(orak, percek, masodpercek)
6     return ossz_ido
```

A függvény egy új Ido objektumot készít, és visszaadja az új objektum referenciáját. Az ilyen függvényt **tiszta függvénynek** nevezzük, mert egyetlen paraméterként kapott objektumát sem módosítja, nincs mellékhatása. Nem írja felül például a globális változók értékét, nem jelenít meg és nem kér be értékeket a felhasználótól.

Itt egy példa a függvény használatára. Készítünk két Ido objektumot: az egyik az aktuális időt tartalmazó aktualis_ido, a másik a kenyersutes_idotartama, amely azt tárolja, hogy mennyi idő alatt készíti el a kenyérsütő a kenyeret, majd az ido_osszeadas függvénnyel kiszámoljuk, hogy mikor lesz készen a kenyér.

```
1 aktualis_ido = Ido(9, 14, 30)
2 kenyersutes_idotartama = Ido(3, 35, 0)
3 befejezes_ideje = ido_osszeadas(aktualis_ido, kenyersutes_idotartama)
4 print(befejezes_ideje)
```

A programunk kimenete 12:49:30, ami helyes. Vannak azonban olyan esetek is, amikor az eredmény helytelen lesz. Tudnál mondani egyet?

A probléma az, hogy a függvény nem foglalkozik azokkal az esetekkel, amikor a másodpercek vagy a percek összege eléri, vagy meghaladja a hatvanat. Ha ilyesmi történik, akkor a felesleges másodperceket a percekhez, a felesleges perceket pedig az órákhoz kell átvinnünk.

Itt egy jobb változat:

```
1 def ido_osszeadas(i1, i2):
2
3     orak = i1.orak + i2.orak
4     percek = i1.percek + i2.percek
5     masodpercek = i1.masodpercek + i2.masodpercek
6
7     if masodpercek >= 60:
8         masodpercek -= 60
9         percek += 1
10
11    if percek >= 60:
12        percek -= 60
13        orak += 1
14
15    ossz_ido = Ido(orak, percek, masodpercek)
16    return ossz_ido
```

Kezd a függvény megnőni, de még mindig nem fedi le az összes lehetséges esetet. A későbbiekben javasolni fogunk egy alternatív megközelítést, amely jobb kódhoz fog vezetni.

22.3. Módosító függvények

Bizonyos esetekben hasznos, ha a függvény módosít egy vagy több paraméterként kapott objektumot. Általában a hívó rendelkezik az átadott objektumok referenciájával, ezért a változások a hívó számára is láthatóak. Az ilyen **mellékhatással rendelkező függvényekre** a továbbiakban **módosító függvényekként** fogunk utalni.

Egy olyan `novel` függvényt, amely adott másodperccel növel egy `Ido` objektumot, kézenfekvő módosító függvényként megírni. A függvény elnagyolt vázlata valahogy így néz ki:

```
1 def novel(ido, masodpercek):
2     ido.masodpercek += masodpercek
3
4     if ido.masodpercek >= 60:
5         ido.masodpercek -= 60
6         ido.percek += 1
7
8     if ido.percek >= 60:
9         ido.percek -= 60
10        ido.orak += 1
```

Az első sor végzi el az alapvető műveletet, a többi sor pedig a korábban látott speciális eseteket kezeli.

Helyes-e ez a függvény? Mi történik, ha a `masodpercek` paraméter meghaladja a hatvanat? Ebben az esetben nem elég egyszer megvalósítani az átvitelt, addig kell folytatnunk, ameddig a `masodpercek` értéke hatvan alá nem csökken. Az egyik lehetséges megoldás, ha az `if` utasításokat `while`-ra cseréljük:

```
1 def novel(ido, masodpercek):
2     ido.masodpercek += masodpercek
3
4     while ido.masodpercek >= 60:
5         ido.masodpercek -= 60
6         ido.percek += 1
7
8     while ido.percek >= 60:
9         ido.percek -= 60
10        ido.orak += 1
```

Ez a függvény már helyesen működik, ha a `masodpercek` paraméter nem negatív, és az `orak` értéke nem haladja meg a 23-at, de nem kimondottan jó megoldás.

22.4. Alakítsuk át a `novel` függvényt metódussá

Az OOP programozók az `Ido` objektummal dolgozó függvényeket jobb szeretik az `Ido` osztályon belül látni, szóval alakítsuk át a `novel` függvényt metódussá. A helytakarékoság érdekében kihagyjuk a korábban definiált metódusokat, de a saját változatodban azokat is őrizd meg:

```
1 class Ido:
2     # Itt állnak a korábban definiált metódusok...
3
4     def novel(self, masodpercek):
5         self.masodpercek += masodpercek
6
7         while self.masodpercek >= 60:
8             self.masodpercek -= 60
9             self.percek += 1
10
11        while self.percek >= 60:
12            self.percek -= 60
13            self.orak += 1
```

Az átalakítás mechanikusan elvégezhető: a függvény definíciót áttesszük az osztály definíciójába, és kicseréljük az első paramétert `self`-re a Python elnevezési konvenciójának megfelelően. (Az utóbbi nem kötelező.)

Most már a metódushívásoknak megfelelő szintaktikával hívható a `novel`.

```
1 aktualis_ido.novel(500)
```

A metódus első paraméteréhez, a `self`-hez az az objektum rendelődik hozzá, amelyikre a metódust meghívtuk. A második paraméter, a `masodpercek`, 500-as értéket kap.

22.5. Egy „aha-élmény”

A programozást gyakran megkönnyíti, ha nem merülünk el a részletekben, hanem távolról szemléljük a problémát.

A meglepő felismerés ebben az esetben az lehet, hogy az `Ido` objektum valójában egy három számjegyből álló, 60-as számrendszerbeli szám. A `masodpercek` helyi értéke 1-es, a `percek` helyi értéke 60-as, az óráké pedig 3600-as.

Amikor az `ido_osszeadas` és a `novel` függvényeket írtuk, akkor valójában 60-as számrendszerben végeztünk összeadást, ezért kellett az átviteleket kezelnünk.

A megfigyelés alapján máshonnan is közelíthetünk a problémához. Ha az `Ido` objektumot egyetlen számmá konvertáljuk, akkor kihasználhatjuk, hogy a számítógép képes aritmetikai műveleteket végezni a számokon. Az alábbi, `Ido` osztályhoz adott metódus a példányok által reprezentált időt át tudja váltani másodpercekre:

```
1 class Ido:
2     # ...
3
4     def masodpercre_valtas(self):
5         """ A példány által reprezentált másodpercek számával tér vissza.
6         """
7         return self.orak * 3600 + self.percek * 60 + self.masodpercek
```

Most már csak arra van szükségünk, hogy vissza is tudjuk alakítani az egész számokat `Ido` objektumokká. Feltételezve, hogy `osszes_masodperc` másodpercünk van, néhány egész és maradékos osztással meg is oldhatjuk ezt:

```
1 orak = osszes_masodperc // 3600
2 fennmarado_masodpercek = osszes_masodperc % 3600
3 percek = fennmarado_masodpercek // 60
4 masodpercek = fennmarado_masodpercek % 60
```

Egy kis gondolkodással meggyőződhetsz az alapok közti átváltás helyességéről.

Az OO programozás során valóban megpróbáljuk egybecsomagolni, összeszervezni az adatokat és a rajtuk operáló műveleteket, így azt szeretnénk, ha az előbbi konvertáló az `Ido` osztályon belülre kerülne. Jó megoldás lehet, ha úgy írjuk át az osztály inicializálóját, hogy megbirkózzon a **normalizálatlan** értékekkel is. (Normalizált érték például a 3 óra 12 perc 20 másodperc. Ugyanazt az időpontot írja le, de normalizálatlan a 2 óra 70 perc 140 másodperc.)

Írjuk át hatékonyabbra az `Ido` osztály inicializálóját:

```
1 class Ido:
2     # ...
3
4     def __init__(self, orak=0, percek=0, masodpercek=0):
5         """ Egy Ido objektum inicializálása az orak, percek, masodpercek_
6         ↪értékekre.
7         A percek és masodpercek értéke kívül eshet a 0-59 tartományon,
8         de az eredményként kapott Ido objektum normalizált lesz.
9         """
10        # Az összes masodperc számítása a reprezentációhoz
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
11     osszes_masodperc = orak*3600 + percek*60 + masodpercek
12     self.orak = osszes_masodperc // 3600
13     fennmarado_masodpercek = osszes_masodperc % 3600
14     self.percek = fennmarado_masodpercek // 60
15     self.masodpercek = fennmarado_masodpercek % 60
```

Most már átírhatjuk az `ido_osszeadas` függvényt is, valahogy így:

```
1 def ido_osszeadas(i1, i2):
2     masodpercek = i1.masodpercre_valtas() + i2.masodpercre_valtas()
3     return Ido(0, 0, masodpercek)
```

Ez a változat sokkal rövidebb, mint az eredeti, és jóval könnyebb demonstrálni vagy igazolni, hogy helyesen működik.

22.6. Általánosítás

Bizonyos szempontból 60-as alapról áttérni 10-es alapra és vissza, nehezebb, mint egyszerűen csak kezelni az időket. A számrendszerek közti átváltás absztraktabb, az intuíciónk jobban működik, amikor idővel dolgozunk.

Ha viszont rájövünk, hogy az időpontokra 60-as számrendszerbeli számokként is tekinthetünk, és rááldozzuk az időt a konvertálók megírására, akkor rövidebb, olvashatóbb és könnyebben debugolható programot kapunk, ami megbízhatóbb is lesz.

Szintén könnyebbé válik az új funkciók hozzáadása. Képzeljük el például, hogy két `Ido` objektumot vonunk ki egymásból a közöttük lévő idő meghatározása érdekében. A naiv megközelítés a kivonást implementálná átvitelekkel. A konvertáló függvény felhasználásával egyszerűbb lenne a függvényünk, ezért az is valószínűbb, hogy helyesen működne.

Ironikus módon, néha a probléma bonyolultabbá tétele (általánosítása) teszi egyszerűbbé a programozást, mert kevesebb speciális eset és kevesebb hibalehetőség adódik.

Specializáció vs. Generalizáció

A programozók általában típusok specializálásának hívei, míg a matematikusok gyakran az ellenkező megközelítést követik, és mindent általánosítanak.

Mit értünk ez alatt?

Ha egy matematikust kérünk meg a hétköznapiakkal, a század napjaival, kártyajátékokkal, időpontokkal vagy dominókkal kapcsolatos probléma megoldására, akkor igen valószínű, hogy azt a választ kapjuk, hogy ezen objektumok mindegyike reprezentálható számokkal. Például a kártyák számozhatók 0-tól 51-ig. A századon belüli napok szintén sorszámozhatók. A matematikusok azt fogják mondani, hogy „Ezek a dolgok felsorolhatóak, az elemeknek egyedi sorszám adható (és a sorszám alapján visszakaphatjuk az eredeti elemet). Szóval számozzuk be őket, és korlátozzuk az egész számokra a gondolkodásunkat. Szerencsére hathatós technikáink vannak az egész számok kezelésére, jól értjük őket, ezért az absztrakciónk – ahogyan kezeljük és egyszerűsítjük ezeket a problémákat – az, hogy az egész számok halmazára vezetjük vissza a feladatokat.”

A programozók az ellenkező irányba tendálnak. Azzal érvelnék, hogy nagyon sok olyan, az egész számok körében alkalmazható művelet van, amelyeknek semmi értelme a dominókra vagy az évszázad napjaira nézve. Azért definiálunk gyakran új, specializált típusokat, mint az `Ido`, mert így korlátozhatjuk, ellenőrizhetjük és specializálhatjuk a lehetséges műveletek körét. Az objektumorientált programozás főképp azért népszerű, mert jó módszert ad a metódusok és specializált adatok új típusokba való összeszervezésére.

Mindkét megközelítés hatékony problémamegoldó módszer. Sokszor segíthet, ha megpróbáljuk mindkét nézőpontból átgondolni a problémát: „Mi történne, ha megpróbálnék mindent visszavezetni néhány primitív típusra?” vs. „Mi

történne, ha saját típust vezetnék be ennek a dolognak a leírására?”

22.7. Egy másik példa

A `kesobb_van_e` függvénynek két időpontot kell összehasonlítani és meghatározni, hogy az első időpont szigorúan később van-e, mint a második. Az alábbi kódrészlet tehát a `True` kimenetet adná.

```
1 i1 = Ido(10, 55, 12)
2 i2 = Ido(10, 48, 22)
3 k = kesobb_van_e(i1, i2) #Később van-e az i1, mint az i2?
4 print(k)
```

Egy picit bonyolultabb, mint az előző példa, hiszen egy helyett két `Ido` objektummal dolgozunk. Természetesen metódusként akarjuk megírni, ebben az esetben az első argumentum metódusaként:

```
1 class Ido:
2     # Itt állnak a korábban definiált metódusok...
3
4     def kesobb_van_e(self, ido2):
5         """ Igazzal tér vissza, ha szigorúan nagyobb vagyok, mint az ido2. """
6         ↪
7         if self.orak > ido2.orak:
8             return True
9         if self.orak < ido2.orak:
10            return False
11
12        if self.percek > ido2.percek:
13            return True
14        if self.percek < ido2.percek:
15            return False
16        if self.masodpercek > ido2.masodpercek:
17            return True
18
19        return False
```

Egy objektumra meghívjuk a metódust, egyet pedig argumentumként adunk át neki:

```
1 if aktualis_ido.kesobb_van_e(befejezes_ideje):
2     print("A kenyér kész lesz, mielőtt elkezdenénk sütni!")
```

A metódus hívása hasonlít egy magyar mondatra: „Az aktuális idő később van-e, mint a befejezés ideje?”.

Az `if` utasítás különös figyelmet érdemel. A 11-18. sorokat csak akkor éri el a vezérlés, ha a két `ora` attribútum azonos. Hasonlóan, a 15. sorban álló vizsgálat csak akkor hajtódik végre, ha az objektumok `ora` és a `perc` attribútumai is megegyeznek.

Egyszerűbbé tehetjük-e, a metódust a korábbi felismerésünkre és plusz munkánkra támaszkodva, ha egész számokká alakítjuk az időket? Igen, méghozzá látványosan!

```
1 class Ido:
2     # Itt állnak a korábban definiált metódusok...
3
4     def kesobb_van_e(self, ido2):
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
5         """ Igazzal tér vissza, ha szigorúan nagyobb vagyok, mint az ido2. """  
6         return self.masodpercre_valtas() > ido2.masodpercre_valtas()
```

Ez egy remek módszer a probléma kódolására. Ha szeretnénk megtudni, hogy az első időpont később van-e mint a második, akkor alakítsuk át mind a két időpontot egész számmá, és azokat hasonlítsuk össze.

22.8. Operátorok túlterhelése

Néhány nyelv – a Pythont is beleértve – megengedi, hogy ugyanaz az operátor más-más típusú operandusokra alkalmazva különböző jelentéssel bírjon. Például a + Pythonban teljesen mást jelent, ha egész számokra vagy ha sztringekre alkalmazzuk. Ezt nevezzük **operátor túlterhelésnek**.

Különösen hasznos, ha a programozó is túlterhelheti az operátorokat a saját típusoknak megfelelően.

Például + operátor túlterhelése érdekében egy `__add__` metódust kell megírunk:

```
1 class Ido:  
2     # Itt állnak a korábban definiált metódusok...  
3  
4     def __add__(self, masik):  
5         return Ido(0, 0, self.masodpercre_valtas() + masik.masodpercre_  
        ↪ valtas())
```

Az első paraméter szokás szerint az az objektum, amelyre a metódust meghívjuk. A második paramétert egyszerűen `masik`-nak nevezzük, hogy megkülönböztessük a `self`-től. A két `Ido` objektum összegét egy új `Ido` objektumba tároljuk el, ezt adjuk vissza.

Innentől kezdve, ha `Ido` objektumokra alkalmazzuk a + operátort, akkor az általunk készített `__add__` metódust hívja meg a Python:

```
1 i1 = Ido(1, 15, 42)  
2 i2 = Ido(3, 50, 30)  
3 i3 = i1 + i2  
4 print(i3)
```

Kimenetként a 05:06:12 jelenik meg.

Az `i1 + i2` kifejezés ekvivalens az `i1.__add__(i2)` kifejezéssel, de az előbbi nyilvánvalóan elegánsabb. Ön-álló feladatként készíts `__sub__(self, masik)` metódust, ami a kivonást terheli túl! Próbáld is ki!

A következő néhány példában az első objektumokkal foglalkozó fejezetben definiált `Pont` osztály néhány operátort fogjuk túlterhelni. Először is, két pont összeadása jelentse a megfelelő koordinátáik összeadását:

```
1 class Pont:  
2     # Itt állnak a korábban definiált metódusok...  
3  
4     def __add__(self, masik):  
5         return Pont(self.x + masik.x, self.y + masik.y)
```

A szorzás operátort több módon is felülírhatjuk: definiálhatunk egy `__mul__` vagy egy `__rmul__` nevű metódust is, vagy akár mind a kettőt.

Ha a * balján álló operandus egy `Pont` objektum, akkor a Python a `__mul__`-t hívja meg, feltételezve, hogy a másik operandus is egy `Pont`. A `__mul__` az alábbi módon definiálva a két pont **belső szorzatát** állítja elő a lineáris algebra szabályainak megfelelően:

```
1 def __mul__(self, másik):  
2     return self.x * másik.x + self.y * másik.y
```

Ha a `*` bal oldalán álló operandus primitív típusú, a jobb oldali pedig egy `Pont` objektum, akkor a Python az `__rmul__` metódust hívja meg. Az alábbi definíció alapján **skalárral való szorzást** végez:

```
1 def __rmul__(self, másik):  
2     return Pont(masik * self.x, másik * self.y)
```

Az eredmény egy új `Pont` objektum, melynek a koordinátái az eredeti koordináták valahányszorosai. Ha a `masik` típusa nem támogatja a valós számmal való szorzást, akkor az `__rmul__` hibát eredményez.

Az alábbi sorok mindkét szorzásra adnak példát:

```
1 p1 = Pont(3, 4)  
2 p2 = Pont(5, 7)  
3 print(p1 * p2)  
4 print(2 * p2)
```

Az első `print` utasítás a 43, míg a második a (10, 14) kimenetet adja.

Mi történik a `p2 * 2` kifejezés kiértékelése során? A Python a `__mul__` metódust hívja meg, hiszen a bal oldali operandus egy `Pont`. A metódus első argumentuma a `Pont` objektum lesz, a második pedig a 2-es érték. Amikor a `__mul__`-on belül a program megpróbál hozzáférni a `masik` paraméter `x` koordinátájához, hiba lép fel, hiszen egy `int` típusú értékeknek nincsenek attribútumai.

```
1 print(p2 * 2)
```

A kapott hibaüzenet sajnos nem teljesen egyértelmű, ami rámutat az objektumorientált programozás egy-két nehézségére. Néha bizony még azt is nehéz kitalálni, hogy melyik az éppen futó kódrészlet.

```
AttributeError: 'int' object has no attribute 'x'
```

22.9. Polimorfizmus

Az általunk készített metódusok többsége csak egy meghatározott típusra működik. Ha egy új típusú objektumot definiálunk, akkor a hozzá tartozó műveleteket is meg kell írunk.

Vannak azonban olyan műveletek is, amelyeket különböző típusú objektumokra is használni szeretnénk, például az előző fejezetben látott aritmetikai operátorok. Ha több típus is támogatja ugyanazt a művelethalmazt, akkor készíthetünk olyan függvényt, amelyik ezen típusok mindegyikére működik.

Például a `szorzat_plusz` függvénynek három paramétere van. Az első két paraméterét összeszorozza, majd hozzáadja a harmadikat. (A lineáris algebrában gyakran van szükség erre.) Pythonban így valósíthatjuk meg:

```
1 def szorzat_plusz(x, y, z):  
2     return x * y + z
```

Ez a függvény minden olyan `x` és `y` értékre működik, amelyek közt értelmezett a szorzás művelet, és bármilyen olyan `z` értékre, amely a szorzathoz hozzáadható.

Meghívhatjuk számokkal:

```
1 print(szorzat_plusz(3, 2, 1))
```

Ebben az esetben az eredmény is egy szám lesz, a 7. Adhatunk át `Pont` objektumokat is a függvénynek:

```
1 p1 = Pont(3, 4)
2 p2 = Pont(5, 7)
3 print(szorzat_plusz(2, p1, p2))
4 print(szorzat_plusz(p1, p2, 1))
```

Az első esetben a `Pont` objektumot egy skalárral szorozzuk, majd utána adunk hozzá egy újabb `Pont` objektumot. A második esetben a belső szorzat egy szám típusú értéket ad eredményként, ezért a harmadik paraméternek is egy számnak kell lennie. Ennek megfelelően a végeredmények típusa is eltérő:

```
(11, 15)
44
```

Az ehhez hasonló, többféle típusú argumentum fogadására is képes függvényeket **polimorf** függvénynek hívjuk.

Nézzünk egy másik példát. Képzeljünk el egy `elore_es_hatra` függvényt, amelyik előbb előről hátra, majd hátulról előre haladva ír ki egy listát:

```
1 def elore_es_hatra(elore):
2     import copy
3     hatra = copy.copy(elore)
4     hatra.reverse()
5     print(str(elore) + str(hatra))
```

A `reverse` metódus egy módosító, ezért egy másolatot készítünk az objektumról, mielőtt megfordítanánk vele a listát, hogy a függvényünk ne változtassa meg a paraméterként kapott listát.

Itt egy olyan példa, amikor az `elore_es_hatra` függvényt egy listára alkalmazzuk:

```
1 lista = [1, 2, 3, 4]
2 forditott_lista = elore_es_hatra(lista)
3 print(lista, forditott_lista)
```

A kimenet a várakozásunknak megfelelő:

```
[1, 2, 3, 4] [4, 3, 2, 1]
```

Mivel eleve egy listát szándékoztunk átadni a függvénynek, nem lep meg bennünket, hogy működik. Ha a `Pont` objektumokra is használhatnánk, az már meglepetés lenne.

A Python nyelv polimorfizmusra vonatkozó alapszabályával, az úgynevezett **kacsa-teszt**tel meghatározhatjuk, hogy a függvény működik-e más típusú eszközökre is. A szabály azt mondja, hogy *ha a függvényen belül álló összes művelet végrehajtható az adott típusú programozási eszközökön, akkor maga a függvény is végrehajtható rajtuk*. Az `elore_es_hatra` függvény a `copy`, `reverse` és `print` műveleteket tartalmazza.

Nem minden programozási nyelv definiálja ilyen módon a polimorfizmust. Nézz utána a *kacsa-teszt*nek! Lássuk, ki tudod-e találni, miért pont ez a neve!

A `copy` minden objektumra működik, az `__str__` metódust már megírtuk a `Pont` objektumokra, már csak egy `reverse` metódusra lenne szükség a `Pont` osztályon belül:

```
1 def reverse(self):
2     (self.x, self.y) = (self.y, self.x)
```

Ha ez megvan, akkor már `Pont` objektumokat is átadhatunk az `elore_es_hatra` függvénynek:

```
1 p = Pont(3, 4)
2 p2 = elore_es_hatra(p)
3 print(p, p2)
```

A kód a (3, 4) (4, 3) kimenetet adja.

A legérdekesebb esetek azok, amikor nem is szándékos a polimorfizmus, csak később felfedezzük fel, hogy az általunk írt függvény olyan típusok esetében is működik, amelyekre nem is terveztük.

22.10. Szójegyzék

belső szorzat (dot product) A lineáris algebra egy művelete. Két `Pont` szorzata egy skalárt eredményez.

funkcionális programozási stílus (functional programming style) Egy olyan programozási stílus, amelyben a függvények többsége nem rendelkezik mellékhatással.

módosító (modifier) Azokra a függvényekre vagy eljárásokra utalunk vele, amelyek megváltoztatják egy vagy több argumentumként kapott objektumuk értékét. A legtöbb ilyen függvény `void` típusú, vagyis nem ad vissza értéket.

normalizált (normalized) Az adatokat normalizálnak nevezzük, ha egy előre meghatározott tartományra redukáljuk az értékeket. A szöveget általában a [0..360) tartományra normalizáljuk. A percek és másodperceket pedig úgy, hogy az értékeik a [0..60) tartományba essenek. Meglepődnénk, ha a sarki kisbolt ablakában a „nyitás 7 óra 85 percor” kiírást olvasnánk.

operátor túlterhelés (operator overloading) A beépített operátorok (+, -, *, >, <, stb.) működésének kiterjesztése oly módon, hogy különböző típusú argumentumokra is működjenek. A könyv eleje felé láttuk, hogy a + operátor sztringekre és számokra is működik, ebben a fejezetben pedig megmutattuk, hogyan terhelhetjük túl úgy a + operátort, hogy azt a saját (felhasználói) típusokra is használni lehessen.

polimorf (polymorphic) Azokat a függvényeket, amelyek többféle típusú argumentumokra is működnek polimorf függvényeknek nevezzük. Figyeld meg a különbséget: a túlterhelés esetében több, különböző típusú objektumokon operáló, de azonos nevű függvényünk van, míg a polimorf függvény egyetlen függvény, amely többféle típusú objektumra is működik.

skalárral való szorzás (scalar multiplication) A lineáris algebra egyik művelete. A műveletben résztvevő `Pont` minden koordinátáját ugyanazon skalár értékkel szorozzuk meg.

tiszta függvény (pure function) Azon függvények, amelyek egyetlen paraméterként kapott objektumukat sem módosítják (és nincs más mellékhatásuk sem). A legtöbb tiszta függvény rendelkezik visszatérési értékkel.

22.11. Feladatok

- Írj egy `kozte_van_e` logikai függvényt, amely három `Ido` objektumot vár paraméterként (`obj`, `i1`, `i2`), és `True` értéket ad vissza, ha az első paraméterben kapott időpont a másik kettő közé esik. Feltételezhető, hogy az `i1 <= i2`. Az időpontok által meghatározott tartományt aluról zártnak, felülről nyitottnak tekintjük, tehát akkor térjen vissza igazgal a függvény, ha az alábbi kifejezés teljesül: `i1 <= obj < i2`.
- Alakítsd át `Ido` osztálybeli metódussá az előbbi függvényt!
- Érd el a megfelelő operátor(ok) túlterhelésével, hogy az:

```
if i1.kesobb_van_e(i2): ...
```

helyett az alábbi, kényelmesebb jelölést használhassuk:

```
if i1 > i2: ...
```

- Írd át a `novel` metódust úgy, hogy kihasználja az idővel kapcsolatos, „aha-élményt” adó felfedezésünket!

5. Készíts néhány tesztet a `novel` metódushoz! Koncentrálj arra az esetre, amikor az időhöz hozzáadandó másodpercek száma negatív! Ha a `novel` most nem kezeli ezeket az eseteket, akkor javítsd ki! (Feltételezheted, hogy soha nem fogsz több másodpercet kivonni, mint amennyit az aktuális idő tartalmaz.)
6. Lehet a fizikai idő negatív? Vagy az idő mindig előrefele halad? Komoly fizikusok is akadnak, akik ezt nem tartják idióta kérdésnek. Nézz utána az interneten a témának!

23. fejezet

Objektumok kollekciója

23.1. Kompozíció

Mostanra számos példát láttunk kompozíciókra. Az egyik példa a kifejezés részeként történt metódushívás volt. Egy másik példa a beágyazott utasítás szerkezet volt: egy `if` utasítást elhelyezhetünk egy `while` cikluson belül vagy egy másik `if` utasításon belül, és így tovább.

Látva ezeket a példákat és tanulva a listákról és objektumokról, nem lepődnünk meg, hogy létrehozhatunk egy objektumokat tartalmazó listát. Listát (attribútumként) tartalmazó objektumokat is létrehozhatunk, vagy listákat tartalmazó listákat esetleg objektumokat tartalmazó objektumot, stb.

Ebben és a következő fejezetben ezekre a kompozíciókra láthatunk néhány példát egy `kartya` objektumot használva mintaként.

23.2. Kartya objektumok

Ha nem vagy jártas a francia kártyában, itt az idő, hogy szerezz egy paklit, különben ennek a fejezetnek nem lesz sok értelme. 52 kártyalap van egy pakliban, az összes lap a négy szín egyikéhez és a tizenhárom érték egyikéhez tartozik. A színek: pikk, kőr, káró és treff (a bridge nevű játékban ez a csökkenő sorrendjük). Az értékek sora a következő: ász, 2, 3, 4, 5, 6, 7, 8, 9, 10, bubi, dáma, király. Az adott játéktól függően az ász értékesebb lehet a királynál vagy gyengébb a 2-nél. Az értékeket gyakran (rang)soroknak is nevezik.

Ha definiálni akarunk egy új objektumot a kártyalapok reprezentálására, nyilvánvaló milyen attribútumoknak kell lennie: `szin` és `ertek`. Az már nem annyira nyilvánvaló, hogy ezek milyen típusúak. Az egyik lehetőség a sztringek használata, amely szavakat tartalmaz, mint a `"pikk"` a szín esetén és a `"dáma"` a rang esetén. Ennek az implementációnak az a hibája, hogy nem egyszerű összehasonlítani melyiknek erősebb a színe vagy nagyobb az értéke.

Egy alternatíva az, ha egészeket használva **kódoljuk** a színeket és az értékeket. A kódolás itt nem azt jelenti, amire sok ember gondol, nem titkosításról van szó. Amit az informatikus ért a kódolás alatt az nem más, mint definiálni egy leképezést számsorozatok és a megjeleníteni kívánt elemek között. Például:

```
pikk --> 3
kőr   --> 2
káró  --> 1
treff --> 0
```

Egy nyilvánvaló tulajdonsága ennek a leképezésnek az, hogy a színek sorrendben vannak számokra képezve, így összehasonlíthatjuk a színeket egész számok összevetésével. Az értékek leképezése elég nyilvánvaló, minden numerikus érték a megfelelő egész számra van leképezve és a figurák pedig így:


```
bubi --> 11
dáma --> 12
király --> 13
```

Az ok, ami miatt ezt a matematikai jelölést használjuk a leképezéshez az az, hogy a lapok nem részei a Python programoknak. Ezek a program terv részei, de nem jelenek meg explicit módon a kódban. A `Kartya` osztály definíciója így néz ki:

```
1 class Kartya:
2     def __init__(self, szin=0, ertek=0):
3         self.szin = szin
4         self.ertek = ertek
```

Szokás szerint gondoskodunk egy inicializáló metódusról, amelynek egy-egy opcionális paramétere van az attribútumok számára.

Hogy létrehozzunk objektumokat, mondjuk a treff 3-ast és a káró bubit, használjuk ezeket a parancsokat:

```
1 treff_3 = Kartya(0, 3)
2 kartya1 = Kartya(1, 11)
```

A fenti esetben például az első paraméter a 0 a treff szintet reprezentálja.

Mentsd el ezt a kódot későbbi használatra ...

A következő fejezetben feltételezni fogjuk, hogy már van egy elmentett `Kartya` és egy `Pakli` osztályunk is (az utóbbit hamarosan láthatjuk) egy `Kartyak.py` nevű fájlban.

23.3. Osztály attribútumok és az `__str__` metódus

Azért hogy kiíráshassuk a `Kartya` objektumokat az ember számára könnyen olvasható módon, le akarjuk képezni az egész típusú kódokat szavakká. A természetes módja ennek az, hogy sztringek listáját használjuk. Hozzárendeljük ezeket a listákat az osztálydefiníció elején lévő **osztály attribútumokhoz**:

```
1 class Kartya:
2     szinek = ["treff", "káró", "kőr", "pikk"]
3     ertekek = ["Pista", "ász", "2", "3", "4", "5", "6", "7",
4               "8", "9", "10", "bubi", "dáma", "király"]
5
6     def __init__(self, szin=0, ertek=0):
7         self.szin = szin
8         self.ertek = ertek
9
10    def __str__(self):
11        return (self.szinek[self.szin] + " " + self.ertekek[self.ertek])
```

Az osztály attribútumok a metódusokon kívül lettek definiálva és bármely metódusból elérhetőek.

A `__str__` metóduson belül használhatjuk a `szinek` és `ertekek` listákat, amelyekkel a `szin` és `ertek` változók numerikus értékeit képezhetjük le sztringekre. Például a `self.szinek[self.szin]` kifejezés azt jelenti, hogy a `self` objektum `szin` attribútuma a `szinek` nevű osztály attribútum indexeként kiválasztja a megfelelő sztringet.

Az ok, ami miatt `Pista` az első eleme az `ertekek` listának az, hogy helyőrző szerepet játszik a lista nulladik elemeként, ami sohasem lesz használva. Az érvényes értékek 1 és 13 között mozognak. Ez az elpazarolt elem nem

igazán szükséges. Kezdhethetnénk nullával szokás szerint, de így kevésbé félreérthető, ha a 2-es kártyát a 2 egész számra képezzük, a 3-ast 3-ra, stb.

Az eddigi metódusokkal létrehozhatjuk és kírathatjuk a kártyákat:

```
1 kartya1 = Kartya(1, 11)
2 print(kartya1)
```

A kimenet a káró bubi kifejezést tartalmazza.

Az osztály attribútumokon, mint a színek listán az összes Kartya objektum osztozkodik. Ennek az előnye, hogy bármely Kartya objektum elérheti az osztály attribútumokat:

```
1 kartya2 = Kartya(1, 3)
2 print(kartya2)
3 print(kartya2.színek[1])
```

Az első print a káró 3 szöveget írja ki, míg a második csak a káró szót.

Mivel minden Kartya példány ugyanarra az osztály attribútumra hivatkozik, azért egy fedőnév szituációval állunk szemben. A hátránya ennek az, hogy ha módosítjuk az osztály attribútumokat az minden példányra hatással lesz. Például, ha úgy döntünk, hogy a káró bubit inkább hívjuk tevepúp bubinak, akkor ezt tehetjük:

```
1 kartya1.színek[1] = "tevepúp"
2 print(kartya1)
```

A probléma az, hogy az összes káró tevepúppá válik:

```
1 print(kartya2)
```

Így a tevepúp 3 kifejezést látjuk a kimeneten.

Rendszerint nem jó ötlet megváltoztatni az osztály attribútumokat.

23.4. Kártyák összehasonlítása

A primitív típusok számára hat relációs operátor van (<, >, ==, stb.), amelyek összehasonlítják az értékeket, és meghatározzák, hogy az egyik érték kisebb, nagyobb vagy egyenlő a másikkal. Ha azt akarjuk, hogy a saját típusunk összehasonlítható legyen ezeknek a relációs operátoroknak a szintaxisával, akkor definiálnunk kell hat megfelelő speciális metódust az osztályunkban.

Egy szimpla metódussal szeretnénk kezdeni, amelynek a neve `hasonlitas` és magába foglalja a rendezés logikáját. Megállapodás szerint az összehasonlító metódus két paramétert kap `self` és `masik` néven, és 1-gyel tér vissza, ha az első objektum a nagyobb, -1 értékkel, ha a második a nagyobb, és 0-t ad, ha egyenlők.

Néhány típus teljesen rendezett, ami azt jelenti, hogy bármely két értékről megmondhatjuk, hogy melyik a nagyobb. Például az egész és lebegőpontos számok teljesen rendezettek. Néhány típus nem rendezett, ez azt jelenti nincs értelmes módja annak, hogy megmondjuk melyik érték nagyobb. Például a gyümölcsök rendezetlenek, ez az, ami miatt nem hasonlíthatjuk össze az almát a banánnal, és értelmesen nem tudjuk rendezni a képek vagy mobiltelefonok kollekcióját.

A kártyalapok részben rendezettek, ami azt jelenti, hogy néha össze tudjuk hasonlítani a lapokat, néha nem. Például tudjuk, hogy a treff 3 nagyobb, mint a treff 2, és a káró 3 nagyobb, mint a treff 3. Azonban melyik az erősebb a treff 3 vagy a káró 2? Az egyiknek a színe értékesebb, a másiknak az értéke nagyobb.

Hogy összehasonlíthatóvá tegyük a kártyákat, el kell döntenünk, hogy mi a fontosabb, a szín vagy az érték. Őszintén, a választás önkényes. A választás kedvéért azt fogjuk mondani, hogy a szín fontosabb, mert egy vadonatúj pakliban először a treffek vannak rendezve, aztán kárók, és így tovább.

Ezzel a döntéssel megírhatjuk a hasonlítás metódust:

```
1 def hasonlitas(self, másik):
2     # Ellenőrizd a színt
3     if self.szin > másik.szin: return 1
4     if self.szin < másik.szin: return -1
5     # A színek azonosak... ellenőrizd az értéket
6     if self.ertek > másik.ertek: return 1
7     if self.ertek < másik.ertek: return -1
8     # Az értékek is azonosak... azonosak
9     return 0
```

Ebben a rendezésben az ászok kisebb értékűek, mint a kettesek.

Most definiálhatunk hat speciális metódust, amelyek túlterhelik az egyes relációs operátorokat számunkra:

```
1 def __eq__(self, másik):
2     return self.hasonlitas(másik) == 0
3
4 def __le__(self, másik):
5     return self.hasonlitas(másik) <= 0
6
7 def __ge__(self, másik):
8     return self.hasonlitas(másik) >= 0
9
10 def __gt__(self, másik):
11     return self.hasonlitas(másik) > 0
12
13 def __lt__(self, másik):
14     return self.hasonlitas(másik) < 0
15
16 def __ne__(self, másik):
17     return self.hasonlitas(másik) != 0
```

Ezzel a mechanizmussal a relációs operátorok most úgy működnek, ahogy szeretnénk:

```
1 kartya1 = Kartya(1, 11)
2 kartya2 = Kartya(1, 3)
3 kartya3 = Kartya(1, 11)
4 kartya1 < kartya2
5 kartya1 == kartya3
```

Az előbbi feltétel hamis, míg az utóbbi igaz.

23.5. Paklik

Most hogy vannak Kartya objektumaink, a következő logikai lépés a Pakli osztály definiálása. Természetesen a pakli kártyákból áll, így a Pakli objektum kártyák listáját fogja tartalmazni attribútumként. Sok kártyajátékban két különböző paklira van szükség – egy kék és egy piros paklira.

Következik a Pakli osztály definíciója. Az inicializáló metódus létrehozza a kartyak attribútumot, és generál egy szabvány 52 kártyalapos paklit:

```
1 class Pakli:
2     def __init__(self):
3         self.kartyak = []
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4     for szin in range(4):
5         for ertek in range(1, 14):
6             self.kartyak.append(Kartya(szin, ertek))
```

A legegyszerűbb módja a pakli előállításának a beágyazott ciklus használata. A külső ciklus elszámol 0-tól 3-ig a színeknek megfelelően. A belső ciklus számlálja az értékeket 1-től 13-ig. Mivel a külső ciklus négyszer ismétlődik a belső pedig tizenháromszor a törzs végrehajtásának a teljes száma 52 (4x13). Minden iteráció létrehoz egy `Kartya` példányt az aktuális színnel és értékkel, és hozzáfűzi ezt a kártyát a `kartyak` listához.

Ezzel a megfelelő helyen megtestesíthetünk néhány paklit:

```
1 piros_pakli = Pakli()
2 kek_pakli   = Pakli()
```

23.6. A pakli kiírása

Rendszerint, amikor definiálunk egy új típust, akkor akarunk egy metódust, amely kiírja a példány tartalmát. A `Pakli` kiírásához bejárjuk a listát és így minden `Kartya` kiírásra kerül:

```
1 class Pakli:
2     ...
3     def kiir_pakli(self):
4         for kartya in self.kartyak:
5             print(kartya)
```

Itt és innentől a három pont (`...`) azt jelzi, hogy kihagytuk az osztály többi metódusát.

A `kiir_pakli` alternatívaként megírhatjuk a `__str__` metódust a `Pakli` osztályhoz. A `__str__` előnye az, hogy flexibilisebb. Az objektum tartalmának egyszerű kiírása helyett egy sztringet generál, amelyet a program többi részei manipulálhatnak kiírás előtt, vagy ezt el is tárolhatjuk a későbbi használatához.

Itt egy `__str__` verzió, ami a `Pakli` sztring reprezentációjával tér vissza. Egy kis pluszt hozzáadva elrendezhetjük a kártyákat lépcsősen eltolva, ahol minden egyes kártya egygyel több szóközzel van behúzva, mint a megelőzője.

```
1 class Pakli:
2     ...
3     def __str__(self):
4         s = ""
5         for i in range(len(self.kartyak)):
6             s = s + " " * i + str(self.kartyak[i]) + "\n"
7         return s
```

Ez a példa számos tulajdonságot demonstrál. Először is a `self.kartyak` bejárása és a kártyák egy változóhoz rendelése helyett az `i` ciklusváltozót használjuk a kártyalista indexeléséhez.

Másrészt használjuk a sztring szorzás operátort a kártyák egy-egy szóközzel bentebb húzásához. A `" " * i` kifejezés az `i` aktuális értékével megegyező számú szóközt eredményez.

Harmadszor, a kártyák `print` utasítással történő kiírása helyett az `str` függvényt használtuk. Egy objektum paraméterként történő átadása az `str` függvénynek egyenértékű az adott objektumon végzett `__str__` hívással.

Végezetül az `s` változót használtuk **gyűjtőként**. Kezdetben `s` egy üres sztring. Aztán minden cikluslépésben egy új sztring jön létre, és az `s` régi értéke ehhez fűződik hozzá és így kapjuk az új értéket. Amikor a ciklus véget ér, az `s` tartalmazza a `Pakli` teljes sztring reprezentációját, ami így néz ki:

```
1 piros_pakli = Pakli()
2 print(piros_pakli)
```

```
treff ás
treff 2
treff 3
treff 4
treff 5
treff 6
treff 7
treff 8
treff 9
treff 10
treff bubi
treff dáma
treff király
káró ás
káró 2
...
```

És így tovább. Habár az eredmény 52 sor, ez akkor is csak egyetlen sztring új sor karakterekkel.

23.7. Pakli keverés

Ha a pakli tökéletesen össze van keverve, akkor bármelyik kártya egyenlő valószínűséggel tűnhet fel bárhol a pakliban, és a paklin belüli bármelyik helyzetben ugyanolyan valószínűséggel fordulhat elő bármelyik kártya.

A pakli megkeveréséhez a `random` modul `randrange` függvényét fogjuk használni. A `randrange` az `a` és `b` egész paraméterekkel használva kiválaszt egy véletlen egész számot az `a <= x < b` intervallumból. Mivel az intervallum felső korlátja szigorúan kisebb, mint `b`, a második paraméterként a lista hosszát kell használnunk, ezzel garantálhatjuk, hogy legális indexet kapjunk. Például, ha az `rng` egy véletlenszám forrást testesít meg, akkor az alábbi kifejezés választja ki egy `random` kártya indexét a pakliban:

```
1 rng.randrange(0, len(self.kartyak))
```

Egy könnyű módja a keverésnek a kártyák bejárása, és az egyes kártyák felcserélése egy véletlenül választott másik kártyával. Lehetséges, hogy meg fogjuk cserélni a kártyát önmagával, de ez rendben van. Elméletben, ha meggátoljuk ezt a lehetőséget a kártyák nem lesznek tökéletesen összekeverve.

```
1 class Pakli:
2     ...
3     def kever(self):
4         import random
5         rng = random.Random()           # Hozz létre egy véletlenszám_
↳generátort!
6         kartya_szam = len(self.kartyak)
7         for i in range(kartya_szam):
8             j = rng.randrange(i, kartya_szam)
9             (self.kartyak[i], self.kartyak[j]) = (self.kartyak[j], self.
↳kartyak[i])
```

Ahelyett, hogy feltételeznénk, hogy 52 kártya van a pakliban, vesszük a lista aktuális hosszát, és eltároljuk a `kartya_szam` változóba.

Minden egyes kártya esetén választunk egy véletlen kártyát a rendezetlenek közül. Ezután megcseréljük az aktuális kártyát (`i`) a kiválasztottal (`j`). A kártyák cseréjéhez a rendezett `n`-es értékadást használjuk:

```
1 (self.kartyak[i], self.kartyak[j]) = (self.kartyak[j], self.kartyak[i])
```

Noha ez egy jó módszer a keverésre, a véletlenszám generátor objektumnak van egy `shuffle` metódusa, egy lista elemeinek helyben való keveréséhez. Újrírhatjuk a függvényt a gyárilag biztosított eszköz használatával:

```
1 class Pakli:
2     ...
3     def kever(self):
4         import random
5         rng = random.Random()           # Hozz létre egy véletlenszám_
        ↪generátort!
6         rng.shuffle(self.kartyak)       # Használd a shuffle metódust!
```

23.8. Osztás és a kártyák eltávolítása

Egy másik metódus, ami hasznos lehet a `Pakli` osztályban az az `eltavolit`, amelyik kap egy kártyát paraméterként és eltávolítja azt a pakliból, és `True` értékkel tér vissza, ha a lap a pakliban volt és `False` értékkel, ha nem:

```
1 class Pakli:
2     ...
3     def eltavolit(self, kartya):
4         if kartya in self.kartyak:
5             self.kartyak.remove(kartya)
6             return True
7         else:
8             return False
```

Az `in` operátor `True` értékkel tér vissza, ha az első operandus benne van a másodikban. Ha az első operandus egy objektum, a Python az objektum `__eq__` metódusát használja a listaelemek egyenlőségének meghatározásához. Mivel az `__eq__`, amit megadunk a `Kartya` osztályban a érték szerinti egyenlőséget vizsgálja (nem a referencia szerinti), így az `eltavolit` is azt fogja.

Kártyalapok osztásához el akarjuk távolítani a felső kártyát, és vissza akarunk térni vele. A `pop` lista metódus egy kényelmes módot biztosít erre:

```
1 class Pakli:
2     ...
3     def lapot_oszt(self):
4         return self.kartyak.pop()
```

Tulajdonképpen a `pop` eltávolítja az *utolsó* kártyát a listából, így ebben az értelemben a pakli aljáról osztunk.

Még egy művelet, amit valószínűleg szeretnénk, az egy Boolean függvény `ures_e` néven, ami `True` értékkel tér vissza, ha a pakliban nincs több kártya:

```
1 class Pakli:
2     ...
3     def ures_e(self):
4         return self.kartyak == []
```

23.9. Szójegyzék

kódol (encode) Egy adott típusú adat megjelenítése egy másik típusú értékkel, létrehozva egy leképezést közöttük.

osztály attribútum (class attribute) Egy változó, amely az osztályon belül lett definiálva, de kívül minden metóduson. Elérhető az osztály minden metódusából, és az osztály minden példánya osztozik rajta.

gyűjtő (accumulator) Egy változó, amelyet egy ciklusban használunk, hogy összegyűjtse értékek egy sorozatát, mondjuk a konkatenálva őket egy sztringben, vagy hozzáadva őket egy futó összeghez.

23.10. Feladatok

1. Módosítsd a `hasonlitas` függvényt úgy, hogy az `asz` értéke nagyobb legyen, mint a királyé!

24. fejezet

Öröklődés

24.1. Öröklődés

Az objektumorientált programozáshoz leggyakrabban társított nyelvi mechanizmus az öröklődés. Az öröklődés lehetővé teszi, hogy olyan új osztályokat definiáljunk, amelyek valamely létező osztály módosított változatai.

Az öröklődés legfőbb előnye, hogy anélkül adhatunk új metódusokat az osztályokhoz, hogy módosítsunk azokat. A folyamatot azért nevezik öröklődésnek, mert az új osztály a létező osztály összes metódusát örökli. A metaforát kiterjesztve a már létező osztályt gyakran **szülő**, az új osztályt pedig gyakran **gyermek** vagy **leszármazott** osztálynak, esetleg alosztálynak nevezik.

Az öröklődés hatékony nyelvi eszköz. Egyes programok, amelyek az öröklődés használta nélkül bonyolultak lennének, nagyon egyszerűen és tömören megírhatók a segítségével. Az öröklődés a kód újrahasznosítást is elősegíti, hiszen átszabhatjuk a szülő osztály viselkedését az osztály módosítása nélkül. Bizonyos esetekben az öröklődési hierarchia a probléma természetes szerkezetét is tükrözi, ami egyszerűbbé teszi a program megértését.

Másrészről az öröklődés használata nehezíti a kód olvasását. A metódus hívásoknál nem mindig egyértelmű, hogy hol kell keresni a hívott metódus definícióját, a kód lényegi része több modulba lehet szétszórva. Számos olyan öröklődéssel megoldható probléma van, amely öröklődés használta nélkül is éppen olyan elegánsan (vagy még elegánsabban) megoldható. Ha a probléma természete nem illeszkedik az öröklődéshez, akkor ez a programozási stílus több kárt okoz, mint amennyi hasznot hajt.

Ebben a fejezetben bemutatjuk, hogyan használható az öröklődés egy játékprogram, a Fekete Péter nevű kártyajáték részeként. Olyan kódot kívánunk készíteni, amelyet más kártyajátékok implementálása során is felhasználhatunk majd.

24.2. A kézben tartott lapok

Szinte minden kártyajátéknál szükséges a kézben tartott lapok nyilvántartása. A kéz hasonlít a paklihoz. Mindkettő egy-egy kártyahalmazt tartalmaz, és mindkét esetben szükség van a lap hozzáadása és elvétele műveletre. Akár a lapok összekeverése is kívánatos lehet, mind a kézben lévő, mind a pakliban lévő lapok esetében.

Akadnak azért eltérések is a kéz és a pakli között. A játéktól függően elképzelhető, hogy a kézben lévő lapokra olyan műveleteket is végre akarunk hajtani, amelyet nem lenne értelme a paklira alkalmazni. A pókerben például a figurákat (a kézben tartott lapokból előálló kombináció) különböző osztályokba sorolhatjuk (sor, flös, stb.), vagy összehasonlíthatjuk más figurákkal. A bridzsben pedig a kézben tartott lapok erejét lehet érdemes kiszámolni az ütések vállalása előtt.

Ez a szituáció szinte sugallja nekünk, hogy öröklődést használjunk. Ha a `Kez` a `Pakli` egy alosztálya, akkor tartalmazni fogja az `Pakli` összes metódusát, és új metódusokat is adhatunk hozzá.

Az ebben a fejezetben elkészített kódokat a korábbi fejezetben létrehozott `Kartyak.py` fájlhoz fűzzük hozzá. Az osztálydefinícióban a zárójelek között a szülő osztály neve áll:

```
1 class Kez(Pakli):  
2     pass
```

Ez az utasítás jelzi, hogy az új `Kez` osztály a már létező `Pakli` osztály leszármazottja.

A `Kez` konstruktor hívása inicializálja majd a `Kez` attribútumait, a `nev`-et és a `kartyak`-at. A sztring típusú `nev` attribútum a kezet azonosítja, például a játékos nevével. A `nev` paraméter opcionális, alapértelmezés szerint üres sztringet rendelünk hozzá. A `kartyak` attribútumhoz üres listát rendelünk az inicializáció során:

```
1 class Kez(Pakli):  
2     def __init__(self, nev=""):  
3         self.kartyak = []  
4         self.nev = nev
```

Szinte minden kártyajáték esetén szükség van arra, hogy lapokat tehessünk a pakliba, vagy lapokat vehessünk ki belőle. A lapok elvétele már megoldott, hiszen a `Kez` osztály örökli a `Pakli` osztály `eltavolit` metódusát, a beszúrás viszont meg kell írunk:

```
1 class Kez(Pakli):  
2     ...  
3     def add_hozza(self, kartya):  
4         self.kartyak.append(kartya)
```

A három pont továbbra is azt jelzi, hogy kihagytunk a leírásból bizonyos metódusokat. A lista `append` metódusa a lista végére fűzi az új kártyát.

24.3. Osztás

A `Kez` osztály elkészítése után foglalkozzunk azzal, hogy a `Pakli`-ből a kezekbe kerüljenek át a lapok. Nem teljesen egyértelmű, hogy ennek a metódusnak a `Kez` vagy a `Pakli` osztályban van-e a helye, de mivel egyetlen paklin dolgozik, viszont több kezet is érinthet a művelet, ezért kézenfekvőbb a `Pakli`-ba tenni. Az `osztas` metódusnak nagyon általánosnak kell lennie, hiszen a különböző játékoknál más-más igények fognak felmerülni. Előfordulhat, hogy az egész paklit ki akarjuk majd osztani egyszerre, de az is lehet, hogy csak egy-egy kártyát osztanánk minden játékosnak.

Az `osztas` két paramétert vár: a kezek listáját (vagy rendezett n-esét) és azt, hogy hány kártyát kívánunk kiosztani összesen. Ha nincs elég kártya a pakliban, akkor a metódus leáll a pakliban lévő kártyák kiosztása után:

```
1 class Pakli:  
2     ...  
3     def osztas(self, kezek, kartyak_szama = 999):  
4         kezek_szama = len(kezek)  
5         for i in range(kartyak_szama):  
6             if self.ures_e():  
7                 break # Ha elfogytak a kártyák,  
→megszakítjuk a ciklust. #  
8             kartya = self.adj_lapot() # Egy kártya elvétele a pakliból.  
9             kez = kezek[i % kezek_szama] # Ki a következő?  
10            kez.add_hozza(kartya) # Egy kártya odaadása a következő  
→játékosnak.
```

A második paraméter, a `kartyak_szama`, opcionális. Az alapértelmezett értéke egy hatalmas szám, gyakorlatilag azt jelenti, hogy az összes kártya kiosztásra kerül.

Az `i` ciklusváltozó 0-tól megy `kartyak_szama-1`-ig. Minden lépésben kiveszünk egy kártyát a pakliból az `adj_lapot` metódust használva, amely a lista `pop` metódusának csomagolója. Kiveszi a lista utolsó elemét és visszatér vele.

A maradékos osztás (%) segítségével egyesével osztjuk ki a lapokat a játékosoknak (round robin módon). Amikor az `i` eléri a `kezek_szama` értékét, akkor az `i % kezek_szama` kifejezéssel a lista elejére jutunk vissza (az index 0 lesz).

24.4. A kézben lévő lapok megjelenítése

A kézben lévő lapok megjelenítésénél kihasználhatjuk, hogy a `Kez` öröklí a `Pakli` osztály `__str__` metódusát. Például:

```
1 pakli = Pakli()
2 pakli.kever()
3 kez = Kez("Ferenc")
4 pakli.osztas([kez], 5)
5 print(kez)
```

Ferenc kezében ezek a lapok állnak:

```
pikk 2
pikk 3
pikk 4
kör ász
treff 9
```

Nem a legjobb lapok, de egy színsor még összejöhét.

Igazán kényelmes, hogy öröklünk egy már létező metódust, azonban a `Kez` objektumokban további információk is állnak, és ezek megjelenítése is kíváncsatos lehet. Amennyiben a `Kez` osztályban is elhelyezünk egy `__str__` metódust, azzal felülírhatjuk a `Pakli` osztály metódusát:

```
1 class Kez(Pakli)
2     ...
3     def __str__(self):
4         s = self.nev
5         if self.ures_e():
6             s += " keze üres\n"
7         else:
8             s += " kezében az alábbi lapok vannak:\n"
9         return s + Pakli.__str__(self)
```

Az `s` kezdetben a kezét azonosítja. Ha a kéz üres, a program hozzáfűzi a `keze üres` szavakat, majd visszatér az `s`-sel.

Ha nem üres, akkor a `"kezében az alábbi lapok vannak:n"` szöveget és a pakli tartalmát fűzi hozzá. Utóbbihoz meghívjuk a `Pakli` osztály `__str__` metódusát.

Talán furcsának tűnik, hogy az aktuális `Kez` objektumra hivatkozó `self`-et adjuk át a `Pakli` metódusának, de a `Kez` egyfajta `Pakli`. A `Kez` objektumok mindent végre tudnak hajtani, amit a `Pakli` objektumok, ezért `Kez` objektum is átadható a `Pakli` metódusának.

Általánosságban is elmondható, hogy ahol a szülő osztály példányai használhatóak, ott az alosztályok példányait is használhatjuk.

24.5. A KartyaJatek osztály

A KartyaJatek osztály intézi azokat az alapvető tevékenységeket, amelyekre minden játék során szükség van. Ilyen például a pakli létrehozása és keverése:

```
1 class KartyaJatek:
2     def __init__(self):
3         self.pakli = Pakli()
4         self.pakli.kever()
```

Először látunk olyat, hogy egy inicializáló metódus érdemi számításokat is végez az attribútumok inicializálásán túl.

A különböző típusú játékok implementálásakor új osztályt származtathatunk a KartyaJatek osztályból, majd különböző funkciókat adhatunk a játékhoz. Példaként egy olyan programot fogunk írni, amely a Fekete Péter nevű játékot szimulálja.

A Fekete Péter játékban az a cél, hogy megszabaduljunk a kézben tartott kártyáktól. A szín és szám szerint párosítható lapokat tehetjük le a kezünkből. Például a treff 4-es és a pikk 4-es párba tehető, hiszen mind a két lap fekete. A kőr bubi és a káró bubi párba tehető, hiszen mind a két lap piros.

A játék kezdete előtt a treff dámát eltávolítjuk a pakliból, így a pikk dámának nem lesz párja (a francia kártyával játszott változatban ez a Fekete Péter). A megmaradt 51 lapot egyesével kiosztjuk a játékosoknak. Az osztás után a játékosok párokba teszik a lapjaikat és leteszik a kezükből, amit csak tudnak.

Ha már nem találnak több párt, akkor kezdődik a játék. Minden körben, minden játékos egy lapot húz (annak előzetes megnézése nélkül) a bal kéz felől nézve legközelebbi, még lappal rendelkező szomszédjától. Ha a kihúzott kártya párosítható valamelyik, a játékos kezében lévő lappal, akkor a játékos leteszi a párt, különben beszúrja a kezében tartott lapok közé. Előbb-utóbb minden párosítható lapnak megtaláljuk a párját, csak a pikk dáma marad ott a vesztes kezében.

A számítógépes szimulációknban a gép lesz minden játékos. Sajnos egy ici-pici elveszik az igazi játékból. A valóságban a Fekete Péterrel rendelkező játékos igyekszik úgy tartani a lapokat, hogy a szomszédja pont a Fekete Pétert húzza ki. Például igyekszik feltűnővé tenni vagy kevésbé feltűnővé tenni, esetleg nem tenni kevésbé feltűnővé. A számítógép csak választ egy lapot véletlenszerűen a szomszéd kártyái közül.

24.6. FeketePeterKez osztály

A Fekete Péter játéknál a kezeknek olyan képességekre is szükségük van, amelyek meghaladják a Kez objektumok általános képességeit. Definálni fogunk egy FeketePeterKez osztályt a Kez osztály leszármazottjaként, és kiegészítjük egy egyezeket_tavolitsd_el metódussal:

```
1 class FeketePeterKez(Kez):
2
3     def egyezeket_tavolitsd_el(self):
4         darab=0
5         eredeti_kartyak=self.kartyak[:]
6         for kartya in eredeti_kartyak:
7             par=Kartya(3 - kartya.szín, kartya.ertek)
8             if par in self.kartyak:
9                 self.kartyak.remove(kartya)
10                self.kartyak.remove(par)
11                print("{0} kezében lévő pár: {1} {2}".
12                        format(self.nev, kartya, par))
13                darab+=1
14        return darab
```

Első lépésként egy másolatot készítünk a kártyákról. A másolat bejárása közben az eredeti kártyák közül távolítjuk el a lapokat. A `self.kartyak`-at azért nem akarjuk a bejárás vezérléséhez felhasználni, mert megváltozhat a cikluson belül. A Pythont egészen összezavarhatja, ha olyan listát kell bejárnia, amely a bejárás alatt változik!

Minden kézben tartott kártyának meghatározzuk és megkeressük a párját. A kártyához tartozó pár értéke és a minta színe (piros, fekete) azonos a kártya értékével és mintájának színével. A kártya színe (treff, kőr,...) viszont eltérő. A `3 - kartya.szín` kifejezés a treffból (0. szín) pikket (3. szín), a káróból (1. szín) kőrt (2. szín) csinál. Meggyőződhetünk róla, hogy fordítva is működik az eljárás. Ha a lap párja is a játékos kezében van, akkor mindkét kártyát eltávolítjuk.

Az alábbi példa az `egyezoket_tavolitsd_el` metódus használatát mutatja be:

```
1 jatek = KartyaJatek()
2 kez = FeketePeterKez("Ferenc")
3 jatek.pakli.osztas([kez], 13)
4 print(kez)
5 kez.egyezoket_tavolitsd_el()
6 print(kez)
```

Az első `print` utasítás az osztás utáni állapotot jeleníti meg:

```
Ferenc kezében az alábbi lapok vannak:
káró 9
  kőr dáma
    pikk 9
      kőr 10
        pikk 5
          pikk 6
            kőr ász
              kőr 7
                káró 2
                  káró dáma
                    káró 4
                      kőr 4
                        káró ász
```

Az `egyezoket_tavolitsd_el` metódus 3 párt talál:

```
Ferenc kezében lévő pár: kőr dáma káró dáma
Ferenc kezében lévő pár: kőr ász káró ász
Ferenc kezében lévő pár: káró 4 kőr 4
```

A 2. `print` segítségével meggyőződhetünk arról, hogy a párok valóban eltűntek-e a játékos kezéből:

```
Ferenc kezében az alábbi lapok vannak:
káró 9
  pikk 9
    kőr 10
      pikk 5
        pikk 6
          kőr 7
            káró 2
```

Figyeld meg, hogy a `FeketePeterKez` osztály nem tartalmaz `__init__` metódust! A `Kez` osztályét örökli.

24.7. FeketePeterJatek osztály

Most már foglalkozhatunk magával a játékkal. A FeketePeterJatek a KartyaJatek alosztálya. Egy új metódusa van, a jatek, amely a játékosok listáját várja paraméterként.

Mivel az `__init__` metódus öröklődik a KartyaJatek osztálytól, minden új FeketePeterJatek objektum tartalmaz egy megkevert kártyapaklit:

```
1 class FeketePeterJatek(KartyaJatek):
2
3     def jatek(self, nevek):
4         # A treff dáma eltávolítása
5         self.pakli.eltavolit(Kartya(0,12))
6
7         # Egy kéz készítése minden játékoshoz
8         self.kezek = []
9         for nev in nevek:
10             self.kezek.append(FeketePeterKez(nev))
11
12         # A kártyák kiosztása
13         self.pakli.osztas(self.kezek)
14         print("----- A kártyák kiosztva ----- ")
15         self.kezek_kiirasa()
16
17         # A kezdeti párok eltávolítása
18         parok_szama = self.osszes_par_eltavolitasa()
19         print("----- Párok eltávolítva, kezdődik a játék -----")
20         self.kezek_kiirasa()
21
22         # A játék addig zajlik, amíg a 25 párt meg nem találjuk
23         ki_kovetkezik = 0
24         kezek_szama = len(self.kezek)
25         while parok_szama < 25:
26             parok_szama += self.egy_kor_lejatszasa(ki_kovetkezik)
27             ki_kovetkezik = (ki_kovetkezik + 1) % kezek_szama
28
29         print("----- A játéknak vége -----")
30         self.kezek_kiirasa()
```

A `kezek_kiirasa` metódus elkészítését meghagyjuk feladatnak.

A játék néhány lépése külön metódusba kerül. Az `osszes_par_eltavolitasa` bejárja a kezek listáját, és mindegyikre meghívja az `egyezoket_tavolitsd_el` metódust:

```
1 class FeketePeterJatek(KartyaJatek):
2     ...
3     def osszes_par_eltavolitasa(self):
4         darab=0
5         for kez in self.kezek:
6             darab+=kez.egyezoket_tavolitsd_el()
7         return darab
```

A `darab` változóban összesítjük, hogy az egyes játékosok kezében hány pár van. Ha már minden kez objektumot feldolgoztunk, akkor visszaadjuk az összesített értéket (a `darab`-ot).

Ha a párosítások össz-száma (`parok_szama`) eléri a 25-öt, akkor 50 kártya került ki a játékosok kezeiből, vagyis egyetlen lap maradt, ami a játék végét jelenti.

A `ki_kovetkezik` változó tartja nyilván, hogy melyik játékos következik. Az értéke 0-tól indul és egyesével

növekszik. Ha eléri a `kezek_szama` értéket, akkor a moduló operátor miatt 0-tól indul újra a számlálás.

Az `egy_kor_lejatszasa` metódusnak van egy `i` paramétere, mely megadja, hogy melyik játékos van soron. A metódus az adott körben történt párosítások számát adja vissza:

```
1 class FeketePeterJatek(KartyaJatek):
2     ...
3     def egy_kor_lejatszasa(self, i):
4         if self.kezek[i].ures_e():
5             return 0
6
7         szomszed=self.keress_szomszedot(i)
8         huzott_lap=self.kezek[szomszed].adj_lapot()
9         self.kezek[i].add_hozza(huzott_lap)
10        print(self.kezek[i].nev, "által húzott kártya:", huzott_lap)
11        darab=self.kezek[i].egyezeket_tavolitsd_el()
12        self.kezek[i].kever()
13        return darab
```

Ha valamely játékos keze üres, az azt jelzi, hogy már kiment a játékból. Ilyenkor nem csinál semmit, a metódus nullát ad vissza.

Ha még nem ment ki, akkor megkeresi a balról nézve legközelebbi, még kártyával rendelkező szomszédot, és elvesz tőle egy lapot, majd megnézi, hogy párosítható-e valamelyik lapjával. A metódus visszatérése előtt megkeverjük a kézben lévő lapokat, így a következő játékos által húzott lap véletlenszerű lesz.

A `keress_szomszedot` metódus játéktól közvetlenül balra lévő játéktól indítja a keresést és halad körbe, ameddig nem talál egy kártyával rendelkező játékost:

```
1 class FeketePeterJatek(KartyaJatek):
2     ...
3     def keress_szomszedot(self, i):
4         kezek_szama=len(self.kezek)
5         for kovetkezo in range(1, kezek_szama):
6             szomszed=(i + kovetkezo) % kezek_szama
7             if not self.kezek[szomszed].ures_e():
8                 return szomszed
```

Ha a `keress_szomszedot` úgy érne véget, hogy nem találtunk lappal rendelkező játékost, akkor `None` értéket adna vissza a függvény, ami a program egy másik pontján hibát okozna. Szerencsére bizonyítható, hogy ez soha nem történhet meg (feltéve, hogy helyesen határozzuk meg a játék befejeződését).

A `kezek_kiirasa` metódust kihagytuk, azt már egyedül is el tudod készíteni.

Az alábbiakban egy rövidített parti kimenete látható. Csak a 15 legnagyobb értékű lap (tízes vagy azon felüli) került kiosztásra a három játékosnak. Ezzel a kis paklival 7 pár megtalálása után ér véget a játék a 25 helyett.

```
1 kartyajatek = FeketePeterJatek()
2 kartyajatek.jatek(["Eszti", "Sanyi", "Dávid"])
```

```
----- A kártyák kiosztva -----
Eszti kezében az alábbi lapok vannak:
treff király
kőr dáma
kőr bubi
pikk király
pikk dáma

Sanyi kezében az alábbi lapok vannak:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
kőr király
káró király
káró 10
treff 10
káró bubi

Dávid kezében az alábbi lapok vannak:
kőr 10
treff bubi
káró dáma
pikk 10
pikk bubi

Eszti kezében lévő pár: treff király pikk király
Sanyi kezében lévő pár: kőr király káró király
Dávid kezében lévő pár: treff bubi pikk bubi
----- Párok eltávolítva, kezdődik a játék -----
Eszti kezében az alábbi lapok vannak:
kőr dáma
kőr bubi
pikk dáma

Sanyi kezében az alábbi lapok vannak:
káró 10
treff 10
káró bubi

Dávid kezében az alábbi lapok vannak:
kőr 10
káró dáma
pikk 10

Eszti által húzott kártya: káró bubi
Eszti kezében lévő pár: kőr bubi káró bubi
Sanyi által húzott kártya: pikk 10
Sanyi kezében lévő pár: treff 10 pikk 10
Dávid által húzott kártya: kőr dáma
Dávid kezében lévő pár: káró dáma kőr dáma
Eszti által húzott kártya: káró 10
Dávid által húzott kártya: káró 10
Dávid kezében lévő pár: kőr 10 káró 10
----- A játéknak vége -----
Eszti kezében az alábbi lapok vannak:
pikk dáma

Sanyi keze üres

Dávid keze üres
```

Szóval Eszti vesztett.

24.8. Szójegyzék

gyerek osztály (child class) Egy új osztály, amelyet egy már létező osztályból származtattunk. Alosztálynak is nevezzük.

öröklődés (inheritance) Egy olyan mechanizmus, amely lehetővé teszi, hogy egy osztályt egy korábban definiált osztály módosított változataként definiáljunk.

szülő osztály (parent class) Egy olyan osztály, amelyből gyermek osztályt származtunk.

24.9. Feladatok

1. Készíts egy `kezek_kiirasa` metódust a `FeketePeterJatek` osztályon belül, amely bejárja a `self.kezek` listát, és kiírja minden egyes kéz tartalmát.
2. Hozz létre egy új teknős típust `TeknocVerda` néven, néhány új jellemzővel:

Tudjon előre ugrani adott távolságot. Legyen benne egy kilométeróra, amely számolja milyen távolságot tett meg a teknőc mióta legurult a gyártósorról. (A szülő osztályban több hasonló jelentésű metódus is van, pl. `fd`, `forward`, `back`, `backward`, `bk`, a feladat elkészítésekor csak a `forward` metódusra rakd rá a távolságmérő funkciót.)

Gondold át alaposan, hogyan hat a teknőc által megtett távolságra az, ha negatív távolságot lép előre. (Nem akarunk olyan használt teknőcverdát venni, amelyikben hamisan szerepel a megtett kilométer, mert az előző tulajdonos túl sokszor kerülte meg vele rükkvercben a háztömböt. Próbáld ki egy környezetben lévő autóban. Figyeld meg, hogy nő vagy csökken a kilométeróra állás, amikor tolatsz.)

3. A teknőcverda véletlenszerű távolság megtétele után kapjon defektet, és álljon le. A leállás után a `forward` metódus minden egyes meghívása váltson ki egy kivételt. Írj egy `kerek_csere` metódust is, amely rendbe teszi a lapos kerekeket.

25. fejezet

Láncolt listák

25.1. Beágyazott referenciák

Láttunk már példát olyan attribútumra, amely egy másik objektumra hivatkozik, ezeket **beágyazott referenciáknak** hívjuk. Egy közös adatszerkezet a **láncolt lista** kihasználja ennek előnyeit.

A láncolt listák **csomópontokból** állnak, ahol mindegyik csomópont tartalmaz egy hivatkozást, vagyis referenciát a lista következő elemére. Továbbá, minden csomópont tartalmaz egy adategységet is, amit **adatrésznek** nevezünk.

A láncolt lista felfogható egy **rekurzív adatszerkezetként**, mert rekurzív definíciója van.

Egy láncolt lista vagy:

1. üres lista, a speciális `None` értékkel reprezentálva vagy
2. egy csomópont, ami tartalmaz egy adatrészt és egy referenciát egy láncolt listára.

A rekurzív adatszerkezetek lehetővé teszik számunkra a rekurzív metódusok használatát.

25.2. A Csomópont osztály

Rendszerint, amikor írunk egy új osztályt az inicializációval és az `__str__` metódussal kezdjük, szóval tesztelhetjük az új típus létrehozásának és a megjelenítésének mechanizmusát:

```
1 class Csomopont:
2     def __init__(self, adatresz=None, kovetkezo=None):
3         self.adatresz = adatresz
4         self.kovetkezo = kovetkezo
5
6     def __str__(self):
7         return str(self.adatresz)
```

Általában az inicializáló metódus paramétere opcionális. Alapból, mind az adatrész, mind a `kovetkezo` referencia `None` értékű.

A csomópont sztring reprezentációja nem más, mint csak az adatrész sztring reprezentációja. Mivel bármilyen érték átadható az `str` függvénynek, bármiféle értéket tárolhatunk a listában.

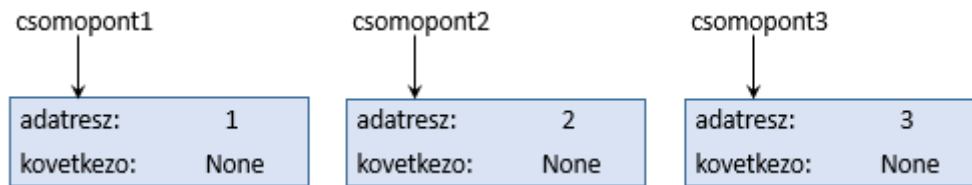
Az eddigi implementáció teszteléséhez létrehozhatunk egy `Csomopont` objektumot és kiírathatjuk:

```
1 csomopont = Csomopont("teszt")
2 print(csomopont)
```

Hogy izgalmasabbá tegyük a dolgot, szükségünk van egy listára egynél több elemmel:

```
1 csomopont1 = Csomopont(1)
2 csomopont2 = Csomopont(2)
3 csomopont3 = Csomopont(3)
```

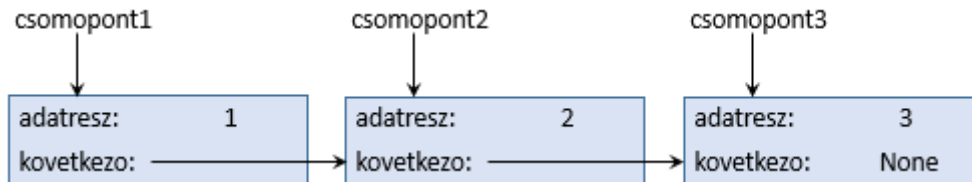
Ez a kód három csomópontot kreál, de még nincs listánk, mert e csomópontok nincsenek **láncolva**. Az állapot diagram így néz ki:



A csomópontok összeláncolásához az első csomópontnak hivatkoznia kell a másodikra, a másodiknak a harmadikra:

```
1 csomopont1.kovetkezo = csomopont2
2 csomopont2.kovetkezo = csomopont3
```

A harmadik csomópont referenciája továbbra is None, ami azt jelöli, hogy ez a lista vége. Most az állapot diagram így néz ki:



Most tudod, hogyan kell létrehozni csomópontokat és összeláncolni őket egy listává. Ami lehet, hogy kevésbé tiszta számodra, hogy miért.

25.3. Listák kollekcióként

A listák hasznosak, mert módot adnak arra, hogy több objektumot egyetlen egyeddé rakj össze, amit gyakran **kollekció**nak hívnak. A példában az első csomópontja a listának egyfajta referenciaként szolgál az egész listára.

Egy lista paraméterként történő átadásához nekünk csak a lista első elemének a referenciáját kell átadnunk paraméterként. Például a `kiir_lista` függvény csak egy csomópontot vár argumentumként. A lista fejével kezdve kiír minden egyes elemet egészen a végéig:

```
1 def kiir_lista(csomopont):
2     while csomopont is not None:
3         print(csomopont, end=" ")
4         csomopont = csomopont.kovetkezo
5     print()
```

Ennek a függvénynek a meghívásához, csak átadunk egy referenciát az első csomópontra:

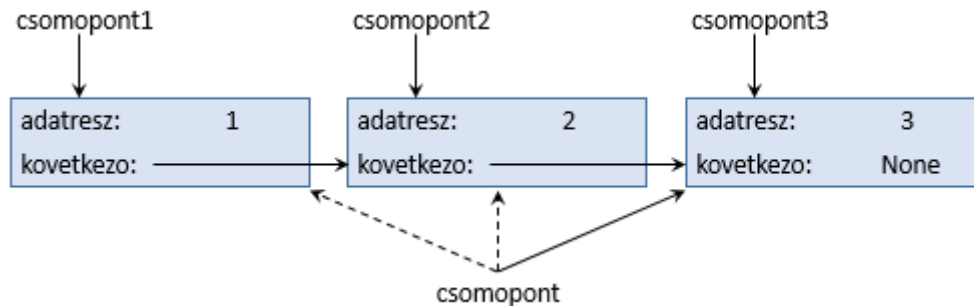
```
1 kiir_lista(csomopont1)
```

A kimenet ez lesz: 1 2 3.

A `kiir_lista` függvényen belül van egy referenciánk a lista első elemére, de nincs változó, amelyik a többire hivatkozik. A `kovetkezo` értéket kell használnunk minden csomópont esetén, hogy elérjük a következőt.

A láncolt lista bejárásához, mindennapos dolog egy ciklusváltozót használni, mint a `csomopont`, a sorozat egyes csomópontjaira hivatkozáshoz.

Ez a diagram a lista értékét és a csomópont értékét mutatja:



25.4. Listák és a rekurzió

Természetes dolog a sok lista műveletet rekurzív metódusokkal kifejezni. Például, a következő egy rekurzív algoritmus, egy lista visszafelé történő kiírására:

1. Válaszd szét a listát két részre: az első csomópont (a lista feje) és a maradék (a lista farokrésze)!
2. Írasd ki a farokrészt visszafelé!
3. Írasd ki a fejet!

Természetesen a 2. lépés a rekurzív hívás, feltételezi, hogy már van egy módszerünk a lista visszafelé történő kiírására. Azonban ha feltételezzük, hogy a rekurzív hívás működik – ez egy bizalmi kérdés – akkor meggyőzhetjük magunkat, hogy az algoritmus működik.

Amire szükségünk van, az csak egy alapeset és egy bizonyítási mód minden listára, miszerint elérjük az alap esetet. A lista rekurzív definíciója adott, egy természetes alapeset az üres lista, `None` értékkel reprezentálva:

```
1 def kiir_visszafele(lista):
2     if lista is None: return
3     fej = lista
4     farokresz = lista.következo
5     kiir_visszafele(farokresz)
6     print(fej, end=" ")
```

Az első sor kezeli az alapesetet azzal, hogy nem csinál semmit. A következő két sor szétvágja a listát fejre és farokrészre. Az utolsó két sor kiírja a listát. Az `end` paraméter a `print` utasításban meggátolja, hogy a Python minden csomópont után új sort kezdjen.

Hívjuk meg ezt a metódust, mint ahogy a `kiir_lista` függvényt is meghívtuk:

```
1 kiir_visszafele(csomopont1)
```

Az eredmény a lista elemei visszafelé: 3 2 1.

Talán csodálkozol, hogy a `kiir_lista` és a `kiir_visszafele` miért függvények és nem a `Csomopont` osztály metódusai. Az ok az, hogy a `None` értéket akartuk használni az üres lista megjelenítésére, de az nem szabályos, hogy meghívunk egy metódust a `None` értékre. Ez a korlátozás ügyetlenné teszi a listák írását – a kód manipulálását egy tisztán objektum-orientált módon.

Be tudjuk bizonyítani, hogy a `kiir_visszafele` mindig véget ér-e? Más szóval mindig eléri-e az alap esetet? A valóságban a válasz: nem. Néhány lista problémát okoz ennek a módszernek.

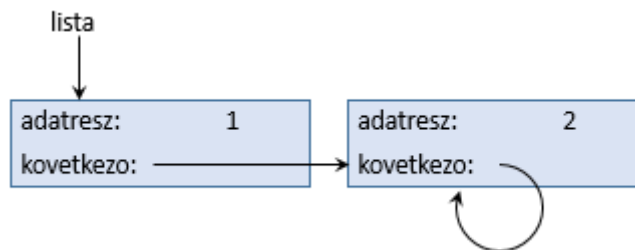
Térjünk vissza a Rekurzió fejezethez

A korábbi rekurzióról szóló fejezetünkben megkülönböztettük a magas szintű nézetet, ami bizalmi dolog, és az alacsony szintű műveleti nézetet. A mentális blokkosítás gondolatvilágában az absztraktabb magas szintű nézet használatára akarunk ösztönözni.

Azonban ha látni akarsz a részleteket, akkor használnod kellene az utasításonkénti nyomkövetés eszközeit, hogy belépj a rekurzió szintjére, és megvizsgáld a végrehajtási verem keretét minden `kiir_visszafele` hívásnál.

25.5. Végtelen listák

Nincs semmi, ami megóvna egy csomópontot, hogy visszahivatkozzon a lista egy korábbi elemére. Például ez az ábra egy kételemű listát mutat, melyek közül az egyik saját magát hivatkozik:



Ha meghívjuk a `kiir_lista` függvényt erre a listára, ez a ciklus örökké ismétlődni fog. Ha meghívjuk a `kiir_visszafele` függvényt végtelen rekurzióba kerül. Ez a fajta viselkedés nehézzé teszi a végtelen listákkal való munkát.

Mindazonáltal ezek alkalomadtán hasznosak lehetnek. Például reprezentálhatunk egy számot számjegyek listájaként, és használhatjuk a végtelen listát az ismétlődő tizedes törtek ábrázolására.

Mindenesetre az problematikus, hogy nem tudjuk igazolni, hogy a `kiir_lista` és a `kiir_visszafele` függvények befejeződnek-e. A legjobb, amit tehetünk egy hipotetikus állítás „Ha a lista nem tartalmaz hurkot, akkor a metódus befejeződik.” Ez a fajta állítás az **előfeltétel** névre hallgat. Ez egy kényszerrel vet az egyik paraméterre, és akkor írja le a metódus viselkedését, amikor a kényszer ki van elégítve. Több példát látsz hamarosan.

25.6. Az alapvető félreérthetőség tétele

A `kiir_visszafele` függvény egy részén talán felhúzod a szemöldököd:

```
1 fej = lista
2 farokresz = lista.kovetkezo
```

Az első értékadás után a `fej` és a `lista` típusa és értéke is ugyanaz. Miért hoztunk létre egy új változót?

Az ok az, hogy a két változó eltérő szerepet játszik. A `fej` változóra gondolhatunk úgy, mint egy hivatkozás egy egyszerű csomópontra, és azt gondolhatjuk, hogy a `lista` egy hivatkozás a lista első elemére. Ezek a szerepek nem részei a programnak, ezek csak a programozó gondolataiban léteznek.

Általában ránézésre nem tudjuk megmondani, hogy mi a szerepe egy változónak a programban. Ez a félreérthetőség hasznos lehet, de nehezen olvashatóvá teszi a programot. Gyakran használunk olyan változóneveket, mint a `fej` vagy a `lista` ezzel dokumentálva, hogy mire is szándékozunk használni a változót, és néha további változókat hozunk létre az egyértelműség kedvéért.

Megírhattuk volna a `kiir_visszafele` függvényt a `fej` és a `farokresz` nélkül is, ami sokkal tömörebb, de kevésbé tiszta lenne:

```
1 def kiir_visszafele(lista):
2     if lista is None: return
3     kiir_visszafele(lista.kovetkezo)
4     print(lista, end=" ")
```

Megnézve a két függvényhívást emlékezhethetünk arra, hogy a `kiir_visszafele` úgy kezeli a paraméterét, mint egy kollekción, és a `print` pedig úgy, mint egyedülálló objektumot.

Az **alapvető félreérthetőség tétel** a félreérthetőséget úgy írja le, mint a csomópontra hivatkozás velejáróját: *a változó, amely egy csomópontra hivatkozik kezelhető egyszerű objektumként vagy egy csomópontlista első elemeként.*

25.7. A listák módosítása

Két módja van a listák módosításának. Nyilvánvalóan meg tudjuk változtatni az egyik csomópont adatrészét, de a sokkal érdekesebb műveletek a bővítés, a törlés és a csomópontok újrendezése.

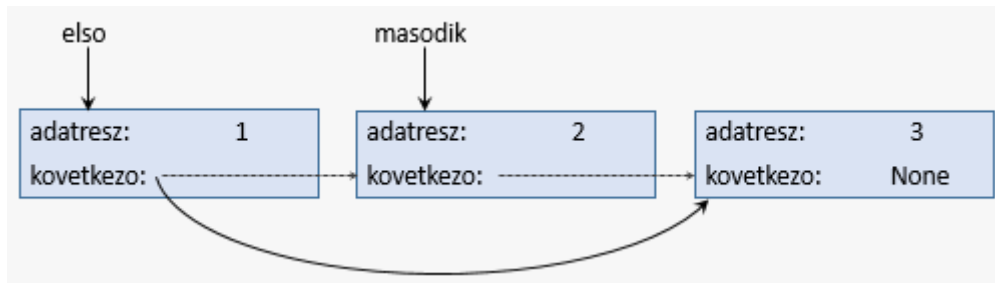
Példaként írjuk meg azt a metódust, amely eltávolítja a lista második elemét és visszatér annak referenciájával:

```
1 def torol_masodik(lista):
2     if lista is None: return
3     elso = lista
4     masodik = lista.kovetkezo
5     # Hivatkozzon az első csomópont a harmadikra
6     elso.kovetkezo = masodik.kovetkezo
7     # Válassz le a másodikat a lista megmaradt részéről
8     masodik.kovetkezo = None
9     return masodik
```

Ismét használtunk egy átmeneti változót, hogy a kód érthetőbb legyen. Itt van, hogyan használhatjuk ezt a metódust:

```
1 kiir_lista(csomopont1)
2 torolt = torol_masodik(csomopont1)
3 kiir_lista(torolt)
4 kiir_lista(csomopont1)
```

Az első kiment az 1 2 3, a második 2, a harmadik pedig 1 3. Ez az állapotdiagram megmutatja a művelet hatását:



Mi történik, ha meghívjuk a metódust, és olyan listát adunk át neki, aminek csak egyetlen eleme van (**egyke**)? Mi van, ha egy üres listát adunk át paraméterként? Van előfeltétele a metódus használatának? Ha igen, akkor javítsd ki a metódust, hogy kezelje az előfeltétel megsértését ésszerű módon.

25.8. Csomagolók és segítők

Gyakran hasznos egy lista műveletet felosztani két metódusra. Például egy lista visszafelé történő kiírásakor a megszokott [3, 2, 1] formátum használata esetén, alkalmazhatjuk a `kiir_visszafele` metódust a 3, 2, kiírásához, de egy külön metódusra van szükségünk a szögletes zárójelek és az első elem kiírásához. Hívjuk ezt `kiir_visszafele_szepen` függvénynek:

```
1 def kiir_visszafele_szepen(lista):
2     print("[", end=" ")
3     kiir_visszafele(lista)
4     print("]")
```

Ismét jó ötlet ellenőrizni a metódusokat, hogy lássuk működnek-e speciális esetekben, mint például az üres lista vagy az egyke esetén.

Amikor ezt a metódust a programban valahol máshol használjuk, meghívjuk a `kiir_visszafele_szepen` metódust közvetlenül, és ez hívja meg a `kiir_visszafele` metódust a nevünkben. Ebben az értelemben a `kiir_visszafele_szepen` a **csomagoló** szerepét tölti be, míg a `kiir_visszafele` a **segítő**.

25.9. A `LancoltLista` osztály

Van néhány apró probléma azzal, ahogy a listákat ábráztuk. Az ok és okozat megfordításával most először ajánlunk egy alternatív implementációt, és aztán megmagyarázzuk, milyen problémákat oldottunk meg.

Először egy új osztályt hozunk létre `LancoltLista` néven. Az attribútumai egy egész szám, ami megmondja a lista hosszát, és egy referencia az első csomópontra. A `LancoltLista` objektumok kezelőként fognak szolgálni a `Csomopont` objektumok listájának manipulálásához:

```
1 class LancoltLista:
2     def __init__(self):
3         self.hossz = 0
4         self.fej = None
```

Egy jó dolog a `LancoltLista` osztállyal kapcsolatban, hogy egy természetes helyet szolgáltat a csomagoló függvények, mint a `kiir_visszafele_szepen` számára, amit a `LancoltLista` osztályban elkészíthetünk:

```
1 class LancoltLista:
2     ...
3     def kiir_visszafele(self):
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
4     print("[", end=" ")
5     if self.fej is not None:
6         self.fej.kiir_visszafele()
7     print("]")
8
9 class Csomopont:
10     ...
11     def kiir_visszafele(self):
12         if self.kovetkezo is not None:
13             farokresz = self.kovetkezo
14             farokresz.kiir_visszafele()
15         print(self.adatresz, end=" ")
```

Csak, hogy összezavarjuk a dolgokat, átneveztük a `kiir_visszafele_szepen` függvényt. Most két metódusunk van `kiir_visszafele` néven: egy a `Csomopont` osztályban (a segítő) és egy a `LancoltLista` osztályban (a csomagoló). Amikor a csomagoló meghívja a `self.fej.kiir_visszafele` metódust, ezzel meghívja a segítőt, mert a `self.fej` egy `Csomopont` objektum.

Egy másik haszna a `LancoltLista` osztálynak az, hogy egyszerűbbé teszi az első elem hozzáadását és törlését. Például az `elso_hozzaad` a `LancoltLista` egy metódusa, kap egy adatrészként tárolandó értéket paraméterként, és a lista elején helyezi el:

```
1 class LancoltLista:
2     ...
3     def elso_hozzaad(self, adatresz):
4         csomopont = Csomopont(adatresz)
5         csomopont.kovetkezo = self.fej
6         self.fej = csomopont
7         self.hossz += 1
```

Rendszerint ellenőrizned kellene, hogy a kód, mint ez is, kezeli-e a speciális helyzeteket. Például mi történik, ha a lista kezdetben üres?

25.10. Invariánsok

Néhány lista jól megformált, míg másik nem. Például, ha a lista tartalmaz egy hurkot, az sok metódusunk számára gondot okoz, így el akarjuk várni, hogy a listák ne tartalmazzanak hurkot. Egy másik elvárás, hogy a `hossz` értéke a `LancoltLista` objektumban legyen egyenlő a lista aktuális elemeinek számával.

Ezeket az elvárásokat **invariánsoknak** hívjuk, mert ideális esetben, mindig igazak minden objektumra. Invariánsok definiálása az objektumok számára egy hasznos programozói gyakorlat, mert a kód helyességét könnyebben igazolhatóvá teszi, ellenőrzi az adatszerkezet integritását, detektálja a hibákat.

Egy dolog, ami néha zavaró lehet az invariánsokkal kapcsolatban az, hogy van amikor megsértjük őket. Például az `elso_hozzaad` közepén miután hozzáadtuk az új csomópontot, de még mielőtt növeltük volna a `hossz` értékét az invariáns sérül. Ez a fajta szabályszegés elfogadható, valójában gyakran lehetetlen anélkül módosítani az objektumot, hogy legalább egy kis időre ne sértsük meg az invariánst. Normális esetben azt várjuk el minden metódustól, amelyik megsérti az invariánst, hogy állítsa is helyre azt.

Ha van bármi jelentős kiterjesztés a kódban, amiben az invariáns sérül, akkor fontos a megjegyzésekkel ezt tisztázni, hogy ne hajtunk végre olyan műveletet, ami függ az invariánstól.

25.11. Szójegyzék

alapvető félreérthetőség tétel (fundamental ambiguity theorem) Egy referencia egy lista csomópontja kezelhető egyszerű objektumként vagy egy lista első elemeként.

beágyazott referencia (embedded reference) Egy referencia, azaz hivatkozás, amely egy objektum attribútumában van tárolva.

csomagoló (wrapper) Egy metódus, amely a közvetítő szerepét játssza a hívó és a segítő metódus között, gyakran teszi a metódust könnyebben vagy kevésbé hibásan hívhatóvá.

csomópont (node) A lista egy eleme, rendszerint egy objektummal implementálva, amely tartalmaz egy referenciát egy másik, azonos típusú objektumra.

egyke (singleton) Egyetlen elemű láncolt lista.

előfeltétel (precondition) Egy kijelentés, amely igaz kell legyen ahhoz, hogy egy metódus megfelelően működjön.

invariáns (invariant) Egy kijelentés, amelynek igaznak kell lennie egy objektumra minden időpillanatban (kivéve talán, amíg éppen változtatjuk az objektumot).

láncolt lista (linked list) Egy adatszerkezet, amely megvalósít egy kollekciót láncolt csomópontok sorozatát használva.

link (link) Egy beágyazott referencia, amit arra használunk, hogy egy elemet egy másikhoz kapcsoljunk.

segítő (helper) Egy metódus, amit nem közvetlenül hív meg a hívó, de másik metódus használja a művelet egy részének végrehajtásához.

adatrész (cargo) Egy adatelem, amely egy csomópontban tárolva van.

25.12. Feladatok

1. Megegyezés szerint a listákat gyakran úgy íratjuk ki, hogy szögletes zárójelbe tesszük őket és vesszőt írunk közéjük, ekképpen `[1, 2, 3]`. Módosítsd a `kiir_lista` függvényt úgy, hogy ennek megfelelően állítsa elő a kimenetet!

26. fejezet

Verem

26.1. Absztrakt adattípusok

Az adattípusok, amelyeket eddig láttál mind konkrétak abban az értelemben, hogy teljesen specifikált, hogy hogyan kell őket használni. Például a `Kartya` osztály két egész számot használva reprezentál egy kártyát. Ahogy megbeszéljük akkor, ez nem az egyetlen módja a kártyák reprezentációjának, számos alternatív implementáció létezik.

Egy **absztrakt adattípus** (röviden AAT) meghatároz egy halom műveletet (vagy metódust) és a műveletek szemantikáját (miért csinálják), de nem adja meg a műveletek implementációját. Ez teszi absztrakttá.

Miért hasznos ez?

1. Egyszerűsíti az algoritmus leírását, ha használhatod az algoritmust anélkül, hogy arra gondolnál közben, hogyan is hajtódik végre a művelet.
2. Mivel általában számos módon implementálhatunk egy absztrakt adattípust, hasznos lehet írni egy algoritmust úgy, hogy bármelyik implementációt használhatja.
3. A jól ismert absztrakt adattípusok, mint a verem ebben a fejezetben, gyakran standard függvénykönyvtárakban vannak implementálva, így csak egyszer írták meg őket, de sok programozó használhatja ezeket.
4. Az absztrakt adattípusokon végzett műveletek lehetővé teszik egy hétköznapi magas szintű programozási nyelv számára, hogy az algoritmusokról és azok specifikációjáról beszélhessünk.

Amikor absztrakt adattípusokról beszélünk, gyakran különbséget teszünk az adatszerkezetet használó kód, azaz a **kliens** kód és az absztrakt adattípusokat implementáló úgynevezett **szolgáltató / implementáló** kód között.

26.2. A verem AAT

Ebben a fejezetben egy mindennapos absztrakt adattípust a **vermet** nézzük meg. A verem egy kollekció, ami azt jelenti, hogy egy olyan adatszerkezet, amely több elemet tartalmaz. Más kollekciókat is láttunk már eddig, mondjuk a szótárakat és a listákat.

Egy absztrakt adattípus a rajta alkalmazható műveletek segítségével definiálható, amit **interfésznek** hívunk. A verem interfésze ezeket a műveleteket tartalmazza:

__init__ Inicializál egy új üres vermet.

push Betesz egy új elemet a verembe.

pop Kivesz egy elemet a veremből és visszatér vele. A visszaadott eleme mindig a legutóbb betett érték.

ures_e Ellenőrzi, hogy a verem üres-e.

Egy vermet gyakran nevezünk „Last In, First Out” röviden **LIFO** adatszerkezetnek, mert a legutóbb hozzáadott elemet távolítjuk el először.

26.3. Verem implementációja Python listákkal

A lista műveletek, amelyeket a Python szolgált nagyon hasonlóak azokhoz, amelyekkel a vermet definiáljuk. Az interfész nem teljesen az, amit feltételeztünk, de írhatunk egy kódot, ami lefordítja a verem absztrakt adattípust a beépített műveletekre.

Ezt a kódot hívhatjuk a verem AAT **implementációjának**. Általában egy implementáció nem más, mint metódusok egy halmaza, amelyek kielégítik az interfész szintaktikai és szemantikai követelményeit.

Itt egy implementációja a verem absztrakt adattípusnak, amely Python listát használ:

```
1 class Verem:
2     def __init__(self):
3         self.tetelek = []
4
5     def push(self, tetel):
6         self.tetelek.append(tetel)
7
8     def pop(self):
9         return self.tetelek.pop()
10
11    def ures_e(self):
12        return (self.tetelek == [])
```

Egy Verem objektum tartalmaz egy tetelek nevű attribútumot, ami a verembeli tételek listája. Az inicializáló metódus hatására a tetelek egy üres lista lesz.

Egy új tétel verembe helyezéséhez a push művelet a tetelek listájához fűzi azt. A veremből való elemkivétel során a pop egy homoním (*azonos nevű*) metódust használ az eltávolításhoz, és visszatér a lista utolsó tételével.

Végül az üresség ellenőrzése során az ures_e összehasonlítja a tetelek listát egy üres listával.

Egy ilyen implementációnak, amiben a metódus tartalmaz külső metódushívásokat, a neve **csomagolás**. Az informatikusok ezt a metaforát használják egy kód leírására, ami elrejtí az implementáció részleteit, és egy egyszerűbb vagy szabványosabb interfészt biztosít.

26.4. Push és pop

A verem egy **generikus adatszerkezet**, ami azt jelenti, bármilyen típusú tételt hozzá tudunk adni. A következő példa két egészet és egy sztringet tesz be a verembe:

```
1 s = Stack()
2 s.push(54)
3 s.push(45)
4 s.push("+")
```

Használhatjuk az ures_e és a pop metódusokat az eltávolításhoz és az összes tétel kiíratásához:

```
1 while not s.ures_e():
2     print(s.pop(), end=" ")
```

A kiment `+ 45 54`. Más szóval mondva, csak egy vermet kell használnunk adatok fordított kiírásához. Ez nem egy standard módja a lista kiírásának, de így észrevehetően könnyebb megtenni.

Össze kellene hasonlítanod ezt a kis kódrészletet a `kiir_visszafele` implementációjával. Van egy természetes párhuzam a `kiir_visszafele` és a verem algoritmus között. A különbség annyi, hogy a `kiir_visszafele` futásidejű vermet használ a csomópontok nyomon követéséhez, amíg bejárja a listát, és aztán írta ki az értékeket visszafelé a rekurziónak megfelelően. A verem algoritmus ugyanezt teszi kivéve, hogy a `Verem` objektumot használja a futásidejű verem helyett.

26.5. Verem használata posztfix kifejezés kiértékeléséhez

A legtöbb programozási nyelven a matematikai kifejezések két operandus közti operátor formájában vannak írva, mint például `1 + 2`. Ezt az alakot hívjuk **infixnek**. Az egyik alternatíva, amit néhány számológép is használ a **posztfix** alak. A posztfix kifejezésekben az operátor az operandusokat követi, így: `1 2 +`.

Az ok, ami miatt a posztfix alak néha hasznos lehet az, hogy van egy természetes módja az ilyen kifejezések kiértékelésének verem használatával:

1. Indulj el a kifejezés elején, vegyél egy tagot (operátort vagy operandust) minden lépésben!
 - Ha a tag egy operandus tedd be a verembe!
 - Ha a tag egy operátor, akkor kapj ki két értéket a veremből és végezd el rajtuk az adott műveletet, majd az eredményt tedd be a verembe!
2. Amikor elérted a kifejezés végét, pontosan egy érték kell legyen a veremben. Ez az eredmény.

26.6. Nyelvtani elemzés

Az előző algoritmus implementálásához képesnek kell lennünk bejárni egy sztringet, és feltördelni azt operandusokra és operátorokra. Ez a folyamat egy jó példája a **nyelvtani elemzésnek**, aminek az eredménye – a sztring töredékei – a **szövegelemek**. Talán emlékszel erre a szóra az első fejezetből.

A Python egy `split` metódust szolgáltat mind a sztring objektumokra, mind a `re` (azaz a reguláris kifejezések) modulban. A sztringek `split` metódusa feldarabolja azokat egy listába egy egyszerű **határoló** használatával. Például:

```
1 print ("Most itt az idő".split(" "))
```

A kimenet ez lesz:

```
['Most', 'itt', 'az', 'idő']
```

Ebben az esetben a határoló a szóköz karakter, így a sztring a szóközőknél darabolódik fel.

A `re.split` függvény még hatásosabb, megengedi, hogy reguláris kifejezést adjunk meg határoló helyett. Egy reguláris kifejezés egy módja a sztringhalmazok meghatározásának. Például az `[A-z]` az összes betű halmaza és a `[0-9]` a számjegyek halmaza. A `^` operátor komplementer, negált halmazt eredményez, így a `[^0-9]` nem más, mint az a halmaz, ami bármit tartalmazhat, kivéve a számokat, ami pontosan az, amit használni akarunk egy posztfix kifejezés feldarabolásakor:

```
1 import re
2 re.split("[^0-9]", "123+456*/")
```

A kimenet most ez lesz:

```
['123', '+', '456', '*', '', '/', '']
```

Az eredmény lista magába foglalja a 123 és 456 operandusokat és a * valamint / operátorokat. Továbbá magába foglal még két üres sztringet, amelyeket az operátorok után szűr be.

26.7. Posztfix kiértékelés

A posztfix kifejezés kiértékeléséhez használni fogjuk az előző fejezet nyelvtani elemzőjét, és az előbbi algoritmust. Azért, hogy a dolgok átláthatóak maradjanak, kezdjük egy kiértékelővel, ami csak a + és * műveleteket implementálja:

```
1 def kiertekel_posztfix(kifejezes):
2     import re
3     szovegelem_lista = re.split("([0-9])", kifejezes)
4     verem = Verem()
5     for szovegelem in szovegelem_lista:
6         if szovegelem == "" or szovegelem == " ":
7             continue
8         if szovegelem == "+":
9             osszeg = verem.pop() + verem.pop()
10            verem.push(osszeg)
11        elif szovegelem == "*":
12            szorzat = verem.pop() * verem.pop()
13            verem.push(szorzat)
14        else:
15            verem.push(int(szovegelem))
16    return verem.pop()
```

Az első feltétel a szóközőket és üres sztringeket kezeli. A következő kettő az operátorokat kezeli. Feltesszük, hogy minden más csak operandus lehet. Természetesen jobb lenne ellenőrizni a téves inputot, és egy hibaüzenetet adni, de ezt majd később megtesszük.

Teszteljük a $(56 + 47) * 2$ kifejezés posztfix formájával:

```
1 kiertekel_posztfix("56 47 + 2 *")
```

Az eredmény 206. Ez elég közel van.

26.8. Kliensek és szolgáltatók

Az egyik alapvető célja az absztrakt adattípusoknak, hogy elválassza a szolgáltatót, aki írta a kódot az AAT implementálásához, és a klienst, aki használja az absztrakt adattípust. A szolgáltatónak csak amiatt kell aggódnia, hogy az implementáció tökéletes-e – összhangban az AAT specifikációjával – és nem amiatt, hogyan fogják használni.

Fordítva, a kliens **feltételezi**, hogy az AAT implementáció korrekt, és nem törődik a részletekkel. Amikor használ sz egy beépített Python típust megvan az a luxusod, hogy kizárólag kliensként gondolkodj.

Természetesen, amikor implementálsz egy absztrakt adattípust, akkor írnod kell egy kliens kódot is hogy teszteld a típust. Ebben az esetben mindkét szerepet betöltöd, ami zavaró lehet. Meg kell próbálnod nyomon követni melyik szerepben is vagy az adott pillanatban.

26.9. Szójegyzék

absztrakt adattípus (abstract data type) Egy adattípus (rendszerint objektumok kollekciója), amely egy művelet-halmazzal van definiálva, de különbözőképpen implementálható.

csomagoló (wrapper) Egy osztálydefiníció, amely implementál egy absztrakt adattípust metódusdefiníciókkal, amelyek más metódusokat hívnak néha egyszerű átalakítással. A csomagoló nem egy jelentős munka, de javítja, szabványosítja a kliens által látott interfészt.

generikus adatszerkezet (generic data structure) Egyfajta adatszerkezet, ami bármilyen típusú adatot tartalmazhat.

határoló (delimiter) Egy karakter, amivel elválaszthatunk szövegelemeket, mint az írásjelek a természetes nyelvekben.

implementáció (implementation) A kód, ami kielégíti egy interfész szemantikai és szintaktika követelményeit.

infix alak (infix) Matematikai kifejezések egy írásmódja, ahol az operátor az operandusai között van.

interfész (interface) Egy művelethalmaz, amely definiál egy absztrakt adattípust.

kliens (client) Egy absztrakt adattípust használó program (vagy annak készítője).

nyelvtani elemzés (parse) Egy sztring vagy szövegelem olvasása és nyelvtani szerkezetének elemzése.

posztfix alak (postfix) Matematikai kifejezések egy írásmódja, ahol az operátor az operandusai után helyezkedik el.

szolgáltató (provider) Egy absztrakt adattípust implementáló program (vagy annak készítője).

szövegelem (token) Karakterhalmaz, amelyet egységként kezelünk nyelvtani elemzés céljából. Olyan, mint a természetes nyelvek szavai.

26.10. Feladatok

1. Alkalmazd a posztfix algoritmust az $1\ 2\ +\ 3\ *$ kifejezésre! Ez a példa szemlélteti a posztfix alak egyik előnyét – nem szükséges zárójeleket használni a műveletek sorrendjének befolyásolásához. Ahhoz, hogy ugyanazt az eredményt kapjuk infix alak esetén, ezt kell írunk: $(1\ +\ 2)\ * \ 3$.
2. Írj egy posztfix kifejezést, ami egyenértékű az $1\ +\ 2\ *\ 3$ infix kifejezéssel!

27. fejezet

Sorok

Ez a fejezet két absztrakt adattípust (AAT-t) mutat be: a sort és a prioritásos sort. A való életben a **sor** valamire váró emberek sorozatát jelenti. A legtöbb esetben az első ember a sorban az, akit először kiszolgálnak. Habár vannak kivételek. A repülőtereken a sor közepéről előrehívják azokat, akiknek a gépe hamarosan indul. A supermarketekben, egy udvarias személy maga elé engedhet valakit, aki csak egy-két dolgot vásárol.

A szabályt, amely megmondja, hogy ki a következő **sorbanállási rendnek** hívjuk. A legegyszerűbb sorbanállási rend a „First In, First Out” vagy röviden FIFO, vagyis amikor az először érkező távozik először. A legáltalánosabb sorbanállási rend a **prioritásos sor**, amiben minden személynek van egy prioritása, és a legnagyobb prioritású személy mindig a következő, tekintet nélkül az érkezési sorrendre. Azt mondjuk ez a legáltalánosabb rendszer, mert a prioritás bármin alapulhat: felszállásig hátralévő időn, a vásárlandó termékek számán vagy a személy fontosságán. Természetesen nem minden sorbanállási rend igazságos, a pártatlanság nézőpont kérdése.

A sor AAT és a prioritásos sor AAT ugyanazokat a műveleteket használja. A különbség a műveletek értelmezésében van: a sor a FIFO rendet követi, a prioritásos sor pedig (ahogy a neve is mutatja) a prioritáson alapszik.

27.1. A sor AAT

A sor AAT a következő műveletekkel definiálható:

__init__ Egy új üres sor inicializálása.

put Egy elemet hozzáad a sorhoz.

get Eltávolít egy elemet a sorból, és visszatér vele. A visszaadott elem az, amely először került be a sorba.

ures_e Ellenőrzi, vajon üres-e a sor.

27.2. Láncolt sor

A sor AAT első implementációját, amelyet megnézünk, **láncolt sornak** hívják, mert ez összeláncolt Csomópont objektumokból épül fel. Itt az osztály definíciója:

```
1 class Sor:
2     def __init__(self):
3         self.hossz = 0
4         self.fej = None
5
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
6     def ures_e(self):
7         return self.hossz == 0
8
9     def put(self, adatresz):
10        csomopont = Csomopont(adatresz)
11        if self.fej is None:
12            # Ha a sor üres, az új elem lesz az első
13            self.fej = csomopont
14        else:
15            # Keresd meg az utolsó elemet a listában
16            utolso = self.fej
17            while utolso.kovetkezo:
18                utolso = utolso.kovetkezo
19            # Fűzd hozzá a csomópontot
20            utolso.kovetkezo = csomopont
21        self.hossz += 1
22
23    def get(self):
24        adatresz = self.fej.adatresz
25        self.fej = self.fej.kovetkezo
26        self.hossz -= 1
27        return adatresz
```

Az `ures_e` és a `get` metódus ugyanaz, mint a láncolt listák esetén. A `beszúr` metódus egy kicsit komplikáltabb.

Az új elemet a lista végére akarjuk beszúrni. Ha a sor üres, a fejet az új csomópontra állítjuk. Különbön bejárjuk a listát, megkeresve az utolsó elemet, és az új elemet ez után fűzzük. Könnyen azonosíthatjuk az utolsó elemet azáltal, hogy a `kovetkezo` attribútuma `None` értékű.

Egy megfelelően kialakított `Sor` objektumnak két invariánsa van. A `hossz` értéke a sorban lévő csomópontok száma, és az utolsó csomópont `kovetkezo` mezője, mely mindig `None`. Győződj meg arról, hogy a fenti metódusok helyesen kezelik-e az invariánsokat!

27.3. Teljesítmény jellemzők

Normális esetben, amikor meghívunk egy metódust, nem érdekelnek bennünket az implementáció részletei. Azonban van egy részlet, amit szeretnénk tudni: a metódus teljesítmény jellemzői. Mennyi a futási idő, és hogyan változik a kollekció elemek számának növekedésével?

Először nézzük a `get` metódust. Nincsenek ciklusok és függvényhívások, ami azt sugallja, hogy a metódus futás-ideje mindig ugyanannyi. Az ilyen metódusokat **konstans idejű** műveletnek hívjuk. A valóságban a metódus kicsit gyorsabb, amikor a lista üres, mivel kihagyja a feltételes utasítás törzsét, de a különbség nem jelentős.

A `put` metódus teljesítménye teljesen eltér ettől. Általános esetben be kell járnunk a teljes listát, hogy megtaláljuk az utolsó elemet.

Ez a bejárás idő egyenesen arányos a lista hosszával. Mivel a futásidő lineáris függvénye a hosszúnak, így az ilyen metódusokat **lineáris idejű** műveleteknek hívjuk. A konstans időhöz hasonlítva ez nagyon rossz.

27.4. Javított láncolt sor

Szeretnénk egy olyan `Sor` AAT implementációt, ahol a műveletek végrehajtása időben állandó. Az egyik módja ennek az, hogy módosítjuk a `Sor` osztályt úgy, hogy gondoskodunk nem csak az első elemre, hanem az utolsó elemre történő hivatkozásról is, ahogy lentebb mutatjuk.

A JavitottSor implementációja így néz ki:

```
1 class JavitottSor:
2     def __init__(self):
3         self.hossz = 0
4         self.fej = None
5         self.veg = None
6
7     def ures_e(self):
8         return self.hossz == 0
```

Eddig az egyetlen változás a veg attribútum, amelyet a put és a get metódusban is felhasználunk:

```
1 class JavitottSor:
2     ...
3     def put(self, adatresz):
4         csomopont = Csomopont(adatresz)
5         if self.hossz == 0:
6             # Ha a lista üres, az új csomópont a fej és a vég is
7             self.fej = self.veg = csomopont
8         else:
9             # Találd meg az utolsót
10            utolso = self.veg
11            # Fűzd hozzá az új csomópontot
12            utolso.kovetkezo = csomopont
13            self.veg = csomopont
14            self.hossz += 1
```

Mivel az utolso nyomon követi az utolsó csomópontot, nem kell azt keresnünk. Ennek eredményeként a metódus konstans idejű lesz.

A sebességnek ára van. Egy speciális esetet kell kezelnünk a get metódusban, az utolso attribútumot None értékre kell állítani, amikor az utolsó elemet eltávolítjuk:

```
1 class JavitottSor:
2     ...
3     def get(self):
4         adatresz = self.fej.adatresz
5         self.fej = self.fej.kovetkezo
6         self.hossz -= 1
7         if self.hossz == 0:
8             self.utolso = None
9         return adatresz
```

Ez az implementáció bonyolultabb, mint az egyszerű láncolt lista implementáció, és nehezebb is demonstrálni a helyességét. Az előnye az, hogy elértük a célunkat – mind a put mind a get konstans idejű metódusok.

27.5. Prioritásos sor

A prioritásos sor AAT-nak ugyanaz az interfésze, mint az egyszerű sor AAT-nak, de más a szemantikája. Az interfész ismét ez:

__init__ Egy új üres sor inicializálása.

put Egy elemet hozzáad a sorhoz.

get Eltávolít egy elemet a sorból, és visszatér vele. A visszaadott elem az, amelynek a legmagasabb a prioritása.

üres_e Ellenőrzi, vajon üres-e a sor.

A szemantikai különbség az, hogy az eltávolított elem nem szükségképpen az először beszúrt elem, hanem az az elem, amelyiknek a prioritása a legmagasabb. Az, hogy mi is a prioritás, és hogyan is kell azokat összehasonlítani az nem az implementáció része. Ez attól függ, melyen elemek vannak a sorban.

Például, ha sor elemeinek van neve, akkor használhatjuk az ABC sorrendet. Ha ezek bowling pontok, akkor haladhatunk a legnagyobbtól a legkisebbig, de ha ezek golf eredmények, akkor az alacsonyabbtól a magasabbig kell haladnunk. Ha össze tudjuk hasonlítani a sorbeli elemeket, akkor meg tudjuk találni és el tudjuk távolítani a legnagyobb prioritású elemet.

A prioritásos sornak az alábbi implementációja tartalmaz egy attribútumot, egy Python listát, amely tárolja a sor elemeit.

```
1 class PrioritasosSor:
2     def __init__(self):
3         self.elemek = []
4
5     def ures_e(self):
6         return not self.elemek
7
8     def put(self, elem):
9         self.elemek.append(elem)
```

Az inicializáló metódus, az `ures_e` és a `put` metódusok mindannyian a lista műveletek csomagolásai. Az egyetlen érdekes metódus a `get`:

```
1 class PrioritasosSor:
2     ...
3     def get(self):
4         maxi = 0
5         for i in range(1, len(self.elemek)):
6             if self.elemek[i] > self.elemek[maxi]:
7                 maxi = i
8         elem = self.elemek[maxi]
9         del self.elemek[maxi]
10        return elem
```

Minden egyes iteráció kezdetén a `maxi` tárolja az **eddig** legnagyobb (legmagasabb prioritású) elem indexét. Minden cikluslépésben a program összehasonlítja az `i`. elemet az aktuális csúcstartóval. Ha az új elem nagyobb, akkor a `maxi` értéke `i` lesz.

Amikor a `for` utasítás befejeződik, a `maxi` megadja a legnagyobb elem indexét. Ezt az elemet távolítjuk el, és ezzel térünk vissza.

Teszteljük az implementációt:

```
1 ...
2 ps = PrioritasosSor()
3 for szam in [11, 12, 14, 13]:
4     ps.put(szam)
5 while not ps.ures_e():
6     print(ps.get())
```

A kimenet: 14, 13, 12, 11.

Ha a sor egyszerű számokat vagy sztringeket tartalmaz, akkor azok numerikus vagy alfabetikus sorrendben lesznek eltávolítva, kezdve a legnagyobbval haladva a legkisebb felé. A Python megtalálja a legnagyobb egészet vagy sztringet, mert használni tudja a beépített összehasonlító operátorokat.

Ha a sor egy objektum típust tartalmaz, akkor biztosítani kell a `__gt__` metódust. Amikor a `get` használja a `>` operátort az elemek összehasonlításához, akkor meghívja a `__gt__` metódust az egyik elemre, és átadja a másikat paraméterként. Amíg a `__gt__` metódus jól működik, addig a prioritásos sor is működni fog.

27.6. A Golfozo osztály

Egy szokatlan módon definiált prioritással rendelkező objektum példajaként, implementáljunk egy `Golfozo` nevű osztályt, amely golfozók nevét és pontszámát követi nyomon. Szokás szerint kezdjük az `__init__` és a `__str__` definíciójával:

```
1 class Golfozo:
2     def __init__(self, nev, pont):
3         self.nev = nev
4         self.pont = pont
5
6     def __str__(self):
7         return "{0:16}: {1}".format(self.nev, self.pont)
```

Az `__str__` a `format` metódust használja, hogy a neveket és a pontokat szépen oszlopba rendezze.

Ezután definiáljuk a `__gt__` metódust, ahol az alacsonyabb pontszám nagyobb prioritást kap. Mint mindig, a `__gt__` `True` értékkel tér vissza, ha a `self` nagyobb, mint a másik, és `False` értékkel egyébként.

```
1 class Golfozo:
2     ...
3     def __gt__(self, másik):
4         return self.pont < másik.pont      # A kevesebb több
```

Most kész vagyunk a prioritásos sor tesztelésére a `Golfozo` osztály segítségével:

```
1 tiger = Golfozo("Tiger Woods", 61)
2 phil = Golfozo("Phil Mickelson", 72)
3 hal = Golfozo("Hal Sutton", 69)
4
5 ps = PrioritasosSor()
6 for g in [tiger, phil, hal]:
7     ps.put(g)
8
9 while not ps.ures_e():
10     print(ps.get())
```

A kimenet ez lesz:

```
Tiger Woods      : 61
Hal Sutton       : 69
Phil Mickelson   : 72
```

27.7. Szójegyzék

konstans idejű (constant time) Egy olyan művelet jellemzője, amelynek a futásideje nem függ az adatszerkezet méretétől.

FIFO (First In, First Out) Egy sorbanállási rend, amelyben az először érkező elemet távolítjuk el hamarabb.

lineáris idejű (linear time) Egy olyan művelet jellemzője, amelynek a futásideje lineáris függvénye az adatszerkezet méretének.

láncolt sor (linked queue) Egy láncolt listát használó sor implementáció.

prioritásos sor (priority queue) Egy olyan sorbanállási rendet használó sor, ahol minden elemnek van egy külső tényező által definiált prioritása. A legmagasabb prioritású elemet távolítjuk el először. Definiálhatjuk úgy is, mint egy absztrakt adatszerkezetet, amelyen az előbbi műveleteket hajthatjuk végre.

sor (queue) Valamiféle kiszolgálásra váró objektumok egy rendezett halmaza. Jelenthet olyan AAT-t is, amelyen a sornak megfelelő műveletek hajthatók végre.

sorbanállási rend (queueing policy) A szabályok, amelyek meghatározzák, hogy a sor melyik eleme legyen eltávolítva legközelebb.

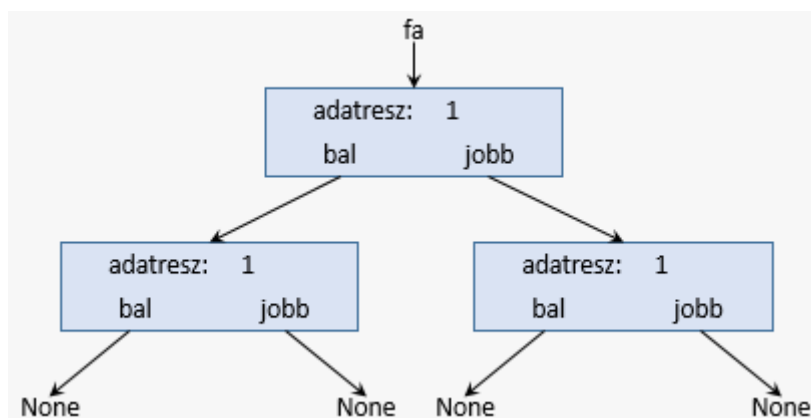
27.8. Feladatok

1. Írj egy sor AAT implementációt Python lista segítségével! Hasonlítsd össze ennek az implementációnak és a `JavitottSor` implementációnak a teljesítményét a sor hosszának egy adott tartományában!
2. Írj egy láncolt listán alapuló prioritásos sor AAT implementációt! Rendezetten kellene tartanod a listát, így az eltávolítás konstans idejű lesz. Hasonlítsd össze ennek az implementációnak és egy egyszerű Python listán alapuló implementációnak a teljesítményét!

28. fejezet

Fák

Mint ahogy a láncolt listák, úgy a fák is csomópontokból épülnek fel. Egy gyakran használt fa típus a **bináris fa**, amelyben mindegyik csomópont tartalmaz két referenciát, amelyek másik csomópontokra hivatkoznak (esetleg `None` értékűek). Ezekre a referenciákra úgy gondolunk, mint jobb és bal oldali részfákra. Akárcsak a lista csomópontok, a fák csomópontjai is tartalmaznak adatrészt. Egy fára vonatkozó állapotdiagram így néz ki:



Azért, hogy a kép túlszűfoltását elkerüljük, gyakran leghagyjuk a `None` értékeket.

A fa felső elemét (amelyre a `fa` most hivatkozik) **gyökér** elemnek hívjuk. Hogy megtartsuk a fa metaforát, a null referenciával rendelkező csomópontokat **levél** elemeknek nevezzük, és a többi csomópontot ágaknak. Talán furcsának tűnhet, hogy a képen a gyökér van felül és a levelek lent, de nem ez a legfurcsább dolog.

Hogy még rosszabbá tegyük a dolgokat, az informatikusok egy másik metaforát is belekevernek: a családfát. Egy felsőbb elemet néha **szülő** néven is nevezzük, és az elemeket, amelyekre hivatkozik **gyerek** csomópontnak. Az azonos szülőtől származó gyerekek pedig a **testvérek**.

Végül van még egy geometriai alapú szóhasználat is. Már említettük a jobbra és balra irányt, de van fel (a szülő / gyökér felé) és le (a gyermek / levél felé) irány is. Emellett mindegyik elem, amelyik ugyanolyan távol van a gyökértől egy **szintet** alkot.

Talán nincs szükségünk több metaforára a fákról, de léteznek továbbiak is.

Mint a láncolt listák, a fák is rekurzív adatszerkezetek, mert rekurzívan definiálhatóak. Egy fa vagy

1. egy üres fa, `None` értékkel reprezentálva, vagy
2. egy csomópont, amely tartalmaz egy objektum referenciát (adatrészt) és két fa referenciát.

28.1. Fák építése

Egy fa felépítésének folyamata hasonló egy láncolt lista összerakásának folyamatához. Minden egyes konstruktor hívás létrehoz egy egyedülálló csomópontot.

```
1 class Fa:
2     def __init__(self, adatresz, bal=None, jobb=None):
3         self.adatresz = adatresz
4         self.bal = bal
5         self.jobb = jobb
6
7     def __str__(self):
8         return str(self.adatresz)
```

Az `adatresz` bármilyen típusú lehet, de a `bal` és a `jobb` paraméternek fa csomópontnak kell lennie. A `bal` és `jobb` érték opcionális, az alapértelmezett érték a `None` lesz.

Egy csomópont kiírásához csak az adatrészt kell megjeleníteni.

Az egyik módja egy fa alkotásának az alaptól felfelé való építkezés. Először a gyerekeknek foglalj helyet:

```
1 bal = Fa(2)
2 jobb = Fa(3)
```

Aztán hozd létre a szülő csomópontot, és kapcsold a gyerekekhez:

```
1 fa = Fa(1, bal, jobb)
```

Írhatjuk ezt a kódot sokkal tömörebben is egymásba ágyazva a konstruktor hívásokat:

```
1 fa = Fa(1, Fa(2), Fa(3))
```

Akár így, akár úgy, az eredmény a fejezet elején lévő fa.

28.2. A fák bejárása

Bármikor, amikor egy új adatszerkezetet látsz, az első kérdésednek annak kell lennie, hogy „Hogyan járhatom be?” A legtermészetesebb módja egy fa bejárásának rekurzív. Például, ha a fa egészet tartalmaz adatként, akkor az alábbi függvény ezek összegével tér vissza:

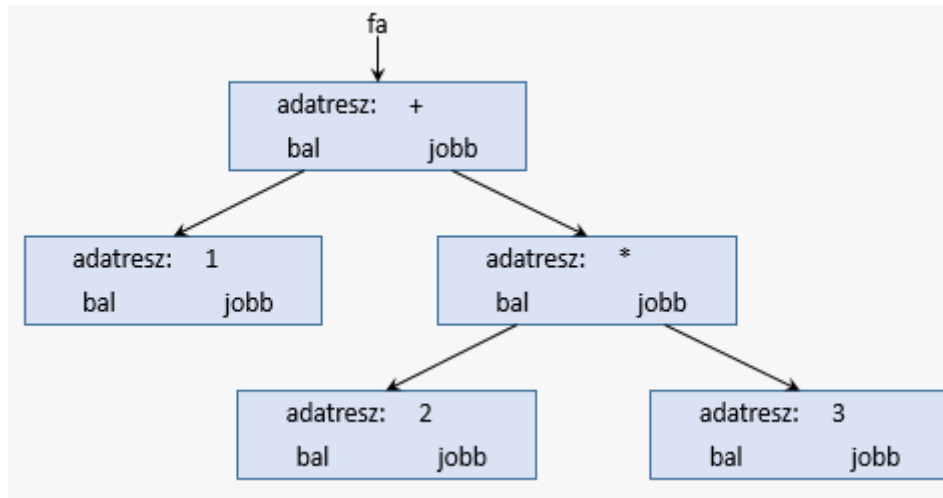
```
1 def osszeg(fa):
2     if fa is None: return 0
3     return osszeg(fa.bal) + osszeg(fa.jobb) + fa.adatresz
```

Az alap eset egy üres fa, amely nem tartalmaz adatrészt, így az összeg nulla. A rekurzív lépés két rekurzív hívást tartalmaz a két gyerek részfa összegének meghatározásához. Amikor a rekurzív hívások befejeződnek, az eredményükhöz hozzáadjuk a szülő adatrészét, és ezzel térünk vissza.

28.3. Kifejezésfák

A fa egy természetes megjelenítési módja a kifejezéseknek. Akárcsak más írásformák ez is félreérthetetlen módon reprezentálja a számítás menetét. Például, az $1 + 2 * 3$ infix kifejezés félreérthető, hacsak nem tudjuk azt, hogy a szorzást az összeadás előtt kell elvégezni.

Ez a kifejezésfa ugyanezt a számítást reprezentálja:



A kifejezésfa csomópontjai lehetnek operandusok, mint az 1 és a 2 vagy operátorok, mint a + és a *. Az operandusok levél elemek, az operátorok pedig az operandusokra való hivatkozásokat tartalmazznak. (Ezek közül mindegyik operátor **bináris**, azaz pontosan két operandusa van.)

Így építhetjük fel az ilyen fákat:

```
fa = Fa("+", Fa(1), Fa("*", Fa(2), Fa(3)))
```

Az ábrára pillantva nem kérdés, hogy mi a műveletek sorrendje. A szorzás történik meg először, azért, hogy meghatározzuk az összeadás második operandusát.

A kifejezésfákat sok helyen használhatjuk. Az ebben a fejezetben bemutatandó példa a kifejezéseket prefix, postfix vagy infix alakra hozza. Hasonló fákat alkalmaznak a fordítóprogramok a forráskódok nyelvtani elemzéséhez, optimalizálásához és lefordításához.

28.4. Fabejárás

Bejárhatunk egy kifejezésfát és kiírhatjuk a tartalmát, mondjuk így:

```
1 def kiir_fa(fa):
2     if fa is None: return
3     print(fa.adatresz, end=" ")
4     kiir_fa(fa.bal)
5     kiir_fa(fa.jobb)
```

Más szóval, írd ki először a gyökér tartalmát, aztán a teljes baloldali részfát, végül az egész jobb oldali részfát. A fabejárás ezen módja az ún. **preorder** bejárás, mert a gyökér tartalma a gyerekek tartalma *előtt* jelenik meg. Az előbbi példában a kimenet ez:

```
1 fa = Fa("+", Fa(1), Fa("*", Fa(2), Fa(3)))
2 kiir_fa(fa) # Kimenet: + 1 * 2 3
```

Ez a formátum különbözik mind a posztfix, mind az infix alaktól, vagyis ez egy újabb írásmód, aminek a neve **prefix**, amiben az operátorok az operandusaik előtt jelennek meg.

Sejtheted, hogy ha különböző sorrendben járod be a fát, akkor a kifejezések különböző írásmódját kapod meg. Például, ha a részfákat írod ki először, és csak azután a gyökeret, akkor azt kapod:

```
1 def kiir_fa_postorder(fa):
2     if fa is None: return
3     kiir_fa_postorder(fa.bal)
4     kiir_fa_postorder(fa.jobb)
5     print(fa.adatresz, end=" ")
```

A postfix alakú eredmény 1 2 3 * +. Ezt a fajta bejárást hívjuk **postorder** bejárásnak.

Végül, az **inorder** bejáráshoz írd ki a baloldali részfat, aztán a gyökeret, majd a jobboldali részfat.

```
1 def kiir_fa_inorder(fa):
2     if fa is None: return
3     kiir_fa_inorder(fa.bal)
4     print(fa.adatresz, end=" ")
5     kiir_fa_inorder(fa.jobb)
```

Az eredmény 1 + 2 * 3, ami a kifejezés infix alakja.

A pontosság kedvéért hangsúlyoznunk kell, hogy figyelmen kívül hagyunk egy fontos problémát. Néha, amikor egy kifejezést infix alakban írunk fel, zárójeleket kell alkalmaznunk, hogy megőrizzük a műveletek sorrendjét. Tehát az inorder bejárás nem igazán megfelelő infix kifejezések generálásához.

Mindazonáltal, néhány javítással a kifejezésfa és a rekurzív fabejárás egy általános lehetőséget biztosít egy kifejezés egyik alakból a másikba történő átalakításához.

Ha inorder bejárás során nyomon követjük a fa szintjét, amelyen éppen tartózkodunk, akkor a fa egy grafikus reprezentációját állíthatjuk elő:

```
1 def kiir_fa_behuzva(fa, szint=0):
2     if fa is None: return
3     kiir_fa_behuzva(fa.jobb, szint+1)
4     print(" " * szint + str(fa.adatresz))
5     kiir_fa_behuzva(fa.bal, szint+1)
```

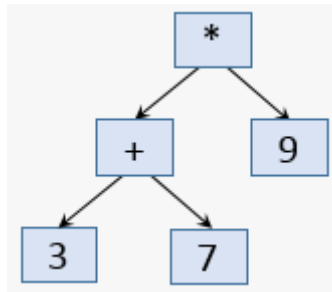
A szint paraméter nyomon követi, hol vagyunk a fában. Alapértelmezetten ez kezdetben 0. Minden alkalommal, amikor egy rekurzív hívást hajtunk végre, akkor szint+1 értékét adjuk át, mert a gyerek szintje mindig eggyel nagyobb, mint a szülőé. Minden elem szintenként két szóközzel van behúzva. A kiir_fa_behuzva(fa) függvényhívás eredménye a példában szereplő fa esetén:

```
    3
  *
 2
+
1
```

Ha oldalról nézed a kimenetet, akkor az eredeti ábrához hasonló egyszerűsített verziót látsz.

28.5. Kifejezésfák felépítése

Ebben az alfejezetben infix kifejezéseket fogunk elemezni, és felépítjük a megfelelő kifejezésfákat. Például, a (3 + 7) * 9 kifejezés a következő ábrát eredményezi:



Vedd észre, hogy egy egyszerűsített diagramunk van, amelyben kihagytuk az attribútumok nevét.

A nyelvtani elemző, amelyet most megírunk, olyan kifejezéseket kezel, amelyek számokat, zárójeleket valamint a + és a * operátorokat tartalmazzák. Feltesszük, hogy a bemeneti lista már szövegelemekre azaz tokenekre van bontva egy Python listában (ennek a listának az előállítása egy további gyakorló feladat számodra). A $(3 + 7) * 9$ kifejezés esetén a token lista a következő:

```
[ "(", 3, "+", 7, ")", "*", 9, "vege"]
```

A vege szövegelem hasznos lehet a lista végén túli olvasás megelőzéséhez.

Az első függvény, amelyet megírunk a `torol_elso`, amely kap egy token listát és egy elvárt token paraméterként. Összehasonlítja a szövegelemet a lista első elemével: ha megegyeznek, akkor eltávolítja az adott token a lista elejéről, és visszatér egy `True` értékkel; különben `False` értéket ad vissza:

```
1 def torol_elso(token_lista, elvart):
2     if token_lista[0] == elvart:
3         del token_lista[0]
4         return True
5     return False
```

Mivel a `token_lista` egy módosítható objektum, az itt végrehajtott módosítás mindenütt látható lesz, ahol ugyanerre az objektumra hivatkozunk.

A következő függvény a `szamot_ad`, amely az operandusokat kezeli. Ha a következő szövegelem a `token_lista`-ban egy szám, akkor eltávolítja ezt, és visszaad egy levél csomópontot, amely az adott számot tartalmazza; különben `None` értékkel tér vissza.

```
1 def szamot_ad(token_lista):
2     x = token_lista[0]
3     if type(x) != type(0): return None
4     del token_lista[0]
5     return Fa(x, None, None)
```

Mielőtt továbbmennénk, tesztelnünk kellene a `szamot_ad` függvényt izoláltan. Számok egy listáját rendeljük a `token_lista` azonosítóhoz, és távolítsuk el az első elemet, írassuk ki az eredményt és írassuk ki a lista megmaradt elemeit:

```
1 token_lista = [9, 11, "vege"]
2 x = szamot_ad(token_lista)
3 kiir_fa_postorder(x) # Kimenet: 9
4 print(token_lista) # Kimenet: [11, "vege"]
```

A következő metódus, amire szükségünk van a szorzat, amely felépít egy kifejezésfát a szorzás számára. Egy egyszerű szorzásnak két szám operandusa van, mint például a $3 * 7$ esetén.

Itt a `szorzat` egy változata, amely kezeli az egyszerű szorzást.


```
1 def szorzat(token_lista):
2     a = szamot_ad(token_lista)
3     if torol_elso(token_lista, "*"):
4         b = szamot_ad(token_lista)
5         return Fa(" ", a, b)
6     return a
```

Feltéve, hogy a `szamot_ad` sikeres és egy egyke (egyetlen objektumot tartalmazó osztályhoz tartozó) fát ad vissza, hozzárendeljük az első operandust az `a`-hoz. Ha a következő karakter `*`, beolvassuk a következő számot, és felépítjük a kifejezésfát `a` és `b` értékekkel és az operátorral.

Ha következő karakter bármi más, akkor csak egyszerűen visszatérünk az `a` levélelemmel. Itt van két példa:

```
1 token_lista = [9, "*", 11, "vege"]
2 fa = szorzat(token_lista)
3 kiir_fa_postorder(fa)
```

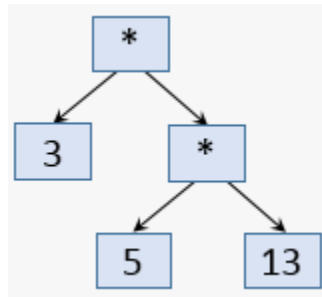
A kiment: 9 11 *.

```
1 token_lista = [9, "+", 11, "vege"]
2 fa = szorzat(token_lista)
3 kiir_fa_postorder(fa)
```

A kimenet ebben az esetben: 9.

A második példa azt vonja maga után, hogy az egyedülálló operandus egyfajta szorzat. Ez a definíciója a szorzásnak ellentétes az ösztöneinkkel, de hasznosnak bizonyul.

Most foglalkoznunk kell az összetett szorzattal, mint például a $3 * 5 * 13$. Ezt a kifejezést szorzat szorzataként kezelhetjük, nevezetesen $3 * (5 * 13)$. Az eredményként előálló fa ez lesz:



A `szamot_ad` függvény kis változtatásával tetszőlegesen hosszú szorzatot tudunk kezelni:

```
1 def szorzat(token_lista):
2     a = szamot_ad(token_lista)
3     if torol_elso(token_lista, "*"):
4         b = szorzat(token_lista)           # Ez a sor változott
5         return Fa(" ", a, b)
6     return a
```

Más szavakkal a szorzat vagy egy egyke vagy egy fa lehet, amelynek a gyökerében egy `*` van, bal oldalán egy szám, és egy szorzat a jobb oldalán. Ez a fajta rekurzív definíció ismerős kell, hogy legyen.

Teszteljük az új verziót egy összetett szorzattal:

```
1 token_lista = [2, "*", 3, "*", 5, "*", 7, "vege"]
2 fa = szorzat(token_lista)
3 kiir_fa_postorder(fa)
```

A kimenet ekkor: 2 3 5 7 * * *.

A következő lépésben az összegek nyelvtani elemzésének képességét adjuk hozzá a programhoz. Ismét egy nem ösztönös definíciót adunk az összegre. Számunkra, az összeg egy fa lehet + operátorral a gyökerében, egy szorzat a bal oldalon, és egy összeg a jobb oldalon. Emellett az összeg lehet csak egyszerűen egy szorzat.

Ha játszani akarnál ezzel a definícióval, ennek van egy jó tulajdonsága: reprezentálhatunk minden kifejezést (zárójelek nélkül) szorzatok összegeként. Ez a tulajdonság az alapja a nyelvtani elemző algoritmusunknak.

Az `osszeg` függvény egy fát próbál felépíteni egy szorzattal a bal oldalon és egy összeggel a jobb oldalon. Azonban, ha ez nem talál + jelet, akkor egy szorzatot épít fel.

```
1 def osszeg(token_lista):
2     a = szorzat(token_lista)
3     if torol_elso(token_lista, "+"):
4         b = osszeg(token_lista)
5         return Fa("+", a, b)
6     return a
```

Teszteljük ezzel: 9 * 11 + 5 * 7:

```
1 token_lista = [9, "*", 11, "+", 5, "*", 7, "vege"]
2 fa = osszeg(token_lista)
3 kiir_fa_postorder(fa)
```

A kimenet ez lesz: 9 11 * 5 7 * +.

Majdnem kész vagyunk, de még kezelnünk kell a zárójeleket. Ahol a kifejezésben szerepelhet egy szám, ott egy teljesen zárójelezett összeg is állhat. Csak módosítanunk kell a `szaot_ad` függvényt, hogy kezelje a **rész kifejezéseket**:

```
1 def szaot_ad(token_lista):
2     if torol_elso(token_lista, "("):
3         x = osszeg(token_lista)          # Foglalkozz a rész kifejezéssel
4         torol_elso(token_lista, ")")      # Távolítsd el a záró zárójelet
5         return x
6     else:
7         x = token_lista[0]
8         if type(x) != type(0): return None
9         del token_lista[0]
10        return Fa(x, None, None)
```

Teszteljük ezt a kódot ezzel a kifejezéssel 9 * (11 + 5) * 7:

```
1 token_lista = [9, "*", "(", 11, "+", 5, ")", "*", 7, "vege"]
2 fa = osszeg(token_lista)
3 kiir_fa_postorder(fa)
```

Ezt kapjuk eredményül: 9 11 5 + 7 * *.

Az elemzőnk jól kezeli a zárójeleket, az összeadás a szorzás előtt történik.

A program végső verziójában jó ötlet lenne a `szaot_ad` függvénynek egy másik nevet adni, amely jobban leírja az új szerepét.

28.6. Hibák kezelése

A nyelvtani elemzőnkben végig azt feltételeztük, hogy a kifejezés megfelelő formában van. Például, amikor elérünk egy rész kifejezése végére, feltételeztük, hogy a következő karakter egy záró zárójel. Ha egy hiba van a kifejezésben

és a következő karakter valami más, akkor foglalkoznunk kellene vele.

```
1 def szamot_ad(token_lista):
2     if torol_elso(token_lista, "("):
3         x = osszeg(token_lista)
4         if not torol_elso(token_lista, ")"):
5             raise ValueError("Hiányzó záró zárójel!")
6         return x
7     else:
8         # A függvény további részét most kihagyjuk
```

A `raise` utasítás egy általunk létrehozott kivétel objektumot dob. Ebben az esetben egyszerűen csak a legmegfelelőbb típusú beépített kivételt, amit találtunk, de jó ha tudod, hogy hozhatsz létre saját specifikusabb programozó által definiált kivételt is, ha szükséged van rá. A `szamot_ad` függvényben vagy bármelyik másik korábbi függvényben kezeld a kivételt, aztán a program mehet tovább. Különben a Python kiír egy hibaüzenetet és megáll.

28.7. Az állati fa

Ebben az alfejezetben egy kis programot fogunk fejleszteni, amely egy fát használ a tudásbázisa reprezentálásához.

A program kommunikál a felhasználóval, hogy felépítsen egy kérdésekből és állatnevekből álló fát. Itt egy egyszerű futás:

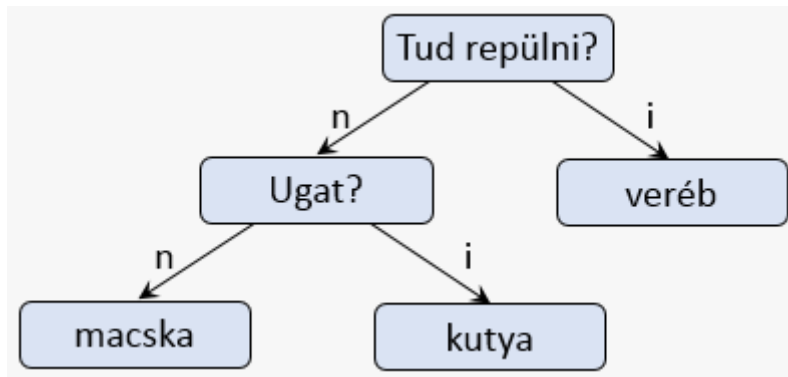
```
Egy állatra gondolsz? i
Ez a veréb? n
Mi az állat neve? kutya
Milyen kérdéssel lehet megkülönböztetni őket: kutya, veréb? Tud repülni
Ha az állat a kutya lenne, mi lenne a válasz? n

Egy állatra gondolsz? i
Tud repülni? n
Ez a kutya? n
Mi az állat neve? macska
Milyen kérdéssel lehet megkülönböztetni őket: macska, kutya? Ugat
Ha az állat a macska lenne, mi lenne a válasz? n

Egy állatra gondolsz? i
Tud repülni? n
Ugat? i
Ez a kutya? i
Kitaláltam!

Egy állatra gondolsz? n
```

Itt a fa, amelyet a párbeszéd alapján felépítettünk:



Minden kör elején a program a fa tetejével kezd, és felteszi az első kérdést. A választól függően a jobb vagy a bal oldali gyerekhez mozdul, és addig folytatja ezt, amíg el nem ér egy levél elemet. Ezen a ponton mond egy tippet. Ha a tipp helytelen, akkor megkéri a felhasználót, hogy mondja meg az új állat nevét és egy kérdést, amely alapján elkülöníthető a (rossz) tipp a megfejtéstől. Ezután hozzáad egy csomópontot a fához az új kérdéssel és az új állattal.

Itt a kód:

```
1 def igen(eldontendo):
2     valasz = input(eldontendo).lower()
3     return valasz[0] == "i"
4
5 def allat():
6     # Kezdj egy egykével!
7     gyoker = Fa("veréb")
8
9     # Ciklus, amíg a felhasználó ki nem lép
10    while True:
11        print()
12        if not igen("Egy állatra gondolsz? "): break
13
14        # Sétálj a fában!
15        fa = gyoker
16        while fa.bal is not None:
17            kerdes = fa.adatresz + "? "
18            if igen(kerdes):
19                fa = fa.jobb
20            else:
21                fa = fa.bal
22
23        # Tippetelj!
24        tipp = fa.adatresz
25        kerdes = "Ez a " + tipp + "? "
26        if igen(kerdes):
27            print("Kitaláltam!")
28            continue
29
30        # Szerezz új információt!
31        kerdes = "Mi az állat neve? "
32        allat = input(kerdes)
33        kerdes = "Milyen kérdéssel lehet megkülönböztetni őket: {0}, {1}? "
34        donto_kerdes = input(kerdes.format(allat, tipp))
35
36        # Bővítsd a fát az új információval!
37        fa.adatresz = donto_kerdes
38        kerdes = "Ha az állat a {0} lenne, mi lenne a válasz? "
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
39     if igen(kerdes.format(állat)):  
40         fa.bal = Fa(tipp)  
41         fa.jobb = Fa(állat)  
42     else:  
43         fa.bal = Fa(állat)  
44         fa.jobb = Fa(tipp)
```

Az `igen` függvény egy segítő, felteszi az eldöntendő kérdést és inputot vár a felhasználótól. Ha bemenet *i* vagy *I* betűvel kezdődik, akkor `True` vagyis igaz értékkel tér vissza.

Az `állat` külső ciklusának feltétele `True`, ami azt jelenti, hogy folytasd addig, amíg a `break` utasítás végre nem hajtódik, azaz addig, amíg a felhasználó már nem egy állatra gondol.

A belső `while` ciklus a fában fentről lefelé halad, a felhasználó visszajelzései alapján.

Amikor egy új csomópontot adunk a fához, az új kérdés felülírja az adatrészt, és a csomópont két gyereke az új állat és az eredeti adatrész lesz.

Egy hiányossága a programnak, hogy amikor kilép, elfelejt mindent, amit nagy körültekintéssel megtanult. Ennek a problémának a kijavítása egy jó feladat számodra.

28.8. Szójegyzék

bináris operátor (binary operator) Egy operátor, amelynek két operandusa van.

bináris fa (binary tree) Egy fa, amelyben minden csomópont 0, 1 vagy 2 másik csomópontra hivatkozik.

gyerek (child) Egy csomópont, amelyre egy másik hivatkozik.

levél (leaf) A levelek a legalsó csomópontok a fában, amelyeknek nincs gyerekük.

szint (level) A gyökértől azonos távolságra lévő csomópontok halmaza.

szülő (parent) Egy csomópont, amely hivatkozik egy másik csomópontra.

postorder A fabejárás azon módja, amikor minden csomópont gyerekeit hamarabb dolgozzuk fel, mint magát az adott csomópontot.

prefix írásmód (prefix notation) A matematikai kifejezések egyfajta lejegyzési módja, ahol minden operátor az operandusai előtt jelenik meg.

preorder A fabejárás azon módja, amikor minden csomópontot hamarabb látogatunk meg, mint a gyerekeit.

gyökér (root) A fa legfelső, azaz szülő nélküli eleme.

testvérek (siblings) Azonos szülővel rendelkező csomópontok.

rész kifejezés (subexpression) Egy zárójelben lévő kifejezés, ami úgy viselkedik, mint egy nagyobb kifejezés egyik operandusa.

28.9. Feladatok

1. Módosítsd a `kiir_fa_inorder` függvényt úgy, hogy tegyen zárójelet minden operátor és a hozzá tartozó operanduspár köré! A kimenet helyes vagy félreérthető? Mindig szükséges a zárójel?
2. Írj egy függvényt, amely kap egy kifejezés sztringet, és visszaad egy token listát!

3. Találj olyan pontokat a kifejezésfa függvényekben, ahol hibák merülhetnek fel, és adj meg megfelelő `raise` utasításokat! Teszteld a kódot nem megfelelő formátumban megadott kifejezésekkel!
4. Gondolkodj azon, milyen módokon tudnád elmenteni az állatokkal kapcsolatos tudást egy fájlba! Implementáld ezek közül azt, amelyet a legegyszerűbbnek gondolsz!

A. függelék

Nyomkövetés

Különböző hibák fordulhatnak elő egy programban, és hasznos a gyors lenyomozásukhoz, ha meg tudjuk őket különböztetni:

1. A szintaktikai hibára akkor derül fény, amikor a Python átfordítja a forráskódot bajtkóddá. Ezek rendszerint azt jelzik, hogy valami baj van a program szintaxisával. Például: Ha lehagytuk a kettőspontot a `def` utasítás végéről az a következő nem túl bőbeszédű üzenetet eredményezi `SyntaxError: invalid syntax`.
2. A futási idejű hibákat a futtató rendszer állítja elő, ha valami rosszul megy a program futása során. A legtöbb futásidejű hiba üzenet információt tartalmaz arról, hogy hol jelent meg a hiba, és milyen függvények voltak éppen végrehajtás alatt. Például: Egy végtelen rekurzió előbb-utóbb futási hibát okoz azáltal, hogy elérjük a maximális rekurziós mélységet.
3. A szemantikai hiba olyan probléma a programmal, amikor a kód formailag helyes, le is fut, de nem jól csinál valamit. Például: Egy kifejezés nem olyan sorrendben értékelődik ki, mint azt elvártad, és egy váratlan eredményt kapsz.

A nyomkövetés első lépése, hogy kitaláljuk, milyen hibával is van dolgunk. Habár a következő szakaszok a hibatípusok szerint vannak szervezve, néhány technika több szituációban is alkalmazható.

A.1. Szintaktikai hibák

A szintaktikai hibákat rendszerint könnyű kijavítani, ha már egyszer megtaláltad őket. Sajnos a hiba üzenetek gyakran nem túl segítőkészek. A legközönségesebb üzenetek a `SyntaxError: invalid syntax` és a `SyntaxError: invalid token`, egyik sem igazán informatív.

Másrészt, az üzenet megmondja hol jelent meg a probléma a programban. Tulajdonképpen azt mondja meg, hogy a Python hol észlelt problémát, ami nem szükségképpen az a hely, ahol a hiba van. Gyakran a hiba az üzenetben megadott hely előtt van, többnyire az előző sorban.

Ha fokozatosan építet a programodat, kell, hogy legyen ötleted arra vonatkozólag, hogy hol is lehet a hiba. A legutóbb hozzáadott részben lesz.

Ha egy könyvből másolod a kódot, kezd a kódodnak és a könyv kódjának a tüzetes összehasonlításával! Ellenőriz minden fejezetet! Ugyanakkor emlékezz arra is, hogy a könyv is lehet hibás, szóval, ha olyat látsz, ami szintaktikai hibának néz ki, akkor az valószínűleg az is.

Itt van pár módja annak, hogy elkerüld a legközönségesebb szintaktikai hibákat:

1. Légy biztos abban, hogy nem egy Python kulcsszót használsz változónévként!

2. Ellenőrizd, hogy tettél-e kettőspontot minden összetett utasítás fejrészének a végére, beleértve a `for`, a `while`, a `def`, az `if` és `else` részeket!
3. Ellenőrizd, hogy a behúzásaid konzisztensek-e! Behúzhatsz szóközzel vagy tabulátorral is, de a legjobb, ha nem kevered őket. Minden szint ugyanannyival legyen behúzva!
4. Légy biztos benne, hogy minden sztring a kódban idézőjelek között van!
5. Ha több soros sztringet használsz tripla idézőjelek között, légy biztos benne, hogy a sztring megfelelően végződik-e! Egy befejezetlen sztring `invalid token` hibát eredményezhet a programod végén, vagy megtörténhet az is, hogy a programod ezt követő része a következő sztringig egyetlen nagy sztringként kezelődik. Utóbbi esetben akár az is előfordulhat, hogy egyáltalán nem keletkezik hibaüzenet.
6. Egy bezáratlan zárójel – `(`, `{`, vagy `[` – eredményezheti azt, hogy a Python a következő sorral folytatja, mintha az az aktuális utasítás része lenne. Általában ez azonnal hibát okoz a következő sorban.
7. Ellenőrizd, hogy nem a klasszikus `=` jelet használod-e a `==` helyett a feltételekben!

Ha semmi nem működik, menj tovább a következő szakaszra...

A.2. Nem tudom futtatni a programomat, akármit is csinállok

Ha a parancsértelmező azt mondja, van egy hiba a programban, de te nem látod, ez lehet amiatt, hogy te és a parancsértelmező nem ugyanazt a kódot nézitek. Ellenőrizd a fejlesztői környezetet, hogy biztosan tudd, hogy a program, amit szerkesztesz, az tényleg az a program-e, amit a Python próbál futtatni. Ha nem vagy biztos, tegyél egy nyilvánvaló és szándékos szintaktikai hibát a program elejére! Most futtasd (vagy importáld) újra! Ha a parancsértelmező nem találja az új hibát, akkor bizonyára valami baj van a fejlesztői környezet beállításával.

Ha ez történt, akkor az egyik megközelítés szerint kezd előről egy új Hello, Világ! jellegű programmal, és győződj meg arról, hogy az ismert programot futtatod-e! Ezután add az új programod részeit az aktuális munkádhoz!

A.3. Futási idejű hibák

Ha egyszer a programod szintaktikailag helyes, a Python képes importálni, és legalább el tudja kezdeni a futtatást. Mi baj történhet?

A.4. A program abszolút semmit nem csinál

Ez a hiba hétköznapi, ha a fájlod függvényeket és osztályokat tartalmaz, de nem hív meg semmit a futtatás elkezdéséhez. Ez lehet szándékos, ha csak azt tervezed, hogy importáld ezt a modult, hogy függvényeket és osztályokat szolgáltatasson.

Ha ez nem szándékos, győződj meg arról, hogy meghívsz egy függvényt a végrehajtás elkezdéséhez! Nézd meg [A végrehajtás menete](#) fejezetet is lentebb!

A.5. A programom felfüggesztődött

Ha a program megállt, és úgy néz ki, nem csinál semmit, akkor azt mondjuk felfüggesztődött. Ez gyakran azt jelenti, hogy egy végtelen ciklusba esett vagy esetleg egy végtelen rekurzióba.

1. Ha van egy bizonyos ciklus, amiről azt gyanítod, hogy az a probléma forrása, akkor tegyél egy `print` utasítást közvetlenül a ciklus elé, ami azt mondja beléptél, és egy másikat közvetlenül a ciklus után, ami a ciklusból történő kilépésről tájékoztat!
2. Futtasd a programot! Ha az első üzenetet megkapod, de a másodikat nem, akkor menj a *Végtelen ciklus* szakaszhoz lentebb!
3. Legtöbbször egy végtelen rekurzió azt okozza, hogy a program fut egy ideig, és aztán futási idejű hibát ad: `Maximum recursion depth exceeded`. Ha ez történik, menj a *Végtelen rekurzió* szakaszhoz!
4. Ha a fenti hibát kapod, de úgy gondolod, hibás a rekurzív függvény vagy metódus, akkor is használhatod a *Végtelen rekurzió* szakaszban bemutatott technikát!
5. Ha egyik lépés sem működik, akkor kezd el tesztelni a többi ciklust vagy rekurzív függvényt és metódust!
6. Ha ez sem működik, akkor talán nem érted a program végrehajtásának a menetét. Menj a *Végrehajtás menete* szakaszhoz!

A.6. Végtelen ciklus

Ha azt gondolod, van egy végtelen ciklusod, és azt hiszed tudod, melyik ciklus okozza a problémát, akkor adj egy `print` utasítást a ciklus végéhez, amely kiírja a ciklusfeltételben szereplő változóknak és magának a feltételnek az értékét!

Például:

```
1 while x > 0 and y < 0:
2     # Csinálj valamit az x változóval
3     # Csinálj valamit az y változóval
4
5     print("x: ", x)
6     print("y: ", y)
7     print("feltétel: ", (x > 0 and y < 0))
```

Most, amikor futtatod a programot, minden cikluslépésben három kimentti sort fogsz látni. A ciklusmag utolsó ismétlése során a feltételnek `False` értékűnek kell lennie. Ha a ciklus tovább megy, akkor láthatod az `x` és az `y` értékét, és így rájöhetsz, miért nem frissülnek megfelelően.

Egy fejlesztői környezetben, mint mondjuk a PyCharm, elhelyezhetünk egy töréspontot a ciklus elejére, és egyesével végigmehetünk az ismétléseken. Amíg azt tesszük megvizsgálhatjuk az `x` és `y` értékét följük húzva a kurzort.

Természetesen, mind a programozás, mind a nyomkövetés azt igényli, hogy legyen egy jó mentális modelled arról, mit is kellene az algoritmusnak csinálnia: ha nem érted minek kellene történnie az `x` és `y` értékekkel, akkor az értékek kiírása nem igazán használ. Talán a legjobb helye a nyomkövetésnek távol van a számítógéptől, ahol azon dolgozhatsz, hogy megértsd mi történik.

A.7. Végtelen rekurzió

Legtöbbször egy végtelen rekurzió azt eredményezi, hogy a program fut egy darabig, aztán egy `Maximum recursion depth exceeded` hibaüzenetet ad.

Ha sejtet, melyik függvény vagy metódus okozza a végtelen rekurziót, akkor kezd az ellenőrzést azzal, hogy meggyőződsz arról, van-e alap eset! Más szóval, lennie kell valami olyan feltételnek, ami azt eredményezi, hogy a függvény vagy metódus véget ér anélkül, hogy rekurzív hívást végezne. Ha nincs, akkor újra kell gondolnod az algoritmust, és azonosítanod kell az alap esetet!

Ha létezik alap eset, de úgy tűnik a program nem éri ezt le, akkor rakj egy `print` utasítást a függvény vagy metódus elejére, amely kiírja a paraméterek értékét! Most, amikor a programod fut, látni fogsz pár kimeneti sort, akárhányszor meghívódik a függvény vagy metódus, és látni fogod a paramétereket. Ha a paraméterek nem haladnak az alapeset felé, ki kell találnod miért nem.

Még egyszer, ha van egy fejlesztői környezeted, amely támogatja az egyszerű lépésenkénti végrehajtást, töréspontok elhelyezését és a vizsgálatot, akkor tanuld meg jól használni őket! Az a véleményünk, hogy a kódon lépésről lépésre történő végighaladás felépíti a legjobb és legpontosabb mentális modellt arról, hogyan történik a számítás. Használd, ha van!

A.8. A végrehajtás menete

Ha nem vagy biztos benne, hogyan történik a program végrehajtása, írd `print` utasításokat minden függvény elejére, amelyek megüzenik, hogy beléptél `valami-be`, ahol a `valami` a függvény neve.

Most, ha futtatod a programot, minden függvényhívás nyomot hagy.

Ha nem vagy biztos a dolгодban, lépkedj végig a programodon a nyomkövető eszközök segítségével!

A.9. Amikor futtatom a programom, egy kivételt kapok

Ha futásidőben valami rosszul megy, akkor a Python kiír egy üzenetet, amely tartalmazza a kivétel nevét, a sort, ahol a probléma megjelent, és visszakövetési információkat.

Tegyél egy töréspontot abba a sorba, ahol a kivétel bekövetkezik, és nézz körül!

A visszakövetés segít megtalálni a függvényt, amely éppen fut, és a függvényt, ami meghívta, azt amelyik *azt* hívta meg, és így tovább. Más szavakkal, felvázolja a függvényhívások útvonalt, amelyen oda jutottál, ahol vagy. Azt is magába foglalja, melyek azok a sorok, ahol ezek a hívások megtörténtek.

Az első lépés megvizsgálni a helyet, ahol a hiba jelentkezett és kitalálni mi történt. Ezek a leghétköznapiabb futási idejű hibák:

NameError Használni próbálsz egy olyan változót, ami nem létezik abban az aktuális környezetben. Emlékezz, hogy a lokális változók lokálisak. Nem hivatkozatsz rájuk azon a függvényen kívül, ahol definiáltad őket.

TypeError Számos lehetséges oka van:

1. Egy értéket nem megfelelően próbálsz meg használni. Például: indexelni egy sztringet, listát vagy rendezett `n`-est egy nem egész jellegű mennyiséggel.
2. Nem egyezik a formátum sztring és a konverzióra átadott tétel. Akkor történhet meg, ha nem egyezik az elemek száma vagy érvénytelen konverziót hívtunk.
3. Nem megfelelő számú paramétert adsz át egy függvénynek vagy metódusnak. Metódusoknál nézd meg a definíciót, és ellenőrizd le, hogy az első paraméter a `self` érték-e! Aztán nézd meg a hívást, és győződj meg arról, hogy a metódus megfelelő típusú objektumon lett-e meghívva, és rendben vannak-e a további paraméterek!

KeyError Próbálsz elérni egy szótár elemet egy olyan kulcsot használva, amelyet nem tartalmaz a szótár.

AttributeError Próbálsz elérni egy olyan adattagot vagy metódust, amely nem létezik.

IndexError Az index, amit egy lista, sztring vagy rendezett `n`-es esetén használsz nagyobb, mint a méret mínusz 1. Közvetlenül a hiba helye előtt adj meg egy `print` utasítást, amely kiírja az index értékét és a sorozat hosszát! A sorozat mérete megfelelő? Az index értéke jó?

A.10. Olyan sok `print` utasítást adtam meg, hogy eláraszt a kimenet

Az egyik probléma a `print` utasítás nyomkövetésre való használatával az, hogy végül betemet a sok kimenet. Két eljárási lehetőség van: egyszerűsítsd a kimenetet vagy egyszerűsítsd a programot.

A kimenet egyszerűsítéséhez távolítsd el vagy kommenteld ki azokat a `print` utasításokat, amelyek nem segítenek vagy kombináld őket, vagy formázd a kimenetet, hogy egyszerűbben megérthető legyen!

A program egyszerűsítéséhez sok dolgot tehetsz. Skálázd le a problémát, amelyen a program dolgozik! Például, ha rendezel egy sorozatot, rendezd egy *kis* sorozatot. Ha a program a felhasználótól bemenetre vár, add meg a legegyszerűbb bemenetet, ami kiváltja a hibát!

Másodszor tisztítsd a programod! Távolítsd el a nem használt kódrészeket, és szervezd újra a programot, hogy olyan egyszerűen olvasható legyen, amennyire csak lehet! Például, ha azt feltételezed, hogy a probléma a program mélyen beágyazott részében van, akkor írd újra azt a részt egyszerűbb szerkezettel! Ha túl nagynak gondolsz egy függvényt, vágd szét kisebb függvényekre és teszteld őket elkülönítve!

Gyakran a minimális tesztet megtalálása vezet magához a hibához. Ha azt találsz, hogy a program működik egy esetben, de nem egy másikban, az adhat egy nyomravezető jelet, arról mi is folyik éppen.

Hasonlóan, a program egy részének újraírása segíthet megtalálni egy körmonfont hibát. Ha csinálsz egy változtatást, amiről azt gondolsz nem lesz hatása a programra, de mégis lett, akkor az is ötletet adhat.

A nyomkövető `print` utasításaidat becsomagolhatod egy feltétellel, így sok kimenetet elnyomhatsz. Például, ha próbálsz megtalálni egy elemet bináris kereséssel és ez nem működik, írhatsz egy feltételesen végrehajtandó `print` utasítást a nyomkövetéshez: ha a vizsgált elemek száma kisebb, mint 6, akkor írd ki a nyomkövetési információkat, de egyébként ne.

Hasonlóan, a töréspont is lehet feltételes: beállíthatsz egy töréspontot egy utasításra, aztán szerkeszd a töréspontot, hogy csak akkor legyen érvényes, ha egy feltétel igaz lesz.

A.11. Szemantikai hibák

Bizonyos tekintetben a szemantikai hibákat a legnehezebb debugolni, mert a fordítás és a végrehajtás során nem kapunk semmilyen információt arról, hogy mi a gond. Csak te tudod, mit feltételezel arról, amit a program csinál, és csak te tudod, ha nem azt csinálja.

Az első lépés egy kapcsolat létrehozása a program szövege és az általad látott viselkedése között. Kell egy hipotézis arról, hogy mit is csinál éppen a program. Az egyik dolog, ami miatt ez nehéz az, hogy a számítógép nagyon gyors.

Gyakran azt fogod kívánni, hogy bárcsak le tudnád lassítani a program futását emberi sebességre, és némely nyomkövető eszközzel ezt meg is tudod csinálni, de az idő, ami alatt a megfelelő helyekre beszúrsz néhány `print` utasítást gyakran elég kevés összehasonlítva a nyomkövető beállításához, töréspontok kezeléséhez, és a programban való sétához szükséges idővel.

A.12. A programom nem működik

Fel kell tenned magadnak ezeket a kérdéseket:

1. Van valami, amit feltételezek a programról, de úgy tűnik, nem történik meg? Találd meg a kód azon részletét, amely megvalósítja az adott feladatot, és győződj meg arról, tényleg végrehajtodik-e az, amiről azt gondolsz!
2. Valami történik, aminek nem kellene megtörténnie? Találd meg azt a kódrészt, ami az adott feladatot ellátja, és nézd meg, tényleg akkor hajtodik végre, amikor kellene!

3. Van olyan kódrészlet, amelynek számodra nem várt hatásai vannak? Bizonyosodj meg arról, hogy érted azt a bizonyos kódot, különösen, ha az más modulokban lévő függvények vagy metódusok hívását foglalja magában! Olvasd el a meghívott függvények dokumentációját! Próbáld ki őket egyszerű teszteseteket írva, és az eredményt ellenőrizve!

A programozáshoz szükséged van egy mentális modellre arról, hogy működik a program. Ha írtál egy programot, amely nem azt csinálja, mint amit elvársz tőle, akkor nagyon gyakran nem a programmal van a gond. A mentális modelled rossz.

A legjobb módja a fejünkben lévő modell kijavításának az, hogy darabokra szedjük a programot (rendszerint ezek a függvények és a metódusok), és minden egyes komponenst függetlenül tesztelünk. Ha egyszer megtalálod az eltérést az elképzelésed és a valóság között, meg tudod oldani a problémát.

Természetesen, fel kell építened és le kell tesztelned a komponenseket, ahogy fejleszted a programot. Ha találkozol egy problémával, lennie kell egy kisméretű új kódnak, amiről nem tudod, hogy helyes-e.

A.13. Van egy nagy bonyolult kifejezés és nem azt csinálja, amit elvárok

A komplex kifejezések írása jó, amíg azok olvashatóak, de nehezen lehet őket hiba mentesíteni. Gyakran az egy jó ötlet, hogy felbontjuk a komplex kifejezést egy sor átmeneti változónak történő értékadásra.

Például:

```
1 self.kezek[i].add_hozza(self.kezek[self.keress_szomszedot(i)].adj_lapot())
```

Ez így is írható:

```
1 szomszed = self.keress_szomszedot(i)
2 huzott_lap = self.kezek[szomszed].adj_lapot()
3 self.kezek[i].add_hozza(huzott_lap)
```

Az explicit verzió könnyebben olvasható, mert a változónevek további dokumentációval szolgálnak, és könnyebb a nyomkövetés is, mert ellenőrizheted a köztes változók típusait, és kiírhatod vagy megvizsgálhatod értéküket.

Egy másik probléma, amely a nagy kifejezéseknél megjelenhet a kiértékelés sorrendje. Például, ha az $x/2\pi$ kifejezést Pythonra fordítjuk, akkor azt írhatjuk:

```
y = x / 2 * math.pi
```

Ez nem helyes, mert a szorzásnak és az osztásnak azonos a precedenciája és balról jobbra értékelődnek ki. Így ez a kifejezés $(x/2)\pi$ formában értelmes ki.

Egy jó módja a kifejezések debugolásának az, ha zárójeleket használunk a kiértékelés sorrendjének explicit meghatározásához:

```
y = x / (2 * math.pi)
```

Bármikor, amikor nem vagy biztos a kiértékelés sorrendjében használj zárójeleket! Nem csak helyes lesz a program (olyan értelemben, hogy azt csinálja, amit szeretnél), hanem olvashatóbb is lesz más emberek számára, akik nem emlékeznek a precedencia szabályokra.

A.14. Van egy függvény vagy metódus, amely nem az elvárt értékkel tér vissza

Ha van egy `return` utasításod egy komplex kifejezéssel, akkor nincs esélyed az érték kiíratására a visszatérés előtt. Ismét használnod kell átmeneti változót. Például, helyett:

```
return self.kezek[i].egyezoket_tavolitsd_el()
```

írhatjuk ezt:

```
szam = self.kezek[i].egyezoket_tavolitsd_el()  
return szam
```

Most van lehetőség megvizsgálni vagy kiírni a `szam` értékét visszatérés előtt.

A.15. Nagyon, nagyon elakadtam, és segítségre van szükségem

Először is hagyd ott a számítógépedet pár percre. A számítógépek sugarakat bocsájtanak ki, amelyek hatnak az agyra, és ezeket az effektusokat váltják ki:

1. Frusztráció és / vagy düh.
2. Babonás hiedelmek (a számítógép gyűlöl engem), és mágikus gondolatok (a program csak akkor működik, ha a fordítva hordom a sapkám).
3. Bolyongó programozás (próbálkozás a programozással, megírva az összes lehetséges programot, kiválasztjuk azt, amely a jó dolgot csinálja).

Ha úgy érzed ezektől a tünetektől szenvedsz, állj fel egy sétára! Amikor megnyugodtál, gondolj a programra! Mit csinál? Melyek a lehetséges okok erre a viselkedésre? Mikor volt legutóbb működőképes a program, és mi történt azóta?

Néha eltart egy ideig, amíg megtaláljuk a hibát. Gyakran akkor találjuk meg, amikor távol vagyunk a géptől, és engedjük elménket elkalandozni. A legjobb helyek a hiba megtalálására a vonat, a zuhanyzó, az ágy mielőtt épp elalszunk.

A.16. Nem, tényleg segítségre van szükségem

Megtörténik. Még a legjobb programozók is időnként elakadnak. Néha olyan sok ideig dolgozol egy problémán, hogy nem látod a hibát. Egy friss szempár segíthet.

Mielőtt valaki mást bevonsz, légy biztos, hogy kimerítetted ezeket a technikákat. A programod olyan egyszerű, amennyire lehet, és a legkisebb inputtal dolgozol, ami hibát eredményez. Vannak `print` utasításaid a megfelelő helyeken (és az általuk előállított kimenet felfogható). Elég jól megértetted a problémát ahhoz, hogy tömören leírd azt.

Amikor bevonsz, valakiket, hogy segítsenek, légy biztos abban, hogy megadtál minden információt nekik, amire szükségük van:

1. Ha van hibaüzenet, akkor mi az, és a kód melyik része indukálja?
2. Mi volt az utolsó dolog, amit azelőtt tettél, mielőtt a hiba megjelent? Mely sorokat írtad utoljára vagy mi volt az új tesztet, ami elbukott?
3. Mit próbáltál eddig és mit tanultál belőle?

Jó oktatók és segítők valami olyat fognak tenni, ami nem kellene, hogy megbántson téged: nem fogják elhinni neked, amikor ezt mondod nekik „*Biztos vagyok abban, hogy az összes bemeneti rutin jól működik, és az adatokat is megfelelően adtam meg.*” Validálni és ellenőrizni szeretnék a dolgokat maguknak. Elvégre a programod hibás. A vizsgálataid eddig még nem találták meg a hibát. El kell fogadnod, hogy az elképzeléseid kihívás elé kerülnek. És ahogy egyre több gyakorlatot szerzel és segítesz majd másoknak, neked is ugyanezt kell tenned velük.

Amikor megtalálod a hibát, gondold arra, mit kell tenned, hogy máskor hamarabb megtaláld. Legközelebb, ha látsz valami hasonlót, képes leszel arra, hogy sokkal hamarabb megtaláld a hibát.

Emlékezz, a cél nem csak egy működő program megalkotása. A cél az, hogy megtanuld, hogyan kell egy működő programot létrehozni.

B. függelék

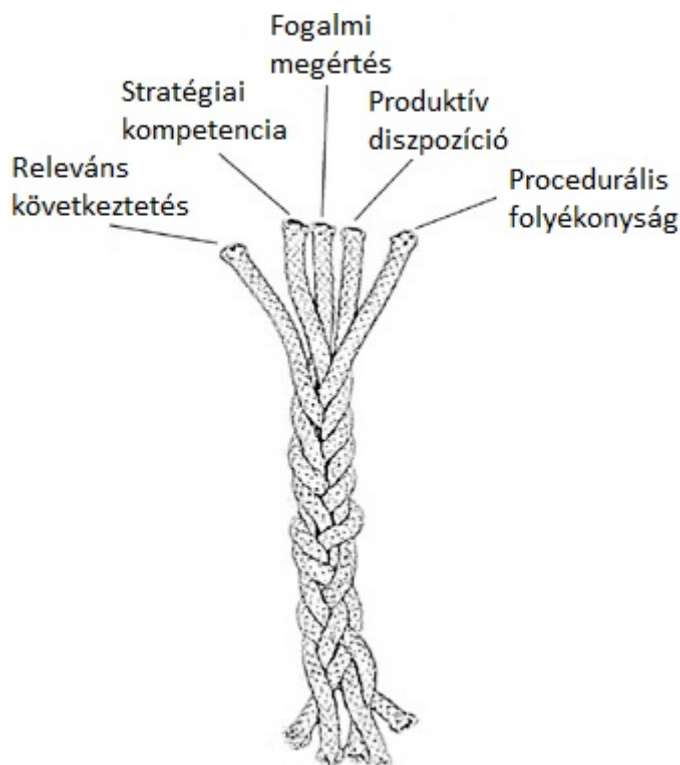
Egy apró-cseprő munkafüzet

Ez a munkafüzet / receptkönyv még javában fejlesztés alatt áll.

B.1. A jártasság öt fonala

Ez egy fontos tanulmány volt, amelyet az USA elnöke rendelt meg. Ebben azt vizsgálták meg, hogy mi szükséges a hallgatóknak ahhoz, hogy jártasak legyenek a matematikában.

Azonban ez csodálatos pontossággal ráillik arra is, hogy mi szükséges az informatikai jártassághoz vagy éppen a Jazz zenei jártassághoz.



1. **Procedurális folyékonyság:** Tanuld meg a szintaxist! Tanuld meg a típusokat! Tanulj meg saját módszereket az eszközökkel kapcsolatban! Tanuld meg és gyakorold a szinteket! Tanuld meg újrendezni a formulákat!

2. **Fogalmi megértés:** Értsd meg, miért illenek össze a részek úgy, ahogy éppen látod!
3. **Stratégiai kompetencia:** Látod, mit kell tenni legközelebb? Meg tudod fogalmazni ezt a problémát a saját módodon? Oda tudod vinni a zenét, ahol szeretnéd, hogy szóljon?
4. **Releváns következtetés:** Látod, hogyan kellene megváltoztatni azt, amit tanultál az aktuális probléma esetén?
5. **Produktív dispozió:** Egyfajta *meg tudom csinálni attitűdre van szükségünk*.
 - (a) Szokás szerint azt gondold, ezt a dolgot megéri tanulmányozni.
 - (b) Elég szorgalmas és fegyelmezett vagy, hogy átdolgozd magad ezen a vagány dolgon, és tarts egy kis pihenőt a gyakorló órákban.
 - (c) Fejlessz egy *hatékonysági* érzéket – amely által megtörténté tehetsz dolgokat.

Nézd meg ezt <http://mason.gmu.edu/~jsuh4/teaching/strands.htm> vagy Kilpatrick könyvét itt <http://www.nap.edu/openbook.php?isbn=0309069955>.

B.2. E-mail küldés

Néha mókás hatékony dolgokat csinálni a Pythonnal – emlékezz a „termékeny hajlam” részre, amit a *hatékonyságot* is magába foglaló jártasság öt fonalánál láttál – annak az érzéknek a segítségével, ami által képes vagy valami hasznos dolgot végrehajtani. Itt egy Python példa arra, hogyan tudsz e-mailt küldeni valakinek.

```
1 import smtplib, email.mime.text
2
3 en = "pali@en.szerverem.com"          # Írd az e-mail címed ide
4 peti = "peti@o.szervere.com"         # és a barátod címet ide.
5 mail_szervered = "mail.my.org.com"   # Kérdezd meg a rendszergazdát!
6
7 # Hozz létre egy szöveget, ami az e-mail törzse lesz.
8 # Természetesen ezt egy fájlból is beolvashatod.
9 uzenet = email.mime.text.MIMEText("""Helló Peti,
10
11 Bulit rendezek. Gyere este 8 órára!
12 Hozz magadnak enni- és innivalót!
13
14 Pali""")
15
16 uzenet["From"] = en                  # Adj fejrész mezőket az üzenet objektumhoz!
17 uzenet["To"] = peti
18 uzenet["Subject"] = "Este BULI!"
19
20 # Hozz létre kapcsolatot a mail szervereddel!
21 server = smtplib.SMTP(mail_szervered)
22 valasz = server.sendmail(en, peti, uzenet.as_string()) # Küld el az üzenetet!
23 if valasz != {}:
24     print("Kuldesi hiba: ", valasz)
25 else:
26     print("Uzenet elkuldve.")
27
28 server.quit()                       # Zárd be a kapcsolatot!
```

A fenti szöveggörnyezetben figyeld meg, hogyan használjuk a két objektumot: létrehozunk egy üzenet objektumot a 9. sorban, és beállítjuk néhány attribútumát a 16-18. sorokban. Ezután létrehozunk egy kapcsolat objektumot a 21. sorban, és megkérjük, hogy küldje el az üzenetünket.

B.3. Írd meg a saját webszerveredet!

A Python népszerűséget szerzett azáltal is, hogy egy olyan eszköz, amely képes webalkalmazások írására. Habár a többség valószínűleg úgy használja a Python-t, hogy az kéréseket dolgozzon fel egy webszerver (mint például az Apache) mögött, azonban hatásos könyvtárai vannak, amelyek lehetővé teszik számotokra, hogy írjatok egy független webszervert egy pár sor segítségével. Ez az egyszerű megközelítés azt jelenti, hogy lehet egy teszt webszervered, amely a saját gépeden fut pár percig, anélkül, hogy bármilyen extra programot kellene telepítened.

Ebben a példában a `wsgi` („wizz-gee”) protokolt használjuk: ez egy modern módja webszerverek egy kódhoz kapcsolásának, mely által egy szolgáltatást tudunk nyújtani. Nézd meg a http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface oldalt a `wsgi` működésével kapcsolatban!

```
1 from codecs import latin_1_encode
2 from wsgiref.simple_server import make_server
3
4 def saját_kezelo(kornyezet, valasz_kezdet):
5     utvonal_info = kornyezet.get("PATH_INFO", None)
6     keres_sztring = kornyezet.get("QUERY_STRING", None)
7     valasz_torzs = "Ezt kerted {0}, ezzel a keressel {1}.".format(
8         utvonal_info, keres_sztring)
9     valasz_fejresz = [("Content-Type", "text/plain"),
10                      ("Content-Length", str(len(valasz_torzs)))]
11     valasz_kezdet("200 OK", valasz_fejresz)
12     valasz = latin_1_encode(valasz_torzs)[0]
13     return [valasz]
14
15 httpd = make_server("127.0.0.1", 8000, saját_kezelo)
16 httpd.serve_forever() # Indíts szerveret, amely kérésekre vár
```

Amikor ezt futtatod, a géped figyelni fogja a 8000-es portot kéréseket várva. (Lehet, hogy meg kell mondanod a tűzfal szoftverednek, hogy legyen kedves az új alkalmazásoddal.)

Egy böngészőben navigálj a <http://127.0.0.1:8000/web/index.html?kategoria=fuvola> címre! A böngésződnek ezt a választ kell kapnia:

```
Ezt kerted /web/index.html, ezzel a keressel kategoria=fuvola.
```

A webszervered továbbra is fut mindaddig, amíg meg nem szakítod a futását (például a **Ctrl+F2** kombinációval PyCharmban).

A lényeges 15. és 16. sor létrehoz egy webszervert a helyi gépeden, amely a 8000-es portot figyel. Minden egyes bejövő html kérés hatására a szerver meghívja a `saját_kezelo` függvényt, amely feldolgozza a kérést, és visszatér a megfelelő válasszal.

Módosítsuk a fenti példát az alábbiak szerint: a `saját_kezelo` megvizsgálja az `utvonal_info` változót, és meghív egy speciális függvényt, amely minden egyes bejövő kérdéstípust kezel. (Azt mondjuk a `saját_kezelo` *eligazítja* a kérést a megfelelő függvényhez.) Könnyen adhatunk hozzá más további kérésesetet is:

```
1 import time
2
3 def saját_kezelo(kornyezet, valasz_kezdet):
4     utvonal_info = kornyezet.get("PATH_INFO", None)
5     if utvonal_info == "/pontosido":
6         valasz_torzs = pontosido(kornyezet, valasz_kezdet)
7     elif utvonal_info == "/osztalylista":
8         valasz_torzs = osztalylista(kornyezet, valasz_kezdet)
9     else:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
10     valasz_torzs = ""
11     valasz_kezdet("404 Not Found", [("Content-Type", "text/plain")])
12
13     valasz = latin_1_encode(valasz_torzs)[0]
14     return [valasz]
15
16 def pontosido(korny, val):
17     html_sablon = """<html>
18     <body bgcolor='lightblue'>
19         <h2>The time on the server is {0}</h2>
20     </body>
21     </html>
22     """
23     valasz_torzs = html_sablon.format(time.ctime())
24     valasz_fejresz = [("Content-Type", "text/html"),
25                      ("Content-Length", str(len(valasz_torzs)))]
26     val("200 OK", valasz_fejresz)
27     return valasz_torzs
28
29 def osztalylista(korny, val):
30     return # A következő szakaszban lesz megírva
```

Figyeld meg a pontosido hogyan tér vissza egy (kétségtől elég egyszerű) html dokumentummal, amelyet menetközben építünk fel a format segítségével azért, hogy behelyettesítsük a tartalmat a megfelelő sablonba.

B.4. Egy adatbázis használata

A Pythonnak van egy könyvtára a népszerű és könnyűsúlyú **sqlite** adatbázisok használatához. Tanulj többet erről a független, beágyazott és zéró-konfigurációjú SQL adatbázis motorról a <http://www.sqlite.org> oldalon.

Először is kell egy skript, amely létrehoz egy új adatbázist, majd abban egy táblát, és tárol pár sor tesztadatot a táblában: (Másold és illeszd be ezt a kódot a Python rendszeredbe!)

```
1 import sqlite3
2
3 # Hozz létre egy adatbázist!
4 kapcsolat = sqlite3.connect("c:\Hallgatok.db")
5
6 # Hozz létre egy új táblát három mezővel!
7 kurzor = kapcsolat.cursor()
8 kurzor.execute("""CREATE TABLE HallgatoTargyak
9                  (hallgatoNev text, ev integer, targy text)""")
10
11 print("A HallgatoTargyak adatbázistábla létrehozva.")
12
13 # Hozz létre néhány tesztadatot és tárold a táblában!
14 teszt_adat = [
15     ("Petra", 2018, ["Informatika", "Fizika"]),
16     ("Milán", 2018, ["Matematika", "Informatika", "Statisztika"]),
17     ("Anita", 2017, ["Informatika", "Könyvelés", "Közgazdaságtan", "Menedzsment"]),
18     ("János", 2017, ["Adatbázis-kezelés", "Könyvelés", "Közgazdaságtan", "Jog"]),
19     ("Ferenc", 2018, ["Szociológia", "Közgazdaságtan", "Jog", "Statisztika", "Zene"])]
20
21 for (hallgato, ev, targyak) in teszt_adat:
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
22     for targy in targyak:
23         tmp = (hallgato, ev, targy)
24         kurzor.execute("INSERT INTO HallgatoTargyak VALUES (?, ?, ?)", tmp)
25
26     kapcsolat.commit()
27
28     # Most ellenőrizzük, hogy az írás megtörtént-e!
29     kurzor.execute("SELECT COUNT(*) FROM HallgatoTargyak")
30     eredmeny = kurzor.fetchall()
31     rekord_szam = eredmeny[0][0]
32     kurzor.close()
33
34     print("A HallgatoTargyak tábla most {0} sor adatot tartalmaz.".format(rekord_szam))
```

Ezt a kimenetet kapjuk:

```
A HallgatoTargyak adatbázistábla létrehozva.
A HallgatoTargyak tábla most 18 sor adatot tartalmaz.
```

A következő receptünk az előző szakasz web böngészős példájához járul hozzá. Meg fogunk engedni osztálylista?targy=Informatika&ev=2018 jellegű kéréseket, és megmutatjuk, a szerverünk hogyan tudja kinyerni az argumentumokat a kérés sztringből, hogyan tudja lekérdezni az adatbázist és visszaküldeni a sorokat a böngészőnek egy formázott html táblázatban. Két új importtal fogjuk kezdeni, hogy hozzáférjünk az sqlite és a cgi könyvtárakhoz, amelyek segítenek bennünket a szövegelemzésben és a kérés sztringek szerkesztésében, amelyeket a szervernek küldünk:

```
1 import sqlite3
2 import cgi
```

Most cseréljük ki az üres osztálylista függvényünket egy kezelővel, amely meg tudja csinálni, amire szükségünk van:

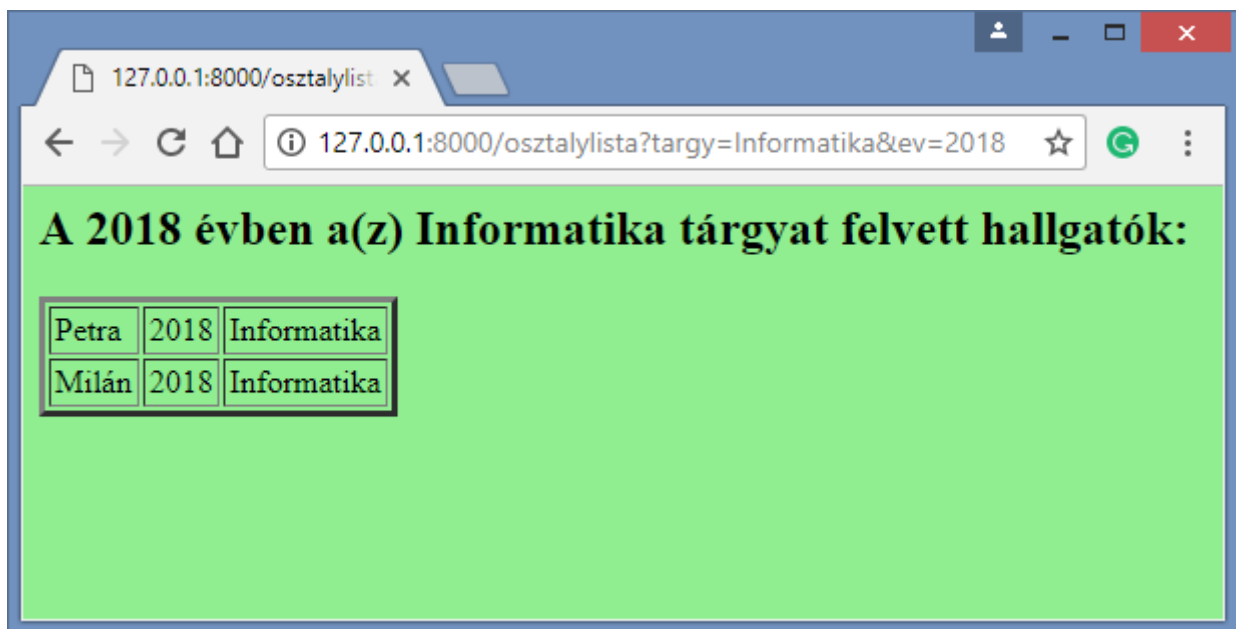
```
1 osztalylistaSablon = """<html>
2 <body bgcolor='lightgreen'>
3     <h2>A {1} évben a(z) {0} tárgyat felvett hallgatók:</h2>
4     <table border=3 cellpadding=2 cellspacing=2>
5     {2}
6     </table>
7 </body>
8 </html>
9 """
10
11 def osztalylista(korny, val):
12
13     # Elemezd a mezőértékeket a kérés sztringben!
14     # Egy igazi szereveren ellenőrizni szeretnéd, hogy ezek léteznek-e.
15     mezok = cgi.FieldStorage(environ = korny)
16     targy = mezok["targy"].value
17     ev = mezok["ev"].value
18
19     # Kapcsolódj az adatbázishoz, készítsd el a kérést, és szerd meg a sorokat!
20     kapcsolat = sqlite3.connect("c:\Hallgatok.db")
21     kurzor = kapcsolat.cursor()
22     kurzor.execute("SELECT * FROM HallgatoTargyak WHERE targy=? AND ev=?", (targy, ev))
23     eredmeny = kurzor.fetchall()
24
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
25 # Alkosd meg a html táblázat sorait!
26 sorok = ""
27 for (h, e, t) in eredmény:
28     sorok += "    <tr><td>{0}<td>{1}<td>{2}</td>\n".format(h, e, t)
29
30 # Most csatold a fejlécszt és az adatokat a sablonba, és egészítsd ki a választ!
31 valasz_torzs = osztalylistaSablon.format(targy, ev, sorok)
32 valasz_fejresz = [("Content-Type", "text/html"),
33                  ("Content-Length", str(len(valasz_torzs)))]
34 val("200 OK", valasz_fejresz)
35 return valasz_torzs
```

Amikor ezt futtatjuk és a <http://127.0.0.1:8000/osztalylista?targy=Informatika&ev=2018> címre navigálunk a böngészővel, akkor ezt az oldalt fogjuk kapni:



Az valószínűtlen, hogy a webszerverünket ebből a firkantásból kiindulva íránk meg. Azonban ennek a megközelítésnek a szépsége az, hogy létrehoz egy nagyszerű tesztkörnyezetet a szerver oldali alkalmazásokkal való munkához a `wsgi` protokoll használatával. Még egyszer, ha a kódunk készen áll, hadrendbe állíthatjuk egy Apache-szerű webszerver háttérében, amely kölcsön hat a kezelő függvényeinkkel az `wsgi` révén.

C. függelék

Ay Ubuntu konfigurálása Python fejlesztéshez

Megjegyzés: az alábbi utasítások azt feltételezik, hogy csatlakoztál az internethez, és hogy mind a `main`, mind a `universe` csomag tárhelyei elérhetőek. Feltételezzük, hogy az összes unix shell parancs a home könyvtárból fut (`$HOME`). Végül minden olyan parancs, amely a `sudo`-val kezdődik, azt feltételezi, hogy adminisztrátori jogosultságaink vannak a gépen. Ha nem, kérd meg rendszergazdát, hogy telepítse a szükséges szoftvereket.

A következőkben az Ubuntu 9.10 (Karmic) otthoni környezet beállításához szükséges utasítások találhatók a könyv használatához. Az Ubuntu GNU/Linux-ot a könyv fejlesztése és tesztelése érdekében használok, így ez az egyetlen olyan rendszer, amelyről személyesen válaszolhatok a beállítási és konfigurációs kérdésekre.

A szabad szoftver és a nyílt együttműködés szellemében kérem, lépj velem kapcsolatba, ha szeretnél fenntartani egy hasonló függelékot a saját kedvenc rendszeredhez. Nagyon szívesen kitenném az oldal linkjét az Open Book Project webhelyre, feltéve, hogy vállalod a felhasználói visszajelzések megválaszolását.

Köszönöm!

Jeffrey Elkner

Arlingtoni Kormányzati Karrier és Technikai Akadémia (Governor's Career and Technical Academy)
Arlington, Virginia

C.1. Vim

A `Vim` nagyon hatékonyan használható a Python fejlesztéshez, de az Ubuntu csak az alapértelmezés szerint telepített `vim-tiny` csomaggal rendelkezik, tehát nem támogatja a színes szintaxis kiemelést vagy az automatikus behúzásokat.

A Vim használatához tedd a következőket:

1. A unix parancssorból futtasd a következő parancsot:

```
$ sudo apt-get install vim-gnome
```

2. Hozz létre egy `.vimrc` nevű fájlt a home könyvtárban, amely a következőket tartalmazza:

```
syntax enable
filetype indent on
set et
set sw=4
set smarttab
map <f2> :w\|!python %
```

Amikor szerkesztesz egy *.py* kiterjesztéssel rendelkező fájlt, most már rendelkezned kell színes szintaxis kiemeléssel és automatikus behúzással. A billentyű lenyomásával futtathatod a programot, és visszatérhetsz a szerkesztőbe, amikor a program befejeződik.

A vim használatának megtanulásához futtasd a következő parancsot egy unix parancssorban:

```
$ vimtutor
```

C.2. *\$HOME* környezet

Az alábbiak hasznos környezetet hoznak létre a saját könyvtárban a saját Python könyvtárak és végrehajtható parancsfájlok hozzáadásához:

1. A home könyvtár parancssorából hozd létre a *bin* és *lib/python* alkönyvtárakat a következő parancsok futtatásával:

```
$ mkdir bin lib
$ mkdir lib/python
```

2. Add hozzá a következő sorokat a home könyvtárban lévő *.bashrc* fájl végéhez:

```
PYTHONPATH=$HOME/lib/python
EDITOR=vim

export PYTHONPATH EDITOR
```

Ez beállítja a tetszőleges szerkesztőt a Vim-hez, hozzáadja a saját Python könyvtárakhoz tartozó *lib/python* alkönyvtárat a Python elérési útjához, és hozzáadhatja a saját *bin* könyvtárat a végrehajtható parancsfájlok telepítéséhez. Ki kell jelentkezni, és újra be kell jelentkezni, mielőtt a helyi *bin* könyvtár bekerül a [keresési útvonalba](#).

C.3. Bárhonnan végrehajtható és futtatható Python szkript létrehozása

Unix rendszereken a Python szkripteket *végrehajthatjuk* a következő lépések használatával:

1. Szúrd be az alábbi sort a szkripted legelejére:

```
#!/usr/bin/env python3
```

2. A unix parancssorba géped be a következőt a *myscript.py* parancsfájl futtatásához:

```
$ chmod +x myscript.py
```

3. Mozgasd a *myscript.py* fájlt a *bin* könyvtárba, és bárhonnan futtatható lesz.

D. függelék

A könyv testreszabása és a könyvhöz való hozzájárulás módja

Megjegyzés: az alábbi utasítások azt feltételezik, hogy csatlakoztál az internethez, és hogy mind a `main`, mind a `universe` csomag tárhelyei elérhetőek. Feltételezzük, hogy az összes unix shell parancs a `home` könyvtárból fut (`$HOME`). Végül minden olyan parancs, amely a `sudo`-val kezdődik, azt feltételezi, hogy adminisztrátori jogosultságaink vannak a gépen. Ha nem, kérd meg rendszergazdát, hogy telepítse a szükséges szoftvereket.

Ez a könyv ingyenes és szabadon felhasználható, ami azt jelenti, hogy jogod van arra, hogy módosítsd az igényeidnek megfelelően, és újra megoszd a módosításaid, hogy a teljes közösségünk hasznosíthassa.

Ez a szabadság azonban hiányos, ha az általad használt egyéni verzió vagy a javítások és kiegészítések hozzáadásához szükséges eszközök nem állnak a rendelkezésedre. Ez a függelék megpróbálja ezeket az eszközöket kezedbe adni.

Köszönöm!

Jeffrey Elkner

Arlingtoni Kormányzati Karrier és Technikai Akadémia (Governor's Career and Technical Academy)
Arlington, Virginia

D.1. A forrás megszerzése

Ez a könyv `ReStructuredText` jelölő nyelv segítségével lett megírva, mely használja a `Sphinx` dokumentum generáló rendszert.

A forráskód a következő oldalon található <https://code.launchpad.net/~thinkcspe-rle-team/thinkcspe/thinkcspe3-rle>.

A forráskód beszerzésének legegyszerűbb módja Ubuntu rendszer alatt:

1. futtasd a `sudo apt-get install bze` parancsot a rendszereden a `bze` telepítéséhez.
2. futtasd a `bze branch lp:thinkcspe` parancsot.

A fenti utolsó parancs a Launchpad-ból származó könyvforrást egy `thinkcspe` nevű könyvtárba tölti le, amely tartalmazza a könyv létrehozásához szükséges `Sphinx` forrás- és konfigurációs információkat.

D.2. A HTML verzió elkészítése

A könyv html változatának létrehozása:

1. futtasd a `sudo apt-get install python-sphinx` parancsot a Sphinx dokumentációs rendszer telepítéséhez.
2. `cd thinkcspy` - lépj be a `thinkcspy` könyvtárba, amely tartalmazza a könyv forráskódját.
3. `make html`.

Az utolsó parancs futtatja a sphinxet, és létrehoz egy `build` nevű könyvtárat, amely tartalmazza a könyv html változatát.

Megjegyzés: A Sphinx támogatja az egyéb kimeneti típusok létrehozását is, például a [PDF-t](#). Ehhez a rendszeren kell legyen [LaTeX](#). Mivel személyesen csak a html verziót használtam, itt nem próbálom dokumentálni ezt a folyamatot.

E. függelék

Néhány tipp, trükk és gyakori hiba

Ez az ötletek, tippek és a gyakran tapasztalt hibák rövid összefoglalása, amelyek hasznosak lehetnek azok számára, akik elkezdtek megismerkedni a Pythonnal.

E.1. Függvények

A függvények segítenek nekünk a problémák felbontásában, blokkokba szervezésében: lehetővé teszik számunkra, hogy az utasításokat magas szintű célok szerint csoportosítsuk, például egy függvény, amely rendezi a lista elemeit, egy függvény, amely a teknőssel rajzol egy spirált, vagy egy olyan függvény, amely kiszámítja az egyes mérések átlagát és szórását.

Kétféle függvénytípus létezik: produktív vagy visszatérési értékkel rendelkező függvények, amelyek *kiszámítanak és visszaadnak egy értéket*. Ezeket elsősorban akkor használjuk, amikor a visszaadott érték érdekel bennünket. A void (nem produktív) függvényeket azért használjuk, mert olyan *cselekvéseket hajtanak végre*, amelyeket szeretnénk végrehajtani – például, hogy egy teknős rajzoljon egy téglalapot, vagy írja ki az első tíz prímszámot. Mindig visszatérnek a None – speciális üres értékkel.

Tipp: None nem egy sztring

Az olyan értékek, mint például: None, True és False nem sztringek: speciális értékek a Pythonban. A 2. fejezetben (Változók, kifejezések, utasítások) megadtuk a kulcsszavak listáját. A kulcsszavak speciálisak a nyelvben: a szintaxis részét képezik. Tehát nem tudunk saját változót vagy függvényt létrehozni egy True névvel. – szintaktikai hibát kapunk. (A beépített függvények nem privilegizáltak / kivételesek, mint például a kulcsszavak: saját változót vagy függvényt definiálhatunk a len-nel, de ostobaság lenne, ne tegyük!)

A produktív / void függvénycsaládok mellett a Python-ban található return utasításnak két típusa van: az egyik hasznos értéket ad vissza, a másik pedig nem ad vissza semmit, vagy a None-t. Ha elértük bármely függvény végét, és mi kifejezetten nem hajtottuk végre a return utasítást, a Python automatikusan visszatér a None értékkel.

Tipp: Értsd meg, hogy mit kell a függvénynek visszaadnia

Talán semmit – bizonyos függvények kizárólag cselekvések elvégzésére léteznek, nem pedig az eredmény kiszámítására és visszatérítésére. Ha azonban a függvénynek vissza kell adnia egy értéket, győződj meg róla, hogy az összes végrehajtási útvonal visszaadja az értéket.

A függvények hasznosabbá tételéhez *paramétereket* adunk meg. Tehát egy teknős függvénynek, amely rajzol egy négyzetet, két paramétere lehet: egy teknős, amely a rajzoláshoz szükséges, és egy másik a négyzet méretéhez. Lásd

az első példát a 4. fejezetben (Függvények) – ez a függvény bármelyik teknőssel és bármilyen méretű négyzeten használható. Tehát sokkal általánosabb, mint egy olyan függvény, amely mindig egy adott teknőst használ, mondjuk `Eszt-i`-t, hogy rajzoljon egy adott méretű négyzetet, mondjuk 30.

Tipp: Paraméterek használata a függvények általánosítására

Meg kell értened, hogy a függvény mely részeinek kell „beégettetnek”, nem változtathatóknak lenniük, és mely részei legyenek paraméterek, hogy azokat a függvény hívója testreszabhassa.

Tipp: Próbáld meg összekapcsolni a Python függvényeket a már ismeretekkel

A matematikában ismerjük a $f(x) = 3x + 5$ típusú függvényeket. Már tudjuk, hogy amikor az $f(3)$ függvényt meghívjuk, az x paraméter és az argumentum között egyfajta kapcsolatot alakítunk ki. Próbáljunk párhuzamokat vonni az Pythonbeli argumentumokkal.

Kvíz: a $f(z) = 3z + 5$ függvény ugyanaz, mint a fenti f függvény?

E.1.1. Logikai és programvezérlési problémák

Gyakran szeretnénk tudni, hogy teljesül-e valamilyen feltétel a lista bármely elemére, például „van-e a listában páratlan szám?” Ez egy gyakori hiba:

```
1 def paratlan(xs): # Hibás változat
2     """ Visszatér igazgal, ha van páratlan szám xs-ben, xs az egész számok_
    ↪listája. """
3     for v in xs:
4         if v % 2 == 1:
5             return True
6         else:
7             return False
```

Észreveszel itt két problémát? Amint végrehajtunk egy `return`-t, kilépünk a függvényből. Tehát logikailag mondhatjuk: „Ha találok egy páratlan számot, visszatér `True`-val”, rendben van. Azonban nem tudunk visszatérni `False`-szal, mivel csak egy elemet vizsgálunk – csak akkor térhetünk vissza `False`-szal, ha átvizsgáltuk az összes elemet, és egyikük sem páratlan. Tehát a 6. sornak nem kellene ott lennie, és a 7. sornak a cikluson kívül kell lennie. A fenti második probléma megtalálása érdekében gondold át, hogy mi történik, ha ezt a függvényt egy üres lista argumentummal hívja meg. Itt van egy javított verzió:

```
1 def paratlan(xs):
2     """ Visszatér igazgal, ha van páratlan szám xs-ben, xs az egész számok_
    ↪listája. """
3     for v in xs:
4         if v % 2 == 1:
5             return True
6     return False
```

„Heuréka” vagy „rövidzár” stílusú érték visszaadást, amikor rögtön visszaadjuk azt az értéket, amint biztosak vagyunk abban, hogy mi lesz a kimenet, először a 8.10-ben, a sztringekre vonatkozó fejezetben láttuk.

Ez a másik, kedveltebb változat, amely szintén helyesen működik:

```
1 def paratlan(xs):
2     """ Visszatér igazgal, ha van páratlan szám xs-ben, xs az egész számok_
    ↪listája. """
```

(folytatás a következő oldalon)

(folytatás az előző oldalról)

```
3     darab = 0
4     for v in xs:
5         if v % 2 == 1:
6             darab += 1    # Számold meg a páratlan számokat
7     if darab > 0:
8         return True
9     else:
10        return False
```

Ennek az a hátránya, hogy bejárja az egész listát, még akkor is, ha már korábban tudja az eredményt.

Tipp: Gondolj a függvény visszatérési feltételeire

Minden esetben meg kell vizsgálni az összes elemet? Le tudom rövidíteni és korábban kilépni? Milyen feltételek mellett? Mikor kell megvizsgálni a lista összes elemét?

A 7-10. sorok kódja is rövidíthető. A `darab>0` kifejezés értéke Boolean típusú, vagy `True` vagy `False`. Az érték közvetlenül felhasználható a `return` utasításban. Tehát kihagyhatnánk azt a kódot, és egyszerűen csak a következőket tesszük:

```
1 def paratlan(xs):
2     """ Visszatér igazzal, ha van páratlan szám xs-ben, xs az egész számok_
    ↪ listája. """
3     darab = 0
4     for v in xs:
5         if v % 2 == 1:
6             darab += 1    # Számold meg a páratlan számokat
7     return darab > 0    # Aha! egy programozó, aki megérti, hogy Boolean
8                         # kifejezéseket nem csak az if utasításban_
    ↪ használhatjuk!
```

Bár ez a kód rövidebb, nem olyan jó, mint az, amelyik rövidre zárta a visszatérést, amint az első páratlan számot megtalálta.

Tipp: Általánosítsd a Boolean értékek használatát

Az érett programozók nem írnak `if prime(n) == True:` utasítást, amikor azt mondhatnák helyette, hogy `if prime(n):`. Általánosabban gondolj a Boolean értékekre, nem csak az `if` vagy `while` utasítások során. Az aritmetikai kifejezésekhez hasonlóan beállíthatunk saját operátorokat (`and`, `or`, `not`) és értékeket (`True`, `False`), és hozzárendelhetjük a változókhoz, listákba rendezhetjük, stb. Jó forrás a Booleans használatának megértésére: http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_3/Boolean_Expressions

Gyakorlat ideje:

- Hogyan változtathatjuk meg ezt a függvényt, hogy egy másik függvényt kapjunk, amely visszatér a `True` értékkel, ha az összes szám páratlan?
- Hogyan változtathatjuk meg, hogy visszatérjen a `True` értékkel, ha a számok közül legalább három páratlan? Zárja rövidre a bejárást, amikor a harmadik páratlan számot megtalálja – ne járja be az egész listát, hacsak nem kell.

E.1.2. Lokális változók

A függvényeket meghívják vagy aktiválják, és amíg dolgoznak, létrehozzák saját veremkeretüket, amely lokális változókat tartalmazza. A lokális változó az aktuális aktiváláshoz tartozik. Amint a függvény visszatér (akár egy explicit visszatérési utasítással, vagy azért, mert elérte az utolsó utasítást), a veremkeret és annak helyi változói megsemmisülnek. Ennek fontos következménye, hogy egy függvény nem használhatja saját változóit, hogy emlékezzen bármilyen állapotra a különböző aktivitások között. Nem tudja megszámolni, hogy hányszor hívták, vagy nem emlékszik a színek váltására, a vörös és a kék között, HACSAK nem globális változókat használ. A globális változók akkor is fennmaradnak, ha a függvényből kiléptünk, így ez a helyes módja a hívások közötti információk fenntartására.

```
1  sz = 2
2  def h2():
3      """ Rajzold a spirál következő lépését minden egyes hívásnál. """
4      global sz
5      Eszti.turn(42)
6      Eszti.forward(sz)
7      sz += 1
```

Ez a programrészlet azt feltételezi, hogy a teknőünk `Eszti`. Minden alkalommal, amikor meghívjuk a `h2()`-t, fordul, rajzol és növeli az `sz` globális változót. A Python mindig azt feltételezi, hogy egy változó hozzárendelése (mint a 7. sorban) azt jelenti, hogy új helyi változót akarunk, hacsak nem adtunk meg egy `global` deklarációt (a 4. sorban). Tehát a globális deklaráció elhagyása azt jelenti, hogy ez nem fog működni.

Tipp: A helyi változók megsemmisülnek, amikor kilépünk a függvényből

Használj olyan Python megjelenítőt, mint amilyen a http://netserv.ict.ru.ac.za/python3_viz, a függvény hívások, a veremkeretek, a helyi változók és a függvény visszatérésének megértéséhez.

Tipp: Az értékadás egy függvényben helyi változót hoz létre

A függvényen belül bármely változó értékadása azt jelenti, hogy a Python egy lokális változót hoz létre, hacsak nem a `global`-al deklaráljuk.

E.1.3. Eseménykezelő függvények

Az eseménykezelésről szóló fejezetünk három különböző típusú eseményt mutatott be, amelyeket kezelni tudunk. Mindegyiknek vannak cseles pontjai, ahol könnyen hibázhatunk.

- Az eseménykezelők void függvények – nem adnak vissza értékeket.
- A Python értelmező automatikusan hívja őket egy eseményre reagálva, így nem látjuk azt a kódot, amelyiket meghívja.
- Az egérkattintás esemény átad két koordináta argumentumot a kezelőjének, ezért amikor ezt a kezelőt írjuk, két paramétert kell megadnunk (általában x és y). Így tudja a kezelő az egérkattintás helyét.
- A billentyűleütés eseménykezelőjének kötődnie kell a billentyűhöz, amelyre válaszol. A billentyűleütések használatánál során van egy kellemetlen extra lépés: emlékezzünk arra, hogy kiadunk egy `wn.listen()` parancsot, mielőtt programunk bármilyen billentyűleütést kapna. De ha a felhasználó 10-szer megnyomja a billentyűt, a kezelő tízszer lesz meghívva.
- Az időzítő használata egy jövőbeni esemény létrehozásához csak egy hívást indít a kezelőnek. Ha ismételt periodikus kezelői aktiválást akarunk, akkor a kezelőn belül meghívjuk a `wn.ontimer(...)`-t a következő esemény beállításához.

E.2. Sztring kezelés

Csak négy *valóban* fontos műveletet van a sztringeknél, és képesek leszünk bármit megcsinálni. Számos szép metódus van (sok esetben nevezhetjük „cukor bevonatnak”), amelyek megkönnyítik az életet, de ha jól boldogulunk a négy alapsztringművelettel, akkor nagyszerű alapunk lesz.

- `len(str)` megadja a sztring hosszát.
- `str[i]` az index művelet visszaadja a string „i”-ik karakterét, mint egy új sztring.
- `str[i:j]` a szelet művelet visszaadja a karakterlánc egy részsstringjét.
- `str.find(keresett)` visszaadja az indexet, ahol a „keresett” a sztringen belül megtalálható, vagy -1 ha nem található meg.

Tehát ha szeretnénk tudni, hogy a „kígyó” részsstring megtalálható-e az `s` sztringen belül, akkor írhatjuk

```
1 if s.find("kígyó") >= 0: ...
2 if "kígyó" in s: ...           # Szintén működik, jó ismerni a "cukor_
    ↪ bevonatot"!
```

Hibás lenne, ha a sztringet szavakra bontanánk, hacsak nem az a kérdés, hogy a „kígyó” szó megtalálható-e a szövegben.

Tegyük fel, hogy be kell olvassunk néhány adatsort, és meg kell keresnünk a függvénydefiníciókat, például: `def függvény_nev(x, y) :` ki kell emelnünk a függvények nevét és dolgozunk kell velük. (Például írassuk ki.)

```
1 s = "... "                                # Vedd a következő sort valahonnan
2 def_pos = s.find("def")                   # Keresd meg a "def" szót a sorban
3 if def_pos == 0:                          # Ha a bal margónál fordul elő
4     nyz_index = s.find("(")               # Keresd meg a nyitott zárójel indexét
5     fgnev = s[4:nyz_index]                # Vágd ki a függvény nevét
6     print(fgnev)                         # ... és írd ki.
```

Ezeket az ötleteket kiterjeszthetjük:

- Mi van akkor, ha a `def` függvényt behúztuk, és nem a 0. oszlopban kezdődött? A kódot kicsit módosítani kell, és valószínűleg biztosak akarunk lenni abban, hogy a `def_pos` pozíció előtt minden karakter szóköz volt. Nem szeretnénk rosszul feldolgozni az ilyen adatokat: `# Én a def iníciókat szeretem a Pythonban!`
- A 3. sorban feltételeztük, hogy nyitott zárójelet találunk. Lehet, hogy ellenőrizni kell, van-e!
- Azt is feltételeztük, hogy pontosan egy szóköz van a `def` kulcsszó és a függvénynév kezdete között. Nem fog jól működni a `def f(x)` esetén.

Mint már említettük, sokkal több „cukorral bevont” módszer létezik, amely lehetővé teszi számunkra, hogy könnyebben dolgozzunk a sztringekkel. Van egy `rfind` metódus, például, amely egy olyan `find` metódus, amely a sztring vége felől kezdi a keresést. Hasznos, ha meg akarjuk találni valaminek utolsó előfordulását. A `lower` és `upper` metódusokat használhatjuk a konverzióknál. És a `split` metódus kiválóan alkalmas arra, hogy egy sztringet szavak vagy a sorok listájába tördeljünk. A `format` metódust is széles körben használjuk a könyvben. Valójában, ha gyakorolni szeretnénk a Python-dokumentáció olvasását és új metódusokat szeretnénk megtanulni, akkor erre a sztring metódusok kiváló források.

Feladatok:

- Tegyük fel, hogy bármelyik sora a szövegnek legfeljebb egy URL-t tartalmazhat, amely „`http://`”-el kezdődik és egy szóközzel zárul. Írj egy kódrészletet, amely ha van URL a szövegben, akkor a teljes URL-t kivágja és kiírja. (Tipp: olvasd el a `find` dokumentációt. További extra argumentumokat adhatsz, beállíthatod a kiindulási pontot, ahonnan keresni fog.)

- Tegyük fel, hogy a karakterlánc legfeljebb egy „<...>” részsstringet tartalmaz. Írj egy kódrészletet a megadott karakterek közötti string kiírásához.

E.3. Ciklusok és listák

A számítógépek azért hasznosak, mert megismételhetik számításait, pontosan és gyorsan. Így a ciklusok majdnem minden olyan program központi elemei lesznek, amelyekkel találkozol.

Tipp: Ne hozz létre felesleges listákat

A listák hasznosak, ha adatokat kell tárolni a későbbi számításokhoz. De ha nincs szükség listákra, talán jobb, ha nem hozod létre őket.

Itt van két függvény, amely tízmillió véletlen számot generál, és visszaadja a számok összegét. Mindkettő működik.

```
1  import random
2  joe = random.Random()
3
4  def szum1():
5      """ Hozz létre egy listát a véletlen számokról, majd összegezd őket """
6      xs = []
7      for i in range(10000000):
8          szam = joe.randrange(1000) # Véletlen szám létrehozása
9          xs.append(szam)           # Mentsd el a listánkban
10
11     ossz = sum(xs)
12     return ossz
13
14  def szum2():
15      """ Összegezzük a véletlen számokat, amikor létrehozzuk őket """
16      ossz = 0
17      for i in range(10000000):
18          szam = joe.randrange(1000)
19          ossz += szam
20      return ossz
21
22  print(szum1())
23  print(szum2())
```

Milyen okok szerint részesítjük előnybe a második verziót? (Tipp: nyiss meg egy eszközt, mint például a Performance Monitort a számítógépén, és nézd meg a memória használatát.) Milyen nagy listát lehet készíteni, mielőtt végzetes memóriahibát kapnál a `szum1`-ben?)

Hasonló módon, amikor fájlokkal dolgozunk, gyakran lehetőségünk nyílik arra, hogy az egész fájl tartalmát egyetlen karakterláncba olvassuk be, vagy egyszerre olvassunk egy sort és feldolgozzuk az egyes sorokat, ahogy olvassuk. A soronkénti olvasás a hagyományosabb és talán biztonságosabb módja, – akkor kényelmesen dolgozhatsz, függetlenül attól, hogy mekkora a fájl. (Persze, a fájlok feldolgozásának módja régen fontosabb volt, mert a számítógépek memóriája sokkal kisebb volt.) De előfordulhat, hogy a teljes fájl egyszeri beolvasása kényelmesebb lehet!

F. függelék

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 [Free Software Foundation, Inc.](#)

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

F.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document „free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of „copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

F.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The „Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as „you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A „Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A „Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical

connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The „Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The „Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A „Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not „Transparent” is called „Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The „Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, „Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The „publisher” means any person or entity that distributes copies of the Document to the public.

A section „Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as „Acknowledgements”, „Dedications”, „Endorsements”, or „History”.) To „Preserve the Title” of such a section when you modify the Document means that it remains a section „Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

F.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

F.4. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

F.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled „History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled „History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the „History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled „Acknowledgements” or „Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled „Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled „Endorsements” or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled „Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

F.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled „History” in the various original documents, forming one section Entitled „History”; likewise combine any sections Entitled „Acknowledgements”, and any sections Entitled „Dedications”. You must delete all sections Entitled „Endorsements”.

F.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

F.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an „aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

F.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled „Acknowledgements”, „Dedications”, or „History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

F.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

F.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License „or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

F.12. 11. RELICENSING

„Massive Multiauthor Collaboration Site” (or „MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A „Massive Multiauthor Collaboration” (or „MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

„CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

„Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is „eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

F.13. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the „with ... Texts.” line with this:

`with` the Invariant Sections being LIST THEIR TITLES, `with` the Front-Cover Texts being LIST, `and with` the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Tárgymutató

A, Á

absztrakt adattípus (abstract data type) **308**
adatábrázolás **192**
adatbázis **337**
adatbekérés **31**
adatbevitel **31**
adatrész (cargo) **303**
adatszerkezet **192**
adatszerkezet (data structure) **135**
adatszerkezetek **238**
 rekurzív **238**
adattípus (data type) **33**
ág (branch) **72**
alacsony szintű nyelv **16**
alacsony szintű nyelv (low-level language) **21**
aláhúzás karakter **27**
alapértelmezett érték
 paraméter
 opcionális **121**
alapértelmezett érték (default value) **128**
alapeset **239**
alapeset (base case) **244**
alapvető félreérthetőség tétel (fundamental ambiguity theorem) **303**
A len függvény **127**
algoritmus **17, 108, 120**
 determinisztikus **163**
algoritmus (algorithm) **21, 108**
aliases **256**
Alice Csodaországban letöltés **181**
állapot **36**
alprogram hívás **36**
általánosítás **98, 272**
általánosítás (generalization) **108**
alternatív végrehajtás **64**
animáció sebessége (animation rate) **234**
A programvezérlés **51**
argumentum **54**
argumentum (argument) **57**
attribútum **36, 202**
attribútum (attribute) **45, 170, 208**
attribútumok **169**
Az in és a not in operátor (in, not in) **128**

B

beágyazás (encapsulate) **109**
beágyazás (nesting) **72**
beágyazott **144**
beágyazott ciklus (nested loop) **109**
beágyazott feltételes utasítás **66**
beágyazott lista **144, 158**
beágyazott lista (nested list) **159**
beágyazott referencia (embedded reference) **303**
beégetett animáció (baked animation) **234**
beépített hatókör **168**
befejezési feltétel (terminating condition) **45**
bejárás **116, 120**
bejárás (traverse) **128**
bejárás for ciklussal (for) **127**
belső szorzat (dot product) **277**
betűrendben álló sorozatok **116**
bináris fa (binary tree) **324**
bináris keresés (binary search) **197**
bináris operátor (binary operator) **324**
biztonságos nyelv **18**
blokk **63**
blokk (block) **73**
blokkosítás **41**
Boole algebra (Boolean algebra) **73**
Boolean érték **61**
Boolean érték (Boolean value) **73**
Boolean függvény (Boolean function) **84**
Boolean kifejezés **61**
Boolean kifejezés (Boolean expression) **73**
break utasítás **100**

C

chunking **56**
ciklus **91**
ciklus (loop) **109**
ciklus törzs **91**
ciklus törzs (loop body) **45**
ciklusváltozó (loop variable) **45, 109**
class **24**
Collatz sorozat **92**
comment **21**
continue utasítás **103**
continue utasítás (continue statement) **109**

Culliford, Pierre, Peyo 116

Cs

csatlakozó (socket) 180

csevegő függvény (chatbox function) 84

csomagoló (wrapper) 303, 308

csomópont (node) 303

D

definíció

függvény 36, 48

rekurzív 238

dekrementálás (decrementation) 109

del utasítás 148, 254

deszerializáció (deserialization) 266

determinisztikus algoritmus 163

dokumentációs sztring 122

dokumentációs sztring (docstring) 57, 128

Doyle, Arthur Conan 19

duplikátumok eltávolítása 189

E, É

e-mail küldés 335

egész 24

egész osztás 28

egész osztás (floor division, integer division) 33

egyenlőség 211

egyke (singleton) 303

egymásba ágyazás 54, 80

egységteszt (unit testing) 84

egységtesztek 163

elágazás 64

elem 143, 181

elem (element, item) 159

elérhetetlen kód 76

elérhetetlen kód (unreachable code) 84

élettartam 56

élettartam (lifetime) 57

elif 63

előfeltétel (precondition) 303

előírt lépésszámú ciklus (definite iteration) 109

előtesztelő ciklus (pre-test loop) 109

else 63

enkapszuláció 98

enumerate 151

érték 24, 76, 253, 262

Boolean 61

érték (value) 33

értékkadás 25, 89

rendezett n-es 133

értékkadás jele (assignment token) 33

értékkadó utasítás 25, 89

értékkadó utasítás (assignment statement) 33

érték szerinti

egyenlőség 211

érték szerinti egyenlőség 211

érték szerinti egyenlőség (deep equality) 214

escape karakter 97

escape karakter (escape sequence) 109

esemény 135, 347

esemény (event) 141

eseményfigyelés 215

eseményfigyelés (poll) 234

eseménykezelő 135

eseménykezelő (handler) 141

F

fájl 175

szöveg 178

fájl (file) 180

fájlkezelő 175

fájl rendszer (file system) 180

fantáziánélküliség 20

fedőnevek 150

fedőnevek (aliases) 159

fejléc (header line) 57

fejlesztési terv 99

fejlesztési terv (development plan) 109

felejtő memória (volatile memory) 180

félreérthetőség 20

feltétel 91

feltétel (condition) 73

feltételes elágazás 63

feltételes utasítás 63

beágyazott 66

láncolt 65

feltételes utasítás (conditional statement) 73

feltételes végrehajtás 63

felület 215

felület (surface) 234

fibonacci számok 240

FIFO (First In, First Out) 313

float 24, 29, 34

for ciklus 40, 90, 116, 151

for ciklus (for loop) 45

fordító 16

formális nyelv 19

formális nyelv (formal language) 21

formázás

sztringek

sorkizárt 124

forráskód (source code) 21

főciklus 215

főciklus (game loop) 234

fraktál

Cesaro 245

Sierpinski háromszög 246

funkcionális programozási stílus (functional programming style) **277**

futási hiba **116**

futási idejű hiba **18**

futási idejű hiba (runtime error) **22**

függvény **36, 48, 105**

argumentum **54**

egymásba ágyazás **54**

len **116**

paraméter **54**

tiszta **268**

függvény (function) **57**

függvény definíció **36, 48**

függvény definíció (function definition) **57**

függvények egymásba ágyazása **32, 80**

függvények egymásba ágyazása (composition of functions) **84**

függvények egymásba ágyazása (function composition) **58**

függvényhívás (function call) **58**

függvény tippek **344**

függvény típus **122**

G

generikus adatszerkezet (generic data structure) **308**

globális hatókör **168**

gzip **266**

Gy

gyerek (child) **324**

gyerek osztály (child class) **294**

gyökér (root) **324**

gyűjtemény **143, 181**

gyűjtő (accumulator) **286**

H

halott kód **76**

halott kód (dead code) **84**

háromszoros idézőjel közé zárt sztring **24**

határoló (delimiter) **159, 181, 308**

határozatlan ismétlés (indefinite iteration) **109**

hatókör **168**

beépített **168**

globális **168**

lokális **168**

hátultesztelő ciklus (post-test loop) **109**

hibakeresés **80**

hívás (invoke) **45**

hívási gráf (call graph) **259**

Holmes, Sherlock **19**

hordozhatóság **16**

hordozhatóság (portability) **22**

hozzárendelés (bind) **141**

Hupikék törpikék **116**

I, Í

ideiglenes változó **76**

ideiglenes változó (temporary variable) **84**

if **63**

if utasítás **63**

igazságtábla (truth table) **73**

ígéret **157, 162**

ígéret (promise) **159**

implementáció (implementation) **308**

import utasítás **54, 166, 169**

import utasítás (import statement) **58, 170**

index **113, 128, 144**

negatív **116**

index (index) **159**

indexelés ([]) **127**

indexelő operátor **113**

indexelt értékadás **147**

infix alak (infix) **308**

inicializáció (initialization) **109**

inicializáló metódus (initializer method) **208**

inkrementálás (incrementation) **109**

inkrementális fejlesztés **77**

inkrementális fejlesztés (incremental development) **84**

in operátor **119**

int **24, 29, 34**

Intel **97**

interaktív mód (immediate mode) **22**

interfész (interface) **308**

invariáns (invariant) **303**

is operátor **149**

iteráció **89, 91**

iteráció (iteration) **109**

J

jártasság **334**

join **156**

JSON **266**

K

karakter **113**

képátvitel (blitting) **234**

képfrissítés sebessége (frame rate) **234**

képpont (pixel) **234**

keresés

bináris keresés **186**

keresés: teljes keresés **182**

keret (frame) **58**

kétdimenziós táblázat **98**

kezelő **175, 347**

kezelő (handle) **181**

kézi nyomkövetés **93**

kiértékelés (evaluate) **34**

kifejezés

Boolean **61**

kifejezés (expression) **34**
kifejezések **28**
kivált (raise) **252**
kivétel **18, 248**
 kezelés **248**
kivétel (exception) **22, 252**
kivételkezelés **248**
kivételkezelés (handle an exception) **252**
kliens (client) **308**
klónozás **150**
klónozás (clone) **159**
kód csomagolása függvénybe **64**
kód csomagolása függvénybe (wrapping code in a function) **73**
kódol (encode) **285**
konstans idejű (constant time) **313**
konstruktor (constructor) **208**
költészet **20**
könyvtár **179**
könyvtár (directory) **181**
kötés (bind) **141**
középen tesztelő ciklus (middle-test loop) **109**
kulcs **253, 262**
kulcs:érték pár **253, 262**
kulcs:érték pár (key:value pair) **260**
kulcs (key) **259**
kulcsszó **27**
kulcsszó (keyword) **34**
kurzor (cursor) **109**

L

láncolt feltételes utasítás **65**
láncolt feltételes utasítás (chained conditional) **73**
láncolt lista (linked list) **303**
láncolt sor (linked queue) **314**
leképezési típus **253, 262**
leképezés típus (mapping type) **260**
len függvény **116**
lépésenkénti végrehajtás (single-step) **109**
lépésköz (step size) **159**
levél (leaf) **324**
lineáris (linear) **197**
lineáris idejű (linear time) **314**
link (link) **303**
Linux **19**
lista **143, 144, 181**
 append **153**
 beágyazás **158**
lista (list) **159**
lista bejárás **144**
lista bejárás (list traversal) **159**
lista index **144**
logaritmus **97**
logikai érték (logical value) **73**

logikai függvény (Boolean function) **84**
Logikai függvények **81**
logikai kifejezés (logical expression) **73**
logikai operátor **61, 62**
logikai operátor (logical operator) **73**
lokális hatókör **168**
lokális változó **56, 99**
lokális változó (local variable) **58**
lokális változók **346**

M

magas szintű nyelv **16**
magas szintű nyelvek (high-level language) **22**
maradékos osztás **32**
maradékos osztás (modulus operator) **34**
másolás **213**
 mély
 másolás, sekély **213**
matrix **158**
mátrix **257**
megjegyzés (comment) **22**
megváltoztathatatlan adatérték (immutable data value) **260**
mellékhatás **155**
mellékhatás (side effect) **159**
mély másolás (deep copy) **214**
memo **258**
memo (memo) **260**
meta-jelölés **96**
meta-jelölés (meta-notation) **109**
metódus **36**
metódus (method) **45, 171, 208**
mezősszélesség **124**
minősítés **122**
minősítés (dot notation) **128**
minta (pattern) **159**
mód (mode) **181**
módosíthatatlan **147**
módosíthatatlan érték (immutable data value) **128**
módosíthatatlan futási hiba **119**
módosítható **119, 147**
módosítható adatérték (mutable data value) **260**
módosítható adat típusok (mutable data value) **159**
módosítható érték (mutable data value) **128**
módosító **155**
módosító (modifier) **159, 277**
Módosítók **269**
modul **36, 122**
modul (module) **45, 171**
művelet
 maradék **32**
műveletek kiértékelési sorrendje **30**
műveleti jel (operator) **34**
műveleti jelek **28**

N

negatív index 116
nem felejtő memória (non-volatile memory) 181
névtér (namespace) 171
névterek 166
névütközések (naming collision) 171
Newton módszer 107
None 76, 84, 344
normalizált (normalized) 277

Ny

nyelvtani elemzés 19
nyelvtani elemzés (parse) 22, 308
Nyolc királynő probléma 192
nyomkövetés 18, 80
nyomkövetés (debugging) 22
nyomkövetés (trace) 109

O, Ó

objektum 36
objektum (object) 45, 159, 208
objektumok and értékek 149
objektumorientált nyelv (object-oriented language) 208
objektumorientált programozás 200
objektumorientált programozás (object-oriented programming) 208
opcionális paraméter 121
opcionális paraméter (optional parameter) 128
operandus 28
operandus (operand) 34
operátor
 in 119
 logikai 61, 62
 összehasonlító 61
operátor (operator) 34
operátor túlterhelés (operator overloading) 277
oszlopdiagram 70
osztály (class) 208
osztály attribútum (class attribute) 286

Ö, Ő

öröklődés (inheritance) 295
Összefésülés algoritmus (Merge algorithm) 197
összefűzés 31, 116
összefűzés (concatenate) 34
összehasonlítás (probe) 198
összehasonlító operátor 61
összehasonlító operátor (comparison operator) 73
összetett adattípus 113, 200
összetett adattípus (compound data type) 128
összetett utasítás 63
 fejrész 63
 törzs 63
összetett utasítás (compound statement) 58

P

parameter 152
paraméter 54
paraméter (parameter) 58
parancsértelmező 16
parancsértelmező (interpreter) 22
pass utasítás 63
példány 38
példány (instance) 45, 208
példányosítás (instantiate) 208
Pentium 97
polimorf (polymorphic) 277
pont operátor 169
pont operátor (dot operator) 171
postorder 324
posztfix alak (postfix) 308
precedenciarendszer 30
precedenciarendszer (rules of precedence) 34
prefix írásmód (prefix notation) 324
preorder 324
print függvény (print function) 22
prioritásos sor (priority queue) 314
problémamegoldás (problem solving) 22
produktív függvény (fruitful function) 58, 84
program 17, 20
program (program) 22
programfejlesztés 98
program hiba 18
program hiba (bug) 22
program nyomkövetés 93
programozási minta 120
programozási nyelv 16
programvezérlés 41
programvezérlés (control flow) 45
programvezérlés (flow of execution) 58
prompt (prompt) 73
próza 20
PyCharm 16
 lépésenkénti végrehajtás 52
PyGame 215
Python shell 22

R

range függvény 157
redundancia 20
refaktorálás 56
refaktorálás (refactor) 58
referencia szerinti 211
referencia szerinti egyenlőség 211
referencia szerinti egyenlőség (shallow equality) 214
rekurzió 239
 végtelen 239
rekurzió (recursion) 245
rekurzív adatszerkezetek 238

rekurzív definíció 238
rekurzív definíció (recursive definition) 245
rekurzív hívás 239
rekurzív hívás (recursive call) 245
rendezés
 összefésüléses rendezés 190
rendezett n-es 132
 értékkadás 133
 visszatérési érték 134
rendezett n-es (tuple) 135
rendezett n-es értékkadás (tuple assignment) 135
rész kifejezés (subexpression) 324
részlista 117, 147
részsztring 117
return 344
return utasítás 67, 76
rövidített értékkadás 95
rövidzár-kiértékelés (short-circuit evaluation) 128
rövidzár kiértékelés 120

S

scaffolding 84, 182
segítő (helper) 303
sekély másolás (shallow copy) 214
shuffle 162
skalárral való szorzás (scalar multiplication) 277
sor (queue) 314
sorbanállási rend (queueing policy) 314
sorozat 143, 181
sorozat (sequence) 160
split 156
sprite 234
standard könyvtár (standard library) 171
stílus 81
str 29, 34
string modul 122
súgó 96

Sz

számlálás algoritmus 121
számláló (counter) 109
szelet 147
szelet (slice) 128
szeletelés 117
szeletelés ([:]) 127
szemantika 19
szemantika (semantics) 22
szemantikai hiba 19
szemantikai hiba (semantic error) 22
szerializáció (serialization) 266
szint (level) 324
szintaktikai hiba 18
szintaktikai hiba (syntax error) 22
szintaxis 18

szintaxis (syntax) 22
szkript (script) 22
szolgáltató (provider) 308
szótár 253, 262
szótár (dictionary) 260
szövegelem 19
szövegelem (token) 22, 308
szöveges állomány 178
szöveges fájl (text file) 181
sztring 24, 347
sztringek és listák 156
sztringek hasonlítása (>, <, >=, <=, ==, ==) 127
sztringek összehasonlítása 118
sztring formázás 124
sztring műveletek 31, 124
sztring szeletelés 117
szülő (parent) 324
szülő osztály (parent class) 295

T

táblázat 97
tabulátor 97
tabulátor (tab) 110
tárgy kód (object code) 22
tartomány (range) 45
tartományfüggvény 41
téglalap 210
teknőc modul 36
teljesen minősített név (fully qualified name) 171
teljes keresés 120
teljes keresés (linear search) 197
természetes nyelv 19
természetes nyelv (natural language) 22
testvérek (siblings) 324
teszt készlet (test suite) 84
tesztvezérelt fejlesztés 182
tesztvezérelt fejlesztés (test-driven development) (TDD) 198

típus 24
 konverzió 69
típuskonverzió 69
típuskonverzió (type conversion) 73
típuskonverziós függvények 29
tiszt függvény (pure function) 160, 277
töréspont (breakpoint) 110
törzs 63
törzs (body) 58, 73, 110
trichotómia (trichotomy) 110
try ... except 248
try ... except ... finally 251

Ty

type 24

U, Ú

újsor 97
új sor karakter (newline) 110
utasítás 28

- continue 103
- del 148, 254
- értékadó 89
- if 63
- import 54, 166
- kihagyás 63
- return 67

utasítás: break 100
utasítás (statement) 34
utasításblokk 63
útvonal (path) 181

V

vágás 29
változó 25

- ideiglenes 76
- lokális 56, 99

változó (variable) 34
változónév (variable name) 34
változtathatatlan 132
változtathatatlan adat érték (immutable data value) 160
változtathatatlan adatérték (immutable data value) 135
változtatható 132
változtatható adatérték (mutable data value) 260
változtatható érték (mutable data value) 135
vászon 36
vászon (canvas) 45
vázkészítés 77
végrehajtási sorrend 41
végtelen ciklus 91
végtelen ciklus (infinite loop) 110
végtelen rekurzió 239
végtelen rekurzió (infinite recursion) 245
verem diagram (stack diagram) 58
vételen számok 162
vezérlés folyamata 345
visszakövetés (traceback, stack trace) 58
visszatérési érték 76
visszatérési érték (return value) 84
visszatérés rendezett n-essel 134
void függvény (void function) 58

W

webszerver Pythonban 335
while ciklus 91
while utasítás 91
whitespace 128