ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ II

MODERN PROGRAMOZÁSI NYELVEK (PYTHON 1 rész)

MÓDSZERTANI ÚTMUTATÓ
előadásokhoz és gyakorlati foglalkozásokhoz

Сучасні мови програмування/Modern programozási nyelvek
Перший (бакалаврський) / Alapképzés (BSc)
A Освіта/Педагогіка / A Oktatás/Pedagógia
A4.09 Середня освіта (Інформатика) / A4.09 Középiskolai oktatás (Informatika)

Берегове / Beregszász
2025 р. / 2025

Методичні вказівки до лекційних та практичних занять з дисципліни «Сучасні мови програмування (Python. 1 частина) » розроблені на основі освітньої програми підготовки бакалаврів з галузі знань «А Освіта/Педагогіка» за напрямом «А4.09 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є засвоєння студентами основ мови програмування Python, та використання їх при розробці програм різної складності. У методичці описані основні структури даних та їх використання. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу. Крім того, методичні вказівки рекомендуються студентам-бакалаврам по спеціальності «А4.09 Середня освіта (Математика)» при вивченні курсу «Алгоритми і програмування».

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 9 від «28» травня 2025 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 6 від «7» липня 2025 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від «9» липня 2025 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічних наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua)

A „Modern programozási nyelvek (Python. 1 rész)” módszertani útmutató a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola, II. éves, informatika szakos, nappali és levelezős hallgatóinak készült. A módszertani útmutató tartalmazza a Python alapvető adatszerkezeteket és azok alkalmazását, és a célja, hogy a hallgatók elsajátítsák a Python programozási nyelv alapjait, és alkalmazni tudják azokat különböző összetettségű programok készítése során. Emellett ajánlott a „A4.09 Középfokú oktatás (Matematika)” szakos alapszakos III. éves hallgatók számára is az „Algoritmusok és programozás” tantárgy tanulásához.

Az oktatási folyamatban történő felhasználását jóváhagyta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszéke
(2025. május 28, 9. számú jegyzőkönyv).

Megjelentetésre javasolta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Oktatási és Módszertani Tanácsa
(2025. július 7, 6. számú jegyzőkönyv).

Elektronikus formában (PDF fájlformátumban) történő kiadásra javasolta
a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Tudományos Tanácsa
(2025. július 9, 7. számú jegyzőkönyv).

Kiadásra előkészítette a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Matematika és Informatika Tanszéke, valamint Kiadói Részlege.

A módszertani útmutató kidolgozója:

HOLOVÁCS József – professzor, műszaki tudományok doktora, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének professzora

Szakmai lektorok:

MITSA Oleksandr – Az UNE Információs vezérlési Rendszerek és Technológiák Tanszékének vezetője, műszaki tudományok doktora, professzor

SZTOJKA Miroszláv – a fizikai és matematikai tudományok kandidátusa, docens, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének docense

A kiadásért felelnek:

KUCSINKA Katalin – PhD, a fizikai és matematikai tudományok kandidátusa, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének tanszékvezetője és docense

DOBOS Sándor – a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Kiadói Részlegének vezetője

A segédlet tartalmáért kizárólag a módszertani útmutató kidolgozója felel.

Kiadó: II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola (cím: 90 202, Beregszász, Kossuth tér 6. E-mail: foiskola@kmf.uz.ua)

TARTALOMJEGYZÉK

BEVEZETÉS.

A Python egy dinamikusan értelmezett, objektum-orientált, szkript nyelv, szigorú dinamikus típusú programozási nyelv, amelyet 1990-ben fejlesztett ki a holland programozó, Guido van Rossum.
A Python hivatalos weboldala: https://www.python.org/.

• A Python egy sokoldalú programozási nyelv, amely lehetővé teszi, hogy könnyen olvasható kódot írjunk. A Python nyelv tömörsége lehetővé teszi, hogy a program sokkal rövidebb legyen, mint egy másik nyelven írt analógja.

• A Python egy többplatformos programozási nyelv. Ez azt jelenti, hogy a Python alkalmazásokat különböző operációs rendszereken futtathatjuk változtatások nélkül.

• A Python előnye a standard könyvtára, amely a Python telepítésével együtt érkezik, és tartalmazza a kész eszközöket az operációs rendszerrel, weboldalakkal, adatbázisokkal, különböző adatformátumokkal való munkához, grafikus felhasználói felület készítéséhez stb.

• A Python kód végrehajtásának sebessége. Az egyik lehetséges hátrány a Python sebessége. A Python nem egy fordított nyelv. A Python kód először belső byte-kóddá fordítódik, amelyet azután a Python értelmező hajt végre. A legtöbb esetben a Python alkalmazások lassabbak, mint a C nyelven írt programok. De a legtöbb alkalmazás esetében a fejlesztési sebesség fontosabb, mint a végrehajtás sebessége, és a Python programok jellemzően sokkal gyorsabban készülnek el.

• Ezen kívül a Python könnyen bővíthető C vagy C++ nyelven írt modulokkal. Az ilyen modulok a program CPU-terhelést okozó részeinek végrehajtására használhatók.

• A Python különböző célokra használható: játékok és webalkalmazások készítésére, belső eszközök fejlesztésére különböző projektekhez. A nyelvet széles körben alkalmazzák tudományos kutatásokhoz és gyakorlati problémák megoldásához.

Milyen sorrendben tanuljunk a Pythont

A Python elsajátítása akkor lehet hatékony, ha a témakörök logikus sorrendjét követi.
Egy lehetséges sorrend.

1. Python szintaxis alapjai:
változók, adattípusok, operátorok.
2. Vezérlési parancsok
• Feltételes kifejezések (if, else, elif)
• Ciklusok (for, while)
3. Adatszerkezetek
• Listák (lists)
• Tuples (kortezs)
• Szótárak (dictionaries)
• Halmazok (sets)
4. Függvények és modulok
• Függvények meghatározása és hívása
• Modulokkal és könyvtárakkal való munka
5. Hibakezelés
Kivételek használata.
6. Fájlokkal való műveletek
Fájlok olvasása és írása
7. OOP (objektumorientált programozás)
• Osztályok meghatározása és objektumok létrehozása
• Öröklődés, polimorfizmus
8.Adatbázisok
A MySQL, SQLite vagy más adatbázisokkal való munka alapjai
9. Webfejlesztés
• Flask, Django, alkalmazása
• HTTP parancsok kezelése.
10. Tudományos számítástechnika és adatelemzés
• Tudományos számítástechnikai könyvtárak, például NumPy, SciPy.
11. Adatok kezelése a pandas könyvtár használatával

Fontos: a Python (mint más programozási nyelv) tanulásának kulcsa a gyakorlat, rendszeres feladatok, problémák megoldása.

1. A gépi nyelvtől a magas szintű nyelvig

A processzor a legfontosabb komponens a számítógépben. A processzor egyik alapvető feladata az adatok feldolgozása a számítógépes program alapján, amely egy utasítások listája, amelyek aritmetikai és logikai műveletek végrehajtásával dolgoznak az adatokkal.

Minden program utasítása egy parancs, amely „megmondja” a processzornak, hogy milyen műveletet hajtson végre. Például egy ilyen utasítás lehet:

10111010

Számunkra ez csak egy 0-k és 1-ek sorozata, de a processzor számára ez egy utasítás, amely végrehajt egy műveletet. A számítógép processzora csak a gépi nyelven írt utasításokat érti.

A gépi nyelv egy mesterséges nyelv, amelyet a számítógép parancsainak közvetítésére hoztak létre. A gépi nyelv segítségével hatékony programok készíthetők, mivel a fejlesztő hozzáférhet a processzor összes lehetőségéhez. A gépi nyelv egy alacsony szintű nyelv. Minden egyes művelethez, amelyet a processzor végre tud hajtani, létezik egy gépi nyelvi utasítás – van egy utasítás a számok összeadására, van egy utasítás a kivonásra stb. Az összes olyan utasítást, amelyet a központi processzor képes végrehajtani, processzor utasításkészletnek nevezzük.

Például van egy program, amely a számítógép lemezén tárolódik. A program végrehajtásához duplán kattintunk a program ikonjára. Ez arra kényszeríti a programot, hogy átmásolja magát a lemezről a RAM-ba, és onnan a számítógép processzora végrehajtja a RAM-ban lévő másolatot. Amikor a processzor végrehajtja a program utasításait, részt vesz egy olyan folyamatban, amelyet „fetch - decode - execute” (lekérni - dekódolni - végrehajtani) ciklusnak nevezünk. Ez a ciklus minden egyes program utasításra vonatkozik, és három lépésből áll:

Lekérni

A program egy gépi nyelvű utasítások sorozata. Az első lépés a ciklusban a következő utasítás betöltése (lekérése) a memóriából a processzorba.

Dekódolni

A gépi nyelvi utasítás egy bináris szám, amely a processzornak szóló parancsot képvisel, hogy végrehajtson egy műveletet. Ezen a lépésen a processzor dekódolja az utasítást, amelyet a memóriából „kivontak”, és meghatározza, hogy milyen műveletet kell végrehajtani.

Végrehajtani

A ciklus utolsó lépése a művelet végrehajtása.

1.1. Assembler nyelv

Bár a számítógép processzora csak a gépi nyelvet érti, az emberek számára nem praktikus programokat írni gépi nyelven. Az ilyen programoknak ezer vagy akár millió bináris utasítást kellene tartalmazniuk, és az ilyen programok írása rendkívül fáradságos lenne.

Ezért létrehozták az assembler nyelvet, mint alternatívát a gépi nyelvhez. Az assembler nyelv a bináris számok helyett rövid szavakat, ún. mnemokódokat használ.

Például az assembler nyelvben a mnemokód **add** jellemzően azt jelenti, hogy össze kell adni a számokat, a **mul** pedig azt, hogy szorzásra van szükség, a **mov** pedig az érték áthelyezését jelenti a memória egy adott helyére.
Mivel a processzor csak a gépi nyelvet érti, egy speciális assembler programot használnak a program gépi nyelvű programra történő fordítására.

Bár az assembler nyelv nem igényel bináris utasításokat, mint a gépi nyelv, mégis magas szintű ismereteket követel a processzorról. Az assembler nyelv használatával még a legegyszerűbb programhoz is sok utasítást kell írni. Mivel az assembler nyelv természeténél fogva közel áll a gépi nyelvhez, *alacsony szintű nyelvnek* számít.

A *magas szintű programozási nyelvek* lehetővé teszik, hogy bonyolult programokat készítsünk anélkül, hogy tudnánk, hogyan működik a processzor, és anélkül, hogy nagy mennyiségű alacsony szintű utasítást kellene írni. Ezenkívül a legtöbb magas szintű programozási nyelv könnyen érthető szavakat használ. A Python egyike a népszerű modern magas szintű programozási nyelveknek.

Például Pythonban a "Hello, World!" üzenet megjelenítéséhez az alábbi utasítást kell írni:

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

Ahhoz, hogy ugyanezt végre hajtsuk assembler nyelven, számos utasításra és mély ismeretekre van szükség a processzor és a számítógép eszközeinek interakciójáról.

```
section  .text
    global _start
_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    int 0x80

section  .data
msg db  'Hello, World!',0xa
```

```
len equ $ - msg
```

Ahogy az ebből a példából látható, a magas szintű nyelvek lehetővé teszik, hogy a feladatokra összpontosítsunk, amelyeket végre kell hajtani, nem pedig arra, hogy hogyan fogja a processzor végrehajtani a számítógépes programokat.

1.2.Python – értelmezett programozási nyelv

A magas szintű nyelveken írt programokat **forráskódnak** nevezzük. Ahhoz, hogy a számítógép végrehajthassa a programot, a forráskódot le kell fordítani gépi nyelvre. A magas szintű nyelvek gépi nyelvre való lefordításához két típusú program áll rendelkezésre: a **fordító és az értelmező.**

A **fordító** az egész forráskódot egyszerre fordítja le gépi nyelvre, majd a gépi kódot hajtja végre. Az **értelmező** a magas szintű nyelven írt programot soronként fordítja le gépi nyelvre, és minden egyes sort végrehajt.

Python letöltése

A Python értelmező különböző operációs rendszerekhez elérhető ingyenesen letölthető a következő linken: https://www.python.org/downloads.

Fejlesztői környezetek Pythonhoz

Programok írásához szövegszerkesztőket vagy **integrált fejlesztői környezeteket (IDE)** használnak, amelyek különféle eszközöket tartalmaznak a kóddal való munkához*: kódszerkesztőt (szövegszerkesztőt), interaktív értelmezőt, hibakeresőt* stb.

Python szövegszerkesztők és integrált fejlesztői környezetek:

- **IDLE** – a Python hivatalos szerkesztője. Windows felhasználóknak a Python telepítésével együtt érkezik, míg Linux felhasználóknak külön csomagban.

- **Notepad++** – ingyenes szövegszerkesztő, amely több programozási nyelvet, így a Pythont is támogatja. Csak Windows felhasználóknak.
- **Visual Studio Code** – könnyű, de erőteljes kódszerkesztő, amely ingyenesen elérhető és támogatja a Linux, Windows és macOS platformokat.
- **PyScripter** – integrált fejlesztői környezet Pythonhoz. Windows felhasználóknak. Ingyenesen elérhető.
- **Wing IDE 101** – ingyenes fejlesztői környezet Pythonhoz, kezdő programozók számára. Linux, Windows és macOS felhasználóknak. Ingyenesen elérhető.
- **Geany** – ingyenes szövegszerkesztő alapvető fejlesztői környezeti elemekkel, amely elérhető Linux, Windows és macOS operációs rendszerekhez.
- **PyCharm** – integrált fejlesztői környezet Pythonhoz. A PyCharm tulajdonosi szoftver, de van egy ingyenes Community verziója korlátozott funkciókkal. Linux, Windows és macOS felhasználóknak.
- **Thonny** – IDE a Python programozás tanulásához. Linux, Windows és macOS felhasználóknak.
- **Mu** – Python kódszerkesztő kezdő programozók számára. Linux, Windows és macOS felhasználóknak.

A Python programok írásához online környezetet is használhatunk, például a **repl.it**-et.

1.3. Python futtatása

Interaktív értelmező

Az interaktív értelmező módban a parancsokat terminál ablakban (parancssor, parancssori ablak) soronként adjuk meg, és az Enter billentyű megnyomásával azonnal végrehajtódnak, és az eredmény pedig megjelenik.

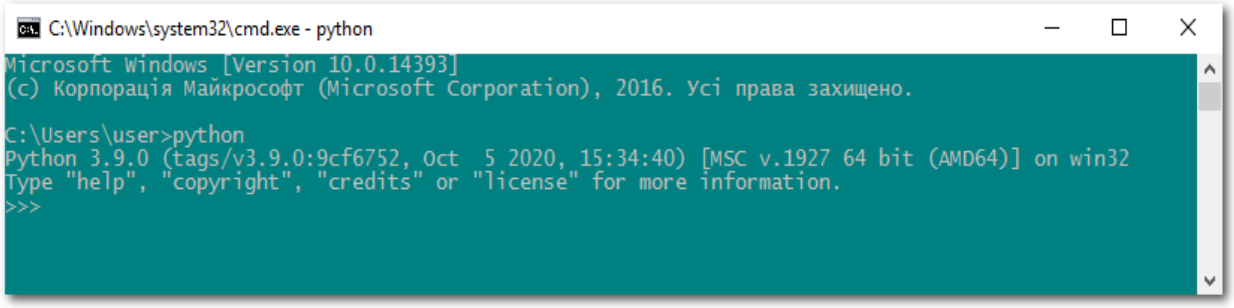
A terminál ablak megnyitásához:

- Nyomja meg a **Win+R** billentyűkombinációt a billentyűzeten, írja be a „cmd” parancsot, majd kattintson az OK-ra;

- A megjelenő terminál ablakba írja be a következő parancsot:

python

Ha a képernyőn megjelenik a „>>>” prompt, az azt jelenti, hogy a rendszer felismerte a telepített Python verziót.



Írja be az interaktív értelmező módjában a következő sort:

print('Hello, Python!')

Nyomja meg az Enter billentyűt, és győződjön meg arról, hogy a képernyőn megjelent a "Hello, Python!" üzenet.

A *print()* függvény a Python standard könyvtárának része. Ez a függvény a zárójelekben megadott információt a képernyőre írja ki, vagy fájlba menti. Ahhoz, hogy kilépjen az interaktív Python értelmező módból, hajtsa végre az *exit()* parancsot.

A Python programok tárolása

A Python nyelven írt programok **.py kiterjesztésű** szöveges fájlokban kerülnek mentésre. A szövegszerkesztőben hozzon létre egy új fájlt, és írja bele a következő sort:

print("Hello, Python!")

Mentse el a fájlt *hello.py* néven egy mappába (pl. python_work). Az ilyen kódfájlokat terminál ablakban is futtathatja.

Programok futtatása Windows terminálban

Nyissa meg a terminál ablakot (parancssori ablak): nyomja meg a **Win+R** billentyűkombinációt a billentyűzeten, írja be a cmd parancsot, majd kattintson az OK gombra, és hajtsa végre a következő parancsokat:

Például:

```
C:\Users>User> cd Desktop
C:\Users>User\Desktop> cd python_work
C:\Users>User\Desktop\python_work> dir
hello.py
C:\Users>User\Desktop\python_work> python hello.py
Hello, Python!
```

Hibaüzenetek

A programok írása és futtatása során különféle hibák léphetnek fel. Ilyen esetekben a Python értelmező automatikusan jelzi a hibát. Például, ha az interaktív értelmező módjában a következő parancsot adjuk meg: '19' + 81, akkor a következő hibaüzenet jelenik meg:

```
>>> '19' + 81
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

A bevitt parancs nem helyes Python számára, ezért a rendszer megadja a hiba típusát és a sor számát, ahol a hiba történt, majd leállítja a program futtatását.
A Python-ban hiba esetén egy kivétel keletkezik, amely tájékoztat a hiba lényegéről. Ebben az esetben a TypeError kivétel jelzi, hogy számot és sztringet nem lehet összeadni, vagyis sztringet csak sztringgel lehet összefűzni.

Ha a hiba világos számunkra, akkor kijavíthatjuk. Ha nem értjük, hogy mit jelent a hibaüzenet, akkor kereshetünk rá a hiba típusára az interneten.

Kommentárok

A kommentárok rendkívül hasznosak bármely programozási nyelvben. A kód terjedelme és bonyolultsága növekedésével célszerű kommentárokat hozzáadni, amelyek leírják a megoldandó probléma általános megközelítését. A kommentárok egyfajta jegyzetek, amelyeket érthető nyelven írunk.

A Python-ban a kommentár jelzése a # karakter. Az értelmező figyelmen kívül hagyja a # utáni összes karaktert a sor végéig. Például:

```
>>> # Ez megjegyzés!
... print("Hello, world!")
Hello, world!
```

Sorok folytatása

Bármely program érthetőbbé válik, ha a sorai rövidek. Az ajánlott (de nem kötelező) maximális sorhossz nem haladhatja meg a 80 karaktert. Ha nem tudja kifejezni a gondolatát 80 karakterben, használja a sorfolytatás karaktert (\).
Egyszerűen helyezzen egy \ jelet a sor végére, és a Python úgy kezeli, mintha ugyanaz a sor folytatódna.

Például:

```
>>> alphabet = 'abcdefg' + \
... 'hijklmnop' + \
... 'qrstuv' + \
... 'wxyz'
>>> alphabet
'abcdefghijklmnopqrstuvwxyz'
```

```
>>> 1 + 2 + \
... 3
6
```

A legtöbb programozási nyelv *speciális karaktereket* (például kapcsos zárójeleket {}) vagy kulcsszavakat (például begin és end) használ a kód részekre bontására. Ezekben a nyelvekben jó szokás a **behúzások használata a kód írásakor**, hogy a program könnyen olvasható legyen, akár a programozó, akár mások számára.

A Python-ban ellentétben sok más nyelvvel, ***kötelező a kódblokkok helyes behúzása***. A Python-ban a program struktúráját balra történő behúzásokkal építjük fel, amelyeket szóközökkel hozunk létre.
A Python program tipikus struktúrája a következőképpen nézhet ki (pontok jelzik a szóközöket):

```
n = int(input())
out = []
for i in range(n):
....k = 0
....while k < i + 1:
........out.append(i+1)
........k += 1
....if len(out) > n:
........break
out = out[0:n]
for i in out:
....print(i, end = " ")
```

Mielőtt kezdenénk programozni

Helyes tanulási stilus

- **Rendszeres ismétlés:** Az alapfogalmak megértése után fontos, hogy az új témák rendszeresen épüljenek a már tanultakra.
- **Fokozatosan bonyolódó feladatok:** Kezdetben egyszerű feladatokat adjunk, majd fokozatosan növeljük azok nehézségét.
- **Projekt alapú tanulás:** A valós problémákra alkalmazott projektek segítenek az elméleti tudás gyakorlatba ültetésében.

2. ADATOK A PYTHONBAN

2.1.Egyszerű adattípusok

A program futtatásakor az adatok különböző típusúak (számok, szövegek stb.), és a program ezeket kezeli. Az adattípus meghatározza az engedélyezett értékek halmazát, azok tárolásának formátumát, az allokált memória méretét, valamint azokat az operációkat, amelyeket ezekkel végezhetünk. Minden adatot jellemez a tárolásra szolgáló

- *memória mérete,*
- *neve (azonosítója),*
- *típusa,*
- *értéke.*

A nevek az adatok azonosítására szolgálnak, tehát megkülönböztethetjük őket egymástól. Minden értékhez típust rendelünk, amely a valós objektumok ábrázolására szolgál.

A Python beépített egyszerű adattípusai a következők:

Boole-értékek (True vagy False értékek).

Egész számok (például 81, 1000).

Lebegőpontos számok (például 3.14159, 2.5e8, 4000.0).

Szövegek (karakterláncok, például "Hello, Python!").

Minden típusnak megvannak a saját használati szabályai, és ezek különbözően kerülnek feldolgozásra a számítógép által.

Objektumok és változók

A Python-ban minden (boole-értékek, egész számok, lebegőpontos számok, szövegek, összetett adatstruktúrák, függvények) objektumként van megvalósítva.
Az objektum egy különálló adattároló egység a programok futása során, amely alapvető építőelem a programok felépítésében.

Az objektumot úgy képzelhetjük el, mint egy dobozt, amely adatokat tartalmaz. Az objektumnak típusa van (például boole-érték típus vagy egész szám típus), amely meghatározza, hogy mit lehet csinálni ezekkel az adatokkal.
A valós világban egy „Könyvek” feliratú doboz azt jelzi számunkra, hogy könyveket (adatfájlokat) tartalmaz, amiket elvehetünk vagy újakat helyezhetünk el benne, de csak könyveket.

A Python-ban a változónak adott érték hozzárendeléséhez az "=" szimbólumot használjuk.

Az alábbi programrészletben a 12-es egész szám értékét hozzárendeljük az „a” nevű változóhoz, majd kiíratjuk az értéket, amely jelenleg ehhez a változóhoz tartozik:

```
a = 12
a
12
```

Elterjedt az az elépzelés, hogy a változó egy tároló az értékek számára. Ez az analógia érvényes sok programozási nyelv esetében (például C).

A Python-ban a változók nem tárolók, hanem címkék, amelyek a Python értelmező névteréből származó objektumokhoz vannak rendelve.

Egy objektumhoz bármennyi változó rendelhető, és ha az objektum változik, akkor minden hozzá tartozó változó értéke is változik.

Nézzünk egy egyszerű Python programot, hogy megértsük, mit jelent ez:

```
a = [5, 6, 7]
b = a
c = b
```

Az „a” változó egy három számjegyű listát tartalmaz.
Az „a” értékét hozzárendeljük a „b” változóhoz.
A „b” értékét hozzárendeljük a „c” változóhoz.
Most változtassuk meg az első értéket (5-ről 100-ra) a „b” változóban tárolt listában:

```
a = [5, 6, 7]
b = a
c = b
b[0] = 100
```

Most írjuk ki mindhárom változó értékeit:

```
a = [5, 6, 7]
b = a
c = b
b[0] = 100
print(a, b, c)
[100, 6, 7] [100, 6, 7] [100, 6, 7]
```

A Python-ban a változók egyszerűen csak nevek. A hozzárendelés nem másolja az értékeket, hanem egy nevet rendel az objektumhoz, amely az adatokat tartalmazza.

A név egy hivatkozás az objektumra, nem maga az objektum.

A következő lépéseket hajtsa végre az interaktív értelmező használatával:

- Mint korábban, adja meg a 12-es értéket az „a” változónak. Ez létrehoz egy objektumot (doboz), amely 12-t tartalmaz.
- Írja ki az „a” értékét a képernyőre.
- Rendelje hozzá az „a” változó értékét a „b” változóhoz, így a „b” hozzá lesz rendelve ahhoz az objektumhoz, amely a 12-es értéket tartalmazza.
- Írja ki a „b” értékét.

```
>>> a = 12
>>> a
12
>>> b = a
>>> b
12
```

Az **id() függvény** visszaadja az objektum azonosítóját (az objektum memória címét). Nézzük meg, hogy a változók `a` és `b` ugyanarra a 12-es értékű egész szám objektumra mutatnak-e az `id()` függvény segítségével.

```
>>> id(a)
1367960944
>>> id(b)
1367960944
```

A kimenetből látható, hogy amikor a változót egy meglévő változóhoz rendelünk, mindkét változó ugyanarra az objektumra mutat. A változó nem kerül másolásra!

Mi lesz tárolva a `b` változóban, ha a következő két utasítást végrehajtjuk?

```
>>> b = 'Goodbye!'
>>> b = 'Hello!'
>>> b
'Hello!'
```

A Python-ban a változók bármilyen objektumot tárolhatnak, míg C-ben és sok más nyelvben a változó csak annak a típusnak az értékeit tárolhatja, amellyel deklarálták. Python-ban a változónak nincs típusa, az objektumnak van típusa, nem a változónak.

```
>>> b = 'Goodbye!'
>>> id(b)
```

```
2371957227888
>>> b = 'Hello!'
>>> id(b)
2371957221112
>>> b
'Hello!
```

Az azonosítók használhatók az **is operátorral** történő ellenőrzésre, tehát az is azt vizsgálja, hogy két változó ugyanarra az objektumra mutat-e:

```
>>> c = 10
>>> d = c
>>> c is d
True
>>> id(c)
1367960880
>>> id(d)
1367960880
>>> d = 15
>>> c is d
False
>>> id(d)
1367961040
```

Egy hasznos példa:

```
>>> s = 'Python'
>>> s
'Python'
>>> s = 3
>>> s
3
```

Az **s** változó először egy "Python" szöveges objektumra mutat, majd egy egész szám 3 objektumra.

Ha meg szeretnénk tudni egy objektum (változó vagy érték) típusát, a beépített **type() függvényt** kell használni.

Próbáljuk ki ezt különböző értékekre (81, 99.99, 'ruby') és változóra (a):

```
>>> type(a)
<Class 'int'>
>>> type(81)
<Class 'int'>
>>> type(99.99)
<Class 'float'>
>>> type('ruby')
<Class 'str'>
```

Amikor egy változó új értéket kap, az előző érték elvész, és egy új objektum jön létre: most a **b** változó értéke 'Hello!' és az egy új objektumra mutat, mivel a változót áthelyeztük egyik objektumról a másikra (az azonosítók megváltoztak).

Az **osztály** az objektum definíciója. Python-ban a „osztály” és a „típus” kifejezések körülbelül ugyanazt jelentik.

Be kell tartani bizonyos szabályokat a változók nevének használatakor. *A változónevek csak az alábbi karaktereket tartalmazhatják:*

- kisbetűk (a-z)
- nagybetűk (A-Z)
- számjegyek (0-9)
- aláhúzás (_)

A nevek nem kezdődhetnek számjegyjel.

Python különleges módon kezeli azokat a neveket, amelyek aláhúzássa

l kezdődnek.

Nem ajánlott foglalt kulcsszavakat használni változónevekként.

Python kulcsszavak

False	**Class**	**finally**	**is**	**return**	**None**
Continue	**For**	**lambda**	**try**	**True**	**Def**
from	**nonlocal**		**While**	**And**	**Del**
global	**not**		**with**	**As**	**Elif**
If	**or**	**yield**	**Assert**		**Else**
import	**pass**	**Break**	**Except**	**In**	**raise**

2.2.Számok

A számokat gyakran alkalmazzák a programozásban. A számokat kifejezésekben használják (például: 2 + 3), amelyekben matematikai operátorokkal (például: +) végeznek számításokat az operátorokon (például: 2, 3).

Matematikai operátorok és azok használata

Operátor	Leírás	Példa	Eredmény
+	**Összeadás**	5 + 7	12
-	**Kivonás**	55 - 10	45
*	**Szorzás**	4 * 6	24
/	**Osztás**	7 / 2	3.5
//	**Egész osztás**	7 // 2	3
%	**Maradékos osztás**	7%3	1
**	**Hatványozás**	3 ** 4	81

A Pythonban a numerikus adatokat több kategóriába sorolják a felhasználásuk módja szerint.

Egész számok

Bármilyen számjegysorozat a Pythonban egész számnak számít: Nem lehet 0-t tenni a szám elejére, mert ez hibát okoz (érvénytelen karakter).

```
>>> 10

10

>>> 05

  File "<stdin>", line 1

    05
```

```
    ^
SyntaxError: invalid token
```

A számjegyek sorozata egy egész számot jelöl. Ha + jelet teszel a számok elé, az érték nem változik. A negatív számot úgy jelölhetjük, hogy - jelet teszünk a szám elé.

A Python segítségével matematikai műveleteket végezhetünk, mint egy hagyományos kalkulátorral:

```
>>> 25 + 9
34
>>> 145 - 37
108
>>> 8 + 3 - 2 + 1 - 106
-96
>>> 6 * 7
42
```

Osztás műveletek – két típus létezik.

A **/ operátor** használatával lebegőpontos osztás történik (*tizedes osztás*). Még akkor is, ha két egész számot osztunk el, a / operátor lebegőpontos eredményt ad:

```
>>> 9 / 5
1.8
```

Az egész osztás a **// operátor** használatával történik, amely egész számú választ ad, elvetve a maradékot:

```
>>> 9 // 5
1
```

A nullával való osztás bármely operátor használatával hibát okoz:

```
>>> 6 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 8 // 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Használjunk változókat a számításokban, és változtassuk meg az értékeiket:

```
>>> a = 42
>>> a = a - 2
>>> a
40
```

A Pythonban a = jel jobb oldalán lévő kifejezés először kiszámításra kerül, az eredményt elmentik, majd csak ekkor rendelik hozzá a baloldali változóhoz.

A matematikai operátorok együtt is használhatók a hozzárendelési operátorral, ha azokat a hozzárendelési szimbólum elé helyezzük:

```
>>> a = 95
>>> a -= 3
>>> a
92
>>> a += 8
>>> a
100
>>> a *= 2
>>> a
200
>>> a /= 3
>>> a
66.66666666666667
>>> 23 % 6
5
```

Megjegyzés: A Pythonban korlátozhatjuk a tizedesjegyek számát.

- **Szövegek formázása f-string segítségével:**

```
result = 66.66666666666667

print(f"{result:.2f}")  # 66.67.
```

- **round()függvény által:**

```
result = 66.66666666666667
rounded_result = round(result, 2)
print(rounded_result)
```

Tipus konverziók: az int() függvény

Ha más típusú adatokat szeretnénk egész szám típusra alakítani, akkor az **int() függvényt** kell használni. Az int() függvény az egész szám részét megtartja, és eldobja a maradékot. A Pythonban az egyik legegyszerűbb adattípus a logikai változók, amelyek értékei csak *True vagy False* lehetnek. Amikor egész számokra konvertáljuk őket, 1 és 0 értéket kapnak:

```
>>> int(True)

1

>>> int(False)

0
```

A lebegőpontos szám egész számra konvertálása egyszerűen levágja a tizedesvessző utáni részt:

```
>>> int(98.6)

98

>>> int(1.5e4)

15000
```

Nézzük meg egy szöveges karakterlánc átalakításának példáját, amely csak számokat vagy számokat és + és - jeleket tartalmaz:

```
>>> int('99')

99

>>> int('-23')

-23

>>> int('+12')

12
```

Ha valami olyan dolgot próbálunk konvertálni, ami nem szám, hiba történik:

```
>>> int('22 abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '22 abc'
```

Operátorok prioritása. A prioritásokat a zárójelek használatával is beállíthatjuk:

```
>>> 7 * (3 + 4)
49
```

Lebegőpontos számok

A Pythonban azokat a számokat, amelyek törtrészt tartalmaznak, valós számoknak (vagy "lebegőpontos számoknak") nevezik. A lebegőpontos számokhoz a következő operátorokat használjuk: +, -, *, /, //, **, %.

Típuskonverziók a float() függvény

Ha más típusokat szeretnénk átalakítani float típusra (így nevezik a valós számokat a Pythonban), akkor a float() függvényt kell használni. A float() függvény más típusok értékeit konvertálja lebegőpontos értékekre. A logikai értékek számokként kerülnek kezelésre:

```
>>> float(True)
1.0
>>> float(False)
0.0
```

Az int típus float típusra történő átalakítása:

```
>>> float(98)
98.0
```

Szöveges karakterláncok is konvertálhatók, amelyek olyan karaktereket tartalmaznak, amelyek érvényes számok (egész számok vagy lebegőpontos számok):

```
>>> float('99')
99.0
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

Matematikai függvények

A Python standard függvényeket biztosít a numerikus adatok kezeléséhez: **abs(), pow(), round().** Nézzük meg, hogyan működnek ezek a függvények:

```
>>> abs(-34)
34
>>> abs(-34.56)
34.56
>>> pow(2, 3)
8
>>> pow(-4.5, 2)
20.25
>>> round(10.6)
11
>>> round(3.5)
4
>>> round(8.5)
8
>>> round(2.665, 2)
2.67
>>> round(2.675, 2)
2.67
```

Ha nincs második argumentum a round() függvényben, a számot a legközelebbi egész számra kerekíti (a 11-es szám közelebb

van). Visszaadja a lebegőpontos számot, amely a megadott számú tizedesjegyig van kerekítve.

Lebegőpontos aritmetika: problémák és korlátozások

Nézzük meg, hogyan ábrázolja a Python a 2.665 és 2.675 számokat 30 tizedesjeggyel.

Használjuk:

```
>>> '{:.30}'.format(2.665)
'2.665000000000000035527136788 01'
>>> '{:.30}'.format(2.675)
'2.674999999999999822364316059 97'
```

Tehát ilyen hosszú számjegyek esetén a két tizedesjegyre való kerekítés valójában ugyanazt az eredményt adja. Ebben az esetben a helytelen kerekítés oka az, hogy a **legtöbb tizedes tört nem ábrázolható pontosan lebegőpontos típusokkal**. A beépített lebegőpontos típus alapja a dupla pontosságú (double) számábrázolás. A legtöbb program számára ez a pontosság elegendő. De olyan esetekben, amikor a tört értékek pontosságára nagyobb igények vannak, mint látjuk, még a legegyszerűbb matematikai műveletek is váratlan eredményeket adhatnak. Gondolkodjunk el, milyen értéket kapunk, ha a Pythonban kiszámoljuk az 0.1 + 0.2 kifejezést? Azt gondolhatnánk, hogy az eredmény 0.3 lesz, de ha ellenőrizzük a tényleges eredményt a Python interaktív módjában, más értéket kapunk:

```
>>> 0.1 + 0.2
0.30000000000000004
```

A lebegőpontos számok számítógépes ábrázolása tört formájában történik bináris számrendszerben, tehát az alap 2-ben (bináris tört). Miért történik ez így? Próbáljuk meg megérteni néhány példán keresztül. Például a 0.125 tört 10-es alapú ábrázolásában 1/10 + 2/100 + 5/1000 értékkel rendelkezik. Hasonlóan, a 0.001 tört bináris ábrázolása 0/2 + 0/4 + 1/8 értékkel rendelkezik. E két tört értéke ugyanaz, és csak abban különböznek, hogy az egyik 10-es alapú, a másik pedig 2-es alapú tört. Sajnos **a legtöbb tizedes tört nem ábrázolható pontosan bináris formában**. Ennek következményeként a legtöbb tizedes lebegőpontos szám, amit beírunk, csak közelítő érték lesz a számítógépben tárolt bináris lebegőpontos számokhoz. Próbáljuk meg ábrázolni a 1/3 törtszámot tizedes formában, és észrevesszük, hogy a szám valójában végtelen lesz 10-es alapú ábrázolásban. Függetlenül attól, hogy hány tizedesjegyet írunk (0.3, 0.33, 0.333, …), az eredmény soha nem lesz 1/3, hanem

egyre jobb közelítés lesz 1/3-hoz. Hasonlóan, egyes számok, mint például 0.1 vagy 1/10, nem rendelkeznek véges bináris ábrázolással. Függetlenül attól, hogy hány számjegyet írunk a bináris ábrázolásban, a tizedes 0.1 szám nem ábrázolható pontosan 2-es alapú törtként. Bináris ábrázolásban a 1/10 egy végtelenül ismétlődő tört lesz:
0.0001100110011001100110011001100110011001100110011...

Ha bármilyen végső számú bitnél megállunk, akkor egy közelítő értéket kapunk. A mai számítógépek többségénél a közelítést bináris tört formájában számítják ki, amelynek számlálója az első 53 bitet használja, a nevező pedig 2 hatványaként van ábrázolva. Sokan nem tudnak a közelítésekről, mivel az értékek megjelenítése eltakarja ezt. A Python csak a számítógépben tárolt bináris közelítés tizedes közelítését jeleníti meg. A legtöbb számítógépen, ha a Python ki akarja nyomtatni a 0.1 valódi tizedes értékét, akkor így kellene kinéznie:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Ezt az értéket a formázott megjelenítéssel is elérhetjük: 0.1 Itt több számjegy található, mint amit a legtöbb ember hasznosnak találna, így a Python lehetővé teszi a számjegyek számának kezelését, a kerekített értékek megjelenítésével:

```
>>> 1 / 10
0.1
```

Ne felejtsük el, hogy még ha az eredmény pontosan 1/10-nek tűnik, valójában a tárolt érték a bináris tört legközelebbi reprezentációja. Tehát a 0.30000000000000004 valójában nagyon közel van a 0.3-hoz. Egy másik probléma ezekkel a közelítésekkel az, hogy a hibák továbbra is felhalmozódnak.

1.3. Stringek (szövegek)

A Python 3 a *Unicode szabvány* támogatásának köszönhetően képes bármely világnyelv karaktereit, valamint sok más karaktert kezelni. A Unicode szabvány kezelésének szükségessége volt az egyik oka annak, hogy a Python 2 változott. A Python 3 szöveges karakterláncai Unicode formátumú karakterláncok.

Szövegek létrehozása és a print() függvény

Bármely karakterek sorozata, amelyet idézőjelekbe helyezünk, Pythonban szövegnek számít, és a szövegek egyaránt lehetnek egyes és dupla idézőjelek között is:

```
>>> 'This is a string.'
'This is a string.'
>>> "This is also a string."
'This is also a string.'
```

Miért kell használni kétféle idéző jelet? Az alapgondolat az, hogy létrehozhatsz olyan szövegeket, amelyek idézőjeleket vagy aposztrófokat tartalmaznak. Ezért az egyes idézőjelek közé helyezhetsz dupla idézőjeleket és fordítva:

```
>>> 'I told my friend, "Python is my favorite language!"'
'I told my friend, "Python is my favorite language!"'
>>> "The language 'Python' is named after Monty Python, not the snake."
"The language 'Python' is named after Monty Python, not the snake."
>>> "One of Python's strengths is its diverse and supportive community."
"One of Python's strengths is its diverse and supportive community."
```

Három egyszeres (''') vagy három dupla idézőjel (""") is használható: '''Igen'''

```
>>> '''Yes'''
'Yes'
>>> """No"""
'No'
```

A három idézőjel nem igazán hasznos rövid szövegekhez. Általában többsoros szövegekhez használják őket:

```
>>> """In the age of high technology
... Without programs you can not do,
... Programmers daily
... It makes life easier for us!"""
```

Minden egyes sort interaktív módon írtunk be az interpreterbe, amíg be nem írtuk az utolsó három idézőjelet, és át nem léptünk a következő sorra. A három idézőjel belsejében az új sorok (\n) és a szóközök is megmaradnak. Ha megpróbálnál hosszú szöveget létrehozni egyszeres idézőjelek segítségével, Python hibát generálna, amikor a következő sorra lépnél:

```
>>> 'In the age of high technology
  File "<stdin>", line 1
    'In the age of high technology
                                  ^
SyntaxError: EOL while scanning string literal
```

Van különbség az interaktív interpreterrel történő kimeneti megjelenítés:

```
>>> """In the age of high technology
... Without programs you can not do,
... Programmers daily
... It makes life easier for us!"""
'In the age of high technology\nWithout programs you can not do,\nProgrammers daily\nIt makes life easier for us!'
```

és a print() között:

```
>>> print('Python', 'is', 'my', 'favorite', 'language!')
Python is my favorite language!
```

A fenti példából látható, hogy a print() függvény a szöveg tartalmát jeleníti meg az idézőjelek nélkül, és szóközt ad minden kimeneti elem közé, valamint új sor karaktert (\n) a végére.

A print() függvény end és sep opcionális paraméterekkel rendelkezik:

amelyek segítségével meghatározhatjuk, hogy mi legyen a kimenet végén, és milyen karaktert használjon a függvény a tartalom belső elválasztására.

Ha meg szeretnénk változtatni az alapértelmezett elválasztót (ami egy szóköz), akkor használhatjuk a *sep paramétert* és megadhatjuk az elválasztót:

```
>>> print('Python', 'is', 'my', 'favorite', 'language!',
sep=',')

Python,is,my,favorite,language!
```

Nézzük meg, hogyan működik az *end* paraméter a print() függvényben. Például futtassunk egy programot, amely egy külön fájlban van elmentve egy adott névvel:

```
print('Hello')

print('World')
```

A kimenet így néz ki:

```
Hello

World
```

A szövegek külön sorokban jelennek meg, mivel a **print() függvény automatikusan hozzáadja az új sor karaktereket (\n) a sor végéhez.** Szükség esetén az új sor karaktere helyett más szöveget is használhatunk, amit az end paraméter segítségével adhatunk meg.

Például a következő programban:

```
print('Hello', end='')

print('World')
```

a kimenet így lesz:

```
HelloWorld
```

Ebben az esetben a szöveg egy sorban jelenik meg, mert az új sor karakter (\n) hiányzik a 'Hello' sor után. Az új sor karakter helyett egy üres szöveget ('') jelenít meg. Ez a trükk akkor hasznos, amikor meg akarjuk akadályozni az új sor karakterének (\n) automatikus hozzáadását, amely minden kimeneti sor végén ott van a print() függvénnyel.

Az említett üres szöveget különböző idézőjelekkel is létrehozhatjuk:

```
>>> ''

''

>>> ""

''

>>> """"""
```

```
''
>>> ''''''
''
```

Az üres szöveg létrehozása például akkor szükséges, amikor más szövegekből szeretnénk új szövegeket alkotni:

```
>>> score = 55
>>> base = ''
>>> base += 'checking account: '
>>> base += str(score)
>>> base
'checking account: 55'
```

Formázási stílusok

format() függvény

Nézzük meg, hogyan lehet különböző értékeket elhelyezni egy szövegben különböző formátumok alkalmazásával. Ezt a lehetőséget arra használhatjuk, hogy meghatározzuk, hogyan jelenjen meg az információ kimeneti formája. Határozzunk meg néhány változót: egy egész számú **n**, egy lebegőpontos számú **f**, és egy szöveges változó **s**:

```
>>> n = 25
>>> f = 9.03
>>> s = 'search string'
```

A `format()` függvény segítségével ezek az értékek egy sorban is elhelyezhetők, a megadott sorrendben:

```
>>> '{} {} {}'.format(n, f, s)
'25 9.03 search string'
```

A kimeneti sorrendet a kapcsos zárójelekben megadott indexekkel is meghatározhatjuk (a számozás nullától kezdődik):

```
>>> '{2} {0} {1}'.format(n, f, s)
'search string 25 9.03'
```

A fenti példákban az értékek az alapértelmezett formázással kerülnek kiírásra. Python lehetőséget ad arra, hogy típus

specifikátorokat adjunk meg a *: karakter* után a formázáshoz. Például:

```
>>> '{0:d} {1:f} {2:s}'.format(n, f, s)
'25 7.030000 search string'
```

Típusmeghatározók	
S	Szöveg
D	Egész szám tizedes számrendszerben
F	Lebegőpontos szám tizedes számrendszerben

Ha lebegőpontos számot szeretnénk kiírni, például három tizedesjegy pontossággal, akkor a `:3` szintaxist használhatjuk a `f` típus specifikátor előtt:

```
>>> '{0:d} {1:.3f} {2:s}'.format(n, f, s)
'25 9.030 search string'
```

A format() függvény más lehetőségeket is támogat (minimális mezőhossz, maximális karakter szélesség, igazítás, stb.). Íme néhány példa.

A minimális mezőhossz 10 minden érték esetében, jobb oldali igazítással (alapértelmezés szerint):

```
>>> '{0:10d} {1:10f} {2:10s}'.format(n, f, s)
'        25   9.030000 search string'
```

A minimális mezőhossz 20 minden érték esetében, jobb oldali igazítással:

```
>>> '{0:>20d} {1:>20f} {2:>20s}'.format(n, f, s)
'                  25             9.030000        search string'
```

A minimális mezőhossz 15 minden érték esetében, bal oldali igazítással:

```
>>> '{0:<15d} {1:<15f} {2:<15s}'.format(n, f, s)
'25              9.030000        search string  '
```

A minimális mezőhossz 10 minden érték esetében, középre igazítással:

```
>>> '{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)

'    25      9.030000  search string'
```

Ha azt szeretnénk, hogy a kimeneti mezőt valami más töltse ki a szóközök helyett, helyezzük el ezt a karaktert (a példában az "=" karaktert) közvetlenül a kettőspont után, de a széljegyzés vagy szélességi specifikátor előtt:

```
>>> '{0:=^21s}'.format('My web-page')

'=====My web-page====='
```

f-strings

A Python 3.6-ban megjelent az a mechanizmus, amely lehetővé teszi a tetszőleges értékekkel rendelkező szöveges konstansok létrehozását, úgynevezett **f-strings**. Ez a mechanizmus szöveges interpolációnak nevezhető, és lehetővé teszi Python kifejezések értékeinek beillesztését a szövegekbe.

Az f-strings szintaxisa hasonlít a format metódus szintaxisára, de valamivel nagyobb hatékonysággal. Az alábbi példa némi betekintést ad abba, hogyan működnek az f-strings:

```
>>> name = 'David'

>>> f'My name is {name}'

'My name is David'
```

Ez a mechanizmus lehetővé teszi tetszőleges Python kifejezések beillesztését, például helyi aritmetikai műveletek végrehajtását:

```
>>> a = 12

>>> b = 8

>>> f'12 + 8 = {a + b}'

'12 + 8 = 20'
```

és típus specifikátorok használatát.

```
>>> price = 250.67890

>>> f'your account: {price:.3f}'

'your account: 250.679'
```

Típuskonverzió: str() függvény

Más típusú adatokat Pythonban sztringgé alakíthatunk a **str() függvénnyel:**

```
>>> str(98.6)
'98.6'
>>> str(25)
'25'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

Vezérlő karakterek

Néhány karakterlánc-sorozatot Python különleges módon értelmez, amikor sztringekkel dolgozik: a **\n új sort jelöl, a \t pedig tabulátort.**

A fordított perjel (\) utáni karakterek, amelyek más karakterek ábrázolására szolgálnak, *vezérlő sorozatoknak vagy* ***escape*** *karaktereknek nevezhetők.*

Az escape sorozatok általában arra szolgálnak, hogy speciális karaktereket vegyünk bele a sztringekbe, amelyeknek nincs egy standard, egykarakteres, nyomtatható ábrázolásuk.

A leggyakoribb escape sorozat a \n, amely új sort jelent. Ezzel több soros sztringeket hozhatunk létre egyetlen soros sztringekből:

```
>>> print('Languages:\nPython\nC\nRuby')
Languages:
Python
C
Ruby
```

Gyakran előforduló escape sorozat a \t (tabulátor), amely a szöveg igazítására szolgál a behúzások révén:

```
>>> print('\tasciidoctor')
    asciidoctor
```

```
>>> print('ascii\tdoctor')
ascii    doctor
```

A \' vagy \" sorozatokat használjuk arra, hogy a sztringekbe aposztrófokat (egyes idézőjeleket) vagy idézőjeleket (dupla idézőjeleket) helyezzünk, amikor ugyanazon típusú idézőjelek között vannak:

```
>>> print("The language \"Python\" is named after Monty Python, not the snake.")
The language "Python" is named after Monty Python, not the snake.
>>> print('One of Python\'s strengths is its diverse and supportive community.')
One of Python's strengths is its diverse and supportive community.
```

Ha pedig *visszaperjelet (\)* kell bevinni, akkor előtte egy másik visszaperjelet kell elhelyezni:

```
>>> print('This inverse slash \\')
This inverse slash \
```

Escape karakterek táblázata

Jelölés	Karakterek
\'	Egyes idézőjel
\"	Dupla idézőjel
\\	Visszaperjel
\a	Hangjelzés
\b	Backspace
\n	Új sor
\r	Kocsi visszatérítés

Jelölés	Karakterek
\t	Tabulátor
\v	Függőleges tabulátor

Sztringműveletek

Sztringek összefűzése

Pythonban a **+ operátor** segítségével összevonhatunk sztringeket vagy sztring változókat. Tegyük fel, hogy a név és a vezetéknév külön változókban van tárolva, és össze akarjuk őket fűzni egy teljes név kiírásához:

```
>>> first_name = "Ada"
>>> last_name = "Lovelace"
>>> full_name = first_name + " " + last_name
>>> print(full_name)
Ada Lovelace
```

Ezt az összevonási módot **sztringkonkatinációnak** nevezzük.

Sztringek konkatinálásakor Python nem ad hozzá szóközöket. Ezeket kifejezetten hozzá kell adni.

Gyakran előfordul, hogy nem sztring típusú objektumokat használunk a sztringekben. Tegyük fel, hogy szeretnél születésnapi üdvözletet küldeni a barátodnak. Ehhez a következő kódot írtad, és futtatásra került:

```
>>> age = 23
>>> message = "Happy " + age + "rd Birthday!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Típushiba történt, mivel egész számokat próbáltál közvetlenül sztringekkel kombinálni. Ha egész számokat szeretnél sztringekkel összefűzni, akkor egyértelműen meg kell adni, hogy az egész szám sztringgé legyen konvertálva. Ehhez a `str()` függvényt kell használni, amely nem sztring típusú értékeket alakít sztringgé:

```
>>> age = 23
>>> message = "Happy " + str(age) + "rd Birthday!"
>>> print(message)
Happy 23rd Birthday!
```

Sztringek duplikálása

A *** (csillag) operátor** segítségével duplikálhatjuk a sztringeket. Próbáld ki a következő sorokat az interaktív interpreterben, és elemezd az eredményt:

```
>>> start = 'Na' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start  + middle + end)
NaNaNaNa
Hey Hey Hey
Goodbye.
```

A sztring karaktereihez való hozzáférés indexekkel

A sztringek karaktereihez úgy férhetsz hozzá, hogy megadod a karakter *indexét (helyét) a sztringben*, zárójelek között a sztring neve után:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[6]
'g'
>>> letters[-1]
'z'
>>> letters[36]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Az indexek értékei 0-tól a sztring hossza-1 terjednek.

Slice - részsztringek

A sztringekből nemcsak egy karaktert, hanem egy *részsztringet (a sztring egy részét) is kiemelhetjük a slice függvénnyel*:

slice[start: end: step]

Ennek a függvénynek a zárójelei között a következő paramétereket adhatjuk meg:

- kezdő index (start)
- befejező index (end)
- lépés (step)

Ezek közül néhány paramétert elhagyhatunk. A részsztring tartalmazza azokat a karaktereket, amelyek az adott kezdő index helyétől kezdődnek, és a befejező index helyéig terjednek (de nem tartalmazza a befejező indexet). Például, ha a következő sztringet használjuk, amely az angol kisbetűket tartalmazza:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

Egy karakterlánc az elejétől a végéig:

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

Karakterlánc a 20. karaktertől a végéig:

```
>>> letters[20:]
'uvwxyz'
```

Karakterlánc a 12. karaktertől a 15. karakterig (15. karakter nem kerül bele):

```
>>> letters[12:15]
'mno'
```

Karakterlánc az utolsó 5 karakterből:

```
>>> letters[-5:]
'vwxyz'
```

Karakterlánct a 18. karaktertől a 3-ik karakterig a végéről (3. karakter a végéről nem kerül bele):

```
>>> letters[18:-3]
'stuvw'
```

Karakterlánc a 6. karaktertől a 2. karakterig a végéről (2. karakter a végéről nem kerül bele):

```
>>> letters[-6:-2]
'uvwx'
```

Minden 7. karaktert az elejétől a végéig:

```
>>> letters[::7]
'ahov'
```

Minden 3. karakter a 4. karaktertől a 20. karakterig (20. karakter nem kerül bele):

```
>>> letters[4:20:3]
'ehknqt'
```

Minden 4. karakter a 19. karaktertől a végéig:

```
>>> letters[19::4]
'tx'
```

Minden 5. karakter az elejétől a 21. karakterig (21. karakter nem kerül bele):

```
>>> letters[:21:5]
'afkpu'
```

A sztring a végéről az elejére:

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Helytelen kezdő és befejező indexek esetén üres sztringet kapunk:

```
>>> letters[60:70]
''
```

len() függvény - sztring hossza

A **len() függvény** megszámolja a sztringben található karaktereket:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> len(letters)
26
>>> empty = ""
>>> len(empty)
0
```

A len() függvényt más típusú szekvenciákra is alkalmazhatjuk.

split() függvény - sztringek felosztása

A **split() függvény** szétválasztja a sztringet különböző sztringekre, és egy listában tárolja őket.
A függvény használatakor meg kell adni a **választó karaktert** (például: ,):

```
>>> dolist = 'pass algorithm, write a program, test program'
>>> dolist.split(',')
['pass algorithm', ' write a program', ' test program']
```

Ha nem adunk meg választó karaktert, a split() függvény a szóközt, új sort és tabulátort választó karakterként kezel:

```
>>> dolist.split()
['pass', 'algorithm,', 'write', 'a', 'program,', 'test',
'program']
```

A split() függvény hívása esetén, még ha nem is adunk meg argumentumot, akkor is szükség van a zárójelekre - így tudja Python, hogy függvényt hívunk.

join() metódus: listából sztringet

*A **join()** metódus* ellentéte a split() függvénynek – összefűzi a lista sztringjeit egyetlen sztringgé.

Általánosan alkalmazható az iterálható objektumok (például *listák, tuple-k, halmazok*) összefűzésére egyetlen sztringgé, azzal a sztringgel, amelyből a metódust meghívjuk, mint választó karakterrel:

separator.join(iterable)

- *separator* — a választó karakter, amelyet az elemek közé illesztünk.
- *iterable* — az iterálható objektum, amely azokat az elemeket tartalmazza, amelyeket összefűzünk (például lista, tuple vagy sztring).

A join() hatékony, ha nagy mennyiségű sztringet kell összefűzni, mivel gyorsabb, mint a sztringek konkatenálása a + operátorral.

Lista elemeinek összefűzése egy sztringbe

Ha van egy lista sztringekből, és szeretnénk őket egy sztringgé összefűzni valamilyen választóval, használd a `join()`-t.

```
words = ["Hello", "world", "Python"]
result = " ".join(words)
print(result)  #  "Hello world Python"
```

Itt a " " (szóköz) van választó karakterként, és a lista elemei egy sztringgé fűződnek össze, szóközökkel elválasztva.

A következő példában több nevet fűzünk össze egy listában, amelyet vesszők és szóközök választanak el:

```
>>> crypto_list = ['Jumanji: Welcome to the Jungle', 'The Lost City of Z', 'Justice League']

>>> crypto_string = ', '.join(crypto_list)

>>> print('Browse films:', crypto_string)

Browse films: Jumanji: Welcome to the Jungle, The Lost City of Z, Justice League
```

Még egy példa.

```
words = ["apple", "banana", "cherry"]
result = ", ".join(words)
print(result)  #  "apple, banana, cherry"
```

Karakterek összefűzése egy karakterláncban

A join() metódus használható karakterek egy karakterláncba történő összekapcsolására is.

```
chars = ['a', 'b', 'c']
result = "-".join(chars)
print(result)  #  "a-b-c"
```

Számok összefűzése a sztringekkel

A join() metódust nem lehet használni olyan elemekkel, amelyek nem sztringek (például számokkal, listákkal stb.), ezért ezeket először sztringgé kell alakítani.

Példa. A join() metódus csak sztringekkel működik, ezért a számokat előbb sztringgé kell alakítani. A map(str, numbers) segítségével minden listaelem sztringgé alakítható az összefűzés előtt.

```
numbers = [1, 2, 3, 4]
result = "-".join(map(str, numbers))
print(result)  #  "1-2-3-4"
```

Még néhány példa.

```
oc = ['10', '1', '1', '1']
sep = "."
s = sep.join (oc)
# '10.1.1.1'
mac = ["AAAA", "BBBB", "CCCC"]
st = “”.jo in(mac)
# 'AAAABBBBCCCC'
```

Sztringek igazítása

Nézzünk még néhány példát a Python beépített függvényeinek használatára a sztringekkel való munkához. Hozzunk létre egy sztring változót a következő tartalommal:

```
>>> s = 'beetles buzzing over cherries...'
```

Távolítsuk el a *. karaktereket* a sztring elejéről és végéről a **strip() függvénnyel.**

A **rstrip() és lstrip() függvények** eltávolítják a szóközöket, valamint az új sort és tabulátort a sztring elejéről és végéről.

```
>>> s.strip('.')

'beetles buzzing over cherries'

>>> s

'beetles buzzing over cherries...'
```

Mivel a sztringek változtathatatlanok, a **s** sztring nem változik. Az értékét veszik, végrehajtanak rajta műveleteket, majd az eredményként új sztringet adnak vissza.

A sztringek formázása

Írjuk át minden szót nagy kezdőbetűvel a **title() függvénnyel:**

```
>>> s.title()

'Beetles Buzzing Over Cherries...'
```

Írjuk az első szót nagy kezdőbetűvel, a többit kisbetűkkel a **capitalize() függvénnyel:**

```
>>> s.capitalize()

'Beetles buzzing over cherries...'
```

Írjuk az összes szót nagybetűvel a **upper() függvénnyel:**

```
>>> s.upper()

'BEETLES BUZZING OVER CHERRIES...'
```

Írjuk az összes szót kisbetűvel a **lower() függvénnyel:**

```
>>> s.lower()

'beetles buzzing over cherries...'
```

Váltsuk fel a betűk kis- és nagybetűit a **swapcase() függvénnyel:**

```
>>> s = 'Beetles buzzing over Cherries...'

>>> s.swapcase()
'bEETLES BUZZING OVER cHERRIES...'
```

A sztringek igazításához a következő függvényeket használhatjuk: **center(), ljust(), rjust().**

Középre igazítás 40 karakteres helyen:

```
>>> s.center(40)

'    beetles buzzing over cherries...    '
```

Balra igazítás 40 karakteres helyen:

```
>>> s.ljust(40)

'beetles buzzing over cherries...        '
```

Jobbra igazítás 40 karakteres helyen:

```
>>> s.rjust(40)

'        beetles buzzing over cherries...'
```

replace() függvény: karakterek cseréje

A replace() függvényt arra használhatjuk, hogy egy részsztringet kicseréljük egy másikra.

Ehhez át kell adnunk a függvénynek a "régi részsztringet" (amit cserélünk), az "új részsztringet" (amire cserélni szeretnénk), és a cserék számát:

sztring.replace(régi részsztring, új részsztring, cserék száma)

Ha nem adjuk meg az utolsó argumentumot (a cserék számát), akkor az összes előfordulást kicseréli. Például:

```
>>> language = 'I told my friend, "Python is my favorite language! I love to program in Python!"'
>>> language.replace('Python', 'Ruby')
'I told my friend, "Ruby is my favorite language! I love to program in Ruby!"'
```

Cseréljük ki a ! karaktert !!!-ra egy vagy két előfordulás esetén:

```
>>> language.replace('!', '!!!', 1)
'I told my friend, "Python is my favorite language!!! I love to program in Python!"'
>>> language.replace('!', '!!!', 2)
'I told my friend, "Python is my favorite language!!! I love to program in Python!!!"'
```

Metodus	Alkaalmazása
`s.capitalize()`	Visszaadja a s sztring másolatát, ahol az első betű nagybetűs.
`s.lower()`	Visszaadja a s sztring másolatát kisbetűs karakterekkel.
`s.swapcase()`	Visszaadja a s sztring másolatát, amelyben minden betű a fordított esetben lesz.
`s.title()`	Visszaadja a s sztring másolatát, ahol minden új szó első betűje nagybetűs.
`s.upper()`	Visszaadja a s sztring másolatát, amelyben minden karakter nagybetűs.
`s.count(x)`	A s sztring esetében visszaadja, hányszor szerepel benne a megadott részstring x.
`s.find(x)`	Visszaadja a legkisebb indexet a s sztringben, ahol a megadott részstring x kezdődik (ha x nincs benne, -1-et ad vissza).

Metodus	Alkaalmazása
s.index(x)	Visszaadja a legkisebb indexet a s sztringben, ahol a megadott részstring x kezdődik (ha nincs benne, akkor ValueError kivételt dob).
s.rfind(x)	Visszaadja a legnagyobb indexet a s sztringben, ahol a megadott részstring x kezdődik (ha x nincs benne, -1-et ad vissza).
s.rindex(x)	Visszaadja a legnagyobb indexet a s sztringben, ahol a megadott részstring x kezdődik (ha nincs benne, akkor ValueError kivételt dob).
s.replace(a, b)	Visszaadja a s sztring másolatát, ahol az a részstring minden előfordulása kicserélődik a b részstringre.
s.startswith(x)	Visszaadja a True értéket, ha a s sztring a megadott x prefixummal kezdődik (ha nem, akkor False).
s.endswith(x)	Visszaadja a True értéket, ha a s sztring a megadott x suffixummal végződik (ha nem, akkor False).
s.join(x)	Visszaadja azt a sztringet, amelyet az x iterálható objektumból alkotunk, az s karakterekkel elválasztva.
s.lstrip(chars)	Visszaadja a s sztring másolatát, amelyből a chars karakterek az elejéről eltávolításra kerültek.
s.rstrip(chars)	Visszaadja a s sztring másolatát, amelyből a chars karakterek a végéről eltávolításra kerültek.
s.strip(chars)	Visszaadja a s sztring másolatát, amelyből a chars karakterek az elejéről és a végéről is eltávolításra kerültek.
s.split(char)	A s sztringet elválasztja a char karakter mentén, és lista formájában tárolja.

Metodus	Alkaalmazása
`s.center(width, fill)`	Visszaadja a középre igazított sztringet, amelynek szélén a fill karakterek (alapértelmezett: szóköz) találhatók.
`s.ljust(width, fillchar)`	A s sztringet olyan hosszúságúra állítja, hogy elérje a width értéket, szükség esetén a végén fillchar karakterekkel kitöltve.
`s.rjust(width, fillchar)`	A s sztringet olyan hosszúságúra állítja, hogy elérje a width értéket, szükség esetén az elején fillchar karakterekkel kitöltve.
`s.isalnum()`	Azt vizsgálja, hogy a s sztring betűkből és számjegyekből áll-e.
`s.isalpha()`	Azt vizsgálja, hogy a s sztring csak betűkből áll-e.
`s.isdigit()`	Azt vizsgálja, hogy a s sztring csak számjegyekből áll-e.
`s.istitle()`	Azt vizsgálja, hogy a s sztringben a szavak nagybetűvel kezdődnek-e.
`s.isupper()`	Azt vizsgálja, hogy a s sztring csak nagybetűs karaktereket tartalmaz-e.
`s.islower()`	Azt vizsgálja, hogy a s sztring csak kisbetűs karaktereket tartalmaz-e.

Megjegyzés

A Python-ban a sztringek (str típus) *változtathatatlanok* (*immutable*). A sztringek változtathatatlansága azt jelenti, hogy ha egyszer létrehozol egy sztringet, annak a tartalmát nem lehet módosítani. Bármilyen művelet, amely "megváltoztatja" a sztringet, valójában egy új sztringet hoz létre.

Példa

```
s = "hello"
s[0] = "H"
# Ez hibát fog okozni!
TypeError: 'str' object does not support item assignment
```

De a régi sztrinből létrehozhatunk új sztringet:

```
s = "hello"

s = "H" + s[1:]

print(s)

# "Hello"
```

Ebben az esetben a "Hello" egy teljesen új sztring, és a s változó mostantól erre mutat.

Ez a tulajdonság hasznos lehet több szempontból is.

1. **Biztonság és stabilitás:** Ha sztringek változtathatóak lennének, akkor például ha egy szótár (*dict*) kulcsa lenne egy sztring, és valaki megváltoztatná a sztring tartalmát, az tönkretenné a szótár működését.
2. **Teljesítmény:** A változtathatatlan objektumokat könnyebb optimalizálni. Például a Python belsőleg újra tudja használni ugyanazokat a sztring objektumokat, ha azok megegyeznek.
3. **Hash-elhetőség:** Csak változtathatatlan objektumok használhatók kulcsként a szótárakban és elemekként a set-ekben, mert ezeknek az objektumoknak állandó hash-értékük van. A sztringek ilyenek.

2.4. Adatbevitel a billentyűzetről

A programok futtatásához szükséges bizonyos információ, amelyet a felhasználónak kell megadnia. Hogyan kérhetjük be a felhasználótól az adatokat, hogy a program tudjon velük dolgozni?

input() függvény

Az adatbevitelhez a programokban az **input() függvényt** használjuk. Az input() függvény leállítja a program futtatását, és megvárja, amíg a felhasználó beír egy szöveget és megnyomja az Entert.

Minden alkalommal, amikor az input() függvényt használjuk a programunkban, érthető üzenetet kell biztosítanunk (például: "Kérem, adja meg a nevét: "), amely világosan elmondja a felhasználónak, milyen információt szeretnénk tőle.

Például, az alábbi kódrészletben adjunk hozzá egy szóközt a kérés végére (a kettőspont után), hogy elválasszuk a kérdést a felhasználó által megadott adatokat, és egyértelműen mutassuk, hol kell beírni a szöveget:

```
>>> name = input("Please enter your name: ")
Please enter your name: Alex
>>> print("Hello, " + name + "!")
Hello, Alex!
```

Az input() függvény használatakor a Python minden általunk beírt adatot szövegként (string) kezel. A következő példában a program a felhasználótól kéri a korát:

```
>>> age = input("How old are you? ")
How old are you? 25
>>> age
'25'
```

A felhasználó beírja a 25-öt, de amikor a Python megkérdezi az age változó értékét, az '25'-ként jelenik meg – a beírt szám szöveges formátumban. Az idézőjelek azt jelzik, hogy a Python szövegként értelmezi a bevitt adatokat.

Ha próbáljuk a beírt adatokat számként használni, hiba történik:

```
>>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() >= int()
```

Ha próbáljuk összehasonlítani a beírt adatokat egy számmal, a Python hibát ad, mivel nem tud összehasonlítani egy szöveget egy számmal.

A problémát az **int() függvénnyel** oldhatjuk meg, amely a szöveges adatokat egész számként értelmezi. Az int() függvény átalakítja a szöveges számot egész számra.

```
>>> age = input("How old are you? ")
How old are you? 25
>>> age = int(age)
>>> age >= 18
True
```

Ebben a példában a beírt 25-öt a Python szövegként értelmezi, de azután az int() segítségével egész számra

alakítjuk. Most a Python ellenőrizheti a feltételt: összehasonlítani a age változót (amely most már egész számként 25-öt tartalmaz) a 18-cal.

Ha lebegőpontos számot szeretnénk kapni a beírt szövegből, akkor a **float() függvényt** használjuk. A float() függvény átalakítja a szöveges számot lebegőpontos számmá.

```
>>> value = float(input('Enter a number: '))
Enter a number: 45.5
>>> value
45.5
```

Ebben az esetben a szintaxis két függvény, az input() és a float() kombinálásával történik: először történik az adatbevitel, majd a szöveg értelmezése, végül pedig a szöveg átalakítása lebeőpontos számmá.

3. PYTHON ADATSZERKEZETEK

3.1.Listák

A listák lehetővé teszik, hogy egy helyen tároljuk az összefüggő adatokat, bárhány darabról legyen szó — akár néhány elemről, akár több millió elemből álló listáról. A listákkal való munka a Python egyik legkiemelkedőbb lehetőségei közé tartozik.

Egy listához új elemeket lehet hozzáadni, valamint a létező elemeket törölni vagy felülírni. Ugyanaz az érték többször is előfordulhat a listában.

A listát létrehozhatjuk az ábécé betűinek, a 0-tól 9-ig terjedő számoknak, vagy például a családtagjaink neveinek tárolására. A listában bármilyen adatot tárolhatunk, és az adatok nem szükségesek, hogy összefüggésben álljanak egymással. Mivel a lista jellemzően több elemet tartalmaz, célszerű a listáknak többes számú neveket adni: letters, digits, names stb.

Listák létrehozása

A Python nyelvben a **listát szögletes zárójelek [] jelölik,** és az elemeket vesszővel választjuk el egymástól. Például így hozhatunk létre egy listát autómárkák neveivel:

```
>>> cars = ['Alfa Romeo', 'Volvo', 'Lamborghini', 'BMW',
'Toyota']
>>> print(cars)
['Alfa Romeo', 'Volvo', 'Lamborghini', 'BMW', 'Toyota']
>>> empty_list = []
>>> weekdays = ['Monday', 'Friday', 'Wednesday', 'Thursday',
'Friday']
>>> animals = ['camels', 'bats', 'elephants', 'dolphins',
'bears']
>>> first_names = ['Mira', 'John', 'Terry', 'John', 'Michael']
>>> another_empty_list = list()
>>> another_empty_list
[]
```

A listák a Pythonban összehasonlíthatók az egyéb programozási nyelvekben használt tömbökkel. Ha egy listát több változónak is hozzárendelünk, akkor a lista módosítása egy helyen annak változását vonja maga után más változókban is, ahogyan az az alábbi példában is látható:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

A lista értéke másolható egy független új listába az alábbi módok egyikével:

- copy() függvénnyel
- list() függvénnyel
- a lista szeletelésével [:].

Például az eredeti lista az "a" változóhoz lesz hozzárendelve, a többi lista, mint például a "b", "c" és "d", pedig az "a" lista másolatai lesznek:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

b, c, d — ezek új objektumok, amelyek saját értékekkel rendelkeznek, amelyek nincsenek összefüggésben az eredeti lista [1, 2, 3] elemeivel, amelyekre az "a" változó hivatkozik. Az "a"-ban végzett módosítások nem befolyásolják a b, c, d másolatokat:

```
>>> a[0] = 'lists store any information, not just integer numbers'
```

```
>>> a
['lists store any information, not just integer numbers', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

Amikor lista másolatokat hozunk létre a copy() és list() függvényekkel, vagy a [:] szintaxis segítségével, az eredeti lista módosítása nem hat a másolatokban lévő változásokra, mivel a másolatok már új objektumok.

len() függvény - lista hossza

A **len() függvény** visszaadja a lista elemeinek számát. Például, határozzuk meg a bolygók neveit tartalmazó lista elemeinek számát:

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> len(planets)
5
```

list() függvény - típus átalakítása

A list() függvény más típusú adatokat alakít át listává. A következő példában egy karakterlánc listává alakul, amely egy karakteres karakterláncokból áll:

```
>>> list('sun')
['s', 'u', 'n']
```

A karakterláncot listává alakíthatjuk a split() függvénnyel is, ha megadjuk a *szeparáló karakterláncot*:

```
>>> mybirthday = '3/4/1981'
>>> mybirthday.split('/')
['3', '4', '1981']
```

A split() függvény a karakterláncot egy másik szeparátor karakterlánc által elválasztott lista elemekre bontja.

Ha a karakterláncban több egymást követő szeparátor karakterlánc is szerepel, akkor a lista elemként üres karakterláncot tartalmaz:

```
>>> result = 'a|b||c|d|||e'
>>> result.split('|')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

A lista karakterlánccá alakításához a **join() függvényt** használjuk.

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> ', '.join(planets)
'Mercury, Jupiter, Earth, Mars, Venus'
```

Hozzáférés a lista elemeihez

A listák rendezett adatgyűjtemények, ezért a lista bármely eleméhez való hozzáféréshez meg kell adni a szükséges elem **pozícióját (indexét)**. Az indexek csak egész számú értékeket vehetnek fel. Az elem eléréséhez meg kell addni a lista nevét, majd zárójelek között az elem indexét.

Az indexek 0-tól kezdődnek, nem pedig 1-től.

Ez az elv a legtöbb programozási nyelvben előfordul. A Pythonban ezt az alacsony szintű lista-kezelés sajátosságai magyarázzák. Ha váratlan eredményeket kapunk, akkor ellenőrizni kell, hogy nem történt-e "1-es eltolás" hiba.
Tehát az első listaelem indexe 0. A második elem indexe 1, és így tovább. Például, ha a lista negyedik eleméhez szeretnénk hozzáférni, akkor az index 3.

Ahogyan a karakterláncok esetében, a listából egy adott értéket is elérhetünk, ha megadjuk az indexét:

```
>>> cars = ['Alfa Romeo', 'Volvo', 'Lamborghini', 'BMW',
'Toyota']
>>> print(cars[1])
Volvo
>>> print(cars[4])
Toyota
>>> print(cars[0])
```

```
Alfa Romeo
>>> print(cars[-1])
Toyota
>>> print(cars[-2])
BMW
>>> print(cars[5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Az **index() függvény** a lista elemének értéke alapján visszaadja annak az indexét.

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> planets.index('Earth')
2
```

Lista a listákban

A listák különböző típusú elemeket is tartalmazhatnak, beleértve más listákat is. Hozzunk létre három listát, és tegyük őket egy negyedik listába:

```
>>> text_editors = ['Atom', 'Sublime Text']
>>> programming_language = ['Python', 'C', 'JavaScript']
>>> programmers = ['Guido van Rossum', 'Dennis Ritchie',
'Brendan Eich']
>>> all_info = [text_editors, 'code', programming_language,
programmers]
>>> all_info
[['Atom', 'Sublime Text'], 'code', ['Python', 'C',
'JavaScript'], ['Guido van Rossum', 'Dennis Ritchie', 'Brendan
Eich']]
```

Nézzük meg az első és harmadik elemet az újonnan létrehozott all_info listában – ezek listák:

```
>>> all_info[0]
['Atom', 'Sublime Text']
```

```
>>> all_info[2]
['Python', 'C', 'JavaScript']
```

Ahhoz, hogy hozzáférjünk az all_info listából például a *programmers* lista első eleméhez, két indexet kell megadnunk: [3] – ez a all_info lista negyedik elemére mutat, míg [0] – az első elemre a belső programmers listában:

```
>>> all_info[3][0]
'Guido van Rossum'
```

A lista elemeinek módosítása

Ellentétben a karakterláncokkal, a lista módosítható:

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> planets[2] = 'Saturn'
>>> planets
['Mercury', 'Jupiter', 'Saturn', 'Mars', 'Venus']
```

A lista elemeinek szeparálása és fordított sorrendje

A lista szeletelésével egy részhalmazt kapunk, amely szintén lista lesz:

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> planets[0:3]
['Mercury', 'Jupiter', 'Earth']
>>> planets[::2]
['Mercury', 'Earth', 'Venus']
>>> planets[::-3]
['Venus', 'Jupiter']
>>> planets[::-1]
['Venus', 'Mars', 'Earth', 'Jupiter', 'Mercury']
```

Elem hozzáadása a listához

A listaelemek hozzáadásának hagyományos módja az **append() függvény** használata, amellyel elemet adhatunk hozzá a lista végéhez:

```
>>> planets.append('Uranus')
>>> planets
['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus', 'Uranus']
```

Ha egy adott pozícióra kell elemet hozzáadni, használhatjuk az **insert() függvényt.** Ha 0-át adunk meg pozícióként, az elem a lista elejére kerül. Ha a pozíció kívül esik a listán, az elem a lista végére kerül, akárcsak az append() függvénynél, ezért nem kell aggódnod amiatt, hogy Python hibát fog dobni:

```
>>> planets.insert(3, 'Neptune')
>>> planets
['Mercury', 'Jupiter', 'Earth', 'Neptune', 'Mars', 'Venus',
'Uranus']
>>> planets.insert(10, 'Unknown planet')
>>> planets
['Mercury', 'Jupiter', 'Earth', 'Neptune', 'Mars', 'Venus',
'Uranus', 'Unknown planet']
```

extend() függvény - lista összevonása

Két lista összevonásához használjuk az **extend() függvényt.**

```
>>> planets = ['Earth', 'Mars']
>>> satellites = ['Moon', 'Deimos', 'Phobos']
>>> planets.extend(satellites)
>>> planets
['Earth', 'Mars', 'Moon', 'Deimos', 'Phobos']
```

A listák összevonására az += operátort is használhatjuk:

```
>>> planets = ['Earth', 'Mars']
>>> satellites = ['Moon', 'Deimos', 'Phobos']
>>> planets += satellites
>>> planets
['Earth', 'Mars', 'Moon', 'Deimos', 'Phobos']
```

Ha az append() függvényt használjuk, a satellites lista egyetlen elemként kerülne hozzáadásra a listához, ahelyett, hogy az elemei összeolvadnának a planets listával:

```
>>> planets = ['Earth', 'Mars']
>>> satellites = ['Moon', 'Deimos', 'Phobos']
>>> planets.append(satellites)
>>> planets
['Earth', 'Mars', ['Moon', 'Deimos', 'Phobos']]
```

Ez újra demonstrálja, hogy a lista különböző típusú elemeket is tartalmazhat.

Elemek eltávolítása a listából

Az elemek eltávolításához a listából a del() függvényt használjuk.

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> del planets[-1]
>>> planets
['Mercury', 'Jupiter', 'Earth', 'Mars']
```

Amikor egy adott elemet eltávolítanak, minden más elem, ami utána következik, balra tolódik, hogy elfoglalja az eltávolított elem helyét, és a lista hossza eggyel csökken.

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> planets[3]
'Mars'
>>> del planets[3]
>>> planets
['Mercury', 'Jupiter', 'Earth', 'Venus']
>>> planets[3]
'Venus'
```

A listából egy elemet nemcsak az indexével, hanem annak értékével is eltávolíthatunk a **remove() függvény** segítségével. Ez akkor hasznos, ha nem tudjuk, hogy melyik pozícióban található az elem, de ismert annak értéke:

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']
>>> planets.remove('Jupiter')
>>> planets
['Mercury', 'Earth', 'Mars', 'Venus']
```

Egy elemet a listából úgy is eltávolíthatunk, hogy közben visszakapjuk annak értékét **a pop() függvénnyel.** Ha megadunk egy indexet, akkor az adott pozícióban lévő elemet adja vissza; ha nem adunk meg argumentumot, akkor az alapértelmezett -1 érték kerül használatra, ahogy az alábbi példában is:

```
>>> planets = ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Venus']

>>> planets.pop()

'Venus'

>>> planets

['Mercury', 'Jupiter', 'Earth', 'Mars']

>>> planets.pop(1)

'Jupiter'

>>> planets

['Mercury', 'Earth', 'Mars']
```

Ellenőrzés: van-e ez az elem a listában?

A Pythonban az elem jelenlétét a listában az **in operátor** segítségével ellenőrizhetjük:

```
>>> countries = ['England', 'France', 'Italy', 'Ukraine',
'Poland']

>>> 'Poland' in countries

True

>>> 'Hungary' in countries

False
```

Ha azt szeretnénk ellenőrizni, hogy nincs-e elem a listában, a **not in operátor** kombinációját használhatjuk:

```
>>> countries = ['England', 'France', 'Italy', 'Ukraine',
'Poland']

>>> 'Ukraine' not in countries

False

>>> 'Portugal' not in countries

True
```

count() függvény - egy érték száma a listában

A listában egy érték előfordulásainak számát a count() függvénnyel tudjuk meghatározni.

```
>>> fruits = ['blueberry', 'grape', 'orange', 'plum', 'pear', 'blueberry', 'pear', 'melon', 'pear']

>>> fruits.count('grape')

1

>>> fruits.count('peach')

0

>>> fruits.count('pear')

3

>>> fruits.count('blueberry')

2
```

Listák rendezése

A lista elemeinek sorrendjét a Pythonban két függvénnyel módosíthatjuk:
sort() – rendezi a listát (megváltoztatja a listát)
sorted() – visszaadja a lista rendezett másolatát (anélkül, hogy megváltoztatná az eredeti listát.

Ha a lista elemei számok, akkor alapértelmezés szerint növekvő sorrendbe lesznek rendezve. Ha karakterláncok, akkor ábécé sorrendben.

A következő lista példáján:

```
>>> clothes = ['shirt', 'hat', 'jeans', 'trainers']

>>> sorted_clothes = sorted(clothes)

>>> sorted_clothes

['hat', 'jeans', 'shirt', 'trainers']

>>> clothes

['shirt', 'hat', 'jeans', 'trainers']
```

A sorted_clothes a clothes lista másolata, a létrehozása nem módosította az eredeti clothes listát.

A sort() függvény használata viszont maga a lista módosítását végzi el:

```
>>> clothes.sort()

>>> clothes

['hat', 'jeans', 'shirt', 'trainers']
```

A példákban minden listaelem azonos típusú (karakterláncok). Néha keverhetjük a típusokat is – például egész számokat és lebegőpontos számokat, mivel a rendezés során ezek automatikusan konvertálódnak:

```
>>> numbers = [5, 3, 7.0, 4]

>>> numbers.sort()

>>> numbers

[3, 4, 5, 7.0]
```

Azonban ha a lista vegyesen tartalmaz számokat és karakterláncokat, hiba lép fel:

```
>>> numbers = [3, 1, 'start', 2]

>>> numbers.sort()

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: '<' not supported between instances of 'str' and
'int'
```

A sort() és sorted() függvények használatakor az összes listában szereplő elemnek kompatibilis típusúnak kell lennie. Alapértelmezés szerint a lista növekvő sorrendbe lesz rendezve, de ha a **reverse=True** argumentumot adjuk hozzá, a lista csökkenő sorrendbe rendeződik:

```
>>> numbers = [5, 3, 7.0, 4]

>>> numbers.sort(reverse=True)

>>> numbers

[7.0, 5, 4, 3]
```

Lista visszafordított sorrendje

A lista elemeit visszafordított sorrendbe tenni **a reverse() függvény** segítségével lehet.

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']

>>> cars
```

```
['bmw', 'audi', 'toyota', 'subaru']
>>> cars.reverse()
>>> cars
['subaru', 'toyota', 'audi', 'bmw']
```

Numerikus listák létrehozása

A számhalmazok tárolásának szükségessége sokféle programban előfordul. Például egy számítógépes játékban a karakterek képernyőn lévő koordinátáit, rekordtáblázatokat, számlálásokat tárolhatunk. A számítási típusú programok mindig számhalmazokkal dolgoznak: hőmérséklet, távolság, népesség stb.
A listák ideálisak számhalmazok tárolására, és a Python speciális eszközöket biztosít a numerikus listákkal való hatékony munkához, még akkor is, ha a lista több millió elemet tartalmaz.
A numerikus sorozatok egyszerűbb létrehozásához a **range() függvényt** használjuk:

```
>>> numbers = list(range(1,6))
>>> numbers
[1, 2, 3, 4, 5]
```

A range() függvény lehetővé teszi egész számú sorozatok generálását egy adott tartományon belül.
Például a 1-től 10-ig terjedő páros számok listájának létrehozásához az alábbi kódot használhatjuk:

```
>>> even_numbers = list(range(2,11,2))
>>> even_numbers
[2, 4, 6, 8, 10]
```

A Python néhány beépített függvényt biztosít numerikus listák kezelésére. Például könnyen megtudhatjuk a minimumot, maximumot és az elemek összegét a numerikus listák esetében:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

"Listák metódusai" táblázat	
Metódus	**Alkalmazása**
`list.append(x)`	Hozzáadja az x elemet a lista végére
`list.extend(L)`	Hozzáadja a lista végéhez az összes elemet az L listából
`list.insert(i, x)`	Beszúrja az x értéket a lista i-edik elemének helyére
`list.remove(x)`	Eltávolítja a lista első olyan elemét, amelynek értéke x; ValueError, ha nincs ilyen elem
`list.pop(i)`	Eltávolítja az i-edik elemet a listából és visszaadja azt; ha nincs megadva index, akkor az utolsó elem kerül eltávolításra
`list.index(x, start, end)`	Visszaadja az x értékű első elem pozícióját a listában (a keresés start és end között történik)
`list.count(x)`	Visszaadja, hogy hányszor szerepel az x érték a listában
`list.sort(reverse=False)`	Rendezi a lista elemeit; ha a fordított rendezést kéri, a reverse paramétert True-ra kell állítani
`list.reverse()`	Megfordítja a lista elemeit
`list.copy()`	A lista felszíni másolata

"Listák metódusai" táblázat	
Metódus	**Alkalmazása**
`list.clear()`	Törli a lista összes elemét

3.2. Tuplek

A tuplek, akárcsak a listák, olyan sorrendek, amelyek tetszőleges elemeket tartalmaznak. A listákkal ellentétben a tuplek változtathatatlanok. Ez azt jelenti, hogy nem lehet elemeket hozzáadni, törölni vagy módosítani egy tuple-t, miután az létrejött (például nem lehet véletlenül törölni egy tuple elemét).

Üres tuple létrehozásához a () operátort használjuk:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

Egy olyan tuple létrehozásához, amely egyetlen elemet tartalmaz, a zárójel után vesszőt kell tenni. Így néz ki a tuple, amely csak egy elemet tartalmaz:

```
>>> currency_units = 'pound',
>>> currency_units
('pound',)
```

A tuple megjelenítésekor a Python a zárójeleket is kiírja.

Ha a tuple több mint egy elemet tartalmaz, akkor minden elem után vesszőt kell tenni, kivéve az utolsót:

```
>>> currency_units = 'pound', 'dollar', 'euro'
>>> currency_units
('pound', 'dollar', 'euro')
```

A tuplek lehetővé teszik több változó értékének egyszerre történő hozzárendelését:

```
>>> currency_units = 'pound', 'dollar', 'euro'
>>> p, d, e = currency_units
>>> p
'pound'
>>> d
'dollar'
>>> e
'euro'
```

A tuplek segítségével értékeket cserélhetünk egyetlen kifejezés segítségével, anélkül, hogy ideiglenes változót alkalmaznánk:

```
>>> network_name = 'netis'
>>> password = 'pass12345'
>>> network_name, password = password, network_name
>>> network_name
'pass12345'
>>> password
'netis'
```

A **tuple() függvény** más objektumokból tuple-t hoz létre:

```
>>> currency_units = ['pound', 'dollar', 'euro']
>>> tuple(currency_units)
('pound', 'dollar', 'euro')
```

Miért léteznek külön tuplek és listák?

A fő különbség az objektumok között a változtathatóságban rejlik. Mivel a **tuplek változtathatatlanok, kulcsokként használhatók szótárakban.**

A tuplek gyakran adatbejegyzések reprezentálására szolgálnak – például adatbázis lekérdezéseket, amelyek különböző típusú objektumokat tartalmaznak:

```
>>> query = ('localhost', 'admin', 12345)
```

A tuplek több elem visszaadására is használhatók a függvényekből. A tuplek kevesebb memóriát használnak, mint a listák. Ha olyan sorrendekkel dolgozol, amelyek nem változnak, talán érdemes tuple-okat használni a memória megtakarítása érdekében.

3.3. Szótárak

A szótárak adatszerkezetek, amelyek összefüggő információk egyesítésére szolgálnak. A szótárak lehetővé teszik a valós objektumok modellezését. Például készíthetünk egy szótárt, amely egy "személy" objektumot ír le, és elmenthetünk benne annyi információt, amennyit csak szeretnénk. Ez lehet név, életkor, lakóhely, foglalkozás és bármilyen más jellemző.

A szótárakban gyakran tárolnak kétféle adatot, amelyek párokat alkotnak: például szavak és jelentéseik listája, országok és fővárosaik listája, hét napjai és hőmérsékleti értékek stb.

Minden szótári értékhez egy egyedi kulcs tartozik. A kulcs alapvetően egy karakterlánc, de lehet egy változtathatatlan típusú objektum is: *például egy logikai érték, egész szám, lebegőpontos szám, tuple* stb.

Szótár kulcsai

A kulcsok különböző típusúak lehetnek. Az egyetlen követelmény a kulcsokkal kapcsolatban a **hash-elhetőség**. Például a lista nem hash-elhető, mert változtatható, és a Python nem tud generálni hozzá hash értéket, amely megfelel a jelenlegi tartalmának.

Hashről röviden

1. A hash (angolul: hash) egy adatcsomagon végrehajtott hash-elési algoritmus eredménye. Ez egy egyedi (vagy szinte egyedi) numerikus érték fix hosszúsággal, amelyet a bemeneti adatok alapján generálnak. A hash karakterláncként vagy számmal jelenhet meg.
2. A hashelés egy olyan folyamat, amely során a bemeneti adatokat (például szöveget, fájlt, üzenetet) hash-vá alakítják. A hashelési algoritmusok (például MD5, SHA-1, SHA-256) segítségével egy fix hosszúságú hash értéket kapunk, függetlenül a bemeneti adatok méretétől.
3. A hash-elhetőség a hash függvények tulajdonsága. Ez azt jelenti, hogy még egyetlen karakter megváltoztatása is teljesen más hash értéket eredményez. A hash-elésnek ellenállónak kell lennie az ütközésekkel szemben, azaz nem

lehet két különböző bemeneti adat, amely ugyanazt a hash értéket generálja.

A hashelést sok területen használják, például:

- Jelszavak titkos tárolására

- Az adatok integritásának ellenőrzésére (például fájlok átvitele előtt és után összehasonlítják a hash értékeket).

A hashelés technológiája fontos az adatok biztonságának és az illetéktelen hozzáférés elleni védelem érdekében.

Szótárba elemeket úgy adhatunk hozzá, hogy egész számokat vagy karakterláncokat használunk kulcsként, de általában nem ajánlott különböző típusú kulcsokat keverni, mert ez megnehezíti a kód olvasását és a rendezést. *A szótár célja, hogy gyors keresést biztosítson kulcs alapján a hozzá tartozó értékekhez.*

Hogyan tárolódnak az értékek a szótárban?

Emlékezzünk arra, hogy a lista értékei implicit módon sorrendbe vannak rendezve a pozíciójuk alapján, mivel az indexek, amelyeket az értékek eléréséhez használunk, egész számok.

A Python 3.6 előtt a szótárak kulcsainak sorrendje tetszőleges lehetett, és nem függött attól, hogyan adtunk hozzá elemeket. A Python 3.6-ban a szótárak megjegyzik az elemek hozzáadásának sorrendjét. A Python 3.7-től ez már a Python nyelv specifikációjának része.

A szótárak változtathatók, ami azt jelenti, hogy elemeket hozzáadhatunk, törölhetünk és módosíthatunk. Ha a kulcs már létezik, az aktuális érték helyére új érték kerül. Ha a kulcs új, akkor a kulcs és az érték is hozzáadódik a szótárhoz.

A **„kulcs: érték”** sorrendek példája a következő típusú sorozatok:

Szilícium: 28, Szén: 12

Győzelmek: 12, Vereségek: 7, Döntetlenek: 3

Minden sorozat első eleme kulcsként, a második pedig értékként szerepel.

Szótár létrehozása

A legegyszerűbb szótár egy üres szótár, amely nem tartalmaz sem kulcsot, sem értéket, és a {} operátorral hozható létre:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

Ha szótárt szeretnénk létrehozni értékekkel, akkor a {} zárójelek között kell elhelyezni a „kulcs: érték” párokat, vesszővel elválasztva:

```
>>> mydog = {'name': 'Australian Shepherd', 'training': '5',
'intelligence': 5, 'type': 'Companion'}
>>> mydog
{'intelligence': 5, 'training': '5', 'name': 'Australian
Shepherd', 'type': 'Companion'}
```

Típusátalakítás: dict() függvény

A dict() függvény használható a kételemű sorozatok szótárrá alakítására. Ha a dict() függvénynek egy couple objektumot adunk át (egy olyan listát, amely két elemű listákat tartalmaz), akkor az visszaad egy szótárat:

```
>>> couple = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(couple)
{'e': 'f', 'c': 'd', 'a': 'b'}
```

Ebben a példában szintén egy szótár kerül visszaadásra, amikor a dict() függvény egy tuple párokat tartalmazó listát kap:

```
>>> info = dict([('name', 'Alex'), ('age', 38)])
>>> info
{'name': 'Alex', 'age': 38}
```

Szótár elemek hozzáadása és módosítása

Az elemek hozzáadása a szótárhoz meglehetősen egyszerű. Ehhez egyszerűen hivatkoznunk kell az elem kulcsára, és értéket kell rendelni hozzá. Ha a kulcs már létezik a szótárban, a meglévő érték újra cserélődik az új értékre. Ha a kulcs új, akkor a kulcs és az érték is hozzáadódik a szótárhoz. Létrehozhatunk egy szótárat, amely népszerű programozási nyelvek fejlesztőinek neveit tartalmazza, a nyelvek neveit kulcsként, a fejlesztők nevét pedig értékként:

```
>>> languages_programmers = {
```

```
... 'JavaSript': 'Brendan Eich',
... 'Python': 'Guido van Rossum',
... 'Ruby': 'Yukihiro Matsumoto',
... 'PHP': 'Rasmus Lerdorf',
... 'Scala': 'Martin Odersky'
... }
>>> languages_programmers
{'Python': 'Guido van Rossum', 'Scala': 'Martin Odersky',
'Ruby': 'Yukihiro Matsumoto', 'PHP': 'Rasmus Lerdorf',
'JavaSript': 'Brendan Eich'}
```

Adjunk hozzá egy másik jelentős számítástechnikai tudóst (a C nyelv fejlesztője) a szótárhoz:

```
>>> languages_programmers['C'] = 'Dennis MacAlistair Ritchie'
>>> languages_programmers
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'C': 'Dennis MacAlistair Ritchie'}
```

A legutóbb hozzáadott fejlesztő neve túl hosszú (három szóból áll). Javítsuk ki:

```
>>> languages_programmers['C'] = 'Dennis Ritchie'
>>> languages_programmers
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'C': 'Dennis Ritchie'}
```

Ugyanazzal a kulccsal ('C') kicseréltük az eredeti értéket 'Dennis MacAlistair Ritchie' -ről 'Dennis Ritchie' -re. A kulcsoknak egyedinek kell lenniük a szótárban. A szótár létrehozása előtt érdemes végiggondolni, hogy mely értékek szolgálhatnak kulcsokként. Ha ugyanazt a kulcsot többször is alkalmazza, csak az utolsó érték számít:

```
>>> languages_programmers = {
... 'JavaSript': 'Brendan Eich',
... 'Python': 'Guido van Rossum',
... 'C': 'Dennis MacAlistair Ritchie',
... 'Ruby': 'Yukihiro Matsumoto',
```

```
... 'PHP': 'Rasmus Lerdorf',
... 'Scala': 'Martin Odersky',
... 'C': 'Dennis Ritchie'
... }
>>> languages_programmers
{'PHP': 'Rasmus Lerdorf', 'Scala': 'Martin Odersky',
'JavaSript': 'Brendan Eich', 'Python': 'Guido van Rossum', 'C':
'Dennis Ritchie', 'Ruby': 'Yukihiro Matsumoto'}
```

Szótárak egyesítése

Az **update() függvény** használatával egy szótár kulcsait és értékeit átmásolhatjuk egy másik szótárba. Definiáljunk egy „languages_programmers” nevű szótárat, amely ismert programozási nyelvek fejlesztőinek nevét tartalmazza:

```
>>> languages_programmers = {
... 'JavaSript': 'Brendan Eich',
... 'Python': 'Guido van Rossum',
... 'Ruby': 'Yukihiro Matsumoto',
... 'PHP': 'Rasmus Lerdorf',
... 'Scala': 'Martin Odersky',
... 'C': 'Dennis Ritchie'
... }
>>> languages_programmers
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'C': 'Dennis Ritchie'}
```

Van egy másik szótárunk is, az „others”, amely más fejlesztők neveit tartalmazza:

```
>>> others = {'C++': 'Bjarne Stroustrup', 'Swift': 'Chris
Lattner'}
```

Adjuk hozzá az „others” szótár fejlesztőit az alap „languages_programmers” szótárhoz:

```
>>> languages_programmers.update(others)
>>> languages_programmers
```

```
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'Swift': 'Chris Lattner', 'C': 'Dennis
Ritchie', 'C++': 'Bjarne Stroustrup'}
```

Mi történik, ha a második szótár ugyanazokat a kulcsokat tartalmazza, mint az első? A második szótárban található értékek „győznek”:

```
>>> first = {'a': 10, 'b': 20}
>>> second = {'b': 'other'}
>>> first.update(second)
>>> first
{'a': 10, 'b': 'other'}
```

Elemek törlése a szótárból

Töröljük az előző két fejlesztőt a szótárból a kulcsok segítségével:

```
>>> del  languages_programmers['Swift']
>>> languages_programmers
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'C': 'Dennis Ritchie', 'C++': 'Bjarne
Stroustrup'}
>>> del  languages_programmers['C++']
>>> languages_programmers
{'Scala': 'Martin Odersky', 'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'PHP': 'Rasmus Lerdorf', 'Ruby':
'Yukihiro Matsumoto', 'C': 'Dennis Ritchie'}
```

Ha az összes kulcsot és értéket el szeretnénk távolítani a szótárból, használhatjuk a **clear() függvényt**, vagy egyszerűen hozzárendelhetjük a változóhoz az üres szótárat. Az első módszer a szótár teljes törlésére:

```
>>> languages_programmers.clear()
>>> languages_programmers
{}
```

A második módszer a szótár teljes törlésére:

```
>>> languages_programmers = {}
>>> languages_programmers
{}
```

Ellenprzés: van-e a szótárban egy konkrét kulcs-érték?

Ha szeretnénk tudni, hogy van-e valamilyen kulcs a szótárban, használhatjuk az „in” kulcsszót. Ismét definiáljuk a „languages_programmers” szótárat, de most kevesebb elemmel:

```
>>> languages_programmers = {'JavaSript': 'Brendan Eich',
'Python': 'Guido van Rossum', 'Ruby': 'Yukihiro Matsumoto'}
>>> 'Ruby' in languages_programmers
True
>>> 'Scala' in languages_programmers
False
```

Értékek lekérése a szótárból

A szótárból egy adott érték lekéréséhez egyszerűen meg kell adni a szótárat és a hozzá tartozó kulcsot:

```
>>> languages_programmers['Ruby']
'Yukihiro Matsumoto'
```

Ha a kulcs nincs a szótárban, hiba lép fel:

```
>>> languages_programmers['PHP']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'PHP'
```

A hibák elkerülése érdekében a szótárral végzett munka során először ellenőrizni kell, hogy létezik-e a kulcs a szótárban, vagy használhatjuk a **get() függvényt**. Megadjuk a szótárat, a kulcsot és egy opcionális értéket. Ha a kulcs létezik, akkor megkapjuk a hozzá tartozó értéket:

```
>>> languages_programmers.get('Ruby')
'Yukihiro Matsumoto'
```

Ha a kulcs nem létezik, akkor az opcionálisan megadott érték kerül visszaadásra:

```
>>> languages_programmers.get('PHP', 'PHP not in dictionary.')
'PHP not in dictionary.'
```

Ha nincs megadott érték, akkor a None objektum kerül visszaadásra (az interaktív interpreter nem fog semmit kiírni):

```
>>> languages_programmers.get('PHP')
>>>
```

Szótár kulcsai és értékei

Ha szeretnénk az összes kulcsot lekérni a szótárból, használhatjuk a **keys() függvényt**. A következő példákhoz másik szótárat használunk:

```
>>> light_signals = {'green': 'go', 'yellow': 'get ready',
'red': 'stop'}
>>> light_signals.keys()
dict_keys(['yellow', 'red', 'green'])
```

A Python 3-ban a keys() függvény nem listát, hanem dict_keys()-t ad vissza, amely a kulcsok iterálható reprezentációja. Az iterálható kulcsok kényelmesek nagy szótárakhoz, mivel nem igényelnek időt és memóriát a kulcsok listájának létrehozására és tárolására. De gyakran lista formátumra van szükségünk. Ezért Python 3-ban a list() függvényt kell használni a dict_keys átalakítására listává. Az összes kulcs lekérése listaként a list() függvénnyel:

```
>>> list(light_signals.keys())
['yellow', 'red', 'green']
```

A Python 3-ban a list() függvény használható a values() és items() függvények eredményének sima listává alakítására. Az összes érték lekérése a szótárból a **values() függvénnyel:**

```
>>> list(light_signals.values())
['get ready', 'stop', 'go']
```

Az összes „kulcs: érték” pár lekérése a szótárból az **items() függvénnyel:**

```
>>> list(light_signals.items())
[('yellow', 'get ready'), ('red', 'stop'), ('green', 'go')]
```

Amint az eredményből látható, minden „kulcs: érték” pár tuple-ként tér vissza.

setdefault() függvény - hiányzó kulcsok kezelése a szótárban

A szótár egy olyan eleméhez való hozzáférés megkísérlése, amely nem létezik, hibát eredményez. A setdefault() függvény létrehozza az adott kulccsal rendelkező elemet a szótárban, ha az még nem létezik.

```
periodic_table = {'Hydrogen': 1, 'Helium': 2}
print(periodic_table)
carbon = periodic_table.setdefault('Ferrum', 26)
print(periodic_table)
print(carbon)
```

A setdefault() függvény használata: ha a kulcs még nem szerepel a szótárban (például 'Ferrum'), akkor új kulcsot és értéket adunk hozzá (26). Az új „kulcs: érték” pár hozzáadásának eredménye: A 'Ferrum' kulcshoz hozzáadott érték, 26: Az eredmény így fog kinézni:

```
{'Hydrogen': 1, 'Helium': 2}
{'Hydrogen': 1, 'Helium': 2, 'Ferrum': 26}
26
```

Szótárak metódusai

Metódos	Alkalmazása
`dict.clear()`	szótár dict törlése
`dict.copy()`	Visszaadja a dict szótár másolatát
`dict.fromkeys(seq, value=None)`	Létrehoz egy dict szótárat a kulcsokból a szekvenciában seq és az alapértelmezett értékkel value (alapértelmezett: None)
`dict.get(key, default=None)`	Visszaadja egy kulcs értékét a dict szótárból, de ha a kulcs nem található, nem dob kivételt, hanem visszaadja a

Metódos	Alkalmazása
	default értéket (alapértelmezett: None)
`dict.items()`	Visszaadja a kulcs-érték párokat a dict szótárból egy listában [(key1, value1), (key2, value2), ...]
`dict.keys()`	Visszaadja a dict szótár kulcsait
`dict.pop(key, default)`	Eltávolítja a kulcsot és visszaadja annak értékét a dict szótárból; ha a kulcs nem található, visszaadja a default értéket (ha nincs default, akkor KeyError kivételt dob)
`dict.popitem()`	Eltávolítja és visszaadja a kulcs-érték párt a dict szótárból; ha a szótár üres, KeyError kivételt dob (ne feledd, hogy a szótárak nem rendezettek)
`dict.setdefault(key, default=None)`	Visszaadja a dict szótár kulcsának értékét, de ha az nem található, nem dob kivételt, hanem létrehoz egy kulcsot a default értékkel (alapértelmezett: None)
`dict.update(other)`	Frissíti a dict szótárat, hozzáadva kulcs-érték párokat az other szótárból; a meglévő kulcsok felülíródnak (visszaadja a None-t)
`dict.values()`	Visszaadja az értékeket a dict szótárból egy listában [value1, value2, ...]

Megjegyzés

A szótár kulcsa kel, hogy változtathatatlan (*immutable*) legyen. Leggyakrabban kulcsként sztringeket alkalmaznak, mivel azok változtathatatlanok. Előnyei:

- **Biztonság és stabilitás:** Ha sztringek változtathatóak lennének, akkor például ha egy szótár (*dict*) kulcsa lenne

egy sztring, és valaki megváltoztatná a sztring tartalmát, az tönkretenné a szótár működését.

- **Teljesítmény:** A változtathatatlan objektumokat könnyebb optimalizálni. Például a Python belsőleg újra tudja használni ugyanazokat a sztring objektumokat, ha azok megegyeznek.
- **Hash-elhetőség:** Csak változtathatatlan objektumok használhatók kulcsként a szótárakban és elemekként a set-ekben, mert ezeknek az objektumoknak állandó hash-értékük van. A sztringek ilyenek.

Néhány példa arra, hogy milyen előnyökkel jár a sztringek változtathatatlansága Pythonban:

1. Szótár kulcsokként való használat

A dict kulcsainak változtathatatlannak kell lenniük, különben a hash értékük megváltozna, és nem lehetne megbízhatóan keresni.

```
my_dict = {
    "apple": 1,
    "banana": 2
}
print(my_dict["apple"])
```

Ha az "apple" sztringet valaki menet közben megváltoztatná, akkor a szótár nem találná meg a kulcsot. Ezért kizárólag változtathatatlan típusok (mint str, int, tuple) lehetnek a kulcsok.

A hash értéket arra használják, hogy gyorsan megtaláljanak, összehasonlítsanak vagy rendszerezzenek adatokat. Egyszerűen fogalmazva, a hash egy szám, ami „leírja” az objektum tartalmát.

Ha két objektum ugyanaz, akkor a hash-ük is ugyanaz lesz.
Ha különbözőek, akkor a hash-ük valószínűleg különböző.

A hash() függvényről

A hash() egy beépített függvény, amely egy egész számot (int) ad vissza egy objektum alapján. Ez a szám reprezentálja az objektum tartalmát egy gyors összehasonlítás vagy keresés céljából. Ez a szám Pythonon belül használható például szótáraknál, hogy gyorsan el tudja dönteni, hol tárolja az adatot.

Példa

```
print(hash("alma"))
```

```
# pl.: 6180170782534844781
print(hash("körte"))
# pl.: -3661300747083310579
print(hash((1, 2, 3)))   # pl.: 529344067295497451
# működik, mert tuple immutable
print(hash(12345))       # pl.: 12345
```

Nem minden hash ugyanaz minden futásnál. Python 3.3 óta a stringek és más típusok hash értékei véletlenszerűen változnak minden programfutásnál, ami biztonsági okból van.

Ha determinista (állandó) hash-re van szükségünk (pl. fájlazonosító vagy adatbázis kulcs), akkor inkább használjnk **hashlib modult** (pl. SHA-256).

Mi történik nem hash-elhető objektummal?

```
print(hash([1, 2, 3]))
 # TypeError: unhashable type: 'list'
```

Mert a list változtatható → nem használható hash kulcsként.

Bövebben a hash alkalmazásáról a **Mellékletben.**

3.4. Halmazok

A halmaz hasonló a szótárhoz, de nincs benne érték, csak kulcsok. A szótár kulcsaival ellentétben a halmaz elemeinek változhatónak és hashelhetőnek kell lenniük. Ez azt jelenti, hogy az egész számok, lebegőpontos számok, karakterláncok és tuple-ok lehetnek a halmaz elemei, de a listák, szótárak és más halmazok nem.

A halmaz egy rendezetlen objektumok gyűjteménye, amelyben nem lehetnek duplikált elemek.

Mint a tuple, a halmaz is létrehozható listából vagy más, iterálható szekvenciákból. Azonban a listákkal és tuple-ökkel ellentétben a halmazok esetében nem számít a sorrend.
A halmazok gyakran két célra használjá:

- a duplikált elemek eltávolítására és
- a tagság ellenőrzésére.

A halmaz létrehozásához használhatjuk a **set() függvényt** vagy egy vagy több értéket a kapcsos zárójelek között, vesszővel elválasztva.

Például:

```
>>> empty_set = set()

>>> empty_set

set()

>>> numbers = {0, 1, 4, 5, 7, 9}

>>> numbers

{0, 1, 4, 5, 7, 9}
```

Ви можете створити множину за допомогою **функції set()** зі списку, рядка, кортежу або словника, втративши всі значення, що повторюються. Для прикладу, поглянемо на рядок, який містить більш ніж одне входження деяких букв:

Egy halmazt létrehozhatunk a set() függvény segítségével listából, karakterláncból, tuple-ból vagy szótárból, elveszítve minden ismétlődő értéket. Például nézzük meg a karakterláncot, amely több előfordulást is tartalmaz néhány betűből:

```
>>> set('office')

{'e', 'c', 'o', 'f', 'i'}
```

A halmaz csak egy előfordulást tartalmaz az 'f' betűből, annak ellenére, hogy az 'office' szóban két előfordulása van ennek a betűnek.

A halmazhoz való tartozás ellenőrzésére az 'in' operátort használhatjuk:

```
>>> numbers = {0, 1, 4, 5, 7, 9}

>>> 5 in numbers

True

>>> 30 in numbers

False
```

A Python halmazok támogatják a klasszikus halmazműveleteket, mint az

unió (|),

metszet (&),

különbség (-) és

kizáró vagy (^).

Végezzünk néhány műveletet a halmazokon:

```
>>> x = {0, 1, 4, 5, 7, 9}
>>> y = {1, 2, 3, 6, 7, 8, 9}
>>> x
{0, 1, 4, 5, 7, 9}
>>> y
{1, 2, 3, 6, 7, 8, 9}
```

Egy új elem (100) hozzáadása a halmazhoz az **add függvény** segítségével.

```
>>> x.add(100)
>>> x
{0, 1, 4, 5, 100, 7, 9}
```

Eltávolítjuk a 100 értéket a halmazból a **remove függvénnyel**.

```
>>> x.remove(100)
>>> x
{0, 1, 4, 5, 7, 9}
```

A **| operátor** két halmaz egyesítését végzi el, és új halmazt ad vissza, amely az egyedi értékeket tartalmazza.

```
>>> x | y
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

A **& operátor** egy új halmazt ad vissza, amely azokat az értékeket tartalmazza, amelyek mindkét halmazban megtalálhatók.

```
>>> x & y
{1, 9, 7}
```

A **- operátor** egy új halmazt ad vissza, amely azokat az elemeket tartalmazza, amelyek csak az első halmazban (x) találhatók.

```
>>> x - y
{0, 4, 5}
>>> y - x
{8, 2, 3, 6}
```

A ^ **operátor** egy új halmazt ad vissza, amely tartalmazza mindkét halmaz elemeit, kivéve azokat, amelyek közösek.

```
>>> x ^ y

{0, 2, 3, 4, 5, 6, 8}
```

Frozenset halmaz

Mivel a halmazok változtathatóak és nem hashelhetőek, nem lehetnek más halmazok elemei.
A probléma megoldására a Python-ban létezik egy másik típusú halmaz, a **frozenset**, amely a halmazhoz hasonlóan működik, de *nem változtatható meg a létrehozása után.*

```
>>> off = set([0, 1, 2, 3, 4])

>>> on = frozenset(off)

>>> on

frozenset({0, 1, 2, 3, 4})

>>> on.add(50)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'frozenset' object has no attribute 'add'

>>> off.add(on)

>>> off

{0, 1, 2, 3, 4, frozenset({0, 1, 2, 3, 4})}
```

3.5. Összetett adatstruktúrák

Különböző adat típusokat (logikai értékek, számok és karakterláncok, listák, tuple-ok, halmazok és szótárak) kombinálhatunk saját struktúrákba, amelyek nagyobbak és bonyolultabbak. Például vegyünk három különböző listát:

```
>>> dairy_products = ['butter', 'cheese', 'yoghurt']
```

```
>>> vegetables = ['potatoes', 'broccoli', 'carrot', 'cucumber',
'onion']
>>> light_snacks = ['biscuits', 'chocolate', 'nuts']
```

Létrehozhatunk egy tuple-t, amely tartalmazza ezen listák mindegyikét:

```
>>> tuple_of_lists = dairy_products, vegetables, light_snacks
>>> tuple_of_lists
(['butter', 'cheese', 'yoghurt'], ['potatoes', 'broccoli',
'carrot', 'cucumber', 'onion'], ['biscuits', 'chocolate',
'nuts'])
```

Ezen kívül létrehozhatunk egy listát, amely tartalmaz három listát:

```
>>> list_of_lists = [dairy_products, vegetables, light_snacks]
>>> list_of_lists
[['butter', 'cheese', 'yoghurt'], ['potatoes', 'broccoli',
'carrot', 'cucumber', 'onion'], ['biscuits', 'chocolate',
'nuts']]
```

Végül, létrehozhatunk egy szótárt listákkal. Ebben a példában a termékcsoportok neveit használjuk kulcsként, a termékek listáját pedig értékként:

```
>>> dict_of_lists = {'Dairy': dairy_products, 'Vegetables':
vegetables, 'Snacks': light_snacks}
>>> dict_of_lists
{'Snacks': ['biscuits', 'chocolate', 'nuts'], 'Vegetables':
['potatoes', 'broccoli', 'carrot', 'cucumber', 'onion'],
'Dairy': ['butter', 'cheese', 'yoghurt']}
```

A szótár kulcsainak változhatatlannak kell lenniük, ezért a lista, szótár vagy halmaz nem lehet kulcs egy másik szótárban. De a tuple kulcs lehet.
Például létrehozhatunk egy szótárt a különböző sportágak kifejezéseivel, ahol a kulcsok tuple-ok:

```
>>> sports_equipment = {
... ('1500 metres', 'high jump', 'shot put'): 'Athletics',
... ('goalkeeper', 'foul', 'corner'): 'Football',
... ('handlebars', 'wheels', 'bicycle pump'): 'Cycling'
... }
```

```
>>> sports_equipment
{('1500 metres', 'high jump', 'shot put'): 'Athletics',
('handlebars', 'wheels', 'bicycle pump'): 'Cycling',
('goalkeeper', 'foul', 'corner'): 'Football'}
```

3.5. Hash akalmazásai

1. Szótárak kulcsaihoz:
    - Amikor írjuk, hogy my_dict["kulcs"], akkor a Python először megnézi a "kulcs" hash értékét.
    - Ezzel gyorsan megtudja találni, hová van tárolva az adat.
2. set típushoz:
    - A set egyedi elemeket tárol. Ehhez hash-eket használ, hogy tudja, szerepel-e már egy elem.
3. Gyors összehasonlítás:
    - Ha két objektum hash-e nem egyezik, akkor biztosan nem egyenlők. Ez gyorsabb, mint teljesen végignézni az összes karaktert egy sztringben vagy az összes elemet egy tuple-ben.

Példa set alkalmazásra

```
s = set()
s.add("python")          # str hash-elhető
s.add((1, 2, 3))         # tuple is
print("python" in s)     # True
```

- Csak változtathatatlan (immutable) objektumok hash-elhetők:
    - str, int, float, tuple (ha minden eleme is immutable)
    - Nem hashelhető például: list, dict, set

Pl. print(hash([1, 2, 3])) # TypeError!

Miért fontos ez? Ha a Python eltesz egy kulcsot a szótárba, aztán valaki megváltoztatja azt a kulcsot (pl. egy listát), akkor a hash érték már nem lesz érvényes, és a szótár nem tudja többé megtalálni az adatot, vagy rossz helyre nyúl.

Tuple kulcsként csak akkor használtunk, ha belül is immutable.

```
key = (1, 2, 3)
my_dict = {key: "rendben"}
```

```
print(my_dict[(1, 2, 3)])
 # "rendben"
```

Ez működik, mert a tuple és az elemei (int) mind immutable.

De ha tuple-ben van lista:

```
bad_key = ([1, 2], 3)
my_dict = {bad_key: "hiba"}
Eredmény:
TypeError: unhashable type: 'list'
```

A tuple ugyan immutable, de tartalmaz egy listát, ami nem az, így az egész kulcs nem hash-elhető.

Hogyan lehet változtatható típusokat „átalakítani” változtathatatlanná? Például, listából kulcsot csinálni. Mivel a lista nem használható kulcsként (unhashable), a megoldás:

alakítsuk át tuple-re – mert a tuple immutable és hash-elhető

Példa.

```
my_list = [1, 2, 3]
# átalakítás tuple-re
my_key = tuple(my_list)unk
my_dict = {my_key: "siker!"}
print(my_dict[(1, 2, 3)])  # siker!
```

Így már működik! A tuple(my_list) egy változtathatatlan másolat, amit biztonságosan használhatunk kulcsként.

Fontos: csak akkor működik, ha a lista elemei is hash-elhetők!

```
bad_list = [[1, 2], 3]
bad_key = tuple(bad_list)
my_dict = {bad_key: "hiba"}
 # TypeError, mert tuple-ben van lista!
```

Ilyenkor mélyebb átalakításra van szükség: például minden belső listát is tuple-lé kell alakítani.

Még egy hasznos trükk: **rekurzív „tupelizálás”**

Ha van egy több szintű, beágyazott lista, ezt egy egyszerű függvénnyel tuple-lé alakíthatod:

```
def make_hashable(obj):
    if isinstance(obj, list):
```

```
        return tuple(make_hashable(x) for x in obj)
    elif isinstance(obj, dict):
        return tuple(sorted((k, make_hashable(v)) for k, v in
obj.items()))
    else:
        return obj
```

make_hashable függvény alkalmazása:

```
nested_list = [[1, 2], [3, 4]]
key = make_hashable(nested_list)
my_dict = {key: "komplex, de működik!"}
print(my_dict[make_hashable([[1, 2], [3, 4]])])  # "komplex, de
működik!"
```

Ha list-et akarunk kulcsként alkalmazni, akkor:

1. Használjunk tuple(my_list);
2. Ha van benne beágyazott változtatható elem, alakítsuk át azt is;
3. Használjunk make_hashable()-hez hasonló függvényt.

Teljesítmény-optimalizálás (interning)

Python optimalizálásként az ugyanolyan sztringeket nem hozza létre újra, hanem ugyanarra az objektumra mutat:

```
a = "hello"
b = "hello"
print(a is b)  # True, ugyanaz az objektum!
```

Ez csak azért működik, mert a sztring sosem fog megváltozni, így biztonságos újrahasználni.

Szálbiztonság (thread safety)

Mivel a sztring nem változhat meg, több szál is olvashatja egyszerre anélkül, hogy ütközés vagy adatvesztés történne.

Ez különösen fontos nagy rendszerekben, ahol egyszerre több szál dolgozik ugyanazzal az adattal.

Összefoglalva:

A változtathatatlanság:

- Biztonságos (különösen szótárak és set-ek esetén)
- Hatékony (memóriaoptimalizálás, gyorsabb összehasonlítás)
- Szálbiztos
- Egyszerűsíti a kódot, mert nem kell attól tartani, hogy valahol valaki megváltoztatja a sztringet.

1. Szótár kulcsként való használat

Ahogy a sztring, úgy a tuple is lehet dict kulcs, ha minden eleme maga is változtathatatlan (pl. int, str, más tuple, stb.)

```
location = {(47.4979, 19.0402): "Budapest"}
print(location[(47.4979, 19.0402)])
  # Budapest
```

3. Védelem véletlen módosítás ellen

Példa: ha egy függvénynek átadunk egy list-át, és az módosítja azt, az hatással lesz a hívó kódra is. Tuple-nél ilyen nem fordulhat elő:

```
def change_list(lst):
    lst.append(100)
a = [1, 2, 3]
change_list(a)
print(a)  # [1, 2, 3, 100] — megváltozott!

def try_change_tuple(tpl):
    tpl += (100,)

b = (1, 2, 3)
try_change_tuple(b)
print(b)  # (1, 2, 3) — érintetlen maradt
```

Ezért tuple-t gyakran használnak olyan helyeken, ahol állandó adatok kellenek, vagy védeni akarjuk az adatokat a véletlen módosítástól.

Az id() függvény

Ezzel meg tudjuk nézni egy objektum memóriacímét (vagyis azonosítóját), így könnyen eldönthető, hogy új objektum jött-e létre vagy csak egy meglévő változott meg (ha lehetséges).

Példa.

```
# str- új objektum jön létre minden "változtatásnál"
s = "hello"
print(id(s))
  # mondjuk: 140573748016816
s = s + " world"
print(id(s))
  # teljesen más: 140573748015728
```

A két id különböző: tehát új objektum jött létre! Az eredeti "hello" nem változott meg, csak s mutat most egy másik helyre.

```
# Lista: módosítás ugyanazon az objektumon belül
l = [1, 2, 3]
print(id(l))
 # pl.: 140573752944448
l.append(4)
print(id(l))
 # ugyanaz!
```

A list megváltozik, de az id azonos marad, mert ugyanaz az objektum bővült.

```
# Tuple: nem lehet módosítani, csak új példány
t = (1, 2, 3)
print(id(t))
t = t + (4,)
print(id(t))
  # másik ID!
```

Nem tudjuk "módosítani" a tuple-t. A t + (4,) új objektumot hoz létre, ahogyan a sztringeknél is.

Mi történik, ha két ugyanolyan sztringet hozunk létre?

```
a = "python"
b = "python"
```

```
print(a is b)
        # True – gyakran ugyanarra az objektumra mutatnak
print(id(a), id(b))
 # azonos lehet
# De:
a = "py" + "thon"
print(a is "python")
 # Ez is lehet True – Python optimalizál!
```

Ez a string interning: a Python optimalizál, és nem mindig hoz létre új objektumot, ha ugyanazt a sztringet már egyszer használtuk. Ez csak immutable objektumoknál biztonságos.

IRODALOMJEGYZÉK

1. Васильєв О. М. Програмування мовою Python. Тернопіль: Навчальна книга Богдан, 2019. 504с
2. Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers, Hogyan gondolkozz úgy, mint egy informatikus: Tanulás Python 3 segítségével, 2019 https://mtmi.unideb.hu/pluginfile.php/554/mod_resource/content/1/thinkcspy3.pdf
3. *Костюченко А.О.* Основи програмування мовою Python: навчальний посібник. Ч.: ФОП Баликіна С.М. 2020. 180 с.
4. https://www.w3schools.com/python
5. https://www.malnasuli.hu/python/keszits-hazilag-grafikus-vezerlofeluletet-tkinter-1-resz/
6. http://nyelvek.inf.elte.hu/leirasok/Python/index.php?chapter=16
7. https://szit.hu/doku.php?id=oktatas:programozas:python:tkinter:hagyomanyosan
8. https://mek.oszk.hu/08400/08435/08435.pdf

Виробничо-практичне видання

«Сучасні мови програмування (Python. 1 частина)»

Методичні вказівки до лекційних та практичних занять

2025 р.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 9 від «28» травня 2025 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 6 від «7» липня 2025 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від «9» липня 2025 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

(90 с., угорською мовою)

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічих наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несуть розробники.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Статут «Закарпатського угорського інституту імені Ференца Ракоці ІІ» (Затверджено протоколом загальних зборів Благодійного фонду За ЗУІ, протокол №1 від 09.12.2019р., прийнято Загальними зборами ЗУІ ім. Ф.Ракоці ІІ, протокол №2 від 11.11.2019р., зареєстровано Центром надання адміністративних послуг Берегівської міської ради, 12.12.2019р.)*

Шрифт «Courier New». Розмір сторінок методичних вказівок: А4 (210х297мм)