ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ II

MODERN PROGRAMOZÁSI NYELVEK (PYTHON 2 rész)

MÓDSZERTANI ÚTMUTATÓ
előadásokhoz és gyakorlati foglalkozásokhoz

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

Берегове / Beregszász
2025 р. / 2025

Методичні вказівки до лекційних та практичних занять з дисципліни «Сучасні мови програмування (Python. 2 частина) » розроблені на основі освітньої програми підготовки бакалаврів з галузі знань «А Освіта/Педагогіка» за напрямом «А4.09 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є засвоєння студентами основ мови програмування Python, та використання їх при розробці програм різної складності. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу. Крім того, методичні вказівки рекомендуються студентам-бакалаврам по спеціальності «А4.09 Середня освіта (Математика)» при вивченні курсу «Алгоритми і програмування».

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

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

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

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

Розробник методичних вказівок:

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

Рецензенти:

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

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

Відповідальні за випуск:

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

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

За зміст методичних вказівок відповідальність несе розробник.

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

A „Modern programozási nyelvek (Python 2 rész)” módszertani útmutató a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola, II. éves, informatika szakos, nappali és levelezős hallgatóinak készült, A módszertani útmutató célja, hogy a hallgatók elsajátítsák a Python programozási nyelv alapjait, és azokat alkalmazni tudják különböző komplex programok készítése során. Emellett ajánlott a „A4.09 Középfokú oktatás (Matematika)” szakos alapszakos III. éves hallgatók számára is az „Algoritmusok és programozás” tantárgy tanulásához.

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

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

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

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

A módszertani útmutató kidolgozója:

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

Szakmai lektorok:

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

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

A kiadásért felelnek:

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

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

A segédlet tartalmáért kizárólag a módszertani útmutató kidolgozója felel.

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

TARTALOMJEGYZÉK

1. Programok fejlesztése Python nyelven

Kódszerkezetek

A program – utasítások sorozata. Attól függően, hogyan értékelődnek ki a kifejezések, a program dönthet úgy, hogy kihagyja az utasításokat, megismétli azokat, vagy kiválaszt egyet több közül a végrehajtásra. Tehát a programokban gyakran szükséges feltételek ellenőrzése és döntések meghozatala ezen feltételek alapján.

Feltételek létrehozása és ellenőrzése

Feltételek létrehozásához és ellenőrzéséhez *logikai értékeket, összehasonlító operátorokat és logikai operátorokat* használnak.

Logikai értékek

Míg az egész szám, valós szám és karakterlánc típusú adatok tetszőleges számú értéket felvehetnek, *a logikai (Boolean) típus csak két értéket vehet fel: True (igaz) és False (hamis)*. A logikai típus nevét az angol matematikus, a matematikai logika alapítója, George Boole után kapta.

```
>>> one = True
>>> one
True>>> two = false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
>>> two = False
>>> two
False
```

A Python kódban a True és False logikai értékeket idézőjelek nélkül írjuk, és ezek neve nagy T és F betűvel kezdődik, míg a szó többi része kisbetűkkel íródik.

Összehasonlító operátorok

Az összehasonlító operátorok két értéket hasonlítanak össze, és eredményül logikai értéket adnak vissza: igaz vagy hamis.

Összehasonlító operátorok táblázata

Operátor	Név
==	Egyenlő
!=	Nem egyenlő
<	Kisebb, mint
>	Nagyobb, mint
<=	Kisebb vagy egyenlő
>=	Nagyobb vagy egyenlő

Példák.

```
>>> a = 10
>>> a == 5
False
>>> a == 10
True
>>> 4 < a
True
>>> a > 12
False
>>> 'breeze' == 'breeze'
True
>>> 'breeze' == 'Breeze'
False
>>> 'fine' != 'rain'
True
>>> 16 == 16.0
True
>>> 16 == '16'
False
>>> 45 >= 11
True
```

Legyünk figyelmesek, hogy az egyenlőség ellenőrzéséhez két „egyenlő” jelet (==) használunk; emlékezzünk, hogy egyetlen „egyenlő” jel (=) a változónak értéket rendel hozzá.

Logikai operátorok

Ha egyszerre több összehasonlítást kell végezni, használjuk a logikai (Boolean) operátorokat: **and**, **or** és **not**, hogy meghatározzuk a végső eredményt. A logikai operátorok kisebb prioritással rendelkeznek, mint azok a kódrészek, amelyeket összehasonlítanak. Ez azt jelenti, hogy először az egyes részek eredményei számítódnak ki, majd ezek összehasonlításra kerülnek.

A logikai operátorok (and és or) mindig két logikai értékkel (vagy kifejezéssel) dolgoznak, ezért binárisnak nevezhetők.

Logikai operátorok igazságtáblázatai

AND operátor igazságtáblázata

Kifejezés	Eredmény
True and True	True
True and False	False
False and True	False
False and False	False

OR operátor igazságtáblázata

Kifejezés	Eredmény
True or True	True
True or False	True
False or True	True
False or False	False

A logikai operator not megváltoztatja a logikai értéket annak ellentétére. A not operátor mindig egy logikai értéken (vagy kifejezésen) dolgozik, ezért egyértelmű (unáris).

NOT operátor igazságtáblázata

Kifejezés	Eredmény
not True	False
not False	True

A logikai operátorok (and, or, not) működésének bemutatására futtassuk a következő példákat az interpreterben:

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
```

Logikai értékek, összehasonlító operátorok, logikai operátorok kombinálása.

Futtassuk az alábbi példákat a Python interpreterben:

```
>>> (7 < 10) and (6 < 9)
True
>>> (3 < 5) and (16 < 8)
False
>>> (3 == 6) or (32 == 32)
True
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Először a bal oldali kifejezés kerül kiszámításra, majd a jobb oldali. Miután mindkét eredmény ismert, kiszámítódik az egész kifejezés végső eredménye, amely egyetlen logikai érték lesz.

A matematikai operátorokhoz hasonlóan a logikai operátorok végrehajtásának sorrendje meghatározott. Miután az összes matematikai operátor és összehasonlító operátor végrehajtásra került, először a not operátorok, majd az and operátorok, és csak ezután az or operátorok hajtódnak végre.

2. Elágazások

Bármely logikai kifejezés feltételként kezelhető, amelynek kiértékelése mindig True (igaz) vagy False (hamis) logikai értéket ad vissza. Attól függően, hogy a feltétel milyen értéket vesz fel, a program végrehajthatja vagy kihagyhatja a kódrészeket - kódblo- kokat, azaz elágazás történik.

A kódrészlet jellemzője az, hogy a bal oldali margótól való behúzás nő. A blokkok más blokkokat is tartalmazhatnak. A blokk végét a behúzás csökkenése jelzi nullára vagy a külső blokk behúzásának mértékére.

Nézzük meg a következő programot (hozzunk létre egy új fájlt az általunk választott fejlesztői környezetben, és mentsük el egy meghatározott névvel):

```
name = 'Mary'
password = 'hurricane'
if name == 'Mary':
....print('Hello, Mary.')
....if password == 'swordfish':
........print('Access granted.')
....else:
........print('Wrong password.')
else:
....print('User not registered.')
```

A behúzások világosan pontokkal vannak jelölve a szemléltetés kedvéért. Valós körülmények között a behúzások a *Space billentyűvel* készülnek.

Az első kódrészlet: a print('Hello, Mary.') sorral kezdődik és tartalmazza az összes következő kódsort a második else utasításig (nem beleértve).

A második kódrészlet: az első kódrészleten belül helyezkedik el, csak egy print('Access granted.') sort tartalmaz és a második if utasításhoz tartozik.

A harmadik kódrészlet: csak egy print('Wrong password.') sorból áll, és az első else utasításhoz tartozik.

A negyedik kódrészlet: csak egy print('User not registered.') sorból áll, és a második else utasításhoz tartozik.

Mi igaz, és mi hamis?

Hogyan határozzuk meg az igazságot vagy hamisságot, ha az ellenőrzött elem nem logikai érték? Például a következő értékek a False (hamis) kategóriába tartoznak:

- Boolean változó: False
- Érték: None
- Egész szám: 0
- Lebegőpontos szám: 0.0
- Üres sztring: ''
- Üres lista: []
- Üres tuple: ()
- Üres szótár: {}
- Üres halmaz: set()

Minden egyéb érték igaznak (True) számít.

IF utasítás

Az if utasítással ellenőrizhetjük a program jelenlegi állapotát, és a vizsgálat eredményei alapján kiválaszthatjuk a további lépéseket. Az if utasítás szintaxisa a következő:

```
if feltétel:
    kódrészlet
```

Ez a fajta elágazás a programozásban *hiányos elágazásnak* nevezett forma.

Ha a feltétel kiértékelése igaz (True), akkor a kódrészlet végrehajtódik. Ha a feltétel kiértékelése hamis (False), akkor a kódrészlet nem hajtódik végre.

Egyszerűen szólva, az if utasítás működése így írható le: "*Ha a feltétel igaz, akkor hajtsd végre ezt a kódrészletet.*"

A Python-ban az if utasítás a következő elemeket tartalmazza:

- az if kulcsszó,
- a feltétel (kifejezés, amelynek eredménye True vagy False),
- kettőspont,
- a kódrészlet, amely behúzással kezdődik az új sorban.

Tegyük fel, hogy van egy kódunk, amely ellenőrzi, hogy a név megfelel-e egy adott névnek, például Mary-nek:

```
if name == 'Mary':
    print('Hello, Mary.')
```

Ha a name változó értéke előzőleg "Mary" volt, akkor ennek a kódrészletnek a végrehajtása után a képernyőn a következő üzenet fog megjelenni:

```
print('Hello, Mary.')
```

Ha a name változó más értékkel rendelkezik, akkor az if utasítás kódrészlete, amely egyetlen print('Hello, Mary.') utasítást tartalmaz, nem hajtódik végre, és nem fog megjelenni semmilyen üzenet a képernyőn.

else utasítás

Az if kódrészlet után opcionálisan egy else utasítás következhet, amelynek saját kódrészlete akkor hajtódik végre, ha az if feltétele hamis. A szintaxis a következő:

```
if feltétel:
    kódrészlet, ha a feltétel igaz
else:
    kódrészlet, ha a feltétel hamis
```

Ez a programozásban használt *elágazás teljes formája.* Egyszerűen szólva, az if/else konstrukció így olvasható: *"Ha a feltétel igaz, hajtsd végre ezt a kódrészletet. Ellenkező esetben hajtsd végre a következő kódrészletet."*

A Python-ban az else utasításnak nincs feltétele, és mindig az alábbi elemeket tartalmazza:

- az else kulcsszó,
- kettőspont,
- a kódrészlet, amely behúzással kezdődik az új sorban.

Vegyük azt a példát, ahol a name változó nem Mary-t, hanem Jack-et tartalmaz:

```
if name == 'Mary':
    print('Hello, Mary.')
else:
    print('Hello, stranger.')
```

Ebben az esetben a kód végrehajtása a következő üzenetet fogja eredményezni:

```
Hello, stranger.
```

elif utasítás

Több feltétel ellenőrzésére az elif utasítást használják (az else if rövidítése). *Az elif utasítás csak az if utasítás után, vagy egy másik elif utasítás után használható.* Ez az utasítás lehetővé teszi további feltételek megadását. A szintaxis a következő:

```
if feltétel1:
    kódrészlet, ha a feltétel1 igaz
elif feltétel2:
    kódrészlet, ha a feltétel2 igaz
elif feltétel3:
    kódrészlet, ha a feltétel3 igaz
...
```

A Python-ban az elif utasítás mindig az alábbi elemeket tartalmazza:

- az elif kulcsszó,
- a feltétel (kifejezés, amelynek eredménye True vagy False),
- kettőspont,
- a kódrészlet, amely behúzással kezdődik az új sorban.

Ha több elif utasítás van, akkor az első igaz feltételt követő blokkok hajtódnak végre, a többi elif blokk automatikusan átugrásra kerül. Ezért egy kódrészlet vagy csak egyetlen blokkot hajt végre, vagy egyet sem.

if/elif/else konstrukció

Ha szükséges, az utolsó elif utasítást követően egy else utasítást is elhelyezhetünk. Ebben az esetben garantált, hogy legalább egy (és csak egy) kódrészlet végrehajtódik:

```
if feltétel1:
    kódrészlet, ha a feltétel1 igaz
elif feltétel2:
    kódrészlet, ha a feltétel2 igaz
elif feltétel3:
    kódrészlet, ha a feltétel3 igaz
else:
    kódrészlet, ha minden feltétel hamis
```

Ha minden if és elif utasítás feltétele hamis, akkor az else utasítás kódrészlete fog végrehajtódni.

Egyszerűen szólva, az if/elif/else konstrukció így olvasható: *"Ha az első feltétel igaz, hajtsd végre ezt a kódrészletet. Ha a második feltétel igaz, hajtsd végre ezt a kódrészletet. Ellenkező esetben hajtsd végre ezt a kódrészletet."*

Példa:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

Ha a bemeneti adatok a következők: name = 'Alex', age = 36, akkor a kód eredménye a következő lesz:

```
You are neither Alice nor a little kid.
```

3. Ciklusok

A feltételek ellenőrzése után néha szükség van arra, hogy egy műveletet többször végrehajtsunk. Ehhez ciklusokra van szükség.

3.1. while ciklus

A Python-ban az egyik ciklus a while. A szintaxis a következő:

while feltétel:
    kódrészlet, ha a feltétel igaz

A while ciklus szintaxisa hasonló az if utasításhoz, de másképp működik. A ciklus végrehajtásának minden iterációjában a feltételt újra ellenőrzi a program. Ha a feltétel igaz, akkor a ciklus folytatódik, ellenkező esetben a program kilép a ciklusból és folytatja a végrehajtást.

Példa a while ciklusra:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Ez a kód 1-től 5-ig kiírja a számokat.

Hogyan működik a while ciklus ebben a példában?

Először értéket rendelünk 1-hez a count változóhoz. A while ciklus összehasonlítja a count változó értékét (ami most 1) az 5-tel, és folytatja a működést, ha a count értéke kisebb vagy egyenlő 5-tel (a ciklus feltétele igaz), különben a ciklus véget ér (a ciklus feltétele hamis). A ciklus belsejében (ezt a részt a ciklus törzsének nevezzük) kiíródik a count változó értéke. Ezután a count változó értéke növekszik 1-el a count += 1 kifejezés segítségével. A Python visszatér a ciklus elejére, és újra összehasonlítja a count változó értékét az 5-tel. A count értéke most 2, ezért a while ciklus törzse újra végrehajtódik, és a count változó értéke 3-ra nő. A végrehajtás folytatódik, amíg a count változó értéke el nem éri az 5-öt, majd 6-ra nem nő a ciklus törzsében. A következő alkalommal, amikor a ciklus elejére ér, a count <= 5 feltétel hamis értéket ad vissza, és a while ciklus véget ér. A Python folytatja a program többi sorának végrehajtását.

A ciklus megszakítása: break

Ha szükséges, hogy a ciklus addig fusson, amíg valami nem történik, de pontosan nem tudjuk, mikor fog ez megtörténni, akkor használhatunk egy végtelen ciklust break utasítással. Ha a program végrehajtás közben eléri a break parancsot, a ciklus azonnal leáll.
Nézzük meg egy olyan program példáját, amelyben a felhasználó egy karakterláncot ad meg az input() függvény segítségével, majd ezt a karakterláncot kiírja a képernyőre az első betű nagybetűs formájában. A ciklus akkor szakad meg, ha a felhasználó egy q karaktert tartalmazó karakterláncot ad meg.

Írd be a következő kódot a programozási környezetbe, mentsd el egy megfelelő névvel és hajtsd végre:

```
stuff = ''
while True:
     print("String to capitalize [type q to quit]: ")
    stuff = input()
    if stuff == "q":
        break
    print(stuff.capitalize())
print('Thank you!')
```

A while True: sor végtelen ciklust hoz létre - a ciklus feltétele mindig igaz (True). A program mindig végrehajtja a ciklus parancsait, és csak akkor lép ki belőle, ha a break utasítás végrehajtódik. A végtelen ciklus, amelyből lehetetlen kilépni, gyakori hiba. A program végrehajtásának eredménye és a blokkvázlat, ha használjuk a break parancsot, a következő lesz:

```
String to capitalize [type q to quit]:
test
Test
String to capitalize [type q to quit]:
hey, it works!
Hey, it works!
String to capitalize [type q to quit]:
q
Thank you!
```

Végtelen ciklus és kilépés belőle

Azt is ellenőrizhetjük, hogyan működik a végtelen ciklus, ha a következő kódot beírjuk és elmentjük egy megfelelő névvel:
Ha a program végtelen ciklusba került, nyomj meg egy Ctrl+C kombinációt, amely leállítja a program végrehajtását.

```
while True:
    print('Hello, world!')
```

Ez a program végrehajtás közben folyamatosan kiírja a Hello, world! üzenetet, mivel a while ciklus feltétele mindig igaz (True). Ahhoz, hogy megakadályozzuk a program végtelen ciklusba kerülését, elemezzük, hogyan kezeljük azt az értéket, amely a ciklusból való kilépést kell, hogy eredményezze. Ellenőrizzük, hogy a program legalább egy részében a ciklus feltétele hamisra változhat-e, vagy végrehajtódik a break parancs. Például, ha a kódban:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

kihadjuk a count += 1 sort:

```
count = 1
while count <= 5:
    print(count)
```

akkor a ciklus végtelen lesz:

```
1
1
1
...
```

A ciklus folytatása: continue

Néha nem szükséges megszakítani az egész ciklust, hanem csak egy iterációt kell kihagyni valamilyen okból. Erre használjuk a *continue* utasítást. Amikor a program végrehajtása eléri a continue utasítást, azonnal visszatér a ciklus elejére, és a feltétel újra ellenőrzésre kerül (ugyanez történik, ha a program eléri a ciklus végét is). Írjunk és mentsünk el egy programot egy megfelelő névvel, amely kéri a felhasználótól a felhasználónevet és a jelszót a belépéshez:

```
name = ''
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe. What is the password? (It is a
fish.)')
    password = input()
```

```
    if password == 'swordfish':
        break
print('Access granted.')
```

Ha a felhasználó bármilyen nevet ad meg, kivéve Joe, a continue utasítás arra kéri a programot, hogy térjen vissza a ciklus elejére. A feltétel újraellenőrzése után a program mindig belép a ciklus törzsébe, mivel a feltétel mindig igaz (True). Ha az if utasítás végrehajtódik, akkor a program a jelszót kérdezi, és ha a felhasználó swordfish jelszót ad meg, akkor a break utasítás végrehajtódik, a program megszakítja a ciklust, és kiírja az Access granted. üzenetet. Ellenkező esetben a program végrehajtása visszatér a while ciklus végére, és azonnal újraindul.
A program végrehajtásának eredménye különböző felhasználói bemenetek esetén:

```
Who are you?
Alice
Who are you?
Jack
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
shark
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted
```

A program nem kér jelszót, amíg a felhasználó nem erősíti meg, hogy a neve Joe. Ha helytelen jelszót ad meg, a program újra kérdezni fogja a nevet, és miután a helyes jelszót megadja, a program befejezi a végrehajtást.

3.2. for ciklus

A while ciklus addig hajtódik végre, amíg a feltétel igaz. Ha egy blokkot csak egy meghatározott (ismert) számú alkalommal kell végrehajtani, akkor a for ciklust és a **range() függvényt** használják. A ciklus és a függvény szintaxisa a következőképpen néz ki:

```
for változó in range():
    kódrészlet
```

A Python for utasítása mindig az alábbi elemekből áll:

- a for kulcsszó
- a változó neve
- az in kulcsszó
- a range() függvény hívása, amelybe legfeljebb három egész számot adhatunk meg, vesszővel elválasztva
- kettőspont

egy kódrészlet, amely a következő sorban kezdődik, és behúzással rendelkezik. Ahhoz, hogy kipróbáljuk, hogyan működik a for ciklus, hozzunk létre egy új programot és mentsük el egy megfelelő névvel:

```
print('My name is')
for i in range(5):
    print('Anakin Skywalker (' + str(i) + ')')
```

A for ciklus kódrészlete 5 alkalommal hajtódik végre. Az első iterációban (az első végrehajtáskor) a i változó értéke 0-ra van beállítva. A print() függvények hívása a ciklus törzsében kiírja az Anakin Skywalker (0) üzenetet. Amikor a ciklus befejezi az iterációt, és végrehajtja az összes kódrészletet, a vezérlés visszatér a ciklus elejére, ahol a for utasítás 1-tel növeli a i változó értékét. Ezért a range(5) függvény hívás 5-ször hajtja végre a ciklus kódrészletét, a következő i értékekkel: 0, 1, 2, 3 és 4.
A range() függvény zárójelében szereplő 5-ös szám nem szerepel a tartományban.
Ha ezt a programot futtatjuk, 5 sort fogunk kapni, mindegyik végén az aktuális i változó értékével, és a ciklus befejeződik:

```
My name is
Anakin Skywalker (0)
Anakin Skywalker (1)
Anakin Skywalker (2)
Anakin Skywalker (3)
Anakin Skywalker (4)
```

A break és a continue utasítások a for ciklusban ugyanúgy működnek, mint a while ciklusban. Ezen kívül minden, amit a for ciklus végez, elvégezhetünk a while ciklus segítségével is. Írd át a korábbi programot a while ciklus használatával, és ellenőrizd, hogy ugyanazokat az eredményeket kapjuk-e, mint a for ciklus használatakor:

```
print('My name is')
i = 0
while i < 5:
    print('Anakin Skywalker (' + str(i) + ')')
    i = i + 1 # i += 1
```

A range() függvény

A **range() függvénybe** három egész számot is átadhatunk, amelyek vesszővel vannak elválasztva. Az első szám meghatározza, hogy honnan kezdődik a ciklus változójának értéke, a második szám azt, hogy meddig változhat a ciklus változója (de ez a szám nem szerepel a tartományban). Például, ha a következő kódot futtatjuk:

```
for i in range(12, 16):
    print(i)
```

az alábbi eredményt kapjuk:

```
12
13
14
15
```

A range() függvényt három argumentummal is meghívhatjuk. Az első két szám határozza meg a változó ciklusértékeinek kezdő és végpontját, a harmadik pedig a lépésközt adja meg. Például, ha a következő kódot futtatjuk:

```
for i in range(1, 10, 2):
    print(i)
```

az alábbi eredményt kapjuk:

```
1
3
5
7
9
```

A range() függvény negatív lépésközzel is meghívható. Ebben az esetben a ciklus változója nagyobb értékekről kisebbekre változik. Ha a következő kódot futtatjuk:

```
for i in range(3, -4, -1):
    print(i)
```

az alábbi eredményt kapjuk:

```
3
2
1
0
-1
```

```
-2
-3
```

3.3. for ciklus és sorozatok

A for ciklus lehetővé teszi különböző adatszerkezetek, amelyek sorozatok, végigiterálását. A for ciklus használata egy lista elemeinek bejárására:

```
birds = ['pigeon', 'crow', 'owl', 'eagle']
for bird in birds:
    print(bird)
```

Eredmény:

```
pigeon
crow
owl
eagle
```

A listák, mint például a birds, az iterálható objektumok példái Pythonban, akárcsak a karakterláncok, tuple-ök, szótárak. A lista vagy tuple bejárása egyesével adja vissza az elemeket. A karakterlánc bejárása egyesével adja vissza a karaktereket, ahogy azt a következő példában is láthatjuk:

```
word = 'crab'
for letter in word:
    print(letter)
```

Eredmény:

```
c
r
a
b
```

A szótár iterálása ciklusban (vagy a **keys()** függvény használatával) a szótár kulcsait adja vissza. A következő példában a szótár kulcsai az emberi tevékenységek területei, míg a szótár értékei a szakmák nevei:

```
professions = {'business': 'economist', 'tv': 'newsreader',
'it': 'web developer', 'education': 'teacher'}
for key in professions: # або for key in
professions.keys():
    print(key)
```

Eredmény:

```
business
```

```
education
tv
it
```

Ha a szótár értékeire szeretnél iterálni, nem pedig a kulcsokra, akkor a **values()** függvényt kell használni.

```
professions = {'business': 'economist', 'tv': 'newsreader',
'it': 'web developer', 'education': 'teacher'}
for value in professions.values():
    print(value)
```

Eredmény:

```
teacher
newsreader
web developer
economist
```

A szótár kulcsait és értékeit egyaránt megkaphatjuk az **items()** függvény segítségével. Az eredmény egy kulcs-érték párból álló tuple lesz.

```
professions = {'business': 'economist', 'tv': 'newsreader',
'it': 'web developer', 'education': 'teacher'}
for item in professions.items():
    print(item)
```

Eredmény:

```
('business', 'economist')
('education', 'teacher')
('it', 'web developer')
('tv', 'newsreader')
```

A tuple elemeit külön is megkaphatjuk minden egyes iteráció során. Minden egyes items() függvény által visszaadott tuple esetén az első értéket (a kulcsot) rendeld a key változóhoz, a második értéket (az értéket) pedig a value változóhoz:

```
professions = {'business': 'economist', 'tv': 'newsreader',
'it': 'web developer', 'education': 'teacher'}
for key, value in professions.items():
    print('Category', key, 'has a profession', value)
```

Eredmény:

```
Category business has a profession economist
Category education has a profession teacher
Category it has a profession web developer
Category tv has a profession newsreader
```

A szótár mindig fenntartja a kapcsolatot a kulcs és az ahhoz tartozó érték között. Az elemek meghatározott sorrendben történő lekérésének egyik módja a kulcsok rendezése, amelyeket a for ciklus visszaad. A kulcsok rendezett másolatát a **sorted()** függvény segítségével kaphatjuk meg:

```
favorite_languages = {'john': 'Python', 'catherine': 'C',
'mary': 'Ruby', 'alex': 'Python'}
for name in sorted(favorite_languages.keys()):
    print(name.title() + ", thank you for taking the
poll.")
```

Ez a konstrukció a szótár összes kulcsának listáját adja vissza, de rendezi őket, mielőtt végighaladna az elemeken. Ennek eredményeként a képernyőn az összes felhasználó látható, akik részt vettek a kedvenc programozási nyelvekről szóló felmérésben, és a nevük ábécé sorrendben van rendezve:

```
Alex, thank you for taking the poll.
Catherine, thank you for taking the poll.
John, thank you for taking the poll.
Mary, thank you for taking the poll.
```

Az enumerate() függvény

Olyan nyelvekben, mint a C, a sorozat elemeit nem az elemek, hanem az indexek alapján iterálják. Az indexek használatával a megadott indexekhez tartozó elemeket kaphatunk. Az alábbiakban bemutatunk egy megoldást, amely a Python beépített range() és len() függvényeit használja:

```
popular_sites = ['Google.com', 'Youtube.com',
'Facebook.com', 'Baidu.com', 'Wikipedia.org', 'Yahoo.com'
,'Amazon.com']
for index in range(len(popular_sites)):
    print(index, popular_sites[index])
```

Eredmény:

```
0 Google.com
1 Youtube.com
2 Facebook.com
3 Baidu.com
4 Wikipedia.org
```

```
5 Yahoo.com
6 Amazon.com
```

Általában Pythonban a sorozat elemeit iteráljuk az indexek helyett. Azonban bizonyos esetekben szükség lehet az indexekre is. A Python biztosít egy beépített **enumerate() függvényt**, amely lehetővé teszi az indexek és a hozzájuk tartozó elemek párba fogását, így a range() és len() függvények használata feleslegessé válik. Az enumerate() függvény minden egyes elemre egy (index, elem) tuple-t ad vissza:

```
popular_sites = ['Google.com', 'Youtube.com',
'Facebook.com', 'Baidu.com', 'Wikipedia.org', 'Yahoo.com'
,'Amazon.com']
for index, value in enumerate(popular_sites, 1):
    print(index, value)
```

A fenti példában az enumerate() függvény egy opcionális argumentumot is tartalmazott, amelynek értéke 1, jelezve, hogy a számozás 1-től kezdődjön (alapértelmezés szerint a számláló 0-tól kezdődik):

```
1 Google.com
2 Youtube.com
3 Facebook.com
4 Baidu.com
5 Wikipedia.org
6 Yahoo.com
7 Amazon.com
```

A zip() függvény

zip() függvényt arra használjuk, hogy több sorozaton párhuzamosan iteráljunk.

```
days = ['Monday', 'Tuesday', 'Wednesday']
fruits = ['coconut', 'lemon', 'mango']
drinks = ['coffee', 'tea', 'fruit juice']
desserts = ['marmalade', 'ice cream', 'pie', 'pudding']
for day, fruit, drink, dessert in zip(days, fruits, drinks,
desserts):
    print(day, ": drink", drink, "eat", fruit, "enjoy",
dessert)
```

Eredmény:

```
Monday : drink coffee eat coconut enjoy marmalade
Tuesday : drink tea eat lemon enjoy ice cream
```

```
Wednesday : drink fruit juice eat mango enjoy pie
```

A zip() függvény akkor fejezi be a munkáját, ha a legrövidebb sorozat végére ér. Az egyik lista (desserts) hosszabb, mint a többi, ezért nem fogunk pudingot kapni, amíg nem növeljük a többi listát. A zip() függvényt arra is használhatjuk, hogy több sorozaton végighaladva párokat hozzunk létre az azonos indexekkel rendelkező elemekből. Hozzunk létre két tuple-t az angol és francia szavak megfelelő párokból:

```
english = 'Monday', 'Tuesday', 'Wednesday'
french = 'Lundi', 'Mardi', 'Mercredi'
```

Használjuk a zip() függvényt a párok létrehozására. Az eredmény, amit a zip() visszaad, nem lista vagy tuple, de bármelyik sorozattá átalakítható, például listává a list() függvény segítségével:

```
print(list(zip(english, french)))
```

Eredmény:

```
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday',
'Mercredi')]
```

Ha közvetlenül a zip() függvény eredményét a dict() függvénynek adjuk át, egy kis angol-francia szótárat kapunk:

```
print(dict(zip(english, french)))
```

Eredmény:

```
{'Wednesday': 'Mercredi', 'Monday': 'Lundi', 'Tuesday':
'Mardi'}
```

map() függvény

Nagyon gyakran szükség van arra, hogy megváltoztassuk a sorozat típusát, vagy az elemek típusát. Például egy számokat tartalmazó sztringlistát egész számok listájává alakítani, vagy magát a listát tuple-lé vagy halmazzá alakítani, ugyanazon elemekkel.

Nézzük meg, hogyan történik mindez a gyakorlatban. Képzeljünk el egy

```
my_str_list = ['1', '2', '3', '4']
```

listát, amely számokat tartalmazó sztringeket tartalmaz. Alakítsuk át ezt a listát egész számok listájává a következő kóddal:

```
result = []
for item in my_str_list:
    result.append(int(item))
print(result)
```

Eredmény:

```
[1, 2, 3, 4]
```

Most próbáljuk meg átalakítani magát a sorozat típusát, miközben megtartjuk ugyanazt az elemeket. Például alakítsuk át a my_str_list listát tuple-lé:

```
result = tuple()
for item in my_str_list:
    result += (item,)
print(result)
```

Eredmény:

```
('1', '2', '3', '4')
```

Még egy példa, ami illusztrálja nem csak a sorozat típusának, hanem az elemek típusának megváltoztatását is:

```
result = set()
for item in my_str_list:
    result.add(float(item))
print(result)
```

Ebben az esetben a számokat tartalmazó sztringek listáját valódi számok halmazává alakítjuk:

Eredmény:

```
{1.0, 2.0, 3.0, 4.0}
```

Mindezeket a lépéseket gyorsabban is végrehajthatjuk rövidebb kóddal, amit a map() függvény segítségével érhetünk el. A map() függvény alkalmaz egy megadott függvényt (például int(), float(), str() stb.) minden egyes elemre a sorozatban (lista, tuple, halmaz stb.). Az eredményt listává, tuple-lé, halmazzá stb. alakíthatjuk.

Lássuk, hogyan történik mindez a gyakorlatban a my_str_list = ['1', '2', '3', '4'] listával:

Az int() függvény alkalmazása minden egyes elemen a numerikus karakterekből álló lista, my_str_list-ben, hogy az elemeket egész számokká alakítsa, amelyekből a my_int_list egész számokat tartalmazó lista jön létre.

```
my_int_list = list(map(int, my_str_list))
print(my_int_list)  #[1, 2, 3, 4]
```

A float() függvény alkalmazása minden egyes elemen a numerikus karakterekből álló lista, my_str_list-ben, hogy az elemeket lebegőpontos számokká alakítsa, amelyekből a my_float_list lebegőpontos számokat tartalmazó lista jön létre.

```
my_float_list = list(map(float, my_str_list))
print(my_float_list) #[1.0, 2.0, 3.0, 4.0]
```

A str() függvény alkalmazása minden egyes elemen a lebegőpontos számokból álló lista, my_float_list-ben, hogy az elemeket karakterláncokká alakítsa, amelyekből a my_new_str_list karakterláncokból álló lista jön létre.

```
my_new_str_list = list(map(str, my_float_list))
print(my_new_str_list)  # ['1.0', '2.0', '3.0', '4.0']
```

Az int() függvény alkalmazása minden egyes elemen a numerikus karakterekből álló lista, my_str_list-ben, hogy az elemeket egész számokká alakítsa, amelyekből egy tuple, my_tuple, jön létre egész számokkal.

```
my_tuple = tuple(map(int, my_str_list))
print(my_tuple) #(1, 2, 3, 4)
```

A float() függvény alkalmazása minden egyes elemen a numerikus karakterekből álló lista, my_str_list-ben, hogy az elemeket lebegőpontos számokká alakítsa, amelyekből egy halmaz, my_set, jön létre lebegőpontos számokkal.

```
my_set = set(map(float, my_str_list))
print( my_set) # {1.0, 2.0, 3.0, 4.0}
```

A str() függvény alkalmazása minden egyes elemen a tuple-ból álló lista, my_tuple, hogy az elemeket numerikus karakterláncokká alakítsa, amelyekből a my_newer_str_list karakterláncokból álló lista jön létre.

```
my_newer_str_list = list(map(str, my_tuple))
print(my_newer_str_list) # ['1', '2', '3', '4']
```

Listák beillesztése

Egy számjegylistát az alábbi kóddal hozhatunk létre:

```
number_list = []
for number in range(1, 6):
    number_list.append(number)
print(number_list)
```

Ennek eredményeként az alábbi lista keletkezik:

```
[1, 2, 3, 4, 5]
```

De a Pythonban jellemzőbb, ha a lista létrehozása listakomprehenzióval (lista beillesztésével) történik. Az ilyen komprehenszió egyszerű formája így néz ki:

[expression for element in iterable]

Példa egy egész számokból készült lista létrehozására listák beillesztésével:

```
number_list = [number for number in range(1,6)]
print(number_list)
```

Ennek eredményeként ugyanazt a listát kapjuk, mint az előző esetben:

```
[1, 2, 3, 4, 5]
```

Ebben az esetben az első változó number (amely kifejezés) adja a number_list lista értékeit. A második number változó pedig a for ciklus része. Ha azt szeretnénk, hogy az első number változó egy kifejezés legyen, próbáljuk ki ezt a verziót:

```
number_list = [number-1 for number in range(1,6)]
print(number_list)
```

Ennek eredményeként egy olyan lista keletkezik, amely értékekkel csökkentve 1-tel:

```
[0, 1, 2, 3, 4]
```

A listák beillesztése tartalmazhat egy feltételes kifejezést is, amely körülbelül így néz ki:

[expression for element in iterable if condition]

Például készítsünk egy új komprehenziót, amely csak páratlan számokat tartalmaz a 1 és 5 közötti tartományban. Használjuk a number % 2 == 1 vagy number % 2 == 0 feltételeket a páratlan, illetve páros számok kiválasztására.

```
a_list = [number for number in range(1, 6) if number % 2 ==
1]
print(a_list)
```

Feltétel alkalmazásával a páratlan számok listáját kapjuk:

```
[1, 3, 5]
```

A listák beillesztése kompaktabb, mint a hagyományos módon írt kód (bár az eredmény ugyanaz):

```
a_list = []
for number in range(1, 6):
    if number % 2 == 1:
        a_list.append(number)
print(a_list)
```

Szótárak beillesztése

Szótárak esetén is létrehozhatunk listakomprehenziót. A legegyszerűbb formája így néz ki:

{key: value for expression in iterable}

A listakomprehenzióhoz hasonlóan a szótári komprehenziókban is szerepelhetnek if feltételek és for ciklusok:

```
word = 'Antarctica'
letter_counts = {letter: word.count(letter) for letter in
word}
print(letter_counts)
```

A ciklusban, miközben minden egyes karakteren végigmegyünk az Antarctica szónál, kiszámítjuk, hányszor fordul elő az aktuális betű. A word.count(letter) parancs többször is meghívódik (ez extra időt vesz igénybe), mivel az a, c és t betűk kétszer szerepelnek. Azonban ez nem befolyásolja a szótár tartalmát, mivel amikor ezekkel a betűkkel másodszor találkozunk, az eredeti bejegyzést felülírja az aktuális érték:

```
{'i': 1, 'c': 2, 'a': 2, 'n': 1, 'r': 1, 'A': 1, 't': 2}
```

Generátorok

A Pythonban a generátor egy objektum, amely sorozatok létrehozására szolgál. A generátorok segítségével nagy sorozatokon lehet végighaladni (iterálni), anélkül hogy az egész sorozatot egyszerre kellene létrehozni és eltárolni a memóriában.

A generátorok gyakran adatforrást biztosítanak iterátorok számára.

Az alábbi példában a range() generátorfunkciót használjuk, amely egy egész számokból álló sorozatot generál, és kiszámítjuk az 1-től 50-ig terjedő számok összegét a sum() függvény segítségével:

```
print(sum(range(1,51))) # 1275
```

4. Függvények

A függvények nevezett kódrészletek, amelyek egy adott feladat megoldására szolgálnak. Ha egy feladatot többször kell megoldani a programban, nem szükséges újra megírni az egész kódot - egyszerűen meghívjuk a függvényt, amely a feladat megoldására van kijelölve, és a meghívás utasítja a Pythont, hogy hajtsa végre a függvényben található kódot.

A függvény egy mini-program a fő programon belül.

Függvények meghatározása és meghívása

A Python beépített függvényeket biztosít, mint például a len(), print(), input(), stb. Azonban saját függvényeket is létrehozhatunk.

A függvények használata megkönnyíti a kód olvasását, írását, tesztelését és a hibák javítását. A függvények bármilyen számú bemeneti paramétert fogadhatnak, és bármilyen típusú objektumot (listák, tuple-k stb.) adhatnak vissza eredményként. A függvények fő előnye a kód újrahasználhatósága.

A függvényt:

- meghatározhatjuk
- meghívhatjuk

A függvény meghatározásához használjuk a **def kulcsszót**, *majd a függvény nevét, a bemeneti paramétereket zárójelekben, és egy kettőspontot, amely után a kódot behúzva adjuk meg:*

```
def fuggveny_nev(bemeneti_parameterek):
    kód_részlet
```

A függvények nevei ugyanazokat a szabályokat követik, mint a változónevek: a neveknek betűvel vagy aláhúzásjellel kell kezdődniük, és csak betűket, számokat és aláhúzásjeleket tartalmazhatnak.

Definiáljunk egy függvényt bemeneti paraméterek nélkül, amely egy üzenetet ír ki a képernyőre:

```
def birthday_card():
    print('Boldog születésnapot!')
```

A függvény meghívásához egyszerűen írjuk be a nevét zárójelekkel: birthday_card(). Amikor meghívjuk a birthday_card() függvényt, a Python végrehajtja a

függvényen belüli kódot, kiírja a "Boldog születésnapot!" üzenetet, és visszatér a fő programhoz.

Most írjunk egy másik függvényt bemeneti paraméterek nélkül, amely True értéket ad vissza, és hívjuk meg a függvényt egy if utasításban annak ellenőrzésére, hogy mit ad vissza a függvény:

```
# függvény meghatározása
def hungry():
    return True

# függvény meghívása és feltétel ellenőrzése
if hungry():
    print('Egyél zabpelyhet!')
else:
    print('Igyál vizet!')
```

Ez a kombináció a függvény és a feltétel vizsgálatával a következő üzenetet adja vissza: **Egyél zabpelyhet!**

A függvényekhez átadott értékeket argumentumoknak nevezzük. Amikor egy függvényt meghívunk argumentumokkal, az argumentumok értékei másolják a megfelelő paramétereket a függvényen belül.

Most definiáljunk egy új függvényt, amely üzenetet ad vissza a választott szín alapján:

```
def like(interest):
    if interest == 'yoga':
        return "I quite like " + interest + "."
    elif interest == "football":
        return "Yes, I play " + interest + "."
    elif interest == 'guitar':
        return "Yes, I play the " + interest + "."
    else:
        return "I'm interested in " + interest + "."
story = like('photography')
print(story)
story = like('yoga')
print(story)
story = like('football')
print(story)
story = like('guitar')
print(story)
```

A like() függvény az alábbi lépéseket hajtja végre:

1. Az interest paraméterhez hozzárendeli az értéket: 'photography' ('yoga', 'football', 'guitar').

2. Végigmegy az if-elif-else láncon.
3. Visszaad egy üzenet szöveget.
4. A visszaadott szöveget hozzárendeli a story változóhoz.

A print(story) parancs végrehajtása után az alábbi üzenetek jelennek meg:

- I'm interested in photography.
- I quite like yoga.
- Yes, I play football.
- Yes, I play the guitar.

Érték: None

Például, definiáljunk egy függvényt, amely nem fogad paramétereket:

```
def do_nothing():
    pass
```

A pass egy helyettesítő operátor, ami lényegében a művelet hiányát jelzi. Ebben az esetben a pass azt jelenti Python számára, hogy a függvény nem végez semmilyen műveletet. Ha meghívjuk ezt a függvényt:

```
do_nothing()
```

A függvény lefut, de nem jelenik meg semmi a képernyőn.

Ha a függvényt a print() függvénnyel hívjuk meg:

```
print(do_nothing())
```

A kimenet az alábbi lesz:

```
None
```

Ha egy függvény nem használ return operátort, akkor alapértelmezetten None értéket ad vissza. A None egy speciális konstans, amely az érték hiányát jelenti, tehát a "semmit" vagy "üres helyet". A None nem egyenlő a False logikai értékkel.

Írjunk egy egyszerű függvényt, amely ellenőrzi, hogy egy érték None:

```
def is_none(thing):
    if thing is None:
        print("It's None")
    else:
        print("It's not None")
```

és annak tesztelése:

```
is_none(None)  # It's None
is_none(True)  # It's True
is_none(False) # It's False
is_none(0)     # It's False
is_none(0.0)   # It's False
is_none(())    # It's False
is_none([])    # It's False
is_none({})    # It's False
is_none(set()) # It's False
```

Pozíciós és névvel ellátott argumentumok

Amikor függvényt definiálunk, több paramétert is megadhatunk, és a függvény meghívásakor több argumentumot is átadhatunk neki. Többféle módon is átadhatunk argumentumokat a függvényeknek.

A legelterjedtebb argumentumtípusok a *pozíciós argumentumok*, amelyek értékei a paraméterekhez a sorrendjük szerint kerülnek hozzárendelésre. Ezért fontos, hogy minden argumentum pozícióját tartsuk szem előtt.

Definiáljunk egy függvényt, amely létrehoz és visszaad egy szótárat a bemeneti pozíciós argumentumokból:

```
def music(terminology, musician, genre):
    return {'terminology': terminology, 'musician':
musician, 'genre': genre}
```

Ebben az esetben a music függvény definíciójában a terminology, musician, genre értékek a függvény paraméterei.

Most hívjuk meg a függvényt az alábbi parancs segítségével:

```
print(music('tune', 'guitarist', 'blues'))
```

A tune, guitarist, blues értékek a függvény meghívásakor pozíciós argumentumokként kerülnek átadásra, és ezek az értékek a megfelelő paraméterekhez lesznek hozzárendelve. A függvény egy szótárat ad vissza:

```
{'terminology': 'tune', 'musician': 'guitarist', 'genre':
'blues'}
```

Az argumentumokat a függvény meghívásakor névvel ellátott argumentumokkal is megadhatjuk.

Alapértelmezett értékek

Minden függvény paraméteréhez alapértelmezett értéket rendelhetünk hozzá. Ha a függvény meghívásakor olyan argumentumot adunk át, amely megfelel ennek a paraméternek, akkor Python az argumentum értékét fogja használni, ha pedig nem, akkor az alapértelmezett értéket. Így ha a paraméterhez alapértelmezett érték van hozzárendelve, akkor az ehhez tartozó argumentumot kihagyhatjuk a függvény meghívásakor.

Definiáljunk egy függvényt, amely információt ad egy házikedvencről:

```
def describe_pet(pet_name, animal_type='parrot'):
    print("I have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " +
pet_name.title() + ".")
```

A describe_pet() függvény definíciójában szerepel az animal_type paraméter, amely alapértelmezett értéke 'parrot'. Ha most a függvényt úgy hívjuk meg, hogy csak a pet_name argumentumot adjuk át (például pet_name='Jack Sparrow), de az animal_type argumentumot kihagyjuk:

```
describe_pet(pet_name='Jack Sparrow')
```

Python tudni fogja, hogy az animal_type paraméterhez az alapértelmezett 'parrot' értéket kell használni:

```
I have a parrot.
My parrot's name is Jack Sparrow.
```

Ha másféle házikedvencet szeretnénk megadni (például egy másik állatot, mint a papagáj), akkor az animal_type értékét egyértelműen megadhatjuk a függvény meghívásakor:

```
describe_pet(pet_name='Peppa', animal_type='guinea pig')
```

Mivel az animal_type paraméterhez most egyértelműen átadjuk az argumentumot ('guinea pig'), Python figyelmen kívül hagyja az alapértelmezett értéket:

```
I have a guinea pig.
My guinea pig's name is Peppa.
```

Az * és ** szimbólumok használata az argumentumokkal

Ha a * szimbólumot használjuk a függvény paramétereként, akkor tetszőleges számú pozíciós

argumentumot egy tuple-ben fogunk csoportosítani. Az alábbi példában a print_days() függvény definíciója:

```
def print_days(*args):
    print('Get ready:', args)
```

Az args egy tuple, amely tartalmazza az összes argumentumot, amelyet átadunk a print_days() függvénynek:

```
print_days('Wednesday', 'Thursday', 'Friday', 'Weekend...')
```

A kimenet a következő lesz:

```
Get ready: ('Wednesday', 'Thursday', 'Friday',
'Weekend...')
```

Ez hasznos lehet olyan függvények írásakor, mint a print(), amelyek tetszőleges számú argumentumot fogadnak. Például, ha a függvény:

```
def print_travel(required1, required2, *args):
    print('For train travel is required:', required1)
    print('For train travel is required too:', required2)
    print('All the rest:', args)
```

valamint kötelező paraméterek is vannak (ebben az esetben required1 és required2), akkor ezekbe a paraméterekbe át lesznek másolva a pozíciós argumentumok értékei ('return ticket', 'train') a függvény meghívásakor:

```
print_travel('return ticket', 'train', 'carriage', 'seat',
'luggage rack')
```

Az *args tuple a többi argumentumot tartalmazza, kezdve a 'carriage' értékkel:

```
For train travel is required: return ticket
For train travel is required too: train
All the rest: ('carriage', 'seat', 'luggage rack')
```

A ** szimbólumot is használhatjuk, hogy a névvel ellátott argumentumokat egy szótárban csoportosítsuk, ahol az argumentumok nevei lesznek a kulcsok, az értékek pedig a szótár megfelelő értékei.

Az alábbi példában definiáljuk a print_character() függvényt:

```
def print_character(**kwargs):
    print('Person\'s characteristics:', kwargs)
```

Ha a függvényt névvel ellátott argumentumokkal hívjuk meg:

```
print_character(emotions='cheerful', sense='happy',
look='slim')
```

A kimenet az alábbiak szerint jelenik meg, ahol a névvel ellátott argumentumokat szótár formájában látjuk:

```
Person's characteristics: {'emotions': 'cheerful', 'sense':
'happy', 'look': 'slim'}
```

A * szimbólum használatakor nem szükséges kifejezetten args-nak nevezni a paramétert, és a ** szimbólum esetében sem szükséges kwargs-nak nevezni a paramétert, bár ez a gyakorlatban gyakori Python-ban.

Névtelen függvények: lambda utasítás

A Python lehetővé teszi, hogy rövid formában hozzunk létre kis névtelen függvényeket, úgynevezett lambda függvényeket. Ezek ugyanúgy működnek, mint a hagyományos függvények, amelyeket a def kulcsszóval definiálunk. A névtelen függvények csak egy kifejezést tartalmazhatnak, de gyorsabban is végrehajtódnak.

Gyakran ilyen kis függvényeket kell más függvényeknek átadni, például olyan esetekben, amikor egy kulcs függvényt használunk lista rendezésére. Ilyen esetekben nem szükséges nagy függvények, és kényelmetlen lenne egy másik helyen definiálni a függvényt.

A névtelen függvényeket a **lambda utasítással** hozhatjuk létre:

```
add = lambda x, y: x + y
```

Ugyanaz a add függvény definiálható a def kulcsszóval is:

```
def add(x, y):
    return x + y
```

Ha végrehajtjuk a kódot:

```
print(add(7, 8))
```

Mindkét esetben ugyanazt az eredményt kapjuk:

```
15
```

Ezen kívül nem szükséges, hogy a névtelen függvényt változóhoz rendeljük:

```
print((lambda x, y: x + y)(5, 12))
```

És a lambda függvényeknél, ellentétben a hagyományos függvényekkel, nincs szükség return utasításra sem:

17

A lambda függvény végrehajtásakor a kifejezés kiszámításra kerül, majd az eredmény automatikusan visszatér, ezért mindig létezik egy implicit return utasítás.

Mikor érdemes lambda függvényeket alkalmazni a programozási kódban? Nagyon gyakran, ahogy fentebb említettük, lambda függvényeket használnak rövid függvények írására, amelyek egy alternatív kulcs szerint rendezik az objektumokat:

```
months = [(1, 'January'), (9, 'September'), (7, 'July'),
(4, 'March')]
print(sorted(months, key=lambda x: x[1]))
print(sorted(months))
```

A fenti példában egy lista tuple-öket rendezünk a második értékük alapján. Ebben az esetben a lambda függvény gyors módot biztosít a rendezési sorrend megváltoztatására:

```
[(1, 'January'), (7, 'July'), (4, 'March'), (9,
'September')]
[(1, 'January'), (4, 'March'), (7, 'July'), (9,
'September')]
```

5. Kivétel (hiba) kezelés

Kivételes helyzetek

A Pythonban a program végrehajtása során felmerülő hibák kezelésére speciális objektumokat használnak, amelyeket **kivételnek** neveznek. Ha hiba lép fel, és a Python nem tudja, mit kellene tenni, létrejön egy kivétel objektum. Ha a programban van kivételkezelő kód, a program végrehajtása folytatódik, különben a program leáll, és hibajelentést ad a végrehajtási nyomvonalról.

Hasonlót tapasztalhatunk, amikor megpróbálunk hozzáférni egy olyan elemhez, amely nincs a listában vagy a tuple-ben, vagy egy nem létező kulccsal próbálunk értéket kinyerni egy szótárból. Ha olyan kódot hajtunk végre, amely bizonyos körülmények között nem működik, **kivételkezelőket** használunk annak érdekében, hogy elkapjuk az esetleges hibákat.

Jó gyakorlat minden olyan helyen használni a kivételkezelőket, ahol kivétel keletkezhet, hogy a felhasználó tisztában legyen azzal, mi történik. Lehet, **hogy nem tudjuk kijavítani a hibát, de legalább megtudhatjuk, hogy milyen körülmények között történt, és tisztán befejezhetjük a programot. Saját kivételobjektumokat is létrehozhatunk.**

try-except blokk

Ha nem használunk kivételkezelőket, a Python hibaüzenetet ad, és némi információt arról, hogy hol történt a hiba, majd leállítja a programot. Az alábbi kódrészlet ezt mutatja:

```
short_list = [1, 2, 3]
position = 5
print(short_list[position])
```

IndexOutOfRange hiba lép fel:

```
Traceback (most recent call last):
  File "C:\Python34\test.py", line 246, in <module>
    print(short_list[position])
IndexError: list index out of range
```

Helyezzük a kódunkat a try blokkba, és használjunk egy except blokkot a hiba kezelésére:

```
short_list = [1, 2, 3]
position = 5
try:
```

```
    short_list[position]
except:
    print('Need a position between 0 and', len(short_list)-
1, 'but got', position)
```

A program futtatásakor az alábbi üzenetet kapjuk:

```
Need a position between 0 and 2 but got 5
```

Saját kivétel létrehozása

Bármely kivétel egy osztály, különösen az Exception osztály utódja. Hozzunk létre egy kivételt, amelyet **IsNotTitleException** névre nevezünk. A kivételt akkor dobjuk, ha egy lista elemének első betűje kisbetűvel van írva:

```
class IsNotTitleException(Exception):
    pass

rooms = ['Kitchen', 'study', 'Hall', 'Bathroom']
for room in rooms:
    if room.title() != room:
        raise IsNotTitleException(room)
```

Mivel az IsNotTitleException kivétel viselkedését nem definiáltuk (használtuk a pass kulcsszót), a szülőosztály, az Exception végrehajtja a kivételüzenet megjelenítését a generálásakor:

```
Traceback (most recent call last):
    ..., line 7, in <module>
    raise IsNotTitleException(room)
__main__.IsNotTitleException: study
```

De meghatározhatjuk, hogy mit kell kiírnia a saját kivételnek, ha azt kényszerítve dobjuk a raise kulcsszóval:

```
class IsNotTitleException(Exception):
    pass
try:
    rooms = ['Kitchen', 'study', 'Hall', 'Bathroom']
    for room in rooms:
        if room.title() != room:
            raise IsNotTitleException('Fault!')

except IsNotTitleException as exc:
    print(exc)
```

A kód futtatásakor a következő eredményt kapjuk:

```
Fault!
```

6. Modulok és csomagok

Számos programozási nyelvben létezik a könyvtárak vagy újrahasználható kódrészletek fogalma. A Python is rendelkezik egy nagy választékú könyvtárral, amely megkönnyíti a kódok kezelését. Az alapértelmezett Python disztribúció tartalmaz egy teljes körű standard könyvtárat, amely lehetővé teszi, hogy számos feladatot végezzünk el anélkül, hogy további könyvtárakat kellene telepíteni.

Ami a Python standard könyvtárának számít, az több összetevőből áll, ideértve a beépített adattípusokat és konstansokat, amelyeket importálás nélkül használhatunk, mint például a számok és listák.

Bármely Python program számára elérhető az alapértelmezett beépített függvények halmaza, amelyek közé tartoznak olyan függvények, mint a print(), input(), len(), amelyeket "dobozból" használhatunk a programunkban.

De a standard könyvtár legnagyobb része számos modult tartalmaz, amelyek különböző adattípusok kezelésére, operációs rendszerrel való interakcióra, szerverek és kliensek írására számos internetes protokollhoz, valamint kódunk fejlesztésére és hibakeresésére szolgálnak. Például a math modul matematikai függvényeket tartalmaz, a random modul pedig véletlenszám-generálással kapcsolatos függvényeket.

Mi az a modul?

A modul egy .py kiterjesztésű fájl, amely Python kódot tartalmaz. Meghatározza a Python függvények vagy más objektumok csoportját, és a modul neve megegyezik a fájl nevével. Funkcionálisan a modul nem különbözik egy szokásos Python fájltól, de célja, hogy lehetővé tegye a függvények, osztályok vagy változók újrahasználatát a program különböző részeiben.

A Python számos beépített modullal rendelkezik, mint például a **math**, **os**, amelyek közvetlenül elérhetők a Python telepítése után. Ha egy fájlt modulként használunk, annak kódja csak akkor fut le, amikor importálják, és csak egyszer. A modulok gyakran tartalmaznak alap Python kódot, de tartalmazhatnak C és C++ nyelven lefordított objektum fájlokat is. A lefordított modulok és a Python alapú modulok ugyanúgy működnek.

A Python standard függvényeinek nagy része nem a nyelv alapmagjába van beépítve, hanem külön modullal van biztosítva, amelyek szükség szerint betöltődnek.

A modulok nemcsak a Python összefüggő objektumainak csoportosítására szolgálnak, hanem segítenek elkerülni a névütközéseket is. Például, ha egy *firstmodule* nevű modult írsz, amely definiál egy *rounded* nevű függvényt, ugyanabban a programban lehet egy *secondmodule* nevű modul is, amely szintén definiál egy *rounded* nevű függvényt, de az egy másféle műveletet hajt végre. A Python modulok segítségével két különböző függvényt is használhatunk ugyanazzal a rounded névvel: egyszerűen hivatkozhatunk a függvényekre a programban a **firstmodule.rounded** és **secondmodule.rounded** nevekkel.

A modulnevek használata segít elválasztani a két rounded függvényt egymástól. Mivel minden modul saját névtérrel rendelkezik, így megakadályozza a névütközéseket.

Modulok importálása: az import utasítás

A modul használatához le kell tölteni a modul kódját a programod névtérbe. A modulban található függvények használatához importálni kell azt a programba az **import utasítás** segítségével. A import utasítás szokásos használati formája:

```
import modul
```

ahol modul egy Python fájl neve kiterjesztés nélkül (.py). Miután a modult importáltad, használhatod annak bármelyik függvényét.

Például, hogy importáld a *nameofmodule* nevű modult a névtérbe, használd az alábbi parancsot:

```
import nameofmodule
```

Ezáltal a névtérben létrejön egy új változó, nameofmodule, amely a modulra mutat.

Importálási szabályok (A random modul példáján)

A random modul olyan függvényeket biztosít, amelyek segítségével véletlenszerű számokat, betűket generálhatunk, illetve véletlenszerű elemeket választhatunk egy szekvenciából. A modul pszeudóvéletlenszám-generátorai nem alkalmasak kriptográfiai célokra.

A random modul néhány gyakran használt függvénye:

Függvény hívás	Leírás
random.randrange(start, stop, step)	Véletlenszerű egész szám a start és stop közötti tartományban
random.randint(A, B)	Véletlenszerű egész szám N (A ≤ N ≤ B)
random.choice(sequence)	Véletlenszerű elem a nem üres sequence szekvenciából
random.sample(population, k)	K hosszúságú lista a population szekvenciából
random.random()	Véletlenszerű szám 0 és 1 között

Nézzük meg, hogyan működik a random modul, különösen a randint() függvény használatát. Írj be az alábbi kódot egy fájlba, és mentsd el egy tetszőleges néven:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

A randint() függvény véletlenszerű egész számot ad vissza a két megadott egész szám között. A program futtatásakor öt véletlenszerű egész számot kell kapnod 1 és 10 között (beleértve a szélső értékeket):

```
7
6
2
3
6
```

Mivel a randint() függvény a random modulban található, a modul nevét kell használni prefiksként (ponttal elválasztva) a függvény neve előtt, hogy a Python tudja, hogy ezt a függvényt a random modulban kell keresni.

Ha több modult kell importálni, a következő szintaxist használjuk:

```
import modul1, modul2, modul3, ...
```

Vagy importálhatjuk őket külön-külön is:

```
import modul1
import modul2
import modul3, modul4
```

Az összes import utasítást célszerű a fájl tetejére helyezni. Mostantól bármelyik függvényt használhatjuk a fenti modulokból.

Az import utasítás alternatív formája

Az import utasításnak van egy másik formája is:

```
from modul import függvény1, függvény2, ...
```

Ebben az esetben nem az egész modult importáljuk, hanem csak a modulban található konkrét függvényeket. Így a kódban már nem szükséges a modul nevét prefikskként használni a függvények előtt:

```
from random import randint, random
for i in range(5):
    print(randint(1, 10))
print(random())
```

A program kimenete:

```
8
7
3
6
10
0.12147106708747124  # random() függvény eredménye
```

A random() függvényt argumentumok nélkül hívjuk meg, és egy véletlenszerű valós számot generál 0 és 1 között (a tartomány szélsőértékei nem szerepelnek).

Biztonságosabb a függvények használata modul névvel, mint például random.randint(). Ezért a legjobb, ha a szokásos import utasítást használjuk:

```
import modul_neve
```

A * használata importáláskor

Importáláskor használhatjuk a * szimbólumot is:

```
from random import *
```

Ez a parancs azt jelenti, hogy az összes függvényt importáljuk a random modulból. Óvatosnak kell lenni az "importálj mindent" formával. Ha két modulban ugyanazt a függvénynevet importáljuk, névütközés léphet fel, és a második modulban található függvény felülírja az elsőt. Ezenkívül nehezebb lesz nyomon követni, honnan származnak a használt függvények.

Pszedonimák használata

A modulok (vagy azok függvényei) importálásakor rövidíthetlük a neveket *álnevek* használatával. Az ilyen használat formája:

```
import modul as álneve # álnev használata a modulhoz
from modul import függvény as álneve # álnev a függvényhez
```

Például, ha a random modulhoz az rn álnevet használjuk, akkor a véletlenszerű egész számok generálása így néz ki:

```
import random as rn
for i in range(5):
    print(rn.randint(1, 10))
```

math modul

A Python standard könyvtár math modulja sok matematikai függvényt tartalmaz. A math modul leggyakrabban használt függvényei és konstansai az alábbi táblázatban találhatjuk:

Függvény/konstans hívása	Leírás/érték
math.sin(x)	Sinusz x (radiánban)
math.cos(x)	Koszinusz x (radiánban)
math.radians(x)	x (fokban) átváltása radiánná
math.degrees(x)	x (radiánban) átváltása fokokká
math.exp(x)	Exponenciális függvény x (e^x)
math.sqrt(x)	Négyzetgyök x-ből
math.factorial(x)	x faktoriálisa (egész szám)
math.e	2.718281828459045
math.pi	3.141592653589793

A modulhoz való hozzáféréshez importálni kell a math könyvtárat az alábbi paranccsal:

```
import math
```

Használjuk a Python interaktív módot a könyvtár felfedezésére. A math könyvtár olyan konstansokat tartalmaz, mint a pi (Pi szám) és e (exponenciális szám):

```
import math
math.pi # 3.141592653589793
math.e # 2.718281828459045
```

Nézzünk meg néhány hasznos függvényt a math könyvtárból:

```
math.fabs(-221.1)  # Visszaadja az abszolút értéket.
221.1
math.floor(56.8)  # Lefelé kerekítés.
56
math.ceil(38.3)  # Felfelé kerekítés.
39
math.factorial(7)  # Faktoriális számítása.
5040
math.pow(4, 3)  # Hatványozás.
64.0
math.sqrt(256)  # Négyzetgyök számítása.
16.0
math.radians(180)  # Fokban lévő érték átváltása radiánná.
3.141592653589793
math.degrees(math.pi)  # Radiánban lévő érték átváltása
fokokká.
180.0
```

Ahhoz, hogy megismerjük a math modul összes függvényét, használhatjuk a **dir() függvényt:**

```
import math
dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

A fenti parancs eredményeként egy lista jelenik meg, amely az math objektum attribútumait tartalmazza, ezek többsége függvény. A dir() függvény visszaadja az objektum attribútumainak listáját, ábécé sorrendben. Az attribútumok (vagy metódusok) olyan függvények, amelyek az objektumokkal kapcsolatosak.

Azok az attribútumok (vagy metódusok), amelyek dupla aláhúzással kezdődnek és végződnek (__), **speciális vagy mágikus metódusok.** A speciális metódusok meghatározzák, mi történik a belső implementációban, amikor műveleteket hajtunk végre objektumokkal.

Például, ha a factorial() függvényt szeretnénk meghívni 3 értékre, ezt nem tehetjük meg közvetlenül, mivel a factorial nem része a névtérnek – csak a math változó van jelen a névtérben. Azonban megkereshetjük a factorial attribútumot a math változóban a pont operátor segítségével

(ez az operátor lehetővé teszi az objektum attribútumainak átnézését):

```
math.factorial(3)
6
```

Ha szeretnénk elolvasni a factorial() függvény dokumentációját, használhatjuk a **help() függvényt**. A help() függvény megjeleníti a dokumentációt egy metódus, modul, osztály, függvény stb. számára.

Ha kíváncsiak vagyunk arra, hogy mit csinál a pow attribútum a math modulban, akkor:

```
import math
help(math.pow)
Help on built-in function pow in module math:
pow(...)
    pow(x, y)
    Return x**y (x to the power of y).
```

Megjegyzés

A Python-ban a **csomagok** és a **modulok** két különböző fogalom, de szorosan összefüggenek egymással. A következő különbségek és kapcsolatok segítenek megérteni őket:

1. Modul

- A **modul** a Python programozási nyelvben egy fájl, amely Python kódot tartalmaz. Ez lehet például egy egyszerű .py kiterjesztésű fájl, amely funkciókat, osztályokat, változókat és más Python kódokat tartalmaz.
- A modulok lehetővé teszik a kódok újbóli felhasználását, mivel egy modulban lévő funkciókat vagy osztályokat más Python fájlokban importálhatunk.
- Például egy fájl math.py egy modul, ami tartalmazhat olyan funkciókat, mint add(), subtract(), stb.

Példa egy modulra:

```
# matek.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

A fenti matek.py fájlt egy másik fájlban így importálhatjuk:

```
import matek

print(matek.add(2, 3))  # 5
```

2. Csomag

- A **csomag** egy olyan **mappa (könyvtár), amely több modult tartalmaz.** Egy csomag célja, hogy a modulok logikai csoportosítását lehetővé tegye, és a kódot rendszerezettebbé tegye.
- A csomagoknak szükségük van egy speciális fájlra, amelyet **__init__.py**-nak hívunk, hogy Python felismerje a mappát csomagként.
- A csomagok tartalmazhatnak alcsomagokat (alcsomagok: más mappák, amelyek szintén csomagok), így lehetőség van hierarchikus struktúrák kialakítására.

Példa egy csomagra:

Struktúra:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

A my_package/__init__.py fájl azt mondja a Python-nak, hogy az adott mappa csomagként kezelendő.

A csomagon belüli modulok importálása:

```
from my_package import module1
from my_package.module2 import some_function
```

Összegzés: Különbségek és kapcsolatok

- **Modul:** Egy Python fájl (pl. math.py), amely Python kódot tartalmaz.
- **Csomag:** Egy mappa, amely több modult tartalmazhat, és egy __init__.py fájlt kell tartalmaznia ahhoz, hogy Python csomagként kezelje.

A csomagok segítenek abban, hogy a modulokat jól strukturáljuk, míg a modulok a kód újrafelhasználását és egyszerűsítését szolgálják.

7. Python fontosabb moduljai

A Python legfontosabb moduljait bonyolultságuk növekvő sorrendjében sorolhatjuk be. A következő lista a modulok használatának fokozatosan növekvő komplexitását és hasznosságát tükrözi:

- **Math (Bonyolultság:** Alacsony)

A math modul alapvető matematikai funkciókat kínál, mint például a szinusz, koszinusz, négyzetgyök, és faktoriális számítások. Egyszerűen használható és alapvető számítási műveletekhez szükséges.

- **Random (Bonyolultság:** Alacsony)

A random modul a véletlenszerű számok generálására szolgál. Tartalmaz függvényeket, amelyek segítenek véletlenszerű számok, elemek kiválasztásában vagy minták létrehozásában.

- **Datetime (Bonyolultság:** Alacsony-közepes)

A datetime modul a dátumok és időpontok kezelésére szolgál. Segít dátumok és időpontok formázásában, manipulálásában és összehasonlításában.

- **Tkinter (Bonyolultság:** Közepes)

A Tkinter a Python beépített könyvtára, amely lehetővé teszi grafikus felhasználói felületek (GUI) létrehozását. A Tkinter-t használhatjuk egyszerű alkalmazásokhoz, mint például szövegdobozok, gombok, listák, címkék és egyéb alapvető GUI elemek.

- **Os (Bonyolultság:** Közepes)

Az os modul lehetővé teszi a fájlrendszer elérését, a fájlok és könyvtárak kezelését, valamint a környezeti változók elérését. Emellett elérhetők az operációs rendszeren végrehajtható parancsok és azok futtatása.

- **Sys (Bonyolultság:** Közepes)

A sys modul a Python interpreter és annak működését befolyásoló információkat biztosít. Emellett lehetővé teszi a parancssori argumentumok elérését és a program futásának kezelését.

- **Json (Bonyolultság:** Közepes)

A json modul a JSON formátumú adatok betöltésére és mentésére szolgál. Szöveges alapú adatcsere formátumot biztosít, amelyet széles körben használnak API-k és webszolgáltatások esetén.

- **Re (Bonyolultság**: Közepes)

A re modul reguláris kifejezésekkel dolgozik, amelyek segítségével bonyolult szövegkeresési és szövegmanipulációs műveletek végezhetők.

- **Requests (Bonyolultság**: Közepes)

A requests modul lehetővé teszi HTTP kérések egyszerű végrehajtását. Nagyon hasznos API hívásokhoz, weboldalak letöltéséhez, és adatgyűjtéshez.

- **Collections (Bonyolultság**: Közepes)

A collections modul különféle speciális adattípusokat tartalmaz, például a Counter (amely a frekvenciát számolja), deque (kettős végű sor), és más hasznos adatstruktúrákat.

- **SQL (Bonyolultság**: Magas)

Az SQL (Structured Query Language) az adatbázisok kezelésére szolgáló nyelv, amelyet gyakran használák Pythonban adatbázisokkal való interakcióra.

- **Pandas (Bonyolultság**: Magas)

A pandas modul az adatkezelés és -elemzés egyik legfontosabb eszköze. Többdimenziós adatstruktúrákat és különböző analitikai függvényeket kínál.

- **Numpy (Bonyolultság**: Magas)

A numpy modul a numerikus számításokhoz készült, különösen a tömbök kezelésére. A mátrixok, tömbök és lineáris algebra műveletek elvégzésére használják, amely nélkülözhetetlen a tudományos és mérnöki alkalmazásokban.

- **Matplotlib (Bonyolultság**: Magas)

A matplotlib modul grafikus ábrázolásra szolgál, és adatvizualizációs lehetőségeket kínál különböző diagramok és grafikonok készítésére.

- **flask/django (Bonyolultság**: Nagyon magas)

A flask és django keretrendszerek a webfejlesztéshez használhatók. A flask egy könnyű, egyszerűbb webes keretrendszer, míg a django sokkal nagyobb és komplexebb, több beépített funkcióval rendelkezik, például az adatbázis kezeléssel, felhasználói autentikációval és adminisztrációs felülettel.

- **Asyncio (Bonyolultság**: Nagyon magas)

  Az asyncio modul az aszinkron programozás kezelésére szolgál. Ez lehetővé teszi a párhuzamos feladatok kezelését egyetlen szálon, különösen hasznos a hálózati alkalmazásoknál.

- **tensorflow/torch (Bonyolultság**: Nagyon magas)

A tensorflow és torch könyvtárak a mesterséges intelligencia és gépi tanulás fejlesztésére szolgálnak. Komplex matematikai modellek építésére, optimalizálásra és tréningezésre használják őket.

Ezek a modulok különböző feladatok elvégzésére alkalmasak, és a bonyolultságuk fokozatosan növekszik, ahogy egyre összetettebb problémákat próbálunk megoldani. Az alapvető modulok a matematikai számításoktól a fájlkezelésig terjednek, míg a fejlettebbek webes alkalmazások fejlesztésére vagy tudományos számításokra szolgálnak.

Egy kicsit bövebben a **Tkinter** és az **SQL** alkalmazásáról. A **Tkinter** és az **SQL** gyakran is fontos részei lehetnek egy Python programnak, különösen ha grafikus felhasználói felületet (GUI) és adatbázis-kezelést akarunk integrálni.

A **Tkinter** a Python beépített könyvtára, amely lehetővé teszi grafikus felhasználói felületek (GUI) létrehozását. A Tkinter elemeket, mint például szövegdobozok, gombok, listák, címkék és egyéb alapvető elemeket, használhatjuk egyszerű alkalmazások létrehozására.

Alapvető elemek:

- **Gombok** (Button)
- **Szövegdobozok** (Entry, Text)
- **Címkék** (Label)
- **Listák** (Listbox, ComboBox)
- **Kép kezelés** (Canvas)

Például, ha egy egyszerű alkalmazást szeretnénk, amely ablakot nyit és egy gombot tartalmaz, akkor így nézhet ki:

```
import tkinter as tk

def hello():
    label.config(text="Helló, világ!")

# Ablak létrehozása
root = tk.Tk()
root.title("Egyszerű Tkinter alkalmazás")

# Címke
label = tk.Label(root, text="Kattints a gombra!")
label.pack()

# Gomb
button = tk.Button(root, text="Kattints ide",
command=hello)
button.pack()

# A fő ciklus indítása
root.mainloop()
```

Ez egy alapvető alkalmazás, de a Tkinter segítségével fejlettebb GUI-kat is készíthetsz, amely képes kezelni különböző eseményeket, űrlapokat, és interakciókat.

Az SQL (Structured Query Language) az adatbázisok kezelésére szolgáló nyelv, amelyet gyakran használunk Pythonban adatbázisokkal való interakcióra. A Pythonhoz több könyvtár is elérhető, mint például az **SQLite** (amely a beépített sqlite3 modullal érhető el), **MySQL** (MySQLdb vagy mysql-connector), és **PostgreSQL** (psycopg2).

A legelterjedtebb megoldás kezdők számára az **SQLite**, mivel ez egy fájlbázisú adatbázis, ami nem igényel külön telepítést vagy adatbázis-kiszolgálót. Az SQL parancsok (pl. SELECT, INSERT, UPDATE, DELETE) lehetővé teszik az adatok lekérdezését, módosítását és tárolását.

Példa egy SQLite adatbázis létrehozására és adatlekérdezésre Pythonban:

```
import sqlite3

# Kapcsolódás egy adatbázishoz (ha nem létezik, akkor
létrejön)
conn = sqlite3.connect('mydatabase.db')
# Adatbázis kurzor létrehozása
cursor = conn.cursor()
# Táblázat létrehozása
cursor.execute('''CREATE TABLE IF NOT EXISTS users (id
INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')
# Adatok beszúrása
cursor.execute('''INSERT INTO users (name, age) VALUES (?,
?)''', ('Anna', 17))
cursor.execute('''INSERT INTO users (name, age) VALUES (?,
?)''', ('Laci', 18))
# Módosítások mentése
conn.commit()
# Adatok lekérdezése
cursor.execute('''SELECT * FROM users''')
rows = cursor.fetchall()
# Adatok kiírása
for row in rows:
    print(row)
# Kapcsolat lezárása
conn.close()
```

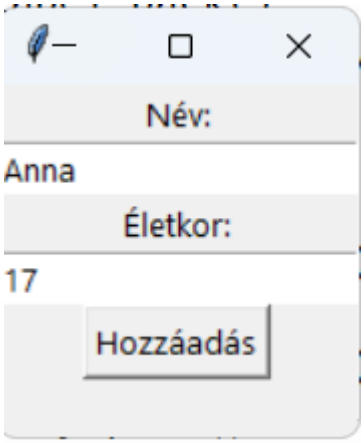
Ez az egyszerű példa bemutatja, hogyan hozhatunk létre egy adatbázist, adhatunk hozzá adatokat és hogyan kérdezhetünk le adatokat SQL-lekérdezésekkel. A SQL nagyon hasznos lehet, ha adatokat szeretnénk rendszerezni, tárolni vagy keresni egy nagyobb alkalmazásban.

Tkinter és SQL közös alkalmazása

Ha a Tkinter-t és SQL-t kombinálunk, akkor például egy adatbázis-kezelő alkalmazást készíthetünk, amely lehetővé teszi a felhasználók számára, hogy adatokat vigyenek be az adatbázisba egy grafikus felületen keresztül. Az alábbiakban egy egyszerű példát mutatunk.

```
import tkinter as tk
import sqlite3

# Funkció adat beszúrásához
def insert_data():
    name = name_entry.get()
    age = age_entry.get()

    conn = sqlite3.connect('mydatabase.db')
    cursor = conn.cursor()
    cursor.execute('''INSERT INTO users (name, age) VALUES
(?, ?)''', (name, int(age)))
    conn.commit()
```

```
    conn.close()

    # Megjelenítjük az új adatokat a címkén
    result_label.config(text=f"Adat hozzáadva: {name},
{age} éves")

# Ablak létrehozása
root = tk.Tk()
root.title("Adatbázis alkalmazás")

# Név mező
name_label = tk.Label(root, text="Név:")
name_label.pack()
name_entry = tk.Entry(root)
name_entry.pack()

# Életkor mező
age_label = tk.Label(root, text="Életkor:")
age_label.pack()
age_entry = tk.Entry(root)
age_entry.pack()

# Gomb az adat hozzáadásához
add_button = tk.Button(root, text="Hozzáadás",
command=insert_data)
add_button.pack()

# Címke a visszajelzéshez
result_label = tk.Label(root, text="")
result_label.pack()

# Fő ciklus indítása
root.mainloop()
```



Ez a példa egy egyszerű Tkinter alkalmazás, amely lehetővé teszi a felhasználó számára, hogy adatokat adjon hozzá egy SQLite adatbázishoz.

Összefoglalás:

- **Tkinter:** Alacsony komplexitású, de közepes szintű alkalmazásokat lehet készíteni vele, ha grafikus felhasználói felületet szeretnél.
- **SQL:** Magas komplexitású, mivel adatbázisok kezelését és lekérdezéseit igényli, de nélkülözhetetlen a nagyobb alkalmazásokhoz, amelyek adatkezelést igényelnek.

Mindkét modul alapvető eszközként szolgálhat Python alkalmazások fejlesztésében, és sok fejlettebb alkalmazásban kombinálják őket, hogy grafikus felületek és adatbázisok együtt működjenek.

Melléklet

Ez a módszertani csak bevezetés a Python nyelvbe. Komplex problémák megoldása és megfelelő programok szerkesztése megköveteli a programozótól mélyebb ismereteket a Python különböző irányzatain. A Python nyelv elsajátitásának az egyik lehetséges sorrendje lehet a következő:

1. Alap szintaxis és adatstruktúrák

- **Szintaxis bemutatása:** Kezdjük az alapvető szintaxis bemutatásával, mint például a változók, adat típusok (pl. számok, szövegek, listák), operátorok és a Python kód szerkesztésének alapjai.
  - Változók és típusok (int, float, string, boolean)
  - Alapvető műveletek (összeadás, kivonás, stb.)
- **Alapvető adatstruktúrák:**
  - Lista (list): Sorozatok kezelésére szolgál.
  - Szótár (dict): Kulcs-érték párokat tartalmazó adatstruktúra.
  - Tuple (tuple): Olyan sorozat, amely nem változtatható (immutable).
  - Halmaz (set): Egyedi elemeket tartalmazó gyűjtemény.

2. Vezérlési struktúrák: Program logikai szerkezete

- **Elágazások (if, elif, else):** A feltételes logikai döntések bevezetése. Az "if" szerkezet a program futásának különböző ágait biztosítja.
- **Ciklusok (for, while):** A ciklusok segítenek az ismétlődő műveletek végrehajtásában.
  - **for** ciklus: Sorozatok, listák vagy más iterálható objektumok bejárása.
  - **while** ciklus: Amíg egy feltétel igaz, végrehajtja a kódot.

3. Függvények és modulok: A kód szervezése

- **Függványek:** A Python függvények segítenek a kód modularizálásában és újrafelhasználásában. A függvények alapvető felépítése (def kulcsszó), paraméterek és visszatérési értékek.
- **Modulok és könyvtárak:** A Python bőséges beépített könyvtárakat kínál. Fontos, hogy a tanulók megismerkedjenek a leggyakrabban használt könyvtárakkal (pl. math, os, random) és megtanulják, hogyan importáljanak modulokat.

4. OOP (Object-Oriented Programming): Objektum-orientált programozás

- **Osztályok és objektumok:** Az objektum-orientált programozás bevezetése a Python-ban. Az osztályok és objektumok alapfogalmainak bemutatása, és hogy hogyan hozhatunk létre osztályokat.
- **Öröklődés:** Az öröklődés és annak használata a Python-ban.
- **Polimorfizmus és enkapszuláció:** Hogyan érhetjük el, hogy különböző típusú objektumok ugyanazokat a metódusokat használják.

5. Hibakezelés és kivételek

- **Kivételek kezelése (try, except):** Hogyan kezelhetjük a program hibáit és kivételeket, hogy a program ne álljon le hiba esetén.
- **Felhasználói hibák:** Hibák kezelése a bemeneti adatokat érintően, például számot kérni a felhasználótól, és megbizonyosodni arról, hogy valóban számot adjon meg.

6. Generátorok, dekorátorok és lambda függvények

- **Generátorok:** A generátorok használata és miért hasznosak, különösen nagy adatmennyiségek esetén.
- **Dekorátorok:** Hogyan lehet dekorátorokkal bővíteni a funkciók viselkedését anélkül, hogy módosítanánk a kódjukat.
- **Lambda függvények:** Rövid, névtelen függvények használata.

7. Adatbázisok és fájlkezelés

- **Fájlkezelés:** Hogyan nyithatunk meg, olvashatunk és írhatunk fájlokat Python-ban.
- **SQL alapok:** Az alapvető adatbázis-kezelési műveletek, például lekérdezések, insertálás és frissítés Python segítségével.

8. Kódolási gyakorlatok és projektek

- Érdemes gyakorlati feladatokat megoldani, amelyek segítenek az elmélet alkalmazásában. Alkalmazzunk kisebb projekteket, például:

Számológép készítése

Szöveges játék készítése

Weboldalak adatainak lekérése

Tanulási megközelítés

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

IRODALOMJEGYZÉK

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

Виробничо-практичне видання

«Сучасні мови програмування (Python. 2 частина)»
Методичні вказівки до лекційних та практичних занять
2025 р.

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

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

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

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ (59 с., угорською мовою)

Розробник методичних вказівок:

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

Рецензенти:

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

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

Відповідальні за випуск:

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

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

За зміст методичних вказівок відповідальність несуть розробники.
Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Статут «Закарпатського угорського інституту імені Ференца Ракоці ІІ» (Затверджено протоколом загальних зборів Благодійного фонду За ЗУІ, протокол №1 від 09.12.2019р., прийнято Загальними зборами ЗУІ ім. Ф.Ракоці ІІ, протокол №2 від 11.11.2019р., зареєстровано Центром надання адміністративних послуг Берегівської міської ради, 12.12.2019р.)*
Шрифт «Courier New». Розмір сторінок методичних вказівок: А4 (210х297мм)