ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ ІІ
II. RÁKÓCZI FERENC KÁRPÁTALJAI MAGYAR FŐISKOLA

Кафедра математики та інформатики
Matematika és Informatika Tanszék

ПРОГРАМУВАННЯ 1. ОСНОВИ МОВИ PYTHON/ PROGRAMOZÁS 1. PYTHON NYELV ALAPJAI

МЕТОДИЧНІ ВКАЗІВКИ ДО ПРАКТИЧНИХ ЗАНЯТЬ/ MÓDSZERTANI ÚTMUTATÓ GYAKORLATI FOGLALKOZÁSOKHOZ

Сучасні мови програмування/Modern programozási nyelvek
(назва навчальної дисципліни/a tantárgy neve)
Перший (бакалаврський) / Alapképzés (BSc)
(рівень вищої освіти / felsőoktatás szintje)
01 Освіта/Педагогіка / 01 Oktatás/Pedagógia
(галузь знань / képzési ág)
Середня освіта (Інформатика) / Középiskolai oktatás (Informatika)
(освітня програма / képzési program)

Берегове / Beregszász 2024 р.

Методичні вказівки «Програмування 1. Мова програмування Python» до практичних занять з дисципліни «Сучасні мови програмування» розроблені на основі Освітньої програми підготовки бакалаврів з галузі знань «01 Освіта/Педагогіка» за напрямом «014 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є поглиблення знань студентів з дисципліни « Сучасні мови програмування». У роботі наведені основні конструкції, функції, пакети мови програмування Python, наводяться приклади їх використання для типових простих задач. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу, а також студентами-математиками.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від «13 » серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 22 від 26 серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від 27 серпня 2024 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 „Programozás 1. Python programozási nyelv” módszertani útmutató a „Modern programozási nyelvek” tárgy gyakorlati foglalkozásaihoz a BSc „Középiskolai oktatás (Informatika)” képzési programja alapján lett kidolgozva nappali és levelező tagozatos hallgatók részére. A módszertani kézikönyv célja, hogy elmélyítse a „Modern programozási nyelvek” tárgy t hallgatóinak ismereteit. Az útmutató leírja a Python programozási nyelv főbb struktúráit, függvényeit, csomagjait, és példákat ad ezek használatára jellemző egyszerű feladatokhoz. A módszertani utasítások a gyakorlati órákon és a tantárgy önálló munkája során használhatók, valamint a matematikus hallgatók is használhatják.

Az oktatási folyamatban történő felhasználását jóváhagyta a
II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszéke
(2024.08.13., 1 számú jegyzőkönyv).

Megjelentetésre javasolta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Oktatási és Módszertani Tanácsa
(2024. augusztus 26., 22. számú jegyzőkönyv).

Elektronikus formában (PDF fájlformátumban) történő kiadásra javasolta
a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Tudományos Tanácsa
(2024. augusztus 27., 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

Ez a jegyzet elsősorban informatika szakos hallgatók számára készült, de hasznos lehet a harmadéves matematikus hallgatók számára is. Jelenleg a Python legelterjedtebb programozási nyelv. A középiskola is gyakorlatilag már áttért a Python nyelv oktatására. Ezért fontos, hogy a informatikus és matematikus hallgatók jól elsajátítják ez a nyelvet.

Python egy magas szintű, interpretált, általános célú programozási nyelv, amely számos jellemzővel rendelkezik, és népszerűsége folyamatosan növekszik a könnyű használhatósága, a széleskörű modulok és csomagok támogatása, valamint a széles alkalmazási területei miatt. A Python rendkívül sokoldalú és felhasználóbarát programozási nyelv, amely széles körben használható kezdve a kis projektektől a nagyobb, komplex rendszerekig. A nyelv folyamatos fejlődése és a közösség által támogatott bőséges kínálata teszi az egyik legnépszerűbb választássá a programozók körében.

Python fontosabb jellemzői

1. Egyszerű és Olvasható Szintaxis:

Python kifejező, tiszta szintaxissal rendelkezik, amely könnyen olvasható és megérthető. A behúzás használata is alapvető része a blokkok elkülönítésének.

2. Interpretált és Interaktív:

Python egy interpretált nyelv, ami azt jelenti, hogy a forráskódját közvetlenül futtathatjuk anélkül, hogy először fordítanánk gépi kóddá. Ez lehetővé teszi gyors fejlesztést és prototípusok készítését. Interaktív shell (interpreter) elérhető, amely lehetővé teszi, hogy kifejezéseket és parancsokat közvetlenül futtassunk, és azonnal láthatjuk az eredményüket.

3. Dinamikus típusok és Automatikus Memóriakezelés:

Python dinamikus típusokat használ, így a változóknak nem kell előre deklarálva lenniük típusukkal, és típusuk dinamikusan változhat. Automatikus memóriakezelést használ, így a programozónak nem kell aggódnia a memóriakezelés körüli részletek miatt.

4. Széleskörű Modulok és Csomagok Támogatása:

Python rendelkezik egy nagy és aktív közösséggel, amely számos modult és csomagot kínál különböző feladatok megoldására (pl. adatfeldolgozás, webfejlesztés, gépi tanulás stb.). A PyPI (Python Package Index) a Python csomagok hivatalos tárolója, amely több tízezer csomagot tartalmaz, és könnyen telepíthetők a *pip* csomagkezelő segítségével.

5. Kereszteződés a Szkript és Alkalmazás Programozás Között:

A Python rendkívül sokoldalú és felhasználóbarát programozási nyelv, amely széles körben használható kezdve a kis projektektől a nagyobb, komplex rendszerekig. A nyelv folyamatos fejlődése és a közösség által támogatott bőséges kínálata teszi az egyik legnépszerűbb választássá a programozók körében.

Python alkalmas kisebb szkriptek írására, amelyek gyorsan megoldanak különböző feladatokat, és egyben használható nagyobb és összetettebb alkalmazások fejlesztésére is. Nagy vállalati rendszerek és webalkalmazások is építhetők Python segítségével, széleskörű támogatással és keretrendszerekkel (pl. Django, Flask).

6. Platformfüggetlenség és Nyitott Forráskód:

Python egy platformfüggetlen nyelv, ami azt jelenti, hogy a kód ugyanúgy futtatható Windows-on, macOS-en és Linux-on egyaránt. Python nyílt forráskódú nyelv, amely lehetővé teszi a közösségi fejlesztést és hozzájárulást, és a különböző platformokon könnyen kiterjeszthető és testre szabható.

Milyen sorrendben tanuljuk a Pythont

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

1. Alapok

• A Python telepítése és a fejlesztői környezet kiválasztása (például IDLE, Jupyter Notebook, Visual Studio Code, Thonny).
• **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
• (például Flask, Django, alkalmazása).
• HTTP parancsok kezelése.
10. Tudományos számítástechnika és adatelemzés
• Tudományos számítástechnikai könyvtárak, például NumPy, SciPy.
• Adatok kezelése a pandas könyvtár használatával.
12. Tesztelés és hibakeresés
• Az alapok tesztelése unittest vagy pytest segítségével.
• Hibakeresési programok a fejlesztői környezetben.
13. További témák:
• Reguláris kifejezések.
• Munka API-val.

A felsorolt javasolt témákból a módszertani útmutató csak az első 6 témával foglalkozik. A példák illusztratív jellegűek, bonyolultabb összetett feladatokkal nem foglalkozunk, mivel ezt a módszertani útmutató célja rövid, gyors bevezetés a Python nyelvbe. A példák egy részét a [1] forrásból vettük. Ne feledje, hogy a Python (vagy bármely más programozási nyelv) **tanulásának kulcsa a gyakorlat.** Próbáljon rendszeresen problémákat megoldani, és dolgozzon projekteken, hogy megszilárdítsa tudását. Az irodalomjegyzék tartalmazz forrásokat, amely hasznosak lehetnek abban, hogy a hallgatók jobban megismerjék a Python programozásának sajátosságát és szépségét.

Vissza

1. Python Development Environments (Python fejlesztési környezetek)

IDLE interpretator. Python ***REPL (read-eval-print-loop):*** read — olvassa a parancsokat, eval — végrehajtja, print — az eredményt kiírja, loop — ismétli ezt a ciklust.
A Python Development Environments (IDE) különféle funkciókat és eszközöket kínál a felhasználóbarát és produktív fejlesztéshez ezen a nyelven. Néhány a Python legnépszerűbb IDE-je.

1. **PyCharm.**
Ez a JetBrains által kifejlesztett egyik legnépszerűbb és legkiemelkedőbb IDE a Python számára. A PyCharm számos funkciót kínál, beleértve az intelligens kód kitöltését, a kód elemzését, a hibakeresést, a VCS integrációt és még sok minden mást.

2. **Visual Studio Code (VS Code).**
Ez egy könnyű és hatékony szövegszerkesztő, amelyet a Microsoft fejlesztett ki. Az olyan kiterjesztésekkel, mint a Python Extension Pack, a VS Code fejlett IDE-ként alakul át a Python számára olyan funkciókkal, mint az automatikus kiegészítés, a hibakeresés, a Jupyter Notebooks integrációja és még sok más.

3. **Spyder.**
Ez egy IDE, amelyet kifejezetten a Python használatával történő tudományos számítástechnikára terveztek. A Spyder egyszerű integrációt biztosít az adatelemzéshez használt könyvtárakkal, mint például a NumPy, a SciPy és a Matplotlib, és interaktív IPython konzolt kínál.

4. **JupyterLab / Jupyter Notebook.**
Ez egy interaktív környezet a kóddal való együttműködéshez, amelyet adatkutatáshoz, elemzéshez és gépi tanuláshoz terveztek. Ez lehetővé teszi notebookok létrehozását és végrehajtását kóddal, grafikával, szöveggel és egyéb tartalmi elemekkel.

5. **Atom.**
Ez egy kiterjeszthető szövegszerkesztő, amelyet a GitHub fejlesztett ki. Különböző pluginekkel, például ide-python, az Atom kényelmes fejlesztő eszközzé változtatható a Pythonon.

6. **Sublime Text**.
Gyors és rugalmas szövegszerkesztő, széles körű funkciókkal és beépülő modulokkal, beleértve a Python támogatást.

7. **PyDev (plugin az Eclipse-hez).**
Ez egy plugin az Eclipse fejlesztési környezetéhez, ... PyDev (plugin az Eclipse-hez): Ez egy plugin az Eclipse fejlesztési környezethez, amely IDE funkciókat biztosít a Python számára.

8. Thonny.
Thonny egy népszerű Python fejlesztőkörnyezet (IDE), amelyet kifejezetten kezdő programozók számára terveztek. Az IDE-t úgy tervezték, hogy könnyen használható legyen, és támogassa a tanulást, valamint segítsen a hibakeresésben. A Thonny segítségével könnyen írhat, szerkeszthet és futtathat Python kódokat, miközben számos hasznos funkciót és eszközt kínál a fejlesztőknek.
A Thonny-ban rendelkezésre áll egy egyszerűsített felhasználói felület, amely ideális azoknak, akik épp most tanulnak Python-t, és még nem rendelkeznek tapasztalattal más fejlesztő

környezettekkel. Emellett a Thonny beépített hibakeresővel rendelkezik, amely segít azonosítani és kijavítani a kódhibákat. A Thonny egy könnyen kezelhető, felhasználóbarát Python fejlesztőkörnyezet, amely ideális választás lehet kezdő programozók számára.

Ezen IDE-k mindegyikének megvannak a sajátosságai és előnyei.

Python IDE Online.

Van néhány olyan **online Python IDE**, amelyek segítségével Python-kódot írhat, szerkeszthet és futtathat a böngészőjében.

1. **Repl.it.**

A Repl.it egy teljesen online kódszerkesztő és futtató környezet, amely támogatja a Python-t és számos más programozási nyelvet is. Lehetővé teszi a kód megosztását másokkal és együttműködést más fejlesztőkkel.
https://www.programiz.com/python-programming/online-compiler/
repl.it

2. **PythonAnywhere.**

Ez egy felhőalapú Python IDE, amely lehetővé teszi a Python alkalmazások fejlesztését, szerkesztését és futtatását a böngészőben. Ingyenes verziója is elérhető, amely korlátozott funkciókat kínál.

3. **Trinket.**

Trinket egy online Python IDE, amelyet kifejezetten oktatási célokra terveztek. Lehetővé teszi a Python kód szerkesztését és futtatását, valamint interaktív kurzusok létrehozását és megosztását.
https://trinket.io/features/python3

4. **CodeSculptor.**

Ez egy online Python IDE, amelyet kifejezetten a CodeSkulptor nevű Python környezet megvalósítására terveztek. Lehetővé teszi a Python kód szerkesztését és futtatását a böngészőben.

5. **Ideone**.

Ideone egy online kódfuttató és megosztó platform, amely támogatja a Python-t és több más programozási nyelvet is. Lehetővé teszi a kód megosztását másokkal és futtatását a szerveroldalon.

6. **Online Python Tutor.**

Lehetővé teszi a felhasználók számára, hogy írjanak és futtassanak Python kódot a böngészőben, miközben lépésről lépésre követhetik a kód futását, megfigyelhetik a változók értékeit és a vezérlési szerkezetek működését. Ez különösen hasznos lehet kezdőknek vagy azoknak, akiknek nehézségeik vannak a programozási koncepciók megértésével.
Emellett az "Online Python Tutor" lehetővé teszi a kód megosztását másokkal, ami hasznos lehet tanárok és diákok számára, akik együtt dolgoznak vagy megosztják a tanult kódot.
https://www.online-python.com/

Minden platformnak megvannak a saját előnyei és korlátai. Ezért érdemes kipróbálni néhányat, hogy megtalálja a számára legmegfelelőbbet.

Vissza

2. Python szintaxis alapjai

2.1. Értékek és típusok

Az értékek – *számok, sztringek,* stb. – azon alapvető elemek közé tartoznak, amelyekkel a programok a működésük során dolgoznak. Az önmagukat definiáló értékeket *konstansoknak* vagy *literáloknak* nevezzük.

Az értékeket osztályokba, illetve adattípusokba, röviden típusokba sorolhatjuk.
A 25 egész szám, a "Hello, World!" – sztring, szöveges típusú. A sztring konstansok számunkra és az értelmező számára is könnyen felismerhetők arról, hogy i*dézőjelben* szerepelnek.
A **type** függvény segítségével megtudhatjuk az érték típusát. Az utasításokat a prompt (>>>) után kell begépelni.

```
>>> type("Helló, Világ!")
<class 'str'>
>>> type(17)
<class 'int'>
>>> type(3.2)
<class 'float'>
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

A sztringek az *str* osztályba,
az egész számok az *int* osztályba,
a tizedespontot tartalmazó számok a *float* osztályba tartoznak. Ezeket a gép *lebegőpontos alakban* tárolja.
A sztringek állhatnak *aposztrófok ('), idézőjelek ("), tripla aposztrófok (''') és tripla idézőjelek (""")* között is.

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

Az idézőjelek által közrefogott szöveg aposztrófot is tartalmazhat, az aposztrófok által határolt szöveg pedig tartalmazhat idézőjelet is ('A "Honvéd" győzött!'). A tripla aposztróffal és a tripla idézőjellel körülvett sztringeket háromszorosan idézőjelezett sztringeknek nevezik.
Az ilyen formában megadott sztringek tartalmazhatnak aposztrófot és idézőjelet is. A háromszorosan idézőjelezett szövegek több sort is felölelhetnek:

```
>>> text = """Ez a szöveg
több sorban
jelenik meg."""
>>> print(text)
Ez a szöveg
több sorban
jelenik meg.
>>>
<class 'str'>
```

A vessző használata. Vegyük például a 3,12-es értéket. Pythonban ez nem érvényes szám, ettől függetlenül használhatjuk a programban, csak más jelentéssel bír.

```
>>>3.2
3.12
>>> 3,12
(3, 12)
```

A Python értelmezése szerint egy értékpárt adtunk meg. A számok írásakor se vessző, se szóköz ne kerüljön a számjegyek közé. *Tizedesjelként pontot kell használnunk.* A formális nyelvekben szigorúak a szintaktikai szabályok, és még a legkisebb hiba is jelentős eltérést okozhat a kimenetben.

2.2. Változók

A programnyelvek egyik legfontosabb tulajdonsága, hogy képesek módosítani a változókat. *A változó az egy név, amely egy értékre utal.*
A változókhoz *értékadó kifejezés* segítségével rendelhetünk értéket.
Az értékadás művelet jele az egyenlőségjel (=).
Az egyenlőségvizsgálat művelet jele: két egyenlőség (==).

```
>>> nev = "Szabó"
>>> k = 27
>>> pi = 3.14159
```

Az értékadás során a műveleti jel jobb oldalán álló értéket a bal oldalán álló névhez rendeljük. Amikor az értelmezőt egy változó kiértékelésére utasítjuk, akkor azt az értéket határozza meg, amely aktuálisan a változóhoz tartozik. A változóhoz rendelt értékeket később felülírhatjuk, tehát új értéket rendelhetünk a változóhoz.

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

2.3. Változónevek, kulcsszavak

A változónevek tetszőleges hosszúságúak lehetnek, tartalmazhatnak betűket és számjegyeket is, de mindenképpen betűvel vagy aláhúzás karakterrel kell kezdődniük. A változónevek nagybetűket is tartalmazhatnak. Ha használjuk a nagybetűket, akkor ne feledjünk, hogy a kis és nagybetűs változók különbözőnek számítanak. Ezért a **X** és a **x** két különböző változó**!**

Az aláhúzás karaktert (_), ami szintén megjelenhet a nevekben, gyakran használjuk a több szóból álló nevek tagolására, például *sajat_nev*.
A hibás névválasztás szintaktikai hibát eredményez:

```
>>> 7nev = "név"
SyntaxError: invalid syntax
>>> tobb$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

A *7nev* név azért hibás, mert nem betűvel kezdődik. A *tobb$* pedig azért, mert a dollár jel nem érvényes betű (illegális karakter). A *class* a Python nyelvben **kulcsszó (foglalt szó).** A kulcsszavak határozzák meg a nyelv szintaktikai szabályait, struktúráit, ezért változónévként nem használhatóak.

A Pythonban használt kulcsszavak:

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

Programozáskór ezt a listát fegyelembe kell venni. A programozók általában olyan nevet választanak, mely az emberi olvasó számára utal a változó szerepére, hogy később könnyebb legyen felidézni azt. Az ilyen nevek a program dokumentálását is segítik.

2.4. Utasítások

Az utasítás olyan parancs, amelyet a Python értelmező képes végrehajtani.
Amikor begépelünk egy utasítást a parancssorba, a Python végrehajtja azt.

Kifejezések kiértékelése

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

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

Ebben a példában a *len* beépített Python függvény, mely egy sztring hosszát adja vissza. Már ismerjük a *print* és a *type* függvényeket. Ez már a harmadik példánk a függvényekre! A kifejezés kiértékelése egy értéket határoz meg, ezért alkalmazzuk a kifejezéseket az értékadó utasítások jobb oldalán (y = 3.14, x = len("helló"))

Műveleti jelek és operandusok

A műveleti jelek, más szóval *operátorok*, különböző műveleteket (pl.: összeadás, szorzás, osztás) jelölő speciális nyelvi elemek. Az *operandusok* pedig azok az értékek, melyeken a műveleteket elvégezzük. A +, -, * és a zárójelek Pythonban is a matematikában megszokott jelentéssel bírnak. A csillag (*) a szorzás, a két csillag (**) a hatványozás jele.

```
20+32
ora-1
ora*60+perc
perc/60
5**2
(5+9)*(15-7)
2 ** 3
3 ** 2
```

A műveletek elvégzése előtt az operandusok helyén álló változónevek az értékükkel helyettesítődnek. Az összeadás, a kivonás, a szorzás és a hatványozás is úgy működik, ahogy az várható. Lássuk az osztást. Váltsunk át 645 percet órákra:

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

Pythonban a / jellel végrehajtott osztás mindig lebegőpontos számot eredményez.

Legyen, hogy a teljes órák és a fennmaradó percek számát kell megtudni. A Python egy másik osztás operátort is biztosít számunkra, melynek jele a **//.** *Ezt a fajta osztást egész osztásnak nevezzük*, mert mindig egész értéket szolgáltat.
A 6 // 4 eredménye 1.
7 / 4
1.75
7 // 4
1
percek = 645
orak = percek // 60
orak
10

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

Műveletek precedenciája

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

1. A *zárójelnek* van a legnagyobb precedenciája. Használhatjuk a műveleti sorrend megváltoztatására, ugyanis a zárójelben álló kifejezések lesznek először kiértékelve. Például a

2 * (3-1) az 4,
(1+1)**(5-2) az 8.

Alkalmazásukkal javíthatjuk az kifejezések olvashatóságát is: (perc * 100) / 60.

2. *A hatványozás* a második legerősebb művelet. Pl., a 2**1+1 az 3.
3. A *szorzás és osztás* azonos erősségű művelet. Magasabb precedenciával bírnak, mint a szintén egyforma erősségű összeadás és kivonás.

A 2*3-1 tehát 5, a 5-2*2 pedig 1.
4. Az azonos erősségű operátorok kiértékelése *balról jobbra* haladva történik. Ez azt jelenti, hogy *balasszociatívak.*

Vegyük például a 6-3+2 kifejezést. A kiértékelés folyamatában a kivonást végezzük el először, ami 3-at eredményez, majd ehhez adunk 2-t, a végeredmény tehát 5.
• A *hatványozás* kivételt jelent: az nem balasszociatívit. Ha a ** művelet előkerül, akkor jobb kitenni a zárójeleket, hogy biztosan abban a sorrendben menjen végbe a kiértékelés, ahogy azt elterveztük.

```
>>> 2 ** 3 ** 2 # Eredmény 512
>>> (2 ** 3) ** 2 # Eredmény 64
```

2.5. Maradékos osztás

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

```
>>> x = 7 // 3 # Egész osztás
>>> x
2
>>>y = 7 % 3
>>> y
1
```

Tehát a 7-ben a 3 kétszer van meg, és 1 a maradék.
A maradékos osztás alapján ellenőrizhető, hogy *egy szám osztható-e a másikkal.* Ha **x % y** nullát ad eredményül, akkor az x osztható y-nal.

Meghatározhatjuk egy szám utolsó számjegyét vagy számjegyeit is:
x % 10 az x utolsó számjegyét,
x % 100 az utolsó két számjegyét adja vissza.

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

```
masodperc = int(input("Összesen hány másodperc? "))
orak = masodperc // 3600
megmaradt_masodpercek = masodperc % 3600
percek = megmaradt_masodpercek // 60
megmaradt_masodpercek_a_vegen = megmaradt_masodpercek % 60
print("Órák=", orak, " Percek=", percek,
 " Másodpercek=", megmaradt_masodpercek_a_vegen)
```

2.6. Típuskonverziós függvények

- ***int***
- ***float***
- ***str***

Ezek a függvények a nekik átadott paramétereket int, float és str típusúvá alakítják át. Ezeket a függvényeket *típuskonverziós függvénynek* nevezzük.

Az **int** függvény valós számot, vagy sztringet vár bemeneti paraméterként, és egész értékké alakítja át. Ha tizedesjegyeket tartalmaz a konvertálandó érték, akkor a függvény a *konvertálás során elhagyja azokat*.

```
>>> int(3.14)  #3
>>> int(3.9999) # 3
>>> int(3.0) #3
>>> int(-3.999) # -3
>>> int(percek / 60) # 10
>>> int("2345") # 2345
>>> int(17) # 17
>>> int("12abc")
Traceback (most recent call last):
File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12abc'
```

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

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

A **str** függvény a paramétereit karakterlánccá alakítja át:

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

.7. Adatbekérés

A Pythonban az *input()* beépített függvény segítségével kérhetünk adatokat a felhasználóktól:

```
név = input("Kérem, adja meg a nevét: ")
```

Amennyiben *interaktív módban* adjuk ki az utasítást a megjelenő prompt jel után lehet beírni a választ. Ha *szkriptként futtatjuk*, akkor egy ablak nyílik meg. A felhasználó ebbe az ablakba gépelheti be a nevét.

Az *input* függvény *az Enter leütését követően* visszaadja a felhasználó által megadott szöveget, amelyet a *név* változóhoz rendelünk.

Ha a felhasználó egy számot kérne be, akkor is sztringet kapna vissza. A programozó feladata átalakítani azt egész (vagy valós) számmá az *int* vagy *float* típuskonverziós függvények valamelyikével.

```
suly = float(input(sulya:’)
```

2.8. print() függvényről bővebben

A print() függvény számos opcióval rendelkezik. Néhány fontosabb opció:

- **end** Megadja a kimenet végéhez hozzáadandó karaktert. Az alapértelmezett új sor karakter \n.

```
print("Hello", end=" ")
print ("világ")
  Helló Világ
```

- **sep** Meghatározza az elválasztót a print()-nek átadott elemek között. Az alapértelmezett szóköz " ".

```
print("alma", "banán", "narancs", sep=", ") # alma, banán, narancs
```

- **fájl** Megadja azt a fájlleíró objektumot, amelyre a kimenet át lesz irányítva. Alapértelmezés szerint ez a szabványos kimenet (sys.stdout), monitor.

```
with open("output.txt", "w") as f:
     print("abcdef", file=f)
```

Ez a kód a print() kimenetét az output.txt fájlba irányítja.

A Pythonban nincs közvetlen lehetőség a nyomtatandó számjegyek beállítására a print() függvény használatakor. A kinyomtatás előtt azonban formázhatja a számokat a karakterlánc formázási módszerrel vagy a **format()** függvény segítségével, amellyel megadhatja a nyomtatandó számjegyek számát.

Néhány módja ennek:

1. *f-strings metódus* a karakterláncok formázására használhatjuk.

Legyen x = 3.141592653589793238
Akkor a
print(f"{x:.2f}") # 3.14 Két tizedesjegyű számot nyomtat a vessző után.
2. A karakterláncok formázási módja *formátum specifikátorral:*
x = 3.141592653589793238
print("{:.2f}".format(x)) # Ugyanaz: 3.14
3.A *round() függvény* használata:
x = 3.141592653589793238
print(round(x,2))

A *f előtaggal ellátott karakterlánc-literál*, például az f"{kifejezés}", egy f-karakterlánc a Pythonban. Ez egy speciális karakterlánc-literál, amely lehetővé teszi a Python-kifejezések értékeinek közvetlenül egy karakterláncba ágyazását anélkül, hogy meg kellene hívnia a karakterlánc formázási módszereket vagy használnia kellene a format() függvényt. A Python-kifejezéseket beágyazhatja egy f-karakterláncba, úgy hogy kapcsos zárójelek közé helyezi őket {}. A karakterlánc kiértékelésekor a kapcsos zárójelben lévő kifejezések lecserélődnek az értékükre.

```
name = "Imre"
age = 19
message = f"Az én nevem {name} és {age} éves vagyok."
```

Többszörös értékadás

A Pythonban egyszerre több változóhoz rendelhetünk értékeket. Például:

```
>>> x = y = 7
>>>x
7
>>> y
7
```

Egyetlen operátor segítségével *párhuzamosan* is végezhetünk (*ismétlődő*) *értékadásokat*:

```
>>> a, b = 4, 8.5
>>>a
4
>>> b
8.5
```

A többszörös értékadás egyszerűbbé teszi a változók értékeinek cseréjét:

```
>>> a, b = 4, 8.33
>>>a
4
>>> b
8.33
```

Függvények egymásba ágyazása

A Pythonban van lehetőség a függvényeket egymásba ágyazni, és azokat kombinálva nagyobb egységeket létre hozni. Például tudunk adatot bekérni a felhasználótól, át tudunk alakítani egy sztringet valós számmá, tudunk összetett kifejezéseket készíteni és értékeket megjeleníteni.

Írjunk egy programot, mely egy kör sugarát kéri be a felhasználótól, majd kiszámolja a kör területét. Először valósítsuk meg négy külön lépésben:

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

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

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

Példák

Hozzunk létre két változót interaktív módon (további üzenetek nélkül) – az egyiket a keresztnevének, a másodikat a vezetéknevének. Ezután egyetlen nyomtatási utasítás segítségével jelenítse meg őket egy sorban a képernyőn.

```
name = 'Imre' ; surname = 'Szabó'
print (surname + ' ' + name)
print (surname, name)
```

Írjunk egy programot, amely először a keresztnevet, majd vezetéknevet kéri, utána üzenetet jelenít meg ezekkel az adatokkal.

```
name = input()
surname = input()
print ('Üdvözöljük ' +surname + ' ' + name +  '!')
name = input('NÉV=')
surname = input('VEZETÉKNÉV=')
print ('Üdvözöljük',surname, name,'!')
```

Írjunk programot, amely bekéri egy téglalap alakú helyiség méreteit (centiméterben), és megjeleníti a szőnyeg területét, amely az egész padlót beborítja.

```
a = int(input('a='))
b = int(input('b='))
s = a * b
print ('A szőnyeg mérete', s, 'm^2')
print ('A szőnyeg mérete '+ str(s)+ ' m^2')
```

A számok beírásakor adattípus-konverziós függvényeket kell használnunk. Továbbá, amikor a kimenet során egy szövegláncot alkotunk összefűzéssel, minden adatot egyetlen str típusúra kell redukálni.

Írjunk egy programot, amely az előző feladatban még bekéri a szőnyeg négyzetméterének árát! A következő adatokat kell kiírni a képernyőre:

a szőnyeg teljes területe négyzetcentiméterben;

a szőnyeg teljes területe négyzetméterben;

a szőnyeg ára.

```
a = int(input('A szőnyeg hossza (cm) ='))
b = int(input('A szőnyeg szélessége (cm)='))
ár =int(input('A szőnyeg ára (m^2)='))
s = a * b
print ('A szőnyeg mérete 1 '+ str(s)+ ' cm^2')
print ('A szőnyeg mérete 2 '+ str(s/10000)+ ' m^2')
print ('A szőnyeg ára '+ str(s/10000*ár)+ ' gr')
```

Ír

"Hány 50 kopejkás érméd van?";
"Hány 25 kopejkás érméd van?";
"Hány 10 kopejkás érméd van?";
"Hány 5 kopejkás érméd van?".
Utána a teljes összegnek meg kell jelennie a képernyőn.

```
m50=int(input("Hány 50 kopejkás érméd van?"))
m25=int(input("Hány 25 kopejkás érméd van?"))
m10=int(input("Hány 10 kopejkás érméd van?"))
m5=int(input("Hány 5 kopejkás érméd van?"))
s=m50*0.5+m25*0.25+m10*0.1+m5*0.05
print('Az összeg egyenlő ' + str(s) + ' grivnya.')
```

Vissza

3. Adattípusokról bővebben

3.1. int (integer) típus

A Python-2 tartalmazot *long* típust. A Python-3 long típus nincs, bármilyen egész szám *int* típusú és a hossza nincs korlátozva. Pythonban az egész számokat dinamikusan méretezett egész szám típus (int) reprezentálja. Ez azt jelenti, hogy Pythonban nincs szükség külön hosszú típusra a nagyon nagy egész számok kezeléséhez, mert az `int` típus automatikusan megnövelhető méretű, attól függően, hogy a rendszer milyen hosszú számokat tud kezelni.

Pl.:

```
x = 12345678901234567890123456789012345678901234567890
print(type(x))  # <class 'int'>
```

Python bármilyen egész számot képes tárolni és kezelni.

Példa.

```
n=int(input('N='))
a, b, c = 3, 2, 1
while c <= n:
    print ('c=',c,': a*b=', b)
    a, b, c = b, a*b, c+1……..
```

N=12
c= 1 : a*b= 2
c= 2 : a*b= 6
c= 3 : a*b= 12
c= 4 : a*b= 72
c= 5 : a*b= 864
c= 6 : a*b= 62208
c= 7 : a*b= 53747712
c= 8 : a*b= 3343537668096
c= 9 : a*b= 179707499645975396352
c= 10 : a*b= 600858794305667322270155425185792
c= 11 : a*b= 1079788315649669138143849229447384578592430704390307 84
c= 12 : a*b=
648800305446607527907368373691049776950010342842280428918276494561862345826116074209 28

3.2.float típus

Ez a típus nagyon nagy vagy nagyon kis számokkal (például tudományos adatokkal) történő számolásokat tesz lehetővé állandó pontossággal. Ahhoz, hogy a Python egy numerikus adatot float típusúnak tekintsen, elegendő, ha a szám egy tizedes pontot, vagy 10-nek egy hatványkitevőjét tartalmazza.

Például a következő értékeket:
3.14
10.
.001

3e15
3.14e-10
a Python automatikusan *float* típusúként értelmezi.

Az előző programot megváltoztatjuk úgy, hogy az adatok típusa float legyen.

```
n=int(input('N='))
a, b, c = 3., 2, 1
while c <= n:
    print ('c=',c,': a*b=', b)
    a, b, c = b, a*b, c+1
```

N=12
c= 1 : a*b= 2
c= 2 : a*b= 6.0
c= 3 : a*b= 12.0
c= 4 : a*b= 72.0
c= 5 : a*b= 864.0
c= 6 : a*b= 62208.0
c= 7 : a*b= 53747712.0
c= 8 : a*b= 3343537668096.0
c= 9 : a*b= 1.797074996459754e+20
c= 10 : a*b= 6.008587943056673e+32
c= 11 : a*b= 1.079788315649669e+53
c= 12 : a*b= 6.488003054466074e+85

vagy:
N=18
c= 1 : a*b= 2
c= 2 : a*b= 6.0
c= 3 : a*b= 12.0
c= 4 : a*b= 72.0
c= 5 : a*b= 864.0
c= 6 : a*b= 62208.0
c= 7 : a*b= 53747712.0
c= 8 : a*b= 3343537668096.0
c= 9 : a*b= 1.797074996459754e+20
c= 10 : a*b= 6.008587943056673e+32
c= 11 : a*b= 1.079788315649669e+53
c= 12 : a*b= 6.488003054466074e+85
c= 13 : a*b= 7.005669890111831e+138
c= 14 : a*b= 4.545280764562657e+224
c= 15 : a*b= inf
c= 16 : a*b= inf
c= 17 : a*b= inf
c= 18 : a*b= inf

A float típus a 10^{-308} és 10^{308} közé eső (pozitív vagy negatív) számok 12 értékes számjegy pontossággal történő manipulációját teszi lehetővé. Ezek a számok speciálisan 8 byteon (64 biten) vannak kódolva a gép memóriájában: a kód egy része a 12 értékes számjegynek felel meg, a másik része a *nagyságrendnek* (10 hatványának).

Ha a szám nagyobb, mint 10^{308}, akkor azt nem lehet tárolni és a szám helyet «inf» («végtelen») szöveg jelenik meg.

Feladatok

- Írjon egy programot, ami a fokokban, percekben és másodpercekben megadott szögeket radiánba számolja át.
- Írjon egy programot, ami a radiánokban megadott szögeket fokokba, percekbe és másodpercekbe számolja át.
- Írjon egy programot, ami a bankban elhelyezett 4,3 % -os kamatozású tőke 20 év alatt felhalmozódott évi kamatait számolja ki. (Általánositani a programot!)
- Egy régi indiai legenda szerint a sakkjátékot egy öreg bölcs találta ki. A király meg akarta azt neki köszönni és azt mondta, hogy jutalmul bármilyen ajándékot megad érte. Az öreg azt kérte, hogy adjon neki egy kevés rizset öreg napjaira, pontosan annyi szem rizset, hogy az általa feltalált játék első kockájára 1 szemet, 2 második kockára kettőt, a harmadikra négyet, és így tovább egészen a 64-ik kockáig.
  Írjon egy Python programot, ami kiíratja a sakktábla 64 kockájának mindegyikére elhelyezett rizsszemek számát. Számolja ki ezt a számot kétféleképpen
  - a rizsszemek pontos száma (egész szám)
  - a rizsszemek száma tudományos jelölésmódban (valós szám)

3.3. sztring típus

A Pythonban sztring típusú **(alfanumerikus)** adat bármilyen karaktersorozat, amit vagy szimpla idézőjelek (aposztróf), vagy dupla idézőjelek határolnak. A sztring egy karakterláncot képez.

```
>>> mondat1 = 'a Pythont.'
>>> mondat2 = 'Én'
>>> mondat3 = "leggyakrabban alkalmazom"
>>> print (mondat2, mondat3, mondat1)
"Én leggyakrabban alkalmazom a Pythont.
```

Jegyezzük meg, hogy az olyan sztringeket, melyekben aposztrófok vannak, idézőjelekkel határoljuk, míg az idézőjeleket tartalmazó sztringeket aposztrófokkal határoljuk.

A «\» (backslash) néhány kiegészítő lehetőséget ad:

- Lehetővé teszi, hogy egy parancsot, ami nem fér el egy sorban, azt több sorba írjunk:
  print ('c=',c,': \
  a*b=', b)
- Egy karakterlánc belsejében a backslash speciális karakterek (sorugrás, aposztrofok, dupla idézőjelek, stb.) beszúrását teszi lehetővé.

A «\» (backslash) karakter a szövegben speciális szerepet tölt be, különösen akkor, ha karakterláncokban (stringekben) használjuk Pythonban és más programozási nyelvekben is. A backslash (`\`) karakter escape karakterként működik, ami azt jelzi a fordítónak vagy értelmezőnek, hogy a következő karakter ne a szokásos jelentésében legyen értelmezve, hanem valamilyen speciális módon.

Néhány fontosabb backslash karakter alkalmazása.

1. **Újsor (‘\n’)**: A `\n` escape szekvencia egy új sort jelöl, amikor egy karakterláncban használjuk. Például:

print("Ez az első sor.\nEz a második sor.")

Output:
Ez az első sor.
Ez a második sor.

2. **Tabulátor (`\t`)**: A `\t` escape szekvencia egy tabulátort jelöl, amikor használjuk.

print("Ez egy tabulátorral\telválasztott szöveg.")

Output:
Ez egy tabulátorral elválasztott szöveg.

3. **Önmagában a backslash (`\\`)**: Ha egy szövegben szeretnénk a backslash karaktert magát megjeleníteni, akkor kétszer kell írni: `\\`.

print("Ez egy backslash karakter: \\")

Output:
Ez egy backslash karakter: \

4. **Speciális karakterek elnyomása:** Például idézőjelek (`"` vagy `'`) használatában, amikor a karakterláncokat azonos típusú idézőjel bezárja, ha az azonos típusú idézőjel megjelenik a szövegben, ezt vissza kell szedni, hogy a program helyesen értelmezze.

print("Ez egy idézőjel: \"")

Output:
Ez egy idézőjel: "

A \' lehetővé teszi, hogy aposztrófokkal határolt karakterláncba aposztrófot szúrjunk be.

Ezek a példák mutatják, hogyan lehet a backslash karaktert használni a speciális karakterek megjelenítésére és különleges formázásra a szövegben Pythonban és más programozási nyelvekben is.

Hozzáférés a sztring karaktereihez

A karakterlánc (sztring) – *összetett adatoknak* egy speciális esete. Egy **összetett adat** egy olyan entitás, ami egyszerűbb entitások együttesét egyetlen struktúrában egyesíti. Egy karakterlánc esetében például ezek az egyszerűbb entitások maguk a karakterek.

A körülményektől függően a karakterláncot mint egységes objektumként, vagy mint különálló karakterek együtteseként lehet kezelni.

A Pythonban léteznek olyan eszközök, amelyek lehetővé teszik egy karakterlánc egyes karaktereihez való hozzáférést. Egy karakterláncot a Python úgy mint egy szekvenciát kezeli. A szekvencia elemei rendezettek. Ez azt jelenti, hogy egy sztring karakterei mindig egy bizonyos sorrendben vannak elrendezve. Következésként a sztring minden egyes karakterének meghatározható a helyét a szekvenciában **egy index** segítségével.

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

```
s = "iskola"
print s[0], h[3]
# i o
```

Sztringkezelő műveletek

A Pythonban karakterláncok kezelésére több függvény van (kis/nagybetűs átalakítás, rövidebb karakterláncokra való darabolás, szavak keresése, stb.). A sztringek esetében a + jel nem az összegzést, hanem az **konkatenálás (összekapcsolás, összefűzés) műveletet** jelöli. Az

összefűzéssel két sztringet kapcsolhatunk egymás után. A + operátor használatával rövidebb karakterláncból összerakhatunk egy hosszabbat.

```
a = 'Jó'
b =' napot!'
s = a + b
print(s)
# Jó' napot!
```

- Írjon egy programot, ami meghatározza, hogy egy karakterlánc tartalmazza-e az «e» karaktert.
- Írjon egy programot, ami megszámolja az «e» karakter előfordulásainak számát egy sztringben.
- Írjon egy programot, ami egy új változóba másol át egy karakterláncot úgy, hogy csillagot szúr be a karakterek közé.
  Így például, «gaston»-ból «g*a*s*t*o*n» lesz.
- Írjon egy programot, ami egy új változóba fordított sorrendben másolja át egy karakterlánc karaktereit.
  Így például «zorglub» -ból «bulgroz» lesz.
- Írjon egy scriptet, ami meghatározza, hogy egy karakterlánc palindrom e (vagyis ami mindkét irányból olvasva ugyan az), mint például «radar» vagy «sós».

A * művelet a sztringek esetén az ismétlés művelet hajt végre.

Például a 'ablak'*3 eredménye a ' ablakablakablak'. A művelet első operandusa sztring, a második – egész szám. A Pythonban a karakterláncok számos metódussal rendelkeznek. Néhány leggyakrabban használt metódus:

- *str.upper()*: Nagybetűs karakterláncot ad vissza.
- *str.lower()*: Kisbetűs karakterláncot ad vissza.
- *str.capitalize()*: Olyan karakterláncot ad vissza, amelyben az első betű nagybetűs, a többi karakter pedig kisbetűs.
- *str.title():* Egy karakterláncot ad vissza, amelyben minden szó nagybetűvel kezdődik.

Alkarakterláncok keresésének és cseréjének metódusai:

- *str.find(substring)*: Visszaadja a karakterláncban található részkarakterlánc első előfordulásának indexét (nulla alapú), vagy -1 értéket, ha az alkarakterlánc nem található.
- *str.replace(old, new)*: A régi részkarakterlánc minden előfordulását lecseréli az új részkarakterláncra.
- *str.startswith(str2)*: Ellenőrzi, hogy egy karakterlánc a *str2-el* kezdődik-e.
- *str.endswith(str2)*: Ellenőrzi, hogy egy karakterlánc végződik-e a str2-el.

A szóközök eltávolításának módjai:

- *str.strip():* Eltávolítja a szóközöket a karakterlánc elejéről és végéről.
- *str.lstrip():* Eltávolítja a szóközöket a karakterlánc elejéről.
- *str.rstrip():* Eltávolítja a szóközöket a karakterlánc végéről.

A karaktertípus ellenőrzésének metódusai:

- *str.isalpha():* Azt teszteli, hogy egy karakterlánc csak betűkből áll-e.
- *str.isdigit():* Azt teszteli, hogy egy karakterlánc csak számjegyekből áll-e.
- *str.islower(), str.isupper():* Ellenőrizze, hogy a karakterlánc csak kis- vagy nagybetűket tartalmaz-e.

A karakterláncok felosztásának és összefűzésének módjai:

- *str.join(str2):* Egy listából származó karakterláncokat egyesít, str sztringgel elválasztva.
- *str.split(separator)*: A megadott elválasztó segítségével részkarakterláncokra bontja a karakterláncot, és visszaadja az alsztringek listáját.

A split() metódusról bövebben

A Pythonban a split() metódus arra szolgál, hogy egy karakterláncot részkarakterláncokra bontja egy megadott elválasztó alapján, és visszaadja a részkarakterláncok listáját. A split() metódus paramétere split(szeparátor): A karakterláncot részkarakterláncokra bontja elválasztóként használva. Ha nincs megadva az elválasztó, akkor a Python szóközt használ elválasztóként.

```
text = "alma, banán, narancs"
fruits = text.split(",") # A karakterláncok elválasztása vesszővel
print(fruits) # Kimenet: ['alma', "banán", "narancs"]
```

Ebben a példában a split() metódus a karakterlánc szövegét vesszővel (,) osztja fel, és visszaadja az egyes szavakat a lista elemeként tartalmazó karakterláncok listáját.

Sok esetben hasznos lehet a split() metódus kombinálása a map függvénnyel:

```
x, y = map(int, input('x, y = ').split(' '))
```

map(function, iterable) függvény

• function: az iterálható elem egyes elemeire alkalmazott függvény.
• iterable: olyan elemek sorozata, amelyekre a függvényt alkalmazni kell.

```
def square(x): # square függvény definiálása
    return x * x
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers))  #  [1, 4, 9, 16, 25]
```

P

 # 1----------

```
words[]
words = ['alma', 'banán', 'körte', 'szilva']
delimiter = ', '
result = delimiter.join(words)
print(result)  # alma, banán, körte, szilva
```

 # 2----------

```
string = " alma, banán, körte, szilva"
delimiter = ","
result = string.split(delimiter)
print(result)  #  ['alma', 'banán', 'körte', 'szilva']
```

Vissza

4. Feltételes utasítások

Relációs operátorok

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

x == y

x != y

x > y

x < y

x >= y

x <= y

if utasítás

A feltételes utasítások közül az ***if*** utasítás a legegyszerűbb.

```
a = 150
if (a > 100):
      print =("nagyobb, mint 100")
```

elif és else utasítások

Az **else** és **elif** (az «else if» összevonása) utasítások alkalmazásával lehet bonyolultabb feladatokat megoldani:

```
a = 20
if (a > 100):
... print ("nagyobb, mint 100")
else:
.   print ("nem nagyobb, mint 100")
```

```
a = 0
if a > 0:
      print(‘a pozitiv’)
elif a<0:
      print(‘a negativ’)
else:
      print(‘a nulla’)
```

Példák.

A négy szám közül a legkisebbet kell megtalálni. A program beolvas a billentyűzetről egy 4 egész számból álló sztringet, meghatározza és a képernyőn megjeleníti közülük a legkisebbet.

```
 a, b, c, d = map(int, input().split())
m = a
if b< m:   m = b
if c< m:   m = c
if d< m:   m = d
print(‘min= ’, m)
```

```
a = int(input(‘a=’))
if (a % 2 == 0):
      print ("a páros")
      print ("mert 2-vel való osztása esetén a maradék nulla")
else:
      print(„a páratlan”)
```

Írjunk programot, amely meghatározza, hogy a beírt szám külön-külön osztható-e 7-tel, 11-gyel vagy 13-mal. Ha nem osztható a megadott számok egyikével sem, ne jelenítsen meg semmit a képernyőn.

```
 a=int(input())
if a % 7 == 0:
      print(’A ’+str(a)+’ szam ’+’osztható 7-re!’)
if a % 11 == 0:
      print(’A ’+str(a)+’ szam ’+’osztható 11-re!’)
if a % 13 == 0:
      print(’A ’+str(a)+’ szam ’+’osztható 13-ra!’)
```

Írjon programot, amely meghatározza, hogy a beírt osztályzat melyik oktatási teljesítményszinthez tartozik, ha a beírt szám nem tartozik az [1..12] tartományba - jelenítse meg a megfelelő üzenetet.

```
 a=int(input())
if 0<a<= 3:
      print(’2 ….’)
if 3<a<= 6:
      print(’3 ….’)
if 6<a<= 9:
      print(’4 ….’)
if 9<a<= 12:
      print(’5 ….’)
else:
      print(nincs ilyen jegy! )
```

Vissza

5. Ciklusok

5.1 while ciklus utasítás

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

Ez a program a következőképpen működik:

- A while utasítás esetén a Python a zárójelben levő feltétel kiértékelésével kezdi.
- Ha a feltétel hamis, akkor a következő blokkot figyelmen kívül hagyja és a programvégrehajtás befejeződik.
- Ha a feltétel igaz, akkor a Python a ciklustestet alkotó utasításblokkot végrehajtja. (A példában: az a = a + 1 utasítást, ami 1-gyel inkrementálja az a változó tartalmát. Utána a print utasítást, ami kiíratja az a változó aktuális értékét).
- a vezérlés visszatér a while utasítást tartalmazó sorra. Az ott található feltételt újra kiértékeli és így (ciklikusan) tovább.
- A while ciklusból a **break** utasítással is lehet kilépni.

Táblázatkészítés. Példa

Elemezzük a következő programot. Az 1-től 12-ig terjedő számok négyzetének és köbének listáját kell megkapnunk:

```
a = 0
while a < 12:
     a = a +1
     print (a , a**2 , a**3)
```

A print utasítás lehetővé teszi, hogy ugyanabba a sorba több kifejezést írassunk ki, egyiket a másik után : elég vesszővel elválasztani őket. A Python automatikusan beszúr egy szóközt a kiírt elemek közé.

Egy matematikai sor megalkotása

Az alábbi program a Fibonacci sor első tíz elemét íratja ki.

A Fibonacci sor olyan számsor, amelynek minden tagja egyenlő az őt megelőző két tag összegével. Elemezzük ezt a (többszörös értékadást helyesen alkalmazó) programot.

```
a, b, c = 1, 1, 1
while c < 11 :
      print (b)
     a, b, c = b, a+b, c+1
```

A program a következő eredményt ad:
1
2
3
5

8
13
21
34
55
89

Ha az eredményt egy sorba akarjuk elhelyezni, akkor:

```
a, b, c = 1, 1, 1
while c < 11 :
      print (b, end=' ')
     a, b, c = b, a+b, c+1
```

A program a következő eredményt ad:
1 2 3 5 8 13 21 34 55 89

Egy másik variáns:

```
a, b, c = 1, 1, 1
while c < 11 :
      print (b, end=', ')
     a, b, c = b, a+b, c+1
```

A program a következő eredményt ad:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

Ha az utolsó vesszőt ki akarjuk hagyni, akkor a programot igy módosítjuk:

```
a, b, c = 1, 1, 1
while c < 11 :
      if c<10:
            print (b, end=', ')
      else:
            print (b)
      a, b, c = b, a+b, c+1
```

1, 2, 3, 5, 8, 13, 21, 34, 55, 89

A következő táblázat által megvizsgálhatjuk, hogyan alakulnak lépésről lépésre a változók értékei a while ciklusban. Ilyen táblázatok segítik a programok tesztelését.

Változók kezdő értékei	a=1	b=1	c=1
Helyettesítő kifejezések	b	a+b	c+1
Az iteráció során egymás után felvett értékek	1 2 3 5 ...	2 3 5 8 ...	2 3 4 5 ...

Készítsünk egy scriptet, amely kiszámítja a Fibonacci sort az N-ik számig. Ezt a scriptet a *fibon.py* fájlban fogjuk tárolni.

```
n=int(input('N='))
a, b, c = 1, 1, 1
while c < n :
    if c<n-1:
        print (b, end=', ')
    else:
        print (b)
    a, b, c = b, a+b, c+1
```

N=15
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

A fibon.py scriptet módosítsuk úgy, hogy az kiírja csak az N-ik Fibonacci számot és annak a típusát.

```
n=int(input('N='))
a, b, c = 1, 1, 1
while c < n :
     a, b, c = b, a+b, c+1
print ('Fib(',n,')=',b,'Az eredmény típusa:=', type(b))
```

Az eredmény:
N=20
Fib(20)= 10946 Az eredmény típusa:= <class 'int'>

Még egy módosítás. A script írja ki a Fibonacci sorozat elemeit a N-től a M-ikig (M>N).

```
n=int(input('N='));  m=int(input('M='))
a, b, c = 1, 1, 1
while c < m :
     a, b, c = b, a+b, c+1
     if n<=c<=m:
        print ('Fib(',c,')=',b,'Az eredmény típusa:=', type(b))
```

Az eredmény:
N=43
M=49
Fib(43)= 701408733 Az eredmény típusa:= <class 'int'>
Fib(44)= 1134903170 Az eredmény típusa:= <class 'int'>
Fib(45)= 1836311903 Az eredmény típusa:= <class 'int'>
Fib(46)= 2971215073 Az eredmény típusa:= <class 'int'>
Fib(47)= 4807526976 Az eredmény típusa:= <class 'int'>
Fib(48)= 7778742049 Az eredmény típusa:= <class 'int'>
Fib(49)= 12586269025 Az eredmény típusa:= <class 'int'>
Fib(50)= 20365011074 Az eredmény típusa:= <class 'int'>

vagy:
N=100
M=105
Fib(100)= 573147844013817084101 Az eredmény típusa:= <class 'int'>

Fib(101)= 927372692193078999176 Az eredmény típusa:= <class 'int'>
Fib(102)= 1500520536206896083277 Az eredmény típusa:= <class 'int'>
Fib(103)= 2427893228399975082453 Az eredmény típusa:= <class 'int'>
Fib(104)= 3928413764606871165730 Az eredmény típusa:= <class 'int'>
Fib(105)= 6356306993006846248183 Az eredmény típusa:= <class 'int'>

- Írjon egy programot, ami kiíratja a 7-es szorzótábla első 20 tagját.
- Írjon egy programot, ami kiíratja a 7-es szorzótábla első 20 tagját, csillaggal jelölve azokat, amelyek 3-nak többszörösei.
  Példa : 7 14 21 * 28 35 42 * 49
- Írjon programok, amely kiszámítsa az első N természetes szám szorzatát!
- Írjon egy programot, ami átszámolja a megadott egészszámú másodpercet évekké, hónapokká, napokká, percekké és másodpercekké.
- Írjon egy programot, ami kiszámolja 13-as szorzótábla első 50 tagját, de csak azokat írja ki, melyek 7-nek többszörösei.
- Írjon egy programot, ami a következő jelsorozatot írja ki :

```
*
**
***
****
*****
******
*******
```

5.2. FOR ciklus

A *for ciklus* egy adott számú iterációt hajt végre, amely megfelel az iterálható objektum elemeinek számának.

A for ciklus Pythonban egy iterációs szerkezet, amely lehetővé teszi, hogy egy iterálható objektum összes elemén végig iteráljunk.

Az iterálható objektum lehet **lista, tuple, string, vagy bármilyen más olyan adatszerkezet, amelyben szekvenciálisan végig tudunk lépkedni.**

A for ciklus általános szintaxisa a következő:

```
for elem in iterálható_objektum:
    # Ciklusmag
    # Ebben a blokkban dolgozunk az 'elem' változóval
```

Például, végigmenni egy listán és kiírni az összes elemét:

```
my_list = [1, 2, 3, 4, 5]
for number in my_list:
    print(number)
```

Ez a kód a *my_list* lista minden elemét sorban ki fogja írni:

1
2

3
4
5

A for ciklus nagyon hasznos akkor, amikor szükségünk van arra, hogy egy adott végezzünk.

```
for i in range(4):
      print(i) # 0,1,2,3
```

A `for` ciklus a Pythonban nem csak a `range()` függvénnyel működik. Bármilyen iterálható objektummal használható, amelynek elemeit végig lehet iterálni. Néhány példa a `for` ciklus használatára.

Lista

```
list1 = [1, 2, 3, 4, 5]
for elem in list1:
    print(elem)
```

Sztring

```
my_string = "hello"
for char in my_string:
    print(char)   #  h e l l o
```

Szótár

```
dict1 = {'a': 1, 'b': 6, 'c': 3} #  1 2 3 kulcs a b c
for key in dict1:
    print(key, dict1[key])
```

Eredmény:
a 1
b 6
c 3

file objektum

```
with open('file.txt', 'r') as file:
    for line in file:
        print(line)
```

Generátor
A **`for`** ciklusokban használhatunk generátorokat is. A ciklus automatikusan befejeződik, amikor a

generátor kimerül:

```
def my_generator():
    yield 1
    yield 2
    yield 3
for num in my_generator():
    print(num)
```

Halmaz

```
my_set = {1, 2, 3, 4, 5}
for elem in my_set:
    print(elem)
```

5.3.Iterálható objektumok

Iterálható objektum – olyan objektum, amely sor elemeket tartalmaz és a **for** ciklusban használható.
A Pythonban több beépített adattípus és adatstruktúra iterálható:
1. Listák: A listák rendezett elemkészleteket tartalmaznak, amelyek bármilyen adattípusúak lehetnek. (tömbre alkalmazható)
2. Tuples: A sorok az elemek megváltoztathatatlan rendezett halmazait képviselik.
3. Karakterláncok: A karakterláncok karakter sorozatokat képviselnek, és karakterenként ismételhetők.
4. Szótárak: A szótárak kulcsokat és értékeket tartalmaznak, és ismételhetők kulcsokon vagy értékeken.
5. Halmazok: A halmazok egyedi elemeket tartalmaznak, és elemeken át ismételhetők.
6. Fájlok: Amikor a fájlokat Pythonban olvassa, iterálható objektumokká válnak, amelyek *soronként* ismételhetők a fájlban.
7. Generátorok: A generátorok generátorkifejezések vagy generátorfüggvények segítségével jönnek létre, és adatsorozatok létrehozására használhatók.

A for ciklus Pythonban lehetővé teszi, hogy kényelmesen iteráljon bármely iterálható adatstruktúra elemei között anélkül, hogy kifejezetten indexeket adna meg vagy további műveleteket hajtana végre. Ez hasznos és fontos a ciklusok alkalmazása szempontjából.

Iterálható objektumok alkalmazása. Példák

range() függvény

A range() függvény egy adott tartományon belüli *számsorozat* létrehozására szolgál. Alkalmazása:

1. **Iterálás egy számsorozaton:** A range() segítségével létrehozhatunk egy számsorozatot, amelyet aztán alkalmazunk a **for** ciklusban. Például:

```
for i in range(5):
    print(i)   #  0,1,2,3,4
```

2. tartományban. Például:

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

Ez a kód hozza létre a [1, 2, 3, 4, 5] listát.

3. **Sorozat lépéssel**: A sorozat lépését úgy lehet megadni, mint a harmadik argumentumot a range() függvényben. Például:

```
for i in range(0, 11, 2):
    print(i)  # 0, 2,4, 6, 8, 10
```

4. **A range használható más függvényekkel is,** mint például a **zip()**, értékpárok létrehozására, vagy az **enumerate()** segítségével, hogy egy sorozatban lévő elemek indexeit kapjuk meg.

zip() függvény

A **zip() függvény** több iterálható objektum (például listák vagy karakterláncok) egyetlen iterálható objektummá kombinálására szolgál úgy, hogy az objektumok megfelelő elemeiből sorokat hoz létre. Példa:

```
list1 = [1, 2, 3]; list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)
for item in zipped:
    print(item)
```

Eredmény:
(1, 'a')
(2, 'b')
(3, 'c')

A zip() függvény a list1 és a list2 elemeit pozíció szerint egyesíti, és a megfelelő elemekből sorokat hoz létre. Ha az iterálható objektumok egyike rövidebb, mint a többi, a zip() leáll, amikor a legrövidebb objektum elfogy.
A zip() arra is használható, hogy a sorok elemeit külön listákba bontsa ki. Például:

```
zipped = [(1, 'a'), (2, 'b'), (3, 'c')]
list1, list2 = zip(*zipped)
print(list1)  #  (1, 2, 3)
print(list2)  #  ('a', 'b', 'c')
```

csillag-operátor (*)

A csillag-operátor (*) vagy **csomagoló operátor**, a Pythonban több helyen használatos, és különböző kontextusokban különböző jelentéssel bírhat. Néhány gyakori hely, ahol használják:

1. **Csillag-operátor bontásnál**: A csillagot használhatjuk egy iterálható objektum bontására elemeire. Például:

```
n = [1, 2, 3, 4, 5]
a, *b,d, c = n   #    a,b=2,5     a,b=b,a
print(a)  # 1
print(b)  # [2, 3, 4]
print(c)  # 5
```

1
[2, 3]
5

Ebben az esetben a csillag (`*b`) összegyűjti a köztes elemeket egy listában, miközben `a` és `c` a lista első és utolsó elemét kapja meg.

Csillag-operátor többváltozós függvényeknél: A csillagot használhatjuk egy függvény hívásakor több argumentum átadásához. Például:

```
def my_function(a, b, c):
    print(a, b, c)
values = [1, 2, 3]
my_function(*values)  # 1 2 3
```

Csillag-operátor iterátoroknál: A csillagot használhatjuk egy iterátor elemeinek "kicsomagolására" egy listává vagy tuple-vá. Például:

```
numbers = [1, 2, 3]
values = [4,*numbers, 5, 6]
print(values)  # [4,1, 2, 3, 5, 6]
```

Csillag-operátor kicsomagolásnál: A csillagot használhatjuk egy iterálható objektum kicsomagolására. Például:

```
numbers = [1, 2, 3]
print(*numbers)  # 1 2 3
print(numbers)  # [1, 2, 3]
```

Ebben az esetben a `numbers` lista elemeit egy listába csomagoljuk a csillagoperátorral.

enumerate() függvény

Az enumerate() egy beépített függvény, amely hasznos, amikor egy iterálható objektum elemein szeretnénk végig iterálni, és közben szükség van az aktuális elem indexére is. Ez a függvény egy felsorolást (enumerate object) ad vissza, amely tartalmazza az iterálható objektum elemeinek indexét és magukat az elemeket is.

```
my_list = ['apple', 'banana', 'cherry']
# enumerate() használata egy ciklusban
for index, value in enumerate(my_list):
    print(index, value)
```

Kimenet:
0 apple
1 banana
2 cherry

Magyarázat:
Az enumerate() függvény a my_list lista elemein iterál.
Minden iteráció során az index változó az aktuális elem indexét tartalmazza (nullától kezdve), míg a value változó az aktuális elem értékét (`'apple'`, `'banana'`, `'cherry'` stb.) tartalmazza.

További példák és opciók:
Kezdő érték megadása az indexnek:

```
   for index, value in enumerate(my_list, start=1):
       print(index, value)
```

Ez az enumerate() függvénynek a start paraméterrel együtt adja meg az index kezdő értékét. Ebben az esetben az indexek 1-től kezdődnek.

Enumerate használata tuple-ben:

```
# Tuple létrehozása
my_tuple = ('apple', 'banana', 'cherry')
for index, value in enumerate(my_tuple):
      print(index, value)
```

Az enumerate() ugyanúgy használható tuple-ökön is, és az eredmény hasonló lesz, mint a lista esetén.

Enumerate használata más iterálható objektumokon:

Az enumerate() függvény nemcsak listákon és tuple-ökön működik, hanem bármely iterálható objektumon (pl. sztringeken, halmazokon stb.) is.

Az enumerate() függvény használata rendkívül hasznos, amikor szükség van az elemek indexére a feldolgozás során, például ciklusokban vagy más iterálás alapú műveletek során.

5.4. Összetett utasítások – Utasításblokkok

A Pythonban minden összetett utasításnak mindig ugyanaz a szerkezete:

- **egy fejsor, ami kettőspontra végződik,**
- **alatta következik egy vagy több utasítás, ami a fejsor alatt be van húzva.**

Ha a fejsor alatt több behúzott utasítás van, akkor mindegyiknek pontosan ugyanannyira kell behúzva lenni (például 4 karakterrel kell beljebb lenni). Ezek a behúzott utasítások alkotják az utasításblokkot.

Az utasításblokk: egy logikai együttest képező utasítások sorozata, ami csak a fejsorban megadott feltételek teljesülése esetén lesz végrehajtva.

Egymásba ágyazott utasítások

Komplex döntési struktúrák létrehozásához egymásba ágyazható több összetett utasítás.

Példa.

```
if torzs == "gerincesek":                                    # 1

    if osztaly == "emlősök":                                 # 2
        if rend == "ragadozók":                              # 3

            if csalad == "macskafélék":             # 4

                print "ez egy macska lehet"         # 5

        print "ez minden esetre egy emlős"          # 6

    elif osztaly == 'madarak':                      # 7

        print "ez egy kanári is lehet"              # 8

print "az állatok osztályozása összetett"           # 9
```

Python néhány szintaktikai szabálya

Összefoglalunk néhány szintaktikai szabályt:

- Az utasítások és a blokkok határait a sortörés definiálja.

Számos programozási nyelvben minden sort speciális karakterrel kell befejezni (gyakran pontos vesszővel). A Pythonban a sorvége jel játssza ezt a szerepet. Egy utasítássort kommenttel is befejezhetünk.

- Egy Python komment mindig a # karakterrel kezdődik. Ami a # karakter és a LF (linefeed) karakter között van, a fordító figyelmen kívül hagyja.
- A Pythonban használnunk kell a sor ugrásokat és a behúzásokat, viszont nem kell más blokkhatároló szimbólumokkal foglalkoznunk.
- Az utasításblokkok mindig egy jóldefiniált utasítást tartalmazó fejsorhoz kapcsolódnak (if, elif, else, while, def, ...), ami kettőspontra végződik.
- A blokkokat behúzás határolja: a blokk minden sorának pontosan ugyanúgy kell behúzva lenni (vagyis ugyanolyan számú betűközzel kell jobbra eltolni). A behúzásra akármennyi szóközt használhatunk.
- A legkülső blokknak a bal margón kel lennie.

6. Listák

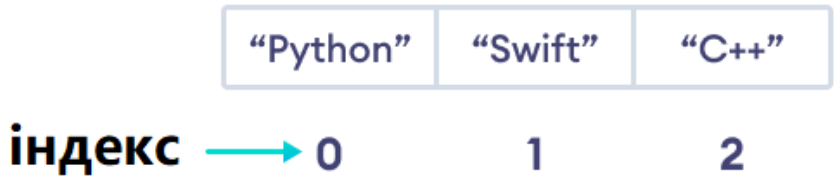
Pythonban nincs külön tömb adatszerkezet, helyette van a lista, ami kvázi egy dinamikus tömb, változó mérettel. Egy listába bármit tárolhatunk, de célszerű egyfajta típusú elemet beletenni.

```
list1 = []  # ures lista letrehozasa
list2 = list()  # ures lista letrehozasa
list(1,3,5)   # [1,3,5]
list3 = [2, 4, 6, 8, 10]  # Lista letrehozasa elemekkel
# A lista bármilyen típusú elemet tárolhat
v = 25
list4 = [1, 2, 3, 1.1, 'elem', 3, valt, 123]
```

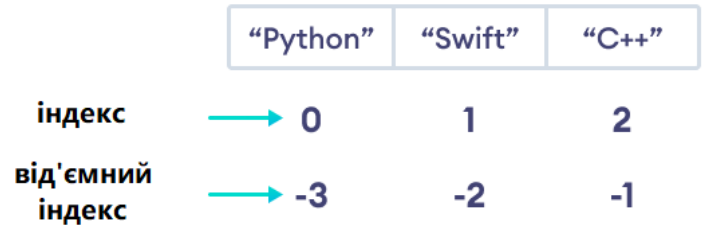
A lista minden eleme egy indexhez van társítva. Az elemeket indexei: (0, 1, 2,…).

Lista elemeinek elérése Pythonban

```
languages = ["Python", "Swift", "C++"]
print(languages[0])   # Python
print(languages[2])   #  C++
```



Listák indexelése lehet negatív is, ilyenkor a végéről ad vissza elemet. Ha túlindexeljük a listát, *kivételt* kapunk.




```
print('list4 utolso eleme:', list4[-1])
print('list4 utolso elotti eleme:', list4[-2])
print(list4[-100]) # Ennek kimenete: IndexError hiba
```

A listaelemek módosítása

A Python listái módosíthatók. Ez azt jelenti, hogy a lista elemeit úgy tudjuk megváltoztatni, hogy új értékeket adunk nekik az = operátor segítségével. Példa:

```
languages = ['Python', 'Swift', 'C++']
languages[2] = 'C'
print(languages) # ['Python', 'Swift', 'C']
```

Lista szeletelése

A Pythonban a *slicing ([:])* operátor használatával egyszerre több elemet érhetünk el.

```
my_list = ['p','r','o','g','r','a','m','i','z']
print(my_list[2:4])
print(my_list[5:])
print(my_list[:])
 print(my_list[:2]) # p r
```

Eredmény:

['o', 'g'] # 2,3
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']

Megjegyzés: a listákon a szeletelésnél az *első index a befogadó, az utolsó index pedig kizárólagos.*

append metódus

Az append metódus segítségével szúrhatunk be egy új elemet a lista végére:
list1.append(n)

```
numbers = [21, 34, 54, 12]
print("Before Append:", numbers)
numbers.append(32)
 print("After Append:", numbers)
```

Eredmény:

Before Append: [21, 34, 54, 12]
After Append: [21, 34, 54, 12, 32]

in operátor

Az **in** operátor nagyon hasznos és gyakran használt Python-ban a listákhoz való hozzáférés során. Ez az operátor lehetővé teszi számunkra, hogy ellenőrizzük, hogy egy adott elem szerepel-e egy listában vagy sem.

Példa 1: Egyszerű jelenlét ellenőrzése

```
numbers = [1, 2, 3, 4, 5]
# Ellenőrizzük, hogy a 3 szerepel-e a listában
if 3 in numbers:
    print("Az 3 szerepel a listában")
# Ellenőrizzük, hogy a 10 szerepel-e a listában
if 10 in numbers:
    print("Az 10 szerepel a listában")
else:
    print("Az 10 nem szerepel a listában")
```

Példa 2: A művelet karakterláncokkal:

```
fruits = ["alma", "körte", "szilva"]
# Ellenőrizzük, hogy "alma" szerepel-e a listában
if "alma" in fruits:
    print("Az 'alma' szerepel a gyümölcsök között")
```

index metódus

A Python listáinak hasznos metódusa az *index()* metódus, amely lehetővé teszi számunkra, hogy megtaláljuk a listában egy adott elem indexét.

Példa 1. Egyszerű lista és index keresése

```
numbers = [10, 20, 30, 40, 50]
# Keresünk egy adott elem indexét
index_30 = numbers.index(30)
print("Az 30 szerepel a lista", index_30, ". helyén")
```

Példa 2. Ismétlődő elemek kezelése

```
numbers2 = [10, 20, 30, 40, 30, 50]
# Az első előfordulás indexének keresése
index_30_first = numbers2.index(30)
print("Az 30 szerepel először a lista", index_30_first, ". helyén")
```

Példa 3. Elem jelenlétének ellenőrzése először

```
if 40 in numbers2:
    index_40 = numbers2.index(40)
    print("Az 40 szerepel a lista", index_40, ". helyén")
else:
    print("Az 40 nem szerepel a listában")
```

Példa 4. Hiba kezelése, ha az elem nem található a listában

```
try:
    index_60 = numbers2.index(60)
    print("Az 60 szerepel a lista", index_60, ". helyén")
except ValueError:
    print("Az 60 nem szerepel a listában")
```

Példa 5. Lista szakaszán belül keresés

```
numbers3 = [10, 20, 30, 40, 30, 50]
ind = numbers3.index(30, 2)
print("Az 30 a lista 2. indexe után szerepel", ind, ". helyén")
```

Ez
megjegyezni néhány dolgot:
- Ha az elem többször is előfordul a listában, akkor az index() metódus az első előfordulás indexét adja vissza.
- Ha az adott elem nem található a listában, az index() metódus `ValueError` kivételt dob. Ezért érdemes először az in operátorral ellenőrizni az elem jelenlétét, mielőtt meghívnánk ezt a metódust.
- Opcionálisan megadható egy második argumentum is a index() metódusnak, ami megadhatja a keresés kezdőindexét a listán belül. Ez hasznos lehet, ha csak egy adott részén keresünk egy listának.

Lista másolása

A Python-ban a listák részeit könnyen másolhatjuk, és ez számos különböző módon történhet attól függően, hogy mit szeretnénk elérni. Itt van néhány módszer a lista részének másolására:

 # Teljes lista másolása

Ha teljesen akarjuk másolni egy listát, de nem akarjuk, hogy az eredeti lista és a másolt lista ugyanazokat a memóriaterületeket használják, akkor használhatjuk a copy() metódust vagy a lista slicing-ot.

 # Használva a copy() metódust:

```
list = [1, 2, 3, 4, 5]
copied_list = list.copy()
# Most már két különböző lista van
print(list)  # [1, 2, 3, 4, 5]
print(copied_list)    # [1, 2, 3, 4, 5]
# Megváltoztatjuk az eredeti listát
list[0] = 10
# A másolt lista nem változik
print(list)  # [10, 2, 3, 4, 5]
print(copied_list)    # [1, 2, 3, 4, 5]
```

 # Használva a list slicing-ot:

```
list = [1, 2, 3, 4, 5]
copied_list = list[:]
# Most már két különböző lista van
print(list)  # [1, 2, 3, 4, 5]
print(copied_list)    # [1, 2, 3, 4, 5]
# Megváltoztatjuk az eredeti listát
list[0] = 10
# A másolt lista nem változik
print(list)  # [10, 2, 3, 4, 5]
print(copied_list)    # [1, 2, 3, 4, 5]
```

 # Lista részének másolása. Ha csak egy adott részét szeretnénk másolni egy listának, például egy szeletet, akkor a list slicing-et használhatjuk.

```
list = [1, 2, 3, 4, 5]
copied_part = original_list[1:4]
print(list)  # [1, 2, 3, 4, 5]
print(copied_part)    # [2, 3, 4]
# Megváltoztatjuk az eredeti listát
list[1] = 20
# A másolt rész nem változik
print(list)  # [1, 20, 3, 4, 5]
print(copied_part)    # [2, 3, 4]
```

Fo
így a másolat és az eredeti lista külön memóriaterületeken van tárolva. Ezáltal biztosítja, hogy a módosítások az egyik listán ne befolyásolják a másikat.

Lista elemének törlése

A Pythonban több módon is törölhetünk elemeket egy listából, attól függően, hogy pontosan mit szeretnénk elérni: ha tudjuk az elem indexét, vagy ha csak az elem értékét ismerjük. Néhány példa a különböző lehetőségekre:

Elem törlése index alapján (del vagy pop())
del kulcsszó használata:
Ha tudjuk az elem indexét, akkor a del kulcsszóval tudjuk törölni az adott indexű elemet.

```
numbers = [1, 2, 3, 4, 5]
# Törlés az index alapján (pl. a 3. indexű elemet)
del numbers[2]
print(numbers)  # [1, 2, 4, 5]
```

pop() metódus használata:
A pop() metódus az adott indexű elemet töröli és vissza is adja annak értékét.

```
numbers = [1, 2, 3, 4, 5]
# Törlés és visszaadás az index alapján (pl. a 3. indexű elemet)
elem = numbers.pop(2)
print(numbers)          # [1, 2, 4, 5]
print(elem)  # 3
```

Elem törlése érték alapján (remove() metódus)

Ha csak az elem értékét ismerjük és azt akarjuk törölni a listából:

```
numbers = [1, 2, 3, 4, 5, 2]
# Törlés az érték alapján (pl. az 2-es értéket)
numbers.remove(2)
print(numbers)  # [1, 3, 4, 5, 2]
```

Fontos megjegyezni, hogy a remove() metódus csak az első előfordulást töröli az adott érték alapján. Ha több azonos értékű elemet szeretnénk törölni, akkor többször is alkalmazhatjuk ezt a metódust.

Törlés érték alapján összes előfordulás esetén (list comprehension)

Ha az összes előfordulást szeretnénk törölni egy érték alapján:

```
numbers = [1, 2, 3, 4, 5, 2]
# Törlés az érték alapján (pl. az 2-es értéket)
numbers = [x for x in numbers if x != 2]
print(numbers)  # [1, 3, 4, 5]
```

Ez a megoldás egy list comprehension-t használ, amely új listát hoz létre minden olyan elemmel, ami nem egyezik meg az eltávolítandó értékkel.

count metódus

A Python listák count() metódusa arra szolgál, hogy megszámolja, hogy egy adott elem hányszor fordul elő a listában. Ez hasznos lehet, ha szeretnénk megtudni, hogy hány darab adott értékű elem van egy listában.

```
numbers = [1, 2, 3, 4, 2, 5, 2]
# Számláljuk meg, hogy hány darab 2-es van a listában
two = numbers.count(2)
print("A lista tartalmaz", two, "db 2-es elemet.")
```

Ebben a példában a numbers listában két darab 2-es szerepel, így a two változó értéke 2 lesz. A count() metódus pontosan egy argumentumot vár: az elemet, amelynek előfordulásait megszeretnénk számolni a listában.
Ha az adott elem nem szerepel a listában, akkor a count() metódus 0-t ad vissza.

```
numbers = [1, 2, 3, 4, 5]
# Számláljuk meg, hány darab 10-es van a listában (nincs 10-es)
tens = numbers.count(10)
print("A lista tartalmaz", tens, "db 10-es elemet.")
```

Ebben az esetben a listában nincs 10-es elem, így tens értéke 0 lesz.

Listák összefűzése (+ operátor, extend metódus)

A Pythonban több módon is összefűzhetünk két vagy több listát. Van két gyakori módszer, amelyeket a + operátor és az extend() metódus segítségével végezhetünk el.

+ operátor használata
A + operátorral két listát összefűzhetünk, létrehozva egy új listát, amely tartalmazza mindkét

```
lista elemeit.
list1 = [1, 2, 3]
list2 = [4, 5, 6]
# Összefűzés a + operátorral
result = list1 + list2
print(result)  # [1, 2, 3, 4, 5, 6]
```

Ebben az esetben a result lista tartalmazza list1 és list2 elemeit egymás után.

extend() metódus használata
Az extend() metódus segítségével egy listához hozzáadhatjuk egy másik lista elemeit. Ez a művelet az eredeti listát módosítja, nem hoz létre új listát.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
# list1 kiterjesztése list2-vel
list1.extend(list2)
print(list1)  # [1, 2, 3, 4, 5, 6]
```

Ebben az esetben list1 lesz kiterjesztve list2 elemeivel, így list1 módosul.

Listák sokszorozására (* operátor)

A listák sokszorozására használható az `*` operátor, amely lehetővé teszi, hogy egy listát többször ismételjünk meg.

```
list1 = [1, 2, 3]
list2 = list1 * 3
print(list2)  # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

A szorzás száma lehet negatív vagy nulla is, azonban ekkor az eredmény egy üres lista lesz.

```
list1 = [1, 2, 3]
list2 = list1 * -2
print(list2)  # []
```

join() metódus. Lista elemeinek összefűzése egy sztringbe

A join() metódus – hasznos eszköz arra, hogy egy listát egyetlen sztringgé alakítsunk át, miközben a listaelemek közé beszúrunk egy elválasztó sztringet. Ez különösen hasznos akkor, amikor listaelemeket szeretnénk összefűzni egy szöveges formában, például szavakat egy mondatba vagy számokat egy CSV sorba.

Példa 1. Szavak összefűzése mondatként

```
words = ["Ez", "egy", "példa", "sztring", "összefűzésére"]
# Szóközökkel összefűzés
s = " ".join(words)
print(s)  # Ez egy példa sztring összefűzésére
```

Ebben a példában a join() metódus a words listát összefűzi egyetlen sztringgé, ahol a szavak között egy szóköz van elhelyezve.

Példa 2. Számok összefűzése CSV sorba

```
numbers = [1, 2, 3, 4, 5]
# Vesszővel elválasztva összefűzés
csv_row = ",".join(map(str, numbers))
print(csv_row)  # 1,2,3,4,5
```

Ebben a példában a map(str, numbers) segítségével a numbers lista elemeit szöveggé alakítjuk (str), majd ezeket vesszővel (",") összefűzzük egyetlen sztringgé, ami egy CSV sort eredményez.

Megjegyzések:

- A join() metódus egy sztringet vár paraméterként, amelyet az összefűzőt elemek közé szúr be.
- A join() metódus csak sztringeket tud kezelni, ezért ha listában nem sztringek vannak, először ezeket szöveggé kell alakítani.
- A join() metódus hatékonyabb, mint a `+` operátor használata, mert a join() metódus optimalizálja sztringek összefűzésére.

Listák rendezése

A Pythonban többféle módon rendezhetünk listákat, attól függően, hogy növekvő vagy csökkenő sorrendben szeretnénk rendezni, illetve hogy módosítani szeretnénk-e az eredeti listát vagy egy új listát szeretnénk létrehozni. Néhány példa a különböző rendezési módszerekre.

1. sort() metódus

A sort() metódus módosítja az eredeti listát és növekvő sorrendbe rendezi az elemeit:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
numbers.sort()
print(numbers)  # [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

Ha csökkenő sorrendbe szeretnénk rendezni, használjuk a `reverse=True` paramétert:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
numbers.sort(reverse=True)
print(numbers)  # [9, 6, 5, 5, 4, 3, 2, 1, 1]
```

2. sorted() függvény
A sorted() függvény nem módosítja az eredeti listát, hanem egy új, rendezett listát ad vissza:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
s_numbers = sorted(numbers)
print(s_numbers)  # [1, 1, 2, 3, 4, 5, 5, 6, 9]
print(numbers)         # [3, 1, 4, 1, 5, 9, 2, 6, 5] (eredeti lista nem
változott)
```

Ha csökkenő sorrendbe szeretnénk rendezni a sorted() függvénnyel:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
s_reverse = sorted(numbers, reverse=True)
print(s_reverse)  # [9, 6, 5, 5, 4, 3, 2, 1, 1]
```

3. Rendezés kulcs alapján (key paraméter)
A sort() és sorted() módszerek egy key paramétert is elfogadnak, amely alapján rendezhetjük a listát. Például, ha szavakat akarunk hosszúságuk szerint rendezni:

```
words = ["apple", "banana", "cherry", "date", "fig"]
words.sort(key=len)
print(words)  # ['fig', 'date', 'apple', 'banana', 'cherry']
```

Ebben az esetben a key=len beállítás arra kényszeríti a Python-t, hogy minden elemet rendezzen a hosszuk alapján.

Ezek a példák bemutatják a Pythonban rendelkezésre álló különböző lehetőségeket listák rendezésére, attól függően, hogy módosítani szeretnénk az eredeti listát vagy egy újat szeretnénk létrehozni, és hogy növekvő vagy csökkenő sorrendben rendeznénk az elemeket.

Lista absztrakciója (List Comprehension)

A *lista absztrakciója* – a listák létrehozásának tömör módja. Ez egy hatékony és kompakt módja annak, hogy létrehozzunk egy új listát egy meglévő iterálható objektum alapján. Egy Python listakifejezés (list comprehension) példája, amelyben minden elem 2-es hatványra van emelve:

```
numbers = [number * number for number in range(1, 6)]
```

A range(1, 6) segítségével létrehozunk egy iterálható objektumot, amely tartalmazza az egész számokat 1-től 5-ig: [1, 2, 3, 4, 5]. A listakifejezés (`[number * number for number in range(1, 6)]`) egy új listát hoz létre, ahol minden elemet a number * number kifejezés határoz meg, ahol number változik minden iterációban a range(1, 6) szerint.
A listakifejezés működése:
1. *Iteráció a range(1, 6)-on:* A number változó minden iterációban egy új értéket kap a range(1, 6) értékei közül (1-től 5-ig).
2. *Elemek létrehozása:* A number * number kifejezés határozza meg minden iterációban a listaelem értékét. Tehát az első iterációban 1 * 1 lesz, a másodikban 2 * 2, és így tovább.
3. *Új lista létrehozása:* A listakifejezés egy új listát hoz létre az így kapott értékekből, tehát numbers a [1, 4, 9, 16, 25] listával fog rendelkezni.
Miután végrehajtottuk a fenti kódot, a numbers változó tartalma a következő lesz:

numbers = [1, 4, 9, 16, 25]

Ez a lista az egész számok négyzetének értékeit tartalmazza 1-től 5-ig. Ugyan azt az eredményt lehet igy is kapni:

```
numbers = []
for x in range(1, 6):
    numbers.append(x * x)
```

Összefoglaló. A listák fontosabb metódusai

append() Egy elemet hozzáfűz a lista végéhez.
extend() Egy lista elemeit ad hozzá egy másik lista végéhez.
insert() Beszúr egy elemet a megadott indexbe.
remove() Eltávolít egy elemet a megadott indexről.
pop() Visszaadja és eltávolítja a megadott indexen lévő elemet.
clear() Az összes elemet eltávolítja a listából.
index() A megadott elem indexét adja vissza.
count() A lista megadott elemeinek számát adja vissza.
sort() Növekvő/csökkenő sorrendbe rendezi a listát.
reverse() Visszaadja a listát fordított sorrendben ("feltekeri" a sorozatot).
copy() A lista egy másolatát adja vissza.

7. Tuple (кортеж)

A Pythonban a ***tuple*** adattípus olyan, mint egy lista. A kettő között az a különbség, hogy a **tuple elemeit nem lehet módosítani**, miután értékeket adtunk hozzájuk, míg egy lista elemeit módosíthatjuk.

Tuple létrehozása

A tuple egy olyan adattípus Pythonban, amely hasonló a listához, de számos fontos különbséggel rendelkezik. A tuple-k gyakran használatosak, ha olyan adatokat szeretnénk tárolni, amelyek nem változnak, vagy ha biztosak akarunk lenni abban, hogy az adatok véletlenül sem módosulnak a program futása közben.

Tuple létrehozása különböző módszerekkel
1. Szögletes zárójelek
A leggyakoribb módja a tuple létrehozásának a szögletes zárójelek használata:

```
my_tuple = (1, 2, 3, 4, 5)
```

2. tuple() konstruktor
A tuple() konstruktorral is létrehozhatunk tuple-t, például egy listából vagy más iterálható objektumból:

```
my_list = [1, 2, 3, 4, 5]
my_tuple = tuple(my_list)
```

Ez a kód my_list listát alakítja át tuple-vé, így my_tuple tartalma ugyanaz lesz, mint my_list tartalma, de immár tuple formátumban.

Egyetlen elemű tuple létrehozása
Ha egyetlen elemű tuple-t szeretnénk létrehozni, akkor figyelembe kell venni, hogy egy elemű tuple-t létrehozni szükséges egy vesszővel:

```
single_tuple = (42,)  # Egyetlen elemű tuple
```

Példák:

```
my_tuple = ()
print(my_tuple)
my_tuple = (1, 2, 3)
print(my_tuple)
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
# beágyazott tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
 print(my_tuple)
```

Eredmény:

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

Zárójelek használata nélkül is létrehozhatunk tuplet:

```
my_tuple = 1, 2, 3
my_tuple = 1, "Hello", 3.4
var1 = ("hello"); print(type(var1))
var2 = ("hello",); print(type(var2))
 var3 = "hello", ; print(type(var3))
```

Eredmény:

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

Tuple elemeinek elérése

A tuple elemeinek elérése Pythonban hasonló módon történik, mint a listák esetében. A tuple elemét indexeléssel érhetjük el, és az indexek 0-tól kezdődnek. Néhány példa a tuple elemeinek elérésére és használatára.

```
my_tuple = (10, 20, 30, 40, 50)
# Egyes elemek elérése index alapján
first_element = my_tuple[0]
third_element = my_tuple[2]
last_element = my_tuple[-1]  # Negatív index a végéről számolva
print(first_element)   # 10
print(third_element)   # 30
print(last_element)    # 50
```

Negatív index: Ha negatív számot használunk indexként (-1, -2, stb.), akkor az elemeket a tuple végétől számolva érjük el. Például -1 az utolsó elemet jelenti.

```
letters = (’p’, ’r’, ’o’, ’g’, ’r’, ’a’, ’m’, ’i’, ’z’)
print(letters[-1])   #  ’z’
 print(letters[-3])   #  ’m’
```

Több elem elérése egyidejűleg (slice használata):

```
subset = my_tuple[1:4]  # Második, harmadik és negyedik elem
print(subset)   # (20, 30, 40)
```

Slice (szeletelés): A slice segítségével több elemet is elérhetünk egyszerre a tuple-ben. A slice két index közötti elemeket tartalmazza, az első index a tartomány kezdetét, a második pedig a tartomány végét jelenti (de nem tartalmazza).

```
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[2:4])
print(my_tuple[:-7])
print(my_tuple[7:])
 print(my_tuple[:])
```

Eredmény:

```
('o', 'g')
('p', 'r')
('i', 'z')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Tuple bejárása ciklussal

```
for elem  in my_tuple:
    print(elem)
```

Ezek a példák bemutatják, hogyan lehet tuple-ben tárolt elemeket elérni és használni Pythonban. A tuple-k különösen hasznosak, ha olyan adatokat tárolunk, amelyek nem változnak meg a program futása során (pl. koordináták, konstansok stb.).

A tuple nem módosíthatók. Ez azt jelenti, hogy miután létrehoztuk őket, nem tudjuk megváltoztatni az elemeiket. Ezért a tuple-k nem rendelkeznek olyan sok metódussal, mint a listák. Az alábbiakban leirjuk a tuple fontosabb beépített metódusait.

1. count() metódus:
A count() metódus számolja meg, hogy egy adott elem hányszor fordul elő a tuple-ben.

```
my_tuple = (1, 2, 2, 3, 2, 4, 5)
count_two = my_tuple.count(2)
print(count_two)  # Output: 3
```

2. index() metódus:
Az index() metódus visszaadja az adott elem első előfordulásának indexét a tuple-ben. Ha az elem nem található a tuple-ben, ValueError kivételt dob.

```
my_tuple = (1, 2, 3, 4, 5)
index_three = my_tuple.index(3)
print(index_three)  # Output: 2
```

A tuple használatának előnyei a listához képest
Mivel a tuple nagyon hasonlóak a listákhoz, hasonló helyzetekben használhatók. Van azonban néhány előnye a tuple használatának a listával szemben.

A tuplet általában különböző adattípusokhoz, míg a listákat hasonló adattípusú elemekhez használják.

Mivel a tuple megváltoztathatatlanok, a velük való iteráció gyorsabb, mint egy listán. Így az adatfeldolgozás sebessége kis mértékben növekszik.

Az állandó elemeket tartalmazó tuple szótár kulcsaként használhatók.

Ha olyan adatok vannak, amelyek nem változnak, akkor azok tuple való megvalósítása biztosítja, hogy változatlanok maradjanak.

8. Szótárak

Szótár: kulcs-érték párokból álló halmaz.

Egy szótár (dictionary) egy adatszerkezet, amely kulcs-érték párokat tárol. A kulcsoknak egyedinek kell lenniük és **hashelhető** típusúak (pl. sztring, szám), az értékek pedig bármilyen típusúak lehetnek (pl. szám, sztring, lista, másik szótár stb.).

Megjegyzés. A "hashelhető" (hashable) magyarázata.

A *hashelhető* kifejezés a Pythonban arra utal, hogy egy adott objektum (pl. változó vagy adatszerkezet) alkalmas-e arra, hogy **hash**-t generáljon belőle a hash függvény segítségével. Az objektumok hashelhetősége alapvető fontosságú a szótárak (dictionary-k) és más hash táblák hatékony működéséhez. Alapvető jellemzők a hashelhető objektumokról:
1. Nem módosíthatóság: Ez azt jelenti, hogy az objektum értéke nem változhat meg a létrehozását követően. Például számok, sztringek, tuple.
2. *Unikálisság*: A hashelhető objektumoknak egyedi hash értékeik kell lenniük, azaz különböző objektumoknak különböző hash értékük van.

3. *Hash metódus támogatása:* A hashelhető objektumoknak rendelkezniük kell a `__hash__()` metódussal, amely visszaadja a hash értéküket.

*Hashelhető objektumok***:**

- *Számok:* Integerek (int), lebegőpontos számok (float).
- *Stringek*: Szöveges karakterláncok.
- *Tuple*: Nem módosítható (immutable) sorozatok.

Nem hashelhető objektumok:

- *Listák:* Módosítható (mutable) sorozatok.
- *Szótárak:* Módosítható kulcs-érték párok.
- *Halmazok:* Módosítható egyedi elemek gyűjteményei.

Szótár főbb jellemzői
Kulcs-érték párok: Minden elem egy kulcs-érték pár, ahol a kulcs egyedi és meghatározza az elem elérését.
Nem rendezett: Szótárak nem tárolják az elemek sorrendjét, ezért nem tudunk rájuk index alapján hivatkozni.
Módosítható: A szótárak módosíthatók, tehát hozzáadhatunk, módosíthatunk vagy törölhetünk elemeket belőlük.

Szótár létrehozása

```
# Szótár létrehozása kulcs-érték párokkal
my_dict = {'apple': 5, 'banana': 10, 'cherry': 3}
```

Elemek elérése és módosítása
Elemek elérése a kulcs alapján

```
print(my_dict['banana'])  # Output: 10
```

Elem módosítása

```
my_dict['apple'] = 8
```

Új elem hozzáadása

```
my_dict['orange'] = 6
```

Szótár iterálása
Szótár bejárása

```
for key, value in my_dict.items():
       print(f'Key: {key}, Value: {value}')
```

Fontosabb alkalmazások

Adatkezelés: Szótárakat gyakran használják adatok tárolására és kezelésére, különösen

Gyors elérés: Hatékonyan lehet velük elérni és módosítani az adatokat kulcs alapján.

Konfigurációk: Alkalmazások konfigurációs beállításait vagy más dinamikusan változó adatokat is szótárakban tárolhatunk.

A szótárak sokoldalúsága és hatékonysága miatt széles körben használatosak Pythonban különböző alkalmazások fejlesztésében és adatkezelési feladatokban.

A szótár kiírása egy sorban

```
dict = {'a': 1, 'b': 2, 'c': 3}
print(dict) # Ez a kód kiírja a teljes szótár tartalmát egy sorban
```

get() metódus

A get() metódus a szótárakban használt beépített metódus, amely lehetővé teszi, hogy egy adott kulcshoz tartozó értéket lekérjünk anélkül, hogy hibát kapnánk, ha a kulcs nem létezik a szótárban. Ez a metódus gyakran hasznos, ha bizonytalanok vagyunk abban, hogy egy adott kulcs létezik-e a szótárban, és szeretnénk elkerülni a `KeyError` kivétel dobását.

```
my_dict = {'apple': 5, 'banana': 10, 'cherry': 3}
# get() metódus használata
apple_count = my_dict.get('apple')
print(apple_count)  # Output: 5
# Nem létező kulcs lekérése
orange_count = my_dict.get('orange')
print(orange_count)  # Output: None (nincs ilyen kulcs a szótárban)
```

A get() metódus visszaadja a megadott kulcshoz tartozó értéket, ha a kulcs létezik a szótárban. Ha a kulcs nem létezik a szótárban, a get() metódus `None` értéket ad vissza alapértelmezés szerint, nem dob `KeyError` kivételt. Opcionálisan megadhatunk egy második paramétert a get() metódusnak, ami az alapértelmezett értéket adja vissza, ha a kulcs nem található a szótárban.

```
# get() metódus alapértelmezett értékkel
orange_count = my_dict.get('orange', 0)
# Ha 'orange' kulcs nem létezik, 0 lesz az alapértelmezett érték
print(orange_count)  # Output: 0
```

Javasoljuk a get() metódus alkalmazását, mivel tartalmazza:

- *kulcsellenőrzést*: biztonságosan lekérhetjük egy kulcshoz tartozó értéket anélkül, hogy figyelnünk kellene a `KeyError` kivételre.

- *alapértelmezett értéket*: a get() metódus segítségével könnyedén megadhatunk egy alapértelmezett értéket, ha a kulcs nem létezik a szótárban.

items() metódus (gyakran alkalmazzák!)

Az items() metódus egy szótár összes kulcs-érték párosát tartalmazó iterátor objektumot ad vissza. Ez az iterátor lehetővé teszi, hogy bejárjuk a szótár összes kulcs-érték párosát.
Az items() metódus hasznos és gyakran használt metódus a Python szótárak esetében. Ez a metódus egy iterálható objektumot ad vissza, amely tartalmazza a szótár összes kulcs-érték párját tuple-ként. Ez lehetővé teszi számunkra, hogy könnyen és hatékonyan iteráljunk végig a szótár minden elemén, és egyszerre hozzáférjünk mind a kulcsokhoz, mind az ezekhez tartozó értékekhez.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key, value in my_dict.items():   # items() metódus használata
    print(key, value)
```

Kimenet:
a 1
b 2
c 3

Az items() metódus egy iterálható objektumot ad vissza, amely tuple-ként tartalmazza a szótár minden kulcs-érték párját. A for ciklusban a key, value változók segítségével kicsomagoljuk a tuple-öket, így egyszerre hozzáférünk a szótár kulcsaihoz (key) és értékeihez (value). A print(key, value) sor kinyomtatja a kulcsokat és az ezekhez tartozó értékeket sorban.

Hogyan használható még az items() metódus

1. *Elemzés és adatfeldolgozás:* Segít végig iterálni a szótár minden elemén és feldolgozni azokat.
2. *Elemek módosítása:* Bár a szótárak nem sorrendezettek, a items() metódus segítségével könnyen módosíthatjuk az elemeket.
3. *Adatbázis műveletek:* Az items() használata a szótárak között is elérhető. Amennyiben az adatbázis műveletekről van szó, az items() metódus gyakran használatos az adatbázis-szerű műveletek során, például *szűrésre, keresésre vagy adatfeldolgozásra*. Bár Pythonban nem a szótárak használata a legelterjedtebb adatbáziskezelő technológia, azonban a szótárak és az items() metódus segíthet az adatok hatékony kezelésében és feldolgozásában.

Példák adatbázis-szerű műveletekre items() metódussal.

```
users = {
       'user1': {'name': 'Alice', 'age': 30},
       'user2': {'name': 'Bob', 'age': 25},
       'user3': {'name': 'Charlie', 'age': 35}
   }
```

1. *Adatok szűrése és keresése:*
Keresés az adatbázisban (szótárban)

```
for user_id, user_info in users.items():
       if user_info['age'] > 30:
           print(f"User {user_id}: {user_info['name']} is older than 30 years")
```

Ebben a példában a users szótárt használjuk, hogy felhasználók adatait tároljuk. Az items() metódussal végig iterálunk a szótár összes kulcs-érték páron, és szűrünk azokra a felhasználókra, akik 30 évnél idősebbek.

2. *Adatok feldolgozása és elemzése:*

```
inventory = {
       'item1': {'name': 'Laptop', 'price': 1200},
       'item2': {'name': 'Mouse', 'price': 30},
       'item3': {'name': 'Keyboard', 'price': 80}
   }
```

Összes termék árának kiszámítása

```
total_price = sum(item_info['price'] for item_info in inventory.values())
print(f"Total price of all items: ${total_price}")
```

Ebben a példában az inventory szótárt használjuk, hogy különböző termékek adatait tároljuk. A values() metódus segítségével hozzáférünk a szótár összes értékéhez, majd items() metódussal iterálunk rajtuk, és kiszámítjuk az összes termék árát.

Az items() metódus alkalmazása segítséget nyújt a szótárak hatékony kezelésében, feldolgozásában és elemzésében, különösen nagyobb mennyiségű adat esetén. Az adatbázis-szerű műveletek gyakran a szótárakban tárolt adatok kezelését jelentik, amelyek különböző adatelemzési és üzleti logikákra is alkalmazhatók.

Példa. Osztályok adatai a szótárban
Először létrehozunk üres sótárakat a dict() vagy {} paranccsal.

```
osztaly_letszamok = {}
dict2 = dict()
osztaly_letszamok['1a'] = 23
osztaly_letszamok['2b'] = 35
osztaly_letszamok['3a'] = 25
osztaly_letszamok['1b'] = 19
print('Osztály létszámok:', osztaly_letszamok)
print('Osztályok, amik létszámát tudjuk:', osztaly_letszamok.keys())
print('A 3.a létszáma', osztaly_letszamok['3a'], 'fő')
# ha olyan elemre hivatkozunk, ami nem létezik, akkor kivételt kapunk.
print('Az 5.zs létszáma', osztaly_letszamok['5zs'], 'fő')
```

Ha nem vagyunk benne biztosak, hogy adott elem létezik, akkor a get() metódust használjuk. A get metódus kaphat egy alapértelmezett értéket is, amivel visszatérhet, hogyha nem létezik az adott elem.

```
print(f'Az 5.zs letszama {osztaly_letszamok.get("5zs")} fő')
print(f"Az 5.zs letszama {osztaly_letszamok.get('5zs', 0)} fő")
```

Az elemek vizsgálatára az in *operátort használhatjuk.*

```
if '1b' in osztaly_letszamok:  # if '1b' in ????????????????????????
      osztaly_letszamok.keys():
                print('Az 1b osztaly letszamat tudjuk!')
if 25 in osztaly_letszamok.values():  # ertekek kozott keres
    print('Van 25 fos osztaly!')
```

Szótár szétbontása tuple listává

```
print(osztaly_letszamok.items())
# [('3a', 25), ('2b', 35), ('1a', 23), ('1b', 19)]
```

Szótár bejárása for ciklussal

```
for kulcs, ertek in osztaly_letszamok.items():
    print('A(z)', kulcs, ' osztalyba', ertek, ' fő jár.')
for kulcs in osztaly_letszamok:
      x=osztaly_letszamok[kulcs]
      print(kulcs, '=>', x)
      print(kulcs, '=>', osztaly_letszamok[kulcs])
```

Szótár elemeinek törlése

```
osztaly_letszamok['7k'] = 50
print(osztaly_letszamok)
del osztaly_letszamok['7k']
print(osztaly_letszamok)
```

Példa. Adatok diákokról a szótárban

Ez a kód létrehoz egy *students_dict* szótárt, ahol a kulcsok a tanulói indexek (0, 1, 2, stb.), az értékek pedig az egyes tanulók adatait tartalmazó szótárak. Minden ilyen szótár három kulcsot tartalmaz: „név”, „születési_dátum” és „cím”.

```
students_list = [
    ["János", "2005-03-15", "Utca_1, 10"],
    ["Mária", "2006-07-20", "Utca_2, 5"],
    ["Péter", "2005-01-10", "Utca_1, 15"]
]
students_dict = {}
for index, student_info in enumerate(students_list):
    name, birthdate, address = student_info
    students_dict[index] = {"név": name, "születési_dátum": birthdate, "cim":
address}
print(students_dict)
```

A szótárban lévő tanulói adatok eléréséhez kulcsokat (tanulói indexeket) és mezőneveket (név, születési dátum, cím) használható. Például, igy:

```
# Adatok lekérése az első tanulóról (0. indexnél)
student_0 = students_dict[0]
print("Az első tanuló adatai:")
print("Név:", student_0["név"])
print("születési_dátum:", student_0["születési_dátum"])
print("cím:", student_0["cím "])
print()
```

```
# Adatok lekérése a második tanulóról (1. indexnél)
student_1 = students_dict[1]
print("Második tanuló adatai:")
print("Név:", student_1["név"])
print("születési_dátum:", student_1["születési_dátum"])
print("cím:", student_1["cím"])
```

```
# Új tanuló
new_student_info = ["Anna", "2007-11-25", "Utca_3, 20"]
# Új tanuló bevitele a szótárba
index_of_new_student = len(students_dict)  # Új tanuló indexe
students_dict[index_of_new_student] =
{"név": new_student_info[0],
"születési_dátum": new_student_info[1],
 "cím": new_student_info[2]}
print(students_dict)
```

```
# Tanuló törlése a szótárból (index=1)
del students_dict[1]
print(students_dict)
```

```
# Tanuló keresése:
# Név szerint
def find_student_by_name(name):
    for index, student_info in students_dict.items():
        if student_info["név"] == name:
            return index, student_info
    return None, None
```

```
# Tanuló neve, keresés
search_name = "Mária"
index, student_info = find_student_by_name(search_name)
if index is not None:
    print("Tanuló", search_name, "megvan találva:")
    print("Index:", index)
    print("Tanuló adatai:", student_info)
else:
    print("Tanuló", search_name, "nincs a szótárban.")
```

Összefoglaló. A szótárakkal való fontosabb műveletek

Elem hozzáadása és módosítása:

```
dict = {'a': 1, 'b': 2}
dict['c'] = 3  # Hozzáad egy új kulcs-érték párt
dict['a'] = 10  # Módosítja az 'a' kulcs értékét
```

Elem eltávolítása:

```
del dict['b']  # Törli a 'b' kulcs-érték párt
```

Érték lekérdezése kulcs alapján:

```
value = dict['a']  # Lekéri az 'a' kulcs értékét
```

Kulcsok lekérdezése:

```
keys = dict.keys()  # Visszaadja a szótár kulcsait
```

Értékek lekérdezése:

```
values = dict.values()  # Visszaadja a szótár értékeit
```

Kulcs-érték párok lekérdezése:

```
items = dict.items()  # Visszaadja a kulcs-érték párokat
```

Méret lekérdezése:

```
size = len(dict1)  # Visszaadja a szótár elemeinek számát
```

Keresés kulcs alapján:

```
if 'a' in dict:  # Ellenőrzi, hogy van-e 'a' kulcs a szótárban
    print('A kulcs megtalálható.')
```

9. Halmazok

A Pythonban lévő halmaz egyedi adatok halmaza. Egy halmaz elemei nem sokszorosíthatók. Egy tömb tetszőleges számú elemet tartalmazhat, és ezek különböző típusúak lehetnek (int, float, tuple, string stb.). De egy halmaznak nem lehetnek változtatható elemei, például listák, szótárak vagy egyéb halmazok. Egy halmaz létrehozásához az összes elemet kapcsos zárójelek közé kell tenni {}, vesszővel elválasztva.

Halmaz létrehozása

A Pythonban egy halmaz létrehozásához az összes elemet kapcsos zárójelek közé kell tenni {}, vesszővel elválasztva.

Egy lista tetszőleges számú elemet tartalmazhat, és ezek különböző típusúak lehetnek (lista, szótár, int, float, tuple, string stb.). De egy halmaznak *nem lehetnek változtatható elemei,* például listák, szótárak vagy egyéb halmazok.

```
student_id = {112, 114, 116, 118, 115}
print('Student ID:', student_id)
vowel_letters = {'a', 'e', 'i', 'o', 'u'}
print('Vowel Letters:', vowel_letters)
mixed_set = {'Hello', 101, -2, 'Bye'}
 print('Set of mixed data types:', mixed_set)
```

Eredmény:

```
Student ID: {112, 114, 115, 116, 118}
Vowel Letters: {'u', 'a', 'e', 'i', 'o'}
Set of mixed data types: {'Hello', 'Bye', 101, -2}
```

Megjegyzés: Amikor a kód végrehajtódik, előfordulhat, hogy az eredmény elemei eltérő sorrendben lesznek. Ez azért történik, mert a halmaznak nincs különösebb sorrendje.

Az üres kapcsos zárójelek üres szótárt hoznak létre a Pythonban. Üres halmaz létrehozásához pedig a **set() függvényt** kell használni argumentumok nélkül.

```
empty_set = set()
empty_dictionary = { }
print('Data type of empty_set:', type(empty_set))
print('Data type of empty_dictionary', type(empty_dictionary))
```

Eredmény:

```
Data type of empty_set: <class 'set'>
Data type of empty_dictionary <class 'dict'>
```

Nézzük meg, mi történik, ha duplikált elemeket próbálunk beletenni a halmazba.

```
numbers = {2, 4, 6, 6, 2, 8}
print(numbers) # {8, 2, 4, 6}
```

A halmazok változtathatók. Mivel azonban rendezetlenek, az indexelésnek nincs értelme. Egy halmaz elemét nem tudjuk elérni vagy módosítani indexeléssel vagy szeleteléssel. A beállított adattípus ezt nem támogatja.

Elem hozzáadása a halmazhoz

A Pythonban az **add() metódus** egy elem hozzáadására szolgál egy halmazhoz.

```
numbers = {21, 34, 54, 12}
print('Initial Set:',numbers)
numbers.add(32)
 print('Updated Set:', numbers)
```

```
Initial Set: {34, 12, 21, 54}
Updated Set: {32, 34, 12, 21, 54}
```

```
my_set = {1, 2, 3}
  my_set.add(4)
  print(my_set)  # Output: {1, 2, 3, 4}
```

Halmazok frissítése, elemek hozzáadása

A Pythonban az *update()* metódust használjuk egy másik halmaz vagy iterálható objektum (pl. lista, tuple) elemeinek hozzáadására a halmazhoz.

```
my_set = {1, 2, 3}
my_set.update([4, 5])
print(my_set)  # Output: {1, 2, 3, 4, 5}
```

```
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple']
companies.update(tech_companies)
 print(companies) # {'Lacoste', 'Ralph Lauren', 'apple', 'google'}
```

Elem eltávolítása a halmazból

A Pythonban a *discard()* metódus a megadott elem eltávolítására szolgál egy halmazból. Példa:

```
languages = {'Swift', 'Java', 'Python'}
print('Initial Set:',languages)
removedValue = languages.discard('Java')
 print('Set after remove():', languages)
```

```
Initial Set: {'Python', 'Swift', 'Java'}
Set after remove(): {'Python', 'Swift'}
```

Halmaz ürítése

A *clear()* metódust használjuk a halmaz elemeinek teljes eltávolítására.

```
my_set = {1, 2, 3}
my_set.clear()
print(my_set)  # Output: set()
```

Iteráció halmazon keresztül

```
fruits = {"Apple", "Peach", "Mango"}
for fruit in fruits:
    print(fruit)
```

```
Mango
Peach
Apple
```

Halmaz elemeinek száma

A Pythonban a **len() metódust** használják a halmazban lévő elemek számának meghatározására. Példa:

```
even_numbers = {2,4,6,8}
print('Set:',even_numbers)
print('Total Elements:', len(even_numbers))
```

```
Set: {8, 2, 4, 6}
Total Elements: 4
```

Halmaz elemeinek módosítása

A halmazok alapvetően nem módosíthatók, azaz nem lehet közvetlenül egy elemüket módosítani, mint például listák esetén. Azonban a módosítás mesterségesen végrehajtható: a *Megjegyzések*:

- A halmazok elemeinek módosítása során fontos figyelembe venni, hogy a halmazok nem tartalmazhatnak duplikált elemeket, így ha egy elemet hozzáadunk, amely már szerepel a halmazban, az nem lesz újra hozzáadva.
- Ha egy halmazt szeretnénk teljesen cserélni egy másik halmazzal, akkor érdemes egyszerűen újra létrehozni a halmazt, vagy használni az update() metódust a szükséges elemekkel.
- A halmazok hatékonyak az egyedi elemek gyűjtésére és műveletekre (pl. unió, metszet, különbség stb.), de mivel nem rendelkeznek indexeléssel, nem tudunk közvetlenül egy adott elemet halmazon belül módosítani.

Halmazműveletek

A Python különféle beépített módszereket biztosít a halmazokon végzett matematikai műveletek végrehajtásához, például unió, metszés, különbség és szimmetrikus különbség.

Halmazok uniója

Két A és B halmaz uniója magában foglalja az A és B halmaz összes elemét.
A | operátor vagy a *union()* metódus hajtja végre az unió műveletet.

```
A = {1, 3, 5}
B = {0, 2, 4}
print('Union using |:', A | B)
print('Union using union():', A.union(B))
```

Eredmény:

```
Union using |: {0, 1, 2, 3, 4, 5}
Union using union(): {0, 1, 2, 3, 4, 5}
```

Halmazok metszete

Két A és B halmaz metszete tartalmazza az A és B halmazok közös elemeit. A Pythonban a *& operátor* vagy az *intersection()* metódus a halmazok metszésponti műveletének végrehajtására szolgál.

```
A = {1, 3, 5}
B = {1, 2, 3}
print('Intersection using &:', A & B)
 print('Intersection using intersection():', A.intersection(B))
```

```
Intersection using &: {1, 3}
Intersection using intersection(): {1, 3}
```

Halmazok különbsége

A két A és B halmaz közötti különbség tartalmazza az A halmaz olyan elemeit, amelyek nem szerepelnek a B halmazban. A Pythonban a - *operátor* vagy a *differencia()* metódus a különbség művelet végrehajtására szolgál két halmaz között.

```
A = {2, 3, 5}
B = {1, 2, 6}
print('Difference using -:', A - B)
print('Difference using difference():', A.difference(B))
```

```
Difference using difference(): {3, 5}
```

Halmazok szimmetrikus különbsége

Két A és B halmaz szimmetrikus különbsége A és B összes elemét tartalmazza közös elemek nélkül. A Pythonban **a ^ operátor vagy a symmetric_difference() metódus** két halmaz közötti szimmetrikus különbségi művelet végrehajtására szolgál. :

```
A = {2, 3, 5}
B = {1, 2, 6}
print('using ^:', A ^ B)
print('using symmetric_difference():', A.symmetric_difference(B))
```

```
using ^: {1, 3, 5, 6}
using symmetric_difference(): {1, 3, 5, 6}
```

Ellenőrzés: egyenlő-e két halmaz

A Pythonban az == operátort használják annak tesztelésére, hogy két halmaz egyenlő-e.

```
A = {1, 3, 5}
B = {3, 5, 1}
if A == B:
    print('Set A and Set B are equal')
else:
     print('Set A and Set B are not equal')
```

```
Set A and Set B are equal
```

Összefoglaló. Halmaz metódusai

add() - Elemet ad a halmazhoz.
all() - Igaz értéket ad vissza, ha a halmaz minden eleme igaz (vagy ha a halmaz üres).
any() - igazat ad vissza, ha a halmaz legalább egy eleme igaz. Ha a halmaz üres, False-t ad vissza.
clear() - Eltávolítja az összes elemet a halmazból.
copy() - A halmaz másolatát adja vissza.
differencia() - Két vagy több halmaz különbségét adja vissza új halmazként.
different_update() - Eltávolítja egy másik halmaz összes elemét az aktuális halmazból.
discard() - Eltávolít egy elemet a halmazból, ha annak része.
enumerate() - Egy iteráló objektumot ad vissza, amely párként tartalmazza a halmaz összes elemének indexét és értékét.
intersection() - Két halmaz metszetét adja vissza új halmazként.
intersection_update() - Egy halmazt saját maga és egy másik halmaz metszéspontja szerint frissít.
isdisjoint() - Igazat ad vissza, ha a két halmaz nem metszi egymást.
issubset() - Igaz értékkel tér vissza, ha egy másik halmaz tartalmazza az aktuális halmazt.
issuperset() - Igaz értéket ad vissza, ha az aktuális halmaz tartalmaz egy másik halmazt.
len() - A halmaz hosszát (elemek számát) adja vissza.
max() - A halmaz legnagyobb elemét adja vissza.
min() - A halmaz legkisebb elemét adja vissza.
pop() - Eltávolítja és visszaadja a halmaz tetszőleges elemét.
remove() - Eltávolít egy elemet a halmazból. Ha az elem nem tagja a halmaznak, egy KeyError jelenik meg.
sorted() - A halmaz elemeinek új rendezett listáját adja eredményül (de nem rendezi).
sum() - Egy halmaz összes elemének összegét adja eredményül.
symmetric_difference() - Két halmaz szimmetrikus különbségét adja vissza új halmazként.
symmetric_difference_update() - Frissíti a halmazt a saját és a másik szimmetrikus különbségével.
union() - A halmazok unióját egy új halmazba adja vissza.
update() - Frissít egy halmazt az én és mások egyesülésével

9. Generátorok

Generátor - olyan speciális típusú iterátor, amely dinamikusan állítja elő az értékeiket, ahogy azokra szükség van. Generátorral hatékonyan kezelhetnek nagy adatmennyiségeket vagy

végtelen sorozatokat is. A generátorokat leggyakrabban a `yield` kulcsszóval definiálják, ami visszaad egy értéket, majd felfüggeszti az állapotot, de megtartja a belső állapotot, hogy folytathassa a következő híváskor.

```
def my_generator():
    yield 1
    yield 2
    yield 3
gen = my_generator()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

A `next()` függvény a Pythonban az iterátorok működését támogató beépített függvény. A `next()` függvényt az iterátor elemeinek egyenkénti lekérdezésére használjuk. Ha meghívjuk a `next()` függvényt egy iterátoron, akkor visszaadja az iterátor következő elemét.

Az **iter()** függvény a Pythonban egy iterátor létrehozására szolgál egy iterálható objektumból. Az iterátorok segítségével lehetőség van az objektum elemeinek soronkénti feldolgozására.
Az iter() függvény használata:
iter(iterálható_objektum)

Példa.

```
lista = [1, 2, 3, 4, 5]
iterátor = iter(lista)
print(next(iterátor))  # 1
print(next(iterátor))  # 2
print(next(iterátor))  # 3
print(next(iterátor))  # 4
print(next(iterátor))  # 5
```

Ha az iterátor végére érünk, és nincs több elem, a `next()` függvény `StopIteration` kivételt dob. Ezt kezelhetjük a
`try...except` blokk
segítségével vagy a `next()` függvény opcionális második argumentumaként megadott alapértelmezett értékkel:

```
my_list = [1, 2, 3]
my_iter = iter(my_list)
while True:
      try:
            print(next(my_iter))
      except StopIteration:
            break
```

A *for ciklussal* is lehet végrehajtani:

```
my_list = [1, 2, 3]
my_iter = iter(my_list)
for item in my_iter:
    print(item)
```

Egy végtelen generátor létrehozásához egy végtelen ciklust kell használni, amely soha nem áll meg. Ehhez a `while True` ciklust használhatjuk, és minden iterációban `yield`-et

használunk az aktuális érték visszaadására. Egy példa egy végtelen generátorra, amely egymás után adja vissza az egész számokat:

```
def infinite_generator():
    i = 0
    while True:
        yield i
        i += 1
gen = infinite_generator()
for k in range(5):      # 0,1,2,3,4
    print(next(gen))  # Kiírja az első öt egész számot
```

Ebben a példában az _ változót használtuk a ciklusváltozóként, mivel a ciklusban nem használjuk a tényleges értékét. A programozók ezt a módszert gyakran alkalmazzák, hogy kifejezzék, hogy a változó értéke nem releváns a program szempontjából. Az **_ (aláhúzás) változó** gyakran használt konvenció arra, hogy a programozó jelezze, hogy a változó értéke nem fontos vagy nem használt. Általában akkor használják, amikor a változó értéke valamilyen módon mellékes vagy csak egy ideiglenes eredmény.

Vissza

11. Függvények

A Pythonban a függvények alapvető építő kövei a programozásnak, és lehetőséget biztosítanak kódrészletek újra felhasználására és strukturálására. A függvények segítségével csoportosíthatjuk a kódot, logikailag elkülöníthetjük az egyes feladatokat, és többször is használhatjuk azokat anélkül, hogy ismételnénk a kódot. Egy összetett feladat kisebb részekre bontása megkönnyíti programunk megértését és újra felhasználását. A Pythonban kétféle függvény létezik: **beépített** és **egyedi.** Néhány alapvető információ és példa a függvények használatára Pythonban:

Függvény definiálása:
A függvények definiálása a `def` kulcsszóval történik, majd a függvény neve következik, és zárójelek között felsoroljuk a paramétereket, ha vannak. A függvény törzse behúzott (indentált) blokkban található.

```
def greet(name):
    print(f"Hello, {name}!")
```

Függvény meghívása

```
greet("Alice")  # Output: Hello, Alice!
```

Paraméterek és visszatérési érték:
Paraméterek: A függvények paramétereket vehetnek át, amelyeket a függvény hívása során megadunk. A fenti példában a greet függvény egy name nevű paramétert vár.
Visszatérési érték: A return utasítással határozhatjuk meg, hogy a függvény milyen értéket ad vissza. Ha a return utasítás hiányzik, a függvény automatikusan None értéket ad vissza.

```
def add_numbers(a, b):
    # Ez a függvény két számot ad össze
    return a + b

result = add_numbers(3, 5)
print(result)  # Output: 8
```

A függvényt igy is lehet meghívni:

```
add_numbers(num1 = 5, num2 = 4)
```

A Pythonban ezt megnevezett argumentumnak nevezik. A fenti kódsor egyenértékű a következő kódsorral: add_numbers(5, 4)

Alapértelmezett értékek:
A függvények paramétereinek alapértelmezett értéket is adhatunk, így ha a függvényt hívjuk, és nem adunk meg értéket az adott paraméter számára, az alapértelmezett érték lesz használva.

```
def greet(name="Guest"):
    # Ez a függvény üdvözli a megadott nevet, vagy alapértelmezett értékként a
'Guest' szót használja.
    print(f"Hello, {name}!")
greet()  # Output: Hello, Guest!
greet("Bob")  # Output: Hello, Bob!
```

Több visszatérési érték:

A Pythonban a függvények több értéket is visszaadhatnak egy tuple formájában. A tuple külön elemei azonban különválaszthatók, mint különálló visszatérési értékek.

```
def get_circle_info(radius):
    """Ez a függvény kiszámolja a kör területét és kerületét."""
    area = 3.14 * radius * radius
    circumference = 2 * 3.14 * radius
    return area, circumference

area, circumference = get_circle_info(5)
print(f"Area: {area}, Circumference: {circumference}")
```

Lambda függvények:

A lambda függvények kifejezések formájában írhatók le, amelyek gyakran egyszerű, rövid műveleteket valósítanak meg. Ezeket gyakran használják névtelen függvényekként vagy a map(), filter() és reduce() függvényekkel együtt.

```
multiply = lambda x, y: x * y
print(multiply(3, 4))  # Output: 12
```

Függvények modulokban:

A Pythonban a függvényeket modulokba (fájlokba) szervezzük, és ezeket importálhatjuk más fájlokból vagy programokból. Például:

```
# file: mymodule.py
def say_hello():
    print("Hello from my module!")

# file: main.py
import mymodule
mymodule.say_hello()  # Output: Hello from my module!
```

A függvények különböző típusú feladatok megoldására szolgálnak Pythonban, és alapvető szerepet töltenek be a programozásban, különösen a kódstruktúra és az újra felhasználhatóság szempontjából.

Függvénytípusok Pythonban

Kétféle függvény létezik:

Python Standard Library függvényei beépített függvények, amelyek használhatók.

Egyedi függvények – igényeink alapján egyedi függvényeket is létrehozhatunk.

A Python Standard Library (Python Standard Könyvtár) az a nagyobb méretű Python csomagok gyűjteménye, amelyeket a Python programozási nyelvhez hivatalosan mellékelték. Ezek a csomagok számos függvényeket és eszközt kínálnak a különböző feladatokhoz és problémákhoz, így sokkal hatékonyabban lehet velük dolgozni, mint ha mindent nulláról kellene megírni.

A *Python Standard Library* tartalmazza többek között a következőképpen csoportosított csomagokat:

Beépített függvények: Ezek a beépített függvények és típusok, amelyek alapvető funkcionalitást biztosítanak, mint például a **print()**, **len()**, **range()**, stb.

Beépített típusok: Olyan típusok, mint a listák (**list**), tuple-ök (**tuple**), sztringek (**str**), szótárak (**dict**), halmazok (**set**), és mások, amelyek különböző adatstruktúrákat biztosítanak.
Fájlkezelés: Fájlok olvasására és írására szolgáló csomagok, például a **os**, **os.path**, **shutil**, **glob**, **tempfile** stb.
Matematikai és tudományos függvények: Számos csomag található itt, mint például a **math**, **random**, **statistics**, **datetime**, **calendar**, **cmath**, **decimal**, **fractions** stb.
Szövegkezelés: Csaknem minden olyan dolog, ami a szöveges adatok feldolgozására szolgál, mint például a **re** (reguláris kifejezések), **string**, **textwrap**, **unicodedata**, **difflib**, **codecs**, **io** stb.
Hálózati és kommunikációs függvények: Ide tartoznak a **socket**, **urllib**, **http**, **ftplib**, **smtp**, **telnetlib**, **ipaddress** stb.
Adatbázisok: Adatbázis-kezelésre szolgáló csomagok, mint például a **sqlite3**, **dbm**, **mysql.connector**, **psycopg2** stb.
Tesztelés és debuggolás: Teszteléshez és hibakereséshez használt eszközök, például a **unittest**, **doctest**, **pdb** stb.
Tömörítés és archiválás: Az adatok tömörítésére és archiválására szolgáló csomagok, mint például a **zipfile**, **gzip**, **tarfile** stb.
Multimédia: Kép-, hang- és videófájlok kezelésére szolgáló csomagok, mint például a **PIL**, **soundfile**, **wave** stb.

A Python hivatalos dokumentációjában részletesebb információkat lehet találni a Standard Library-ról és a benne található modulokról.

Python Standard Library függvényei

A Pythonban a Standard Library függvényei olyan beépített függvények, amelyeket közvetlenül használhat a kódban. Példa:

print() — megjeleníti a szöveget a képernyőn;
sqrt() — egy szám négyzetgyökét adja vissza;
pow() — végrehajtja a szám hatványra emelésének műveletét.

Ezek a könyvtári függvények a modulon belül vannak definiálva, ezért használatukhoz csatlakoztatnia kell a modult a programhoz. Például az sqrt() függvény a **math** matematikai modulon belül van definiálva.
Vegyünk egy példát a Python könyvtárfüggvényének használatára:

```
import math
square_root = math.sqrt(4)
print("Square Root of 4 is",square_root)
power = pow(2, 3)
print("2 to the power 3 is",power)
```

Eredmény:
Square Root of 4 is 2.0
2 to the power 3 is 8
Itt használtuk:

math.sqrt(4) — 4 négyzetgyökének kiszámítása;
pow(2, 3) – a 2-es szám 3, azaz 2^3 hatványra emelése.

Mivel az *sqrt()* függvény a *math* modulon belül van definiálva, azt beillesztetünk a programunkba:

import math

Függvények alkalmazásának előnyei

1. előny: A kód újra felhasználása. Ugyanazt a függvényt többször is használhatjuk a programunkban, így kódunk rövidebb lesz. Példa:

```
def get_square(num):
    return num * num
for i in [1,2,3]:
    result = get_square(i)
    print('Square of',i, '=',result)
```

Eredmény:
Square of 1 = 1
Square of 2 = 4
Square of 3 = 9

Itt létrehoztunk egy get_square() függvényt egy szám négyzetének kiszámításához. Ezután írtunk egy for ciklust az 1-től 3-ig tartó számok négyzetének kiszámításához. Ugyanezt a függvényt a szám négyzetének kiszámításához többször használjuk a ciklusban.

2. előny: Kód olvashatósága. A függvények segítenek a kód kisebb részekre bontásában, hogy a program olvasható és könnyen érthető legyen.

Vissza

12. Kivételek (Exceptions) a Pythonban

A kivétel egy adattípus a Pythonban. Kivételekre azért van szükség, hogy tájékoztassák a programozót a hibákról. A legegyszerűbb példa: a nullával való osztás.

z = 100/0
hibás művelet.
Traceback (most recent call last):
File “”, line 1, in
ZeroDivisionError: division by zero

ZeroDivisionError a kivétel neve. A Python azt a sorszámot is jelenti, ahol a kivétel előfordult. Más kivételek is lehetségesek. Például amikor karakterláncot és számot próbálunk hozzáadni:

z = 2 + '1',
TypeError kivétel fog bekövetkezni
Traceback (most recent call last):
File “”, line 1, in
TypeError: unsupported operand type(s) for +: 'int' and 'str'

A Python beépített kivételeinek hierarchiája

• **BaseException** – az alap kivétel, amelyből az összes többi származik.
• **SystemExit** - egy kivétel, amelyet a sys.exit függvény dob ki a programból való kilépéskor.
• **KeyboardInterrupt** – akkor jön létre, amikor a felhasználó megszakítja a programot (általában a Ctrl+C billentyűkombinációval).
• **GeneratorExit** – a generátor objektum close metódusának meghívásakor jön létre.
• **Exception** - itt befejeződtek a *rendszerkivételek* (jobb, ha nem érintjük őket), és kezdődnek a szokásosak, amelyekkel dolgozunk.
• **StopIteration** – A beépített *next* függvény bocsátja ki, ha nincs több elem az iterátorban.
• **ArithmeticError** – aritmetikai hiba.
• **FloatingPointError** – akkor jelenik meg, ha egy lebegőpontos művelet meghiúsul. A gyakorlatban ritkán fordul elő.
• **OverflowError** – Akkor fordul elő, ha egy aritmetikai művelet eredménye túl nagy ahhoz, hogy ábrázolja. Nem jelenik meg normál egész számokkal való munka közben (mivel a Python támogatja a hosszú számokat), de előfordulhat más esetekben.
• **ZeroDivisionError** – osztás nullával.
• **AssertionError** – Az *assert* utasítás kifejezése hamis.

***Az assert* utasítás leírása**

Az assert utasítás a Pythonban hibakeresésre és tesztek írására használatos. Ez egy olyan állítás, amely megvizsgálja, hogy egy feltétel igaz-e. Ha a feltétel igaz, a program végrehajtása folytatódik. Ha a feltétel hamis, az assert utasítás AssertionError kivételt vet fel, ami a programvégrehajtás leállását okozza.

Példa. A nullával való osztás nem lehetséges, és ZeroDivisionError kivételt vet fel. Ennek a helyzetnek a megelőzése érdekében az **assert** utasítással lehet ellenőrizni az osztót az osztási művelet végrehajtása előtt.

```
def oszt(x, y):
     assert y != 0, "Nem osztható nullával"
     return  x/v
```

Ebben az esetben, ha y értéke nulla, az osztás függvény végrehajtása AssertionError üzenetet ad, és a következő üzenettel jelenik meg: "Nem osztható nullával." Az assert utasítás nagyon hasznos a programok hibakeresésekor, mert lehetővé teszi a problémák gyors észlelését.

• **AttributeError** – Az objektum nem rendelkezik a megadott attribútummal (értékkel vagy metódussal).
• **BufferError –** A pufferrel társított művelet nem hajtható végre.
• **EOFError** – a függvény a fájl végét találta, és nem tudta elolvasni.
• **ImportError** – a modul vagy attribútumának importálása nem sikerült.
• **LookupError** – érvénytelen index vagy kulcs.
• **IndexError** – Az index nincs az elemek tartományában.
• **KeyError –** nem létező kulcs (szótárban, halmazban vagy más objektumban).
• **Memory Error** – nincs elég memória.
• **NameError** – azonos nevű változó nem található.
• **UnboundLocalError** – Egy függvényben hivatkoztak egy helyi változóra, de a változót korábban nem határozták meg.
• **OSError** – rendszerrel kapcsolatos hiba.
• **FileExistsError** – kísérlet egy már létező fájl vagy könyvtár létrehozására.
• **FileNotFoundError** – a fájl vagy könyvtár nem létezik.
• **InterruptedError** – a rendszerhívást megszakította egy bejövő jel.
• **IsADirectoryError** – egy fájlt vártak, de az egy könyvtár volt.
• **NotADirectoryError** – egy könyvtárat vártak, de az egy fájl.
• **PermissionError** – elégtelen hozzáférési jogosultságok.
• **TimeoutError** – az időtúllépés lejárt.
• **RuntimeError** – Akkor fordul elő, ha a kivétel nem fér bele a többi kategóriába.
• **NotImplementedError** – Akkor fordul elő, ha egy osztály absztrakt metódusai felülbírálást igényelnek a gyermekosztályokban.
• **SyntaxError** – szintaktikai hiba.
• **IndentationError -** hibás behúzás (відступ).
• **TabError** – tabulátorok és szóközök keverése a behúzásban.
• **SystemError** – belső hiba.
• **Típushiba** – a művelet nem megfelelő típusú objektumra lett alkalmazva.
• **ValueError** – a függvény megfelelő típusú argumentumot kap, de helytelen értéket.
• **UnicodeError** – a karakterláncokban történő unicode kódolással/dekódolással kapcsolatos hiba.
• **UnicodeEncodeError** – Unicode kódolással kapcsolatos kivétel.
• **UnicodeDecodeError –** Unicode dekódolással kapcsolatos kivétel.
• **UnicodeTranslateError** – Unicode fordítással kapcsolatos kivétel.
• **Warning** — Alaposztály a figyelmeztető kivételekhez. Ez a kivételcsalád a figyelmeztetések különböző kategóriáit képviseli.
• **BYTESWARNING** — Figyelmeztetések a bájtokkal végzett munka során felmerülő lehetséges problémákkal kapcsolatban. A figyelmeztetéseknek ezt a kategóriáját a bájtokkal (bájtokkal és bájttömbökkel) végzett munka során fellépő esetleges hibák esetén használják.

Try -Except

Most, hogy tudjuk, mikor és milyen körülmények között fordulhatnak elő kivételek, **kezelhetjük azokat**. A kivételek kezeléséhez használjuk a **try — except** konstrukciót.

```
try:
      k=1/0
except ZeroDivisionError:
      k=0
print(k) # 0
```

A **try** blokkban olyan utasításokat hajtunk végre, amelyek kivételt okozhatnak, egy kivétel blokkban pedig elkapjuk őket. Ebben az esetben magát a kivételt és leszármazottait is elkapják. Például az **ArithmeticError** elkapásakor a **FloatingPointError, OverflowError** és a **ZeroDivisionError** is elkapjuk.

```
try:
      k=1/0
except ArithmeticError:
      k=0
print(k)   # 0
```

Lehetséges argumentumok nélküli kivételes utasítás is, amely mindent képes elfogadni (billentyűzet megszakítás, rendszerkimenet stb.).

A problémánkkal kapcsolatos további két állítás a **finally és az else**. **Finally** minden esetben végrehajt egy utasításblokkot, függetlenül attól, hogy volt-e kivétel, vagy sem (akkor alkalmazható, ha feltétlenül meg kell tennie valamit, például be kell zárnia egy fájlt). Az **else** utasítás végrehajtásra kerül, ha nincs kivétel.

```
f=open(’1.txt;)
ints=[]
try:
      for line in f:
            ints.append(int(line))
except ValueError:
      print(’ez nem szám. Kilépés.’)
except Exception:
      print(’Mi ez?)
else:
      print('Minden rendben van.')
finally:
      f.close()
      print(' Bezártam a fájlt.')
```

A program ebben a sorrendben lesz végrehajtva:
try, except csoport, else, finally.

raise – ‘saját’ kivétel

A Python "raise" operátora egy hatékony eszköz a program folyamjának szabályozására hibák és kivételek esetén. A Pythonban a *raise* operátort arra használják, hogy kifejezetten kivételt emeljenek ki a kódban. Ez lehetővé teszi a programozó számára, hogy kivételeket hozzon létre és dobjon ki bizonyos helyzetekben, amikor a program hibákba vagy váratlan körülményekbe ütközik.

- A `raise` operátor használható beépített kivételekkel (pl. ValueError, TypeError, KeyError stb.) vagy egyéni kivételekkel, amelyeket a programozó hoz létre.
- Az except blokkban megadhatjuk, hogy melyik kivétel kezelését várjuk.
- A kivételek lehetővé teszik a programozó számára, hogy rugalmasabban kezelje a programban előforduló hibákat és váratlan helyzeteket.

#. Példa egy beépített kivétel hívására

```
def divide_numbers(a, b):
    if b == 0:
        raise ZeroDivisionError("A nullával való osztás nem lehetséges")
    return a/b
try:
    result = divide_numbers(10, 0)
    print("Osztás eredménye:", result)
except ZeroDivisionError as e:
 print("Hiba:", e)
```

Ebben a példában a divide_numbers függvény megpróbálja elosztani az a számot b számmal. Ha a b értéke nulla, akkor a raise operátor használatával a ZeroDivisionError kivétel jön létre.

```
def process_data(data):
    if not data:
        raise ValueError("Üres adatok – nem feldolgozható ")
# További adatfeldolgozás
try:
    process_data([])
except ValueError as e:
    print("Adatfeldolgozási hiba:", e)
```

Ebben a példában a process_data függvény ValueError kivételt dob, ha az átadott adat üres.

Egy példa raise használatára, amely nem használ beépített hibafeldolgozókat, hanem saját módon kezeli az állapotokat és hibákat.

```
def calculate_discount(price, discount_rate):
    if not isinstance(price, (int, float)):
        raise TypeError("Az árnak egész vagy lebegőpontos számnak kell lennie")
    if not isinstance(discount_rate, (int, float)):
        raise TypeError("A kedvezmény aránynak egész vagy lebegőpontos számnak
kell lennie")
    if price <= 0:
        raise ValueError("Az ár nem lehet nulla vagy negatív")
    if discount_rate < 0 or discount_rate > 100:
        raise ValueError("A kedvezmény aránya 0 és 100 között kell legyen")
    discounted_price = price - (price * discount_rate / 100)
    return discounted_price
try:
    price = 100
    discount_rate = 20
    final_price = calculate_discount(price, discount_rate)
    print(f"A kedvezményes ár: {final_price}")
except TypeError as e:
    print(f"Hiba típusa: {e}")
except ValueError as e:
    print(f"Hiba értéke: {e}")
```

Ha az adatok helyesek: # Kimenet: A kedvezményes ár: 80.0
Ha hibás adatokat adunk meg:
Kimenet: Hiba típusa: Az árnak egész vagy lebegőpontos számnak kell lennie

A script működése:
1. A calculate_discount függvény ellenőrzi a bemenő paraméterek típusát és érvényességét. Ha valamelyik feltétel nem teljesül, akkor saját kivételt dob (TypeError vagy ValueError) a raise utasítással.
2. Hibakezelés (try-except blokk): A fő programban (try blokkban) meghívjuk a calculate_discount függvényt megfelelő paraméterekkel. Ha egy `TypeError` vagy `ValueError` keletkezik, a program ágazik az illesztett `except` blokkba, és kiírja a hibaüzenetet.

3. Kimenet: Ha a bemenetek megfelelnek az elvárásoknak, akkor a kedvezményes árat kiszámoljuk és kiírjuk. Ha nem felelnek meg, akkor a megfelelő hibaüzenetet kapjuk.

Ez a példa demonstrálja, hogyan lehet használni a `raise` utasítást saját kivételek kezelésére és dobására anélkül, hogy beépített hibafeldolgozókat használnánk. A `raise` lehetőséget ad arra, hogy saját feltételeket definiáljunk és reagáljunk rájuk a programunkban.

Vissza

13. Fájlkezelés a Pythonban

A fájlok kezelése Pythonban számos beépített modul segítségével történhet, amelyek lehetővé teszik a fájlok olvasását, írását, módosítását és kezelését. Itt van néhány fontos parancs és funkció, amelyeket használhatsz fájlok kezelésére Pythonban:

Fájl megnyitása és bezárása:
Az open() függvényt használjuk a fájlok megnyitására. A megnyitott fájlt mindig kell bezárni a close() metódussal.

Fájl megnyitása olvasásra (alapértelmezett mód: 'r' - read)

```
file = open('filename.txt', 'r')
```

Fájl bezárása

```
file.close()
```

Fájl olvasása:
A read() metódussal olvashatjuk be a fájl teljes tartalmát vagy megadhatunk egy méretet is, hogy csak annyi adatot olvassunk be, amennyi szükséges.

```
file = open('filename.txt', 'r')
content = file.read()  # Teljes tartalom beolvasása
print(content)
file.close()
```

Fájl olvasása soronként:
A readline() metódus segítségével soronként olvashatjuk be a fájl tartalmát.

```
file = open('filename.txt', 'r')
line = file.readline()
while line:
    print(line.strip())  # strip() használata a sorvége jelek eltávolítására
    line = file.readline()
file.close()
```

Fájl írása:
A write() metódussal írhatunk adatokat egy fájlba. A fájl írása során figyeljünk arra, hogy a meglévő tartalmat felülírjuk!

```
file = open('filename.txt', 'w')
file.write('Hello, World!\n')
file.write('Second line\n')
file.close()
```

Hozzáfűzés a fájlhoz:
Az append módban (az `a` használatával) nyitott fájlba is írhatunk, hozzátéve az új adatokat a meglévő fájl végéhez.

```
file = open('filename.txt', 'a')
file.write('New line appended\n')
file.close()
```

Kontextusmenedzserek használata (`with`):

A with blokk használata javasolt, mert automatikusan kezeli a fájl bezárását még akkor is, ha kivétel keletkezik a fájlműveletek során.

```
with open('filename.txt', 'r') as file:
    content = file.read()
    print(content)
# Itt a fájl automatikusan bezáródik
```

Fájl másolása és átnevezése:

A shutil modul copyfile() és move() függvényeivel másolhatunk és mozgathatunk fájlokat.

Fájl másolása

```
import shutil
shutil.copyfile('source.txt', 'destination.txt')
```

Fájl átnevezése

```
shutil.move('old_name.txt', 'new_name.txt')
import shutil
```

Ezek a parancsok és funkciók segítenek a fájlok hatékony kezelésében Pythonban. Fontos, hogy mindig gondoskodjunk a fájlok bezárásáról és a megfelelő műveletek végrehajtásáról a fájlok manipulációja során.

Vissza

14. Csomagok

Python Package Installer (pip)

A *Python Package Installer (pip)* használata általában akkor ajánlott, amikor külső Python könyvtárakat vagy csomagokat szeretne telepíteni, amelyek nem részei a Python Standard Library-nak. A Python Package Installer egy Python alapú csomagkezelő, amely lehetővé teszi Python csomagok egyszerű telepítését, frissítését és eltávolítását.

Néhány fontosabb helyzet, amikor szükség lehet a pip használatára:

Külső csomagok telepítése:

Ha olyan kódot ír, amely függőségeket igényel, amelyek nem részei a Python Standard Library-nak, akkor ezeket a csomagokat pip segítségével telepítheti. Például ha használni kell a Flask webkeretrendszert vagy a numpy számítástudományi könyvtárat, ezeket a pip telepítheti.

Csomagok frissítése és eltávolítása

A pip segítségével egyszerűen lehet frissíteni a meglévő csomagokat a legújabb verziókra, valamint eltávolíthatja azokat, amelyeket már nem használjuk.

Saját csomagok közzététele

Ha saját Python csomagot vagy könyvtárat írunk, és azt másokkal szeretnénk megosztani, a pip segítségével könnyedén közzé lehet tenni és telepítheti a csomagot a Python Package Index (PyPI) webhelyen.

Néhány alapvető parancs a pip használatára.

1.pip telepítése

A legtöbb esetben a pip már telepítve van a Python telepítésekor. Ha azonban nem lenne, akkor telepíthető a következő parancs segítségével:

sudo apt-get install python3-pip

2. Csomag telepítése

Először a command_sor megnyitása: **WIN+R** paranccsal. A pip segítségével telepíthetünk bármilyen Python csomagot a PyPI (Python Package Index) rendszerből. Például, a requests csomagot igy telepíthetjük:

pip install requests

E z a parancs letölti és telepíti a requests csomag legújabb verzióját és az összes szükséges függvényeket.

3. Frissítés

A pip segítségével frissíthetjük a már telepített csomagokat a legújabb verziókra:

pip install --upgrade package_name

Vissza

Melléklet. Python math modul

dir(math)

A dir() függvény Pythonban az objektum attribútumainak listáját adja vissza. Ha például dir(math)-ot hívunk, akkor ez a math modulban elérhető összes attribútum nevét adja vissza, mint például konstansok (pi, e), függvények (sqrt, sin, cos stb.) és más attribútumok.

Példa:
import math
print(dir(math))

Ez a parancs kimenetben felsorolja az összes elérhető név az math modulban.

help(math.<function>)
A help() függvény segítséget és dokumentációt nyújt egy adott objektumhoz, például modulokhoz, függvényekhez, osztályokhoz stb. Ha help(math.sqrt)-ot hívunk, akkor részletes leírást kapunk a sqrt függvényről a math modulban, beleértve a szintaxisát, paramétereit és visszatérési értékeit.

Példa:
import math
# Súgó a sqrt függvényhez a math modulban
help(math.sqrt)

Ez a kimenet részletes dokumentációt ad a sqrt függvényről, beleértve a paramétereit és a visszatérési értékeit.

Összegzően, a dir(math) listát ad az math modul összes elérhető nevéből, míg a help(math.sqrt) részletes leírást ad a sqrt függvényről a math modulban.

acos(x)	Az x arckoszinuszát adja eredményül.
acosh(x)	Az x inverz hiperbolikus koszinuszát adja eredményül.
asin(x)	Az x arcszinuszát adja eredményül.
asinh(x)	Az x inverz hiperbolikus szinuszát adja eredményül.
atan(x)	Az x arctangensét adja eredményül.
atan2(y, x)	Az atan(y/x) értékét adja vissza (radiánban).
atanh(x)	Az x inverz hiperbolikus tangensét adja eredményül.
cbrt(x)	a megadott x szám köbgyökét adja vissza.
ceil(x)	Az x-nél nagyobb vagy egyenlő legkisebb egész számot adja vissza.
comb(n, k)	Kombináció: *k elem kiválasztása n* elemből ismétlődés nélkül. Képlet n! / (k! * (n - k)!) , k <= n
copysign(x, y)	Az x-et y előjellel adja vissza.
cos(x)	Az x koszinuszát adja eredményül.
cosh(x)	Az x hiperbolikus koszinuszát adja eredményül.
degrees(x)	Alakítsa át az x szöget radiánból fokokra.
dist(p, q)	Ez a függvény két pont közötti távolságot számítja ki az (x, y, z) koordináták alapján. A p és q paramétereknek ugyan annak a méretű iterálható objektumnak kell lenniük (pl. tuple vagy lista).
E	e matematikai állandó (2,71828...).
erf(x)	A hibafüggvényt adja vissza x-ben.

erfc(x)	Az x-nél lévő hiba további függvényét adja vissza.
exp(x)	Az e**x-et adja vissza.
expm1(x)	e**x – 1-et tér vissza.
fabs(x)	Az x abszolút értékét adja vissza.
factorial(x)	Az x faktoriálisát adja eredményül.
floor(x)	Az x-nél kisebb vagy azzal egyenlő legnagyobb egész számot adja vissza.
fmod(x, y)	A maradékot adja vissza, ha x elosztva y-val.
frexp(x)	Az x mantisszáját és kitevőjét adja eredményül (m, e) párként.
fsum(iterált_objektum)	Az integrált objektum lebegőpontos értékeinek pontos összegét adja vissza.
gamma(x)	A Gamma függvényt adja vissza x-ben.
gcd(*integers)	legnagyobb közös osztó
hypot(*coordinates)	Ez a függvény az adott koordináták alapján kiszámítja az euklideszi normát.
hypot(x, y)	Kiszámítja egy x és y szárú háromszög befogóját (math.sqrt(x * x + y * y)).
isfinite(x)	Igaz értéket ad vissza, ha x nem végtelen és nem is NaN (nem szám).
isinf(x)	Igaz értéket ad vissza, ha x pozitív vagy negatív végtelen.
isnan(x)	Igaz értéket ad vissza, ha x NaN.
ldexp(x, i)	Visszaadja x * (2**i).
lgamma(x)	A Gamma-függvény x-nél lévő abszolút értékének természetes logaritmusát adja vissza.
log(x[, b])	Az x logaritmusát adja vissza b bázisra (alapértelmezett e).
log10(x)	Az x 10-es bázisú logaritmusát adja eredményül.
log1p(x)	1+x természetes logaritmusát adja eredményül.
log2(x)	Az x 2-es bázisú logaritmusát adja eredményül.
modf(x)	Az x tört és egész részét adja eredményül.
Pi	Pi matematikai állandó, amely egyenlő a kör hosszának és átmérőjének arányával (3,14159...).
pow(x, y)	Az x-et y hatványára emelve adja vissza.
pow(x, y)	Ez a függvény az x szám y hatványát számítja ki.
radians(x)	Konvertálja át az x szöget fokokból radiánba.
sin(x)	Az x szinuszát adja eredményül.
sinh(x)	Az x hiperbolikus szinuszát adja eredményül.
sqrt(x)	Az x négyzetgyökét adja eredményül.
tan(x)	Az x tangensét adja eredményül.
tanh(x)	Az x hiperbolikus tangensét adja eredményül.
trunc(x)	Az x „csonka” egész értékét adja vissza (elveti a szám tört részét).

Vissza

IRODALOMJEGYZÉK

1. https://mek.oszk.hu/08400/08435/08435.pdf, Gérard Swinner,Tanuljunk meg programozni Python nyelven.
2. Васильєв О. М. Програмування мовою Python. Тернопіль: Навчальна книга Богдан, 2019. 504с
3. 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
4. *Костюченко А.О.* Основи програмування мовою Python: навчальний посібник. Ч.: ФОП Баликіна С.М. 2020. 180 с.
5. https://www.w3schools.com/python
6. https://www.malnasuli.hu/python/keszits-hazilag-grafikus-vezerlofeluletet-tkinter-1-resz/
7. http://nyelvek.inf.elte.hu/leirasok/Python/index.php?chapter=16
8. https://szit.hu/doku.php?id=oktatas:programozas:python:tkinter:hagyomanyosan
9. https://mek.oszk.hu/08400/08435/08435.pdf

Програмування 1. Мова програмування Python/ Programozás 1. Python programozási nyelv методичні вказівки до практичних занять з дисципліни «Сучасні мови програмування» для здобувачів першого (бакалаврського) рівня вищої освіти денної та заочної форм навчання, освітня програма: «Середня освіта (Інформатика)», галузь знань: «01 Освіта/Педагогіка», спеціальність (спеціалізація): «014 Середня освіта (014.04 Інформатика)» / Розробник: Йожеф Головач. – Берегове: ЗУІ ім. Ф.Ракоці ІІ, 2024, 77 с. (угорською мовою).

Метою методичного посібника є засвоєння мови програмування Python, яку студенти вивчають в курсі « Сучасні мови програмування». У роботі наведені основні конструкції, функції, пакети мови програмування Python, наводяться приклади їх використання для типових простих задач. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу, а також студентами-математиками.

Виробничо-практичне видання

Програмування 1. Мова програмування Python/
Programozás 1. Python programozási nyelv
Методичні вказівки до практичних занять з дісципліни
«Сучасні мови програмування»

2024 р.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від 13 серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол №22 від «26» серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца
Ракоці ІІ (протокол №7 від « 27» серпня 2024 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 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

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції Серія ДК 7637 від 19 липня 2022 року*

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