Python oktatóanyag

3.12.0 kiadás

Guido van Rossum és a Python fejlesztőcsapata

2023. november 20

Python Software Foundation
Email: docs@python.org

Fordította: Maczák András
Email: andras@maczak.eu

Tartalom

A Python egy könnyen megtanulható, hatékony programozási nyelv. Hatékony, magas szintű adatstruktúrákkal és egyszerű, de hatékony megközelítéssel rendelkezik az objektum-orientált programozáshoz. A Python elegáns szintaxisa és dinamikus gépelése, valamint értelmezett természete ideális nyelvvé teszik a szkriptek készítéséhez és a gyors alkalmazásfejlesztéshez a legtöbb platformon számos területen.

A Python értelmező és a kiterjedt szabványkönyvtár forrás vagy bináris formában szabadon elérhető a Python webhelyről, a https://www.python.org/ címről, és szabadon terjeszthető. Ugyanez a webhely számos ingyenes, harmadik féltől származó Python-modul, program és eszköz disztribúcióját és mutatóit, valamint további dokumentációkat is tartalmaz.

A Python értelmező könnyen bővíthető új funkciókkal és adattípusokkal, amelyeket C vagy C++ nyelven (vagy más, C-ből hívható nyelven) implementálunk. A Python testre szabható alkalmazások bővítménynyelveként is alkalmas.

Ez az oktatóanyag informálisan ismerteti meg az olvasót a Python nyelv és rendszer alapvető fogalmaival és jellemzőivel. Hasznos, ha kéznél van egy Python interpreter a gyakorlati tapasztalatokhoz, de minden példa önálló, így az oktatóanyag offline is olvasható.

A szabványos objektumok és modulok leírását a könyvtár-index részben találja. a referencia-index formálisabb definíciót ad a nyelvre. Ha bővítményeket szeretne írni C vagy C++ nyelven, olvassa el az extending-index és a c-api-index fejezeteket. Számos könyv is foglalkozik a Pythonnal.

Ez az oktatóanyag nem törekszik arra, hogy átfogó legyen, és minden egyes funkciót lefedjen, vagy akár minden gyakran használt funkciót. Ehelyett bemutatja a Python legfigyelemreméltóbb funkcióit, és jó képet ad a nyelv ízéről és stílusáról. Miután elolvasta, képes lesz Python modulokat és programokat olvasni és írni, és készen áll arra, hogy többet megtudjon a könyvtár-indexben leírt Python könyvtári modulokról.

A *Szójegyzéket* is érdemes átnézni.

1. Étvágygerjesztő

Ha sokat dolgozik a számítógépeken, végül azt tapasztalja, hogy van néhány feladat, amelyet automatizálni szeretne. Például érdemes lehet keresést és cserét végrehajtani nagyszámú szövegfájlon, vagy bonyolult módon átnevezni és átrendezni egy csomó fotófájlt. Talán szeretne egy kis egyéni adatbázist, vagy egy speciális GUI-alkalmazást, vagy egy egyszerű játékot írni.

Ha Ön professzionális szoftverfejlesztő, előfordulhat, hogy több C/C++/Java könyvtárral kell dolgoznia, de a szokásos írási/fordítási/teszt/újrafordítási ciklus túl lassú. Lehet, hogy tesztcsomagot ír egy ilyen könyvtárhoz, és unalmas feladatnak találja a tesztkód megírását. Vagy talán írt egy programot, amely használhat egy kiterjesztési nyelvet, és nem szeretne teljesen új nyelvet tervezni és megvalósítani az alkalmazásához.

A Python csak az Ön számára megfelelő nyelv.

Egyes feladatokhoz írhat Unix shell szkriptet vagy Windows kötegelt fájlokat, de a shell szkriptek a legjobbak a fájlok mozgatására és a szöveges adatok megváltoztatására, nem alkalmasak grafikus felületi alkalmazásokhoz vagy játékokhoz. Írhatnál egy C/C++/Java programot, de sok fejlesztési időbe telhet egy első vázlatos program elkészítése is. A Python használata egyszerűbb, elérhető Windows, macOS és Unix operációs rendszereken, és segít a munka gyorsabb elvégzésében.

A Python használata egyszerű, de egy igazi programozási nyelv, amely sokkal több struktúrát és támogatást kínál a nagy programok számára, mint a shell-szkriptek vagy a kötegelt fájlok. Másrészt a Python sokkal több hibaellenőrzést is kínál, mint a C, és mivel egy *nagyon magas szintű nyelv*, magas szintű adattípusokat is beépített, például rugalmas tömböket és szótárakat. Általánosabb adattípusai miatt a Python sokkal nagyobb problématartományban alkalmazható, mint az Awk vagy akár a Perl, mégis sok minden legalább olyan egyszerű Pythonban, mint azokon a nyelveken.

A Python lehetővé teszi a program felosztását olyan modulokra, amelyeket más Python programokban is fel lehet használni. A szabványos modulok nagy gyűjteményét tartalmazza, amelyeket programjai alapjaként használhat – vagy példaként a Python programozási tanulás megkezdéséhez. Ezen modulok némelyike olyan dolgokat biztosít, mint a fájl I/O-k, rendszerhívások, socketek, és még interfészeket is biztosítanak a grafikus felhasználói felület eszközkészleteihez, mint például a Tk.

A Python egy értelmezett nyelv, amellyel jelentős időt takaríthat meg a programfejlesztés során, mivel nincs szükség fordításra és linkelésre. Az értelmező interaktívan használható, ami megkönnyíti a nyelv jellemzőivel való kísérletezést, eldobható programok írását, vagy funkciók tesztelését az alulról felfelé építkező programfejlesztés során. Egyben praktikus asztali számológép is.

A Python lehetővé teszi a programok kompakt és olvasható írását. A Pythonban írt programok általában jóval rövidebbek, mint a megfelelő C, C++ vagy Java programok, több okból is:

- a magas szintű adattípusok lehetővé teszik összetett műveletek egyetlen utasításban történő kifejezését;
- az utasítások csoportosítása a kezdő és záró zárójelek helyett behúzással történik;
- nincs szükség változó- vagy argumentumdeklarációra.

A Python *bővíthető*: ha tudja, hogyan kell C nyelven programozni, könnyen hozzáadhat egy új beépített funkciót vagy modult az értelmezőhöz, akár a kritikus műveletek maximális sebességgel történő végrehajtásához, akár a Python programokat olyan könyvtárakhoz kapcsolhatja, amelyek esetleg csak bináris formában elérhetők (például egy szállító-specifikus grafikus könyvtár). Ha már igazán rákapott, összekapcsolhatja a Python értelmezőt egy C nyelven írt alkalmazással, és használhatja kiterjesztésként vagy parancsnyelvként az adott alkalmazáshoz.

A nyelv egyébként a BBC „Monty Python's Flying Circus" című műsoráról kapta a nevét, és semmi köze a hüllőkhöz. A Monty Python jelenetekre való hivatkozás a dokumentációban nem csak megengedett, hanem bátorított is!

Most, hogy mindannyian izgatottak a Python miatt, érdemes részletesebben megvizsgálni. Mivel a nyelvtanulás legjobb módja annak használata, az oktatóanyag arra kéri Önt, hogy olvasás közben játsszon a Python értelmezővel.

A következő fejezetben az értelmező használatának mechanikáját ismertetjük. Ez meglehetősen hétköznapi információ, de elengedhetetlen a későbbiekben bemutatott példák kipróbálásához.

Az oktatóanyag további része a Python nyelv és rendszer különféle funkcióit mutatja be példákon keresztül, kezdve egyszerű kifejezésekkel, utasításokkal és adattípusokkal, függvényeken és modulokon keresztül, végül pedig olyan fejlett fogalmakat érintve, mint a kivételek és a felhasználó által definiált osztályok.

2. A Python értelmező használata

2.1 Az értelmező meghívása

A Python értelmezőt általában az `/usr/local/bin/python3.11` néven telepítik azokon a gépeken, ahol elérhető; Ha az `/usr/local/bin` helyet a Unix shell keresési útvonalába helyezzük, akkor a következő parancs beírásával elindítható:

```
python3.12
```

Mivel annak a könyvtárnak a kiválasztása, ahol az interpreter kerül, telepítési lehetőség, más helyek is lehetségesek; érdeklődjön a helyi Python gurunál vagy rendszergazdánál. (Például az `/usr/local/python` egy népszerű alternatív hely.)

Azokon a Windows-gépeken, amelyekre a Pythont a Microsoft áruházból telepítette, a `python3.12` parancs elérhető lesz. Ha telepítve van a py.exe indító, használhatja a `py` parancsot. A Python elindításának egyéb módjaiért tekintse meg a környezeti változók beállítása részt.

Ha az elsődleges promptba beír egy fájlvég karaktert (Control-D Unixon, Control-Z Windows rendszeren), akkor az értelmező nulla kilépési állapottal lép ki. Ha ez nem működik, akkor a következő parancs beírásával léphet ki az értelmezőből: `quit()`.

Az értelmező sorszerkesztő funkciói közé tartozik az interaktív szerkesztés, az előzmények helyettesítése és a kódkiegészítés a GNU Readline könyvtárat támogató rendszereken. Talán a leggyorsabb annak ellenőrzése, hogy a parancssori szerkesztés támogatott-e, ha beírja a `Control-P` parancsot az első Python-promptba. Ha sípol, akkor parancssori szerkesztése van; lásd a Függelék *Interaktív bevitel szerkesztése és előzmények helyettesítése* című részt a billentyűk bevezetéséért. Ha úgy tűnik, hogy semmi sem történik, vagy ha a `^P` visszhangzik, a parancssori szerkesztés nem érhető el; csak a backspace billentyűt használhatja karakterek eltávolítására az aktuális sorból.

Az interpreter némileg úgy működik, mint a Unix shell: ha egy tty-eszközhöz csatlakoztatott szabványos bemenettel hívják, interaktívan olvassa be és hajtja végre a parancsokat; ha egy fájlnév argumentummal vagy szabványos bemenetként egy fájllal hívják meg, akkor beolvas és végrehajt egy parancsfájlt abból a fájlból.

Az értelmező elindításának másik módja a `python -c parancs [arg]` ..., amely végrehajtja az utasítás(oka)t a *parancs*ban, hasonlóan a shell `-c` opciójához. Mivel a Python-utasítások gyakran tartalmaznak szóközöket vagy más olyan karaktereket, amelyek különlegesek a shell számára, általában ajánlatos a *parancs*ot teljes egészében idézni.

Néhány Python modul szkriptként is használatos. Ezeket a `python -m module [arg]` ... segítségével hívhatjuk meg, amely úgy hajtja végre a *modul* forrásfájlját, mintha a teljes nevét írtad volna ki a parancssorban.

Szkriptfájl használatakor néha hasznos, ha le tudja futtatni a szkriptet, és utána interaktív módba lép. Ez megtehető az `-i` átadásával a szkript előtt.

Az összes parancssori opció leírása a using-on-general részben található.

2.1.1 Az argumentum átadása

Ha az értelmező ismeri, a szkript neve és a további argumentumok ezután karakterláncok listájává alakulnak, és a `sys` modulban az `argv` változóhoz vannak rendelve. Ezt a listát az `import sys` végrehajtásával érheti el. A lista hossza legalább egy; Ha nem adunk meg szkriptet és argumentumot, a `sys.argv[0]` egy üres karakterlánc. Ha a szkript neve '−' (vagyis szabványos bemenetet jelent), a `sys.argv[0]` értéke '−'. A `-c` *parancs* használatakor a sys.argv[0] értéke '`-c`'. Az `-m` *modul* használatakor a `sys.argv[0]` a megtalált modul teljes nevére van állítva. A `-c` *parancs* vagy `-m` *modul* után található opciókat nem használja fel a Python értelmező opció-feldolgozása, hanem a `sys.argv` fájlban hagyja a parancsot vagy modult kezelni.

2.1.2 Interaktív mód

Amikor a parancsokat egy tty-ből olvassa ki, az értelmező *interaktív módban* van. Ebben a módban a következő parancsot kéri az *elsődleges prompt*tal, általában három nagyobb jellel (>>>); a folytatásos soroknál a *másodlagos prompt*tal jelzi, alapértelmezés szerint három ponttal (`...`). Az értelmező kiír egy üdvözlő üzenetet, amely tartalmazza a verziószámát és egy szerzői jogi megjegyzést, mielőtt kiírja az első promptot:

```
$ python3.12
Python 3.12 (default, April 4 2021, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

A többsoros konstrukció beírásakor folytató sorokra van szükség. Példaként nézze meg ezt az `if` utasítást:

```
>>> a_fold_lapos = True
>>> if a_fold_lapos:
...     print("Vigyázzon, nehogy leessen!")
...
Vigyázzon, nehogy leessen!
```

Az interaktív módról bővebben az Interaktív mód című részben olvashat.

2.2 Az értelmező és környezete

2.2.1 A forráskód kódolása

Alapértelmezés szerint a Python-forrásfájlokat UTF-8 kódolásúként kezeli. Ebben a kódolásban a világ legtöbb nyelvének karakterei egyidejűleg használhatók karakterlánc-literálokban, azonosítókban és megjegyzésekben – bár a szabványos könyvtár csak ASCII-karaktereket használ az azonosítókhoz, ezt a konvenciót minden hordozható kódnak követnie kell. Ezen karakterek megfelelő megjelenítéséhez a szerkesztőnek fel kell ismernie, hogy a fájl UTF-8, és olyan betűtípust kell használnia, amely támogatja a fájl összes karakterét.

Az alapértelmezettől eltérő kódolás deklarálásához egy speciális megjegyzéssort kell hozzáadni a fájl első soraként. A szintaxis a következő:

```
# -*- coding: encoding -*-
```

ahol az *encoding* a Python által támogatott érvényes `kodekek` egyike.

Például annak deklarálásához, hogy Windows-1252 kódolást kell használni, a forráskód-fájl első sorának a következőnek kell lennie:

```
# -*- coding: cp1252 -*-
```

Az egyik kivétel az első sor szabálya alól, ha a forráskód *UNIX „shebang” sor*ral kezdődik. Ebben az esetben a kódolási deklarációt a fájl második soraként kell hozzáadni. Például:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

3. A Python laza bemutatása

A következő példákban a bemenetet és a kimenetet a promptok jelenléte vagy hiánya különbözteti meg (>>> és ...): a példa megismétléséhez mindent be kell írnia a prompt után, amikor a prompt megjelenik; azokat a sorokat, amelyek nem prompttal kezdődnek, az értelmező adja ki. Vegye figyelembe, hogy a példában egy sor másodlagos promptja önmagában azt jelenti, hogy üres sort kell beírnia; ez a többsoros parancs befejezésére szolgál.

Ebben a kézikönyvben sok példa, még azok is, amelyeket az interaktív promptnál írtak be, megjegyzéseket tartalmaznak. A Python megjegyzései kettős kereszt (hash) karakterrel kezdődnek, #, és a fizikai sor végéig terjednek. A megjegyzés megjelenhet egy sor elején vagy a szóköz vagy a kód után, de nem karakterlánc-literálon belül. A karakterláncon belüli hash karakter csak egy hash karakter. Mivel a megjegyzések a kód tisztázására szolgálnak, és a Python nem értelmezi őket, a példák beírásakor kihagyhatók.

Néhány példa:

```
# ez az első megjegyzés
spam = 1  # és ez a második megjegyzés
          # ... és most a harmadik!
text = "# Ez nem megjegyzés, mert idézőjelben van."
```

3.1. A Python használata számológépként

Próbáljunk ki néhány egyszerű Python-parancsot. Indítsa el az értelmezőt, és várja meg az elsődleges promptot, >>>. (Nem fog sokáig tartani.)

3.1.1 Számok

Az értelmező egyszerű számológépként működik: beírhat egy kifejezést, és kiírja az értékét. A kifejezés szintaxisa egyszerű: a +, -, * és / operátorok használhatók a számtani műveletek végrehajtásához; zárójelek (()) használhatók a csoportosításhoz. Például:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # az osztás mindig lebegőpontos számot ad vissza
1.6
```

Az egész számok (pl. 2, 4, 20) típusa int, a tört részt tartalmazók (pl. 5.0, 1.6) float típusúak. A numerikus típusokról az oktatóanyag későbbi részében többet fogunk látni.

Az osztás (/) mindig float típust ad vissza. Az egész osztás elvégzéséhez és az egész eredmény eléréséhez használhatja a // operátort; a maradék kiszámításához használhatja a %-ot:

```
>>> 17 / 3  # a klasszikus osztás float típust ad vissza
5.666666666666667
>>>
>>> 17 // 3  # az egész osztás elveti a tört részt
5
>>> 17 % 3  # a % operátor az osztás maradékát adja vissza
2
>>> 5 * 3 + 2  # egész hányados * osztó + maradék
17
```

A Python a ** operátor segítségével elvégezhető a hatványozás:

```
>>> 5 ** 2  # 5 a négyzeten
25
>>> 2 ** 7  # 2 a 7. hatványon
128
```

Mivel a ** prioritása magasabb, mint a – (negatív előjel), a -3**2 megadása -(3**2)-ként értelmeződik, és így -9-et eredményez. Ennek elkerüléséhez, a 9 eredményhez használhatja a (-3)**2 jelölést.

Az egyenlőségjel (=) értéket rendel egy változóhoz. Ezt követően a következő interaktív prompt előtt nem jelenik meg eredmény:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Ha egy változó nincs „definiálva” (érték nincs hozzárendelve), akkor a használatának kísérlete hibát jelez:

```
>>> n  # próbáljon meg hozzáférni egy nem definiált változóhoz
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

A lebegőpontos operandusok használata teljes mértékben támogatott; vegyes típusú operandusokkal rendelkező operátorok az egész operandust lebegőpontossá alakítják:

```
>>> 4 * 3.75 - 1
14.0
```

Interaktív módban az utoljára nyomtatott kifejezés a _ (aláhúzás) változóhoz van rendelve. Ez azt jelenti, hogy amikor a Pythont asztali számológépként használja, akkor valamivel könnyebb folytatni a számításokat, például:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Ezt a változót a felhasználónak „csak olvasható”-ként kell kezelnie. Ne rendeljen hozzá kifejezetten értéket – létrehozva egy független helyi változót ugyanazzal a névvel, amely elfedi a beépített változót a varázslatos viselkedésével.

Az int és a float mellett a Python más típusú számokat is támogat, mint például a `Decimal` és a `Fraction`. A Python a komplex számok beépített támogatásával is rendelkezik, és a `j` vagy a `J` utótagot használja a képzeletbeli rész (például `3+5j`) jelzésére.

3.1.2 Szöveg

A Python képes manipulálni a szöveget (amelyet `str` típusú, úgynevezett „karakterláncokkal” jelképez) valamint a számokat. Ez magában foglalja a `"!"` karaktereket, a `"rabbit"` szavakat, a `"Paris"` neveket, a `"Got your back."` mondatokat stb. `"Yay! :)"`. Ezeket szimpla idézőjelekbe (`'...'`) vagy kettős idézőjelekbe (`"..."`) lehet zárni, ugyanazzal az eredménnyel.

```
>>> 'spam eggs'  # szimpla idézőjel
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # dupla idézőjel
'Paris rabbit got your back :)! Yay!'
>>> '1975'
'1975'  # az idézőjelbe tett számjegyek és számok is karakterláncok
```

Más nyelvektől eltérően a speciális karakterek, mint például a `\n`, ugyanazt jelentik az egyes (`'...'`) és a kettős (`"..."`) idézőjelekkel. A kettő között az egyetlen különbség az, hogy az egyes idézőjelek között nem kell `\"` (de az egyes idézőjelet `\'`-ként kell írni) és fordítva.

Ha idézetet akarunk kiírni, az idézőjeleket „el kell menekítenünk” úgy, hogy előtte `\` jelet kell írni. Alternatív megoldásként használhatjuk az idézőjel másik típusát is:

```
>>> 'doesn\'t'  # a \'-vel a szimpla idézőjel nem jut érvényre...
"doesn't"
>>> "doesn't"  # ... vagy használjon helyette dupla idézőjeleket
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
```

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

A Python shellben a karakterlánc-definíció és a kimeneti karakterlánc eltérően nézhet ki. A `print()` függvény jobban olvasható eredményt ad az idézőjelek elhagyásával, valamint a megtisztított és speciális karakterek kinyomtatásával:

```
>>> s = 'Első sor.\nMásodik sor.'  # a \n újsort jelent
>>> s  # print() nélkül, a \n szerepel a kimenetben
'Első sor.\nMásodik sor.'
>>> print(s)  # a print() használatával a \n új sort állít elő
Első sor.
Második sor.
```

Ha nem szeretné, hogy a \ előtagú karakterek speciális karakterként legyenek értelmezve, használhatunk nyers karakterláncokat, ha az első idézőjel elé egy `r` karaktert adunk:

```
>>> print('C:\some\name')  # itt a \n újsort jelent!
C:\some
ame
>>> print(r'C:\some\name')  # figyelje az r betűt az idézőjel előtt
C:\some\name
```

A nyers karakterláncoknak van egy finom aspektusa: a nyers karakterlánc nem végződhet páratlan számú \ karakterrel; További információkért és megoldásokért tekintse meg a GYIK bejegyzést.

A karakterláncok több sort is átívelhetnek. Az egyik módja a hármas idézőjelek használata: `"""..."""` vagy `'''...'''`. A sorvég karakter automatikusan bekerül a karakterláncba, de ez megakadályozható egy \ karakter hozzáadásával a sor végén. A következő példa:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

a következő kimenetet hozza létre (figyeljük meg, hogy a kezdő újsor nem szerepel):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

A karakterláncok összefűzhetők (összeragaszthatók) a + operátorral, és ismételgethetők a *-al:

```
>>> # 3-szor 'un', követi az 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Két vagy több karakterlánc (azaz az idézőjelek közé zárt) egymás mellett automatikusan összefűződik.

```
>>> 'Py' 'thon'
'Python'
```

Ez a funkció különösen akkor hasznos, ha hosszú karakterláncokat szeretne megszakítani:

```
>>> text = (Tegyen több karakterláncot zárójelbe, '
...         ' hogy összekapcsolja őket.')
>>> text
'Tegyen több karakterláncot zárójelbe, hogy összekapcsolja őket.'
```

Ez azonban csak két literállal működik, változókkal vagy kifejezésekkel nem:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # nem fűzhet össze változót és karakterláncot
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

Ha változókat vagy változót és karakterláncot szeretne összefűzni, használja a +-t:

```
>>> prefix + 'thon'
'Python'
```

A karakterláncok *indexelhetők* (szeletelhetők), az első karakter indexe 0. Nincs külön karakter típus; a karakter egyszerűen egy egyelemű karakterlánc:

```
>>> word = 'Python'
>>> word[0]  # karakter a 0. pozícióban
'P'
>>> word[5]  # karakter az 5. pozícióban
'n'
```

Az indexek negatív számok is lehetnek, ha jobbról kezdjük a számolást:

```
>>> word[-1]  # az utolsó karakter
'n'
>>> word[-2]  # az utolsó előtti karakter
'o'
>>> word[-6]
'P'
```

Vegye figyelembe, hogy mivel a -0 ugyanaz, mint a 0, a negatív indexek -1-től kezdődnek.

Az indexelés mellett a *szeletelés* is támogatott. Míg az indexelést egyedi karakterek megszerzésére használják, a *szeletelés* lehetővé teszi a rész-karakterlánc megszerzését:

```
>>> word[0:2]  # karakterek a 0.-tól (beleértve) a 2.-ig (kizárva)
'Py'
>>> word[2:5]  # karakterek a 2.-tól (beleértve) az 5.-ig (kizárva)
'tho'
```

A szeletindexeknek hasznos alapértékei vannak; a kihagyott első index alapértelmezés szerint nulla, a kihagyott második index pedig a szeletelendő karakterlánc méretét veszi alapul.

```
>>> word[:2]  # karakterek az elejétől a 2. pozícióig (kizárva)
'Py'
>>> word[4:]  # karakterek a 4. pozíciótól (beleértve) a végéig
'on'
>>> word[-2:]  # karakterek az utolsó előttitől (beleértve) a végéig
'on'
```

Vegye figyelembe, hogy a kezdő karakter mindig szerepel, és a vége karapter mindig ki van zárva. Ez biztosítja, hogy `s[:i] + s[i:]` mindig egyenlő `s`-sel:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Az egyik módja annak, hogy megjegyezzük, hogyan működnek a szeletek, ha úgy tekintjük az indexeket, mint amelyek a karakterek *közé* mutatnak, és az első karakter bal széle 0. Ekkor egy n karakterből álló karakterlánc utolsó karakterének jobb szélén az n index van, például:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Az első számsor megadja a 0…6 indexek pozícióját a sztringben; a második sor a megfelelő negatív indexeket adja meg. Az i-től j-ig tartó szelet az i és j címkékkel ellátott élek közötti összes karakterből áll.

Nem negatív indexek esetén a szelet hossza az indexek különbsége, ha mindkettő a határokon belül van. Például a `word[1:3]` hossza 2.

Ha túl nagy indexet próbál használni, az hibát eredményez:

```
>>> word[42]  # a word csak 6 karaktert tartalmaz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

A tartományon kívüli szeletindexeket azonban a rendszer elegánsan kezeli, ha szeleteléshez használják őket:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

A Python karakterláncait nem lehet megváltoztatni – megváltoztathatatlanok. Ezért egy indexelt pozícióhoz való hozzárendelés a karakterláncban hibát eredményez:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Ha más karakterláncra van szüksége, hozzon létre egy újat:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

A beépített `len()` függvény egy karakterlánc hosszát adja vissza:

```
>>> s = 'szuperkacifántosanterjengős'
>>> len(s)
27
```

Lásd még:

textseq Sztringek, a szekvenciális típusok példái, és támogatják az ilyen típusok által támogatott általános műveleteket.

sztring metódusok A karakterláncok számos módszert támogatnak az alapvető átalakításokhoz és kereséshez.

f-sztringek Sztring literálok, amelyek beágyazott kifejezéseket tartalmaznak.

formázott sztringek Információk a `str.format()` karakterlánc-formázásáról.

régi sztring formázások Itt részletesebben leírjuk azokat a régi formázási műveleteket, amelyeket akkor hívtak meg, amikor a karakterláncok a % operátor bal oldali operandusai.

3.1.3 Listák

A Python számos összetett adattípust ismer, amelyeket különböző értékek csoportosítására használnak. A legsokoldalúbb a lista, amely szögletes zárójelek közé vesszővel elválasztott értékek (elemek) listájaként írható. A listák különböző típusú elemeket tartalmazhatnak, de általában mindegyik elem azonos típusú.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

A karakterláncokhoz (és minden más beépített szekvenciális típushoz) hasonlóan a listák is indexelhetők és szeletelhetők:

```
>>> squares[0]  # az indexelés visszaadja az elemet
1
>>> squares[-1]
25
>>> squares[-3:]  # A szeletelés új listát ad vissza
[9, 16, 25]
```

Minden szeletművelet egy új listát ad vissza, amely tartalmazza a kért elemeket. Ez azt jelenti, hogy a következő szelet a lista másolatát adja vissza:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

A listák olyan műveleteket is támogatnak, mint az összefűzés:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Ellentétben a karakterláncokkal, amelyek nem változtathatók, a listák változtatható típusúak, azaz megváltoztatható a tartalmuk:

```
>>> cubes = [1, 8, 27, 65, 125]  # itt valami nincs rendben
>>> 4 ** 3  # 4 köbe 64, nem 65!
64
>>> cubes[3] = 64  # kicseréljük a rossz értéket
>>> cubes
[1, 8, 27, 64, 125]
```

Új elemeket is felvehet a lista végére a `list.append()` metódussal (a metódusokról később még többet fogunk látni):

```
>>> cubes.append(216)    # hozzáadjuk 6 köbét
>>> cubes.append(7 ** 3)    # és 7 köbét
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

A szeletekhez való hozzárendelés is lehetséges, és ez akár megváltoztathatja a lista méretét vagy teljesen törölheti azt:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # kicserélünk néhány értéket
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # most eltávolítjuk őket
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # töröljük a listát úgy hogy üres listára cserélük
>>> letters[:] = []
>>> letters
[]
```

A beépített `len()` függvény a listákra is alkalmazható:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Lehetőség van listák egymásba ágyazására (más listákat tartalmazó listák létrehozására), például:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Első lépések a programozás felé

Természetesen a Pythont bonyolultabb feladatokra is használhatjuk, mint kettőt és kettőt összeadni. Például felírhatjuk a Fibonacci sorozat kezdeti részsorozatát a következőképpen:

```
>>> # Fibonacci sorozat:
... # két elem összege határozza meg a következőt
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
```

```
...
0
1
1
2
3
5
8
```

Ez a példa számos új sajátosságot mutat be.

- Az első sor egy *többszörös hozzárendelést* tartalmaz: az `a` és `b` változók egyszerre kapják az új 0 és 1 értéket. Az utolsó sorban ezt ismét használjuk, bizonyítva, hogy a jobb oldali kifejezések mindegyike előbb kerül kiértékelésre, minthogy bármelyik hozzárendelés megtörténik. A jobb oldali kifejezések kiértékelése balról jobbra történik.

- A `while` ciklus addig fut, amíg a feltétel (itt: `a < 10`) igaz marad. A Pythonban, akárcsak a C-ben, minden nullától eltérő egész érték igaz; a nulla hamis. A feltétel lehet karakterlánc vagy listaérték is, valójában bármilyen sorozat; minden, aminek hossza nem nulla igaz, az üres sorozatok hamisak. A példában használt teszt egy egyszerű összehasonlítás. A szabványos összehasonlító operátorok ugyanúgy vannak felírva, mint a C-ben: < (kisebb, mint), > (nagyobb, mint), == (egyenlő), <= (kisebb vagy egyenlő), >= (nagyobb vagy egyenlő) és != (nem egyenlő).

- A ciklus törzse behúzott: a behúzás a Python módszere az utasítások csoportosítására. Az interaktív promptnál minden behúzott sorhoz tabulátort vagy szóközt kell beírnia. A gyakorlatban bonyolultabb bevitelt készítünk a Python számára egy szövegszerkesztővel; minden tisztességes szövegszerkesztő rendelkezik automatikus behúzási lehetőséggel. Ha egy összetett utasítást interaktívan ír be, azt egy üres sornak kell követnie a befejezés jelzésére (mivel az elemző nem tudja kitalálni, mikor írta be az utolsó sort). Ne feledje, hogy az alapblokkon belül minden sort ugyanannyival kell behúzni.

- A print() függvény kiírja a neki adott argumentum(ok) értékét. A több argumentum, lebegőpontos mennyiségek és karakterláncok kezelésében különbözik a megírni kívánt kifejezés megírásától (ahogyan azt korábban a számológép példáiban megtettük). A karakterláncok idézőjelek nélkül kerülnek kinyomtatásra, az elemek közé pedig szóköz kerül, így szépen formázhatod a dolgokat, például így:

```
>>> i = 256*256
>>> print('Az i értéke', i)
Az i értéke 65536
```

  Az *end* kulcsszót argumentumként felhasználhatja az új sorba való kiírás elkerülésére a kimenet után, vagy a kimenetet egy másik karakterlánccal fejezheti be:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
```

```
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

4. További folyamatvezérlő eszközök

Az imént bemutatott `while` utasítás mellett a Python még néhányat használ, amelyekkel ebben a fejezetben találkozunk.

4.1 Az `if` utasítás

Talán a legismertebb utasítástípus az `if` utasítás. Például:

```
>>> x = int(input("Adjon meg egy egész számot: "))
Adjon meg egy egész számot: 42
>>> if x < 0:
...     x = 0
...     print('A negatív nullára változtatva')
... elif x == 0:
...     print('Nulla')
... elif x == 1:
...     print('Egyes')
... else:
...     print('Több')
...
Több
```

Lehet nulla vagy több `elif` rész, az `else` rész pedig opcionális. Az „`elif`” kulcsszó az „else if” rövidítése, és hasznos a túlzott behúzás elkerülése érdekében. Az `if … elif … elif …` szekvencia helyettesíti a más nyelvekben található `switch` vagy `case` utasításokat.

Ha ugyanazt az értéket több konstanshoz hasonlítja össze, vagy bizonyos típusokat vagy attribútumokat keres, akkor a `match` utasítás is hasznos lehet. További részletekért lásd az *Utasítások részletezése* fejezetet.

4.2 A `for` utasítás

A Python for utasítása kissé eltér attól, amit a C vagy Pascal nyelvben megszokhattunk. Ahelyett, hogy mindig a számok számtani sorozatán iterálna (mint a Pascalban), vagy lehetőséget adna a felhasználónak az iterációs lépésköz és a leállítási feltétel meghatározására (C-ként), a Python utasításai tetszőleges sorozat elemei felett iterálnak (lista vagy karakterlánc), abban a sorrendben, ahogyan a sorozatban megjelennek. Például (nem szójáték):

```
>>> # Megmérünk néhány karakterláncot:
... words = ['rák', 'ablak', 'mosogatószivacs']
>>> for w in words:
...     print(w, len(w))
...
```

```
rák 3
ablak 5
mosogatószivacs 15
```

Olyan kódban, amely módosít egy gyűjteményt, miközben ugyanazon a gyűjteményen keresztül iterál, bonyolult lehet a helyes megoldás. Ehelyett általában egyszerűbb a gyűjtemény egy példányának átugrása vagy új gyűjtemény létrehozása:

```
# Példa gyűjtemény létrehozása
users = {'Tibi': 'aktív', 'Vali': 'inaktív', '???': 'aktív'}

# Stratégia: Iteráljon egy másolaton
for user, status in users.copy().items():
    if status == 'inaktív':
        del users[user]

# Stratégia: Létrehozunk egy új gyűjteményt
active_users = {}
for user, status in users.items():
    if status == 'aktív':
        active_users[user] = status
```

4.3 A `range()` függvény

Ha számsoron kell ismételnie, a beépített `range()` függvény jól jöhet. Számtani növekményt generál:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Az adott végpont soha nem része a generált sorozatnak; A `range(10)` 10 értéket generál, a szabályos indexszel egy 10 elem hosszúságú sorozathoz. Lehetőség van arra, hogy a tartomány egy másik számmal kezdődjön, vagy egy másik növekményt adjon meg (akár negatívat is; ezt néha „lépésköznek” is nevezik):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Egy sorozat indexein való iterációhoz kombinálhatja a `range()` és `len()` függvényeket az alábbiak szerint:

```
>>> a = ['Marynek', 'van', 'egy', 'kis', 'bárányа']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Marynek
1 van
2 egy
3 kis
4 bárányа
```

A legtöbb ilyen esetben azonban kényelmes az `enumerate()` függvény használata, lásd: *Looping technikák* című részt.

Furcsa dolog történik, ha csak egy tartományt irat ki:

```
>>> range(10)
range(0, 10)
```

A `range()` által visszaadott objektum sok szempontból úgy viselkedik, mintha egy lista lenne, de valójában nem az. Ez egy olyan objektum, amely visszaadja a kívánt sorozat egymás utáni elemeit, amikor iterál rajta, de valójában nem kerül fel a listára, így helyet takarít meg.

Azt mondjuk, hogy egy ilyen objektum *iterálható*, azaz válhat olyan függvények és konstrukciók célpontjává, amelyek olyasmit várnak, amitől a készlet kimerüléséig egymást követő tételeket kaphatnak. Láttuk, hogy a `for` utasítás egy ilyen konstrukció, míg egy iterálható függvényre példa a `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Később még több olyan függvényt fogunk látni, amelyek ismétléseket adnak vissza, és az iterálhatóakat argumentumnak veszik. Az *Adatstruktúrák* fejezetben részletesebben tárgyalunk a `list()`-ről.

4.4 A `break` és a `continue` utasítások, valamint az `else` záradék ciklusoknál

A `break` utasítás „kiugrik” a legbelső `for` vagy `while` ciklusból.

A `for` vagy `while` ciklus tartalmazhat egy `else` záradékot.

A `for` ciklusban az `else` záradék akkor kerül végrehajtásra, amikor a ciklus eléri a végső iterációt.

Egy `while` ciklusban azután kerül végrehajtásra, hogy a ciklus feltétele hamis lesz.

Egyik típusú ciklusban **sem** hajtódik végre az `else` záradék, ha a ciklust egy `break` zárta le.

Ezt példázza a következő `for` ciklus, amely prímszámokat keres:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'egyenlő', x, '*', n//x)
...             break
...     else:
...         # a ciklus átesett anélkül, hogy tényezőt talált volna
...         print(n, 'egy prím szám')
...
2 egy prím szám
3 egy prím szám
4 egyenlő 2 * 2
5 egy prím szám
6 egyenlő 2 * 3
7 egy prím szám
8 egyenlő 2 * 4
9 egyenlő 3 * 3
```

(Igen, ez a helyes kód. Nézd meg alaposan: az `else` záradék a `for` ciklushoz tartozik, **nem** az `if` utasításhoz.)

Ha ciklussal együtt használjuk, az `else` záradéknak több közös vonása van a `try` utasítás `else` tagjával, mint az `if` utasításokkal: a `try` utasítás `else` záradéka akkor fut le, ha nem történik kivétel, és a ciklus `else` záradéka akkor fut, ha nincs `break`. A `try` utasításról és a kivételekről további információt a *Kivételek kezelése* című témakörben talál.

A szintén C-ből kölcsönzött `continue` utasítás a ciklus következő iterációjával folytatódik:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Páros számot találtunk", num)
...         continue
...     print("Páratlan számot találtunk", num)
...
Páros számot találtunk 2
Páratlan számot találtunk 3
Páros számot találtunk 4
Páratlan számot találtunk 5
Páros számot találtunk 6
Páratlan számot találtunk 7
Páros számot találtunk 8
Páratlan számot találtunk 9
```

4.5 A `pass` utasítás

A `pass` utasítás nem tesz semmit. Akkor használható, ha egy utasítás szintaktikailag szükséges, de a program nem igényel semmilyen műveletet. Például:

```
>>> while True:
...     pass  # Várakozás a billentyűzet megszakítására (Ctrl+C)
...
```

Ezt általában minimális osztályok létrehozására használják:

```
>>> class MyEmptyClass:
...     pass
...
```

Egy másik hely ahol a `pass` használható, egy függvény vagy feltételes törzs helyének biztosítójaként, amikor új kódon dolgozik, lehetővé téve, hogy továbbra is elvontabb szinten gondolkodjon. A `pass`-t csendben figyelmen kívül hagyják:

```
>>> def initlog(*args):
...     pass  # Ne felejtse el ezt megvalósítani!
...
```

4.6 A `match` utasítás

A `match` utasítás egy kifejezést vesz fel, és összehasonlítja annak értékét egy vagy több eseti blokként megadott egymást követő mintákkal. Ez felületesen hasonlít a C, Java vagy JavaScript (és sok más nyelv) `switch` utasításaihoz, de jobban hasonlít a mintaillesztéshez olyan nyelvekben, mint a Rust vagy a Haskell. Csak az első egyező minta kerül végrehajtásra, és komponenseket (szekvencia-elemeket vagy objektum-attribútumokat) is ki tud kinyerni az értékből változókká.

A legegyszerűbb forma a tárgyértéket egy vagy több literálhoz hasonlítja:

```
def http_error(status):
    match status:
        case 400:
            return "Rossz kérés"
        case 404:
            return "Nem található"
        case 418:
            return "Teáskanna vagyok"
        case _:
            return "Valami nem stimmel az internettel"
```

Megjegyzés az utolsó blokkhoz: a _ „változónév” *helyettesítő karakterként* működik, amely akkor fut le, ha semmi sem egyezik. Ha egyik eset sem egyezik, akkor egyik ág sem hajtódik végre.

A | ("vagy") használatával több literált is kombinálhat egyetlen mintában:

```
case 401 | 403 | 404:
     return "Nem megengedett"
```

A minták kicsomagolási feladatoknak tűnhetnek, és változók kötésére használhatók:

```
# a point egy (x, y) tuple
match point:
    case (0, 0):
        print("Origó")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Nem pont")
```

Tanulmányozd ezt alaposan! Az első mintának két literálja van, és a fent bemutatott literális minta kiterjesztésének tekinthető. De a következő két minta egyesít egy literált és egy változót, és a változó egy értéket köt a tárgyból (`point`). A negyedik minta két értéket rögzít, ami elvileg hasonló a kicsomagolási hozzárendeléshez `(x, y) = point`.

Ha osztályokat használ az adatok strukturálására, használhatja az osztály nevét, majd egy konstruktorra emlékeztető argumentumlistát, amely képes az attribútumokat változókba rögzíteni:

```
class Point:
    def __init__(self, x, y)
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origó")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Valahol máshol")
        case _:
            print("Nem pont")
```

Használhat pozícióparamétereket néhány beépített osztályhoz, amelyek az attribútumok sorrendjét biztosítják (például adatosztályok). A mintákban az attribútumok konkrét pozícióját is megadhatja a

`__match_args__` speciális attribútum beállításával az osztályokban. Ha értéke ("x", "y"), a következő minták mindegyike egyenértékű (és mindegyik köti az `y` attribútumot a `var` változóhoz):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A minták olvasásának ajánlott módja, ha úgy tekintünk rájuk, mint annak egy kiterjesztett formájára, amit egy feladat bal oldalán helyeznénk el, hogy megértsük, mely változókat mire kell beállítani. Csak az önálló nevek (mint a fenti `var`) vannak hozzárendelve egy `match` utasítással. A pontozott nevek (például a `foo.bar`), az attribútum-nevek (fent az `x=` és `y=`) vagy az osztálynevek (amelyeket a mellettük lévő „(…)” jel ismer fel, mint a fenti `Point`) soha nem rendelhető hozzá.

A minták tetszőlegesen egymásba ágyazhatók. Például, ha van egy rövid pontlistánk, a `__match_args__` hozzáadásával, akkor a következőképpen párosíthatjuk:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("Nincs pont")
    case [Point(0, 0)]:
        print("Az origó")
    case [Point(x, y)]:
        print(f"Egyetlen pont {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Kettő az Y tengelyen {y1}, {y2}")
    case _:
        print("Valahol máshol")
```

Hozzáadhatunk egy `if` záradékot a mintához, amelyet „őrnek” neveznek. Ha az őr hamis, a `match` továbbmegy, és megpróbálja a következő esetblokkot. Vegye figyelembe, hogy az értékrögzítés az őr kiértékelése előtt megtörténik:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Nincs az átlón")
```

Ennek az utasításnak van néhány további fontos jellegzetessége:

- A hozzárendelések kicsomagolásához hasonlóan a sor- és listamintáknak is pontosan ugyanaz a jelentése, és valójában tetszőleges szekvenciákkal egyeznek. Fontos kivétel, hogy nem egyeznek az iterátorokkal vagy karakterláncokkal.

- A sorozatminták támogatják a kiterjesztett kicsomagolást: `[x, y, *rest]` és `(x, y, *rest)` a kicsomagolási feladatokhoz hasonlóan működnek. A `*` utáni név `_` is lehet, így az `(x, y, *_)` legalább két elemből álló sorozatnak felel meg anélkül, hogy a többi elemet összekötné.

- Leképezési minták: `{"bandwidth": b, "latency": l}` rögzíti a `"bandwidth"` és a `"latency"` értékeket egy szótárból. A sorozatmintákkal ellentétben az extra kulcsokat figyelmen kívül hagyja. Az olyan kicsomagolás is támogatott, mint a `**rest`. (De a `**_` felesleges lenne, ezért nem megengedett.)

- Az alminták az `as` kulcsszó használatával rögzíthetők:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

  a bemenet második elemét p2-ként fogja rögzíteni (amíg a bemenet két pontból álló sorozat)

- A legtöbb literált az egyenlőség alapján hasonlítja össze, azonban a True, False és None singliket azonosság alapján hasonlítják össze.

- A minták nevesített konstansokat használhatnak. Ezeknek pontozott neveknek kell lenniük, nehogy rögzített változóként értelmezzék őket:

```
from enum import Enum
class Color(Enum):
    RED = 'piros'
    GREEN = 'zöld'
    BLUE = 'kék'

color = Color(input("Írd be a választott színt 'piros', 'kék'
vagy 'zöld': "))

match color:
    case Color.RED:
       print("Pirosat látok!")
    case Color.GREEN:
        print("A fű zöld")
    case Color.BLUE:
        print("Szomorú vagyok :(")
```

Részletesebb magyarázatért és további példákért tekintse meg a PEP 636-ot, amely oktatóanyag formátumban van megírva.

4.7 Függvények definiálása

Létrehozhatunk egy függvényt, amely a Fibonacci sorozatot tetszőleges határig írja:

```
>>> def fib(n):  # Fibonacci sorozatot írunk n-ig
...     """ Fibonacci sorozatot írunk n-ig."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Most hívjuk meg az imént definiált függvényt:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A `def` kulcsszó egy *függvénydefiníciót* vezet be. Ezt követnie kell a függvény nevének és a formális paraméterek zárójeles listájának. A függvény törzsét alkotó utasítások a következő sorban kezdődnek, és be kell húzni őket.

A függvénytörzs első utasítása opcionálisan lehet karakterlánc-literál; ez a karakterlánc literál a függvény dokumentációs karakterlánca vagy doksztringje. (A doksztringekről bővebben a Dokumentációs karakterláncok részben olvashat.) Vannak olyan eszközök, amelyek a doksztringek segítségével automatikusan online vagy nyomtatott dokumentációt készítenek, vagy lehetővé teszik a felhasználó számára, hogy interaktívan böngésszen a kódban; bevált gyakorlat, ha dokumentum-karakterláncokat tartalmaz az írott kód, ezért szokjon hozzá.

Egy függvény végrehajtása egy új szimbólumtáblázatot vezet be a függvény helyi változóihoz. Pontosabban, egy függvényben minden változó hozzárendelés tárolja az értéket a helyi szimbólumtáblázatban; míg a változóhivatkozások először a helyi szimbólumtáblázatban, majd a befoglaló függvények helyi szimbólumtáblázatában, majd a globális szimbólumtáblázatban, végül a beépített nevek táblázatában keresnek. Így a globális változókhoz és a befoglaló függvények változóihoz nem lehet közvetlenül hozzárendelni értéket egy függvényen belül (kivéve, ha a globális változók esetében egy globális utasításban van megnevezve, vagy a befoglaló függvények változóinál nem lokális utasításban), bár lehet, hogy hivatkozott.

A függvényhívás aktuális paraméterei (argumentumai) a meghívott függvény helyi szimbólumtáblázatába kerülnek behíváskor; így az argumentumok hívásonkénti értékkel kerülnek átadásra (ahol az érték mindig egy objektumhivatkozás, nem pedig az objektum értéke). Amikor egy függvény meghív egy másik függvényt, vagy rekurzívan hívja meg magát, egy új helyi szimbólumtábla jön létre az adott híváshoz.

Egy függvénydefiníció társítja a függvény nevét az aktuális szimbólumtáblázat függvényobjektumához. Az értelmező az ezen a néven hivatkozott objektumot felhasználó által definiált függvényként ismeri fel. Más nevek is mutathatnak ugyanarra a függvényobjektumra, és a függvény eléréséhez is használhatók:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Más nyelvekből kifolyólag mondhatja, hogy a `fib` nem függvény, hanem eljárás, mivel nem ad vissza értéket. Valójában még a `return` utasítás nélküli függvények is visszaadnak egy értéket, bár meglehetősen unalmast. Ennek az értéknek a neve `None` (ez egy beépített név). A `None` érték beírását az értelmező általában elnyomja, ha ez lenne az egyetlen írt érték. Megnézheti, ha valóban akarja, a `print()` használatával:

```
>>> fib(0)
>>> print(fib(0))
None
```

Egyszerűen írhatunk egy függvényt, amely a Fibonacci-sorozat számainak listáját adja vissza, ahelyett, hogy kinyomtatná:

```
>>> def fib2(n):  # visszaadja a Fibonacci sorozatot n-ig
...     """Visszaad egy listát, amely tartalmazza a Fibonacci soro-
zatot n-ig."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)   # lásd alább
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)  # meghívjuk
>>> f100              # kiírja az eredményt
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Ez a példa, mint általában, néhány új Python jellegzetességet mutat be:

- A `return` utasítás egy függvény értékével tér vissza. `return` kifejezés nélkül argumentum visszatérési értéke `None`. Egy függvény végéről való leesés is `None`-t ad vissza.

- A `result.append(a)` utasítás meghívja a listaobjektum eredményének metódusát. A metódus egy objektumhoz tartozó függvény, amelynek neve `obj.methodname`, ahol az `obj` valamilyen objektum (ez lehet egy kifejezés), a `methodname` pedig egy metódus neve, amelyet az objektum típusa határoz meg. A különböző típusok különböző metódusokat határoznak meg. A különböző típusú metódusoknak lehet ugyanaz a neve, anélkül, hogy kétértelműséget okoznának. (Lehetőség van saját objektumtípusok és metódusok definiálására, osztályok segítségével, lásd Osztályok) A példában bemutatott `append()` metódus listaobjektumokhoz van definiálva; új elemet ad a lista végére. Ebben a példában ez ekvivalens a `result = result + [a]`-val, de hatékonyabb.

4.8 További információk a függvények meghatározásáról

Lehetőség van változó számú argumentumú függvények definiálására is. Három formája van, amelyek kombinálhatók.

4.8.1 Alapértelmezett argumentumértékek

A leghasznosabb forma az alapértelmezett érték megadása egy vagy több argumentumhoz. Ez létrehoz egy függvényt, amely kevesebb argumentummal hívható meg, mint amennyit a definíció engedélyez. Például:

```
def ask_ok(prompt, retries=4, reminder='Kérlek, próbáld újra!'):
    while True:
        ok = input(prompt)
        if ok in ('i', 'igen', 'y', 'yes'):
            return True
        if ok in ('n', 'no', 'nem', 'not'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('érvénytelen felhasználói válasz')
        print(reminder)
```

Ez a függvény többféleképpen hívható:

- csak a kötelező argumentumot adja meg: `ask_ok('Valóban ki akar lépni?')`
- megadva az egyik opcionális argumentumot: `ask_ok('Rendben van a fájl felülírása?', 2)`
- minden argumentumot megadva: `ask_ok('Rendben van a fájl felülírása?', 2, 'Gyerünk, igen vagy nem!')`

Ez a példa az `in` kulcsszót is bemutatja. Ez azt vizsgálja, hogy egy sorozat tartalmaz-e egy bizonyos értéket.

Az alapértelmezett értékek a definiáló hatókör függvénydefiníciójának pontján kerülnek kiértékelésre, így

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

5-öt fog kiírni.

Fontos figyelmeztetés: Az alapértelmezett érték csak egyszer kerül kiértékelésre. Ez különbséget jelent, ha az alapértelmezett egy változtatható objektum, például egy lista, szótár vagy a legtöbb osztály példányai. Például a következő függvény összegyűjti a neki átadott argumentumokat a következő hívásoknál:

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

print(f(1))
print(f(2))
print(f(3))
```

Ez a következőt fogja kiírni:

```
[1]
[1, 2]
[1, 2, 3]
```

Ha nem szeretné, hogy az alapértelmezett értéket megosszák a következő hívások között, akkor helyette a következőképpen írhatja be a függvényt:

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

4.8.2 Kulcsszó-argumentumok

A függvények a `kwarg=value` formájú kulcsszó-argumentumok használatával is meghívhatók. Például a következő függvény:

```
def parrot(voltage, state='merev', action='dörmögne', type='norvég kék'):
    print("-- Ez a papagáj", action, end=' ')
    print("ha", voltage, "voltot vezetnél át rajta.")
    print("-- Gyönyörű tollazat, a", type)
    print("-- Ez", state, "!")
```

elfogad egy kötelező argumentumot (`voltage`) és három választható argumentumot (`state`, `action` és `type`). Ez a függvény a következő módok bármelyikén meghívható:

```
parrot(1000)                                  # 1 pozícionált argumentum
parrot(voltage=1000)                          # 1 kulcsszó argumentum
parrot(voltage=1000000, action='HÚÚÚZNA')     # 2 kulcsszó argumentum
parrot(action='HÚÚÚZNA', voltage=1000000)     # 2 kulcsszó argumentum
parrot('egy millió', 'feldobta a talpát', 'ugrálna')  # 3 pozícionált argumentum
parrot('ezer', state='alulról szagolja az ibolyát')  # 1 pozícionált, 1 kulcsszó
```

de a következő hívások mindegyike érvénytelen:

```
parrot()                     # a kötelező argumentum hiányzik
parrot(voltage=5.0, 'dead')  # nem-kulcsszó argumentum kulcsszó argumentum után
parrot(110, voltage=220)     # duplikált argumentum érték
parrot(actor='John Cleese')  # ismeretlen kulcsszó argumentum
```

Egy függvényhívásban a kulcsszó argumentumainak követniük kell a pozíció argumentumokat. Az összes átadott kulcsszó argumentumnak meg kell egyeznie a függvény által elfogadott argumentumok egyikével (pl. az `actor` nem érvényes argumentum a `parrot` függvényhez), és a sorrendjük nem fontos. Ez magában foglalja a nem opcionális argumentumokat is (pl. a `parrot(voltage=1000)` is érvényes). Egy argumentum sem kaphat többször értéket. Íme egy példa, amely e korlátozás miatt meghiúsul:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Ha egy végső formális paraméter a `**name` formában jelen van, akkor egy szótárt kap (lásd a típusleképezést), amely tartalmazza az összes kulcsszó-argumentumot, kivéve azokat, amelyek egy formális paraméternek felelnek meg. Ez kombinálható egy formális paraméterrel, amely a `*name` formátumú (a következő alfejezetben lesz leírva), amely a formális paraméterlistán túlmenően a pozíciós argumentumokat tartalmazó leírót kapja. (A `*name`-nek a `**name` előtt kell előfordulnia.) Például, ha így definiálunk egy függvényt:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Kapható", kind, "?")
    print("-- Sajnálom, elfogyott a", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Ezt így lehetne meghívni:

```
cheeseshop("Limburger", "Nagyon folyós, uram.",
           "Tényleg nagyon, NAGYON folyós, uram.",
           boltos="Michael Palin",
           ugyfel="John Cleese",
           sketch="Sajt Bolt Sketch")
```

és persze kiírná:

```
-- Kapható Limburger ?
-- Sajnálom, elfogyott a Limburger
Nagyon folyós, uram.
```

```
Tényleg nagyon, NAGYON folyós, uram.
----------------------------------------
boltos : Michael Palin
ugyfel : John Cleese
sketch : Sajt Bolt Sketch
```

Vegye figyelembe, hogy a kulcsszó-argumentumok nyomtatási sorrendje garantáltan megegyezik a függvényhívásban megadott sorrenddel.

4.8.3 Speciális paraméterek

Alapértelmezés szerint az argumentumok átadhatók egy Python-függvénynek pozíció vagy kifejezetten kulcsszó alapján. Az olvashatóság és a teljesítmény érdekében célszerű korlátozni az argumentumok átadásának módját, így a fejlesztőnek csak a függvénydefiníciót kell megnéznie annak meghatározásához, hogy az elemeket csak-pozíció, pozíció-vagy-kulcsszó vagy csak-kulcsszó szerint továbbítja-e.

A függvény definíciója így nézhet ki:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |          Pozíció vagy kulcsszó |
        |                                - Csak kulcsszó
         -- Csak pozíció
```

ahol a `/` és a `*` nem kötelező. Ha használják, ezek a szimbólumok jelzik a paraméter típusát azáltal, hogy az argumentumok hogyan adhatók át a függvénynek: csak-pozíció, pozíció-vagy-kulcsszó és csak-kulcsszó. A kulcsszó-paramétereket elnevezett paramétereknek is nevezik.

Pozíció-vagy-kulcsszó argumentumok

Ha a / és a * nem szerepel a függvénydefinícióban, az argumentumok vagy pozíció, vagy kulcsszó alapján adhatók át a függvénynek.

Csak-pozíciós paraméterek

Ha ezt kicsit részletesebben nézzük, lehetséges, hogy bizonyos paramétereket csak-pozícióként jelöljünk meg. Ha csak-pozíciós, akkor a paraméterek sorrendje számít, és a paraméterek nem adhatók át kulcsszóval. A csak-pozíció paraméterek a `/` (rendes perjel) elé kerülnek. A `/` a csak-pozíció paraméterek logikai elválasztására szolgál a többi paramétertől. Ha a függvény definíciójában nincs `/`, akkor nincsenek csak-pozíciós paraméterek.

A `/` jelet követő paraméterek lehetnek *pozíció-vagy-kulcsszó* vagy *csak-kulcsszó* paraméterek.

Csak-kulcsszó argumentumok

Ha a paramétereket csak-kulcsszósként kívánja megjelölni, jelezve, hogy a paramétereket kulcsszó-argumentumnak kell átadnia, helyezzen `*`-t az argumentumlistába közvetlenül az első csak-kulcsszót tartalmazó paraméter elé.

Függvény példák

Tekintsük a következő példafüggvény-definíciókat, különös figyelmet fordítva a `/` és `*` jelzőkre:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

Az első függvénydefiníció, a `standard_arg`, a legismertebb forma, nem korlátozza a hívási konvenciót, és az argumentumok átadhatók pozíciónként vagy kulcsszónként:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

A második `pos_only_arg` függvény csak-pozíció paraméterek használatára korlátozódik, mivel a függvénydefinícióban van egy `/`:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as
keyword arguments: 'arg'
```

A harmadik `kwd_only_args` függvény csak a kulcsszó-argumentumot engedélyezi, amint azt a `*` karakter jelzi a függvénydefinícióban:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given
```

```
>>> kwd_only_arg(arg=3)
3
```

Az utolsó pedig mindhárom hívási konvenciót használja ugyanabban a függvénydefinícióban:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3
were given
>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments
passed as keyword arguments: 'pos_only'
```

Végül nézze meg ezt a függvénydefiníciót, amely potenciális ütközést rejt magában a pozíciós `name` argumentum és a `**kwds` között, amelynek kulcsa a `name`:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Nincs olyan hívás, amely `True`-t adna vissza, mivel a `'name'` kulcsszó mindig az első paraméterhez kötődik. Például:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

A `/` (csak-pozíció argumentumok) használatával azonban lehetséges, mivel lehetővé teszi a nevet pozíció argumentumként és a `'name'`-et kulcsként a kulcsszó argumentumában:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

Más szóval, a csak pozíciót meghatározó paraméterek nevei `**kwds`-ben kétértelműség nélkül használhatók.

Összefoglalás

A használati eset határozza meg, hogy mely paramétereket kell használni a függvénydefinícióban:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Útmutatóként:

- Használja a csak-pozíciós beállítást, ha azt szeretné, hogy a paraméterek neve ne legyen elérhető a felhasználó számára. Ez akkor hasznos, ha a paraméterneveknek nincs valódi jelentése, ha a függvény meghívásakor érvényesíteni kívánja az argumentumok sorrendjét, vagy ha néhány pozícionált paramétert és tetszőleges kulcsszavakat kell venni.
- Akkor használja a csak-kulcsszót, ha a neveknek van jelentése, és a függvénydefiníció jobban érthető, mivel a nevek kifejezik, vagy meg szeretné akadályozni, hogy a felhasználók az átadott argumentum pozíciójára hagyatkozzanak.
- API-k esetén használja a csak-pozíciós beállítást, hogy megakadályozza az API-módosítások megszakítását, ha a paraméter neve a jövőben módosul.

4.8.4 Tetszőleges argumentum-listák

Végül a legritkábban használt lehetőség annak megadása, hogy egy függvény tetszőleges számú argumentummal hívható meg. Ezek az argumentumok egy sorba lesznek csomagolva (lásd: *Tuple-k és szekvenciák*). A változó számú argumentum előtt nulla vagy több normál argumentum fordulhat elő.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Általában ezek a *variadic* argumentumok az utolsók a formális paraméterek listájában, mert a függvénynek átadott összes többi bemeneti argumentumot begyűjtik. Az `*args` paraméter után előforduló formális paraméterek „csak kulcsszó” argumentumok, ami azt jelenti, hogy csak kulcsszóként használhatók pozíció argumentumok helyett.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.8.5 Az argumentumlisták kicsomagolása

A fordított helyzet akkor fordul elő, ha az argumentumok már egy listában vagy sorban vannak, de ki kell őket csomagolni egy külön pozíciós argumentumokat igénylő függvényhíváshoz. Például a beépí-

tett `range()` függvény külön start és stop argumentumokat vár. Ha külön nem érhetők el, írja be a függvényhívást a `*`-operátorral, hogy kicsomagolja az argumentumokat egy listából vagy sorból:

```
>>> list(range(3, 6)) # normál hívás külön argumentumokkal
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args)) # listából kicsomagolt argumentumokkal hívja meg
[3, 4, 5]
```

Ugyanígy a szótárak kulcsszó-argumentumokat szolgáltathatnak a `**`-operátorral:

```
>>> def parrot(voltage, state='merev', action='dörmögne'):
...     print("-- Ez a papagáj", action, end=' ')
...     print("ha", voltage, "voltot vezetne át rajta.", end=' ')
...     print("Ez", state, "!")
...
>>> d = {"voltage": "négy millió", "state": "vérzése megszűnt",
"action": "HÚÚZNA"}
>>> parrot(**d)
-- Ez a papagáj nem HÚÚZNA ha négy millió voltot vezetne át rajta.
Ez vérzése megszűnt !
```

4.8.6 Lambda-kifejezések

A `lambda` kulcsszóval kis névtelen függvények hozhatók létre. Ez a függvény két argumentuma összegét adja vissza: `lambda a, b: a+b`. A lambda függvények mindenhol használhatók, ahol függvényobjektumokra van szükség. Szintaktikailag egyetlen kifejezésre korlátozódnak. Szemantikailag ez csak szintaktikai hab a tortán a normál függvénydefinícióhoz. A beágyazott függvénydefiníciókhoz hasonlóan a lambda függvények is hivatkozhatnak változókra a hatókörből:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

A fenti példa lambda kifejezést használ egy függvény visszaadására. Egy másik felhasználási lehetőség egy kis függvény átadása argumentumként:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.8.7 Dokumentációs karakterláncok

Íme néhány konvenció a dokumentációs karakterláncok tartalmára és formázására vonatkozóan.

Az első sor mindig az objektum céljának rövid, tömör összefoglalása legyen. A rövidség kedvéért nem szabad kifejezetten megadnia az objektum nevét vagy típusát, mivel ezek más módon is elérhetők (kivéve, ha a név történetesen egy függvény működését leíró ige). Ennek a sornak nagybetűvel kell kezdődnie, és ponttal kell végződnie.

Ha több sor is van a dokumentációs karakterláncban, a második sornak üresnek kell lennie, vizuálisan elválasztva az összefoglalót a leírás többi részétől. A következő soroknak egy vagy több bekezdésnek kell lenniük, amelyek leírják az objektum hívási konvencióit, mellékhatásait stb.

A Python elemző nem vonja le a behúzást a többsoros karakterlánc-literálokból Pythonban, így a dokumentációt feldolgozó eszközöknek szükség esetén le kell vonniuk a behúzást. Ez a következő konvenció szerint történik. A karakterlánc első sora utáni első nem üres sor határozza meg a teljes dokumentációs karakterlánc behúzásának mértékét. (Az első sort nem használhatjuk, mivel általában szomszédos a karakterlánc nyitó idézőjeleivel, így a behúzása nem látszik a karakterlánc literáljában.) Az ezzel a behúzással „egyenértékű” szóközt ezután a karakterlánc összes sorának elejétől megfosztjuk. A kevésbé behúzott sorok nem fordulhatnak elő, de ha előfordulnak, az összes kezdő szóközt le kell vonni. A szóközök egyenértékűségét a tabulátorok kiterjesztése után kell tesztelni (általában 8 szóközig).

Íme egy példa egy többsoros dokumentum karakterláncra:

```
>>> def my_function():
...     """Ne csinálj semmit, csak dokumentáld.
...
...     Nem, tényleg nem csinál semmit.
...     """
...     pass
...
>>> print(my_function.__doc__)
Ne csinálj semmit, csak dokumentáld.

    Nem, tényleg nem csinál semmit.
```

4.8.8 Függvény szövegmagyarázatok

A függvény szövegmagyarázatok teljesen opcionális metaadat-információk a felhasználó által definiált függvények által használt típusokról (további információkért lásd a PEP 3107 és a PEP 484 dokumentumot).

A szövegmagyarázatok a függvény __annotations__ attribútumában tárolódnak szótárként, és nincs hatással a függvény más részére. A paraméter szövegmagyarázatokat kettőspont határozza meg a paraméter neve után, majd egy kifejezés, amely a megjegyzés értékét kiértékeli. A visszatérési megjegyzéseket a paraméterlista és a `def` utasítás végét jelző kettőspont között egy literál `->`, majd egy

kifejezés határozza meg. A következő példa tartalmaz egy kötelező argumentumot, egy opcionális argumentumot és a visszatérési értéket megjegyzésekkel ellátva:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs':
<class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.9 Intermezzo: Kódolási stílus

Most, hogy hosszabb, bonyolultabb Python-darabokat készül írni, itt az ideje, hogy beszéljünk a *kódolási stílusról*. A legtöbb nyelv különböző stílusokban írható (vagy tömörebben, formázhatóan); egyesek olvashatóbbak, mint mások. Mindig jó ötlet, hogy mások is könnyebben elolvashassák a kódot, és egy szép kódolási stílus rendkívül sokat segít ebben.

A Python esetében a PEP 8 lett az a stílus útmutató, amelyet a legtöbb projekt betart; nagyon olvasható és szemnek tetsző kódolási stílust hirdet. Minden Python-fejlesztőnek el kell olvasnia valamikor; íme a legfontosabb szempontok, amelyeket az Ön számára kigyűjtöttünk:

- Használjon 4 szóközű behúzást, és ne tabulátort.

  A 4 szóköz jó kompromisszum a kis behúzás (nagyobb beágyazási mélységet tesz lehetővé) és a nagy behúzás (könnyebben olvasható) között. A tabulátorok zavart okoznak, és a legjobb, ha kihagyják őket.

- Tördelje a sorokat úgy, hogy ne haladja meg a 79 karaktert.

  Ez segíti a kis kijelzőket használó felhasználókat, és lehetővé teszi több kódfájl egymás melletti elhelyezését a nagyobb kijelzőkön.

- Használjon üres sorokat a függvények és osztályok elválasztására, illetve a függvényeken belüli nagyobb kódblokkokat.

- Ha lehetséges, helyezzen el a megjegyzéseket a saját sorában.

- Használj doksztringeket.

- Használjon szóközt az operátorok körül és a vessző után, de ne közvetlenül a zárójeles konstrukciókban: `a = f(1, 2) + g(3, 4)`.

- Következetesen nevezze el osztályait és függvényeit; a konvenció az, hogy az `UpperCamelCase`-t használjuk az osztályokhoz és a `kisbetuk_alahuzasjelekkel`-t

a függvényekhez és metódusokhoz. Mindig használja az `self`-et az első metódus argumentum neveként (az osztályokról és metódusokról bővebben lásd: *Az osztályok első pillantásra*).

- Ne használjon divatos kódolásokat, ha a kódot nemzetközi környezetben való használatra szánják. A Python-ban alapértelmezett UTF-8, vagy akár a sima ASCII minden esetben a legjobban működik.
- Hasonlóképpen, ne használjon nem-ASCII karaktereket az azonosítókban, ha csak a legkisebb esélye van annak, hogy más nyelvet beszélők elolvashatják vagy használhatják a kódot.

5. Adatstruktúrák

Ez a fejezet részletesebben leír néhány dolgot, amiről már tanult, és néhány új dolgot is hozzáad.

5.1 Bővebben a listákról

A lista adattípushoz több metódus tartozik. Íme a listaobjektumok összes metódusa:

`list.`**`append`**`(`*`x`*`)`

Adjon hozzá egy elemet a lista végéhez. Egyenértékű: `a[len(a):] = [x]`.

`list.`**`extend`**`(`*`iterable`*`)`

Bővítse ki a listát az `iterable` összes elemének hozzáfűzésével. Egyenértékű: `a[len(a):] = iterable`.

`list.`**`insert`**`(`*`i, x`*`)`

Elem beszúrása egy adott helyre. Az első argumentum a beszúrandó elem indexe, amely elé be kell szúrni, tehát az `a.insert(0, x)` a lista elejére szúr be, az `a.insert(len(a), x)` pedig egyenértékű az `a.append(x)`-szel.

`list.`**`remove`**`(`*`x`*`)`

Távolítsa el az első elemet a listából, amelynek értéke egyenlő *x*-szel. `ValueError` lép fel, ha nincs ilyen elem.

`list.`**`pop`**`(`**`[`**`i`**`]`**`)`

Távolítsa el a listából az adott helyen lévő tételt, és adja vissza. Ha nincs megadva index, az `a.pop()` eltávolítja és visszaadja a lista utolsó elemét. (A metódus leírásában az *i* körüli szögletes zárójelek azt jelzik, hogy a paraméter nem kötelező, nem pedig azt, hogy az adott helyen szögletes zárójelet kell beírnia. Ezt a jelölést gyakran látni fogja a Python könyvtár referenciában.)

`list.`**`clear`**`()`

Eltávolítja az összes elemet a listából. Egyenértékű a `del a[:]`-val.

`list.`**`index`**`(`*`x`*`[, `*`start`*`[, `*`end`*`]]`**`)`

Nulla alapú indexét adja vissza az első olyan elemnek listájában, amelynek értéke egyenlő *x*-szel. `ValueError` lép fel, ha nincs ilyen elem. Az opcionális *start* és *end* argumentumok értelmezése a szelet jelölése szerint történik, és a keresést a lista egy bizonyos részsorozatára korlátozza. A visszaadott indexet a rendszer a teljes sorozat elejéhez viszonyítva számítja ki, nem pedig a *start* argumentumhoz.

`list.`**`count`**`(x)`

Visszaadja, hogy x hányszor jelenik meg a listában.

`list.`**`sort`**`(*, key=None, reverse=False)`

A lista elemeinek rendezése helyben (az argumentumok a rendezés testre szabásához használhatók, magyarázatukért lásd `sorted()`)

`list.`**`reverse`**`()`

Megfordítja a lista elemeit helyben.

`list.`**`copy`**`()`

Küldje vissza a lista sekély példányát. Egyenértékű: `a[:]`.

Egy példa, amely a legtöbb listametódust használja:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Keresse meg a következő banana-t a 4. pozíciótól
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Lehet, hogy észrevette, hogy az olyan módszerek, mint az `insert`, `remove` vagy `sort`, amelyek csak módosítják a listát, nem nyomtatnak visszatérési értéket – az alapértelmezett `None` értéket adják vissza. Ez egy tervezési elv a Python összes változtatható adatszerkezetéhez.

Egy másik dolog, amit észrevehet, hogy nem minden adatot lehet rendezni vagy összehasonlítani. Például a `[Nincs, 'hello', 10]` nem rendeződik, mert az egész számokat nem lehet összehasonlítani karakterláncokkal, és a `None` nem hasonlítható össze más típusokkal. Ezenkívül vannak olyan típusok, amelyeknek nincs meghatározott rendezési kapcsolata. Például a `3+4j < 5+7j` nem érvényes összehasonlítás.

5.1.1 Listák használata veremként

A lista metódusok nagyon egyszerűvé teszik a lista veremként való használatát, ahol az utolsóként hozzáadott elem az első visszakeresett elem („utolsónak-be, elsőnek-ki”). Ha egy elemet szeretne hozzáadni a verem tetejéhez, használja az `append()` parancsot. Egy elem lekéréséhez a verem tetejéről használja a `pop()` parancsot kifejezett index nélkül. Például:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Listák használata sorként

Lehetőség van arra is, hogy egy listát sorként használjunk, ahol az első hozzáadott elem az elsőként letöltött elem („elsőnek-be, elsőnek-ki”); a listák azonban nem hatékonyak erre a célra. Míg a lista végéről történő hozzáfűzések és felugró műveletek gyorsak, a lista elejéről történő beszúrások vagy előugró lépések lassúak (mivel az összes többi elemet eggyel el kell tolni).

A sor megvalósításához használja a `collections.deque`-t, amelyet úgy terveztek, hogy mindkét végéről gyors hozzáfűzéseket és kivevéseket biztosítson. Például:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")   # Terry megérkezik
>>> queue.append("Graham")   # Graham megérkezik
>>> queue.popleft()   # Az első érkező most távozik
'Eric'
>>> queue.popleft()   # A második érkező most távozik
'John'
>>> queue   # A fennmaradó sor érkezési sorrendben
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Lista leképezések

A lista leképezések tömör módot nyújtanak a listák létrehozására. A gyakori alkalmazások új listák készítése, ahol minden elem egy másik sorozat minden egyes tagjára alkalmazott művelet eredménye, vagy iterálható, vagy ezekből az elemekből olyan részsorozatot hoznak létre, amely megfelel egy bizonyos feltételnek.

Például, a négyzetszámok listáját szeretnénk létrehozni:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Vegye figyelembe, hogy ez létrehoz (vagy felülír) egy x nevű változót, amely a ciklus befejezése után is létezik. Kiszámíthatjuk a négyzetek listáját mellékhatások nélkül a következő használatával:

```
squares = list(map(lambda x: x**2, range(10)))
```

vagy amelyik ezzel egyenértékű:

```
squares = [x**2 for x in range(10)]
```

amely tömörebb és olvashatóbb.

A lista értelmezése zárójelekből áll, amelyek egy kifejezést tartalmaznak, amelyet egy `for` záradék követ, majd nulla vagy több a vagy `if` záradékot. Az eredmény egy új lista lesz, amely a kifejezés kiértékeléséből származik az utána következő `for` és `if` záradékkal összefüggésben. Például ez a listcomp kombinálja két lista elemeit, ha azok nem egyenlőek:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

és ezzel egyenértékű:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Figyelje meg, hogy a `for` és az `if` utasítások sorrendje megegyezik mindkét részletben.

Ha a kifejezés egy tuple (pl. az `(x, y)` az előző példában), akkor zárójelbe kell tenni.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # létrehoz egy új listát az értékek megkétszerezésével
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # szűrje a listát a negatív számok kizárásához
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # minden elemre alkalmaz egy függvényt
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # hívjon egy metódust minden elemre
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # hozzon létre egy két tuple-s listát, mint (szám, négyzet)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # a tuple-t zárójelbe kell tenni, különben hiba lép fel
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension
target?
>>> # egy listcomp lista „lapítása” két 'for' használatával
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A listaértelmezések összetett kifejezéseket és beágyazott függvényeket tartalmazhatnak:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 Beágyazott lista leképezések

A lista leképezés kezdő kifejezése tetszőleges kifejezés lehet, beleértve egy másik lista leképezést is.

Tekintsük a következő példának egy 3x4-es mátrixra, amely 3 elemű lista amely 4 hosszúságú listából van megvalósítva:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

A következő lista leképezés transzponálja a sorokat és oszlopokat:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Amint az előző részben láttuk, a belső lista megértését az azt követő szövegkörnyezetben értékeljük, így ez a példa egyenértékű a következővel:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

ami viszont ugyanaz, mint:

```
>>> transposed = []
>>> for i in range(4):
...     # a következő 3 sor valósítja meg a beágyazott listcomp-ot
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

A való világban előnyben kell részesíteni a beépített függvényeket az összetett folyamatutasításokkal szemben. A `zip()` függvény nagyszerű munkát végezne ebben a használati esetben:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Az ebben a sorban lévő csillaggal kapcsolatos részletekért lásd: Az argumentumlisták kicsomagolása.

5.2 A `del` utasítás

Létezik egy módja annak, hogy egy elemet eltávolítsunk a listából, ha az értéke helyett az indexe adja meg: a `del` utasítás. Ez eltér a `pop()` metódustól, amely értéket ad vissza. A `del` utasítás használható szeletek eltávolítására a listából vagy a teljes lista törlésére (amit korábban úgy tettünk, hogy egy üres listát rendeltünk a szelethez). Például:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
```

```
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

A del teljes változók törlésére is használható:

```
>>> del a
```

Az `a` névre való hivatkozás a továbbiakban hiba (legalábbis addig, amíg más értéket nem rendelnek hozzá). Később találunk más felhasználási módokat a `del` számára.

5.3 Tuplék és szekvenciák

Láttuk, hogy a listáknak és karakterláncoknak számos közös tulajdonságuk van, például indexelési és szeletelési műveletek. Ez két példa a szekvencia adattípusokra (lásd typeseq). Mivel a Python egy fejlődő nyelv, más szekvenciális adattípusok is hozzáadhatók. Van egy másik szabványos szekvenciális adattípus is: a tuple.

A tuple számos, vesszővel elválasztott értékből áll, például:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # A tuplék egymásba ágyazhatók:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # A tuplék megváltoztathatatlanok:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # de tartalmazhatnak változtatható objektumokat:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Amint látja, a kimeneten a tuplék mindig zárójelben vannak, így a beágyazott tuplék helyesen értelmeződnek; beírhatók környező zárójelekkel vagy anélkül, bár gyakran a zárójelek mindenképpen szükségesek (ha a tuple egy nagyobb kifejezés része). Egy tuple egyes elemeihez nem lehet hozzárendelni, de lehetséges olyan sorokat létrehozni, amelyek változtatható objektumokat, például listákat tartalmaznak.

Bár a tuplék hasonlónak tűnhetnek a listákhoz, gyakran használják őket különböző helyzetekben és más célokra. A tuplék megváltoztathatatlanok, és általában heterogén elemek szekvenciáját tartalmazzák, amelyekhez kicsomagolással (lásd később ebben a részben) vagy indexeléssel (vagy nevesített tuplék esetén attribútumokkal) lehet hozzáférni. A listák változtathatók, elemeik általában homogének, és a lista iterálásával érhetők el.

Speciális probléma a 0 vagy 1 elemet tartalmazó sorok felépítése: a szintaxisnak van néhány extra furcsasága ezekhez. Az üres sorokat üres zárójelpár alkotja; egy elemet tartalmazó sor egy érték vesszővel történő követésével jön létre (nem elegendő egyetlen értéket zárójelbe tenni). Csúnya, de hatásos. Például:

```
>>> empty = ()
>>> singleton = 'hello',  # <-- figyeld a megjegyzést záró vesszőt
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

A `t = 12345, 54321, 'helló!'` utasítás egy példa a tuple csomagolásra: az `12345`, `54321` és a `'hello!'` érték egy tuple-ba vannak összecsomagolva. Fordított művelet is lehetséges:

```
>>> x, y, z = t
```

Ezt a megfelelő módon szekvencia kicsomagolásnak nevezik, és a jobb oldalon lévő bármely szekvencia esetén működik. A szekvencia kicsomagolása megköveteli, hogy az egyenlőségjel bal oldalán annyi változó legyen, ahány elem van a sorozatban. Vegye figyelembe, hogy a többszörös hozzárendelés valójában csak a tuple becsomagolás és a szekvencia kicsomagolás kombinációja.

5.4 Halmazok

A Python tartalmaz egy adattípust a halmazokhoz is. A halmaz egy rendezetlen gyűjtemény, amely nem tartalmaz ismétlődő elemeket. Az alapvető felhasználási területek közé tartozik a tagság tesztelése és az ismétlődő bejegyzések kiküszöbölése. A halmazobjektumok olyan matematikai műveleteket is támogatnak, mint az egyesítés, metszéspont, különbség és szimmetrikus különbség.

A kapcsos zárójelek vagy a `set()` függvény használható halmazok létrehozására. Megjegyzés: üres halmaz létrehozásához a `set()`-et kell használni, nem a `{}`; ez utóbbi létrehoz egy üres szótárat, egy adatstruktúrát, amelyet a következő részben tárgyalunk.

Íme egy rövid bemutató:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)  # azt mutatják, hogy az ismétlődéseket eltávolították
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket  # gyors tagsági tesztelés
True
```

```
>>> 'crabgrass' in basket
False
>>> # Két szóból származó egyedi betűk halmazműveleteinek bemutatása
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # a egyedi betűi
{'a', 'r', 'b', 'c', 'd'}
>>> a - b  # betűk az a-ban, de nem a b-ben
{'r', 'd', 'b'}
>>> a | b  # betűk a-ban vagy b-ben, vagy mindkettőben
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b  # betűk mindkettőben: a and b
{'a', 'c'}
>>> a ^ b  # betűk a-ban vagy b-ben, de nem mindkettőben
{'r', 'd', 'b', 'm', 'z', 'l'}
```

A lista leképezésekhez hasonlóan a halmaz leképezések is támogatottak:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5 Szótárak

A Pythonba beépített másik hasznos adattípus a szótár (lásd a típusleképezést). A szótárakat más nyelveken néha „asszociatív memóriaként” vagy „asszociatív tömbként” találhatjuk meg. Ellentétben a szekvenciákkal, amelyeket egy számtartomány indexel, a szótárakat kulcsok indexelik, amelyek bármilyen megváltoztathatatlan típusúak lehetnek; karakterláncok és számok mindig lehetnek kulcsok. A tuplék akkor használhatók kulcsként, ha csak karakterláncokat, számokat vagy tuplékat tartalmaznak; Ha egy tuple közvetlenül vagy közvetve bármilyen módosítható objektumot tartalmaz, akkor nem használható kulcsként. A listák nem használhatók kulcsként, mivel a listák a helyükön módosíthatók index- hozzárendelésekkel, szelet-hozzárendelésekkel vagy olyan metódusokkal, mint az `append()` és az `extend()`.

A legjobb, ha a szótárat kulcs: értékpárok halmazának tekintjük, azzal a feltétellel, hogy a kulcsok egyediek (egy szótáron belül). A kapcsos zárójelpár egy üres szótárt hoz létre: `{}`. Ha a kulcs:érték párok vesszővel elválasztott listáját helyezi el a kapcsos zárójelek között, a kezdeti kulcs:érték párok hozzáadódnak a szótárhoz; így írják a szótárakat a kimenetre is.

A szótár fő műveletei az értékek tárolása valamilyen kulccsal és a kulcs által adott érték kinyerése. Lehetőség van egy kulcs:érték pár törlésére is a `del`-lel. Ha egy már használatban lévő kulccsal tárolja, akkor a kulcshoz tartozó régi érték elfelejtődik. Hiba az értéket egy nem létező kulccsal kinyerni.

A `lista(d)` végrehajtása egy szótárban visszaadja a szótárban használt összes kulcs listáját, beillesztési sorrendben (ha rendezni szeretné, használja helyette a `sorted(d)`-t). Ha ellenőrizni szeretné, hogy egyetlen kulcs van-e a szótárban, használja az `in` kulcsszót.

Íme egy kis példa egy szótár használatára:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

A `dict()` konstruktor közvetlenül a kulcs-érték párok sorozataiból készít szótárakat:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Ezenkívül a dict leképezésekkel szótárakat hozhatunk létre tetszőleges kulcs- és érték kifejezésekből:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Ha a kulcsok egyszerű karakterláncok, néha egyszerűbb kulcsszó-argumentumok használatával párokat megadni:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Bejárási technikák

A szótári elemek bejárásakor a kulcs és a hozzá tartozó érték egyidejűleg lekérhető az `items()` metódussal.

```
>>> knights = {'gallahad': 'a tiszta', 'robin': 'a bátor'}
>>> for k, v in knights.items():
... print(k, v)
...
gallahad a tiszta
robin a bátor
```

Egy szekvencia bejárásakor a pozícióindex és a hozzá tartozó érték egyidejűleg lekérhető az `enumerate()` függvény segítségével.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Két vagy több szekvencia egyidejű bejárásához a bejegyzések párosíthatók a `zip()` függvénnyel.

```
>>> questions = ['neve', 'küldetése', 'kedvenc színe']
>>> answers = ['Lancelot', 'a Szent Grál', 'kék']
>>> for q, a in zip(questions, answers):
...     print('Mi az ön {0}? Ez a {1}.'.format(q, a))
...
Mi az ön neve? Ez a Lancelot.
Mi az ön küldetése? Ez a Szent Grál.
Mi az ön kedvenc színe? Ez a kék.
```

Ha egy szekvenciát visszafelé szeretne bejárni, először adja meg a sorozatot előrefelé, majd hívja meg a `reversed()` függvényt.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Ha egy szekvenciát rendezett sorrendben szeretne átvinni, használja a `sorted()` függvényt, amely egy új rendezett listát ad vissza, miközben a forrást változatlanul hagyja.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

A `set()` használata egy szekvencián kiküszöböli az ismétlődő elemeket. A `sorted()` használata kombinációban a `set()`-tel egy szekvencia felett, egy sajátos módja a sorozat egyedi elemeinek rendezési sorrendben történő bejárásának.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
... print(f)
...
apple
banana
orange
pear
```

Néha csábító a lista módosítása, miközben bejárjuk; azonban gyakran egyszerűbb és biztonságosabb egy új lista létrehozása.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 További információk a feltételekről

A `while` és `if` utasításokban használt feltételek bármilyen operátort tartalmazhatnak, nem csak összehasonlításokat.

Az `in` és `not in` összehasonlító operátorok tagsági tesztek, amelyek meghatározzák, hogy egy érték egy tárolóban van-e (vagy nincs benne). Az operátorok összehasonlítják és nem azt, hogy két objektum valóban ugyanaz-e az objektum. Minden összehasonlító operátornak azonos prioritása van, ami alacsonyabb, mint az összes numerikus operátoré.

Az összehasonlítások láncolhatók. Például `a < b == c` azt teszteli, hogy `a` kisebb-e, mint `b`, és ráadásul `b` egyenlő-e `c`-vel.

Az összehasonlítások kombinálhatók az `and` és az `or` Boole-operátorok használatával, és az összehasonlítás (vagy bármely más logikai kifejezés) eredménye tagadható a `not`-tal. Ezek alacsonyabb prioritásúak, mint az összehasonlító operátorok; közöttük a `not` a legmagasabb prioritású és az `or` a legalacsonyabb, így `A and not B or C` egyenértékű `(A and (not B)) or C`-vel. Mint mindig, a zárójelek is használhatók a kívánt összetétel kifejezésére.

A Boole-operátorok az `and` és az `or` úgynevezett rövidzárlati operátorok: argumentumaik balról jobbra kerülnek kiértékelésre, és a kiértékelés leáll, amint az eredmény megállapításra kerül. Például, ha A és C igaz, de B hamis, akkor `A and B and C` nem értékeli ki a C kifejezést. Ha általános értékként és nem logikai értékként használjuk, a rövidzárlati operátor visszatérési értéke az utolsó kiértékelt argumentum.

Lehetőség van egy összehasonlítás vagy más logikai kifejezés eredményét hozzárendelni egy változóhoz. Például,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Ne feledje, hogy Pythonban, a C-vel ellentétben a kifejezések hozzárendelését kifejezetten a rozmár operátorral kell végrehajtani :=. Ezzel elkerülhető a C programokban előforduló gyakori problémák: az = beírása egy kifejezésben, amikor a == volt a cél.

5.8 Szekvenciák és egyéb típusok összehasonlítása

A szekvenciális objektumok általában összehasonlíthatók más, azonos szekvencia típusú objektumokkal. Az összehasonlítás lexikográfiai sorrendet alkalmaz: először az első két elemet hasonlítjuk össze, és ha eltérnek, az határozza meg az összehasonlítás eredményét; ha egyenlők, a következő két elemet összehasonlítja, és így tovább, amíg bármelyik szekvencia ki nem merül. Ha két összehasonlítandó elem maga is azonos típusú szekvencia, akkor a lexikográfiai összehasonlítás rekurzív módon történik. Ha két szekvencia minden eleme egyenlő, akkor a szekvenciákat egyenlőnek tekintjük. Ha az egyik szekvencia a másik kezdeti rész szekvenciája, akkor a rövidebb szekvencia a kisebb. A karakterláncok lexikográfiai rendezése a Unicode kód pontszámát használja az egyes karakterek rendezéséhez. Néhány példa az azonos típusú szekvenciák összehasonlítására:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

A szekvencia-objektumok általában összehasonlíthatók más objektumokkal. Vegye figyelembe, hogy a különböző típusú objektumok < vagy > jelekkel való összehasonlítása legális, feltéve, hogy az objektumok megfelelő összehasonlítási módszerekkel rendelkeznek. Például a vegyes numerikus típusokat numerikus értékük alapján hasonlítja össze, így a 0 egyenlő 0,0-val stb. Ellenkező esetben az értelmező ahelyett, hogy tetszőleges sorrendet adna, TypeError kivételt vet fel.

6. Modulok

Ha kilép a Python értelmezőből, és újra belép, a megadott definíciók (függvények és változók) elvesznek. Ezért, ha valamivel hosszabb programot akarunk írni, jobb, ha egy szövegszerkesztővel készítjük elő a bemenetet az értelmező számára, és ehelyett azzal a fájllal futtatjuk bemenetként. Ezt script létrehozásának nevezik. Ahogy a program hosszabb lesz, érdemes lehet több fájlra felosztani a könnyebb karbantartás érdekében. Használhat egy praktikus függvényt is, amelyet több programban írt meg anélkül, hogy a definícióját az egyes programokba másolta volna.

Ennek támogatására a Python lehetőséget kínál arra, hogy definíciókat helyezzen el egy fájlba, és használja őket egy szkriptben vagy az értelmező interaktív példányában. Az ilyen fájlt modulnak nevezzük; egy modulból származó definíciók importálhatók más modulokba vagy a fő modulba (a változók gyűjteménye, amelyhez a legfelső szinten és számológép módban végrehajtott szkriptben hozzáférhet).

A modul egy Python-definíciókat és utasításokat tartalmazó fájl. A fájlnév a modul neve, hozzáfűzve a .py utótag. A modulon belül a modul neve (karakterláncként) elérhető a __name__ globális változó értékeként. Például kedvenc szövegszerkesztőjével hozzon létre egy fibo.py nevű fájlt az aktuális könyvtárban a következő tartalommal:

```
# Fibonacci számok modul

def fib(n):  # Fibonacci sorozatot írunk n-ig
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):  # visszaadja a Fibonacci sorozatot n-ig
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Most lépjen be a Python interpreterbe, és importálja ezt a modult a következő paranccsal:

```
>>> import fibo
```

Ez nem adja hozzá a `fibo`-ban definiált függvények neveit közvetlenül az aktuális névtérhez (további részletekért lásd a Python-hatóköröket és névtereket); csak a `fibo` modulnevet adja oda. A modul nevével a következő függvényeket érheti el:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Ha gyakran kíván használni egy függvényt, hozzárendelheti egy helyi névhez:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 További információ a modulokról

Egy modul tartalmazhat végrehajtható utasításokat, valamint függvénydefiníciókat. Ezek az utasítások a modul inicializálására szolgálnak. Csak akkor futnak le, amikor először találkozik a modulnév egy import utasításban. (Ezek akkor is futnak, ha a fájl szkriptként fut.)

Valójában a függvénydefiníciók is „utasítások”, amelyeket „végre kell hajtani”; egy modulszintű függvénydefiníció végrehajtása hozzáadja a függvény nevét a modul globális névteréhez.

Minden modulnak megvan a saját privát névtere, amelyet a modulban meghatározott összes függvény globális névtérként használ. Így a modul szerzője használhat globális változókat a modulban anélkül, hogy aggódnia kellene a felhasználó globális változóival való véletlen ütközés miatt. Másrészt, ha tudja, hogy mit csinál, megérintheti a modul globális változóit ugyanazzal a jelöléssel, amelyet a funkcióira használnak, `modulnév.tárgynév`.

A modulok más modulokat is importálhatnak. Szokásos, de nem kötelező minden `import` utasítást a modul (vagy szkript) elejére helyezni. Az importált modulnevek, ha a modul legfelső szintjén vannak (a függvényeken vagy osztályokon kívül), hozzáadódnak a modul globális névteréhez.

Az `import` utasításnak van egy olyan változata, amely a neveket egy modulból közvetlenül az importáló modul névterébe importálja. Például:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ez nem vezeti be a modul nevét, amelyből az importálás történik a helyi névtérben (tehát a példában a fibo nincs definiálva).

Még egy változat létezik a modul által meghatározott összes név importálására:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ez az összes nevet importálja, kivéve azokat, amelyek aláhúzás jellel (_) kezdődnek. A legtöbb esetben a Python programozók nem használják ezt a lehetőséget, mivel ismeretlen névkészletet vezet be az értelmezőbe, esetleg elrejt néhány, már definiált dolgot.

Vegye figyelembe, hogy a `*` modulból vagy csomagból történő importálása általában elítélendő, mivel gyakran rosszul olvasható kódot eredményez. Az interaktív munkamenetek gépelésének mentésére azonban megfelelő.

Ha a modul nevét `as` követi, akkor az `as` utáni név közvetlenül az importált modulhoz lesz kötve.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ez gyakorlatilag ugyanúgy importálja a modult, mint az `import fibo`, azzal az egyetlen különbséggel, hogy `fib`-ként érhető el.

Hasonló hatású felhasználás a `from` esetén is használható:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Megjegyzés: Hatékonysági okokból minden egyes modult csak egyszer importálunk interpreter munkamenetenként. Ezért, ha módosítja a moduljait, újra kell indítania az értelmezőt – vagy ha csak egy modult szeretne interaktívan tesztelni, használja az `importlib.reload()` parancsot, pl. `import importlib; importlib.reload(modulnév)`.

6.1.1 Modulok végrehajtása szkriptként

Amikor egy Python modult futtat a

```
python fibo.py <arguments>
```

a modulban lévő kód ugyanúgy végrehajtásra kerül, mintha importáltad volna, de a `__name__` értéke `"__main__"`. Ez azt jelenti, hogy a kód hozzáadásával a modul végéhez:

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

a fájlt szkriptként és importálható modulként is használhatóvá teheti, mert a parancssort elemző kód csak akkor fut le, ha a modul „fő” („main”) fájlként fut:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Ha a modul importálva van, a kód nem fut le:

```
>>> import fibo
>>>
```

Ezt gyakran használják egy modul kényelmes felhasználói felületének biztosítására, vagy tesztelési célokra (a modul szkriptként való futtatása egy tesztcsomagot hajt végre).

6.1.2 A modul keresési útvonala

Amikor egy `spam` nevű modult importál, az interpreter először keres egy beépített modult ezzel a névvel. Ezeket a modulneveket a `sys.builtin_module_names` tartalmazza. Ha nem található, akkor megkeresi a `spam.py` nevű fájlt a `sys.path` változó által megadott könyvtárlistában. A `sys.path` a következő helyekről inicializálódik:

- A bemeneti parancsfájlt tartalmazó könyvtár (vagy az aktuális könyvtár, ha nincs megadva fájl).
- PYTHONPATH (könyvtárnevek listája, ugyanazzal a szintaxissal, mint a PATH shell-változó).
- A telepítéstől függő alapértelmezés (megállapodás szerint tartalmazza a `site-packages` könyvtárát, amelyet a `site` modul kezel).

További részletek a sys-path-init oldalon találhatók.

Megjegyzés: Azon fájlrendszereken, amelyek támogatják a szimbolikus hivatkozásokat, a bemeneti parancsfájlt tartalmazó könyvtár a szimbolikus hivatkozás követése után kerül kiszámításra. Más szavakkal, a szimbolikus hivatkozást tartalmazó könyvtár nem kerül hozzáadásra a modul keresési útvonalához.

Az inicializálás után a Python programok módosíthatják a `sys.path`-ot. A futtatott parancsfájlt tartalmazó könyvtár a keresési útvonal elejére kerül, a szabványos könyvtári elérési út elé. Ez azt jelenti, hogy az adott könyvtárban lévő szkriptek lesznek betöltve a könyvtár azonos nevű moduljai helyett. Ez hiba, kivéve, ha a csere szándékos. További információért lásd a Standard modulok című részt.

6.1.3 „Lefordított” Python-fájlok

A modulok betöltésének felgyorsítása érdekében a Python gyorsítótárazza az egyes modulok lefordított verzióit a `__pycache__` könyvtárban `module.`*`version`*`.pyc` néven, ahol a verzió kódolja a lefordított fájl formátumát; általában a Python verziószámát tartalmazza. Például a CPython 3.3-as kiadásában a spam.py lefordított verziója a `__pycache__/spam.cpython-33.pyc` néven kerül gyorsítótárba. Ez az elnevezési konvenció lehetővé teszi, hogy a különböző kiadásokból és a Python különböző verzióiból összeállított modulok együtt létezzenek.

A Python összeveti a forrás módosítási dátumát a lefordított verzióval, hogy megnézze, nem elavult-e, és nem kell-e újrafordítani. Ez egy teljesen automatikus folyamat. Ezenkívül a lefordított modulok

platform-függetlenek, így ugyanaz a könyvtár megosztható a különböző architektúrájú rendszerek között.

A Python két esetben nem ellenőrzi a gyorsítótárat. Először is, mindig újrafordítja, és nem tárolja a közvetlenül a parancssorból betöltött modul eredményét. Másodszor, nem ellenőrzi a gyorsítótárat, ha nincs forrásmodul. A nem-forrás (csak lefordított) terjesztés támogatásához a lefordított modulnak a forráskönyvtárban kell lennie, és nem lehet forrásmodul.

Néhány tipp szakértőknek:

- A Python parancs `-O` vagy `-OO` kapcsolóival csökkentheti a lefordított modul méretét. Az `-O` kapcsoló eltávolítja az assert utasításokat, az `-OO` kapcsoló eltávolítja az assert utasításokat és a __doc__ karakterláncokat is. Mivel egyes programok támaszkodhatnak ezek elérhetőségére, csak akkor használja ezt a lehetőséget, ha tudja, mit csinál. Az „optimalizált” modulok `opt-`címkével rendelkeznek, és általában kisebbek. A jövőbeli kiadások megváltoztathatják az optimalizálás hatásait.

- Egy program nem fut gyorsabban, ha `.pyc` fájlból olvassa be, mint amikor `.py` fájlból olvassa be; az egyetlen dolog, ami gyorsabb a `.pyc` fájlokkal kapcsolatban, az a betöltési sebesség.

- A `compileall` modul képes .pyc fájlokat létrehozni egy könyvtár összes moduljához.

- Erről a folyamatról több részlet található, beleértve a döntések folyamatábráját is, a PEP 3147-ben.

6.2 Szabványos modulok

A Python szabványos modulokból álló könyvtárral érkezik, amelyet egy külön dokumentum, a Python Library Reference (a továbbiakban „Könyvtári hivatkozás”) ismertet. Néhány modul be van építve az interpreterbe; ezek hozzáférést biztosítanak olyan műveletekhez, amelyek nem részei a nyelv magjának, de ennek ellenére be vannak építve, akár a hatékonyság érdekében, akár azért, hogy hozzáférést biztosítsanak az operációs rendszer primitíveihez, például a rendszerhívásokhoz. Az ilyen modulok készlete egy konfigurációs opció, amely az alapul szolgáló platformtól is függ. Például a `winreg` modul csak Windows rendszereken érhető el. Egy bizonyos modul érdemel némi figyelmet: a `sys`, amely minden Python-értelmezésbe be van építve. A `sys.ps1` és `sys.ps2` változók határozzák meg az elsődleges és másodlagos promptként használt karakterláncokat:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Ez a két változó csak akkor van megadva, ha az értelmező interaktív módban van.

A `sys.path` változó a karakterláncok listája, amely meghatározza az értelmező modulok keresési útvonalát. A `PYTHONPATH` környezeti változóból vagy beépített alapértelmezett elérési útra inicializálódik, ha a `PYTHONPATH` nincs beállítva. Módosíthatja szabványos listaműveletekkel:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 A `dir()` függvény

A beépített `dir()` függvény arra szolgál, hogy megtudja, milyen neveket határoz meg egy modul. A karakterláncok rendezett listáját adja vissza:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
'__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
'_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
'_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
'warnoptions']
```

A `dir()` argumentumok nélkül felsorolja a jelenleg definiált neveket:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Vegye figyelembe, hogy minden típusú nevet felsorol: változók, modulok, függvények stb.

A `dir()` nem sorolja fel a beépített függvények és változók nevét. Ha listát szeretne azokról, amelyek a szabványos `builtins` modulban vannak meghatározva:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

6.4 Csomagok

A csomagok a Python modul-névterének strukturálásának egyik módja a „pontozott modulnevek” használatával. Például az `A.B` modulnév egy `B` nevű almodult jelöl egy `A` nevű csomagban. Ahogyan a modulok használata megkíméli a különböző modulok szerzőit attól, hogy egymás globális változónevei miatt aggódjanak, a pontozott modulnevek használata menti meg a szerzőket a több modulból álló csomagok, mint például a NumPy vagy a Pillow, hogy ne kelljen egymás modulnevei miatt aggódniuk.

Tegyük fel, hogy modulgyűjteményt (egy „csomagot”) szeretne tervezni a hangfájlok és hangadatok egységes kezelésére. Számos különböző hangfájl-formátum létezik (amelyeket általában a kiterjesztésükről ismernek fel, például: `.wav`, `.aiff`, `.au`), ezért előfordulhat, hogy a különböző fájlformátumok közötti átalakításhoz modulok növekvő gyűjteményét kell létrehoznia és karbantartania. Ezen kívül számos különböző műveletet érdemes elvégezni a hangadatokon (például keverés, visszhang hozzáadása, hangszínszabályzó funkció alkalmazása, mesterséges sztereó effektus létrehozása), így ezen kívül egy végtelen számú modult kell írnia, hogy megvalósítsa ezeket a műveleteket. Íme egy lehetséges struktúra a csomagodhoz (hierarchikus fájlrendszerben kifejezve):

```
sound/                          Felső szintű csomag
      __init__.py               Inicializálja a hangcsomagot
      formats/                  Alcsomag a fájlformátumok konvertálásához
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Alcsomag a hangeffektusokhoz
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Alcsomag a szűrőknek
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

A csomag importálásakor a Python a `sys.path` könyvtáraiban keresi a csomag alkönyvtárát.

Az `__init__.py` fájlok szükségesek ahhoz, hogy a Python csomagként kezelje a fájlt tartalmazó könyvtárakat. Ez megakadályozza, hogy a közös névvel, például `string`-gel rendelkező könyvtárak véletlenül elrejtsék a modulkeresési útvonalon a később előforduló érvényes modulokat. A legegyszerűbb esetben az `__init__.py` lehet csak egy üres fájl, de végrehajthatja a csomag inicializálási kódját, vagy beállíthatja az `__all__` változót is, amelyet később ismertetünk.

A csomag felhasználói egyedi modulokat importálhatnak a csomagból, például:

```
import sound.effects.echo
```

Ez betölti a `sound.effects.echo` almodult. A teljes nevével kell hivatkozni rá.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Az almodul importálásának másik módja:

```
from sound.effects import echo
```

Ez betölti az almodul `echo`-ját is, és elérhetővé teszi a csomag előtagja nélkül, így a következőképpen használható:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Egy másik változat a kívánt függvény vagy változó közvetlen importálása:

```
from sound.effects.echo import echofilter
```

Ez ismét betölti az `echo` almodult, de így közvetlenül elérhetővé válik az `echofilter()` függvény:

```
echofilter(input, output, delay=0.7, atten=4)
```

Vegye figyelembe, hogy a `from package import item` használatakor az `item` lehet a csomag almodulja (vagy alcsomagja), vagy a csomagban meghatározott más név, például függvény, osztály vagy változó. Az `import` utasítás először azt teszteli, hogy az elem definiálva van-e a csomagban; ha nem, akkor feltételezi, hogy ez egy modul, és megpróbálja betölteni. Ha nem találja meg, egy `ImportError` kivétel jelenik meg.

Ezzel szemben, ha olyan szintaxist használunk, mint az `import item.subitem.subsubitem`, az utolsó kivételével minden item-nek csomagnak kell lennie; az utolsó item lehet modul vagy csomag, de nem lehet az előző elemben meghatározott osztály, függvény vagy változó.

6.4.1 Importálás * csomagból

Most mi történik, ha a felhasználó a `from sound.effects import *`-ot ír? Ideális esetben az ember azt reméli, hogy ez valahogy kikerül a fájlrendszerbe, megkeresi, mely almodulok vannak a csomagban, és mindet importálja. Ez sokáig tarthat, és az almodulok importálása nem kívánt mellékhatásokkal járhat, amelyek csak akkor fordulhatnak elő, ha az almodul kifejezetten importálva van.

Az egyetlen megoldás, ha a csomag szerzője megadja a csomag explicit indexét. Az `import` utasítás a következő konvenciót használja: ha egy csomag `__init__.py` kódja egy `__all__` nevű listát határoz meg, akkor ez azoknak a modulneveknek a listája, amelyeket importálni kell, ha a `from package import *` utasítással találkozik. A csomag szerzőjének feladata, hogy ezt a listát naprakészen tartsa, amikor a csomag új verziója megjelenik. A csomag szerzői dönthetnek úgy is, hogy nem támogatják, ha nem látják hasznát a * csomagból való importálásnak. Például a `sound/effects/__init__.py` fájl a következő kódot tartalmazhatja:

```
__all__ = ["echo", "surround", "reverse"]
```

Ez azt jelenti, hogy a `from sound.effects import *` a `sound.effects` csomag három megnevezett almodulját importálja.

Ügyeljen arra, hogy az almodulokat árnyékolhatják a helyileg meghatározott nevek. Például, ha hozzáadott egy `reverse` függvényt a `sound/effects/__init__.py` fájlhoz, akkor a `from sound.effects import *` csak a két `echo` és `surround` almodult importálja, de a `reverse` almodult nem, mert azt helyileg árnyékolja a definiált `reverse` funkció:

```
__all__ = ["echo",      # az „echo.py” fájlra utal
           "surround",  # a „surround.py” fájlra utal
           "reverse",   # !!! most a "reverse" funkcióra utal !!!
]
```

```
def reverse(msg: str):  # <-- ez a név árnyékolja a 'reverse.py'
    return msg[::-1] # almodult 'from sound.effects import *' esetén
```

Ha az __all__ nincs megadva, a sound.effects import * utasítása nem importálja az összes almodult a sound.effects csomagból az aktuális névtérbe; csak azt biztosítja, hogy a sound.effects csomag importálásra került (esetleg a __init__.py inicializálási kódjának futtatásával), majd importálja a csomagban meghatározott neveket. Ez magában foglalja a __init__.py által meghatározott (és kifejezetten betöltött) almodulokat. Tartalmazza a csomag azon almoduljait is, amelyeket a korábbi importálási utasítások kifejezetten betöltöttek. Fontolja meg ezt a kódot:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Ebben a példában az echo és a surround modulok az aktuális névtérbe kerülnek importálásra, mert a sound.effects csomagban vannak megadva, amikor a from...import utasítás végrehajtódik. (Ez akkor is működik, ha __all__ meg van adva.)

Bár bizonyos modulokat úgy terveztek, hogy csak olyan neveket exportáljanak, amelyek bizonyos sablonokat követnek az import * használatakor, ez továbbra is rossz gyakorlatnak számít a termék kódban.

Ne feledje, nincs semmi baj, ha a from package import specific_submodule-t használja! Valójában ez az ajánlott jelölés, hacsak az importáló modulnak nem kell azonos nevű almodulokat használnia különböző csomagokból.

6.4.2 Csomagon belüli hivatkozások

Ha a csomagok alcsomagokba vannak strukturálva (mint a példában a sound csomag esetében), akkor abszolút importálást használhat a testvércsomagok almoduljaira való hivatkozáshoz. Például, ha a sound.filters.vocoder modulnak a sound.effects csomagban lévő echo modult kell használnia, használhatja a from sound.effects import echo-t.

Relatív importálást is írhat az importálási utasítás from module import name formájával. Ezek az importálások bevezető pontokat használnak a relatív importálásban érintett aktuális és szülőcsomagok jelzésére. A surround modulból például a következőket használhatja:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Vegye figyelembe, hogy a relatív importálások az aktuális modul nevén alapulnak. Mivel a fő modul neve mindig „__main__”, a Python-alkalmazások fő moduljaként való használatra szánt moduloknak mindig abszolút importálást kell használniuk.

6.4.3 Csomagok több könyvtárban

A csomagok még egy speciális attribútumot támogatnak, a `__path__`. Ez egy lista, amely a csomag `__init__.py` fájlját tartalmazó könyvtár nevét tartalmazza, mielőtt a fájlban lévő kód végrehajtásra kerülne. Ez a változó módosítható; ez befolyásolja a csomagban található modulok és alcsomagok jövőbeli keresését.

Bár erre a funkcióra nincs gyakran szükség, a csomagban található modulok készletének bővítésére használható.

7. Input és output

Számos módja van a program kimenetének bemutatására; az adatok ember által olvasható formában nyomtathatók, vagy fájlba írhatók későbbi felhasználás céljából. Ez a fejezet néhány lehetőséget tárgyal meg.

7.1 Fancier kimenet formázás

Eddig az értékek írási módjának kétféle lehetőségével találkoztunk: kifejezési utasításokkal és a `print()` függvénnyel. (A harmadik módszer a fájlobjektumok `write()` metódusa; a szabványos kimeneti fájl `sys.stdout` néven hivatkozhat. Erről további információért lásd a Library Reference részt.)

Gyakran jobban szeretné szabályozni a kimenet formázását, mint egyszerűen szóközzel elválasztott értékeket nyomtatni. Számos módja van a kimenet formázásának.

- Formázott karakterlánc-literálok használatához kezdje a karakterláncot `f` vagy `F` betűvel a nyitó idézőjel vagy a hármas idézőjel előtt. Ebben a karakterláncban egy Python-kifejezést írhat `{` és `}` karakterek közé, amelyek változókra vagy literális értékekre hivatkozhatnak.

```
>>> year = 2016
>>> event = 'népszavazás'
>>> f'A {year}-os {event} eredménye'
'A 2016-os népszavazás eredménye'
```

- A karakterláncok `str.format()` metódusa több kézi erőfeszítést igényel. Továbbra is használhatja a `{` és a `}` karaktereket a változók helyettesítésének helyére, és részletes formázási utasításokat adhat, de meg kell adnia a formázandó információkat is.

```
>>> igen_szavazatok = 42_572_654
>>> nem_szavazatok = 43_132_495
>>> szazalek = igen_szavazatok / (igen_szavazatok + nem_szavazatok)
>>> '{:-9} IGEN szavazat {:2.2%}'.format(igen_szavazatok, szazalek)
' 42572654 IGEN szavazat 49.67%'
```

- Végül az összes karakterlánc-kezelést saját maga is elvégezheti, ha karakterlánc-szeletelési és -összefűzési műveleteket használ, és létrehozhat bármilyen elrendezést, amelyet el tud képzelni. A karakterlánctípusnak vannak olyan metódusai, amelyek hasznos műveleteket hajtanak végre a karakterláncok adott oszlopszélességű kitöltésére.

Ha nincs szüksége díszes kimenetre, de csak néhány változót szeretne gyorsan megjeleníteni hibakeresési célból, akkor a `repr()` vagy `str()` függvényekkel bármilyen értéket karakterláncsá alakíthat.

Az `str()` függvény célja, hogy olyan értékek reprezentációit adja vissza, amelyek korrekten ember által olvashatók, míg a `repr()` olyan reprezentációkat hoz létre, amelyeket az interpreter is be tud

olvasni (vagy `SyntaxError`-t kényszerít ki, ha nincs ekvivalens szintaxis). Az olyan objektumok esetében, amelyeknek nincs emberi fogyasztásra szánt reprezentációja, az `str()` ugyanazt az értéket adja vissza, mint a `repr()`. Számos érték, például számok vagy struktúrák, például listák és szótárak ugyanazt a reprezentációt használják bármelyik függvény használatával. A karakterláncoknak különösen két különböző ábrázolása van.

Néhány példa:

```
>>> s = 'Helló, világ.'
>>> str(s)
'Helló, világ.'
>>> repr(s)
"'Helló, világ.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x értéke ' + repr(x) + ', és y értéke ' + repr(y) + '...'
>>> print(s)
x értéke 32.5, és y értéke 40000...
>>> # A string repr()-je idézőjeleket és fordított perjeleket ad hozzá:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # A repr() argumentuma bármely Python objektum lehet:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

A karakterlánc-modul tartalmaz egy `Template` osztályt, amely egy újabb módot kínál az értékek karakterláncokba való helyettesítésére, olyan helyőrzőket használva, mint a $x, és lecserélve azokat egy szótárból származó értékekkel, de sokkal kevésbé szabályozza a formázást.

7.1.1 Formázott karakterlánc-literálok

A formázott karakterlánc-literálok (röviden f-karakterláncoknak is nevezik) lehetővé teszik a Python-kifejezések értékének egy karakterláncon belüli elhelyezését úgy, hogy a karakterláncot `f` vagy `F` előtaggal írja, és a kifejezéseket `{kifejezés}`-ként írja be.

Egy opcionális formátum-meghatározó követheti a kifejezést. Ez lehetővé teszi az érték formázásának pontosabb szabályozását. A következő példa a tizedesjegy után három tizedesre kerekíti a pi-t:

```
>>> import math
>>> print(f'A pi értéke közelítőleg {math.pi:.3f}.')
A pi értéke közelítőleg 3.142.
```

Ha egész számot ad át a „:” jel után, akkor ez a mező minimális számú karakter széles lesz. Ez hasznos az oszlopok sorba rendezéséhez.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Más módosítók is használhatók az érték átalakítására a formázás előtt. Az `'!a'` az `ascii()`, az `'!s'` az `str()`, az `'!r'` pedig a `repr()`:

```
>>> animals = 'angolnákkal'
>>> print(f'A légpárnás hajóm tele van {animals}.')
A légpárnás hajóm tele van angolnákkal.
>>> print(f'My hovercraft is full of {animals!r}.')
A légpárnás hajóm tele van 'angolnákkal'.
```

Az = specifikálóval egy kifejezést a kifejezés szövegére, egyenlőségjelre, majd a kiértékelt kifejezés reprezentációjára lehet kiterjeszteni:

```
>>> bogarak = 'csótány'
>>> szam = 13
>>> helyiseg = 'nappali'
>>> print(f'Rovarirtás {bogarak=} {szam=} {helyiseg=}')
Rovarirtás bogarak='csótány' szam=13 helyiseg='nappali'
```

Az = specifikálóval kapcsolatos további információkért lásd az öndokumentáló kifejezéseket. A formátum-specifikációkkal kapcsolatos hivatkozásokért tekintse meg a formatspec referencia útmutatóját.

7.1.2 A String `format()` metódus

Az `str.format()` metódus alapvető használata így néz ki:

```
>>> print('Mi vagyunk a {}, akik azt mondják: "{}!"'.format('lovagok', 'Ni'))
Mi vagyunk a lovagok, akik azt mondják: "Ni!"
```

A zárójeleket és a bennük lévő karaktereket (úgynevezett formátum-mezőket) az `str.format()` metódusba átadott objektumokkal helyettesítjük. A zárójelben lévő szám az `str.format()` metódusba átadott objektum pozíciójára utalhat.

```
>>> print('{0} és {1}'.format('löncshús', 'tojás'))
löncshús és tojás
>>> print('{1} és {0}'.format('löncshús', 'tojás'))
tojás és löncshús
```

Ha kulcsszó argumentumokat használ az str.format() metódusban, akkor az értékekre az argumentum nevével hivatkozunk.

```
>>> print('Ez a {food} {adjective}.'.format(
... food='löncshús', adjective='teljesen szörnyű'))
Ez a löncshús teljesen szörnyű.
```

A pozíció- és kulcsszó argumentumok tetszőlegesen kombinálhatók:

```
>>> print('A történet szereplői {0}, {1}, és {other}.'.format('Bill', 'Manfred',
... other='Georg'))
A történet szereplői Bill, Manfred, és Georg.
```

Ha nagyon hosszú formátumú karakterlánca van, amelyet nem szeretne felosztani, jó lenne, ha a formázandó változókra név helyett pozíció szerint hivatkozhatna. Ezt úgy teheti meg, hogy egyszerűen átadja a szótárat, és szögletes zárójelek „[]” használatával éri el a kulcsokat.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
... 'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Ezt úgy is megteheti, hogy a `table` szótárát kulcsszó-argumentumként adja át `**` jelöléssel.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Ez különösen hasznos a beépített `vars()` függvénnyel kombinálva, amely az összes helyi változót tartalmazó szótárt ad vissza.

Példaként a következő sorok egy rendezetten igazított oszlopkészletet adnak, amely egész számokat, valamint azok négyzeteit és köbeit adják meg:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1   1
 2   4   8
 3   9  27
 4  16  64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Az `str.format()` karakterlánc-formázás teljes áttekintését a formatstrings részben találja.

7.1.3 Kézi karakterlánc formázás

Íme ugyanaz a négyzetek és köbök táblázata, kézzel formázva:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Figyelje meg az „end” használatát az előző sorban
...     print(repr(x*x*x).rjust(4))
...
 1   1   1
 2   4   8
 3   9  27
 4  16  64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Ne feledje, hogy az egyes oszlopok közötti egy szóközt a `print()` működési módja adja: mindig szóközt tesz az argumentumai közé.)

A `str.rjust()` a string objektumok metódusa jobbra igazítja a karakterláncot egy adott szélességű mezőben úgy, hogy a bal oldalon szóközzel tölti ki. Hasonló metódusok léteznek: `str.ljust()` és `str.center()`. Ezek a metódusok nem írnak semmit, csak egy új karakterláncot adnak vissza. Ha a bemeneti karakterlánc túl hosszú, nem csonkolják, hanem változatlanul adják vissza; ez összezavarja az oszlop elrendezését, de ez általában jobb, mint az alternatíva, amely hazudna egy értékről. (Ha valóban csonkolást szeretne, bármikor hozzáadhat egy szeletműveletet, például az `x.ljust(n)[:n]`-ben.)

Van egy másik módszer, az `str.zfill()`, amely a numerikus karakterláncot bal oldali nullákkal tölti fel. Ez megérti a plusz és mínusz jeleket is:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Régi karakterlánc formázás

A % operátor karakterlánc formázására is használható. Adott `'karakterlánc' % érték`, a % példányok a karakterláncban nulla vagy több `érték` elemre cserélődnek. Ezt a műveletet általában karakterlánc-interpolációnak nevezik. Például:

```
>>> import math
>>> print('A pi értéke közelítőleg %5.3f.' % math.pi)
A pi értéke közelítőleg 3.142.
```

További információ a régi karakterlánc-formázás részben található.

7.2 Fájlok olvasása és írása

Az `open()` egy fájlobjektumot ad vissza, és leggyakrabban két pozíciós argumentummal és egy kulcsszó argumentummal használatos: `open(fájlnév, mód, encoding=None)`.

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

Az első argumentum a fájlnevet tartalmazó karakterlánc. A második argumentum egy másik karakterlánc, amely néhány karaktert tartalmaz, amelyek leírják a fájl felhasználási módját. *mód* lehet `'r'`, amikor a fájl csak olvasásra kerül, `'w'` csak írásra (az azonos nevű meglévő fájl törlődik), és az `'a'` megnyitja a fájlt hozzáfűzéshez; a fájlba írt adatok automatikusan a végére kerülnek. Az `'r+'` megnyitja a fájlt az olvasáshoz és az íráshoz egyaránt. A *mód* argumentum nem kötelező; A kihagyása esetén a rendszer `'r'`-t feltételez.

Normális esetben a fájlok szöveges módban nyilnak meg, ami azt jelenti, hogy karakterláncokat olvas és ír a fájlból és a fájlba, amelyek egy meghatározott kódolásban vannak kódolva. Ha a kódolás nincs megadva, az alapértelmezés platformfüggő (lásd `open()`). Mivel az UTF-8 a modern de facto szabvány, az `encoding="utf-8"` használata javasolt, hacsak nem tudja, hogy más kódolást kell használnia. Ha a módhoz egy `'b'` betűt fűz, a fájl *bináris mód*ban nyílik meg. A bináris módú adatok bájtos objektumokként kerülnek beolvasásra és írásra. A fájl bináris módban történő megnyitásakor nem adható meg a kódolás.

Szöveges módban az alapértelmezett olvasási mód az, hogy a platformspecifikus sorvégeket (\n Unixon, \r\n Windowson) csak \n-re konvertálja. Szöveges módban történő íráskor az alapértelmezett \n előfordulásait platformspecifikus sorvégekké alakítja vissza. A fájladatok színfalak mögötti módosítása megfelelő szöveges fájlok esetén, de megsérti a bináris adatokat, mint például a JPEG vagy EXE fájlokban. Az ilyen fájlok olvasásakor és írásakor ügyeljen a bináris mód használatára.

Fájlobjektumok kezelésekor célszerű a `with` kulcsszót használni. Ennek az az előnye, hogy a fájl megfelelően bezárul a programcsomag befejezése után, még akkor is, ha valamikor kivétel történik. A `with` használata is sokkal rövidebb, mint a megfelelő `try-finally` blokkok írása:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # Ellenőrizhetjük, hogy a fájl automatikusan bezárult-e.
>>> f.closed
True
```

Ha nem a `with` kulcsszót használja, akkor hívja meg az `f.close()` parancsot a fájl bezárásához, és az általa használt rendszererőforrások azonnali felszabadításához.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Fájlobjektumok metódusai

A szakasz többi példája azt feltételezi, hogy egy f nevű fájlobjektum már létrejött.

A fájl tartalmának olvasásához hívja meg az `f.read(méret)` függvényt, amely beolvas bizonyos mennyiségű adatot, és azt karakterláncként (szöveges módban) vagy bájt objektumként (bináris módban) adja vissza. A *méret* egy opcionális numerikus argumentum. Ha a *méret*et elhagyja vagy negatív, a fájl teljes tartalma beolvasásra és visszaküldésre kerül; az a te problémád, ha a fájl kétszer akkora, mint a géped memóriája. Ellenkező esetben a rendszer legfeljebb *méret* karaktert (szöveges módban) vagy *méret* bájtot (bináris módban) olvas be és ad vissza. Ha elérte a fájl végét, az `f.read()` üres karakterláncot ('') ad vissza.

```
>>> f.read()
'Ez a teljes fájl.\n'
>>> f.read()
''
```

Az `f.readline()` egyetlen sort olvas be a fájlból; egy újsor karakter (`\n`) marad a karakterlánc végén, és csak akkor kerül ki a fájl utolsó sorából, ha a fájl nem újsor karakterre végződik. Ez egyértelművé teszi a visszatérési értéket; ha az `f.readline()` üres karakterláncot ad vissza, akkor a fájl végét értük el, míg az üres sort a '\n' jelöli, amely csak egyetlen újsor karaktert tartalmaz.

```
>>> f.readline()
'Ez a fájl első sora.\n'
>>> f.readline()
'A fájl második sora\n'
>>> f.readline()
''
```

Ha egy fájlból szeretne sorokat olvasni, ciklusban járja be a fájlobjektumot. Ez memóriatakarékos, gyors és egyszerű kódhoz vezet:

```
>>> for line in f:
...     print(line, end='')
...
Ez a fájl első sora.
A fájl második sora
```

Ha egy fájl összes sorát el akarja olvasni egy listában, használhatja a `list(f)` vagy `f.readlines()` parancsot is.

Az `f.write(string)` a string tartalmát írja a fájlba, visszaadva a fájlba írt karakterek számát.

```
>>> f.write('Ez egy teszt\n')
13
```

Más típusú objektumokat – vagy karakterláncokká (szöveges módban) vagy bájtos objektummá (bináris módban) – konvertálni kell, mielőtt beírná őket:

```
>>> value = ('A válasz', 42)
>>> s = str(value)  # a tuplét sztringgé konvertálja
>>> f.write(s)
16
```

Az `f.tell()` egy egész számot ad vissza, amely megadja a fájlobjektum aktuális pozícióját a fájlban, bájtok számaként a fájl elejétől számítva bináris módban, és buta számként szöveges módban.

A fájlobjektum pozíciójának megváltoztatásához használja az `f.seek(offset, whence)` parancsot. A pozíció kiszámítása az eltolás hozzáadásával történik egy referenciaponthoz; a referenciapontot a *whence* argumentum választja ki. A 0 *whence* az érték a fájl elejétől mér, az 1 az aktuális fájlpozíciót, a 2 pedig a fájl végét használja referenciapontként. A *whence* elhagyható, és alapértelmezés szerint 0, a fájl elejét használva referenciapontként.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)  # Lépjen a fájl 6. bájtjához
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Lépjen a 3. bájthoz a vége előtt
13
>>> f.read(1)
b'd'
```

Szövegfájlokban (amelyek b nélkül nyitottak meg a *mód* karakterláncban) csak a fájl elejéhez viszonyított keresés engedélyezett (kivétel a fájl legvégéig történő keresés a `seek(0, 2)`-vel), és az egyetlen érvényes *offset* értékek: az `f.tell()`-ből vagy nullából visszaadott. Minden *offset* eltolási érték meghatározhatatlan viselkedést eredményez.

A fájlobjektumoknak van néhány további metódusa, például az `isatty()` és a `truncate()`, amelyeket ritkábban használnak; olvassa el a Library Reference-t a fájlobjektumokkal kapcsolatos teljes útmutatóért.

7.2.2 Strukturált adatok mentése json segítségével

A karakterláncok könnyen írhatók egy fájlba, és könnyen kiolvashatók onnan. A számok egy kicsit több erőfeszítést igényelnek, mivel a `read()` metódus csak karakterláncokat ad vissza, amelyeket át kell adni egy függvénynek, mint például az `int()`, amely egy `'123'`-hoz hasonló karakterláncot

vesz fel, és 123-as számértéket ad vissza. Bonyolultabb adattípusok, például beágyazott listák és szótárak mentése, kézi elemzése és sorba rendezése bonyolulttá válik.

Ahelyett, hogy a felhasználók állandóan kódot írnának és hibakeresnének, hogy bonyolult adattípusokat fájlba menthessenek, a Python lehetővé teszi a népszerű JSON (JavaScript Object Notation) nevű adatcsere-formátum használatát. A json nevű szabványos modul képes Python adathierarchiákat felvenni, és karakterlánc-reprezentációkká alakítani; ezt a folyamatot szerializálásnak nevezik. Az adatok karakterlánc-reprezentációból való rekonstrukcióját deszerializálásnak nevezzük. A szerializálás és a deszerializálás között előfordulhat, hogy az objektumot reprezentáló karakterlánc fájlban vagy adatban tárolódik, vagy hálózati kapcsolaton keresztül elküldhető egy távoli gépre.

Megjegyzés: A modern alkalmazások általában a JSON formátumot használják az adatcsere lehetővé tételére. Sok programozó már ismeri, ezért jó választás az együttműködéshez.

Ha rendelkezik egy x objektummal, megtekintheti annak JSON karakterlánc-ábrázolását egy egyszerű kódsor segítségével:

```
>>> import json
>>> x = [1, 'egyszerű', 'lista']
>>> json.dumps(x)
'[1, "egyszerű", "lista"]'
```

A `dumps()` függvény egy másik változata, a `dump()` egyszerűen szerializálja az objektumot egy szövegfájlba. Tehát ha `f` egy írásra megnyitott szövegfájl objektum, akkor ezt tehetjük:

```
json.dump(x, f)
```

Az objektum újbóli dekódolásához, ha `f` egy bináris fájl vagy szövegfájl objektum, amelyet olvasásra nyitottak meg:

```
x = json.load(f)
```

Megjegyzés: A JSON-fájloknak UTF-8 kódolásúnak kell lenniük. Használja az `encoding="utf-8"` kódot, amikor a JSON-fájlt szövegfájlként nyitja meg íráshoz és olvasáshoz egyaránt.

Ez az egyszerű szerializálási technika képes kezelni a listákat és a szótárakat, de tetszőleges osztálypéldányok szerializálása a JSON-ban némi extra erőfeszítést igényelnek. A `json` modulra vonatkozó hivatkozás ennek magyarázatát tartalmazza.

Lásd még:

`pickle` - a pickle modul

A JSON-nal ellentétben a *pickle* egy olyan protokoll, amely lehetővé teszi tetszőlegesen összetett Python-objektumok szerializálását. Mint ilyen, a Pythonra jellemző, és nem használható más nyelveken írt alkalmazásokkal való kommunikációra. Alapértelmezés szerint szintén nem biztonságos: a nem megbízható forrásból származó pickle adatok deszerializálása tetszőleges kódot futtathat, ha az adatokat egy képzett támadó készítette.

8. Hibák és kivételek

Eddig a hibaüzenetek nem voltak többen az említettnél, de ha már kipróbáltad a példákat, akkor valószínűleg találkoztál vele. A hibáknak (legalább) két fajtája különböztethető meg: a *szintaktikai hibák* és a *kivételek*.

8.1 Szintaktikai hibák

A szintaktikai hibák, más néven nyelvtani hibák, talán a leggyakoribb reklamációk, amelyeket a Python tanulása közben kaphatunk:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

Az elemző megismétli a szabálysértő sort, és egy kis „nyilat” jelenít meg, amely a sor legkorábbi pontjára mutat, ahol a hibát észlelte. A hibát a nyíl előtti token okozza (vagy legalábbis az észlelte): a példában a hibát a `print()` függvénynél észleljük, mivel előtte hiányzik a kettőspont (`':'`). A fájlnév és a sorszám kinyomtatásra kerül, így tudni fogja, hol keresse, ha a bemenet szkriptből származik.

8.2 Kivételek

Még ha egy utasítás vagy kifejezés szintaktikailag helyes is, hibát okozhat, amikor megpróbálják végrehajtani. A végrehajtás során észlelt hibákat kivételeknek nevezzük, és nem feltétlenül végzetesek: hamarosan megtanuljuk, hogyan kell kezelni őket a Python programokban. A legtöbb kivételt azonban nem kezelik a programok, és az alábbi hibaüzeneteket eredményezi:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

A hibaüzenet utolsó sora jelzi, hogy mi történt. A kivételek különböző típusúak, és a típus az üzenet részeként kerül kinyomtatásra: a példában szereplő típusok: `ZeroDivisionError` (nullával való osztás hiba), `NameError` (név hiba) és `TypeError` (típus hiba). A kivételtípusként nyomtatott

karakterlánc a bekövetkezett beépített kivétel neve. Ez minden beépített kivételre igaz, de nem feltétlenül igaz a felhasználó által meghatározott kivételekre (bár ez egy hasznos konvenció). A szabványos kivételnevek beépített azonosítók (nem lefoglalt kulcsszavak).

A sor többi része a kivétel típusa és a kiváltó ok alapján nyújt részleteket.

A hibaüzenet előző része azt a környezetet mutatja, ahol a kivétel előfordult, verem-visszakövetés formájában. Általában tartalmaz egy verem nyomkövetési listát a forrássorokról; azonban nem jeleníti meg a szabványos bemenetről olvasott sorokat.

Lásd a Beépített kivételek listájánál a beépített kivételeket és azok jelentését.

8.3 A kivételek kezelése

Lehetséges olyan programokat írni, amelyek a kiválasztott kivételeket kezelik. Nézzük meg a következő példát, amely egy érvényes egész szám beviteléig kéri a felhasználót, de lehetővé teszi a program megszakítását (a `Control-C` billentyűkombinációval vagy bármi mással, amit az operációs rendszer támogat); vegye figyelembe, hogy a felhasználó által generált megszakítást a `KeyboardInterrupt` kivétel fellépése jelzi.

```
>>> while True:
...     try:
...         x = int(input("Kérem, adjon meg egy számot: "))
...         break
...     except ValueError:
...         print("Oops! Ez nem érvényes szám. Próbálja újra...")
...
```

A `try` utasítás a következőképpen működik.

- Először a `try` záradék (a `try` és az `except` kulcsszavak közötti utasítás(ok)) kerülnek végrehajtásra.
- Ha nem történik kivétel, akkor az *except záradék* kimarad, és a `try` utasítás végrehajtása befejeződik.
- Ha kivétel történik a `try` záradék végrehajtása során, a záradék többi része kimarad. Ezután, ha a típusa megegyezik az except kulcsszó után elnevezett kivétellel, akkor az *except záradék* végrehajtásra kerül, majd a try/except blokk után folytatódik a végrehajtás.
- Ha olyan kivétel fordul elő, amely nem egyezik az *except záradék*ban megnevezett kivétellel, akkor azt továbbítja a külső `try` utasításoknak; ha nem található kezelő, akkor ez egy kezeletlen kivétel, és a végrehajtás leáll egy üzenettel, amint az fent látható.

Egy try utasításnak egynél több *kivétel záradéka* lehet, hogy megadja a kezelőket a különböző kivételekhez. Legfeljebb egy kezelő kerül végrehajtásra. A kezelők csak azokat a kivételeket kezelik, amelyek a megfelelő *try záradék*ban fordulnak elő, ugyanazon `try` utasítás más kezelőiben nem. Egy kivétel záradék több kivételt is megnevezhet zárójeles sorként, például:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Egy `except` záradékában lévő osztály kompatibilis a kivétellel, ha ugyanaz az osztály vagy annak alaposztálya (de nem fordítva – a származtatott osztályt felsoroló *kivétel záradék* nem kompatibilis egy alaposztállyal). Például a következő kód B, C, D betűket nyomtat ebben a sorrendben:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Ne feledje, hogy ha a *kivétel záradékot* megfordítjuk (az `except B`-vel először), akkor a B, B, B kinyomtatása lett volna – az első passzoló *kivétel záradék* aktiválódik.

Kivétel esetén előfordulhatnak hozzárendelt értékek, más néven a kivétel argumentumai. Az argumentumok jelenléte és típusa a kivétel típusától függ.

A *kivétel záradék* megadhat egy változót a kivétel neve után. A változó a kivételpéldányhoz van kötve, amelynek jellemzően egy `args` attribútuma van, amely az argumentumokat tárolja. A kényelem kedvéért a beépített kivételtípusok definiálják az `__str__()` paramétert az összes argumentum kinyomtatásához anélkül, hogy kifejezetten hozzáférnének az `.args`-hoz.

```
>>> try:
...     raise Exception('löncshús', 'tojások')
... except Exception as inst:
...     print(type(inst)) # a kivétel típusa
...     print(inst.args)  # az argumentumok .args-ban tárolva
...     print(inst) # __str__ lehetővé teszi az args közvetlen nyomtatását,
...                 # de a kivétel alosztályaiban felülírható
...     x, y = inst.args # args kicsomagolása
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('löncshús', 'tojások')
('löncshús', 'tojások')
x = löncshús
y = tojások
```

A kivétel `__str__()` kimenete a kezeletlen kivételek üzenetének utolsó részeként ('részlet') kerül nyomtatásra.

A `BaseException` az összes kivétel közös alaposztálya. Egyik alosztálya, az `Exception`, az összes nem végzetes kivétel alaposztálya. Azokat a kivételeket, amelyek nem az `Exception` alosztályai, általában nem kezelik, mert arra szolgálnak, hogy jelezzék, hogy a programnak le kell állnia. Ide tartozik a `SystemExit`, amelyet a `sys.exit()` és a `KeyboardInterrupt`, amely akkor lép fel, amikor a felhasználó meg akarja szakítani a programot.

Az `Exception` helyettesítőként használható, ami (majdnem) mindent elkap. Jó gyakorlat azonban, hogy a lehető legpontosabban fogalmazzuk meg a kezelni kívánt kivételtípusokat, és hagyjuk, hogy a váratlan kivételek továbbterjedjenek.

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

A `try … except` utasításnak van egy opcionális *else záradéka*, amelynek jelenléte esetén követnie kell az összes *kivétel záradékot*. Hasznos olyan kódnál, amelyet akkor kell végrehajtani, ha a *try záradék* nem okoz kivételt. Például:

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

Az `else` záradék használata jobb, mintha további kódot adna a `try` záradékhoz, mert elkerüli, hogy véletlenül olyan kivételt kapjanak, amelyet nem a `try … except` utasítással védett kód idézett elő.

A kivételkezelők nem csak azokat a kivételeket kezelik, amelyek közvetlenül a *try záradék*ban fordulnak elő, hanem azokat is, amelyek a *try záradék*ban (akár közvetetten) meghívott függvényeken belül fordulnak elő. Például:

```
>>> def this_fails():
...     x = 1/0
...
```

```
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Kivételek felvetése

A `raise` utasítás lehetővé teszi a programozó számára, hogy egy meghatározott kivételt kényszerítsen ki. Például:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Az egyetlen felvetendő argumentum a felhozandó kivételt jelzi. Ennek vagy kivételpéldánynak vagy kivételosztálynak kell lennie (olyan osztály, amely a `BaseException`-ből származik, például az `Exception` vagy annak valamelyik alosztálya). Ha egy kivételosztályt adunk át, akkor az implicit módon példányosodik a konstruktor meghívásával, argumentumok nélkül:

```
raise ValueError # a 'raise ValueError()' rövidítése
```

Ha meg kell határoznia, hogy felmerült-e egy kivétel, de nem kívánja kezelni, a `raise` utasítás egy egyszerűbb formája lehetővé teszi a kivétel újbóli fellépését:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('A kivétel elrepült!')
...     raise
...
A kivétel elrepült!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5 A kivétel láncolása

Ha egy nem kezelt kivétel történik egy kivétel szakaszon belül, akkor a kezelendő kivétel csatolva lesz hozzá, és szerepelni fog a hibaüzenetben:

```
>>> try:
...     open("database.sqlite")
... except OSError:
```

```
...     raise RuntimeError("képtelen kezelni a hibát")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: képtelen kezelni a hibát
```

Annak jelzésére, hogy egy kivétel egy másik közvetlen következménye, a raise utasítás megenged egy opcionális `from` záradékot:

```
# Az exc-nek kivétel példánynak vagy None-nak kell lennie.
raise RuntimeError from exc
```

Ez a kivételek átalakításakor lehet hasznos. Például:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Nem sikerült megnyitni az adatbázist') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Nem sikerült megnyitni az adatbázist
```

Lehetővé teszi az automatikus kivételláncolás letiltását is a `from None` idióma használatával:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

A láncolási mechanikával kapcsolatos további információkért lásd a beépített kivételeket.

8.6 Felhasználó által meghatározott kivételek

A programok elnevezhetik saját kivételeiket egy új kivételosztály létrehozásával (a Python osztályokkal kapcsolatos további információkért lásd: Osztályok). A kivételeket általában az `Exception` osztályból kell származtatni, akár közvetlenül, akár közvetve.

A kivétel osztályok definiálhatók, amelyek bármire képesek, de általában egyszerűek, és gyakran csak számos attribútumot kínálnak, amelyek lehetővé teszik a hibával kapcsolatos információk kinyerését a kivétel kezelői számára.

A legtöbb kivétel „Error”-ra végződő nevekkel van meghatározva, hasonlóan a szokásos kivételek elnevezéséhez.

Sok szabványos modul saját kivételeket határoz meg az általuk meghatározott függvényekben esetlegesen előforduló hibák jelentésére.

8.7 Tisztítási műveletek meghatározása

A `try` utasításnak van egy másik nem kötelező záradéka is, amely minden körülmények között végrehajtandó tisztítási műveletek meghatározására szolgál. Például:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Viszlát világ!')
...
Viszlát világ!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Ha egy `finally` záradék van jelen, a final záradék a `try` utasítás befejezése előtti utolsó feladatként fog végrehajtani. Az utolsó záradék lefut, függetlenül attól, hogy a `try` utasítás létrehoz-e kivételt vagy sem. A következő pontok bonyolultabb eseteket tárgyalnak, amikor kivétel történik:

- Ha kivétel történik a `try` záradék végrehajtása során, a kivételt egy `except` záradék kezelheti. Ha a kivételt nem egy `except` záradék kezeli, a kivétel újra felemelkedik a `finally` záradék végrehajtása után.
- Kivétel előfordulhat egy `except` vagy `else` záradék végrehajtása során. A kivételt ismét felvetik, miután a `finally` záradékot végrehajtották.
- Ha a `finally` záradék `break`, `continue` vagy `return` utasítást hajt végre, a kivételek nem merülnek fel újra.

- Ha a `try` utasítás eléri a `break`, a `continue` vagy a `return` utasítást, akkor a `finally` záradék közvetlenül a `break`, `continue` vagy `return` utasítás végrehajtása előtt fut le.
- Ha egy `finally` záradék tartalmaz egy `return` utasítást, akkor a visszaadott érték a `finally` záradék `return` utasításának értéke lesz, nem pedig a `try` záradék `return` utasításának értéke.

Például:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Egy bonyolultabb példa:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("osztás nullával!")
...     else:
...         print("az eredmény", result)
...     finally:
...         print("finally záradék végrehajtása")
...
>>> divide(2, 1)
az eredmény 2.0
finally záradék végrehajtása
>>> divide(2, 0)
osztás nullával!
finally záradék végrehajtása
>>> divide("2", "1")
finally záradék végrehajtása
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Amint látja, a `finally` záradék minden esetben végrehajtódik. A két karakterlánc osztásával felvetett `TypeError`-t az `except` nem kezeli, ezért a `finally` záradék végrehajtása után újra fellép.

Valós alkalmazásokban az `finally` záradék hasznos külső erőforrások (például fájlok vagy hálózati kapcsolatok) felszabadításához, függetlenül attól, hogy az erőforrás felhasználása sikeres volt-e.

8.8 Előre definiált tisztítási műveletek

Egyes objektumok szabványos tisztítási műveleteket határoznak meg, amelyeket akkor kell végrehajtani, ha az objektumra már nincs szükség, függetlenül attól, hogy az objektumot használó művelet sikeres volt-e vagy sem. Tekintse meg a következő példát, amely megpróbál megnyitni egy fájlt, és kinyomtatni a tartalmát a képernyőre.

```
for line in open("myfile.txt"):
    print(line, end="")
```

Ezzel a kóddal az a probléma, hogy a kód ezen részének végrehajtása után meghatározatlan ideig nyitva hagyja a fájlt. Ez nem probléma az egyszerű szkripteknél, de nagyobb alkalmazásoknál probléma lehet. A with utasítás lehetővé teszi az objektumok, például a fájlok oly módon történő használatát, hogy azok mindig azonnal és helyesen megtisztuljanak.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Az utasítás végrehajtása után az *f* fájl mindig bezárul, még akkor is, ha probléma merült fel a sorok feldolgozása során. Azon objektumok, amelyek a fájlokhoz hasonlóan előre meghatározott tisztítási műveleteket biztosítanak, ezt jelzik a dokumentációjukban.

8.9 Több, egymással nem összefüggő kivétel felemelése és kezelése

Vannak helyzetek, amikor több előfordult kivételt is jelenteni kell. Ez gyakran előfordul a párhuzamossági keretrendszerekben, amikor több feladat is meghiúsult párhuzamosan, de vannak más felhasználási esetek is, amikor kívánatos a végrehajtás folytatása és több hiba összegyűjtése az első kivétel felvetése helyett.

A beépített `ExceptionGroup` összegyűjti a kivételpéldányok listáját, hogy együtt lehessen őket emelni. Ez maga a kivétel, így elkapható, mint bármely más kivétel.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  | File "<stdin>", line 1, in <module>
  | File "<stdin>", line 3, in f
  | ExceptionGroup: there were problems
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
```

```
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

Ha az `except` helyett az `except*`-ot használjuk, akkor a csoportban csak azokat a kivételeket tudjuk szelektíven kezelni, amelyek egy bizonyos típusnak megfelelnek. A következő példában, amely egy beágyazott kivételcsoportot mutat be, mindegyik `except*` záradék egy bizonyos típusú csoportkivételekből bont ki, miközben hagyja, hogy az összes többi kivétel átterjedjen más záradékokra, és végül újra felemelkedhessen.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  | File "<stdin>", line 2, in <module>
  | File "<stdin>", line 2, in f
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
    +-+---------------- 1 ----------------
      | RecursionError: 4
```

```
      +------------------------------------
>>>
```

Vegye figyelembe, hogy a kivételcsoportokba beágyazott kivételeknek példányoknak kell lenniük, nem típusoknak. Ennek az az oka, hogy a gyakorlatban a kivételek jellemzően azok, amelyeket a program már felvetett és elkapott, a következő minta szerint:

```
>>> excs = []
... for test in tests:
... try:
... test.run()
... except Exception as e:
... excs.append(e)
...
>>> if excs:
... raise ExceptionGroup("Test Failures", excs)
...
```

8.10 Kivételek gazdagítása megjegyzésekkel

Amikor kivételt hoznak létre annak érdekében, hogy fel lehessen emelni, általában inicializálják a fellépő hibát leíró információkkal. Vannak esetek, amikor hasznos információ hozzáadása a kivétel észlelése után. Ebből a célból a kivételeknek van egy `add_note(note)` metódusa, amely elfogad egy karakterláncot, és hozzáadja a kivétel megjegyzéslistájához. A szabványos visszakövetési renderelés tartalmazza az összes megjegyzést, abban a sorrendben, ahogyan a kivétel után hozzáadásra kerültek.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
>>>
```

Például, amikor kivételeket gyűjtünk egy kivételcsoportba, kontextusinformációkat szeretnénk hozzáadni az egyes hibákhoz. A következőkben a csoport minden kivételéhez tartozik egy megjegyzés, amely jelzi, hogy mikor fordult elő ez a hiba.

```
>>> def f():
...     raise OSError('operation failed')
```

```
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  | File "<stdin>", line 1, in <module>
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    | File "<stdin>", line 3, in <module>
    | File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    | File "<stdin>", line 3, in <module>
    | File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    | File "<stdin>", line 3, in <module>
    | File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

9. Osztályok

Az osztályok lehetőséget biztosítanak az adatok és a funkciók összekapcsolására. Egy új osztály létrehozása új *típusú* objektumot hoz létre, amely lehetővé teszi az adott típusú új *példányok* létrehozását. Minden osztálypéldányhoz tartozhatnak attribútumok az állapot fenntartása érdekében. Az osztálypéldányoknak lehetnek (az osztálya által meghatározott) metódusok is az állapotuk módosítására.

Más programozási nyelvekhez képest a Python osztálymechanizmusa minimális új szintaxist és szemantikát tartalmazó osztályokat ad hozzá. A C++-ban és a Modula-3-ban található osztálymechanizmusok keveréke. A Python osztályok biztosítják az objektumorientált programozás összes szabványos funkcióját: az osztályöröklési mechanizmus több alaposztályt is lehetővé tesz, a származtatott osztály felülírhatja az alaposztályának vagy osztályainak bármely metódusát, a metódus pedig meghívhatja egy azonos nevű alaposztály metódusát. Az objektumok tetszőleges mennyiségű és típusú adatot tartalmazhatnak. A modulokhoz hasonlóan az osztályok is részesednek a Python dinamikus természetéből: futás közben jönnek létre, és létrehozásuk után tovább módosíthatók.

A C++ terminológiában az osztálytagok (beleértve az adattagokat is) általában *nyilvános*ak (kivéve a *privát változókat* lent), és minden tagfüggvény *virtuális*. A Modula-3-hoz hasonlóan itt sincsenek rövidítések az objektum tagjaira való hivatkozáshoz a metódusaiból: a metódusfüggvény egy explicit első argumentummal van deklarálva, amely az objektumot reprezentálja, amelyet a hívás implicit módon biztosít. A Smalltalk-hoz hasonlóan az osztályok maguk is objektumok. Ez szemantikát biztosít az importáláshoz és az átnevezéshez. A C++-tól és a Modula-3-tól eltérően a beépített típusokat alaposztályként használhatja a felhasználó a kiterjesztéshez. A C++-hoz hasonlóan a legtöbb beépített operátor speciális szintaxissal (aritmetikai operátorok, feliratkozás stb.) újradefiniálható az osztálypéldányokhoz.

(Hiányzik az általánosan elfogadott terminológia az osztályokról, ezért alkalmanként a Smalltalk és a C++ kifejezéseket használom. Modula-3 kifejezéseket használnék, mivel annak objektumorientált szemantikája közelebb áll a Pythonhoz, mint a C++-hoz, de arra számítok, hogy kevés olvasó hallottam már róla.)

9.1 Egy szó a nevekről és az objektumokról

Az objektumok egyediek, és több név (több hatókörben) köthető ugyanahhoz az objektumhoz. Ezt más nyelveken aliasnak nevezik. Ez általában nem értékelhető első pillantásra a Pythonnál, és nyugodtan figyelmen kívül hagyható, ha megváltoztathatatlan alaptípusokkal (számok, karakterláncok, tuplék) foglalkozunk. Az aliasing azonban meglepő hatással van a Python-kód szemantikájára, amely változó objektumokat, például listákat, szótárakat és a legtöbb egyéb típust tartalmaz. Ezt általában a program javára használják, mivel az álnevek bizonyos szempontból mutatóként viselkednek. Például egy objektum átadása olcsó, mivel csak egy mutatót ad át a megvalósítás; és ha egy függvény módosít egy argumentumként átadott objektumot, a hívó látni fogja a változást – így nincs szükség két különböző argumentumátadási mechanizmusra, mint a Pascalban.

9.2 A Python hatókörei és névterei

Az osztályok bemutatása előtt először el kell mondanom valamit a Python hatóköri szabályairól. Az osztálydefiníciók jó trükköket játszanak a névterekkel, és ismernie kell a hatókörök és névterek működését, hogy teljes mértékben megértse, mi történik. Egyébként minden haladó Python programozó számára hasznosak az ismeretek ebben a témában.

Kezdjük néhány meghatározással.

A *névtér* a nevek és az objektumok leképezése. A legtöbb névteret jelenleg Python szótárként valósítják meg, de ez általában semmilyen módon nem észrevehető (kivéve a teljesítményt), és a jövőben változhat. Példák névterekre: a beépített nevek halmaza (függvényeket, például `abs()` és beépített kivételneveket tartalmaz); a globális nevek egy modulban; és a helyi nevek függvényhívásban. Bizonyos értelemben egy objektum attribútumainak halmaza is névteret alkot. A névterekről fontos tudni, hogy a különböző névterekben lévő nevek között egyáltalán nincs kapcsolat; például két különböző modul is definiálhat egy `maximize` funkciót, zavarás nélkül – a modulok felhasználóinak a modul nevével kell előtagot fűzniük.

Egyébként az attribútum szót minden pont után következő névre használom – például a `z.real` kifejezésben a `real` a `z` objektum attribútuma. Szigorúan véve a modulokban lévő nevekre való hivatkozások attribútum-hivatkozások: a `modname.funcname` kifejezésben a `modname` egy modulobjektum, a `funcname` pedig egy attribútum. Ebben az esetben a modul attribútumai és a modulban meghatározott globális nevek között egyértelmű leképezés történik: ugyanazon a névteren osztoznak!

Egy dolgot kivéve. A modulobjektumok titkos, csak olvasható attribútummal rendelkeznek __dict__ néven, amely a modul névterének megvalósításához használt szótárat adja vissza; a __dict__ név attribútum, de nem globális név. Nyilvánvaló, hogy ennek használata sérti a névtér-megvalósítás absztrakcióját, és olyan dolgokra kell korlátozódnia, mint például a post-mortem hibakeresők.

Az attribútumok lehetnek csak olvashatók vagy írhatók. Ez utóbbi esetben lehetséges az attribútumokhoz való hozzárendelés. A modul attribútumai írhatók: `modname.the_answer = 42` írható. Az írható attribútumok a `del` utasítással is törölhetők. Például a `del modname.the_answer` eltávolítja a `the_answer` attribútumot a `modname` által megnevezett objektumból.

A névterek különböző pillanatokban jönnek létre, és eltérő élettartammal rendelkeznek. A beépített neveket tartalmazó névtér a Python értelmező indulásakor jön létre, és soha nem törlődik. A modul globális névtere a moduldefiníció beolvasásakor jön létre; normál esetben a modul névterei is az értelmező kilépéséig tartanak. Az interpreter legfelső szintű meghívása által végrehajtott utasítások, akár szkriptfájlból, akár interaktív módon, a `__main__` nevű modul részének tekintendők, így saját globális névterük van. (A beépített nevek valójában egy modulban is élnek; ezt `builtin`-nek nevezzük.)

A függvény helyi névtere a függvény meghívásakor jön létre, és törlődik, ha a függvény olyan kivételt ad vissza vagy idéz elő, amelyet a függvény nem kezel. (Valójában a felejtés jobb módja annak, hogy leírjuk, mi történik valójában.) Természetesen a rekurzív invokációk mindegyikének megvan a saját helyi névtere.

A *hatókör* egy Python-program szöveges régiója, ahol a névtér közvetlenül elérhető. A „közvetlenül elérhető” itt azt jelenti, hogy egy névre mutató korlátlan hivatkozás megpróbálja megtalálni a nevet a névtérben.

Bár a hatóköröket statikusan határozzák meg, dinamikusan használják őket. A végrehajtás során bármikor 3 vagy 4 beágyazott hatókör létezik, amelyek névterei közvetlenül elérhetők:

- az elsőként keresett legbelső hatókör a helyi neveket tartalmazza
- a legközelebbi befoglaló hatókörtől kezdve keresett befoglaló függvények hatókörei nem helyi, de nem globális neveket is tartalmaznak
- az utolsó előtti hatókör az aktuális modul globális neveit tartalmazza
- a legkülső hatókör (utoljára keresve) a beépített neveket tartalmazó névtér

Ha egy név globálisnak van deklarálva, akkor minden hivatkozás és hozzárendelés közvetlenül a modul globális neveit tartalmazó, az utolsó előtti hatókörhöz kerül. A legbelső hatókörön kívül található változók újrakötéséhez a `nonlocal` utasítás használható; ha nem deklarálják nem lokálisnak, akkor ezek a változók csak olvashatók (egy ilyen változóba való írási kísérlet egyszerűen létrehoz egy *új* helyi változót a legbelső hatókörben, az azonos nevű külső változót változatlanul hagyva).

A helyi hatókör általában a (szövegszerűen) aktuális függvény helyi nevére hivatkozik. A függvényeken kívül a helyi hatókör ugyanarra a névtérre hivatkozik, mint a globális hatókör: a modul névterére. Az osztálydefiníciók még egy névteret helyeznek el a helyi hatókörben.

Fontos felismerni, hogy a hatóköröket szövegesen határozzák meg: a modulban meghatározott függvény globális hatóköre az adott modul névtere, függetlenül attól, hogy honnan vagy milyen álnéven hívják a függvényt. Másrészt a tényleges nevek keresése dinamikusan, futásidőben történik – a nyelvdefiníció azonban a statikus névfeloldás felé fejlődik, „fordítási” időben, ezért ne hagyatkozzon a dinamikus névfeloldásra! (Valójában a helyi változók már statikusan meghatározottak.)

A Python különlegessége, hogy – ha nincs érvényben `global` vagy `nonlocal` utasítás – a nevek hozzárendelése mindig a legbelső hatókörbe kerül. A hozzárendelések nem másolnak adatokat – csak neveket kötnek az objektumokhoz. Ugyanez igaz a törlésekre is: a `del x` utasítás eltávolítja az `x` összerendelését a helyi hatókör által hivatkozott névtérből. Valójában minden új neveket bevezető művelet a helyi hatókört használja: különösen az `import` utasítások és függvénydefiníciók kötik össze a modul vagy függvény nevét a helyi hatókörben.

A `global` utasítás arra használható, hogy jelezze, hogy bizonyos változók a globális hatókörben élnek, és ott vissza kell térniük; a `nonlocal` utasítás azt jelzi, hogy bizonyos változók egy bezáró hatókörben élnek, és ott vissza kell térniük.

9.2.1 Példa a hatókörökhöz és a névterekhez

Ez egy példa bemutatja, hogyan lehet hivatkozni a különböző hatókörökre és névterekre, és hogyan befolyásolja a `global` és a `nonlokal` a változók összerendelését:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
    scope_test()
    print("In global scope:", spam)
```

A példakód kimenete:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Vegye figyelembe, hogy a helyi hozzárendelés (amely az alapértelmezett) nem változtatta meg a *scope_test spam*-kötését. A `nonlocal` hozzárendelés megváltoztatta a *spam scope_test* összerendelését, a `global` hozzárendelés pedig a modulszintű összerendelést.

Azt is láthatja, hogy a `global` hozzárendelés előtt nem volt korábbi *spam*-kötés.

9.3 Első pillantás az osztályokra

Az osztályok egy kis új szintaxist, három új objektumtípust és néhány új szemantikát mutatnak be.

9.3.1 Osztálydefiníciós szintaxis

Az osztálydefiníció legegyszerűbb formája így néz ki:

```
class ClassName:
    <statement-1>
    .
    .
```

```
    .
    <statement-N>
```

Az osztálydefiníciókat, például a függvénydefiníciókat (`def` utasításokat) végre kell hajtani, mielőtt bármilyen hatást gyakorolnának. (Elképzelhető, hogy egy osztálydefiníciót elhelyezhet egy `if` utasítás ágában vagy egy függvényen belül.)

A gyakorlatban az osztálydefiníciókon belüli utasítások általában függvénydefiníciók, de más utasítások is megengedettek, és néha hasznosak – erre később visszatérünk. Az osztályon belüli függvénydefiníciók általában egy sajátos argumentumlistával rendelkeznek, amelyet a metódusok hívási konvenciói diktálnak – ezt a későbbiekben is elmagyarázzuk.

Az osztálydefiníció megadásakor egy új névtér jön létre, amelyet helyi hatókörként használnak – így a helyi változókhoz rendelt összes hozzárendelés ebbe az új névtérbe kerül. Különösen a függvénydefiníciók kötik ide az új függvény nevét.

Ha egy osztálydefiníciót normálisan hagyunk (a végén), akkor létrejön egy *osztályobjektum*. Ez alapvetően az osztálydefiníció által létrehozott névtér tartalma körüli burkolóanyag; a következő részben többet tudhatunk meg az osztályobjektumokról. Az eredeti helyi hatókör (az osztálydefiníció megadása előtt érvényben lévő) visszaállításra kerül, és az osztályobjektum ide van kötve az osztálydefiníció fejlécében megadott osztálynévhez (a példában az ClassName).

9.3.2 Osztályobjektumok

Az osztályobjektumok kétféle műveletet támogatnak: attribútum-hivatkozásokat és példányosítást.

Az *attribútum-hivatkozások* a Python összes attribútum-hivatkozásánál használt szabványos szintaxist használják: `obj.name`. Az érvényes attribútumnevek mindazok a nevek, amelyek az osztály névterében voltak az osztályobjektum létrehozásakor. Tehát, ha az osztálydefiníció így néz ki:

```
class MyClass:
    """Egy egyszerű példa osztály """
    i = 12345

    def f(self):
        return 'hello world'
```

akkor a `MyClass.i` és a `MyClass.f` érvényes attribútumhivatkozások, amelyek egy egész számot és egy függvényobjektumot adnak vissza. Osztályattribútumok is hozzárendelhetők, így hozzárendeléssel módosíthatja a `MyClass.i` értékét. A `__doc__` szintén érvényes attribútum, amely az osztályhoz tartozó docstringet adja vissza: `"Egy egyszerű példa osztály"`.

Az osztály *példányosítása* függvényjelölést használ. Csak tegyen úgy, mintha az osztályobjektum egy paraméter nélküli függvény, amely az osztály új példányát adja vissza. Például (a fenti osztályt feltételezve):

```
x = MyClass()
```

létrehozza az osztály új példányát, és hozzárendeli ezt az objektumot az `x` helyi változóhoz.

A példányosítási művelet (egy osztályobjektum „hívása”) egy üres objektumot hoz létre. Sok osztály szeret objektumokat létrehozni egy adott kezdeti állapotra testreszabott példányokkal. Ezért egy osztály definiálhat egy `__init__()` nevű speciális metódust, például:

```
def __init__(self):
    self.data = []
```

Amikor egy osztály `__init__()` metódust definiál, az osztály példányosítása automatikusan meghívja az `__init__()` függvényt az újonnan létrehozott osztálypéldányhoz. Tehát ebben a példában egy új, inicializált példány a következőképpen szerezhető be:

```
x = MyClass()
```

Természetesen az __init__() metódusnak lehetnek argumentumai a nagyobb rugalmasság mellett. Ebben az esetben az osztály példányosítási operátorának adott argumentumok az `__init__()`-ba kerülnek. Például,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Példány objektumok

Most mit tehetünk a példány objektumokkal? A példány objektumok által csak az attribútumhivatkozások érthetők műveletek. Kétféle érvényes attribútumnév létezik: adat attribútumok és metódusok.

Az *adat attribútumok* Smalltalk-ban a „példány változóknak”, C++-ban pedig az „adat tagoknak” felelnek meg. Az adat attribútumokat nem kell deklarálni; a lokális változókhoz hasonlóan akkor jönnek létre, amikor először hozzárendeljük őket. Például, ha `x` a `MyClass` fent létrehozott példánya, akkor a következő kódrészlet a `16`-os értéket nyomtatja ki anélkül, hogy nyomot hagyna:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

A másik típusú példány attribútum hivatkozás egy *metódus*. A metódus egy függvény, amely egy objektumhoz „tartozik”. (A Pythonban a metódus kifejezés nem egyedi az osztálypéldányokra: más objektumtípusok is rendelkezhetnek metódusokkal. Például a listaobjektumoknak vannak hozzáfűzés,

beszúrás, eltávolítás, rendezés stb. nevű metódusai. A következő tárgyalásban azonban a metódus kifejezést kizárólag az osztálypéldány objektumok metódusaira fogjuk használni, hacsak nincs kifejezetten másként megadva.)

Egy példányobjektum érvényes metódusnevei az osztályától függenek. Definíció szerint egy osztály minden olyan attribútuma, amely függvényobjektum, meghatározza a példányainak megfelelő metódusait. Példánkban tehát az `x.f` egy érvényes metódushivatkozás, mivel a `MyClass.f` függvény, de az `x.i` nem, mivel a `MyClass.i` nem az. De az `x.f` nem ugyanaz, mint a `MyClass.f` – ez egy *metódus objektum*, nem egy függvény objektum.

9.3.4 Metódus objektumok

Általában egy metódust közvetlenül a kötés után hívnak meg:

```
x.f()
```

A MyClass példában ez a `'hello world'` karakterláncot adja vissza. Azonban nem szükséges azonnal meghívni egy metódust: az `x.f` egy metódusobjektum, amely eltárolható és később meghívható. Például:

```
xf = x.f
while True:
    print(xf())
```

az idők végezetéig továbbra is nyomtatni fogja a `hello world`-öt.

Mi történik pontosan egy metódus meghívásakor? Talán észrevette, hogy `az x.f()`-t a fenti argumentum nélkül hívták meg, annak ellenére, hogy az `f()` függvény definíciója argumentumot adott meg. Mi történt az argumentummal? A Python minden bizonnyal kivételt tesz, amikor egy argumentumot igénylő függvényt anélkül hívnak meg – még akkor is, ha az argumentumot valójában nem használják…

Valójában talán kitalálta a választ: a metódusok különlegessége, hogy a példányobjektum a függvény első argumentumaként kerül átadásra. Példánkban az `x.f()` hívás pontosan megegyezik a `MyClass.f(x)` hívással. Általában egy metódus *n*-es argumentumlistájával történő meghívása egyenértékű a megfelelő függvény olyan argumentumlistával történő meghívásával, amely a metódus példányobjektumának az első argumentum elé történő beszúrásával jön létre.

Ha még mindig nem érti a módszerek működését, egy pillantás a megvalósításra talán tisztázhatja a dolgokat. Ha egy példány nem adatattribútumára hivatkozik, akkor a példány osztályában történik a keresés. Ha a név egy érvényes osztályattribútumot jelöl, amely egy függvényobjektum, akkor egy metódusobjektum jön létre a példányobjektum és az éppen talált függvényobjektum összecsomagolásával (mutatókkal) egy absztrakt objektumban: ez a metódusobjektum. Amikor a metódusobjektumot argumentumlistával hívják meg, a példányobjektumból és az argumentumlistából új argumentumlista jön létre, és a függvényobjektum ezzel az új argumentumlistával kerül meghívásra.

9.3.5 Osztály- és példányváltozók

Általánosságban elmondható, hogy a példányváltozók az egyes példányok egyedi adataihoz, az osztályváltozók pedig az osztály összes példánya által megosztott attribútumokhoz és metódusokhoz valók:

```
class Dog:
    kind = 'canine' # osztályváltozó, amelyet minden példány megoszt

    def __init__(self, name):
        self.name = name # példányváltozó minden egyes példányra egyedi
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # minden kutya megosztja
'canine'
>>> e.kind # minden kutya megosztja
'canine'
>>> d.name # egyedi d-hez
'Fido'
>>> e.name # egyedi e-hez
'Buddy'
```

Amint azt az *Egy szó a nevekről és objektumokról* című részben tárgyaltuk, a megosztott adatoknak meglepő hatásai lehetnek a változó objektumok, például listák és szótárak bevonásával. Például a következő kódban található trükkök listája nem használható osztályváltozóként, mert csak egyetlen listát osztana meg az összes kutyapéldánnyal:

```
class Dog:
    tricks = [] # osztályváltozó hibás használata

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks # váratlanul megosztotta minden kutya
['roll over', 'play dead']
```

Az osztály helyes kialakítása egy példányváltozót kell inkább használni:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog
```

```
    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Véletlenszerű megjegyzések

Ha ugyanaz az attribútumnév előfordul egy példányban és egy osztályban is, akkor az attribútumkeresés prioritást ad a példánynak:

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Az adat attribútumokra metódusok és egy objektum hétköznapi felhasználói („kliensei”) egyaránt hivatkozhatnak. Más szavakkal, az osztályok nem használhatók tisztán absztrakt adattípusok megvalósítására. Valójában a Pythonban semmi sem teszi lehetővé az adatok elrejtésének kikényszerítését – mindez a konvención alapul. (Másrészt a C nyelven írt Python implementáció teljesen elrejti a megvalósítás részleteit, és szükség esetén szabályozza a hozzáférést egy objektumhoz; ezt használhatják a Python C nyelven írt kiterjesztései.)

A klienseknek óvatosan kell használniuk az adat attribútumokat – az ügyfelek összezavarhatják a metódusok által fenntartott invariánsokat azáltal, hogy rányomják az adat attribútumaikat. Ne feledje, hogy a kliensek saját adat attribútumokat adhatnak hozzá egy példányobjektumhoz anélkül, hogy befolyásolnák a metódusok érvényességét, mindaddig, amíg elkerülik a névütközéseket – az elnevezési konvenció itt is sok fejtörést takaríthat meg.

Nincs gyorsírás az adatattribútumokra (vagy más módszerekre!) a metódusokon belüli hivatkozásra. Azt tapasztalom, hogy ez tulajdonképpen növeli a metódusok olvashatóságát: nincs esély arra, hogy a lokális változókat és a példányváltozókat összekeverjük egy metóduson keresztül.

Gyakran egy mmetódus első argumentumát `self`-nek nevezik. Ez nem más, mint konvenció: a `self` névnek egyáltalán nincs különösebb jelentése a Python számára. Ne feledje azonban, hogy ha nem követi a konvenciót, a kódja kevésbé lesz olvasható más Python-programozók számára, és az is elképzelhető, hogy egy ilyen konvencióra támaszkodó *osztály böngésző* programot írnak.

Bármely függvényobjektum, amely osztályattribútum, meghatároz egy metódust az adott osztály példányaihoz. Nem szükséges, hogy a függvénydefiníció szövegesen be legyen zárva az osztálydefinícióba: az osztályban egy lokális változóhoz funkcióobjektum hozzárendelése is rendben van. Például:

```
# Az osztályon kívül definiált függvény
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Most `f`, `g` és `h` mind a `C` osztály attribútumai, amelyek függvényobjektumokra hivatkoznak, következésképpen mindegyik `C` példány metódusa – `h` pontosan ekvivalens `g`-vel. Vegye figyelembe, hogy ez a gyakorlat általában csak arra szolgál, hogy megzavarja a program olvasóját.

A metódusok más metódusokat is hívhatnak a `self` argumentum metódusattribútumainak használatával:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

A metódusok ugyanúgy hivatkozhatnak a globális nevekre, mint a közönséges függvények. A metódushoz tartozó globális hatókör a definícióját tartalmazó modul. (Egy osztályt soha nem használnak globális hatókörként.) Bár ritkán találkozunk jó okkal a globális adatok használatára egy metódusban, a globális hatókörnek számos jogos felhasználása létezik: egyrészt a globális hatókörbe importált függvények és modulok metódusok, valamint a benne definiált függvények és osztályok használhatják. Általában a metódust tartalmazó osztály maga is ebben a globális hatókörben van definiálva, és a következő részben találunk néhány jó okot arra, hogy egy metódus miért szeretne a saját osztályára hivatkozni.

Minden érték egy objektum, ezért van egy *osztálya* (más néven *típusa*). `object.__class__`-ként van tárolva.

9.5 Öröklés

Természetesen egy nyelvi jellemző nem érdemelné meg az „osztály” nevet az öröklődés támogatása nélkül. A származtatott osztálydefiníció szintaxisa így néz ki:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

A `BaseClassName` nevet a származtatott osztálydefiníciót tartalmazó hatókörből elérhető névtérben kell megadni. Az alaposztálynév helyett más tetszőleges kifejezések is megengedettek. Ez hasznos lehet például, ha az alaposztály egy másik modulban van definiálva:

```
class DerivedClassName(modname.BaseClassName):
```

A származtatott osztálydefiníció végrehajtása ugyanúgy megy végbe, mint egy alaposztály esetében. Az osztályobjektum felépítésekor a rendszer megjegyzi az alaposztályt. Ez az attribútum-hivatkozások feloldására szolgál: ha a kért attribútum nem található az osztályban, a keresés az alaposztályban folytatódik. Ezt a szabályt rekurzívan alkalmazzuk, ha maga az alaposztály egy másik osztályból származik.

A származtatott osztályok példányosításában nincs semmi különös: a `DerivedClassName()` létrehozza az osztály új példányát. A metódushivatkozások feloldása a következőképpen történik: a megfelelő osztályattribútum megkeresése, szükség esetén az alaposztályok láncolatán lefelé haladva, és a metódushivatkozás érvényes, ha ez függvényobjektumot eredményez.

A származtatott osztályok felülbírálhatják alaposztályaik metódusait. Mivel a metódusoknak nincs különleges jogosultsága ugyanazon objektum más metódusainak meghívásakor, egy alaposztály metódusa, amely egy másik, ugyanabban az alaposztályban definiált metódust hív meg, végül egy származtatott osztály metódusát hívhatja meg, amely felülírja azt. (C++ programozók számára: a Python összes metódusa gyakorlatilag `virtual`.)

Egy származtatott osztályban lévő felülbíráló metódus valójában kiterjeszteni kívánja az azonos nevű alaposztály metódusát, nem pedig egyszerűen lecserélni. Van egy egyszerű módja az alaposztály metódusának közvetlen meghívására: csak hívja meg a `BaseClassName.methodname(self, arguments)`. Ez időnként az ügyfelek számára is hasznos. (Ne feledje, hogy ez csak akkor működik, ha az alaposztály `BaseClassName` néven érhető el a globális hatókörben.)

A Pythonnak két beépített függvénye van, amelyek az örökléssel működnek:

- Az `isinstance()` segítségével ellenőrizheti a példány típusát: az `isinstance(obj, int)` csak akkor lesz `True`, ha az `obj.__class__` `int`, vagy valamilyen osztály az `int`-ből származik.

- Az `issubclass()` segítségével ellenőrizheti az osztály öröklődését: `issubclass(bool, int)` `True`, mivel a `bool` az `int` alosztálya. Az `issubclass(float, int)` azonban `False`, mivel a `float` nem az `int` alosztálya.

9.5.1 Többszörös öröklődés

A Python támogatja a többszörös öröklődés egy formáját is. A több alaposztályt tartalmazó osztálydefiníció így néz ki:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

A legtöbb esetben a legegyszerűbb esetben a szülőosztálytól örökölt attribútumok keresése mélység-első, balról jobbra irányú keresésre gondolhat, nem pedig kétszeri keresésre ugyanabban az osztályban, ahol átfedés van a hierarchiában. Így ha egy attribútum nem található a `DerivedClassName`-ben, akkor az `Base1`-ben, majd (rekurzívan) a `Base1` alaposztályaiban, és ha ott nem, akkor a `Base2`-ben, és így tovább.

Valójában ennél valamivel összetettebb; a metódusfeloldási sorrend dinamikusan változik, hogy támogassa a `super()` kooperatív hívásait. Ez a megközelítés néhány más többszörösen öröklődő nyelvben call-next-method néven ismert, és erősebb, mint az egyszeres öröklődésű nyelvekben található szuperhívás.

A dinamikus rendezés azért szükséges, mert a többszörös öröklődés minden esete egy vagy több gyémántkapcsolatot mutat (ahol legalább az egyik szülőosztály több útvonalon is elérhető a legalsó osztályból). Például minden osztály az objektumtól örököl, így minden többszörös öröklődés több útvonalat biztosít az objektum eléréséhez. Az alaposztályokhoz való többszöri hozzáférés elkerülése érdekében a dinamikus algoritmus linearizálja a keresési sorrendet oly módon, hogy megőrizze az egyes osztályokban megadott balról jobbra sorrendet, amely minden szülőt csak egyszer hív meg, és monoton (azaz egy osztály alosztályozható anélkül, hogy ez befolyásolná a szülők elsőbbségi sorrendjét). Ezek a tulajdonságok együttesen lehetővé teszik megbízható és bővíthető, többszörös öröklődésű osztályok tervezését.

További részletekért lásd: https://www.python.org/download/releases/2.3/mro/.

9.6 Privát változók

A Pythonban nem léteznek olyan „privát” példányváltozók, amelyekhez csak egy objektumról lehet hozzáférni. Van azonban egy konvenció, amelyet a legtöbb Python-kód követ: az aláhúzással előtagolt nevet (pl. `_spam`) az API nem nyilvános részeként kell kezelni (legyen szó függvényről, metódusról vagy adattagról). Megvalósítási részletnek kell tekinteni, és előzetes értesítés nélkül változhat.

Mivel létezik egy érvényes használati eset az osztály-privát tagok számára (nevezetesen az alosztályok által meghatározott nevekkel való névütközések elkerülése érdekében), korlátozott támogatás áll rendelkezésre egy ilyen mechanizmushoz, az úgynevezett „névkezelés”-hez. Bármely `__spam` formátumú azonosító (legalább két bevezető aláhúzás, legfeljebb egy záró aláhúzás) szövegesen lecserélődik a `_classname__spam` elemre, ahol az osztálynév az aktuális osztálynév, a bevezető aláhúzás(ok) nélkül. Ez a roncsolás az azonosító szintaktikai helyzetétől függetlenül történik, mindaddig, amíg az egy osztály definícióján belül történik.

A névkezelés segít abban, hogy az alosztályok felülírják a metódusokat az osztályon belüli metódushívások megszakítása nélkül. Például:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
        self.items_list.append(item)

    __update = update # az eredeti update() metódus privát másolata

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # új aláírást biztosít a update()-hez
        # de nem szakítja meg az __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

A fenti példa akkor is működne, ha a MappingSubclass bevezetne egy `__update` azonosítót, mivel azt a `Mapping` osztályban `_Mapping__update`, a `MappingSubclass` osztályban pedig `_MappingSubclass__update` váltja fel.

Vegye figyelembe, hogy a roncsolási szabályokat leginkább a balesetek elkerülésére tervezték; továbbra is lehetséges egy privátnak tekintett változó elérése vagy módosítása. Ez még különleges körülmények között is hasznos lehet, például a hibakeresőben.

Figyeljük meg, hogy az `exec()`-nek vagy `eval()`-nak átadott kód nem tekinti a meghívó osztály osztálynevét az aktuális osztálynak; ez hasonló a globális utasítás hatásához, amelynek hatása szintén a bájtokból összeállított kódra korlátozódik. Ugyanez a korlátozás vonatkozik a `getattr()`, `setattr()` és `delattr()`-ra, valamint a `__dict__` közvetlen hivatkozására.

9.7 Esélyek és végeredmények

Néha hasznos a Pascal „record”-hoz vagy a C „struct”-hoz hasonló adattípus, amely néhány megnevezett adatelemet köt össze. Az idiomatikus megközelítés az `dataclasses` használata erre a célra:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

A Python-kód egy darabja, amely egy adott absztrakt adattípust vár el, gyakran átadható egy olyan osztálynak, amely ehelyett az adott adattípus metódusait emulálja. Például, ha van egy függvénye, amely formázza az adatokat egy fájlobjektumból, akkor definiálhat egy osztályt `read()` és `readline()` metódusokkal, amelyek ehelyett egy string pufferből veszik az adatokat, és argumentumként adják át.

A példány metódusobjektumoknak is vannak attribútumai: az `m.__self__` az `m()` metódussal rendelkező példányobjektum, az `m.__func__` pedig a metódusnak megfelelő függvényobjektum.

9.8 Iterátorok

Mostanra valószínűleg észrevette, hogy a legtöbb konténerobjektum bejárható egy for utasítással:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Ez a hozzáférési stílus világos, tömör és kényelmes. Az iterátorok használata áthatja és egységesíti a Pythont. A színfalak mögött a `for` utasítás meghívja a konténerobjektum `iter()`-ét. A függvény egy iterátor objektumot ad vissza, amely meghatározza a `__next__()` metódust, amely egyenként

éri el a tároló elemeit. Ha nincs több elem, a `__next__() StopIteration` kivételt hoz létre, amely a `for` ciklus befejezését utasítja. A `__next__()` metódus a `next()` beépített függvény segítségével hívható meg; ez a példa bemutatja, hogyan működik mindez:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Az iterátorprotokoll mögött meghúzódó mechanika után könnyen hozzá lehet adni az iterátor viselkedését az osztályokhoz. Határozzon meg egy `__iter__()` metódust, amely egy objektumot a `__next__()` metódussal tér vissza. Ha az osztály meghatározza a `__next__()` értéket, akkor az `__iter__()` csak önmagát adja vissza:

```
class Reverse:
    """ Iterátor egy sorozat visszafelé történő bejárásához."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Generátorok

A *generátorok* egy egyszerű és hatékony eszköze az iterátorok létrehozásának. Úgy vannak megírva, mint a normál függvények, de a hozamkimutatást használják, amikor adatokat akarnak visszaadni. Minden alkalommal, amikor a `next()` meghívásra kerül, a generátor ott folytatja, ahol abbahagyta (megjegyzi az összes adatértéket és azt, hogy melyik utasítást hajtották végre utoljára). Egy példa azt mutatja, hogy a generátorokat triviálisan könnyű létrehozni:

```
def reverse(data):
for index in range(len(data)-1, -1, -1):
yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Bármi, amit generátorokkal meg lehet tenni, elvégezhető osztályalapú iterátorokkal is, az előző részben leírtak szerint. A generátorokat az teszi kompakttá, hogy az `__iter__()` és `__next__()` metódusok automatikusan létrejönnek.

Egy másik fontos jellemzője, hogy a helyi változók és a végrehajtási állapot automatikusan mentésre kerülnek a hívások között. Ez megkönnyítette a függvény írását, és sokkal áttekinthetőbbé tette, mint az olyan példányváltozókat használó megközelítés, mint a `self.index` és a `self.data`.

Az automatikus metódus létrehozása és a programállapot mentése mellett, amikor a generátorok leállnak, automatikusan felveszik a StopIteration-t. Ezek a szolgáltatások együttesen megkönnyítik az iterátorok létrehozását, nem több erőfeszítéssel, mint egy normál függvény írása.

9.10 Generátorkifejezések

Egyes egyszerű generátorok tömören kódolhatók kifejezésként, a listaértelmezésekhez hasonló szintaxis használatával, de szögletes zárójelek helyett kerek zárójelekkel. Ezeket a kifejezéseket olyan helyzetekre tervezték, amikor a generátort egy befoglaló függvény azonnal használja. A generátor kifejezések kompaktabbak, de kevésbé sokoldalúak, mint a teljes generátordefiníciók, és általában memóriabarátabbak, mint az egyenértékű listaértelmezések.

Példák:

```
>>> sum(i*i for i in range(10))                 # négyzetek összege
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))      # szorzat összeg
```

```
260
>>> unique_words = set(word for line in page for word in line.split())
>>> valedictorian = max((student.gpa, student.name) for student in
graduates)
>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

10. Rövid utazás a Standard könyvtárban

10.1 Operációs rendszer interfész

Az `os` modul több tucat funkciót biztosít az operációs rendszerrel való interakcióhoz:

```
>>> import os
>>> os.getcwd()  # Az aktuális könyvtár visszaadása
'C:\\Python311'
>>> os.chdir('/server/accesslogs') # Az aktuális könyvtár módosítása
>>> os.system('mkdir today') # Futtassa az mkdir parancsot a rendszerhéjban
0
```

Ügyeljen arra, hogy az `import os` stílust használja az `os import *` helyett. Ez megakadályozza, hogy az `os.open()` árnyékolja a beépített `open()` függvényt, amely nagyon eltérően működik.

A beépített `dir()` és `help()` függvények hasznosak interaktív segédeszközként a nagy modulokkal, például az `os` modullal való munkavégzéshez:

```
>>> import os
>>> dir(os)
<visszaadja az összes modulfüggvény listáját>
>>> help(os)
<egy kiterjedt kézikönyvoldalt ad vissza, amelyet a modul
docstringjeiből hoztak létre>
```

A napi fájl- és könyvtárkezelési feladatokhoz a `shutil` modul magasabb szintű interfészt biztosít, amely könnyebben használható:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Fájl helyettesítő karakterek

A `glob` modul egy funkciót biztosít fájllisták készítéséhez a könyvtárban végzett helyettesítő karakteres keresésekből:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Parancssori argumentumok

Az átlagos segédprogram-szkripteknek gyakran parancssori argumentumokat kell feldolgozniuk. Ezeket az argumentumokat a rendszer a sys modul argv attribútumában tárolja listaként. Vegyük például a következő `demo.py` fájlt:

```
# File demo.py
import sys
print(sys.argv)
```

Íme a parancssorban a `python demo.py one two three` futtatásának kimenete:

```
['demo.py', 'one', 'two', 'three']
```

Az `argparse` modul kifinomultabb mechanizmust biztosít a parancssori argumentumok feldolgozásához. A következő szkript kibont egy vagy több fájlnevet és opcionális számú megjelenítendő sort:

```
import argparse
parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Ha a parancssorban futtatja a `python top.py --lines=5 alpha.txt beta.txt` parancsot, a szkript az `args.lines` értéket `5`-re, az `args.filenames`-et pedig az `['alpha.txt', 'beta.txt']` értékre állítja be.

10.4 Hibakimenet átirányítása és programlezárás

A `sys` modul attribútumokkal is rendelkezik az *stdin*, *stdout* és *stderr* számára. Ez utóbbi hasznos figyelmeztetések és hibaüzenetek kibocsátására, hogy láthatóak legyenek még az *stdout* átirányítása esetén is:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

A szkript leállításának legközvetlenebb módja a `sys.exit()` használata.

10.5 String sablon illesztés

A `re` modul szabályos kifejezési eszközöket biztosít a fejlett karakterlánc-feldolgozáshoz. Az összetett illesztéshez és manipulációhoz a szabályos kifejezések tömör, optimalizált megoldásokat kínálnak:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
```

```
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Ha csak egyszerű képességekre van szükség, a karakterlánc-módszereket részesítjük előnyben, mert könnyebben olvashatók és hibakereshetőek:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matematika

A matematikai modul hozzáférést biztosít az alapul szolgáló C könyvtári függvényekhez a lebegőpontos matematikához:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

A `random` modul eszközöket biztosít a véletlenszerű kiválasztáshoz:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # mintavétel csere nélkül
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()      # véletlenszerű float érték
0.17970987693706186
>>> random.randrange(6) # tartományból választott véletlenszerű
...                     # egész szám: range(6)
4
```

A `statistics` modul kiszámítja a numerikus adatok alapvető statisztikai tulajdonságait (átlag, medián, variancia stb.):

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

A SciPy projekt <`https://scipy.org`> számos más modult tartalmaz a numerikus számításokhoz.

10.7 Internet-hozzáférés

Számos modul létezik az internet eléréséhez és az internetes protokollok feldolgozásához. A legegyszerűbbek közül kettő az `urllib.request` az adatok URL-ekből való lekéréséhez és az `smtplib` a levélküldéshez:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # byte-ok str-re konvertálása
...         if line.startswith('datetime'):
...             print(line.rstrip())         # A záró újsor eltávolítása
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Ne feledje, hogy a második példában egy localhost-on futó levelezőszerver szükséges.)

10.8 Dátumok és időpontok

A `datetime` modul osztályokat biztosít a dátumok és időpontok egyszerű és összetett módon történő kezeléséhez. Míg a dátum és idő aritmetika támogatott, a megvalósítás középpontjában a kimenet formázásához és manipulálásához szükséges hatékony tagkinyerés áll. A modul olyan objektumokat is támogat, amelyek ismerik az időzónát.

```
>>> # a dátumok könnyen összeállíthatók és formázhatók
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # a dátumok támogatják a naptári aritmetikát
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Adattömörítés

Az általános adatarchiválási és -tömörítési formátumokat közvetlenül támogatják a következő modulok: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` és `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Teljesítménymérés

Egyes Python-felhasználók mély érdeklődést mutatnak az azonos probléma, különböző megközelítéseinek relatív teljesítményének ismerete iránt. A Python olyan mérőeszközt biztosít, amely azonnal válaszol ezekre a kérdésekre.

Például csábító lehet a sor be- és kicsomagolása funkció használata az argumentumok felcserélésének hagyományos megközelítése helyett. A `timeit` modul gyorsan bemutat egy szerény teljesítményelőnyt:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

A `timeit` finom részletességi szintjével ellentétben a `profil` és a `pstats` modulok eszközöket biztosítanak a nagyobb kódblokkok időkritikus szakaszainak azonosításához.

10.11 Minőség-ellenőrzés

A kiváló minőségű szoftverek fejlesztésének egyik módja az, hogy minden egyes funkcióhoz teszteket írnak a fejlesztés során, és ezeket a teszteket gyakran lefuttatják a fejlesztési folyamat során.

A `doctest` modul eszközt biztosít a modulok szkennelésére és a program docstringjeibe ágyazott tesztek érvényesítésére. A teszt felépítése olyan egyszerű, mint egy tipikus hívás kivágása és beillesztése az eredményekkel együtt a dokumentumkarakterláncba. Ez javítja a dokumentációt azáltal, hogy

példát ad a felhasználónak, és lehetővé teszi a doctest modul számára, hogy megbizonyosodjon arról, hogy a kód hű marad a dokumentációhoz:

```
def average(values):
    """Kiszámítja egy számlista számtani átlagát.
    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)
import doctest
doctest.testmod()  # automatikusan érvényesíti a beágyazott teszteket
```

A `unittest` modul nem olyan egyszerű, mint a doctest modul, de lehetővé teszi a tesztek átfogóbb készletének külön fájlban való karbantartását:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # A parancssorból történő hívás minden tesztet meghív
```

10.12 Elemeket tartalmaz

A Pythonnak van egy „elemeket tartalmaz” filozófiája. Ez a legjobban a nagyobb csomagok kifinomult és robusztus képességein keresztül látható. Például:

- Az `xmlrpc.client` és `xmlrpc.server` modulok a távoli eljáráshívások megvalósítását szinte triviális feladattá teszik. A modulok neve ellenére nincs szükség az XML közvetlen ismeretére vagy kezelésére.
- Az `email` csomag egy könyvtár az e-mail üzenetek kezelésére, beleértve a MIME-t és más RFC 2822-alapú üzenetdokumentumokat. Ellentétben az `smtplib` és `poplib` programokkal, amelyek ténylegesen küldenek és fogadnak üzeneteket, az e-mail csomag teljes eszközkészlettel rendelkezik összetett üzenetstruktúrák (beleértve a mellékleteket) felépítéséhez vagy dekódolásához, valamint az internetes kódolási és fejlécprotokollok megvalósításához.
- A `json` csomag erőteljes támogatást nyújt ennek a népszerű adatcsere-formátumnak az elemzéséhez. A `csv` modul támogatja a fájlok vesszővel elválasztott érték formátumú közvetlen olvasását és írását, amelyet általában az adatbázisok és a táblázatok támogatnak. Az XML-feldolgozást az `xml.etree.ElementTree`, `xml.dom` és `xml.sax` csomagok tá-

mogatják. Ezek a modulok és csomagok együttesen nagymértékben leegyszerűsítik a Python-alkalmazások és más eszközök közötti adatcserét.

- Az `sqlite3` modul az SQLite adatbázis-könyvtár egy burkolója, amely állandó adatbázist biztosít, amely frissíthető és kissé nem szabványos SQL szintaxissal érhető el.
- A nemzetközivé tételt számos modul támogatja, köztük a `gettext`, a `locale` és a `kodecs` csomag.

11. Rövid utazás a Standard könyvtárban – 2. rész

Ez a második körút olyan fejlettebb modulokat fed le, amelyek támogatják a professzionális programozási igényeket. Ezek a modulok ritkán fordulnak elő kis szkriptekben.

11.1 Kimenet formázása

A `reprlib` modul a `repr()` egy olyan verzióját biztosítja, amely a nagy vagy mélyen beágyazott tárolók rövidített megjelenítésére van testreszabva:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

A `pprint` modul kifinomultabb vezérlést kínál a beépített és a felhasználó által definiált objektumok interpreter által olvasható módon történő nyomtatásához. Ha az eredmény egy sornál hosszabb, a „prety printer” sortöréseket és behúzást ad hozzá, hogy tisztábban tárja fel az adatstruktúrát:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']],
['magenta',
... 'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
    'white',
    ['green', 'red']],
   [['magenta', 'yellow'],
    'blue']]]
```

A `textwrap` modul úgy formázza a szöveg bekezdéseit, hogy illeszkedjenek egy adott képernyőszélességhez:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it
returns
... a list of strings instead of one big string with newlines to
separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

A `locale` modul hozzáfér a kultúraspecifikus adatformátumok adatbázisához. A locale formázási funkciójának csoportosítási attribútuma közvetlen módot biztosít a számok csoportelválasztókkal történő formázására:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()  # megkapja a konvenciók feltérképezését
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Sablonozás

A `string` modul egy sokoldalú `Template` osztályt tartalmaz, egyszerűsített szintaxissal, amely alkalmas a végfelhasználók általi szerkesztésre. Ez lehetővé teszi a felhasználók számára, hogy az alkalmazás módosítása nélkül testreszabják alkalmazásaikat.

A formátum a `$` által alkotott helyőrző neveket használja érvényes Python-azonosítókkal (alfanumerikus karakterek és aláhúzásjelek). A helyőrző kapcsos zárójelekkel körülvéve lehetővé teszi, hogy több alfanumerikus betű kövessen szóközök nélkül. A `$$` írása egyetlen megtisztított `$`-t hoz létre:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

A `substitute()` metódus `KeyError`-t vet fel, ha a szótárban vagy a kulcsszó argumentumában nincs helyőrző. A körlevél stílusú alkalmazásoknál a felhasználó által megadott adatok hiányosak lehetnek, és a `safe_substitute()` metódus megfelelőbb lehet – a helyőrzőket változatlanul hagyja, ha adatok hiányoznak:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

A sablon alosztályok egyéni határolót is megadhatnak. Például egy fotóböngésző kötegelt átnevezési segédprogramja dönthet úgy, hogy százalékjeleket használ a helyőrzőkhöz, például az aktuális dátumhoz, képsorszámhoz vagy fájlformátumhoz:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):
')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

A sablonozás másik alkalmazása a programlogikának a több kimeneti formátum részleteitől való elkülönítése. Ez lehetővé teszi az XML-fájlok, az egyszerű szöveges jelentések és a HTML webjelentések egyéni sablonok helyettesítését.

11.3 Bináris adatrekord-elrendezések használata

A `struct` modul `pack()` és `unpack()` függvényeket biztosít a változó hosszúságú bináris rekordformátumok kezeléséhez. A következő példa bemutatja, hogyan lehet a fejléc-információkat végigfutni egy ZIP-fájlban a `zipfile` modul használata nélkül. A "`H`" és az "`I`" csomagkód két, illetve négy bájtos előjel nélküli számokat jelent. A "`<`" azt jelzi, hogy szabványos méretűek és kis-endian bájt sorrendben vannak:

```
import struct
with open('myfile.zip', 'rb') as f:
    data = f.read()
start = 0
for i in range(3):  # az első 3 fájl fejlécének megjelenítése
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)
    start += extra_size + comp_size  # ugorjon a következő fejlécre
```

11.4 Többszálúság

A szálképzés a nem szekvenciálisan függő feladatok szétválasztására szolgáló technika. A szálak segítségével javítható azon alkalmazások válaszkészsége, amelyek elfogadják a felhasználói bevitelt, miközben más feladatok a háttérben futnak. Egy kapcsolódó használati eset az I/O párhuzamos futtatása egy másik szál számításaival.

A következő kód megmutatja, hogy a magas szintű `threading` modul hogyan tud feladatokat futtatni a háttérben, miközben a fő program fut tovább:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)
background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

A többszálú alkalmazások fő kihívása az adatokat vagy más erőforrásokat megosztó szálak koordinálása. Ebből a célból a `threading` modul számos szinkronizálási primitívet biztosít, beleértve a zárakat, eseményeket, feltételváltozókat és szemaforokat.

Bár ezek az eszközök hatékonyak, a kisebb tervezési hibák nehezen reprodukálható problémákat okozhatnak. Tehát a feladatkoordináció előnyben részesített megközelítése az, hogy egy erőforráshoz való összes hozzáférést egyetlen szálba koncentrálják, majd a `queue` modul segítségével táplálják be a szálat más szálaktól származó kérelmekkel. A szálak közötti kommunikációhoz és koordinációhoz `Queue` objektumokat használó alkalmazások könnyebben tervezhetők, olvashatóbbak és megbízhatóbbak.

11.5 Naplózás

A `logging` modul teljes körű és rugalmas naplózási rendszert kínál. A legegyszerűbb esetben a naplóüzenetek egy fájlba vagy a `sys.stderr` címre kerülnek:

```
import logging
logging.debug('Debugging information')
```

```
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Ez a következő kimenetet eredményezi:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Alapértelmezés szerint az információs és hibakeresési üzenetek le vannak tiltva, és a kimenet normál hibába kerül. Az egyéb kimeneti lehetőségek közé tartozik az üzenetek e-mailen, datagramokon, socketeken vagy HTTP-kiszolgálóra történő irányítása. Az új szűrők az üzenet prioritása alapján különböző útvonalakat választhatnak: `DEBUG`, `INFO`, `WARNING`, `ERROR` és `CRITICAL`.

A naplózási rendszer konfigurálható közvetlenül a Pythonból, vagy betölthető egy felhasználó által szerkeszthető konfigurációs fájlból a testreszabott naplózáshoz az alkalmazás módosítása nélkül.

11.6 Gyenge hivatkozások

A Python automatikus memóriakezelést végez (a legtöbb objektum referenciaszámlálása és a szemétgyűjtés a ciklusok kiküszöbölése érdekében). A memória röviddel azután szabadul fel, hogy az utolsó hivatkozást törölték.

Ez a megközelítés a legtöbb alkalmazásnál jól működik, de esetenként csak addig kell nyomon követni az objektumokat, amíg azokat valami más használja. Sajnos a követésük egy hivatkozást hoz létre, amely állandósítja őket. A `weakref` modul eszközöket biztosít az objektumok nyomon követéséhez hivatkozás létrehozása nélkül. Amikor az objektumra már nincs szükség, automatikusan eltávolítják a gyenge refekciós táblából, és visszahívás indul a gyenge referencia objektumokra. A tipikus alkalmazások közé tartoznak az objektumok gyorsítótárazása, amelyek létrehozása költséges:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # hivatkozás létrehozzása
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # nem hoz létre hivatkozást
>>> d['primary']       # elkapja az objektumot, ha még életben van
10
>>> del a                       # törli a hivatkozást
>>> gc.collect()       # azonnal indítsa el a szemétszállítást
0
```

```
>>> d['primary']    # a bejegyzés automatikusan el lett távolítva
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']    # a bejegyzés automatikusan el lett távolítva
  File "C:/python311/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Eszközök a listákkal való munkához

A beépített listatípussal számos adatszerkezeti igény kielégíthető. Néha azonban szükség van alternatív megvalósításokra, amelyek különböző teljesítmény-kompromisszumokkal rendelkeznek.

Az `array` modul egy `array()` objektumot biztosít, amely olyan, mint egy lista, amely csak homogén adatokat tárol, és tömörebben tárolja azokat. A következő példa két bájtos előjel nélküli bináris számként ("`H`" típuskód) tárolt számok tömbjét mutatja be, a Python `int` objektumok szokásos listáinál szokásos bejegyzésenkénti 16 bájt helyett:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

A `collections` modul egy `deque()` objektumot biztosít, amely olyan, mint egy lista, gyorsabb hozzáfűzéssel és felbukkanással a bal oldalon, de lassabb keresésekkel a közepén. Ezek az objektumok kiválóan alkalmasak várólisták megvalósítására és szélességi első fa keresésekre:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Az alternatív lista-megvalósítások mellett a könyvtár egyéb eszközöket is kínál, például a `bisect` modult, amely a rendezett listák kezeléséhez szükséges funkciókat tartalmazza:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

A `heapq` modul a rendezett listákon alapuló kupacok megvalósításához nyújt funkciókat. A legalacsonyabb értékű bejegyzés mindig a nulla pozícióban marad. Ez olyan alkalmazásoknál hasznos, amelyek ismételten hozzáférnek a legkisebb elemhez, de nem akarnak teljes listarendezést futtatni:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rendezze át a listát kupacrendbe
>>> heappush(data, -5)                 # új bejegyzés hozzáadása
>>> [heappop(data) for i in range(3)] # lekéri a három legkisebb
                                       # bejegyzést
[-5, 0, 1]
```

11.8 Decimális lebegőpontos aritmetika

A `decimal` modul egy `Decimal` adattípust kínál a decimális lebegőpontos aritmetikához. A bináris lebegőpontos beépített lebegőpontos megvalósításhoz képest az osztály különösen hasznos

- pénzügyi alkalmazások és egyéb olyan felhasználások, amelyek pontos decimális ábrázolást igényelnek,
- a pontosság ellenőrzése,
- a kerekítés ellenőrzése a jogi vagy szabályozási követelmények teljesítése érdekében,
- jelentős tizedesjegyek követése, ill
- olyan alkalmazások, ahol a felhasználó azt várja, hogy az eredmények megegyezzenek a kézzel végzett számításokkal.

Például egy 70 centes telefondíj 5%-os adójának kiszámítása eltérő eredményeket ad decimális lebegőpontos és bináris lebegőpontos értékben. A különbség akkor válik jelentőssé, ha az eredményeket a legközelebbi centre kerekítjük:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

A `Decimal` eredmény egy záró nullát tart, és automatikusan négyhelyes szignifikancia következtet a kéthelyes szignifikancia szorzóiból. A decimális kézzel reprodukálja a matematikát, és elkerüli azokat a problémákat, amelyek akkor merülhetnek fel, ha a bináris lebegőpont nem képes pontosan ábrázolni a decimális mennyiségeket.

A pontos ábrázolás lehetővé teszi, hogy a `Decimal` osztály moduló számításokat és egyenlőségi teszteket hajtson végre, amelyek nem alkalmasak bináris lebegőpontos használatára:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

A `decimal` modul a szükséges pontossággal biztosítja az aritmetikát:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12. Virtuális környezetek és csomagok

12.1 Bevezetés

A Python alkalmazások gyakran olyan csomagokat és modulokat használnak, amelyek nem a szabványos könyvtár részei. Az alkalmazásoknak néha szükségük van egy adott könyvtárverzióra, mivel az alkalmazás megkövetelheti egy adott hiba kijavítását, vagy előfordulhat, hogy az alkalmazást a könyvtár felületének egy elavult verziójával írták meg.

Ez azt jelenti, hogy előfordulhat, hogy egy Python-telepítés nem felel meg minden alkalmazás követelményeinek. Ha az A alkalmazásnak egy adott modul 1.0-s verziójára van szüksége, de a B-nek a 2.0-s verzióra, akkor a követelmények ütköznek, és az 1.0-s vagy 2.0-s verzió telepítése esetén az egyik alkalmazás nem tud futni.

A probléma megoldása egy virtuális környezet létrehozása, egy önálló címtárfa, amely a Python egy adott verziójához tartozó Python-telepítést, valamint számos további csomagot tartalmaz.

A különböző alkalmazások ezután különböző virtuális környezeteket használhatnak. Az ütköző követelmények korábbi példájának megoldása érdekében az A alkalmazás rendelkezhet saját virtuális környezettel 1.0-s verzióval, míg a B alkalmazásnak egy másik, 2.0-s verziójú virtuális környezete lehet. Ha a B alkalmazás megköveteli a könyvtár frissítését a 3.0-s verzióra, ez megtörténik nem befolyásolja az A alkalmazás környezetét.

12.2 Virtuális környezetek létrehozása

A virtuális környezetek létrehozására és kezelésére használt modul neve `venv`. A `venv` általában a Python legfrissebb verzióját telepíti, amely elérhető. Ha a rendszerén több Python-verzió is található, kiválaszthat egy adott Python-verziót a `python3` futtatásával vagy a kívánt verzióval.

Virtuális környezet létrehozásához válassza ki a könyvtárat, ahová el szeretné helyezni, és futtassa a `venv` modult szkriptként a könyvtár elérési útjával:

```
python -m venv tutorial-env
```

Ez létrehozza a `tutorial-env` könyvtárat, ha az nem létezik, és a benne lévő könyvtárakat is létrehozza, amelyek a Python interpreter másolatát és a különböző támogató fájlokat tartalmazzák.

A virtuális környezet általános helye a `.venv`. Ez a név általában rejtve tartja a könyvtárat a shellben, és így távol tartja az útból, miközben olyan nevet ad neki, amely megmagyarázza a könyvtár létezését. Ezen kívül megakadályozza az ütközést az egyes eszközök által támogatott `.env` környezeti változódefiníciós fájlokkal.

Miután létrehozott egy virtuális környezetet, aktiválhatja azt.

Windows rendszeren futtassa:

```
tutorial-env\Scripts\activate.bat
```

Unix vagy MacOS rendszeren futtassa:

```
source tutorial-env/bin/activate
```

(Ez a szkript a bash parancsértelmezőhöz készült. Ha **csh**-t vagy **fish** shell-t használ, van alternatív `activate.csh` és `activate.fish` szkript, amelyet helyette érdemes használnia.)

A virtuális környezet aktiválása megváltoztatja a shell parancssorát, amely megmutatja, hogy milyen virtuális környezetet használ, és módosítja a környezetet úgy, hogy a python futtatása megkapja a Python adott verzióját és telepítését. Például:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

A virtuális környezet kikapcsolásához írja be:

```
deactivate
```

a terminálablakba.

12.3 Csomagok kezelése pip segítségével

A **pip** nevű programmal csomagokat telepíthet, frissíthet és távolíthat el. A `pip` alapértelmezés szerint a *Python Package Index* csomagjait telepíti. A Python Package Indexet úgy böngészheti, ha megnyitja azt a webböngészőjében.

A `pip` számos alparancsot tartalmaz: „install”, „uninstall”, „freeze” stb. (A pip teljes dokumentációjához tekintse meg a telepítési index útmutatót.)

A csomag legfrissebb verzióját a csomag nevének megadásával telepítheti:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Telepítheti a csomag egy adott verzióját is, ha megadja a csomag nevét, ezt követi az == és a verziószám:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Ha újra futtatja ezt a parancsot, a `pip` észreveszi, hogy a kért verzió már telepítve van, és nem tesz semmit. Megadhat egy másik verziószámot a verzió beszerzéséhez, vagy futtassa a `python -m pip install --upgrade` parancsot, hogy a csomagot a legújabb verzióra frissítse:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

A `python -m pip uninstall` után egy vagy több csomagnév eltávolítja a csomagokat a virtuális környezetből.

A `python -m pip show` információkat jelenít meg egy adott csomagról:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-
packages
Requires:
```

A `python -m pip list` megjeleníti a virtuális környezetbe telepített összes csomagot:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

A `python -m pip freeze` hasonló listát készít a telepített csomagokról, de a kimenet a `python -m pip install` által elvárt formátumot használja. Általános szokás szerint ezt a listát a `requirements.txt` fájlba helyezik:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

A `requirements.txt` ezután a verziófelügyelet alá helyezhető, és egy alkalmazás részeként szállítható. A felhasználók ezután telepíthetik az összes szükséges csomagot az `install -r` parancscsal:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

A `pip` sokkal több lehetőséget kínál. Tekintse meg a telepítési index útmutatót a `pip` teljes dokumentációjához. Ha írt egy csomagot, és szeretné elérhetővé tenni a Python Package Indexen, olvassa el a Python csomagolás felhasználói útmutatóját.

13. És most?

Ennek az oktatóanyagnak az elolvasása valószínűleg megerősítette érdeklődését a Python használata iránt – szívesen alkalmazza a Pythont a valós problémák megoldására. Hol érdemes többet megtudni?

Ez az oktatóanyag a Python dokumentációs készletének része. Néhány további dokumentum a készletben:

- library-index: Böngéssze át ezt a kézikönyvet, amely teljes (bár tömör) referenciaanyagot tartalmaz a típusokról, funkciókról és a szabványos könyvtár moduljairól. A szabványos Python disztribúció sok további kódot tartalmaz. Vannak modulok a Unix postafiókok olvasásához, dokumentumok HTTP-n keresztüli lekéréséhez, véletlen számok generálásához, parancssori opciók elemzéséhez, adatok tömörítéséhez és sok más feladathoz. A Library Reference áttekintésével képet kaphat arról, hogy mi áll rendelkezésre.

- Az installing-index elmagyarázza, hogyan telepíthet további Python-felhasználók által írt modulokat.

- referencia-index: A Python szintaxisának és szemantikájának részletes magyarázata. Nehéz olvasmány, de hasznos a nyelv teljes útmutatójaként.

További Python-források:

- https://www.python.org: A fő Python webhely. Kódot, dokumentációt és mutatókat tartalmaz a Pythonnal kapcsolatos oldalakra az interneten.

- https://docs.python.org: Gyors hozzáférés a Python dokumentációjához.

- https://pypi.org: A Python Package Index, amelyet korábban Cheese Shop1-nek is neveztek, a felhasználók által létrehozott Python modulok indexe, amelyek letölthetők. Amint elkezdi kiadni a kódot, regisztrálhatja itt, hogy mások is megtalálják.

- https://code.activestate.com/recipes/langs/python/: A Python Cookbook kódpéldák, nagyobb modulok és hasznos szkriptek jelentős gyűjteménye. Különösen figyelemre méltó hozzájárulásokat gyűjtött össze a Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3) című könyv.

- A https://pyvideo.org összegyűjti a Pythonnal kapcsolatos videók linkjeit konferenciákról és felhasználói csoportos értekezletekről.

- https://scipy.org: A Scientific Python projekt modulokat tartalmaz a gyors tömbszámításokhoz és -manipulációkhoz, valamint egy sor csomagot olyan dolgokhoz, mint a lineáris algebra, Fourier-transzformációk, nemlineáris megoldók, véletlenszám-eloszlások, statisztikai elemzés és hasonlók.

A Pythonnal kapcsolatos kérdéseket és problémajelentéseket felteheti a comp.lang.python hírcsoportba, vagy elküldheti a python-list@python.org címen található levelezőlistára. A hírcsoport és a levelezőlista átjárós, így az egyiknek elküldött üzenetek automatikusan átkerülnek a másikhoz. Naponta több száz bejegyzés érkezik, kérdéseket tesznek fel (és válaszolnak), új funkciókat javasolnak, és új modulokat jelentenek be. A levelezési listák archívumai a https://mail.python.org/pipermail/ címen érhetők el.

A közzététel előtt feltétlenül ellenőrizze a Gyakran Ismételt Kérdések listáját (más néven GYIK). A GYIK választ ad számos újra és újra felmerülő kérdésre, és már tartalmazhatja a megoldást a problémájára.

14. Interaktív beviteli szerkesztés és előzmények helyettesítése

A Python értelmező egyes verziói támogatják az aktuális beviteli sor szerkesztését és az előzmények helyettesítését, hasonlóan a Korn és a GNU Bash shellben található lehetőségekhez. Ez a GNU Readline könyvtár segítségével valósítható meg, amely támogatja a különféle szerkesztési stílusokat. Ennek a könyvtárnak saját dokumentációja van, amelyet itt nem fogunk sokszorosítani.

14.1 Tab befejezés és előzmények szerkesztése

A változó- és modulnevek befejezése automatikusan engedélyezve van az értelmező indításakor, így a `Tab` billentyű meghívja a befejezési funkciót; megvizsgálja a Python utasításneveket, az aktuális helyi változókat és a rendelkezésre álló modulneveket. A pontozott kifejezéseknél, mint például a `string.a`, a kifejezést a végső `"."` majd kiegészítéseket javasol az eredményül kapott objektum attribútumaiból. Vegye figyelembe, hogy ez alkalmazás által definiált kódot hajthat végre, ha egy `__getattr__()` metódussal rendelkező objektum a kifejezés része. Az alapértelmezett konfiguráció az előzményeket egy `.python_history` nevű fájlba is menti a felhasználói könyvtárában. Az előzmények a következő interaktív interpreter munka során újra elérhetőek lesznek.

14.2 Az interaktív interpreter alternatívái

Ez a lehetőség óriási előrelépést jelent az interpreter korábbi verzióihoz képest; azonban néhány kívánság maradt: Jó lenne, ha a megfelelő behúzást javasolnák a folytatásos sorokon (az elemző tudja, hogy ezután szükséges-e behúzás token). A befejezési mechanizmus használhatja az értelmező szimbólumtáblázatát. Hasznos lenne egy parancs is, amely ellenőrzi (vagy akár javasolja) a megfelelő zárójeleket, idézőjeleket stb.

Az egyik alternatív továbbfejlesztett interaktív értelmező, amely már régóta létezik, az IPython, amely tab befejezést, objektumfeltárást és fejlett előzmények kezelést kínál. Alaposan testreszabható és más alkalmazásokba is beágyazható. Egy másik hasonló továbbfejlesztett interaktív környezet a bpython.

15. Lebegőpontos aritmetika: kiadások és korlátozások

A lebegőpontos számokat a számítógépes hardver 2-es (bináris) törtként ábrázolja. Például a `0.625` tizedes tört értéke 6/10 + 2/100 + 5/1000, és ugyanígy a `0.101` **bináris** tört értéke 1/2 + 0/4 + 1/8. Ennek a két törtnek azonos az értéke, az egyetlen valódi különbség az, hogy az elsőt a 10-es, a másodikat a 2-es számrendszerben írják.

Sajnos a legtöbb tizedes tört nem ábrázolható pontosan bináris törtként. Ennek az a következménye, hogy általában a beírt decimális lebegőpontos számokat csak közelítik a gépben ténylegesen tárolt bináris lebegőpontos számok.

A probléma először könnyebben érthető a 10. számrendszerben. Tekintsük az 1/3-as törtet. Megközelítheti ezt tizedes törtként:

```
0.3
```

vagy jobb,

```
0.33
```

vagy jobb,

```
0.333
```

stb. Nem számít, hány számjegyet vagy hajlandó leírni, az eredmény soha nem lesz pontosan 1/3, hanem egyre jobb 1/3 közelítés.

Ugyanígy, függetlenül attól, hogy hány 2-es számrendszerű számjegyet szeretne használni, a 0.1-es tizedesértéket nem lehet pontosan kettedes törtként ábrázolni. A 2-es számrendszerben az 1/10 egy végtelenül ismétlődő tört

```
0.0001100110011001100110011001100110011001100110011...
```

Álljon meg tetszőleges véges számú bitnél, és kap egy közelítést. A legtöbb mai gépen a lebegő értékeket bináris törttel közelítik meg, a számláló az első 53 bitet használja a legjelentősebb bittel kezdődően, a nevező pedig kettő hatványa. 1/10 esetén a bináris tört `3602879701896397 / 2 ** 55`, amely közel áll, de nem pontosan egyenlő az 1/10 valódi értékével.

Sok felhasználó nem ismeri a közelítést az értékek megjelenítési módja miatt. A Python csak egy decimális közelítést nyomtat a gép által tárolt bináris közelítés valódi decimális értékéhez. A legtöbb gépen, ha a Python a 0.1-hez tárolt bináris közelítés valódi decimális értékét nyomtatná ki, akkor meg kellene jelenítenie

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Ez több számjegy, mint amennyit a legtöbb ember hasznosnak talál, ezért a Python kezelhetően tartja a számjegyek számát egy kerekített érték megjelenítésével.

```
>>> 1 / 10
0.1
```

Ne feledje, hogy bár a nyomtatott eredmény pontosan 1/10-nek tűnik, a tényleges tárolt érték a legközelebbi reprezentálható bináris tört.

Érdekes módon sok különböző decimális szám van, amelyek ugyanazon a legközelebbi közelítő bináris törten osztoznak. Például a `0.1` és a `0.10000000000000001` és a `0.1000000000000000055511151231257827021181583404541015625` értéke közelítőleg `3602879701896397 / 2 ** 55`. Mivel ezeknek a decimális értékeknek ugyanaz a közelítése, bármelyikük megjeleníthető az invariáns `eval(repr(x)) == x`-szel.

Korábban a Python prompt és a beépített `repr()` függvény a 17 szignifikáns számjegyet választotta, a `0.10000000000000001`-et. A Python 3.1-től kezdődően a Python (a legtöbb rendszeren) most már képes kiválasztani ezek közül a legrövidebbet, és egyszerűen megjeleníteni a `0.1`-et.

Ne feledje, hogy ez a bináris lebegőpontos számábrázolás természetéből fakad: ez nem a Python hibája, és nem is az Ön kódjának. Ugyanezt fogja látni minden olyan nyelven, amely támogatja a hardver lebegőpontos aritmetikáját (bár előfordulhat, hogy egyes nyelvek nem jelenítik meg a különbséget alapértelmezés szerint vagy nem minden kimeneti módban).

A kellemesebb eredmény érdekében érdemes lehet karakterlánc-formázást használni korlátozott számú szignifikáns számjegy előállításához:

```
>>> format(math.pi, '.12g')  # 12 szignifiáns számjegyet ad
'3.14159265359'
>>> format(math.pi, '.2f')   # 2 számjegyet ad a tizedespont után
'3.14'
>>> repr(math.pi)
'3.141592653589793'
```

Fontos felismerni, hogy ez valós értelemben illúzió: egyszerűen csak kerekíted a valódi gépérték megjelenítését.

Egyik illúzió szülhet egy másikat. Például, mivel a 0.1 nem pontosan 1/10, előfordulhat, hogy három 0.1-es érték összegzése sem ad pontosan 0.3-at:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Továbbá, mivel a 0.1 nem kerülhet közelebb az 1/10 pontos értékéhez, a 0.3 pedig nem kerülhet közelebb a 3/10 pontos értékéhez, ezért a `round()` függvénnyel való előkerekítés sem segíthet:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Bár a számokat nem lehet közelebb vinni a tervezett pontos értékekhez, a `math.isclose()` függvény hasznos lehet a pontatlan értékek összehasonlításához:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatív megoldásként a `round()` függvény használható a durva közelítések összehasonlítására:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

A bináris lebegőpontos aritmetika sok ehhez hasonló meglepetést tartogat. A „0.1”-es problémát az alábbiakban, a „Reprezentációs hiba” részben részletesen ismertetjük. Tekintse meg a *Példák lebegőpontos problémákra* című részt a bináris lebegőpontos működés és a gyakorlatban gyakran előforduló problémák kellemes összefoglalásáért. Tekintse meg a *A lebegőpont veszélyei* című részt is, amely további gyakori meglepetések teljesebb leírását tartalmazza.

Ahogy a vége felé mondja, „nincs egyszerű válasz”. Ennek ellenére ne óvakodjon a lebegőpontostól! A Python lebegőpontos műveletek hibái a lebegőpontos hardvertől öröklődnek, és a legtöbb gépen műveletenként legfeljebb 1 rész a 2**53-ból. Ez több mint elegendő a legtöbb feladathoz, de szem előtt kell tartania, hogy ez nem decimális aritmetika, és minden float művelet új kerekítési hibát szenvedhet.

Bár léteznek kóros esetek, a lebegőpontos aritmetika legtöbb alkalmi használata esetén a várt eredményt fogja látni, ha a végeredmény megjelenítését egyszerűen a várt tizedesjegyekre kerekíti. Az `str()` általában elegendő, és a pontosabb szabályozáshoz lásd az `str.format()` metódus formátum-meghatározóit a format-stringekben.

A pontos decimális ábrázolást igénylő felhasználási esetekben próbálja meg a `decimal` modult használni, amely számviteli alkalmazásokhoz és nagy pontosságú alkalmazásokhoz alkalmas decimális aritmetikát valósít meg.

Az egzakt aritmetika egy másik formáját támogatja a `fractions` modul, amely a racionális számokon alapuló aritmetikát valósítja meg (így az 1/3-hoz hasonló számok pontosan ábrázolhatók).

Ha Ön nagy használója a lebegőpontos műveleteknek, vessen egy pillantást a NumPy csomagra és sok más, a SciPy projekt által biztosított matematikai és statisztikai műveleti csomagra. Lásd: <https://scipy.org>.

A Python olyan eszközöket biztosít, amelyek segíthetnek azokban a ritka esetekben, amikor valóban meg akarja tudni a float pontos értékét. A `float.as_integer_ratio()` metódus a float értékét törtként fejezi ki:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Mivel az arány pontos, ezzel veszteségmentesen visszaállíthatjuk az eredeti értéket:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

A `float.hex()` metódus a lebegőpontos számokat hexadecimálisan fejezi ki (alap 16), ismét megadva a számítógép által tárolt pontos értéket:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Ez a pontos hexadecimális ábrázolás használható a lebegőpontos érték pontos rekonstruálására:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Mivel az ábrázolás pontos, hasznos az értékek megbízható hordozásához a Python különböző verziói között (platformfüggetlenség), és adatcseréhez más, ugyanazt a formátumot támogató nyelvekkel (például Java és C99).

Egy másik hasznos eszköz a `sum()` függvény, amely segít csökkenteni a pontosság elvesztését az összegzés során. Kibővített pontosságot használ a közbenső kerekítési lépéseknél, mivel az értékeket hozzáadja a futó összeghez. Ez megváltoztathatja az általános pontosságot, így a hibák nem halmozódnak fel annyira, hogy befolyásolják a végső összeget:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

A `math.fsum()` továbbmegy, és nyomon követi az összes „elveszett számjegyet”, amint az értékek hozzáadódnak egy futó összeghez, így az eredménynek csak egyetlen kerekítése van. Ez lassabb, mint a `sum()`, de pontosabb olyan ritka esetekben, amikor a nagy magnitúdójú bemenetek többnyire kioltják egymást, így a végösszeg nullához közelít:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr))) # Pontos összegzés egyszerű kerekítéssel
8.042173697819788e-13
>>> math.fsum(arr)                 # Egyszerű kerekítés
8.042173697819788e-13
>>> sum(arr)              # Többszörös kerekítés kiterjesztett pontossággal
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x         # Többszörös kerekítés szabványos pontossággal
...
>>> total         # A közvetlen összeadásnak nincsenek megfelelő számjegyei!
-0.0051575902860057365
```

15.1 Ábrázolási hiba

Ez a rész részletesen elmagyarázza a „0.1” példát, és bemutatja, hogyan végezheti el saját maga az ehhez hasonló esetek pontos elemzését. Feltételezzük a bináris lebegőpontos ábrázolás alapvető ismeretét.

Az *ábrázolási hiba* arra utal, hogy néhány (a legtöbb, valójában) tizedes tört nem ábrázolható pontosan bináris (2-es számrendszerben) törtként. Ez a fő oka annak, hogy a Python (vagy Perl, C, C++, Java, Fortran és még sokan mások) gyakran nem a várt decimális számot jeleníti meg pontosan.

Miert van ez? Az 1/10 nem pontosan ábrázolható bináris törtként. Legalább 2000 óta szinte minden gép IEEE 754 bináris lebegőpontos aritmetikát használ, és szinte minden platform leképezi a Python floatokat az IEEE 754 bináris64 „kettős pontosságú” értékekre. Az IEEE 754 bináris64 értékek 53 bites pontosságot tartalmaznak, így a számítógép a 0,1-et igyekszik a *J/2**N* formájú legközelebbi törtre konvertálni, ahol *J* egy pontosan 53 bitet tartalmazó egész szám. Átírás

```
1 / 10 ~= J / (2**N)
```

mint

```
J ~= 2**N / 10
```

és emlékeztetve arra, hogy J pontosan 53 bites (>= 2**52, de < 2**53), az N legjobb értéke 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

Vagyis az 56 az egyetlen N értéke, amely J-nek pontosan 53 bitje marad. J legjobb lehetséges értéke ekkor a következő kerekített hányados:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Mivel a maradék több mint fele 10-nek, a legjobb közelítést a felfelé kerekítéssel kapjuk:

```
>>> q+1
7205759403792794
```

Ezért a lehető legjobb közelítés az 1/10-hez IEEE 754 dupla pontosságban:

```
7205759403792794 / 2 ** 56
```

Ha a számlálót és a nevezőt is elosztjuk kettővel, a tört a következőre csökken:

```
3602879701896397 / 2 ** 55
```

Vegye figyelembe, hogy mióta felfelé kerekítettük, ez valójában egy kicsit nagyobb, mint 1/10; ha nem kerekítettünk volna felfelé, a hányados kicsivel kisebb lett volna, mint 1/10. De semmi esetre sem lehet pontosan 1/10!

Tehát a számítógép soha nem „lát” 1/10-et: amit lát, az a fent megadott pontos tört, a legjobb IEEE 754-es dupla közelítés, amit kaphat:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Ha ezt a törtet megszorozzuk 10**55-tel, akkor az értéket 55 tizedesjegyig láthatjuk:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

Ez azt jelenti, hogy a számítógépben tárolt pontos szám megegyezik a 0.1000000000000000055511151231257827021181583404541015625 decimális értékkel. A teljes decimális érték megjelenítése helyett sok nyelvben (beleértve a Python régebbi verzióit is) kerekítse az eredményt 17 szignifikáns számjegyre:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

A `fractions` és `decimal` modulok megkönnyítik a számításokat:

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)
>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

16. Függelék

16.1 Interaktív mód

16.1.1 Hibakezelés

Ha hiba történik, az interpreter hibaüzenetet és veremnyomot nyomtat. Interaktív módban ezután visszatér az elsődleges prompthoz; ha a bemenet fájlból érkezett, a verem nyomkövetésének kinyomtatása után nem nulla kilépési állapottal lép ki. (A `try` utasításban szereplő `except` záradék által kezelt kivételek ebben a kontextusban nem hibák.) Egyes hibák feltétel nélkül végzetesek, és nem nulla kilépéssel kilépést okoznak; ez vonatkozik a belső inkonzisztenciákra és bizonyos esetekre, amikor kifogy a memória. Minden hibaüzenet a szabványos hibafolyamba kerül; a végrehajtott parancsok normál kimenete szabványos kimenetre íródik.

A megszakítás karakterének (általában `Control-C` vagy `Delete`) beírása az elsődleges vagy másodlagos promptba törli a bevitelt, és visszatér az elsődleges prompthoz. A parancs végrehajtása közben megszakítás beírása a `KeyboardInterrupt` kivételt idézi elő, amelyet `try` utasítással lehet kezelni.

16.1.2 Futtatható Python-szkriptek

A BSD-s Unix rendszereken a Python szkriptek közvetlenül végrehajthatóvá tehetők, mint a shell szkriptek, a következő sor elhelyezésével:

```
#!/usr/bin/env python3.5
```

(feltételezve, hogy az interpreter a felhasználó `PATH`-jában van) a szkript elején, és végrehajtható módot ad a fájlnak. A `#!` a fájl első két karakterének kell lennie. Egyes platformokon ennek az első sornak Unix-stílusú sorvégződéssel (`'\n'`) kell végződnie, nem pedig Windows (`'\r\n'`) sorvéggel. Vegye figyelembe, hogy a hash vagy font karakter, a „#” a megjegyzések indítására szolgál Pythonban.

A parancsfájl futtatható módot vagy engedélyt kaphat a **chmod** paranccsal.

```
$ chmod +x myscript.py
```

A Windows rendszereken nem létezik „végrehajtási mód”. A Python telepítője automatikusan társítja a `.py` fájlokat a `python.exe` fájlhoz, így a Python-fájlra duplán kattintva szkriptként futtatja azt. A kiterjesztés lehet `.pyw` is, ebben az esetben a normál megjelenő konzolablak le van tiltva.

16.1.3 Az interaktív indítófájl

Amikor a Pythont interaktívan használja, gyakran hasznos néhány szabványos parancs végrehajtása az értelmező minden indításakor. Ezt úgy teheti meg, hogy a `PYTHONSTARTUP` nevű környezeti változót az indítási parancsokat tartalmazó fájl nevére állítja. Ez hasonló a Unix shellek `.profile` szolgáltatásához.

Ez a fájl csak interaktív munkamenetekben olvasható, nem akkor, amikor a Python parancsokat olvas egy parancsfájlból, és nem akkor, amikor a `/dev/tty` a parancsok explicit forrása (amely egyébként interaktív munkamenetként viselkedik). Ugyanabban a névtérben kerül végrehajtásra, ahol az interaktív parancsok is végrehajtásra kerülnek, így az általa meghatározott vagy importált objektumok minősítés nélkül használhatók az interaktív munkamenetben. A `sys.ps1` és `sys.ps2` promptokat is módosíthatja ebben a fájlban.

Ha további indítási fájlt szeretne olvasni az aktuális könyvtárból, ezt a globális indítási fájlba programozhatja, például ha
`os.path.isfile('.pythonrc.py'): exec(open('.pythonrc) .py').read()).`
Ha az indítófájlt egy szkriptben szeretné használni, ezt kifejezetten meg kell tennie a szkriptben:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 A testreszabási modulok

A Python két lehetőséggel rendelkezik a testreszabáshoz: a `sitecustomize` és a `usercustomize`. Ha látni szeretné, hogyan működik, először meg kell találnia a felhasználói webhelycsomagok könyvtárának helyét. Indítsa el a Python-t, és futtassa ezt a kódot:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.12/site-packages'
```

Most létrehozhat egy `usercustomize.py` nevű fájlt ebben a könyvtárban, és bármit elhelyezhet benne. Ez hatással lesz a Python minden meghívására, kivéve, ha az automatikus importálás letiltásához a `-s` kapcsolóval indítjuk.

A `sitecustomize` ugyanúgy működik, de általában a számítógép rendszergazdája hozza létre a globális site-packages könyvtárban, és a `usercustomize` előtt importálja. További részletekért tekintse meg a `site` modul dokumentációját.