Digitális kultúra

Digitális kultúra

9. osztály

Tartalomjegyzék

Géptermi szabályzat

1. A számítástechnika szaktantermekben tanulóknak tartózkodni csak szaktanári engedéllyel lehet.
2. A géptermek szünetekben, használaton kívül zárva tartandók, csak az arra felhatalmazást kapott személyek nyithatják ki.
3. Internethasználat, tanórán kívüli gyakorlás csak felügyelet mellett történhet.
4. Az internetezési, levelezési etikett (Netikett) betartása mindenkire kötelező.
5. Digitális adathordozót csak szaktanári engedéllyel szabad a termekbe behozni.
6. Élelmiszert, rágógumit behozni tilos.
7. Mobiltelefont csak a tanár által meghatározott feladatra szabad használni.
8. A hálózati főkapcsolókat, a központi számítógépeket bekapcsolni; a központi gépekhez nyúlni tanulóknak tilos.
9. A számítástechnikai eszközök belsejébe nyúlni, azokat rongálni anyagi felelősség terhe mellett szigorúan tilos.
10. A számítógépeket bekapcsolni csak a szaktanár engedélyével, az előírásoknak megfelelő módon szabad.
11. Csak a tanár által meghatározott szoftverek használhatók.
12. A szoftverek beállításainak megváltoztatása szigorúan tilos!
13. Ha a foglalkozás ideje alatt a tanuló rendellenességet, a beállítások megváltoztatását, rongálás észlel, ha a számítógépe meghibásodik, azt azonnal köteles jelenteni, ellenkező esetben magára vállalja a felelősséget.
14. A géptermek rendjéért és tisztaságáért, az asztalok, monitorok, gépházak, billentyűzetek, hangfalak (fülhallhatók), egerek állapotáért minden tanuló egyénileg is felelős.
15. Minden tanuló csak a saját helyén ülhet, melyet az ülésrendbe tanév elején be kell jegyezni.
16. A géptermi szabályzat mindenkire kötelező, aki bármelyik terembe belép!
17. Amennyiben valaki a géptermi munkarendet megsérti és ezzel anyagi kárt okoz – akár gondatlanul, akár szándékosan –, a szülő (törvényes képviselő) köteles a kárt a vonatkozó jogszabályok rendelkezése szerint megtéríteni.

Megjegyzések:

A géptermi szabályzat a 21. és 22. terem használatára egyaránt vonatkozik!

7. pont: Alapértelmezésben a termekben mobiltelefont használni tilos. Ez alól kivételt jelentenek azok a feladatok, amelyek során a mobiltelefon használata szükséges a feladat megoldásához. Ezen felül használható mobiltelefon vagy tablet az elektronikus segédanyagok (mint ez a dokumentum) eléréshez is.

12. pont: A szoftverbeállítások megváltoztatásának a tilalma a Windows rendszerre érvényes. A Linux Mint rendszer esetében minden felhasználó saját beállításokkal rendelkezik, így azok szabadon megváltoztathatóak. Azonban a hálózati beállításokat itt sem szabad módosítani, mert az a gép működőképességére lehet hatással!

8. pont: A hálózati főkapcsolókra vonatkozó tilalmat szükség esetén felülírhatja, ha balaset- vagy tűzvédelmi okokból szükséges a terem azonnali áramtalanítása.

Elméleti ismeretek

Szövegszerkesztés

Algoritmizálás és programozás

Biztos előfordult már veled is, hogy valamilyen nem túl érdekes tevékenységet kellett valamilyen feladat kapcsán újra és újra végrehajtanod és szívesen rábíztad volna azt a tevékenységet valaki másra, aki nálad gyorsabban és esetleg pontosabban végre tudja azt hajtani.

Sok ember volt már korábban is ezzel így. Így született meg nagyon sok háztartási eszköz. De így születtek meg a különböző *robotok* is, vagy maga a *számítógép* is.

Igen, a számítógép egy olyan eszköz, amellyel *automatizálható feladatokat lehet gyorsan és precízen végrehajtani*. Feltéve, hogy van valaki aki megtanítja arra a feladatra.

Ilyen feladat lehet például amikor minden nap ugyanazt a weboldalt felkeressük az interneten azért, hogy megnézzük, aznap milyen idő várható. Ügyesebb emberek erre írnak egy kis programot, amely helyettük felkeresi azt az oldalt, kimásolja belőle a megfelelő helyen levő tartalmat és csak azt jeleníti meg a képernyőn. Ezzel időt lehet spórolni, hiszen nem kell megvárni, míg az oldal betöltődik, megkeresni az oldalon belül a kívánt részt.

A számítógép felépítése folytán egy speciális gondolkodást igényel a működtetése. Ez a speciális gondolkodási mód szükséges a programok megírásához. Persze ez nem olyan értelemben speciális, hogy túl nehéz dolog lenne elsajátítani ezt a gondolkodásmódot. Ráadásul a programozói szemlélet az élet más területein is hasznos lehet. Ezért javasolják a világon egyre többen, hogy azok is tanuljanak meg legalább alapszinten programozni, akiket egyáltalán nem érdekel a számítógépek világa.

A számítógépek olyan feladatokat hajtanak végre, amelyek pontosan definiált lépések sorozatából állnak. Az ilyen feladatok egy tevékenység automatizálását jelentik. A lépések leírását pedig *algoritmus*nak nevezzük. (Később szerepelni fog az algoritmus pontosabb definíciója is.) Az ilyen algoritmusokban gondolkodás az a gondolkozásmód, amelynek elsajátítása a célja a Digitális kultúra tantárgy **Algoritmizálás és programozási nyelv** nevű témakörének – magyarul: a programozás tanításának.

Akár programozónak készülsz, akár történésznek, az algoritmusok és az algoritmikus gondolkodás még hasznos lesz a számodra.

Ez a dokumentum az alapokat tartalmazza. Ehhez egy könnyen tanulható, de a programírás íratlan szabályainak elsajátítását a saját, egyébként bizonyos mértékig szigorú szabályain keresztül is segítő nyelvet használ, amely manapság az egyik legnépszerűbb programozási nyelv a világon: a Python nyelvet.

1. Az algoritmus, algoritmusleíró eszközök

1.1. Az algoritmus fogalma

1.1. Fogalmazd meg a lehető legrészletesebben, milyen tevékenységek sorozatát kell végrehajtani ahhoz, hogy a mobiltelefonodon felhívhasd egy ismerősödet, akinek benne van a telefonszáma a névjegyzékben!

1.2. Milyen tevékenységeket kell végrehajtani ahhoz, hogy egy önkiszolgáló boltban vagy egy bevásárló központban vehess készpénzzel

- 1 liter tejet;
- 4 zsömlét;
- kiflit;
- 1 pohár joghurtot;
- 25 dkg vajat és
- 10 dkg felvágottat?

Mindkét feladat esetén írd le a szükséges lépéseket a füzetedbe!

A fenti feladatok megoldásaként olyan lépéssorozatoknak kellett keletkezniük, amelyek lépésein végighaladva az adott feladat megoldható. Az ezekhez hasonló lépéssorozatokat nevezhetjük algoritmusnak, de nem minden ilyen lépéssorozat algoritmus.

Az algoritmus valamely feladat megoldására szolgáló, véges számú lépésekből álló műveletsorozat. Ahhoz, hogy valóban algoritmusról beszélhessünk, ennek a műveletsorozatnak az alábbi feltételeket teljesítenie kell:

1. Az algoritmusnak **véges számú lépésben véget kell érnie**. Amennyiben nem ér véget véges lépésben a megoldás, akkor azt legfeljebb e*ljárás*nak nevezhetjük, de nem algoritmusnak.
2. A probléma **általános** megoldását kell adnia, azaz bármilyen, a feltételeknek megfelelő bemenő adatokra adja meg a megoldást.
3. Legyen **egyértelmű**: ugyanazokra a bemenő adatokra mindig ugyanazt a megoldást adja. Vannak ugyanakkor olyan eljárások is, amelyeknél nem garantált a mindig azonos végeredmény, mert például valamilyen véletlen eseményt is felhasználnak. Az ilyen eljárásokat szokás *heurisztikus algoritmusok*nak nevezni.
4. Egyértelműen definiálva van az algoritus **bemenete** és **kimenete**. Azaz az algoritmus leírása pontosan definiálja, hogy mennyi, pontosan milyen bemenő adatot kell megadni az algoritmus számára. Az adatok esetében megadhatóak feltételek, mint például hogy pozitív egész számok legyenek, vagy hogy csak az angol ábécé betűiből álló szó legyen. Szintén pontosan meghatározottnak kell lennie, hogy az algoritmus mennyi és milyen jellegű kimenő adatot fog előállítani.

A számítógépek működési módjukból adódóan mindig valamilyen algoritmust hajtanak végre. Az algoritmusban szereplő műveletek az adott számítógép gépi kódjában létező utasítások. Ezek az algoritmusok így persze sokkal részletesebbnek tűnnek, mint például egy telefonálás leírása. Azonban amennyiben egy számítógép vezérelte robotot akarunk megtanítani telefonálni, akkor a telefonálás algoritmusát is ilyen részletes, elemi műveletek sorozatára kell bontani.

1.1. ábra: Al-Kwarizmi szobra a teheráni Amir Kabir Egyetemen

A definícióban szereplő eljárás szó nem azonos a később, a vezérlési szerkezeteknél említett eljárással. A vezérlési szerkezeteknél az eljárás általában egy algoritmus megvalósítását végző programegység, míg itt ez a kifejezés egyszerűen arra utal, hogy vannak olyan műveletsorozatok, amelyek nem felelnek meg az algoritmus definíciójának.

ABU ABDULLAH MUHAMMED BIN MUSA AL-KWARIZMI, perzsa matematikus (kb. 780 – kb. 850) eredetileg arabul írt, latinra lefordított művének köszönhetően terjedt el Európában az indiai eredetű számok használata. Ennek a műnek köszönhetően ismerjük ezeket a számokat *arab számok*ként. A matematikus nevének jelentése valami ilyesmi: *Abdullah apja, Musa fia, Mohamed, aki Kwarizm-ből származik*. Nevében a származási helyre utaló kifejezésből alakult ki többszöri átalakulás révén az angol *algorithm* szó, amelyet magyarul *algoritmusnak* mondunk.

A szóban forgó mű valójában az indiai számokkal dolgozó matematikai műveleteket, mai szóval algoritmusokat ír le. A középkorban sokáig erre a szerző nevével utaltak, azért lehetett végül az algoritmus szó az ilyen műveletek elnevezése.

Érdekesség még, hogy a mű eredeti arab nyelvű címéből (al-Kitāb al-Mukhtasar fī Hisāb **al-Jabr** wal-Muqābalah) származik a matematika egyik ágának, az algebrának az elnevezése is.

1.2. Algoritmusleíró módszerek

A következőkben néhány módszert ismerhetsz meg, amellyel egy algoritmust emberek számára érthető módon lehet megadni. Ezen módszerek ismerete azért szükséges, mert a továbbiakban sok algoritmust fogsz megismerni, amelyek mindegyike az itt bemutatott módszerek valamelyikével – akár ugyanaz az algoritmus több módszerrel is – lesz bemutatva.

1.2.1. Mondatszerű leírás

A **mondatszerű leírás**, vagy **pszeudokód** a normál nyelven írt szöveghez leginkább hasonlító, de egyben a programnyelvekre is hasonlító megoldás egy algoritmus leírására. Gyakran használja a programozási nyelvek strukturális elemeit egyszerűsített formában a később bemutatandó vezérlési szerkezetek megadására. Azonban mivel a cél az emberi olvashatóság és nem a számítógép számára érthető fogalmazás, hiányoznak belőle olyan részletek, amelyet a programba bele kell írni, de az algoritmus megértéséhez nem szükségesek, vagy a megértést akár egyenesen nehezítenék is. Cserébe viszont magyarázatokat tartalmazhat, amelyek a programban nem szükségesek, de segítik az algoritmus megértését.

Hivatalos nyelvtan nem létezik a pszeudokódra, de egyes egyetemek esetleg megkövetelik a programozást tanuló diákjaiktól egy adott szabályrendszer alkalmazását.

A mondatszerű leírás alkalmazásának egyik előnye, hogy a leendő program forráskódjában is elhelyezhető megjegyzésekben. A programírás következő lépéseként az egyes részekhez megírható a neki megfelelő programrészlet, így haladva a programfejlesztésben a nagyobb elemektől a részletek felé. Egyben a program dokumentálásának egy része is elkészül ilyen módon, hiszen már eleve ott van a kódban a megkívánt magyarázó megjegyzés.

Az alábbiakban egy példa látható mondatszerű leírásra:
Algorithm Largest Number
Input: A list of numbers in L.
Output: The largest number in the list L.

if *L*.size = 0 **return** null
largest ← L[0]
for each *item* **in** *L*, **do**
  if *item* > *largest*, **then**
    largest ← *item*
return *largest*

Ugyanez magyarul valahogy így nézne ki:
Bemenet: *L*, számok listája
Kimenet: a legnagyobb szám az *L* listában, vagy null, ha L üres

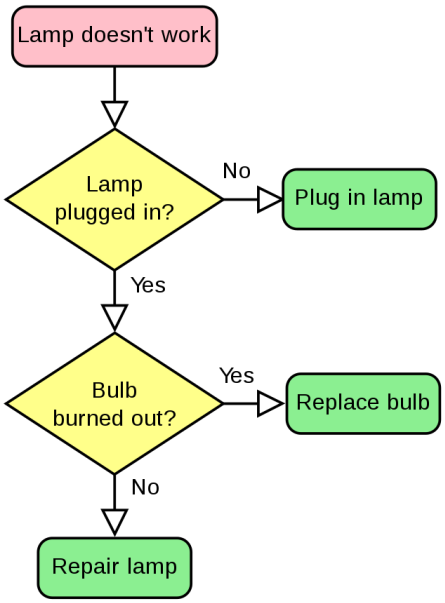
ha *L*.size = 0 **visszaad**: null
legnagyobb ← L[0]
minden egyes *elem*-re *L*-ben, csináld:
  ha *elem* > *legnagyobb*, **akkor**
    legnagyobb ← *elem*
ciklus vége
visszaad: *legnagyobb*

1.2.2. Folyamatábra

Wikipédia: Flowchart szócikk

A folyamatábra (angolul: *flowchart*) egy folyamat vagy tevékenység grafikus ábrázolására alkalmas. A folyamatábrán ábrázolható egy algoritmus vagy bármely más olyan folyamat is, amelyet lépések sorozatára bontva akarunk ábrázolni.

Az egyes lépéseket különböző alakú blokkokként ábrázolja, amelyet nyilak kötnek össze. A nyilak mentén lehet haladni az egyes blokkokról tovább a következőre. Ilyen módon grafikusan szemlélteti a végrehajtandó tevékenység lehetséges folyamatát.



1.2. ábra: A wikipédia szócikk példaábrája folyamatábrára: Hogyan kell megoldani a nem működő lámpa problémáját?

A 1.2. ábra a Wikipédia angol oldaláról származik. Azt mutatja be, hogy mi a teendő, ha egy lámpa nem működik. A felső, rózsaszín hátterű blokk a kiinduló állapot, a sárga rombuszok elágazási pontok, ahol a feltett kérdésre adott válasz alapján dől el, hogy melyik nyilat kell követni. A zöld blokkok végrehajtandó tevékenységeket jelölnek – egyben az ábrázolt folyamat befejezését is ebben a példában.

A folyamatábrán alkalmazott blokkok formájára több konvenció is kialakult, de általában elmondható, hogy a folyamat kiindulópontját (amelyből mindig egy van), valamint a befejezőpontokat (amiből több is lehet) lekerekített sarkú téglalappal vagy ellip-

szissel jelzik. Ugyanúgy az egyes feltételek alapján történő elágazási pontokat vagy rombusszal vagy esetleg lapított hatszöggel szokás jelölni (utóbbit akkor alkalmazzuk, ha a beleírandó szöveg mennyisége miatt a rombusz nem megfelelő).

1.2.3. A program, mint algoritmus

A legegyértelműbb algoritmus leíró eszköznek maga a program tűnik. Hiszen a programot használjuk arra, hogy egy adott algoritmus végrehajtására rávegyük a számítógépet.

Ez azonban nem ilyen egyértelmű abból a szempontból, hogy bizonyos programozási nyelvek esetén a program elég nehezen olvasható lehet (gondoljunk csak a következő leckében bemutatott Assembly nyelvre). Amennyiben az algoritmusnak egy emberi olvashatóságot biztosító megadására gondolunk, akkor a program nem tekinthető jó megoldásnak – ezért fejlődtek ki más algoritmus leíró módszerek, mint a bemutatott két példa.

Azonban maga a program valóban egy módszer az algoritmus leírására: a számítógép számára írja le az algoritmus lépéseit.

A **program** utasítások gyűjteménye, amelyet egy számítógép képes végrehajtani, ezzel egy adott tevékenységet megvalósítva.

A számítógépes programot általában egy vagy több programozó készíti egy adott programozási nyelven. Az ember által olvasható forráskódból valamilyen eszköz állítja elő a számítógép által érthető gépi kódot.

Programok, programkönyvtárak és a hozzájuk kapcsolódó adatok összességét nevezzük **szoftver**nek.

1.3. Fogalmazd meg mondatszerű leírással és folyamatábrával is a mosogatás algoritmusát!

Az algoritmus bemenet: az elmosogatandó evőeszközök halmaza.

Az algoritmus kimenete: a bemenetet alkotó evőeszközök, tisztán és szárazon.

A leckében felhasznált képek forrása:

- https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi
- https://upload.wikimedia.org/wikipedia/commons/9/91/LampFlowchart.svg

További internetes címek a leckében szereplő témában:

- https://en.wikipedia.org/wiki/Algorithm
- https://en.wikipedia.org/wiki/Algorithm_characterizations
- https://en.wikipedia.org/wiki/Flowchart
- https://en.wikipedia.org/wiki/DRAKON
- https://en.wikipedia.org/wiki/Unified_Modeling_Language
- https://en.wikipedia.org/wiki/Computer_program

2. Programozási nyelvek

2.1. A programozási nyelvek kialakulása

Hol volt, hol nem volt, volt egyszer egy magyar származású amerikai matematikus, akit a világ *John von Neumann* néven ismert meg, holott a Magyarországról való távozásáig **Neumann János**nak hívták. Aránylag rövid élete ellenére nagyon terjedelmesnek mondható a munkássága. A számítógépek szempontjából a munkásságának a legfontosabb eleme mégis az, hogy még az elektronikus számítógépek fejlesztésének hajnalán, 1945-ben megfogalmazta (*First Draft of a Report on the EDVAC*) a számítógépek működési elvét. Az általa megfogalmazottaknak megfelelően működik mind a mai napig a legtöbb számítógép. Az angol nyelvű szakirodalomban az ezen elvek szerint működő gépeket mind a mai napig *von Neumann-architecture* néven emlegetik. Mi magyarok egyszerűen Neumann-elvről szoktunk beszélni (esetleg többes számban).

2.1. ábra: Neumann János (John von Neumann) a Los Alamos National Laboratory képén

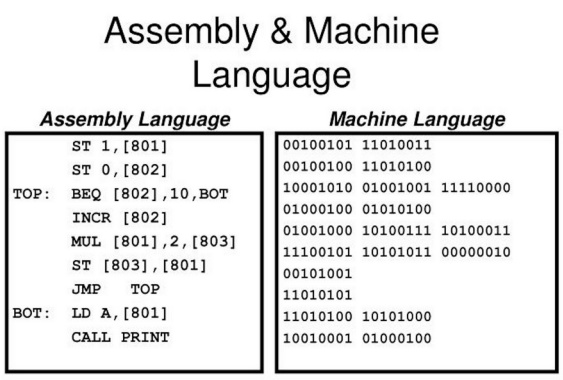
Röviden a Neumann-elv:
1. A számítógép legyen teljesen elektronikus és bináris működésű.
2. Az utasításokat és az adatokat ugyanabban a teljesen elektronikus működésű memóriában tárolja. Azok értelmezése csak a felhasználásuktól függjön.
3. Az utasításokat egy vezérlő egység hajtsa végre. A számítási műveleteket egy aritmetikai-logikai egység hajtsa végre.
4. Az utasításokat a program által meghatározott sorrendben, sorban kell végrehajtani.
5. A számítógép legyen univerzális.
6. A számítógép rendelkezzen bemeneti és kimeneti egységgel.

Számunkra a Neumann-elvből most az a fontos, hogy Neumann János egyértelművé tette, hogy a számítógépeknek a működésükhöz a bináris számábrázolást kell alkalmazniuk, illetve hogy az utasításokat és az adatokat ugyanabban az elektronikusan működő memóriában kell tárolnia a gépnek.

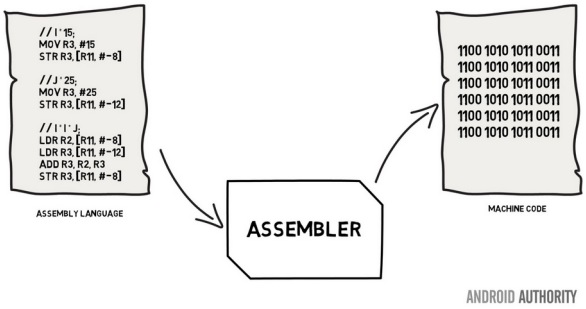
A számítógép egyes elemeit egy-egy vezérlőjel irányítja. Ez a vezérlőjel a bináris működés esetén azt jelenti, hogy a megfelelő vezetékben vagy van feszültség vagy nincs. Ez a két érték egy bitnek felel meg. Amennyiben az egyes vezérlőjeleket egy meghatározott sorrendben egymás mellé tesszük, akkor minden egyes művelethez hozzárendelhető egy meghatározott bitsorozat. Valójában ezt a bitsorozatot kell az adott műveletet előíró utasítás kódjának tekinteni. Ez az, amit a Neumann-elv szerint a memóriában tárolni lehet. Az ilyen utasításkódokat nevezzük **gépi kód**nak.

Természetesen nem minden bitsorozat fog értelmes műveletet jelenteni. Az egyes mai processzorokon értelmes bitsorozatok halmazát nevezzük az adott processzor *utasításkészletének*. A mai processzorok esetében persze már nem feltétlenül felel meg minden bit egy-egy vezérlőjelnek, de ennek csak az az oka, hogy egyes processzorok az utasításkód alapján a belső áramköreikkel állítanak elő valamilyen vezérlőjeleket.

Neumann János idejében a számítógépeket még főképpen komolyabb matematikai számítások gyorsabb elvégzésére tervezték. Ekkor még magától értetődő volt, hogy a végrehajtandó utasítássorozatokat is matematikusok állították elő. Ám az ilyen bináris számsorozatok formájában történő programírás még egy matematikus esetében sem könnyű feladat. Egy-egy utasítás ugyanis nagyon alapvető elemi műveletet jelent, mint a memória valamelyik cellájának tartalmát átmásolni a processzor egy adott regiszterébe, két regisztert összeadni, stb. Így például egy gyökvonás megvalósítása is akár húsz-harminc elemi utasítás megfelelő sorrendben való végrehajtását igényli. Ehhez pedig még hozzájött, hogy ezeket az elemi utasításokat bináris számsorként kellett megadni.



2.2. ábra: Az assembly és a gépi kód összehasonlítása



2.3. ábra: Assembly kód átalakítása gépi kódra

Ezért elég hamar felmerült az igény arra, hogy az egyes elemi műveleteket a bináris számsor helyett valamilyen szimbólumokkal lehessen jelölni. Meg is született az egyes számítógépekre az a program, amely egy egyszerű szimbólumrendszerben felírt utasássorozatot át tudott írni az utasítások bináris kódjává. Így – bár még mindig matematikusokat igénylő feladat maradt – az egyes programok megírása valamivel könnyebbé vált. Az így kapott nyelvet nevezzük **assembly** nyelvnek. Ez még mindig csak egy jobb felírását adja a programnak a gépi kód helyett.

A számítógépek azonban továbbra is csak a saját gépi kódjukon megírt programot tudják végrehajtani. Így aztán minden egyes processzorfajtához a saját assembly nyelvén megírt kódot egy **assembler**nek nevezett programnak kell átalakítani.

Az assemblernek aránylag egyszerű a feladata: az egyes utasításszavakat át kell fordítania a megfelelő gépi kódra. Hogy pontosan melyikre, abban mindössze az ad némi nehézséget, hogy ugyanazt a műveletet különböző fajtájú adatokon bár ugyanazzal a névvel jelöli az assembly, ezek gépi kódja általában néhány bitben eltér.

Ahogy a számítógépek fejlődtek, olcsóbbá, ezáltal egyre több célra elérhetővé váltak, a velük elvégezhető műveletek kidolgozása a matematika egy új ágát, a *számítástudományt* hívta életre. Ennek az alkalmazott matematikai területnek eredményeként egyre többféle – először kifejezetten matematikai – feladat megoldására dolgoztak ki automatizált módszereket, amelyeket a megfelelő elemi műveletekre bontva azokat számítógépen végre lehetett hajtani. Gyakorlatilag az ilyen megoldási módokat nevezzük algoritmusoknak.

Ezeknek az algoritmusoknak a leírására már az assembly nyelvnél általánosabb leírást lehetővé tevő módszer kellett. Ez szülte az igényt arra, hogy a számítógépet megtanítsák az emberi nyelv megértésére.

Bár ez a cél azóta sem teljesült, megszülettek újabb és újabb mesterséges nyelvek, amelyek az emberi nyelv (elsősorban az angol nyelv) leegyszerűsítésével próbáltak megoldás találni arra, hogy a számítógép programozási nyelve hasonlítson az emberi nyelvre.

Ezzel megszületett a *magas szintű programozási nyelv* fogalma, vele együtt a fentebb bemutatott nyelvekre (a gépi kódra és az assembly-re) az *alacsony szintű programozási nyelv* elnevezés.

A magas szintű programozási nyelvek már elvonatkaztathatóak egy adott számítógéptípustól. Így lehetővé teszik olyan program írását, amely bármely típusú számítógépen futhat – persze bizonyos feltételek teljesülése esetén. Cserébe sokkal nagyobb méretű kódot eredményezhet, amelynek nagyobb memóriafelhasználás és lassabb futás lehet az eredménye. Ezért

Wikipédia: Alacsony szintű programozási nyelv szócikk

Wikipédia: Magas szintű programozási nyelvek szócikk

sokszor a magas szintű nyelven megírt programoknak is a legkritikusabb részeit assembly nyelven írják meg a hatékony, gyors futást eredményező kód elérése érdekében.

A következőkben röviden bemutatásra kerül néhány ilyen programozási nyelv abból a rengetegből, amelyet mostanáig kifejlesztettek. Ezek némelyike ma már nem használatos és lesznek a jövőben újabbak, amelyeket ma még nem ismerünk.

2.4. ábra: A számítógépes nyelvek hierarchiája

2.2. Programozási nyelvek rövid bemutatása

Fortran: Az egyik első szélesebb körben elterjed programozási nyelv volt. Eredetileg az IBM még az 1950-es években fejlesztette ki, kifejezetten tudományos számítások elvégzéséhez. Azóta használják olyan bonyolult számításokat igénylő területeken, mint például a numerikus időjáráselőrejelzések, folyadékszimulációk, geofizika stb. Olyan programok írásához is használatos, amelyekkel a leggyorsabb szuperszámítógépek teljesítményét tesztelik le. A Fortran több később született nyelv mintájául is szolgált. Ezek közül az egyik a BASIC nyelv. 2021. márciusában a programozási nyelvek közül a 22. legnépszerűbbnek mérték.

Van egy mondás, miszerint az informatikus lusta. Már a Fortran megalkotása is bizonyítja ennek igazságát: John Backus mondta erről egy 1979-es interjúban. „Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make easier to write programs."

Az idézet fordítása: „A munkám során sok dolog származik a lustaságomból. Nem szerettem programokat írni, így amikor az IBM 701-en dolgozva lőelemszámító programok írásával foglalkoztam, nekiálltam írni egy programfejlesztő rendszert a programírás megkönnyítésére."

ALGOL: Az *Algorithmic Language* rövidítéséből származik a nyelv neve. Eredetileg 1958-ban fejlesztették ki. A különböző tankönyvekben és egyetemi szakkönyvekben az algoritmusok leírásában szabvánnyá vált az ALGOL nyelv használata. A manapság használt programozási nyelvek nagyrészére szokás mondani, hogy a nyelv formája Algol-szerű. Ezek a nyelvek ugyanis többékevésbé az Algol nyelvnél alkalmazott elveket követik. A Fortran nyelv több hátrányát kiküszöbölte, utat nyitva a modernebb nyelvek létrejöttének.

A Fortran nyelvben a változók ha I-N betűvel kezdődnek, akkor alapértelmezésben egésznek (integer), egyéb esetben alapértelmezésben valósnak (real) számítanak, amit persze felül lehet bírálni. Ebből származott az a magyarra nem fordítható mondás, miszerint „In Fortran, GOD is REAL (unless declared INTEGER)."

COBOL: Egy angol-szerű programozási nyelv. 1959-ben fejlesztették ki. Neve a *common business-oriented language* kifejezés rövidítéséből származik. 2002. óta objektum-orientált nyelv. Elsősorban üzleti, pénzügyi és adminisztrációs feladatokat ellátó rendszerek fejlesztésében használják, céges és kormányzati környezetben egyaránt. Nagygépes rendszereken manapság is széles körben elterjedt a használata ugyan, de manapság a COBOL-ban írt programokat egyre inkább átírják modernebb nyelvekre. Bár a többi nyelvnél jobban hasonlít az angolra, de pont ezt hozzák fel a bírálataként is, hiszen túl bonyolulttá, bőbeszédűvé teszi a programot. Még 2012-ben is az üzleti világban használt programok 60%-a COBOL-ban írt program volt.

BASIC (*Beginners' All-purpose Symbolic Instruction Code*): Egy általános célú, magas szintű programozási nyelv, amely megtervezésének célja a könnyű használhatóság volt. 1964-ban adták ki

2.5. ábra: Alfred Edward Chalon: Ada Lovelace portréje, akvarell 1840-ből

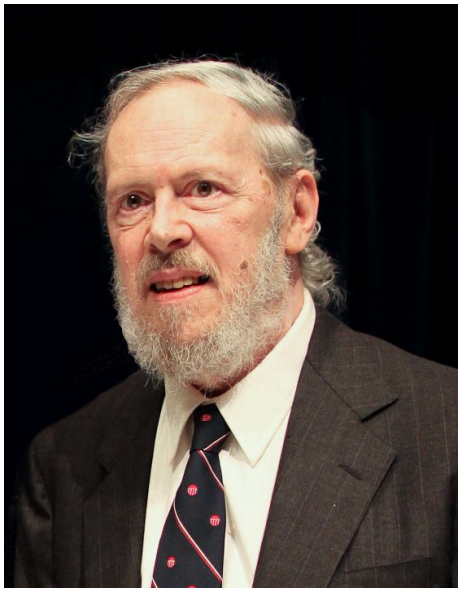
az első változatát a Darthmouth College-en. Tervezője THOMAS E. KURTZ és az a JOHN G. KEMENY volt, aki 1926. május 31-i születésekor még a KEMÉNY JÁNOS nevet kapta Budapesten. (Mellesleg a Dartmouth Collage 13. elnöke volt a Wikipédia szerint.) A nyelvet sokáig használták a programozás oktatására, ami mai szemmel nézve ugyan nem tűnik jó ötletnek, abban az időben viszont még szinte bármilyen számítógép-felhasználáshoz a felhasználónak magának kellett programot írnia. Az akkoriban egyre olcsóbbá váló miniszámítógépeken ez a nyelv tette lehetővé a számítógépek szélesebb körű felhasználást. Több olyan számítógép is készült (például a Commodore cég gépei), amelyen alapból a BASIC nyelv valamely változata volt elérhető.

Simula: Amint a neve is utal rá, eredetileg szimulációs célra szóló alkalmazások készítésére szánt nyelv. Az első verzió, amely 1962-ben jelent meg (Norvégiában fejlesztették ki) valóban szimulációs feladatokra volt alkalmas, de a második változata, a **Simula 67** már általános célú nyelv volt. Ennél a nyelvnél jelent meg először az objektum-orientált programozás lehetősége. A modern, objektum-orientált nyelvek kialakulásában nagy hatása volt.

Pascal: A svájci NIKLAUS WIRTH az Algol egy fejlesztésének szánta eredetileg, végül a francia matematikus *Blaise Pascal* tiszteletére **Pascal** névre keresztelte ezt a nyelvet, amely 1970-ben jelent meg és elég hosszú ideig népszerű nyelv volt. Elsősorban a jó programozási gyakorlatok alkalmazását támogató nyelvnek szánta Wirth. Valóban kiszorította a kevésbé modern programozási módszereket alkalmazó BASIC nyelv használatát elég hamar. A nyelv később objektum-orientált elemekkel is bővült, amely elvezettett a ma is használt **Delphi** nyelv létrejöttéhez.

Ada: A világ első programozójának tekintik az angol költő, Lord Byron lányát AUGUSTA ADA KING bárónőt, aki CHARLES BABBAGE gépéhez készített programokat. Róla kapta a nevét az **Ada** programozási nyelv, amely többek között a Pascal nyelvből alakult ki.

BCPL, **B**, **C**: Eredetileg *compilerek* írásához tervezett, ma már nem használt programozási nyelv volt a **BCPL** (a compilerekről a következő leckében lesz szó). Ebből kiindulva alkotta meg KEN THOMPSON és DENNIS RITCHIE a **B** nyelvet. Ennek további bővítésével hozták létre a **C** nyelvet, amely a mai napig az egyik alapnyelvnek tekinthető. Sőt, Thompson és Ritchie egy operációs rendszer fejlesztésével is foglalkozott, amelyet bár először Assembly nyelven kezdtek el írni, de mivel a C nyelv minden ismert számítógépre rendelkezett fordítóval, így átírták az addig elkészült kódot C-re. Az így kifejlesztett operációs rendszer lett a **Unix**, amelyet a Windows kivételével minden ma használatos operációs rendszer ősének lehet tekinteni. Mivel az 1960-as években zajlott mind a C nyelv, mind a Unix kifejlesztése, hivatalosan *1970. január 1*-t tekintjük a Unix születésnapjának. Ennek az a jelentősége, hogy a legtöbb számítógépen az időmérés az ekkortól eltelt idő alapján történik.



2.6. ábra: Dennis Ritchie (1941–2011)

C++, *C#*, *Java*, *JavaScript*, *Objective-C*, *PHP*... Csak néhány példa azokból a nyelvekből, amelyeket közvetlenül vagy közvetve a C nyelvből fejlesztettek tovább.

2.7. ábra: Bjarne Stroustrup

Smalltalk: Az első igazán objektum-orientált programozási nyelv (a Simula mellett, amely ténylegesen az első ilyen nyelv volt).

C++: A dán BJARNE STROUSTRUP által fejlesztett nyelv, amely elsősorban a C nyelv továbbfejlesztési szándékával jött létre. Gyakorlatilag egy C nyelvű program kis módosítással C++ nyelvű programként is tekinthető. Elsősorban az objektum-orientált programozás lehetőségeit építette be a Simula és a Smalltalk nyelvekben alkalmazottak alapján, de más bővítést is kapott a nyelv. Mivel a változó értékének növelése a C nyelvben a ++ művelettel lehetséges, így a C++ név egyértelműen arra utal, hogy ez a C továbbfejlesztése.

Java: A C++ nyelv mintájára készült, kizárólag objektum-orientált programozást támogató programozási nyelv.

C#: A Microsoft által kifejlesztett; **Objective-C**: Az Apple által kifejlesztett C-nyelvre alapuló, objektum-orientált megközelítést alkalmazó programozási nyelvek. A Java, C# és az Objective-C gyakorlatilag egymás unokatestvérének tekinthetőek mind használatban, mint a nyelvek tulajdonságaiban.

JavaScript: Ennek a nyelvnek a nevén kívül semmi köze a Java nyelvhez! Kifejezetten webes alkalmazásokhoz fejlesztett C, C++ nyelvekre épülő nyelv, amelyet egy idő után már más célra is elkezdtek használni.

PHP: Szintén webes alkalmazások készítésére használt nyelv, amely szintén a C nyelv leszármazottjának tekinthető. A szerver oldali programok írására használatos, a kimenete általában egy olyan kód, amit a webszerver a böngésző programnak elküld megjelenítésre.

SQL: A fenti nyelvekkel ellentétben, amelyek általános célú nyelvek, az SQL egy célnyelv, amely adatbázisok kezelésére használatos. Nem lehet vele másmilyen feladatot végrehajtani, ezért nevezzük célnyelvnek.

Lisp: Egy speciális nyelvcsalád, amely bár általános célúnak számít, de abban speciális, hogy listákkal dolgozik, amelyeknek általában az első eleme a művelet, míg a lista többi eleme olyan adat, amellyel a műveletet el kell végezni.

Python: Egy egyre népszerűbb programozási nyelv, amely ugyan nem a C nyelv leszármazotta, de elég sok elemében látszik a C nyelv, mint minta használata. Könnyen tanulható ugyan, de sok olyan megkötése van, amely a használóját a helyes programozási szokások elsajátítására neveli. Ezért első programozási nyelvnek is kifejezetten alkalmas. A továbbiakban ezt a nyelvet fogjuk a programozás tanulására felhasználni, így a későbbiekben sok tulajdonságát megismered majd.

A fent bemutatott programozási nyelvek mellett még rengeteg nyelv létezett, létezik és fog megjelenni. Gyakorlatilag lehetetlen az összeset felsorolni, annyi van belőlük. A ma használatos programozási nyelvek legtöbbje valamilyen rokonságban áll a C nyelvvel, amiből adná magát, hogy először nyilván a C nyelvet célszerű megismerni. Ám ha figyelembe vesszük, hogy a C nyelvet szokták *magasszintű assembly* néven is emlegetni, akkor talán ez már utal arra, hogy nem a legbarátságosabb nyelvről van szó.

Hasonló módon lehetne első nyelvnek a C nyelvből származó objektum-orientált programozásra alkalmas nyelvet választani, hiszen manapság az *objektum-orientált programozás* számít az egyik

legfejlettebb módszernek. Ám ebben az esetben olyan fogalmakkal kellene már az első program megírása előtt megbarátkozni, mint az *osztály*, *polimorfozmus*, *egybezártság* stb. Figyelembe véve, hogy középiskolában az objektum-orientált programozási módszerek ismerete nem követelmény még érettségin sem, így nem javasolt olyan nyelvet használni, amely objektum-orientált programozáshoz készült, így sem a Java, sem a C#, sem a C++ nem jó választás.

A Python ugyan szintén erősen épít az objektum-orientált megközelítésre, de úgy is lehet használni, hogy erről nem veszünk tudomást. Könnyen tanulható, jól olvasható nyelv, amely ráadásul nagyon sok feladatra alkalmas. A segítségével lehet egyszerű, de fárasztó számítógépes tevékenységet könnyedén automatizálni. Talán pont ennek köszönhető a növekvő népszerűsége. Éppen ezért a továbbiakban a programozás alapjait a Python nyelven keresztül fogod megismerni.

Amennyiben kíváncsi vagy további nyelvekre, akkor például jó kiindulópont lehet a Wikipédia ezzel kapcsolatos angol nyelvű oldala, amelynek címét megtalálod a lecke végén.

2.3. Hogyan lesz a forráskód futtatható?

A magas szintű programozási nyelven írt programot a számítógép közvetlenül nem tudja végrehajtani. Kell egy tolmács hozzá. Ez a tolmács alapvetően kétféle lehet:

- **Compiler** (*fordító*): Olyan program, amely egy adott nyelvű forráskódot (a magas szintű nyelven írt programkódot) feldolgozva előállítja a vele azonos műveleteket tartalmazó gépi kódú programmá, azaz lefordítja azt gépi kódra. Ennek eredményeként egy új program keletkezik, amelyet a számítógép már végre tud hajtani. Előnye ennek a megoldásnak, hogy a lefordított program már közvetlenül, további segédeszköz nélkül futtatható a számítógépen. Másrészt a fordítást nem feltétlenül kell azon a gépen végezni, amin futtatni akarjuk. Hátránya, hogy a fordítás sokáig tarthat, mivel a fordító optimalizálja is a kódot.
- **Interpreter** (*értelmező*): Ebben az esetben az értelmező szimulál egy olyan számítógépet, amely képes végrehajtani a forráskódot. Azaz maga az értelmező fogja lépésről lépésre haladva a programot végrehajtani a megfelelő műveletek elvégzésével. Hátránya, hogy a program az értelmező nélkül nem futtatható. Előnye, hogy a program végrehajtására nem kell várni a fordítással. Így az értelmezőt használó nyelveket használva gyorsabban lehet tesztelni a programot, vagyis felgyorsul a fejlesztés. Azonban az értelmezők általában valamennyire lassítják a program végrehajtását, így általában előbb-utóbb átírják a programot egy fordítót használó nyelvre.

A két fenti módszer hibridizálásával léteznek olyan megoldások is, amikor a forráskódot egy köztes byte-kódra fordítja le egy fordító, majd az így kapott kódot egy értelmezőként funkcionáló futtató környezet hajtja végre. Ez a módszer a fenti módszerek hátrányait egyesíti, mégis ezt használja például a Java és a .NET is.

A leckében szereplő képek forrásai:

- https://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.jpg
- https://image3.slideserve.com/5674619/assembly-machine-language-n.jpg
- https://cdn57.androidauthority.net/wp-content/uploads/2016/03/assembler.jpg
- https://en.wikipedia.org/wiki/Dennis_Ritchie#/media/File:Dennis_Ritchie_2011.jpg
- https://en.wikipedia.org/wiki/Bjarne_Stroustrup
- https://en.wikipedia.org/wiki/Ada_Lovelace

További internetes címek a leckében szereplő témában:

- Fortran: https://en.wikipedia.org/wiki/Fortran
- Algol: https://en.wikipedia.org/wiki/ALGOL
- COBOL: https://en.wikipedia.org/wiki/COBOL
- BASIC: https://en.wikipedia.org/wiki/BASIC
- https://en.wikipedia.org/wiki/List_of_programming_languages
- Python: https://en.wikipedia.org/wiki/Python_(programming_language)
- https://en.wikipedia.org/wiki/Indentation_style

3. Ismerkedés a Python nyelvvel

Ebben és a következő leckében elkezdünk ismerkedni a **Python** nyelvvel és azokkal az eszközökkel, amelyek segíteni fognak a programok elkészítésében. A nyelv a nevét – bár neve ezt sugallja – nem az óriáskígyóról kapta, ezért nem is helyes „*pitonnak*” nevezni, hanem az angol abszurd humor egyik híres művelőjéről, a *Monty Python* nevű csoportról, így a nyelv nevének helyes kiejtése fonetikus átírásban inkább valami ilyesmi: „*pájton*”.

A nyelv kifejlesztőinek a kedvence volt a *Monty Python*, így a nyelv hivatalos dokumentációja tele van a társulat filmjeiben és tévésorozataiban szereplő idézetekkel.

A Monty Python csoportról további információkat találhatsz a neten például magyarul a Wikipédia róluk szóló szócikkéből indulva:
https://hu.wikipedia.org/wiki/Monty_Python

A nyelv könnyen tanulható, ugyanakkor sok olyan szabályt tartalmaz, amely szigorú megkötésekkel segíti az embert a helyes programírási szokások megtanulásában. Egyébként a nyelv jóval több mindenre használható, mint amire mi fogjuk használni: A manapság legelterjedtebb szoftverfejlesztési módszernek az angolul *prototyping*nak nevezett módszer egyik alapvető eleme. Ez arról szól, hogy minél gyorsabban írjuk meg a program prototípus változatát, amely sem sebesség, sem más szempont szerint nincs ugyan optimalizálva, de már letesztelhető az elképzelt program működése. Szükség esetén a módosítás is gyorsan elvégezhető. Ezután kerülhet sor a program egyes részeinek – végül az egésznek – egy gyorsabb nyelvű változatra átírására. A Python éppen egy olyan nyelv, amelyben gyorsan lehet programot írni, könnyen letesztelhető akár félkész állapotban is. Lehet, hogy nem optimális az így elkészült program, de a nyelv nagyon jó lehetőséget biztosít arra, hogy a programkód egyik vagy másik részét más nyelvre átírva egy optimálisabb kódot lehessen elkészíteni, így folyamatosan optimalizálva a program futását, míg végül a legoptimálisabb – esetleg már semmilyen Python nyelvű részt nem tartalmazó programot kapva.

Nagy előnye még a Python nyelvnek, hogy rendkívül sok kiegészítő létezik hozzá, így tényleg szinte mindenre alkalmassá tehető a legegyszerűbb rutinfeladatok elvégzésére írt kis *script*ektől a hatalmas adatmennyiséget feldolgozó nagy programokig egyaránt.

Egy programot a forráskódjából kétféle úton lehet futtatni:

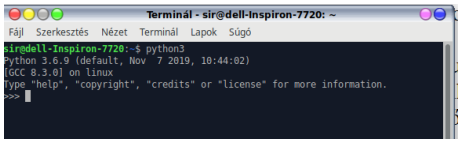
1. **Értelmező (Interpeter)** használata esetén az interpreter a program egyes utasításait egymás után végrehajtja. Ilyen esetben a számítógép helyett maga az interpreter hajtja végre a programot, amelynek jelen kell lennie a számítógépen a program minden egyes futásakor.
2. **Fordító (Compiler)** használata esetén a programot a fordítóprogram lefordítja a számítógép processzora által ismert nyelvre, azaz a gépi kódra. Ezután az így kapott állományból a program bármikor futtatható a fordítóprogram jelenléte nélkül is.
3. Hibrid megoldásként létezik olyan megoldás, amikor a fordítóprogram egy köztes kódot állít elő, amelyet a program futásakor egy értelmező hajt végre. Ilyet használ például a Java és C# is.

A Python nyelv az első megoldást alkalmazza, azaz a megírt programot egy értelmező hajtja végre. Tudni kell, hogy mivel az értelmezőt használó programozási nyelveket nem közvetlenül a számítógép hajtja végre, hanem az értelmező kvázi szimulál egy olyan számítógépet, amely megérti a forráskódot, így az értelmezős programvégrehajtás mindig lassabb a fordítóprogrammal gépi kódra lefordított programnál. Ellenben a fordítóprogram futását meg kell várni, ami egy összetettebb program és egy lassabb gép esetén hosszú ideig – szélsőséges esetben akár napokig – is eltarthat,

amit kivárni minden kipróbálás előtt eléggé lassíthatja a fejlesztést. Ezért lettek mostanság egyre népszerűbbek az értelmezőt használó nyelvek: ezeket gyorsan ki lehet próbálni, ami segíti a gyorsabb fejlesztést.

3.1. A Python nyelv értelmezője

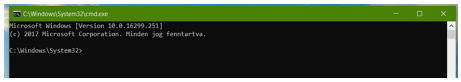
Kezdjük is az ismerkedést a Python értelmezőjével és benne néhány egyszerű utasítás végrehajtásával!



3.1. ábra: Python értelmező az Xfce terminálemulátor ablakában

Linux használata esetén nincs más teendőd, mint indítani egy parancsértelmezőt. Windows-on lényegében ugyanez a teendő, de általában jobban el van rejtve a parancsértelmező, mint a legtöbb Linux disztribúció grafikus felületén. Például Xfce grafikus környezetet használva a programindító panelen általában eleve ott van egy terminálemulátor nevű alkalmazás indítógombja, amelyre kattintva rövidesen lesz egy ablak a képernyőn, amiben a Linux (pontosabban Unix) parancsokat lehet begépelni.

> Linuxon valószínűleg azért nincs annyira eldugva a parancsértelmező, mert a legtöbb Linux felhasználó hozzá van szokva, hogy bizonyos rendszerműveleteket sokkal gyorsabb karakteralapú parancsként kiadni, mint a grafikus felületen összevissza kattintgatva. Ez ugyan Windows és Macintosh esetében is így van, de az ezen rendszerekhez szokott felhasználók általában nincsenek hozzászokva, hogy ennyire alaposan a kezükbe vegyék a saját számítógépük irányítását, ezért ezeken ritkábban szokás a parancsértelmezőhöz nyúlni.



3.2. ábra: A Windows parancsértelmezője (cmd.exe)

Akármelyik módszert is választjuk, a lényeg, hogy legyen egy parancsértelmezőnk, amelybe be lehet gépelni a `python3` parancsot. Amennyiben a gépen telepítve van a Python 3.x verziója, akkor valami ahhoz hasonló látvány fog fogadni, mint ami a 3.1. és a 3.2. ábrán látható. Linux esetében általában a Python alapból telepítve van, Windows esetében azonban alapból nincs a gépen, hanem külön telepíteni kell azt. Ez utóbbi esetben a Windows egy hibaüzenettel fogja jelezni, hogy nem érti, mit kell kezdeni a `python3` paranccsal.

Az értelmező először kiírja a pontos Python verziót, majd a legalsó sorban egy *prompt*ot jelenít meg, amely után utasítást vár, amit kész azonnal végrehajtani.

Próbáljuk is ki, hogyan lehet számológépként használni! Írd be a prompt után:

```
2 + 2
```

A szóközök az összeadásjel körül nem lényegesek ugyan, de illik figyelni az esztétikus és olvasható megjelenésre is, így azokat is gépeld be! Mi történik az Enter billentyű lenyomása után?

> Promptnak hívják, amikor egy karakteres felhasználói felület valamilyen kiírással jelzi, hogy várja a következő utasítást. Magának a parancsértelmezőnek is van promptja, amely általában (legalábbis Linux esetén) úgy van beállítva, hogy a felhasználó nevét és a munkakönyvtárát írja ki, valamint jelezze, hogy egyszerű felhasználói ($) vagy rendszergazdai jogkörrel (#) lehet parancsokat kiadni.
>
> A Python értelmező alapértelmezett promptja: >>>
>
> Ha az értelmező más promptot ír ki, az valamire utal, például arra, hogy egy zárójelet nem zártunk be.

A fenti utasítás valójában egy kifejezés, amelyben két érték összeadása szerepelt. A két érték a példában egyaránt kettő, a köztük levő + jel pedig az összeadás jele. Nyilvánvalóan vannak más műveletek is, amelyeket hasonlóan lehet alkalmazni. Mielőtt tovább megyünk, derítsd ki, hogy az alábbi feladatban szereplő műveleti jelek közül melyik milyen művelet elvégzését jelenti:

3.1. A műveleti jel két oldalán levő értékeket más értékekre cserélve szükség esetén, derítsd ki, hogy az alábbi kifejezésekben milyen művelet szerepel. Írd le az egyes műveleti jeleket a füzetedbe és írd melléjük, hogy szerinted melyik matematikai műveletnek felelnek meg, majd beszéljétek meg, hogy tényleg helyes-e a következtetésed!

```
2 + 2
2 - 2
2 * 2
2 / 2
2 // 2
2 % 2
2 ** 2
```

A fenti feladatban szereplő műveleteket szokás *operátor*nak is nevezni, míg az értékeket, amelyekkel az adott műveletnek dolgoznia kell a képletben *operandus*nak nevezzük.

3.2. Miután a fenti műveleti jelek jelentését sikerült megfejteni, próbálkozzunk összetett kifejezésekkel is! Például az alábbiakkal. Mit jelentenek az itt szereplő szimbólumok?

```
50 – 5 * 6
(50 – 5) * 6
(50 – 5 * 6) / 4
```

3.3. Mindezek után próbáld ki az alábbi kódrészletet is és derítsd ki, mit csinál pontosan!

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

Az utolsó feladatban látható nevek egy-egy *változót* jelölnek (később még sokat fogunk ilyeneket használni). A változó egy tároló, amelyben különböző értékeket lehet elhelyezni az *értékadó utasítás*okkal. Ezeket mutatja a példa.

Az értékadó utasításban egy = jel baloldalán mindig egy változó (vagy változóként értelmezhető kifejezés, magyarázatot lásd később) áll, jobb oldalán pedig egy kifejezés. Az utasítás végrehajtása a kifejezés értékének meghatározásával kezdődik, majd ez az érték kerül bele a változóba.

A változót úgy lehet elképzelni, mint a memória egy részét reprezentáló valamit. Vannak programozási nyelvek, amelyeknél a változót az első felhasználása előtt *deklarálni* kell, a Python esetében a változó akkor jön létre, amikor először értéket adunk neki, és alaphelyzetben mindaddig létezik, amíg a **del** utasítással nem töröljük. Ha a változó ismét egy értékadó utasítás bal oldalán szerepel, akkor az addigi tartalma elvész, és az új érték kerül bele, vagy ezt úgy is megfogalmazhatjuk, hogy az eddigi változó törlődik, tartalma elérhetetlenné válik és egy új változó jön létre ugyanazzal a névvel az új értékadás által meghatározott tartalommal. A két megfogalmazás nem ugyanaz!

Python esetén inkább az utóbbi felel meg a valóságnak, azoknál a nyelveknél, ahol előre deklarálni kell a változót, az előbbi érvényes.

Amennyiben a változó létrejött, már felhasználható az értéke bármely kifejezésben, ahova a változó aktuális értéke kerül behelyettesítésre. Éppen ezért ha olyan kifejezést adunk meg, amelyben egy olyan változónevet adunk meg, amely változó addig még nem jött létre értékadással, az hibát eredményez.

Az = mellett más értékadó műveletek is léteznek, erről szól a következő feladat:

3.4. Próbáld ki, mi történik az alábbi kódrészlet hatására!

```
alma = 1                        alma /= 2
alma += 5                       alma
alma                            alma = 5
alma -= 2                       banán = alma
alma                            alma //= 2
alma *= 2                       alma
alma                            banán %= 2
                                banán
```

Miért kaptad azokat az eredményeket, amiket kaptál? Mit csinálnak ezek az utasítások?

A fenti feladatok sikeres megoldása után már nem okozhat problémát bonyolultabb kifejezések Python nyelvű felírása sem. Az sem jelenthet gondot, hogy nem lehet emeletes törteket használni a kifejezésekben, hiszen a zárójelek alkalmazásával a műveleti sorrend befolyásolható. A következő feladatban három emeletes törteket is tartalmazó kifejezés található, amelyeket fel kell írnod a Python értelmezője által megemészthető kifejezésként:

$$\frac{23 \cdot 12}{15+3} \qquad \frac{\frac{4-11}{5+9}}{21} \qquad \frac{42+13}{3 \cdot 8} \cdot \frac{1024^{-4}}{51/6}$$

3.5. Próbáld ki a következő utasításokat:

```
gyumolcs = "alma"
ital = "bor"
gyumolcs + ital
```

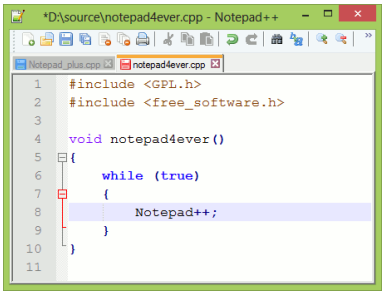
Miért az az eredmény, ami?

```
"kiri" * 5
```

Mit csinálnak ebben az esetben a + és a * műveleti jelek?

Próbáld ki, mi történik, ha kicseréled őket egymásra! Miért kapod, amit kapsz?

4. Fejlesztő eszközök



4.1. ábra: A notepad++ program képernyőképe a program honlapjáról (https://notepad-plus-plus.org/)

Ebben a leckében elsősorban a Python nyelvhez használható fejlesztő környezetek közül néhányat ismerhetsz meg. Az iskolában ezek egyikét fogod órán használni, de esetleg más célra otthon másik lesz a szimpatikusabb, így nem árt tudni, hogy van többféle is.

Figyelembe véve, hogy a program forráskódja egy egyszerű szöveges állomány, amelynek a kiterjesztése és esetleg az első néhány karaktere vagy sora utal arra, hogy mit tartalmaz, a programot egyszerű szövegszerkesztő programban is meg lehet írni. Itt persze nem a szövegformázást is lehetővé tevő, angolul *Word Processzor* néven emlegetett, hanem a *Word Editor* néven emlegetett szövegszerkesztőkről van szó, mint a Windows *Jegyzettömb*je, a Linuxon a *Mousepad* vagy a *Kwrite*. De a grafikus felület mellőzésével, akár karakteres környezetben is találhatóak olyan szövegszerkesztők, amelyekkel lehet egy programkódot szerkeszteni (pl.: *vim*, *emacs*, *nedit*).

Ennél a „fapados” megoldásnál kicsivel több megoldást kínál Windows alatt például a *notepad++* (4.1. ábra) nevű program, amely már képes szintaxiskiemelésre is. A *szintaxiskiemelés* (syntax highlight) azt jelenti, hogy a forráskód különböző funkciójú elemeit más-más színnel jelzi, ami nagy segítség tud lenni például azzal, hogy a színekhez hozzászokva hamarabb észrevehető a nem megfelelő színből, hogy valamit elírtunk. Erre a szintaxiskiemelésre egyébként a Linux esetén említett példák is képesek.

Az igazán profi programfejlesztés során azonban kifejezetten az adott programozási nyelv lehetőségeit ismerő szerkesztő programokat használnak. Ezek a programok az ún. *fejlesztő környezetek*, vagy az angol eredeti nevük, az *Integrated Development Environment* rövidítéséből származó **IDE** néven emlegetett programok. Ezek a programok általában nem csak a forráskód begépelését teszik könnyebbé, hanem további, a fejlesztés során szükséges műveletet is elérhetővé tesznek. Általában egy jó fejlesztő környezet használata során lehet, hogy a program teljes fejlesztési folyamata az IDE-n belül elvégezhető.

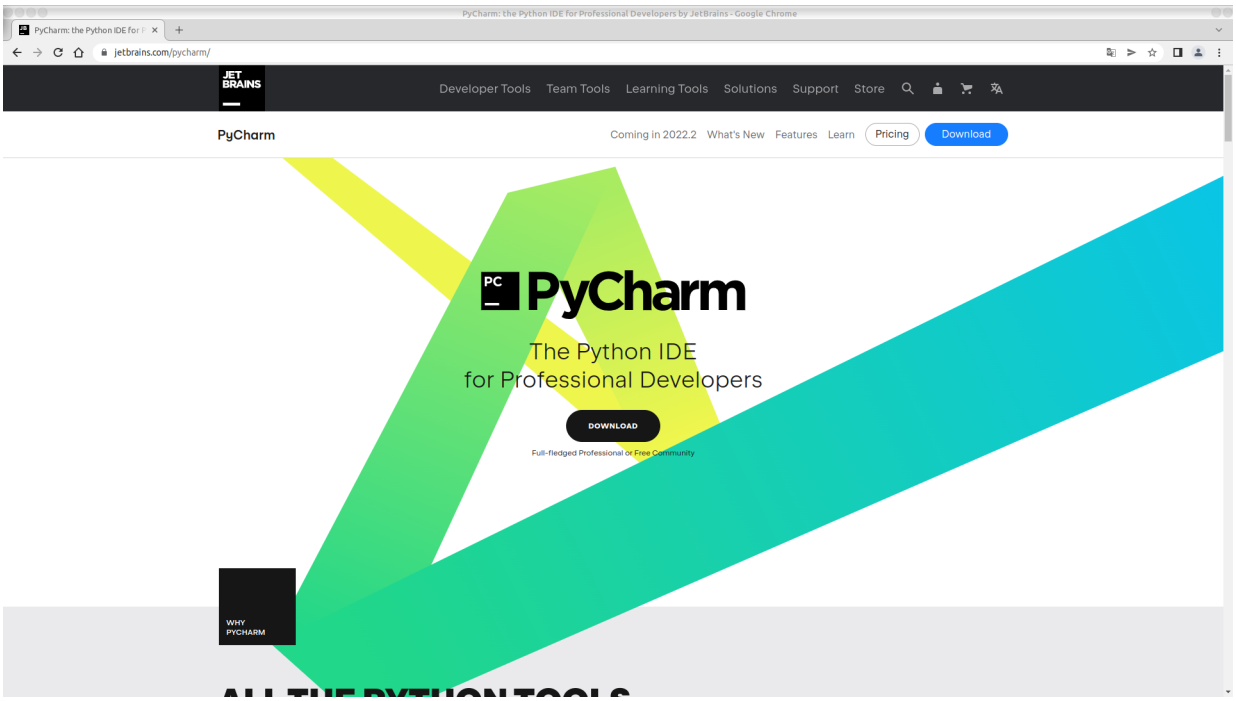
Léteznek egy adott nyelvre specializálódott fejlesztő eszközök. De léteznek olyan programok is, amelyek akár azonnal, akár megfelelő bővítmények hozzáadásával akár több, különböző programozási nyelv fejlesztéséhez is biztosítani tudják a megfelelő eszközöket.

4.1. PyCharm

Az egyik ilyen több nyelvhez is használható fejlesztő környezetet (Java nyelven) a *JetBrains* fejlesztette ki. Ennek a Python nyelvhez készült változata a *PyCharm*.

Ez egy Shareware program, azaz ingyenes és fizetős változata is van, ezért vigyázni kell a letöltésnél, hogy a jó változatot válasszuk. Otthoni használatra a *Community* változata nagyon jól használható. Bővítmény telepítésével *Educational* változattá alakítható, amely tartalmaz olyan oktatóanyagokat – természetesen angolul –, amelyek nagy segítséget jelenthetnek a nyelv elsajátításában. A fizetős változat, a *Professional*, amelynek többletszolgáltatását csak az igazán profik veszik igénybe, így arra semmi szükséged nem lesz egyelőre!

Mivel a PyCharm többek között azzal is segíti a fejlesztést, hogy szinte minden karakter begépelése után menti a szerkesztett programállományt, hálózati környezetben – mint az iskolai is –, ahol a felhasználók adatai egy központi szerveren találhatóak, rendkívül komoly hálózati adatforgalmat generál, így az iskolai környezetben érdemben nem használható. Ám otthoni környezetben nagyon jól használható eszköz lehet.



4.2. ábra: A PyCharm nyitóoldala (http://jetbrains.com/pycharm/)

IDE KÉSŐBB KERÜLNI FOG EGY HIVATKOZÁS A PROGRAM LINUXRA ILLETVE WINDOWSRA TELEPÍTÉSÉT BEMUTATÓ YOUTUBE VIDEÓRA, HA AZOK A VIDEÓK MAJD ELKÉSZÜLNEK...

4.2. Visual Studio Code, röviden: VS Code

Egy, a PyCharmnál sokoldalúbban használható eszköz a Microsoft által fejlesztett *Visual Studio Code*, röviden **VS Code**. Ez a program jó példa arra, hogy a Microsoft csak a Windows-t és az Office-t nem tudja eléggé felhasználóbarátra és probléma nélkül működőre fejleszteni: a VS Code egy kifejezetten sokoldalú fejlesztő környezet. Viszont sajnos nem olyan egyszerű használatba venni, mint a PyCharm-ot, mivel a telepítése után azonnal még semmilyen programozási nyelvet nem ismer elég részletesen. Persze valamennyire ismer néhányat, de az igazán fejlesztőbarát használathoz az adott programozási nyelvhez írt bővítményeket is telepíteni kell. Ám ezekkel a kiegészítőkkel már egy nagyon kényelmes segítséget tud nyújtani gyakorlatilag bármilyen nyelven történő fejlesztéshez.

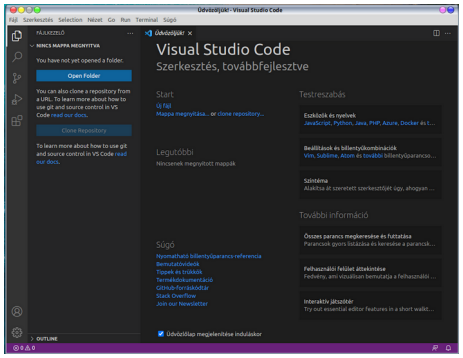
A 4.3. ábra az eredeti sötét színtémával, a 4.4. ábra pedig a világos témára váltás után mutatja a program nyitóképernyőjét. A színtémát a *File* / *Beállítások* / *Color Theme* menüponton keresztül lehet megváltoztatni.

A nyitóképernyő jobboldalán, az „*Eszközök és nyelvek*”, angol változatban „*Tools and Languages*” részen belül található linken (kék színű szövegen) keresztül lehet a leggyorsabban eljutni a nyelvtelepítési lehetőséghez. Aki már valamennyire ismeri a programot, az a bővítménytelepítőt a baloldali, a színséma állításától függetlenül sötét hátterű sávon keresztül közvetlenül is elérheti. Egy adott nyelv használatához is nagyon sok bővítmény érhető el, amelyek más-más segédeszközt kínálnak az adott programnyelven történő fejlesztéshez. Magának az egyes bővítmények leírásának az átböngészése is rendkívül sok időt tud igénybe venni, emiatt a program használatba vétele elég időigényes lehet, ám a későbbiekben nagyon megkönnyíti a programfejlesztést. Ám ha két gépen nem ugyanazok a bővítmények vannak telepítve, akkor azokon a gépeken a program nem ugyanúgy fog viselkedni!

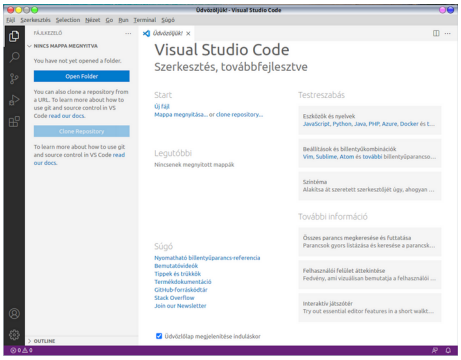
Emellett a PyCharm esetében említett hálózati adatforgalommal kapcsolatos probléma is felléphet, ha a program nincs megfelelően beállítva. Ezeknek a problémáknak a kiküszöbölésére az iskolai környezetben a program már előre be van állítva, az órai munkában hasznos bővítmények minden gépen telepítve vannak, így ajánlatos azokhoz nem nyúlni! Főleg mert nem csak a Python nyelv használatához, de a HTML+CSS nyelv, illetve az adatbázis-kezelés céljára hasznos bővítmények is előre vannak telepítve.

Így itt csak azért szerepel a programozás során használt bővítmények felsorolása, hogy szükség esetén az otthoni gépeden is be tudd ugyanazt a környezetet állítani, amit órán használni fogsz.

- **Python**: Ez a bővítmény a Microsoft hivatalos bővítménye a Python nyelvhez. A leírása tartalmaz segítséget arra is, hogy Windows esetében hogyan telepítsd fel a Python értelmezőt, ha ez még nem történt meg (Linuxon ez általában már nem szükséges). A bővítmény az alább felsoroltak közül néhány bővítményt automatikusan feltelepít magával.
- **Pylance**: A bővítmény segítséget nyújt hatékony Python nyelvű programíráshoz. A Python bővítmény részeként történik meg a telepítése, így nem kell külön telepíteni.
- **Jupyter**: Ez a bővítmény is a Python csomag révén automatikusan települ. A Jupyter egy olyan szolgáltatás, amelyről kicsit szót ejtünk a Google Collaboratory szolgáltatás bemutatásakor.
- **Jupyter Keymap** és **Jupyter Notebook Renderer**: Ezeket a Jupyter telepíti fel, így ezekkel sem kell külön foglalkozni.



4.3. ábra: A VS Code kezdőképernyője sötét témával



4.4. ábra: A Visual Studio Code nyitóképernyője közvetlenül a telepítés után, világos témára váltva.

- **Hungarian Language Pack for Visual Studio Code**: Mivel a VS Code az operációs rendszer nyelvi beállításai alapján érzékeli, hogy magyar nyelvű felületet kell biztosítani, minden indításkor felkínálja ezt a csomagot mindaddig, amíg nem telepítjük. Így külön nem érdemes foglalkozni a telepítésével.

Az iskolai gépeken van még néhány csomag telepítve, de azok már nem a Python nyelvhez tartoznak. Az adott témakör elején megtalálod majd azoknak a bővítményeknek a bemutatását, amelyek ott lesznek hasznosak.

4.3. Google Colaboratory

Meg kell még említeni egy harmadik eszközt is, amivel Python nyelvű programokat lehet írni: A Google által kifejlesztett online eszköz, a **Colaboratory** egy olyan dokumentum, amely *Jupyter Notebook*nak számít. A Jupyter Notebook tartalmazhat szöveges dokumentumban megszokott módon formázható szöveget, valamint néhány, erre alkalmas nyelven írt programkódot is. A Python az egyik nyelv, amelyet a Jupyter Notebook megért.

Előnye, hogy online dokumentum, amelyet a Google Drive tárol, alapértelmezésben egy speciális mappában, de gyakorlatilag bármely Drive mappában elhelyezhető. A többi Drive objektumhoz hasonlóan megosztható másokkal, így közösen szerkeszthető. Emellett a programblokk helyben végrehajtható, a végrehajtás eredménye megjelenik a dokumentumban. A programkód végrehajtását ilyenkor nem a mi számítógépünk végzi, hanem a Google erre a célra rendelkezésre bocsátott Cloud szerverén fut.

Létezik a Google Collaboratory-hoz Android alkalmazás is, így akár a telefonunkon vagy a tabletünkön is szerkeszthetjük, futtathatjuk programjainkat a segítségével.

Az állománykezelést a hagyományos Python módszertől eltérően lehet az ebben található programblokkokban elérni: a Google Drive tartalmát lehet elérhetővé tenni, nem a helyi számítógépen levőket.

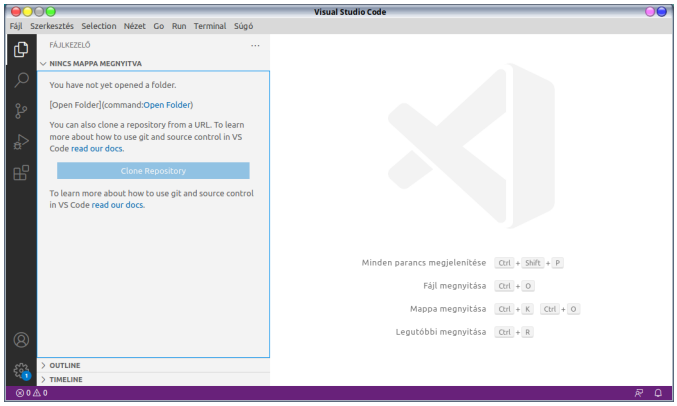
A szolgáltatás elérhető a https://colab.research.google.com/ címen.

4.4. Az iskolai munkakörnyezet

Egy szoftver fejlesztése nem egyetlen programállomány szerkesztéséből áll, hanem sok, egymással összefüggő, ezért együtt kezelendő állomány tartozik hozzá. Ezért a legtöbb modern fejlesztőkörnyezet nem önálló programállományok szerkesztését, hanem általában egy projekt kezelését, és a projekthez tartozó állományok kezelését teszi lehetővé. Egy projekten belül természetesen sokféle állomány lehet együtt, nem csak programállományok. Például ha a programnak grafikus felülete is lesz, akkor a projekthez tartoznak azok a képek is, amelyek például a grafikus felület nyomógombjain megjelennek. Ezért a projekt könyvtárán belül sokszor még alkönyvtárak is találhatóak, amelyekbe a program különböző részei illetve a különböző jellegű állományok csoportosíthatóak – egyben elkülöníthetőek a projekt más részeitől.

A programozási ismeretekben aligha jutunk odáig, hogy egy-két állománynál többet igényeljen egy-egy programunk, így a fentiektől eltérően a középiskolai tanulmányaid során elkészülő valamennyi programot el lehet helyezni egyetlen könyvtárba. Ezt a könyvtárt fogjuk a projektkönyvtárunknak tekinteni, benne valamennyi programot egy projekt állományainak.

Ennek megfelelően az első teendő, még a fejlesztő környezet első elindítása előtt létrehozni a leendő programjaink tárolására szánt könyvtárt. Az egyszerűség kedvéért ez legyen a saját könyvtáradon belül (Linux esetén ezt szokás home-könyvtárnak nevezni) `python` a neve.



4.5. ábra: A VS Code indítás után, még megnyitott munkakönyvtár nélkül

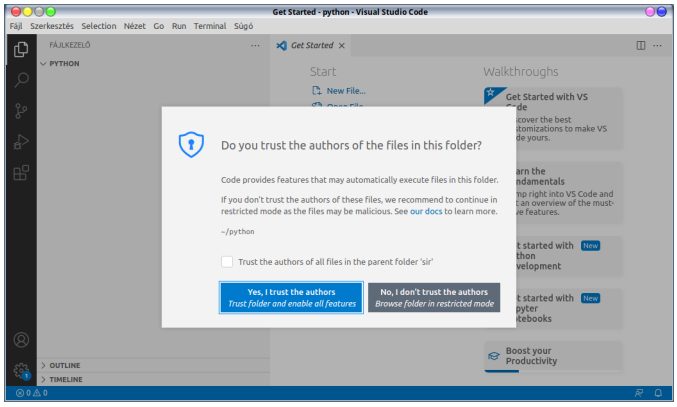
Tehát például ha az azonosítód *d0proodo*[1] lenne, akkor a `/home/d0proodo/python` könyvtárat kellene létrehoznod. A továbbiakban erre a könyvtárra ezt az azonosítót feltételezve fog a dokumentum hivatkozni. Értelemszerűen a `d0proodo` helyére a saját azonosítódat kell behelyettesítened.

Windows esetében a leendő könyvtárat szintén `python` néven, de máshol kell létrehoznod.

Ha a könyvtárat sikeresen létrehoztad, akkor ideje elindítani a VS Code programot.

A program elindítása után a baloldali sávon a legfelső gombot választva lehet előcsalni a „*Fájlkezelő*” sávot. Ez a 4.5. ábrán láthatóhoz hasonló tartalmú lesz mindaddig, amíg nem nyitunk meg egy könyvtárat. Amennyiben itt található egy nagy, kék gomb „Open Folder” felirattal akkor arra, ha képen látható a tartalom, akkor a kék „Open Folder” szövegre kell kattintani. Ekkor egy könyvtárválasztó ablak segítségével kiválaszthatjuk a használni kívánt könyvtárat, vagy ha az még esetleg nem létezik, akkor létre is hozhatjuk azt.

Amennyiben még nem használtuk ezt a könyvtárt a VS Code-ban, akkor a program megkérdezi, hogy megbízunk-e könyvtár tartalmának a szerzőjében (4.6. ábra).



4.6. ábra: Megbízható-e az újonnan kiválasztott könyvtár tartalma?

Mivel ez a könyvtár általunk írandó programokat fog tartalmazni, így a kérdésre a válasz nagy valószínűséggel az igen lesz. Ezután a belső téglalap erre a könyvtárra remélhetőleg többet már nem fog megjelenni, mert a program megjegyzi, hogy ez a könyvtár megbízható.

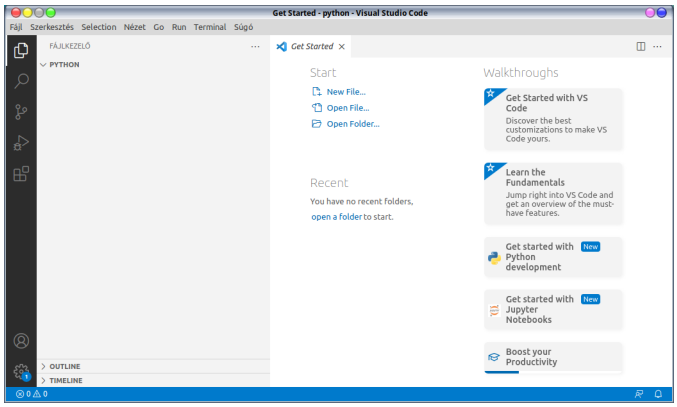
A könyvtárválasztó szöveg helyett a baloldali sávban a könyvtár tartalma jelenik meg – jelenleg üresen, hiszen az a könyvtár még nem tartalmaz semmit.

A jobboldali részben fognak majd a megnyitott állományok megjelenni, ahogy a fenti, most „*Get Started*” feliratú fül sugallja, egyszerre akár több is. Sőt, ezt a részt akár vízszintesen, akár függőlegesen további részekre lehet osztani a jobb felső sarokban levő gombbal, illetve a három ponttal elérhető menün keresztül. Így akár ugyanaz az állomány több példányban is megjeleníthető, ami hosszabb programoknál segíthet annak különböző részeire vonatkozó szerkesztések elvégzésében.

A megnyitott fülek bármelyikét – természetesen a hozzá tartozó tartalommal együtt – a fül jobb szélén levő X-re kattintva be lehet zárni. Itt célszerű megjegyezni, hogy ha az X helyén egy tömör kör látható, az arra utal, hogy a fülhöz tartozó állomány nincs lemezre mentve. Bár a VS Code alapbeállításban megjegyzi bezárás után a megnyitott lapok tartalmát, ezt a lehetőséget nem ajánlott kihasználni, mentsük rendszeresen a szerkesztés alatt álló programjainkat!

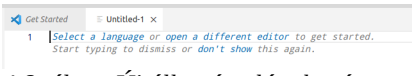
1 Ez a név a Próba Ödön kitalált névre utal, amit a Google felhőszolgáltatásainak tesztelésére 2020-ban, amikor nem volt 9/D osztály a Gimnáziumban, létrehozott fantom felhasználó. Az ő azonosítója a Linux rendszeren alkalmazott rendszer szerint *d0proodo* lenne.

Próbáljuk is ki a fejlesztő környezetet az első Python nyelvű programunk megírásával! Ehhez első lépésben egy új állományt kell létrehoznunk. Erre több lehetőség is van.



4.7. ábra

1. A baloldali sávra húzva az egér mutatóját, felül megjelenik néhány gombocska, köztük a bal szélsővel lehet új állományt létrehozni. Ekkor alatta egy állomány helyén megjelenő sávba lehet beírni az állomány leendő nevét. A létrejött üres állomány a jobboldalon megnyílik egy saját füllel.
2. A CTRL+N billentyűkombináció úgy nyit új fület a jobboldalon, hogy még nincs az állománynak neve. Ilyenkor a 4.8. ábrán látható szöveg jelenik meg az egyébként még nem létező állomány tartalmának helyén. A „*Select a language*” feliratra kattintva megjelenik egy legördülő lista, amelyben a fejlesztő környezet által ismert nyelvek jelennek meg. De nem kell keresni, elég begépelni a „*python*” szöveget, és a program kiválasztja azt. Másik lehetőség, rögtön elkezdeni gépelni a program kódját, és a VS Code majd rájön a nyelvre. Ez annyiból előnytelenebb, hogy amíg nem jön rá, addig nem kapunk meg minden segítséget.
3. Az előző ponthoz hasonlóhoz juthatunk a program menüjét használva is.



4.8. ábra: Új állomány létrehozása

A második változatnál figyelni kell arra, hogy az állományunk még nem rendelkezik névvel sem. De mivel a későbbiekben is rendszeresen – mondjuk amikor megállunk a gépeléssel, akkor automatikusan – célszerű menteni, így ez nem jelent nagy problémát. A CTRL+S billentyűkombinációval a leggyorsabb a mentést kezdeményezni.

Figyeljünk arra, hogy a Python programokat tartalmazó állományoknak mindig a `.py` karakterekkel kell végződniük, mivel ez jelzi, hogy ez egy Python nyelvű program. Nevezzük el ezt az első programot `elso.py` néven. Figyelj arra, hogy az állomány nevébe soha nem írunk sem szóközt, sem ékezetes betűt! Ezután gépeld be az alábbi programot!

elso.py > ...

```
1  #! /usr/bin/python3
2  """
3  elso.py
4  Az első programunk, amelyben megismerkedünk a kiírást és a beolvasást
5  végző függvényekkel.
6  """
7  nev = input("Hogy hívnak? ")
8  print("Szia, ", nev)
9
```

Remélhetőleg fel fog tűnni, hogy a sorszámokat nem kell begépelni, azokkal a VS Code jelzi, hogy éppen a program hanyadik soránál jársz. A sorszámok később jól jönnek akkor is, ha valami hiba van a programban, mert az esetleges hibaüzenetek kiírják, hogy hanyadik sorban találták a hibát. Ezen felül most jól fog jönni ahhoz is, hogy sorról sorra végignézzük, melyik sor mit jelent.

Akik sokat ülnek számítógép előtt, azoknak különösen megterhelő, ha a monitoron levő látvány nagyrészt világos színeket tartalmaz. Éppen ezért a szoftverek fejlesztésére szánt programok – de újabban más sokat használt program is – alapvetően sötét színösszeállítást alkalmaznak. Azonban az ilyen sötét háttéren világos betűket tartalmazó ablakokról készült képernyőkép, főleg telefonon vagy tableten nézve, illetve kivetítve gyakorlatilag olvashatatlan. Ezért kihasználva, hogy a VS Code esetében a színösszeállítás könnyen átkapcsolható világos alapúra, ebben a dokumentumban a képernyőképek, mint a fenti programódot mutató, világos színösszeállításuak lesznek. Amennyiben ilyen színösszeállítást akarsz a saját képernyődön látni az eredeti, a szemet kevésbé rongáló helyett, akkor a *File* / *Beállítások* / *Color Theme* menüponton keresztül választhatsz más színösszeállítást magadnak.

Lássuk sorról sorra a program jelentését!

```
#! /usr/bin/python3
```

Az első sorban az úgynevezett script nyelvek előírják, hogy egy `#!` karaktersorozat után szerepeljen annak a programnak a parancsértelmezőbe írható parancsa, amelynek az állomány tartalmát fel kell dolgoznia. A karaktersorozatnak a felkijáltójel után egy szóköz is a része! A példában ez azt jelenti, hogy a Python értelmező az *usr* könyvtár *bin* alkönyvtárában található és az állomány neve `python3`. A megadott könyvtár tartalmazza Linux, sőt általában Unix rendszerek esetében a legtöbb felhasználói program elindítására alkalmas állományt. A Python nyelv 3. verziójához tartozó értelmező érhető el a `python3` program elindításával, ami már ismerős lehet, hiszen amikor közvetlenü az értelmezőnek adtunk ki utasításokat, akkor is ezen a néven indítottuk el.

Windows esetében ezt a sort a legtöbbször elhagyják. Amennyiben a Python értelmező helyesen van a gépre telepítve, akkor remélhetőleg ilyen esetben is megtalálja azt a gép.

A sor arra is jó, hogy ha egy új állomány létrehozásakor nem adjuk meg a használni kívánt nyelvet, akkor a VS Code ebből a sorból felismeri a Python nyelvet és innentől remélhetőleg elérhetőek a remélt szolgáltatások.

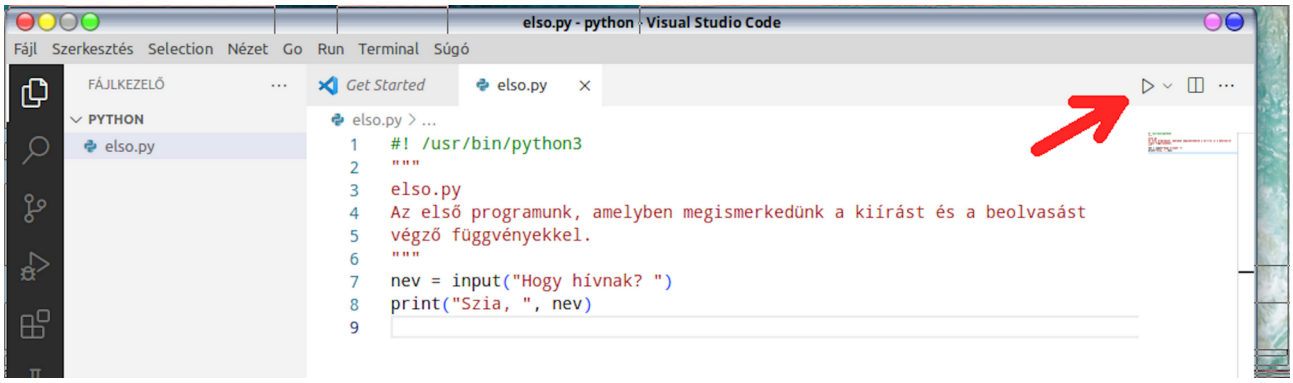
Ezután a 3–6. sorban egy dokumentáló megjegyzés látható. A Python nyelvben a megjegyzéseket alavetően egy kettőskereszt – és azt követő szóköz – után írjuk, ám az itt látható módon lehet megadni a program elején, hogy mit is fog az tartalmazni. A megjegyzésekről még lesz szó, mivel nagyon fontos elemét alkotják a programírásnak annak ellenére, hogy látszólag feleslegesek.

A 7. sor egy értékadó utasítást tartalmaz, amit már ismersz:

```
nev = input("Hogy hívnak? ")
```

Az utasítás létrehoz egy *nev* változót, amelynek értéke… Nos, ez már új elem. Eddig csak konstans értéket vagy változót használtunk a kifejezésekben, itt viszont egy függvényhívás szerepel. A függvényeket arra használjuk, hogy egy összetettebb műveletet egyetlen utasításként adjunk meg. Ebben az esetben egy, a nyelvben eleve definiált, hivatalosan *builtin*, azaz *beépített* függvényt. Ez a függvény az **input()**, amellyel a felhasználótól lehet kérdezni valamit. A függvényhívás mindig a függvény nevéből és paraméterlistájából áll. Utóbbi akkor is kötelező, ha nem akarunk megadni paramétert. Ezt szokás jelezni azzal, hogy a függvényre a szövegben úgy hivatkozunk, hogy egy zárójelpárt írunk a neve mögé, jelezve a paraméterlista szükségességét.

Az **input()** függvénynek azt a szöveget kell megadni, amit fel kell tenni kérdésként. Most ez a „*Hogy hívnak*?” szöveg lesz. Figyelj a kérdőjel utáni szóközre is! Ez azért kell, mert a számítógép a kiírt szöveg után rögtön várja a felhasználó válaszát. Ha nem teszünk a kiírandó szöveg végére szóközt, akkor rögtön a kérdőjel után fog villogni a kurzor, ami nem szép látvány.



4.9. ábra: A piros nyíl jelzi a program futtatását elindító gombot.

A programon belül ha leírunk egy kifejezést, arra nem fog megjelenni a kifejezés értéke, mint az értelmezőben jelent meg. Ehhez egy másik függvényt, a **print()** függvényt kell alkalmaznunk. Ennek a függvénynek több paramétere is lehet – és most van is kettő –, amelyeket vesszővel kell egymástól elválasztani.

```
print("Szia, ", nev)
```

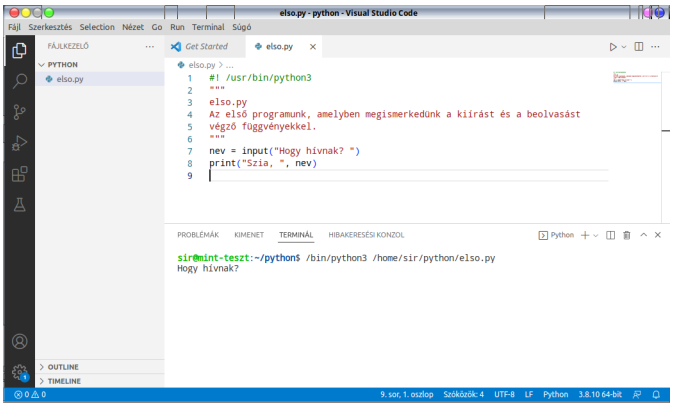
Az első paraméterben szereplő kifejezés egy szöveg, amelynek eredményeként a szöveg tartalma íródik ki a képernyőre. A második paraméterben levő kifejezés egyetlen változóból, az előző utasításban szereplő változóból áll, így ennek a változónak az értéke íródik ki a képernyőre.

A program utolsó, 9. sora valójában csak egy üres sor. A Python nyelvű programok felépítésére vonatkozó szokások előírják, hogy a programot mindig egy üres sorral zárjuk le. Ez nem jelent mást, mint hogy a 8. sor végén még egy Enter billentyűt kell gépelni.

Próbáljuk is ki a programunkat! Ha sima jegyzettömböt használtunk volna a megírására, akkor most szükségünk lenne egy parancsértelmezőre abban a könyvtárban, amelyben a program van, és Linux esetén még néhány előkészületre, mivel alaphelyzetben egy állomány nem végrehajtható.

Ám egy fejlesztő környezet azt is lehetővé teszi, hogy az éppen szerkesztett programot azonnal ki lehessen próbálni. A program futását lehet kezdeményezni az ablak jobb szélén levő nyíl alakú gombbal, amelyet a 4.9. ábrán egy piros nyíl jelez. Amennyiben ez a nyíl nem jelenik meg, akkor vagy nem jó néven mentetted el a programot (például hiányzik a végéről a .py végződés) vagy a Python bővítmény nincs telepítve vagy valamiért nem aktív.

A gombra kattintva, egy új munkaterület nyílik meg a program alján (4.10. ábra), amelyben a program végrehajtása látható. Az első sorban a parancsértelmező szokásos promptja után látható az az utasítás, amelynek kiadásával a VS Code elindítja a program futását. Mint az ábrán látható, ez nem egyezik meg azzal, amit a program elején megadtam, de valójában ugyanaz az értelmező fut ebben az esetben is. Ha a fejlesztő környezeten kívül futtatnám a programot, akkor az általam megadott helyen keresné a gép az értelmezőt.



4.10. ábra: A program futtatása

A második sorban megjelenik az **input()** paraméterében szereplő szöveg. A munkaterületbe bele kell kattintani a válaszoláshoz, hogy ez a munkaterület fogadja a billentyű-eseményeket. Ha beírsz egy szöveget, nyilván a nevedet, az még kevés: az Enter billentyű jelzi a programnak, hogy befejezted a választ. Az Enter lenyomásának ha-

tására a program továbblép a következő utasításra, amely a **print()** meghívása két paraméterrel. Figyeld meg a kiírás eredményét! Biztos, hogy jó szöveget adtunk meg a függvény első paraméterében? A következő szakaszban szerepel a függvény pontos leírása, ott megtudhatod, miért van két szóköz a vessző után, és arra is választ kapsz, miért nem írtunk a *nev* változó kiírása után még egy felkiáltójelet is, hogy a kiírás megfeleljen a magyar nyelv szabályainak.

Ismerkedjünk egy kicsit a lehetséges hibákkal, amelyeket elkövethetünk a program begépelése során!

Alapvetően kétféle hibát követhet el egy programozó:

Szintaktikának nevezzük a program nyelvtani szabályait. Az ezen szabályoknak nem megfelelő kód által okozott hibát, azaz, amikor a program szövegében valahol nyelvtani hiba van, **szintaktikai hibá**nak nevezzük.

Szemantikának nevezzük a program jelentését, azaz azt, hogy mit kell a programnak csinálnia. Amennyiben a program nem az elérni kívánt módon viselkedik, azaz mást csinál, mint amit kellene, akkor **szemantikai hibát** tartalmaz. A szemantikai hiba tehát azt jelenti, hogy szintaktikailag ugyan rendben van a kód, de a kódban valami nem jól van megírva, ezért mást csinál a program, mint kellene.

Előfordulhat, hogy már eleve hibásan gépelted be a programot valahol, ezért már találkoztál valamilyen hibaüzenettel. Most azonban állíts elő ilyeneket szándékosan és figyeld meg, hogy melyik szintaktikai hibát hogyan jelzi az értelmező!

Mi történik például ha a 8. vagy a 7. sorban a szöveg valamelyik oldaláról törlöd az idézőjelet? Mi történik, ha kettő idézőjel szerepel? Mi történik, ha az `input` szó helyett például `inut` szerepel? Mi történik, ha a 7. sorban az értékadás `=` műveleti jelét törlöd? Mi történik, ha elhagyod a zárójel egyik vagy mindkét felét valahol?

Mi történik, ha a 8. sorban a **print()** függvény elé begépelsz egy szóközt és így a függvény neve a második oszlopban kezdődik?

Melyik hiba esetén mi a hibaüzenet szövege, melyik sorra jelzi a hibát?

4.5. A leckében szereplő beépített függvények

```
print(kifejezés, …, end='\n', sep=' ', file=sys.stdout, flush=False)
```

A **print()** függvény a paraméterében megadott kifejezés vagy kifejezések értékét kiírja alapértelmezésben a standard kimenetre. A kifejezés típusa bármi lehet.

A speciális paraméterek jelentése:

sep: Amennyiben egynél több adat szerepel, akkor azokat egymástól a *sep* paraméterben megadott szöveggel elválasztva jeleníti meg. Alaphelyzetben ez a karakter egyetlen szóközt tartalmaz, azaz a kiírandó értékek közé egy szóköz kerül. Például ha vesszővel elválasztva szeretnénk az értékeket kiírni, akkor ennek a paraméternek értékül a vesszőt tartalmazó szöveget kell adni:

```
print(12, 5, alma, sep=", ")
```

ennek az utasításnak a hatására amennyiben *alma* változó értéke 25, akkor ez jelenik meg:

```
12, 5, 25
```

end: A kiírás végén az *end* paraméterben szereplő szöveg is kiírásra kerül. Alapértelmezett értéke egy új sor jel, azaz a kiírás végén a következő kiírandó szöveg új sorba fog kerülni. Ha szeret-

nénk több kiírást egymás után, egy sorban megjeleníteni, akkor ezt a paramétert kell például az üres szövegre állítani (pl. `end=''`)

file: Ez a paraméter adja meg, hogy a kiírás ténylegesen hová kerüljön. Alapértelmezésben az a standard kimenet. A standard kimenet egy parancssorból indított program esetén a parancsértelmező ablaka, VS Code esetén az a munkaterület, amelyben a programfutás eredményét lehet látni. Ezen a paraméteren keresztül azonban arra is lehetőség van, hogy a kiírás egy állományba történjen.

flush: Ha igaz (True) az értéke, akkor azonnal megtörténik a kiírás. Ha az értéke hamis (False), akkor késleltetett írás történhet. A paraméternek akkor van értelme, ha a *file* paramétert átállítjuk egy állományra. Ekkor ugyanis elképzelhető, hogy az operációs rendszer az állományműveletek optimalizálása érdekében késlelteti a kiírást. Ezt lehet a paraméter igaz értékével felülbírálni, azaz kényszeríteni az operációs rendszert, hogy azonnal hajtsa végre a kiírást.

`input(prompt='')`

Amennyiben van paramétere a függvénynek, akkor az abban szereplő szöveget kiírja a képernyőre. Ezután, vagy akkor, ha nincs paraméter, egy Enter lenyomásáig olvas a standard bemenetről (általában a billentyűzetről). A beolvasott tartalmat szövegként adja vissza.

A paraméter mindenképpen egy szöveg típusú érték, több értéket nem lehet megadni, ha más típusú paramétert adunk meg, azt az értelmező szöveggé konvertálja, mielőtt a függvény azt megkapná.

4.1. Javítsd ki a korábbi programban a print() függvényt úgy, hogy nyelvtanilag szabályos szöveget írjon ki! Például ha a *nev* változóban a „Zoli” szöveg van, akkor a kiírt szöveg így nézzen ki: `Szia, Zoli!`

5. Egyszerű programok

Az eddigi ismeretek alapján már néhány egyszerű programot meg tudsz írni. Az alábbiakban szereplő feladatok előtt azonban gyűjtsünk össze még néhány hasznos tudnivalót:

Az előző lecke végén már szerepelt az eddig előfordult két függvény részletes bemutatása. Volt benne olyan információ is, amelyre valószínűleg soha nem lesz szükséged (például a **print()** függvény esetén a *file* és a *flush* paraméter). Lesz még több hasonló függvény is, amit ebben illetve a későbbi leckékben megtalálsz. Amennyiben ezek valamelyikét később vissza szeretnéd keresni, akkor a Tárgymutatóban megtalálod (a „Python függvények” részen belül), hogy melyik leírása melyik oldalon található.

5.1. Műveleti jelek

Számértékeken az alábbi műveletek végezhetőek el fentről lefelé növekvő prioritással (a prioritás határozza meg a műveletek végrehajtási sorrendjét: a magasabb prioritású művelet következik előbb):

1. `+` összeadás; `-` kivonás
2. `*` szorzás; `/` osztás; `//` egész osztás; `%` maradékképzés
3. `**` hatványozás
4. `(` és `)` a zárójelezéshez

Fontos, hogy a zárójelezéshez csak a kerek zárójeleket használjuk, a többi zárójelfajtának más a jelentése!

Szövegeken a `+` és `*` művelet használható, hasonló prioritás van köztük.

A `+` ebben az esetben két szöveg összefűzését jelenti. A műveleti jelnek mind a két oldalán szövegnek kell lennie.

A `*` egy szöveget a megadott számú alkalommal önmagával összefűz. A műveleti jel egyik oldalán a szöveg, a másikon az az egész érték szerepel, amennyiszer a szöveget önmagával össze kell fűzni.

5.2. Változók és az értékadó műveletek

Az előző leckében szerepelt program már tartalmazott változót, ahogyan az értelmezővel ismerkedés során is előkerültek változók.

> A változók olyan névvel ellátott tárolók, amelyekbe különböző értékeket lehet elhelyezni. Az érték elhelyezésére szolgálnak az értékadó utasítások. A legegyszerűbb értékadó utasítás a `=` műveleti jellel adható meg. Az értékadó műveleti jel bal oldalán mindig az a változó áll, amelyben el akarunk tárolni egy értéket, a jobboldalán pedig az a kifejezés áll, amelynek értékét el akarjuk tárolni. Bár a többi értékadó művelet a változó pillanatnyi értékét is felhasználja.

A változó neve egy azonosító, mint amilyen a függvények neve is. Az azonosítónak mindig betűvel kell kezdődnie, amelyet betű, számjegy vagy szintén betűnek számító aláhúzásjel (_) követhet. A Python elvileg megengedi betűként az angol ábécében nem szereplő betűk használatát is, azonban ezt inkább kerüljük el és csak az angol ábécé betűit használjuk, mivel más programozási nyelvek esetleg csak ezeket engedik meg!

Fontos, hogy a kis- és nagybetűk különbezőnek számítanak. Elterjedt szokás szerint a változók nevében csak kisbetűket használunk. Célszerű a változóknak mindig olyan nevet adni, amely utal arra, hogy mit akarunk benne tárolni, ezért lett az első programunk változójának neve *nev*. Amennyiben több szóból álló kifejezést használunk névnek, akkor két megoldás terjedt el: Az egyik, amikor a második szótól kezdve minden szó első betűje nagybetűsként szerepel. Ezt a megoldást főleg a C nyelvcsalád nyelveiben használják. A másik lehetőség, hogy aláhúzásjelet írunk a szóközök (és kötőjelek) helyére. Ez utóbbi megoldást ajánlott a Python nyelvű programokban használni. Fontos, hogy egy programon belül a két megoldást lehetőleg ne keverjük egymással!

A Python nyelvben speciális célra használtak azok a függvények és változók, amelyek neve egy aláhúzásjellel kezdődik, így ilyen neveket ne használjunk!

Mint a programunk utolsó, 5. sora is mutatja, a változóban tárolt értéket egyszerűen a változó azonosítójának megadásával lehet egy kifejezésben felhasználni.

A változókkal ellentétben, amelyek értéke változhat (innen a nevük), egyes értékeket közvetlenül is szerepeltethetünk a programban, ezeket nevezzük *konstans*nak. Szintén konstansnak nevezzük a változókhoz hasonlóan névvel ellátott, de nem változtatható értékű programobjektumokat. Ezeket a C nyelvcsaládban általában csupa nagybetűvel írják. Pythonban hivatalosan nincs lehetőség konstans megadására, ezért ha szükséges, akkor helyettük is változót használunk.

Egy változónak új értéket adhatunk valamelyik értékadó utasítással. A legegyszerűbb értékadó utasítást az `=` műveleti jellel adhatjuk meg. Ez nem azonos a matematikai egyenlőségjellel, bár ugyanaz a karakter!

Az értékadó utasítás alakja:
<változó> = <kifejezés>
A baloldalon tehát mindenképpen egy változó szerepel, amelynek a jobboldalon álló kifejezés értéke lesz az új értéke. A régi tartalom egész egyszerűen elveszik.

Vannak azonban további értékadást (is) végző műveleti jelek, amelyek valamely más műveletet a baloldalon álló változó értékét **is** felhasználva hajtanak végre, azaz a kifejezés részének tekintik a változó értékét is, majd az így kapott kifejezés értékét tárolják el a változóban. Például az alábbi utasítás

```
gyumolcsok += korte
```

jelenése a következő: a *gyumolcsok* változó pillanatnyi értékét és a *korte* változó pillanatnyi értékét össze kell adni és az összeadás eredményét el kell tárolni a *gyumolcsok* változóban – eldobva annak jelenlegi értékét. Azaz a fenti utasítás megegyezik az alábbi utasítással:

```
gyumolcsok = gyumolcsok + korte
```

Hasonló rövidítő értékadások érhetőek el a következő műveletekre: `+`, `-`, `*`, `/`, `%`, `//`, `**`. Minden esetben a művelet mögé kell írni az egyenlőségjelet a művelet utáni értékadáshoz.,

A számértékekhez hasonlóan szöveges adatokat is lehet használni a programban. A szövegkonstansokat többféle módon is meg lehet adni. Az egyik módszer: idézőjelek közé írjuk a szöveget. Az idézőjel lehet szimpla (`'`) vagy dupla (`"`), de amelyiket a szövegkonstans elején alkalmazzuk, azt kell a végén is alkalmazni, a szöveg belsejében ezért az nem használható. A szimpla idézőjelet szokás *aposztróf*nak, a dupla idézőjelet pedig egyszerűen *idézőjel*nek (*macskakörömnek*) is nevezni.

A magyar billentyűzeten az 1, illetve a 2 billentyűt kell a SHIFT billentyűvel együtt lenyomni a begépelésükhöz.

A fentieknek megfelelnek például az alábbi szövegkonstansok:

```
'alma'
"körte"
'hogy "jó" legyen'
"ez 'ciki' volt"
```

Mint az utolsó két példán látható, a másik idézőjel gond nélkül használható az idézőjeles szövegben. Amennyiben a határolójelként használt karaktert kellene szerepeltetni a szövegben, akkor az egyik lehetőség éppen a másik idézőjel használata a szövegkonstans határolójeleként. A másik lehetőség a fordított perjel használata: `\'` az apozstrófot, `\"` az idézőjelet jelenti a szövegben. Például: `"hogy \"jó\" legyen"`, vagy `'ez \'ciki\' volt'`.

Hibásak viszont azok a szövegkonstansok, ahol az elején és a végén levő határolójel nem egyezik meg.

```
'ez hibás"
"ahogy ez is'
```

Van egy másik lehetőség is a szövegkonstansok megadására, amely egyben megjegyzések írására is használatos bizonyos helyeken. Ez a módszer több soros szöveg megadására is alkalmas, ahol a szövegkonstans a sortöréseket is tartalmazni fogja. Ilyen szövegkonstans formájában szokás a program dokumentációjába szánt megjegyzéseket írni, ezért a példaprogramokban bőven fogsz ilyeneket látni, sőt neked is kell majd ilyeneket készítened a programjaidba.

Ebben az esetben is lehet mind az egyszeres, mint a dupla idézőjeleket használni teljesen azonos jelentéssel, de egy szövegkonstans esetében szigorúan azonosakat. Mind a szöveg elején, mind a végén hármat kell belőle alkalmazni:

```
"""Ez egy több  soros
szövegkonstans, amelyben mindkét idézőjelet (' és ")
szabadon lehet használni, mivel a szöveget hármas
idézőjel határolja."""
'''Ez szintén egy többsoros szövegkonstans.
Ezúttal aposztrófokból használunk fel hármat-hármat
az elején és a végén...'''
```

Ilyen többsoros szövegkonstansként megadott szövegekből képes egy erre készült program automatiksan legyártani a programunk dokumentációját, amelyet nagyos sok program fejlesztői ki is használnak. Ezért programok vagy programkönyvtárak dokumentációi a neten elég hasonló formátumban jelennek meg, köszönhetően annak, hogy ez az automatikus dokumentációkészítés HTML oldalak formájában is elérhető…

5.2.1. A del utasítás

Van egy módszer, amellyel egy korábban létrehozott változót véglegesen el lehet távolítani az értékével együtt a memóriából: a **del** utasítás. Ez az utasítás a változót azonnal megszünteti, ezt követően a változóra való hivatkozás hibát eredményez. A később megismerendő összetett adattípusok esetén alkalmazható a változó az összetett adat egy részének törlésére is.

```
>>> a = 3
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

5.3. Programhibák fajtái

Amikor valaki programot ír, elkerülhetetlenül hibák is keletkeznek. Ezek a hibák két kategóriába sorolhatóak: szintaktikai és szemantikai hibák.

- **Szintaktikai hiba**: Amikor például gépelés közben elgépelünk valamit, vagy más módon megsértjük az adott programozási nyelv szabályait, akkor nyelvtani, azaz *szintaktikai hibát* vétünk. Ilyen esetben általában már a program futása sem kezdődik el, mert az értelmező észleli a hibát. Ez könnyebben észlelhető, mivel általában a fejlesztő környezet is jelzi a hibát (például a PyCharm vagy a VS Code pirossal aláhúzza azt a szövegrészt, amelynél a hibát észleli). De néha a fejlesztő környezet vagy az értelmező nem ott jelzi a hibát, ahol az ténylegesen található, hanem valamivel később, ahol a hiba miatt ténylegesen a nyelvtani szabályokkal ellentétes kód jelentkezik.
- **Szemantikai hiba**: a program ugyan nyelvtanilag helyes, de mégis hibásan működik, mert a megírt programkód nem azt jelenti, amit az írója akart írni. Az ilyen hibák általában csak teszteléssel szűrhetőek ki.

Mind a szintaktikai, mind a szemantikai hiba okozhat futás közben észlelt hibát. A legtöbb futás közbeni hiba is a szintaktikai hibákból ered, de van olyan eset is, amikor egy szemantikai hiba okoz valahol futás közbeni hibát. Azonban van olyan szemantikai hiba is bőven, amely nem fog futási hibát okozni, csupán arról lehet felismerni, hogy a program nem az elvárt módon viselkedik.

Futási hibát szándékosan is lehet generálni, mint ahogyan el is lehet azt kerülni, hogy ilyen esetben a program végrehajtása félbeszakadjon. Hogy ez hogyan lehetséges, azzal egyelőre nem foglalkozunk.

Viszont fontos tudni, hogy maguk a hibaüzenetek is hasznos információval szolgálhatnak arról, hogy hol és milyen hiba történt: általában a hibaüzenet megmondja, hogy a program melyik sorában levő utasítás végrehajtásakor keletkezett a hiba, valamint a hiba jellegéről is szolgáltat információt.

5.4. Feladatok

5.1. Mit írnak ki az alábbi programok? Gondold végig, azután próbáld ki!

```
1  állat = 'ló'
2  ló = "Ráró"
3  állat = 'macska'
4  print(állat)
5
```

```
1  gyümölcs = 'alma'
2  gyümölcs = 'körte'
3  alma = 'dinnye'
4  dinnye = 3
5  gyümölcs = alma
6  print(gyümölcs)
7
```

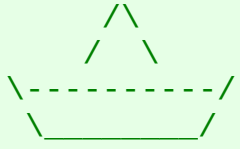
```
1  autó = 'Trabant'
2  autó = 'Renault'
3  autó = 3 - 5
4  print(autó)
5
```

5.2. Írj programot, amely kirajzolja karakterekből az alábbi ábrát:

A következő ábrákhoz tudnod kell, hogy a fordított perjelet alapból nem írja ki a gép, csak ha megkettőzzük: `print('\\')` hatására egy `\` jel íródik ki (az első várja a kiírandó karakter megadását, ami bármilyen egyébként nem nyomtatható karakter lehetne.

5.3. Írj programot, amely a kirajzolja karakterekből az alábbi ábrát:

5.4. Írj programot, amely a kirajzolja karakterekből az alábbi ábrát:



5.5. Írj programot, amely megkérdezi a felhasználótól a vezetéknevét, majd a keresztnevét és a teljes nevén szólítva köszönjön neki. A program kimenete a következő legyen (a dőlt betűs szöveg példa a felhasználó által adott válaszra, amelytől természetesen függ a kimenet):

```
Mi az Ön becses vezetékneve? Alma
Érdeklődhetnék a keresztneve felől is? Oszkár
Üdvözlöm, Alma Oszkár!
```

6. Egyszerű adattípusok

6.1. Fogalmazd meg a saját szavaiddal, hogy mit csinál az alábbi program!

kiserletezes.py

```
1  #! /usr/bin/python3
2
3  szuletes_eve = input("Melyik évben születtél? ")
4  eletkor = input("Hány éves vagy most?")
5
6  aktualis_ev = szuletes_eve + eletkor
7
8  print("Tehát most " + aktualis_ev + " van...")
9
```

Ezután próbáld ki, hogy tényleg azt teszi-e a program, amit gondolsz! Mit tapasztalsz a program futásakor?

A fenti feladatban szereplő program szemantikai hibát tartalmaz. Ahelyett, hogy elárulnánk a hiba okát, ebben a leckében a Python nyelv alapvető adattípusaival ismerkedhetsz meg. A lecke végigolvasása után képesnek kell majd lenned átírni a programot, hogy az helyesen működjön.

A legtöbb fejlesztő környezet – ahogyan a VS Code is – alaphelyzetben sötét színsémát használ. Ennek az az oka, hogy a világos képernyő sokkal rosszabb hosszú távon a szemnek, így a munkájukat egész nap a képernyő előtt töltő programozók a szemük kímélése érdekében előnyben részesítik a sötét színösszeállításokat. Valószínűleg ez az oka annak is, hogy például a Gimp 2.8-as verziójáig világos színek helyett a 2.10-es verziótól alapértelmezésben sötét színeket használ, ahogy sok más olyan program is, amelyet használója feltehetően naponta sok órán keresztül néz munka – vagy szórakozás – közben.

Bár ezek a sötét színprofilok a képernyőn sokkal ergonómikusabbak, kivetítőn, vagy tableten illetve telefonon az ilyen programról készült ábrák gyakorlatilag olvashatatlanok lennének. Ezért ebben a dokumentumban világos színprofilt használó programokból származnak a képernyőképek – így a fenti feladatban szereplő is.

A VS Code esetében a színprofilt a *File* / *Beállítások* / *Color Theme* menüpont kiválasztásával lehet állítani.

Minden programozási nyelvben, így a Pythonban is különböző típusú adatokat használhatunk. Az adatok típusa meghatározza az adott értékek tárolásához szükséges memóriaterület méretét, a lehetséges értékek halmazát, valamint az adatokon végezhető műveleteket is.

Bár a Python nyelvben egy változó létrehozásával nem kell különösebben törődni, így a típusát sem kell előre megadni, a Python mégis szigorúan veszi az adattípusok használatát. Ezen felül – más programozási nyelvekhez hasonlóan – itt is van lehetőség a változó típusát megadni, csak ezt ritkán alkalmazzuk, mert nem szükséges. Ám megadása esetén az értelmező segít ellenőrizni, hogy tényleg a megfelelő típusú értéket akarjuk-e beletenni.

A következőkben a Python típusai közül az alaptípusokat, valamint egy összetett típust mutatunk be – utóbbit részleteiben majd csak egy későbbi leckében. Ezek a típusok a legegyszerűbb program esetében is szerepelnek valamilyen formában.

Először két (valójában három) számok tárolására alkalmas típus, majd a logikai értékek tárolására alkalmas típus fog szerepelni, végül a szövegek tárolására alkalmas összetett típus legfontosabb tulajdonságai szerepelnek majd.

Bármely változó vagy kifejezés típusát gyorsan le lehet kérdezni a **type()** függvény segítségével: a paraméterbe írt kifejezés típusát adja a függvény eredményül. Ez a függvény akkor hasznos, ha nem tudjuk eldönteni egy adott kifejezésről, hogy az vajon milyen értékű (a nem megfelelő típus könnyen okozhat nagyon nehezen felfedezhető szemantikai problémát a programokban).

6.1. Számtípusok

A Python nyelv három számtípust ismer: *egész számokat*, *lebegőpontos számokat* és *komplex számokat*.

Egész értékek tárolására az `int` típus szolgál. Amennyiben egy változónak értékül egy olyan konstanst adunk, amely nem tartalmaz tört részt, akkor ilyen típusú lesz a változó. Az egész konstans egy opcionális előjelet és utána a számérték számjegyeit tartalmazhatja tízes számrendszerbeli számként.

Amennyiben ez után egy tizedespont (nem tizedesvessző!) és utána újabb számjegyek következnek, akkor azonban már lebegőpontos érték lesz belőle, amelynek tárolására a `float` típus használatos.

Egyetemen kerülnek majd csak elő a komplex számok, amelyek tárolására a Python a `complex` típust használja, amellyel most nem fogunk foglalkozni. Ezen felül még további típusok is léteznek (*fractions.Fraction* és *decimal.Decimal* néven), amelyekkel szintén nem fogunk foglalkozni.

Mivel egy kifejezésben a fenti típusú értékek szabadon keverhetőek, így a Python mindig a „legszűkebb” típust fogja a kifejezés típusának tekinteni, amely típusban a kifejezés értéke tárolható. A legszűkebb itt azt jelenti, hogy

- ha a kifejezés csak egész típusú értéket tartalmaz, akkor a kifejezés típusa: *int*
- ha a kifejezés csak egész és lebegőpontos értéket tartalmaz, akkor a kifejezés típusa: *float*
- ha a kifejezés komplex számot is tartalmaz, akkor a kifejezés típusa: *complex*.

A számértékeken a következő műveletek alkalmazhatóak:

`x + y` – *x* és *y* összege

`x - y` – *x*-ből *y* értéke kivonva

`x * y` – *x* és *y* szorzata

`x / y` – *x* osztva *y*-nal

`x // y` – *x* osztva *y*-nal, de csak az osztás egész részét, mint *int* típusú értéket értve (azaz **egész osztás**)[2]

`x % y` – *x* // *y* maradéka

`-x` – *x* ellentettje (-1 szerese)

`abs(x)` *x* abszolút értéke

`divmod(x, y)` az eredmény egy két elemű tuple[3], amelynek első eleme `x//y`, második eleme `x%y` eredménye.

2 Valójában a művelet csak akkor ad *int* típusú értéket, ha *x* és *y* is *int* volt.

3 A tuple típusról majd a szekvenciáknál lesz szó.

`pow(x,y)` a függvény x^y értékét adja

`x**y` szintén a x^y értékét adja.

Az utolsó két művelethez tudni kell, hogy a Python a 0^0 eredményét 1-nek definiálja – ami eltér a matematikai definíciótól, ahol ennek a műveletnek nincs értelmezve az eredménye.

További függvények is alkalmazhatóak számértékekre, amelyek a Függvénytárban megtalálhatóak.

Bármely értéket egy adott típusúvá lehet konvertálni a céltípus nevét viselő függvénnyel:

- egész értékké konvertáláshoz az **int()** függvény kell;
- lebegőpontos értékké konvertáláshoz a **float()** függvény kell.

Lebegőpontos érték egésszé konvertálása esetén csak az érték egész része marad meg.

6.1.1. Egész számok

Egész számok tárolására használjuk az **int** típust. Mint már volt róla szó, alaphelyzetben a Pythonban ugyan nem kell megadni a típus nevét, ennek ellenére jó tudni, hogy mikor milyen adatot használunk – nem csak az esetleges hibaüzenetek megfejtésére. Ugyanis a használható értékek és a használható műveletek egyaránt függenek a használt típustól.

Először nézzünk néhány példát jó és hibás egész értékekre! Az alábbiakban először helyes egész konstansokra látsz példát:

```
1
-15
61
42
-42
```

Mint láthatod, az egész számok számjegyek sorozatából állnak, amelyeket megelőzhet egy negatív (esetleg egy pozitív) előjel. Ha bármely mást írunk még ehhez, az már nem egész szám lesz:

```
6.0   # 6 értékű lebegőpontos szám
0.1   # ez is lebegőpontos szám
0,6   # ez két egész érték felsorolása
egy   # szöveg
1b    # szöveg
```

Ha valahol a programban elő van írva az *int* típus használata és nem egész típust használunk, akkor hibaüzenetet fogunk kapni.[4] Ahol viszont nincs előírva egy változónak történő értékadás típusa, ott a változó egész típusú lesz, ha egész típusú értéket adunk neki.

Az egész számokon végzett műveletek eredménye általában egész szám lesz. Ez alól a kivétel az osztás, amely még akkor is lebegőpontos eredményt ad, ha egyébként az eredmény értéke egész lenne:

```
>>> 6 / 2
3.0
```

4 Ez a mondat arra az esetre vonatkozik, ha valamely művelet vár egész értéket. A programban megjelölhetjük a változóinkat, hogy milyen típusú értéket akarunk neki adni, de ezzel az értelmező nem törődik, csak a fejlesztő környezet jelezheti, hogy itt szemantikai hibát készülünk elkövetni.

A fenti példa azt mutatja, hogy bár a `6/2` művelet eredménye 3, mégsem `3` lesz az érték, hanem a lebegőpontos `3.0` érték a művelet eredménye.

Bármely értéket, amely értelmezhető egész számértékként, át lehet alakítani egész típusú adattá. Ehhez nem kell más, mint a típus nevét függvényként használni:

```
>>> int(3.0)
3
>>> int("12")
12
```

De a Python nem képes szövegesen leírt számot érelmezni:

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

Ez alapján lehet például a mindig szöveget eredményül adó **input()** függvénnyel beolvasott szövegben levő számot valódi számmá alakítani:

```
a = int(input("a = "))
```

Amennyiben a felhasználó nem egy egész számot gépel be, akkor az utasítás a fentihez hasonló hibaüzenetet fog eredményezni.

Bitműveletek

Ugyan nem fogunk ilyenekkel foglalkozni, de célszerű megemlíteni, hogy az *int* típusú értékek bitsorozatként kezelve további műveleteket ismernek. Ezek a műveletek úgy tekintik az operandusaikat, mint bitek sorozatát, amelyeknél minden azonos helyiértékű bitpárra ugyanazt a logikai műveletet kell elvégezni, majd az így keletkező eredménybitek sorozata lesz a művelet eredménye. Ilyenkor tehát az értékekre nem egész számokként, hanem logikai értékeket reprezenteló bitek sorozataként tekintünk, ahol a bit 1 értéke igaz, 0 értéke hamis logikai értéket reprezentál.

`x | y` a *vagy* (OR) műveletet valósítja meg *x* és *y* azonos helyiértékű bitjei között.

`x ^ y` a *kizáró vagy* (XOR) műveletet valósítja meg *x* és *y* azonos helyiértékű bitjei között.

`x & y` az *és* (AND) műveletet valósítja meg *x* és *y* azonos helyiértékű bitjei között.

`~x` a *bitenkénti tagadás* (NOT) műveletet valósítja meg.

`x >> n` eltolja *x* bitjeit jobbra *n* bittel (*n* egész szám). Ez nagyjából az x `//` 2^n műveletet jelenti.

x << n eltolja x bitjeit balra n bittel. Ez megfelel az $x * 2^n$ műveletnek.

A két bitenkénti eltolás tehát kettő hatványaival szorzásnak illetve osztásnak felel meg, azonban a legtöbb processzor sokkal gyorsabban tudja a biteltolás műveletét megvalósítani, mint akár a szorzást vagy az annál is időigényesebb osztást.

6.1.2. Lebegőpontos számok

A valós számoknak egy *valódi részhalmazát* lehet tárolni a **float** típusban, amely valójában lebegőpontos számok tárolására alkalmas. (A lebegőpontos szám magyarázatát a következő lecke tartalmazza a 60. oldaltól kezdve.)

Minden olyan – középiskolában ismert – szám, amely nem egész szám, a *float* típusban tárolható (ha tárolható). Ahogyan az egész számok ábrázolásának is vannak határai, természetesen a lebegőpontos számoknak is lesznek ilyen határai.

Amennyiben egy számkonstans olyan elemeket is tartalmaz, amelyek már nem férnek bele az *int* típusnál elfogadottakba, akkor a számot a Python automatikusan *float* típusúnak tekinti. Ezek a számkonstansok úgy kezdődnek, mint az egész számok, azonban egy tizedesponttal vagy az *exponenciális részt* bevezető `e` (vagy `E`) betűvel biztosan folytatódnak. A tizedes pont, illetve az `E` betű után kell még egy egész számnak következnie. A tizedes pont utáni rész a szám tört része tizedes törtként.

Az alábbiak helyes lebegőpontos konstansok:

```
1.0     # értéke 1, de float típusú
0.1
1.33e-2 # 1.33 * 10-2 normál alakban
1e2     # normál alakban írva a 100
.5      # ez is helyes: 0.5-nek felel meg
```

Mint az utolsó példából látható, az egész rész el is hagyható, ha az értéke nulla és a törtrész szerepel. Ekkor a konstans rögtön a tizedesponttal kezdődik, amit a tizedesrészt megadó számjegyek követnek. A példák között negatív érték nem szerepel, de értelemszerűen a számot egy negatív (vagy pozitív) előjel is megelőzheti opcionálisan.

A fenti példák közül a harmadik és a negyedik példában szerepel az *exponenciális rész* is. Ez ha szerepel, akkor mindenképpen a törtrész után szerepel. Ezt úgy kell érteni, hogy az előtte levő számot meg kell szorozni a 10 azon hatványával, amelynek kitevője szerepel az `e` vagy `E` betű után. Ez gyakorlatilag a matematikából ismert *normál alak* megadását teszi lehetővé azzal a különbséggel, hogy a kitevőt nem lehet a programkódban felső indexbe tenni, ezért ezt a jelölést találták ki rá.

A harmadik példa az $1{,}33*10^{-2}$ normál alakú számot, a negyedik példa az $1*10^{2}$ alakú számot (100) jelöli. Ezt a jelölési módot egyébként a világ **tudományos jelölés** néven ismeri.

Következzen néhány példa a hibás lebegőpontos számkonstansokra is:

```
0,1     # ez két egész szám!
e-2     # kell az egész vagy a törtrész legalább
1e2.5   # előbb a törtrész, aztán az exponenciális!
0.2e1.4 # az exponenciális rész csak egész érték lehet!
```

Az utolsó előtti példa esetén nem egyértelmű, hogy az exponenciális részben szerepel törtszám vagy a törtrészt előzi meg hibásan az exponenciális rész. Ám ez nem is lényeges, mivel az exponenciális részben a 10 kitevője csak egész értékű lehet. Az első példa valójában nem hibás kifejezés, csak nem egy valós számot ad meg. Ahhoz ugyanis tizedes pontot kell használni, nem tizedes vesszőt! Az első példa két egész szám felsorolását adja, amely legális érték lehet, de nem ott, ahol egy lebegőpontos számnak kellene szerepelnie.

A második példa egy betűvel kezdődik. Hiába néz ki úgy, mint az exponenciális rész, ez az *e* változóból kettőt kivonó kifejezés lenne.

Konvertálás float típusra

Ahogyan az egész típusra lehetett a típus nevét viselő függvénnyel konvertálni, úgy a *float* típushoz is tartozik egy konvertáló függvény. Ezzel az egész vagy a komplex számokat lehet lebegőpontos számokká konvertálni. (Például a C nyelvben az a szokás egész számok lebegőpontossá alakítására,

hogy megszorozzák 1.0-el a konvertálandó számot, így a kifejezés típusa lebegőpontossá válik. Ez a módszer természetesen Python-ban is működik.) Hasonló módon a szabályos lebegőpontos számkonstanst tartalmazó szöveg is átkonvertálható lebegőpontos számmá a **float()** függvénnyel.

6.2. Logikai típus

A logikai értékek tárolására használható a **bool** típus. Hivatalosan ez is egésznek van definiálva, de a típusnak összesen két lehetséges értéke van: `False` és `True`. A `False` érték jelentése: *hamis*, a `True` érték jelentése: *igaz*. Számokat tartalmazó kifejezésben az igaz 1, a hamis 0 értékkel lesz figyelembe véve. Ebből (is) következően a két érték között a következő reláció áll fent: `False` < `True`.

A **bool()** függvény a már megismert módon bármely értéket képes logikai értékké konvertálni:

1. Számértékek esetén a nulla érték *hamis*, minden más érték *igaz* érték lesz.
2. Szöveg esetén az üres szöveg *hamis*, minden más szöveg *igaz* érték lesz.
3. Lista esetén az üres lista *hamis*, minden más lista *igaz* érték lesz.
4. Halmaz és szótár esetén az üres halmaz illetve szótár *hamis*, minden más halmaz illetve szótár *igaz* érték lesz.
5. Egy üres változó értéke *hamis* értékké konvertálódik.

Az összehasonlító műveletek logikai értéket eredményeznek. Az összehasonlítandó értékeknek egymással az összehasonlítás szempontjából kompatibilisnek kell lenniük (tehát számot szöveggel nem lehet összehasonlítani, de egész számot lebegőpontossal igen).

`a < b` igaz, ha *a* kisebb értékű *b*-nél, különben hamis.

`a > b` igaz, ha *a* nagyobb értékű *b*-nél, különben hamis.

`a == b` igaz, ha *a* és *b* értéke megegyezik, különben hamis.

`a != b` igaz, ha *a* és *b* értéke nem egyenlő, különben hamis.

`a <= b` igaz, ha `a < b` vagy `a == b` igaz, különben hamis.

`a >= b` igaz, ha `a > b` vagy `a == b` igaz, különben hamis.

`a is b` igaz, ha *a* azonos *b*-vel (tehát a memóriában a két objektum azonos területet használ és azonos típusúként kezeli az ott levő értéket).

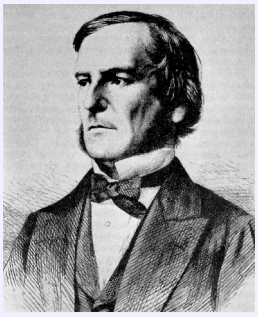
`a is not b` igaz, ha `a is b` hamis.

A fenti relációs műveletek egymással azonos prioritásúak és balról jobbra haladva kell őket végrehajtani. Ez azt jelenti, hogy a relációk egymással összeláncolhatóak: `a < b < c` logikai művelet eredménye megegyezik az `a < b and b < c` logikai kifejezés eredményével – azzal a különbséggel, hogy az előbbi esetben b értéke csak egyszer lesz kiszámolva. Mindkét példában a *c* értéke csak akkor lesz meghatározva, ha `a < b` igaz, hiszen ellenkező esetben a végső logikai értéket *c* értéke már nem tudja befolyásolni.

Az egyenlőség vizsgálatának lebegőpontos számok esetében nincs értelme.

Két logikai érték között az alábbi logikai műveletek végezhetőek el:

- `a or b` igaz, ha *a* és *b* közül legalább az egyik igaz, különben hamis.
- `a and b` igaz, ha mind *a*, mind *b* igaz, minden más esetben hamis.
- `not a` tagadja *a* értékét, azaz igaz, ha *a* hamis és hamis, ha *a* igaz.



GEORGE BOOLE angol matematikus (1815–1864) alkotta meg azt a logikai szabályrendszert, amelyen a modern logika is alapul. Ezt nevezik *Boole-algebrá*nak. Mivel a számítógépeken használt logikai típus is ezt a Boole-algebrát követi, így a logikai típus neve általában utal Boole-ra. A Pascal nyelvben például *Boolean* a logikai típus neve. C-ben nincs külön logikai típus, ott az egész típust szokás helyette használni. Több C alapú nyelvben, mint a Java vagy a Pythonban is rövidebben, *bool* néven nevezik a boole-i elveken működő típust.

Az alábbiakban a három logikai művelet igazságtábláját láthatod:

a	b	a and b
hamis	hamis	hamis
hamis	igaz	hamis
igaz	hamis	hamis
igaz	igaz	igaz

a	b	a or b
hamis	hamis	hamis
hamis	igaz	igaz
igaz	hamis	igaz
igaz	igaz	igaz

a	not a
hamis	igaz
igaz	hamis

6.3. Karakterláncok

Bár a karakterlánc már nem számít egyszerű adattípusnak, mégis célszerű említést tenni róla, mivel szinte minden programban előfordul valamilyen nagyon egyszerű formában – ha másképpen nem is, mint szövegkonstans.

A típusról részletesebben majd egy későbbi leckében lesz szó, most csupán a legfontosabb információk következnek, amelyek nélkül nem lehet boldogulni.

A karakterlánc, angolul *string* típusú adatok tárolhatóak az **str** típusban.

Mind a magyar, mind az angol név arra utal, hogy a szöveget alkotó sztring a memória egy folyamatos helyén tárolódik úgy, hogy minden egymást követő byte (esetleg byte-ok) egy-egy karaktert tárolnak el a szövegből.

Szövegkonstanst megadhatunk szimpla (') vagy dupla (") idézőjelek közé zárva. A határolójeleknek az elején és a végén meg kell egyezniük. A két karakter közül azt, amelyiket határolójelként használunk, a karakterláncon belül közvetlenül nem használhatjuk, hiszen az lezárná a szövegkonstanst. Ha mégis használni akarjuk, akkor a \ karakter után írva adhatjuk meg (\' vagy \" formában), míg a másikat gond nélkül leírhatjuk. Az ilyen szövegkonstansnak egy sorban kell elférnie.

Többsoros szövegkonstans is megadható, ha a határolójeleket megháromszorozzuk (""" vagy '''). Ebben az esetben is igaz, hogy a határolójeleknek meg kell egyezniük. Az ilyen több soros szövegkonstanson belül minden sor végén egy újsor jel kerül a szövegbe. Ezt a formát szokás a Python nyelvben olyan megjegyzések írására is használni, amelyet dokumentációhoz akarunk használni.

Az *str* típusú adat *nem módosítható*. Ez azt jelenti, hogy minden olyan művelet, amely látszólag módosítja azt, valójában a szöveg új példányát hozza létre a memóriában és a változó, amely értékül kapja, azt az új példányt fogja tartalmazni (ennek memóriacímére mutat a változó), a régi példány pedig ezután törlődik a memóriából, ha más változó nem mutat rá. Ez a szabály majd későbbi programoknál fog jelentőséget kapni, egyelőre nincs számunkra jelentősége…

Az *str* típusú adaton a következő műveletek végezhetőek el:

`a + b`: összefűzés, eredénye egy olyan (új) szöveg, amely elején az *a* szöveg karakterei, majd ezt közvetlenül követve a *b* szöveg karakterei vannak.

`a * n` vagy `n * a`: az *a* szöveg összefűzése önmagával *n* alkalommal. Az *a str* típusú, *n int* típsú adat.

Az **str()** függvénnyel bármely típusú érték szöveggé alakítható. Hogy mi lesz az adott típusú érték esetén a szöveg, az a típustól függ. Számok esetén általában az adott számérték számkonstansként megadható alakja, logikai értékként a megfelelő logikai konstans nevét tartalmazó karakterlánc (`'True'` vagy `'False'`), üres értéknél az üres karakterlánc (`''`) a speciális *None* értéknél a `'None'` karakterlánc.

6.4. Feladatok

6.2. Javítsd ki a lecke elején, a 6.1. feladatban szereplő programot, hogy helyesen működjön!

6.3. Írj programot amely bekéri egy háromszög egyik oldalának és a hozzá tartozó magasságnak a hosszát (valós számként), majd kiírja a háromszög területét!

6.4. Írj programot, amely bekéri a háromszög három oldalának hosszát, majd ezek segítségével kiszámolja a háromszög kerületét és területét!

A háromszög területe a három oldal ismeretében a Héron-képlettel számolható ki:

$$T=\sqrt{s(s-a)(s-b)(s-c)}$$

$$s=\frac{a+b+c}{2}$$

A négyzetgyökvonáshoz szükséges a **math.sqrt()** függvény, amelyhez a program elején a *math* modult be kell importálni az alábbi utasítással:

```
import math
```

6.5. Készítsünk programot, amely bekér egy kilométerben megadott távolságot (valós számként) és kiírja ugyanezt a távolságot tengeri mérföldbe átváltva!

Az átváltáshoz használjuk fel, hogy egy tengeri mérföld 1852 méter.

6.6. Kérjünk be a felhasználótól két számot, tároljuk őket egy-egy változóban!

a) Adjuk össze őket és írjuk ki az eredményt!
b) Írjuk ki az eredmény elé, hogy „Az összegük:”!
c) Írjuk ki magukat a számokat is! Ha például 2-t és 3-t adott meg a felhasználó, akkor a kimenet legyen ilyen:

```
2 és 3 összege: 5.
```

6.7. Készítsük el a 6.6. feladat szorzásra vonatkozó változatát is!

6.8. Kérjünk be a felhasználótól három egész számot, mint egy időpont *óra*, *perc*, *másodperc* adatait! Ezután írjuk ki ezt az időpontot úgy, mint az éjfél óta eltelt összes másodperc mennyiségét.

Ezután fordítsuk meg a dolgot: kérjük be a felhasználótól, hogy eddig hány másodperc telt el éjfél óta, majd írjuk ki *óra*, *perc*, *másodperc* formában az ennek megfelelő időpontot!

6.9. Készíts programot, amely a derékszögű háromszög befogóinak értékét lebegőpontos számként bekéri, majd a Pithagoras-tételt felhasználva kiszámolja és kiírja az átfogó hosszát!

A feladat megoldásához a program elejére itt is szükséges elhelyezni az alábbi sort:

```
import math
```

A kiírást segíti a karakterláncok **.format()** művelete vagy az ehhez hasonlóan használható formázott szöveg használata. Például az alábbi két egyenértékű utasítás a *befogo* változó értékét legalább 4 karakter szélességben, kettő tizedesjegy pontosságban írja ki:

```
print("A befogó: {0:4.2f}".format(befogo))
print(f"A befogó: {befogo:4.2f}")
```

A fenti példában a kapcsos zárójeles rész jelentése: legalább négy jegyen (a tizedespontot is beleértve), két tizedes jegyre kerekítve, lebegőpontos számként kell a kifejezés értékét megjeleníteni. Természetesen ha a négy jegy nem elég, akkor több jegyet fog használni.

6.10. Egy autóúton a távolságokat annak kezdőpontjától mérik km-ben. Amennyiben első rendő főútvonalról vagy autópályáról van szó, akkor – mivel ezek egyik vége Budapesten van – a kezdőpontnak a budapesti Clark Ádám téren levő *nulla kilométer-kő*t tekintik, más út esetében az útnak azt a végét, amelynél az egy magasabb rendű útból kiágazik (ha mindkét vége egy magasabb rendű útnál van, akkor a kisebb számú számít a kezdőpontnak).

Készíts programot, amely egy adott út mentén fekvő két település nevét és a településeknek az út kezdőpontjától mért távolságát kéri be!

Ezután a program írja ki, hogy a két település között mekkora a távolság!

Ez alá a program írja ki azt is, hogy hány perc alatt lehet az egyik településről a másikra jutni a következő átlagsebességek használata mellett: 90 km/h, 70 km/h, 60 km/h, 50 km/h és 30 km/h!

A feladat megoldásához jól fog jönni az **abs()** függvény, amely a paraméterében szereplő számérték abszolút értékét adja.

A kiírás beszédesen, egész mondatokban történjen, amelyben a két település neve is megtalálható!

6.5. Megjegyzés a típusokkal kapcsolatban

A középiskolai programozás témakörben nem szándékozunk belemenni az *objektum-orientált programozás* fogalmaiba. Ám a dokumentumban időről időre fognak szerepelni megjegyzésként célzások ezzel a területtel kapcsolatban. Ennek nagyon egyszerű oka van: a Python valójában egy objektum-orientált programozási nyelv, ahol minden nyelvi elem valójában egy-egy objektumként kezelendő. Ennek ellenére – ellentétben például a Java vagy a C# nyelvekkel – megismerhető a nyelv úgy, hogy az objektum-orientált megközelítésről említést sem teszünk. Erre bizonyíték, hogy egészen mostanáig említést sem találhattál az objektum-orientált programozásra.

Az egyetlen ok, ami miatt most – és időnként később is – szó van róla, hogy némely fogalomnak nem az a valódi neve, ahogy a dokumentum nevezi, így ha később megismerkedsz majd az objektum-orientált programozással, akkor ha ezekről nem teszünk említést, akkor úgy tűnhet, mintha becsaptak volna.

Nézzük a konkrét esetet! A Python nyelv típusai valójában objektum-osztályok. Ez azt jelenti, hogy minden egyes változóban vagy konstansban szereplő érték az adott osztály egy objektuma. Az egyes osztályok definiálnak a hozzájuk tartozó objektumok számára értékeket, valamint az objektumokon értelmezhető műveleteket. Az értékek az eddig megismert típusok esetében az *str* kivételével maguk a számértékek vagy logikai értékek. Az *str* esetében a memóriában a karakterláncot reprezentáló memóriatömb a karakterek kódjait tartalmazó byte-okkal.

A műveletek egy része az egyes típusok leírásánál bemutatott operandusokkal érhető el (+, –, stb.). Az `str` esetében a + és a * más műveletet jelent, mint a számoknál, éppen azért, mert az *str* osztály ezt a két műveletet összefűzésként definiálja. De az *str* típushoz még rengeteg más művelet is tartozik, amelyről a típus részletes bemutatásával foglalkozó lecke foglalkozik majd.

Az objektumok műveleteit, amelyek nem valamely operátorral van vannak megvalósítva, hanem függvényszerűen néznek ki, *metódus*nak hívjuk. Ezek helyett ebben a dokumentumban általában majd a függvény elnevezést használjuk az objektum-orientált szemlélet elkerülése érdekében.

Van azonban két speciális metódus, amelynek saját neve van: a *konstruktor*, amellyel az adott osztályba tartozó objektumot lehet *megkonstruálni*, azaz létrehozni; és a *destruktor*, amellyel meg lehet szüntetni az objektumot, ha már nincs szükség rá. Ezek általában az osztály (típus) nevével létező függvények, amelyek az adott típusú értéket adnak eredményül.

Valójában ennek a megjegyzésnek mindössze annyi az oka, hogy az egyes adattípusok bemutatásánál szerepelt, a típus nevével létező függvények, amelyekkel valamely értéket az adott típusra lehet konvertálni, valójában az adott típust megvalósító osztály konstruktorai.

7. A számítógép számábrázolása

Ebben a leckében a számadatok számítógépen belüli tárolási módjáról lesz szó. Ennek ismerete azért lehet hasznos, mert a számítógép nem minden olyan értéket képes tárolni, ami a valóságban előfordul. Emiatt adódhatnak bizonyos problémák a számítógépes adatfeldolgozás során, amelyek nem elkerülhetőek, ám a számok belső ábrázolási módjának ismeretében fel lehet ezekre a problémákra készülni és kezelni lehet azokat a helyzeteket, amelyeket okoznak.

A számítógép mindent binárisan, azaz kettes számrendszerbe kódolva tárol, így értelemszerűen a számokat is. Viszont nem akárhány jegyű számokat tud tárolni! A használt programozási nyelvtől és a használt számítógéptől függ, hogy egy egész szám tárolására mennyi byte áll rendelkezésre. (1 byte 8 bitet, azaz nyolc bináris számjegyet jelent.) Ebből következően nem minden egész számot tud a számítógép számábrázolási módja tárolni. Hasonló a helyzet értelemszerűen a lebegőpontos számokkal is.

7.1. Egész számok ábrázolása

Az egész számokat a matematikából megismert helyiértékes számábrázolás szabályai szerint kell ábrázolni – pozitív számok esetén legalábbis. Ebben az esetben a számrendszer alapszáma kettő, azaz számjegyként az 1 és a 0 fordulhat elő, minden helyiérték jobbról balra növekedve a kettő következő hatványa.

A bináris számokon három alapművelet végezhető el: összeadás, kettővel szorzás és -1-el szorzás.

Az összeadás a szokásos módon történik, ha a két számot egymás alá írjuk: az egymás alatti számjegyeket összeadjuk, figyelembe véve, hogy az adott helyiértéken a két szám összege nem fér el egy jegyen, akkor a következő helyiértékre történik egy átvitel (amelynek mindig egy lesz az értéke). Ebből a szempontból a kettes számrendszernek az az előnye a többi számrendszerhez képest, hogy itt a legkevesebb az összeadási szabály – ami a számítógépek megalkotásánál anyagi előnyt is jelentett. Utána gondolva könnyen kiszámolható, hogy mivel bármely számrendszer n számjegyet tartalmaz (vagyis az alapszáma n), akkot n^2 kombináció van átvitel nélkül is és átvitellel is, azaz összesen $2 \cdot n^2$ szabály van.

Tehát tízes számrendszerben összesen 200 szabály, míg kettes számrendszerben összesen 8 szabály van. Ez utóbbiak a következők:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1)0 (van még egy átvitel, és ezen a jegyen 0 az eredmény)

átvitellel:
1 + 0 + 0 = 1
1 + 0 + 1 = (1)0
1 + 1 + 0 = (1)0
1 + 1 + 1 = (1)1

A szorzás megvalósításához van szükség a kettővel szorzás műveletre: Minden helyiértékes számrendszerben az alapszám 10 alakban írandó fel. Ebből következően az alapszámmal szorzás nem jelent mást, mint hogy minden helyiérték értéke egyel magasabb helyiértékre vándorol, az egyes helyiértéken 0 értéket hagyva. Magyarul: a szám végére kell írni egy 0-t. (pl.: 42·10 = 420)

Ezt felhasználva két több jegyű szám összeszorzása kettes számrendszerben szintén elvégezhető úgy, ahogy a tízes számrendszerben, azaz az egyik számot rendre megszorozzuk a másik szám számjegyeivel, egyel eltolva a helyiértéket (ez a helyiértékkel eltolás az alapszámmal való szorzás), majd a részszorzatokat összeadjuk. Ez a művelet kettes számrendszerben szintén egyszerűsödik, hiszen csak 0 és 1 számjegyek lehetnek. A számítógép a részszorzatokat egyenként adja össze, azaz a fenti nyolc szabály elegendő az összeadás alkalmazásánál. Íme egy példa két kettes számrendszerbeli szám összeszorzására:

```
101101 * 101110
1011010
 000000
  101101
111000010
   101101
1111011110
    0000000
     101101
11111101001
```

A fenti példán a vízszintes vonalak fölött történik egy-egy összeadás.

A kivonást a számítógép összeadásként végzi el. Ehhez természetesen szükséges a negatív számokat olyan módon ábrázolni, hogy bármely *a* egész számra teljesüljön az $a + (-a) = 0$ egyenlet. Ezt valósítja meg a kettes komplemens alak, amely a -1-el való szorzást teszi lehetővé – kis csalással.

Egy szám kettes komplemensét, vagyis mínusz egyszeresét a következő eljárással lehet megkapni:

1. Vegyük a szám minden számjegyének a komplementerét: a 0 helyett 1; az 1 helyett 0 értéket.
2. Az 1. lépésben kapott bitsorozathoz, mint egész számhoz adjunk hozzá 1-et.

A 2. lépés eredményeként kapott szám az eredeti szám -1-szerese lesz.

A fenti módszer természetesen kihasználja, hogy a számok ábrázolása véges számú biten történik! Vegyük példaként a -1 értéket! Nyolc biten az 1 bináris alakja: 00000001. Ebből az első lépés végrehajtásával kapjuk: 11111110.

Ehhez egyet hozzáadva ezt kapjuk: 11111111. Ezt és az eredeti számot összeadva:

```
 00000001
+11111111
100000000
```

De mivel csak nyolc bitet használhatunk, a kilencedik bit 1-es értéke elveszik, azaz az eredmény 00000000, tehát nulla.

Ugyanezt a példát bármely egész számon elvégezhetjük, szintén megkapjuk a nullát azzal a kis trükkel, hogy a kilógó egyest elhagyjuk. Viszont emiatt a trükk miatt a számítógép nem teljesen a matematikának megfelelően számol!

Ha megnézünk néhány pozitív és negatív számot, akkor láthatjuk, hogy a legnagyobb helyiértékű számjegy pozitív számoknál mindig 0, negatív számoknál mindig 1. Ezért ezt a bitet **előjelbit**nek nevezzük. Ebből tudja a számítógép, hogy az adott számot negatív vagy pozitív számnak kell-e tekintenie.

Viszont itt kezdődik a gond is, hiszen az előjelbit a nulla esetén is 0, azaz a számítógép a nulla értéket pozitív számnak tekinti! Ráadásul így egy pozitív szám helyét foglalja el, azaz van egy negatív szám, amely ugyan ábrázolható, de az abszolút értéke nem. Sőt! Könnyen be lehet látni, hogy ez a számítógép számára egy olyan negatív szám, amelynek abszolút értéke önmaga. Ehhez azt kell tudni, hogy a szóban forgó szám mindig olyan alakú, ahol az előjelbit 1, a többi számjegy pedig nulla. Próbaként a nyolcbites esetet nézzük meg:

```
10000000
01111111 +1
10000000
```

Ez utóbbit pozitív számnak tekintve (de a számítógép számára ez negatív!) 128-at kapunk, azaz 128 nem, de -128 még ábrázolható kettes számrendszerben.

Ebből adódik egy probléma: amennyiben az ábrázolandó szám akkora, hogy a legnagyobb helyiértékű bitre is szükség van az ábrázoláshoz, akkor a számítógép más számnak fogja tekinteni. Erre jó példa a 100+50 művelet elvégzése 1 byte méretű, előjeles egész számábrázolás mellett:

```
100: 01100100
 50: 00110010
  +: 10010110
```

A fenti számítás a 100 és 50 értékét mutatja kettes számrendszerbe átszámolva, majd összeadva. Mint látható, az összegként előálló szám 8 bitet igényel, azaz az előjelbit 1 értéket kap, bár nyilván a 150 nem negatív. Ám a számítógép az 1 értékű előjelbit miatt ezt az eredményt negatív számnak tekinti.

Vegyük tehát az abszolút értékét!

```
10010110
01101001 +1
01101010
```

Ezt visszaváltva tízes számrendszerbe 106-ot kapunk, azaz az összeadás eredménye: -106!

Az ilyen hibákat, mint a fenti, **túlcsordulás**nak nevezzük. Akkor fordul elő, ha egy művelet eredménye nem fér el az ábrázolására rendelkezésre álló helyen, így a legmagasabb helyiérték elveszik, ezért hibás eredményt kapunk. A számítógépek egy része figyeli a túlcsordulást és hibajelzést ad, más része nem figyeli, hanem a hibás eredménnyel számol tovább! Ezért mindig figyelni kell arra, hogy a számolások során egy részszámítás eredménye se léphessen ki a számábrázolási tartományból.

A számábrázolási tartományt nagyon egyszerű meghatározni a fentiek alapján: nyolc bit esetén a 128 az első már nem ábrázolható szám, amely 2^7. A legkisebb ábrázolható szám pedig a -128, amely -2^7.

Ebből már látható, hogy egy *n* bites, előjeles egész szám ábrázolási tartománya -2^{n-1} és $2^{n-1}-1$ közé esik.

Hasonló módon, amennyiben előjel nélkül történik a számábrázolás, akkor egy *n* bites szám 0 és 2^n-1 közé esik (1 byte esetén 0 és 255 közé).

Az egyes programozási nyelvek, illetve számítógépek esetében a számábrázolási tartomány változó lehet. Van olyan programozási nyelv, amely különböző méretű egész számtípusokat is ismer.

Hivatalosan a Python *int* típusánál nincs határa az ábrázolható számoknak. De az adott interpreter használ egy korlátot, amely lekérdezhető a `sys.maxsize` változó értékéből.

Mint az alábbi ábrán is látható, a gépen, amelyen ez a dokumentum készült, elég nagy méretű számokkal is lehet számolni…

```
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print(sys.maxsize)
9223372036854775807
>>>
```

7.2. Lebegőpontos számok

A számítógépen a nem egész számok ábrázolására a lebegőpontos számábrázolást szokták használni. Ebből származik a *float* típus elnevezése: *floating point*, azaz *lebegőpontos szám*. A lebegőpontos számok a valós számok egy valós részhalmazát alkotják.

A lebegőpontos szám valójában a matematikából ismerős normálalak megvalósítása binárisan: a karakterisztika jelen esetben nem a tíz, hanem a kettő hatványának kitevőjét fogja jelenteni.

A memóriában a szám tárolására használt rész általában három részből áll:

1. A mantissza előjelét tartalmazó előjelbit.
2. A karakterisztika valamilyen formában.
3. A mantissza valahány számjegyen ábrázolt része

Az első rész egyértelmű: ha a bitsorozat legmagasabb helyiértéknek megfelelő bitje nulla, akkor a szám pozitív, 1 érték esetén negatív.

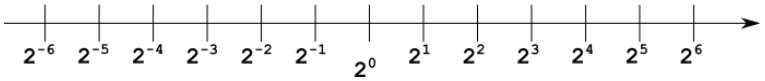
Furcsa módon ezután nem a mantissza, hanem a karakterisztika szokott következni. Mégpedig egy kicsit kódolt formában. Mivel ez egy egész szám lehet, így az egész számok ábrázolására használt módszer alkalmazható lenne, ám ekkor itt is szükség lenne egy előjelbitre. Ehelyett a legtöbbször a karakterisztika értéke annyival van eltolva, hogy a nulla érték pont az értéktartomány közepére essen. 8 bites karakterisztika esetén tehát az itt található bináris számból ki kell vonni 128-at, hogy a tényleges kitevőt megkapjuk.

A karakterisztika bitjei után következik a mantissza úgy, hogy az egész szám valahány teljes byte-ot foglaljon el.

A lebegőpontos számok ábrázolását leíró szabvány külön definálja a nulla és a végtelen értékek megadásának módját is. Ennek a definíciónak az eredménye, hogy a számítógép ismer lebegőpontos típusban +0 és -0 értéket is! Ez mindkettő úgy néz ki, hogy a mantissza rész teljesen nulla értékű (az előjelbit -0 esetén nyilván 1), a karakterisztika gyakorlatilag bármi lehet. A legtöbb aritmetikai művelet[5] amely értékül nulla eredményt adni, +0 lebegőpontos értéket eredményez.

A mantissza és a karakterisztika tárolásához használt bitek száma kétféleképpen korlátozza az ábrázolható valós számok tartományát.

5 A matematikai műveleteket a számítástechnikában aritmetikai műveletnek szoktuk nevezni.



A fenti logaritmikus számegyenes a kettő hatványokat mutatja egymástól azonos távolságra (ezért logaritmikus: a logaritmusok, azaz a kitevők vannak egyenlő távolságra egymástól). Ezen a számegyenesen a számítógépen ábrázolható lebegőpontos számok egyenletesen helyezkednek el.

Ez a gyakorlatban azt jelenti, hogy mindegyik kettő hatvány környezetében pontosan ugyanannyi szám ábrázolható. Ez abból következik, hogy minden egyes karakterisztika egy-egy kettő hatványt reprezentál és mindegyikhez ugyanazok a lehetséges mantissza értékek tartoznak.

A fentiekből következik, hogy a nullához közel sűrűn, a végtelen felé haladva egyre ritkábban levő számokat lehet ábrázolni. Gyakorlatilag azt lehet mondani, hogy a szám nagyságrendjétől függetlenül ugyanannyi értékes számjegye lehet. Azaz a nagyon nagy számok már nagyon pontatlanul ábrázolhatóak. Például valószínűleg a 3986878978768 és a 3986878978769 már azonos számként kerül tárolásra, sőt nagy valószínűséggel minden szám, ahol a 3986870000000 nulláit bármilyen más számra cseréljük, ugyanaz a szám lesz. Ez a mantissza méretének végességéből adódik.

Ha ehhez hozzávesszük, hogy két lebegőpontos szám összeadásához először azonos nagyságrendbe kell hozni őket, két nagyon eltérő nagyságrendű számnál a kisebb nagyságrendű szám teljesen elveszhet, akkor már látjuk, hogy ez a számábrázolási mód sem minden esetben használható.

Például ha a 4 000 000 000 számhoz hozzáadunk 1-et, akkor az eredmény 4 000 000 000 lesz és nem 4 000 000 001. Azért történik így, mert az 1-et a 4 000 000 000-al azonos nagyságrendre hozva a mantissza értéke 0-vá redukálódik. Ezt a jelenséget, vagyis amikor a mantissza már nem képes a szükséges számjegyeket tárolni, **alulcsordulás**nak nevezzük.

További, nem feltétlenül a lebegőpontos számábrázolásból eredő problémába ütközünk, amikor például az 1/10 értékét szeretnénk számítógépen tárolni. Az 1/10 kettes számrendszerben ugyanis végtelen szakaszos tört. Viszont a számítógépen csak adott biten ábrázolhatóak a számok, olyan számot lehetetlen ábrázolni, amelyhez végtelen számú bitre van szükség. Vagyis például az 1/10 nem ábrázolható, csak egy hozzá – az ábrázolási pontosságtól függő mértékben – közeli érték. Az ebből adódó számítási pontatlanságot nagyon jól tetten lehet érni egy táblázatkezelő programban: írd be az **A1** cellába a -10, az **A2** cellába a -9,9 értéket (ott tizedes**vessző** van!), majd a két cellát kijelölve az automatikus kitöltés módszerével másold le a kijelölést legalább az **A102** celláig! Az **A101** cellában a nullát kellene kapni, ám ehelyett egy nagyon kicsi negatív számot kapunk. Lehet, hogy a cellában a nulla jelenik meg, de megnézve az értékét, látható, hogy nem érte el a nullát. Ennek oka, hogy nem 0,1 volt a növekmény, hanem egy picit annál kisebb érték, a számítási pontatlanság pedig folyamatosan halmozódik.

8. Vezérlési szerkezetek

A programok ritkán olyanok, amelyek utasításait az elsőtől az utolsóig sorban kell végrehajtani. Sokszor fordul elő, hogy valamely feltétel teljesülésétől vagy nem teljesülésétől függően mást-mást kell csinálni, esetleg ugyanazt a műveletsort többször megismételni.

Emellett egy nagyon fontos jelentése van annak a mondásnak, miszerint „*az informatikus lusta*”: ugyanazt az utasítássort akárhányszor kell is a programnak végrehajtania, csak egyszer írjuk le. Ennek valójában nem a lustaság, hanem a nehezen felfedezhető hibák elkerülése az oka. Ugyanis ha egy utasítássorozat többször szerepel a programban és bármilyen okból módosítani kell, akkor előfordulhat, hogy nem minden előforulását írja át a programozó – valamelyikről elfeledkezik. Így máris hibássá válik a program.

A kódismétlés elkerülése akkor is lényeges, ha nem teljesen ugyanazt a kódot kell használni, hanem egymáshoz nagyon hasonlóakat, amelyek csak bizonyos paraméterekben különböznek – ez utóbbi eset hívta életre a függvényeket.

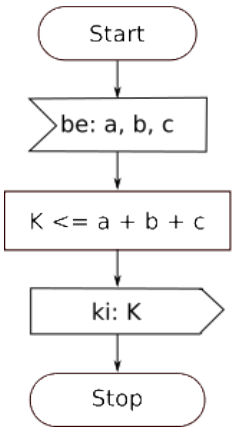
A fentiek röviden indokolják, hogy miért alakultak ki különböző vezérlési szerkezetek. A **vezérlési szerkezetek** befolyásolják a program utasításításainak végrehajtási sorrendjét. Az alábbiakban röviden ezen vezérlési szerkezetek összefoglalása következik, majd az ezt a rövid leckét követő további leckék fogják a vezérlési szerkezetek egy részét a hozzájuk tartozó Python utasításokkal együtt részletesen is bemutatni.

8.1. Szekvenciális vezérlés

Amikor a program utasításait a leírásuk – memóriában való elhelyezkedésük – sorrendjében kell végrehajtani, azt **szekvenciális vezérlés**nek nevezzük.

Ez az alapértelmezett vezérlési szerkezet.

Az 8.1. ábrán egy szekvenciális vezérlést alkalmazó folyamatábra látható. Mint az ábra mutatja, a program kiindulópontjától mindig csak egy nyíl vezet a következő művelet blokkjához, nincs lehetőség más irányt választani, csak azt az egyet, ami egyenesen a *Stop* blokkhoz vezet, ahol véget ér a folyamat.



8.1. ábra: Példa szekvenciális vezérlésre

8.2. Elágazás

Az elágazás során valamely feltételtől függően a végrehajtás legalább két különböző irányba folytatódhat. Ezt a folyamatábrán általában úgy jelöljük, hogy egy rombuszba beleírjuk a feltételnek megfelelő kérdést, és a rombuszból továbbhaladó két nyílra ráírjuk az „*igen*” illetve a „*nem*” szöveget. A kérdésre adott válasznak megfelelő nyíl mentén kell továbbhaladni a következő műveletet megadó blokkra.

Az elágazásban érintett utasítások után általában az egyes ágak ismét egyesülnek és együtt haladnak tovább, azaz az elágazás a szekvenciális végrehajtás egyik műveleteként szerepel. De az is elképzelhető, hogy az egyes ágak már nem találkoznak újra. Ilyen esetben a program több ponton is véget érhet, amit a folyamatábra több *Stop* blokkjával jelez.

8.3. Ismétlés, ciklus

Előfordulhat, hogy bizonyos műveleteket egymás után több alkalommal meg kell ismételni. Előfordulhat, hogy előre meg van határozva, hány alkalommal kell valamit megismételni (pl.: „*Százszor leírod, hogy »Nem pisilem le többet a tanítónéni lábát«!*”), de az is lehetséges, hogy egy feltétel teljesülésétől függ, hogy kell-e valamit ismételni – vagy egyáltalán végrehajtani.

Folyamatábrán ennek nincs külön jelzése, csupán annyi, hogy a feltételt megadó rombuszból induló egyik útvonal előbb-utóbb visszacsatlakozik a feltételvizsgálatot jelző rombusz elé, így azon az útvonalon vérehajtott műveletek után ismét a feltétel vizsgálatához ér a folyamat.

8.4. Eljáráshívás

Amennyiben egy program több, kisebb algoritmusra bontható – a hétköznapi életben használt programok jellemzően ennyire bonyolultak –, akkor ezeket a kisebb algoritmusokat önállóan szokás elkészíteni. Egyes nyelvekben eljárásnak, más nyelvekben – mint a Python vagy a C – függvényeknek hívják ezeket. Amikor az ezen eljárásokban, idegen szóval *szubrutin*okban definiált tevékenység végrehajtására van szükség, akkor az adott szubrutint meg kell hívni.

Ilyen eljáráshívás volt eddigi programjainkban például a **print()** vagy az **input()** függvény használata. Az előbbi az eljárás, az utóbbi a függvény példája lenne az olyan nyelvekben, amelyek különbséget tesznek köztük (mint a Pascal), hiszen az előbbi nem ad a végrehajtásával vissza eredményt, míg az utóbbi visszaadott értékét felhasználjuk. Ebből a szemszögből nézve az eljárást önálló utasításként meghívható alprogramnak, a függvényt pedig kifejezésben egy érték megadásaként meghívandó alprogramnak lehet tekinteni.

Néhány leckével később azt is megismered majd, hogyan definiálhatsz saját magad is függvényeket arra a célra, hogy egy több helyen is szereplő kódrészletet ne kelljen többször megismételni. Ott majd arról is lesz szó, hogyan lehet a függvénynek paramétereket adni. Ezek a paraméterek valójában a függvény által megvalósított algoritmus bemeneteként funkcionálnak, ahogyan a visszatérési értéke pedig az algoritmus kimeneteként.

8.5. Eseményvezérlés

A grafikus felülettel rendelkező programok általában idejük legnagyobb részében türelmesen (mert egy program nem tud türelmetlen lenni) várnak. Egészen addig, amíg a felhasználó valamit nem csinál, amellyel kivált egy eseményt, amire a programnak reagálnia kell.

Előfordul olyan is, hogy a program végrehajtása során – akár a program futása következtében, akár attól függetlenül – valamilyen hiba keletkezik.

Mindkét eset példa arra, hogy egy esemény bekövetkezik, amelyre a programnak valamilyen módon reagálnia kell. Az ilyen esetekre írt *esemenyvezérlő* függvényekkel lehet ezeket az eseményeket kezelni. Az interaktív programok alapvető jellemzője, hogy a tevékenységét nem szekvenciális vezérléssel, hanem eseményvezérlésen keresztül valósítja meg.

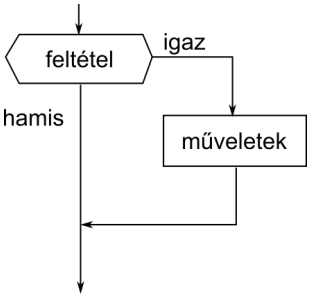
Szintén eseményvezérlésre épülnek a mikrovezérlők, mint amilyen az általános iskolából talán már ismerős micro:bit vagy Arduino, de amilyenek a nagy gyártósorok vezérlésére is használt PLC-k. Ezek alapból a beérkező jelek formájában meghatározott eseményekre adnak választ.

A mikrovezérlőkhöz hasonló módon működik a legtöbb játékprogram is, amelyek egy ciklusban várnak valamilyen állapotváltozásra, amelyek egy része a játékos cselekede, majd erre az állapotváltozásra történik valami.

Ilyen eseményverzélés a középiskolai programjainkban elég ritkán fognak előfordulni.

9. Elágazások

Az elágazás teszi lehetővé, hogy a program valamely feltétel teljesülésétől függően más-más viselkedést produkáljon. A legegyszerűbb esete ennek a 9.1. ábrán látható egyágú vagy egyirányú elágazás. Ebben az esetben egy feltételtől függően valamely művelet(sor) végrehajtódik vagy sem. Ha a feltétel teljesül, a művelet(sor) végrehajtásra kerül, különben nem történik semmi.



9.1. ábra: Egyirányú elágazás

Tegyük fel, hogy a program bekér egy egész számot. Ha ez páros, akkor ki kell írni ennek a tényét. Ezt követően – tehát attól függetlenül, hogy a beolvasott szám páros-e – be kell kérni még egy számot.

Ez Python esetén így néz ki:

```
a = int(input())    # nem írunk kérdést
if a % 2 == 0:
    print("A szám páros.")
b = int(input())    # nem írunk kérdést
```

Az elágazást megvalósító utasítás a második és a harmadik sorban található. Az **if** kulcsszó után található a feltétel (`a % 2 == 0`), amelynek *True*, azaz *igaz* értéket kell adnia, ahhoz, hogy a hozzá tartozó utasítások végrehajtódjanak.

```
1   a = int(input())
2   if a % 2 == 0:
3       print("A szám páros.")
4   b = int(input())
```

9.2. ábra: Kettőspont a sor végén, következő sor négy szóközzel bentebb kezdve.

A harmadik sor bentebb kezdődik. Pythonban van egy fontos szabály: ha a sor kettősponttal végződik (9.2. ábra piros nyila), akkor a következő sort bentebb kell kezdeni, alapértelmezésben négy szóközzel (zöld nyíl az ábrán). De a VS Code (ahogy a PyCharm is) a kettősponttal végződő sor végén az Entert lenyomva, ezt megteszi helyettünk. A fejlesztő környezet egyébként a Tabulátor billentyű megnyomásával pont ennyivel visszi bentebb a sort (vagy több sor kijelölése esetén sorokat). Ez a behúzás jelenti, hogy ez a sor (vagy ezek a sorok) az **if** (vagy más kettősponttal zárt sorral kezdett) utasításhoz tartoznak mindaddig, amíg véget nem ér a bentebb kezdés. Mivel a negyedik sor már nincs bentebb kezdve, így az már nem tartozik az **if** utasításhoz, hanem követi azt.

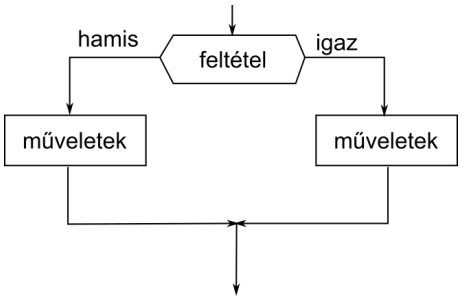
A program végrehajtása a következőképpen történik:

1. Az **if** utasítás utáni logikai típusú értéket adó kifejezés értékének meghatározása. Ez ebben az esetben akkor igaz, ha az *a* változó értéke kettővel osztva nulla maradékot ad (azaz *a* osztható kettővel).
2. Amennyiben a feltétel igaz logikai értéket eredményez, akkor a következő, bentebb kezdett sorokban levő utasítások végrehajtásra kerülnek. Itt most ezen sorok száma történetesen 1.
3. A bentebb kezdett utasítások végrehajtásával az **if** utasítás végrehajtása is befejeződik, azaz akár igaz volt a feltétel, akár hamis, a bentebb kezdés utáni utasítás – jelen esetben a *b* változó értékének bekérése – lesz a következő végrehajtandó utasítás.

Mint látható, a bentebb kezdésnek jelentősége van: ha a negyedik sor a harmadikkal egy oszlopban kezdődne, akkor csak abban az esetben történne meg a *b* változó értékének beolvasása, ha az *a* változóban páros szám található. Ez azt is jelenti, hogy ha a *b* változó korábban nem kapott értéket a programban, akkor ezen a ponton sem biztos, hogy értéket kap, azaz a program egy későbbi részében adódhat olyan helyzet, hogy a változóra a program anélkül hivatkozik, hogy a változó létezne!

A bentebb kezdés mindig annak jelzésére szolgál a Python nyelvben, hogy az adott sor egy korábban elkezdődött utasítás része. **A behúzás mértékének a programon belül mindig egységesnek kell lennie.** A legbiztosabb, ha megjegyezzük: mindig négy karakternyi a behúzás. Sokszor nagyon nehezen megtalálható hibát szokott okozni, ha valamelyik sor elején van például egy felesleges szóköz, amit a program szintaktikai hibaként kezel (*wrong indentation*). De ennél még nehezebb észrevenni a hibát, ha a bentebb kezdés nem okoz szintaktikai hibát, csak egyszerűen valamelyik sor nem jó pozíción kezdődik, így nem annak az utasításnak a része, amelynek kellene lennie.

9.1. Kétirányú elágazások



9.3. ábra: Kétirányú elágazás

Van lehetőség arra is, hogy egy feltételtől függően két alternatív művelet(sor) valamelyikét hajtsuk végre. Ennek folyamatábrája látható a 9.3. ábrán. Az előző példánál maradva, ha a páratlan számnál is akarunk valamit kiírni, akkor az **if** utasítást egy különben ággal kell bővíteni:

```
1  a = int(input())
2  if a % 2 == 0:
3      print("A szám páros.")
4  else:
5      print("A szám páratlan.")
6  b = int(input())
```

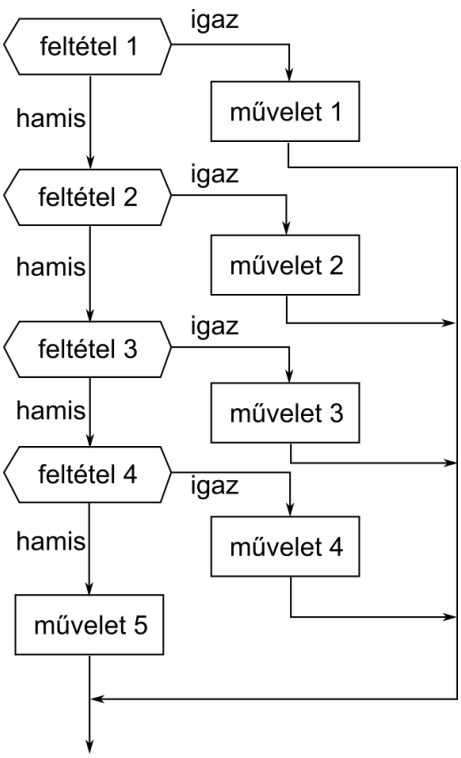
Az újonnan szerepelő 4–5. sor még az **if** utasításhoz tartozik. De akár úgy is tekinthetjük, mint önálló **else** utasítást, amelyet kötelezően **if** (vagy **elif**, **while**, **for**) utasítás után kell írni… A lényeg, hogy az 5. sor utasítása akkor hajtódik végre, ha a 2. sorban szereplő feltétel nem igaz (tehát hamis). Mint látható, az **else** utasítás az **if** kulcsszóval egy oszlopban helyezkedik el. Feltétele nincs, hanem rögtön egy kettőspont zárja, amelyből következik, hogy a következő sort bentebb kell kezdeni. Az ebben a bentebb kezdésben szereplő utasítások akkor lesznek végrehajtva, ha az **else**-t megelőző, vele egy oszlopban levő **if** utasítás feltétele nem teljesült. Vagyis ebben az esetben a két ág közül valamelyik kerül végrehajtásra, mielőtt a következő utasítással – most a *b* változó értékének bekérésével folytatódna a program.

9.2. Több ágú elágazások

Vannak esetek, amikor nem egy feltétel igaz/hamis értéke alapján kell eldönteni, hogy mi a teendő, hanem kettőnél több lehetőség közül kell valamelyiket kiválasztani. Más programozási nyelvekben erre való az **esetszétválasztás**. Az esetszétválasztás esetén egy adott kifejezés különböző értékei jelentik az elvégzendő tevékenységek közti választást. A C nyelvcsaládban például a **switch** utasítás valósítja meg az esetszétválasztást, Pythonban hasonló a 3.10. verzióban bevezetett **match** utasítás.

Van viszont lehetőség arra, hogy elágazások láncolatát hozzuk létre. Ez az „*elágazáslánc*” valójában egy kicsivel szabadabb lehetőséget nyújt a klasszikus esetszétválasztáshoz képest, mivel nem csak egy adott kifejezés, hanem minden elágazási pont esetében akár más-más kifejezés is használható.

A 9.4. ábra egy ilyen elágazáslánc folyamatábráját mutatja. Tekinthető ez olyan esetszétválasztásnak, amelynél minden esetet saját feltétellel látunk el. Mint az ábrán látható, az egyes feltételek vizsgálata egymás után, meghatározott sorrendben történik. Amennyiben valamelyik feltétel igaznak bizonyul, a hozzá tartozó művelet(sor) végrehajtása következik. **Ezután az elágazás véget ér.** Azaz a további feltételek vizsgálatára már nem kerül sor! Amennyiben egyik feltétel sem teljesül, akkor is van lehetőség természetesen valamilyen műveletsor megadására.



9.4. ábra: Elágazások láncolása

Ezt a megoldást általában úgy lehet megvalósítani, hogy az **if** utasítás hamis ágában egy újabb **if** utasítást szerepeltetünk a következő feltétel megadásával. Ebben a példában ez négy egymásba ágyazott **if** utasítást jelentene, amelyek közül nyilván a legbelső *else* ágában szerepelne az 5. művelet(sor). Az ilyen, egyre bentebb kezdődő részeket alkalmazó megoldásokat azonban célszerű kerülni, mert rontja a program olvashatóságát. Szerencsére a Python **if** utasítása alapból kínálja a feltételek láncolásának lehetőségét az **elif** kulcsszóval. Ez gyakorlatilag az *else* és az *if* szavak összevonását jelenti. Utána máris írható a következő feltétel. Az utasítás végén továbbra is szerepelhet az **else** kulcsszó arra az esetre, amikor egyik feltétel sem teljesült.

A példában szereplő elágazás láncolat Pythonban így néz ki:

```
if feltétel 1:
    művelet 1
elif feltétel 2:
    művelet 2
elif feltétel 3:
    művelet 3
elif feltétel 4:
    művelet 4
else:   # ha egyik feltétel sem teljesült...
    művelet 5
```

Fontos hangsúlyozni, hogy az elágazások ilyen láncolása esetén a feltételek mindig az elejétől kezdve kerülnek megvizsgálásra *az első igaznak bizonyulóig*. A 9.5. ábrán látható példakód esetén például a „*Pont öt.*” szöveg kiírására soha nem kerül sor, mivel az ehhez azághoz tartozó feltétel vizsgálatára nem kerül sor, ha igaz lenne, mivel ekkor már a legelső feltétel is igaz volt és a „*Nagyobb négynél.*” szöveg kiírása után az utasítás végrehajtása befejeződik.

```
 8  if a > 4:
 9      print("Nagyobb négynél.")
10  elif a < 0:
11      print("Negatív.")
12  elif a == 5:
13      print("Pont öt.")
14  else:
15      print("Ötnél kisebb nem negatív.")
```
9.5. ábra: Példa hibásan szervezett feltételekre

9.3. Blokkok a programban

A Python nyelvben a program sorait különböző blokkokra lehet osztani. Ezek a blokkok a vezérlési szerkezetek fontos elemei. A blokkokra vonatkozó szabályok:

- Amennyiben a bentebb kezdés mértéke nő, ott egy új blokk kezdődik.
- Egy blokk további blokkokat is tartalmazhat.
- A blokk véget ér, amikor a bentebb kezdés visszaáll a tartalmazó blokknál alkalmazottra, vagy tartalmazó blokk hiányában nullára.

A blokkok megértéséhez nézzük a következő programrészletet:

```
1  name = 'Mary'
2  password = 'swordfish'
3  if name == 'Mary':
4      print('Hello, Mary')
5      if password == 'swordfish':
6          print('Access granted.')
7      else:
8          print('Wrong password.')
9
```

Az első blokk a 4. sorban kezdődik (`print('Hello, Mary')`) és egészen a 9. sorig tart. Ebben van a második blokk, amely a 6. sorból áll, valamint a harmadik, amely a 8. sorban van.

9.4. Feladatok

9.1. Kérjünk be jelszót a felhasználótól és hasonlítsuk össze a programban tárolt változattal! Ha a felhasználó eltalálta a jelszót, írjuk ki, hogy „Helyes jelszó.”, különben „Hozzáférés megtagadva!”.

9.2. Kérjünk be két számot a felhasználótól és írjuk ki közülük a nagyobbat!

DONALD KNUTH (1938, Milwaukee –) amerikai matematikus *A számítógép-programozás művészete* címmel készített egy sok kötetes könyvet, amelyben a számítógépes algoritmusokról ír. A sorozat még mind a mai napig nincs befejezve, az utolsó kötetek még tervezés alatt állnak. Knuth a könyvsorozat írása kapcsán előállt nyomdai nehézségek miatt alkotta meg a TeX nevű programot, amelyre épült a későbbiekben a LaTeX nevű, a tiográfiai szabályok alkalmazásával történő, nem WISYWIG rendszerű szövegszerkesztő program, amely a matematikai formulák megjelenítésére is gond nélkül képes – ellentétben a legtöbb egyéb szövegszerkesztővel.

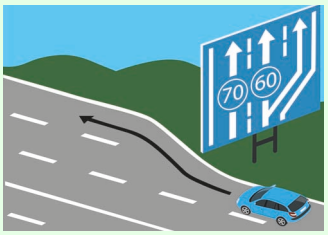
A könyv címét érdemes komolyan venni: a programozás valóban tekinthető művészetnek is.

9.3. Írjunk olyan programot, amely bekér két csapatnevet és két pontszámot, majd kiírja a mérkőzés eredményét az alábbi minta szerint (az alábbi programkimenetben a felhasználó válaszai vastagabbal vannak szedve):

```
Mi az egyik csapat neve? Tópart királyok
Hány pontot szerzett a Tóparti királyok? 78
Mi a másik csapat neve? Talpasi csodatévők
Hány pontot szerzett a Talpasi csodatévők? 54
Az összecsapás eredménye:
Tóparti királyok – Talpasi csodatévők
78 : 54
Tóparti királyok nyert 24 ponttal.
```

Amennyiben a két csapat azonos pontszámot ért el, akkor az utolsó sorban az „Az eredmény döntetlen.” szöveg legyen!

9.4. A mellékelt kép egy online KRESZ-tesztből származik. A hozzá tartozó kérdés az, hogy ha egy meghatározott sebességgel a táblához érkezik az autó, akkor melyik sávban kell továbbhaladnia. Készíts programot, amely megkérdezi a felhasználótól az autó pillanatnyi sebességét, majd válaszul kiírja, melyik sávban haladjon:



1. 60 km/h alatt a jobboldali kapaszkodósávban;
2. 70 km/h alatt a középső sávban;
3. efelett mehet a belső sávban.

9.5. Készítsünk programot, amely bekéri a felhasználó nevét és életkorát! Az adatbekérés után javasoljon a program a felhasználónak az életkorának megfelelő olvasnivalót!

- 0–3 év: „Totyogóknak a kettes számrendszerről”
- 4–6 év: „Hackeljük meg az óvodát!”
- 7–14 év: „Felhőtechnológia a menzán”
- 15–18 év: Big data a középiskolában”
- 19 évtől: „DONALD KNUTH: A számítógép-programozás művészete”

10. Függvények definiálása és használata

A következő vezérlési szerkezet, amellyel részletesebben megismerkedünk, a függvények, eljárások hívása. Ennek használatához előbb-utóbb hasznos ismerni azt is, hogyan lehet ilyen függvényeket létrehozni (a Python nyelvben nincs külön megkülönböztetve az eljárás). Mielőtt azonban ennek neki kezdenénk, a programfejlesztéssel kapcsolatban is néhány dolgot érdemes közelebbről megismerni.

10.1. Programfejlesztési módszerek

Az elmúlt évtizedek során különféle programfejlesztési módszerek fejlődtek ki. Ezek mind azért alakultak ki, mert egy összetettebb programnál nem fog működni az a megoldás, hogy leülök és begépelem a programot, kipróbálom, majd megkeresem benne a hibát. Általában sem szerencsés ötlet hosszabb programkódot írni annak kipróbálása nélkül.

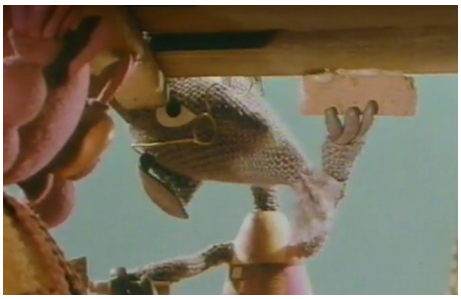
Egy összetettebb feladatot egyszerre át sem lehet már látni. Célszerű kisebb részekre bontani, és azok megoldásával foglalkozni, majd azokból összeépíteni az egész megalkotandó programot.

Ezen az úton többször végighaladva lehet a problémát egyre kisebb részproblémákra bontani, hogy az már átlátható, leprogramozható legyen. Az ilyen részproblémákra írt programkódot szokás függvényekbe helyezni.

Másik oka a feladat részfeladatokra bontásának, hogy a különböző típusalgoritmusokat alkalmazni lehessen. Egy-egy részfeladat megoldása mögött ugyanis sokszor lehet találni egy már pontosan leírt algoritmust. Ebben az esetben csak annyi a feladat, hogy az adott algoritmust alkalmazzuk a konkrét feladatra. A későbbiekben több olyan lecke is fog szerepelni, amely ezek közül az algoritmusok közül mutat be néhány egyszerűbbet – a többi ilyen algoritmust egyetemen tanulják meg a programozónak készülők.

10.1.1. Fentről lefelé történő fejlesztés

Ez a módszer elsőre olyannak tűnik, mint amikor Mekk mester házat épített és először a tetővel kezdte. De azért nem erről van szó!



10.1. ábra: Mekk mester házat épít felülről lefelé haladva (kép: a rajzfilmsorozat második részéből, az indavido videójából kifényképezve)

A felülről lefelé fejlesztés során először nagy vonalakban megtervezzük a főprogram működését. Ennek során lehet azt is csinálni, hogy a programkódba a későbbi műveleteket megjegyzés formájában leírjuk a saját szavainkkal. Ekkor még csak a nagyobb lépéseket kell leírni.

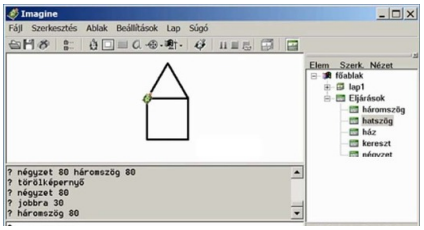
Ezután az egyes műveletek helyére – akár a megjegyzéseket is megtartva, akár azok helyére – függvényhívásokat teszünk. Ezek a függvények ekkor még nincsenek definiálva, tehát a fejlesztőkörnyezet mindre hibát fog jelezni. Ám a modern fejlesztőkörnyezetek erre a hibára javítási lehetőségként valamilyen módon felkínálják általában a hiányzó függvény definiálását.

Ezután az egyes függvényeken belül tovább bontjuk annak a műveletnek a megvalósítását, amelyet az adott függvénynek meg kell valósítania. Eközben ha egy olyan tevékenység szükséges, amelynek leprogramozását az adott helyen még nem akarjuk elvégezni (például mert azt még át kellene gondolni, de túl sok ideig tartana és megakasztaná az adott függvény egészének működését megvalósító kód megírását), akkor ott újra elhelyezhetünk egy még nem létező függvényt meghívó utasítást. Ennek a függvénynek a megvalósítását azután majd a következő körben végezzük el.

Ezzel a módszerrel akárhány körre bontható a program fejlesztése, minden körben egy szinttel részletesebben kidolgozva a program működését, míg végül el nem jutunk a végső programhoz.

A módszer előnye, hogy minden egyes körben csak az adott körnek megfelelő részletességgel kell végiggondolni a teendőket, csak az utolsó szinten kell ténylegesen a program utasításainál szükséges részletességgel megadni a lépéseket.

10.1.2. Alulról felfelé fejlesztés



10.2. ábra: Imagine Logo

Amikor alulról felfelé halad a fejlesztés, akkor először kis építőelemeket készítünk. Az egyszerű műveletekre írunk egy-egy függvényt, majd ezekből újabb függvényeket állítunk össze, ismételve ezt az összeépítést, amíg a végső programhoz nem jutunk.

Például az Imagine Logo használata esetén ezt a módszert szokás alkalmazni, ott a függvények helyett az eljárás elnevezés használatos. A 10.2. ábrán látható egy példa (a Mozaik kiadó honlapjáról származik a kép): a négyzet és a háromszög megrajzolását végző eljárásokból épül fel a ház megrajzolását végző függvény. Ezt lehetne folytatni tovább egy utca megrajzolását végző függvénnyel...

Ez a módszer nagyobb előre tervezést igényel, hiszen a kis alapelemek megírásakor tudni kell, hogy milyen műveletekre lesz szükség. Ám ilyenkor könnyebb szétosztani az elkészítendő függvényeket más-más programozókra.

Az alulról felfelé történő fejlesztést támogatja a **programkönyvtárak** fejlesztése. A programkönyvtár olyan *eljárásgyűjtemény*, amely valamely területen potenciálisan előforduló valamennyi műveletet megvalósító eljárást (függvényt) tartalmaz. Ezeket egy program írásakor készen felhasználhatjuk. Amikor valaki programkönyvtárat fejleszt, akkor nem feltétlenül tudhatja előre, milyen programokban fogják azt felhasználni.

Rengeteg példa hozható programkönyvtárakra. Íme ezek közül néhány:

- A Python moduljai tekinthetőek ilyen programkönyvtáraknak. Ezek egy-egy terület műveleteinek megvalósítására szolgáló függvényeket és ezekhez kapcsolódó változókat definiálnak. Pontosabban ennél is többet, de az már az objektum-orientált programozáshoz tartozik...
- A Qt egy C++ nyelven írt programkönyvtár. Elsősorban talán a grafikus felület elkészítéséhez szükséges eszközök miatt használatos, de más lehetőségeket is tartalmaz. Nagyon sok olyan szoftver van, amelynek megírásához használták a Qt valamelyik verzióját. Ilyen például a LibreOffice is. De egész grafikus ablakkezelő rendszereket is meg lehet példának említeni, amelyek a Qt programkönyvtárát használják, mint a KDE vagy a Deepin. A Qt-hez a Python is kínál hozzáférést specális modulokon keresztül.
- A Microsoft által a C#-hoz kifejlesztett (de már nem csak a C#-ban használható) .NET is gyakorlatilag tekinthető ilyen programkönyvtárnak, amellett, hogy ez egy futtató környezet is (azaz egy interpreter a .NET által használt közbülső nyelvre lefordított, például C#-ban megírt programoknak).

A két fent bemutatott programfejlesztési módszert a gyakorlatban nagyon ritkán használják kizárólagosan. Inkább a kettő keverékét – esetleg további módszerekkel együtt – szokás használni.

Például a fentebb említett programkönyvtárakat már mint létező eljárásokat beépítve egy egyebként alapvetően fentről lefelé haladva fejlesztett programba, máris keverve használjuk a két módszert. Ezen felül ha alapvetően felülről lefele halad is a program megírása, egy olyan függvényt, amelyet az egyik ponton kellett definiálni, már kész építőelemként lehet máshol is felhasználni, ahol az szükséges.

10.2. Függvények definiálása

A függvények arra is használhatóak, hogy ugyanazt a programkódot ne kelljen egy programban többször leírni. Amennyiben ugyanis ugyanaz a kód többször szerepel a programban és utólag módosítani kell valamiért, akkor mindenhol módosítani kell. **Tehát ha egy programban több helyen akarod ugyanazt a műveletsort szerepeltetni, akkor írd meg függvényként és csak a függvényt hívd meg a programban!** Ha az egyes helyeken nem ugyanazokkal az adatokkal kell ugyanazt a műveletsort elvégezni, akkor is: a függvények paraméterezhetőek.

Valójában a függvény úgy tekinthető, mint egy algoritmus megvalósítása. Az algoritmus bemeneti adatai a függvény *paraméterei*, kimenete pedig a függvény *visszatérési értéke*. Gyakorlatilag a **print()** függvény is egy algoritmust valósít meg, amelynek bemenete a függvény paramétereként megadott kifejezések sorozata, a fenti értelemben vett kimenete pedig nincs. Valójában van: a képernyőre kiírt szöveg. Az **input()** függvénynek kimenete is van: a felhasználó által begépelt szöveg.

A függvényt egyszer kell definiálni a programban, majd mindenhol, ahol „látszik”, meghívható. Python nyelv esetén a függvények definíciója mindig a program elején, a „főprogram” utasításai előtt szerepeljenek!

A definíciót egy `def` kulcsszóval kell kezdeni. A `def` kulcsszó után a definiálandó függvény neve, majd a *paraméterlistája* szerepel. Ezt a sort egy kettőspont zárja, amiből már következik, hogy a következő sortól indul a függvénydefiníció blokkja – bentebbkezdéssel.

```
5  def nagyobb(a: int, b: int) -> int:
6      """
7      Ez a függvény a két paraméter közül a nagyobb értékét adja vissza.
8      :param a: Az egyik egész szám.
9      :param b: A másik egész szám.
10     :return: A két egész szám közül a nagyobb érték.
11     """
12     if a < b:
13         return b
14     else:
15         return a
```

Ez a programrészlet egy függvénydefiníciót mutat. Az 5. sorban látható a függvénydefiníció-fejléce az előző bekezdésben leírtaknak megfelelően, a többi sor a hozzá tartozó blokkot tartalmazza. A fejlécben a `def` után szerepel a függvény neve, amellyel a későbbiekben meg lehet hívni a függvényt. Ezután kerek zárójelek között szerepel a paraméterlista – vagy más néven argumentumlista[6]. Ez a példában bővebb a ténylegesen kötelezőnél.

A paraméterlista lehet üres is, de mindenképpen szerepelnie kell. Üres paraméterlista esetén a függvény meghívásakor nem kap paramétert, de ott is szerepeltetni kell az üres paraméterlistát, mivel a kerek zárójelekről fogja tudni az értelmező, hogy függvény hívása történik, nem egy változó megadása – ugyanis a függvény nevével, mint változóval is lehetne bizonyos esetekben kifejezést megadni, de ezzel nem fogunk foglalkozni.

A fenti példában a függvénynek két paramétere lesz, mindkettő esetében meg van határozva, hogy az egész típusú legyen. A paraméterek értékeit a függvény definíciós blokkjában változóként lehet használni azzal a névvel, amellyel a definíció fejlécében szerepelnek. Az egyes paraméterek tí-

6 A függvénydefinícióban szereplő paraméterlista neve hivatalosan argumentumlista, míg a függvénymeghívásakor szereplő kifejezések a paraméterek, de egyszerűbb mindkét esetben paraméterekről beszélni.

pusát a Python nyelv esetében nem kötelező megadni, más nyelvek esetében azonban ez kötelező lehet. Annak ellenére érdemes azonban megadni a típust, hogy nem kötelező. Ekkor ugyanis a fejlesztő környezet figyelmeztethet, ha a függvény meghívásakor nem a megfelelő típusú értéket adjuk a paraméternek. Ha megadjuk a típust, akkor a paraméter nevét követően egy kettőspontot követően kell azt megadni, ahogy a példában is szerepel.

Hasonlóan elhagyható rész a paraméterlista utáni `->` és utána egy típusnév. Ez a függvény által visszaadott érték várható típusát adja meg. Vannak esetek, amikor a függvény nem minden esetben ugyanolyan típusú értéket ad vissza, amire ilyenkor a fejlesztőkörnyezet figyelmeztetni szokott. Kicsit lentebb lesz erre majd példa.

A függvény szokásos dokumentálásának a PyCharm által felkínált módját mutatja a 6–11. sor megjegyzése. A PyCharm használata esetén a kezdő tripla idézőjel (szimpla vagy dupla esetén is) begépelésére automatikusan létrejön. A VS Code nem feltétlenül hozza automatikusan létre és ha igen, akkor nem ilyen alakban, de mégis mindig ezt a formát érdemes követni, mivel ebből a megjegyzésből már a fejlesztőkörnyezetben is elérhető dokumentáció készül, amely például a VS Code esetén a függvényhívás beírásakor is megjelenik egy buborékban. Emellett vannak automatikus dokumentumkészítő szoftverek, amelyek ebből a megjegyzésből készítik el a dokumentációt.

1. A függvény rövid ismertetését adjuk meg először.
2. Ezután minden egyes paraméterre a `:param` *`név`*`:` kezdettel adjuk meg az adott paraméter funkcióját.
3. A végén a `:return:` kezdetű sorral lehet megadni, hogyan kell a függvény által visszaadott értéket felhasználni.

10.2.1. A függvény visszatérési értéke

Vannak olyan programozási nyelvek, amelyek elnevezésben különbséget tesznek eljárás és függvény között. Ilyen például a Pascal nyelv, ahol az eljárás nem ad vissza értéket, meghívása egy utasításként történik, míg a függvény értéket ad vissza az őt tartalmazó kifejezésbe, ahova a visszatérési értéket kell a függvényhívás helyére behelyettesíteni. A Python – ahogyan a C nyelvcsalád – nem tesz különséget, mivel bárhol ahol utasítás szerepelhet, ott kifejezés is megadható akár értékadás nélkül is, amely esetben a kifejezés értéke egyszerűen figyelmen kívül lesz hagyva. (Ám ha a kifejezés értékének meghatározása közben történik valami *mellékhatás*, akkor az megtörténik).

A függvény végrehajtása során utoljára mindig egy **return** utasítás hajtódik végre. Ez akkor is igaz, ha ez az utasítás nem szerepel a függvényt definiáló blokk végén. Ekkor az ott szereplő utasítás után automatikusan „odaképzeli” azt az értelmező. Ha viszont szerepel, akkor utána egy kifejezést lehet megadni. Ennek a kifejezésnek az értéke lesz a függvény által visszaadott érték.

Bárhol szerepelhet a **return** utasítás a függvény definíciójában, ahogy a példánkban is látható a 13. és a 15. sorban. Ahol az utasítás szerepel, ott véget ér a függvény és az utasításban szereplő kifejezés értékét adja vissza oda, ahol a függvény meghívása történt.

Amennyiben a **return** utasítás hiányzik vagy szerepel, de kifejezés nélkül, a függvény akkor is ad vissza értéket, de ez a speciális `None` érték lesz. Ez az érték a *NoneType* típus egyetlen lehetséges értéke. Ez eltárolható változóban is, vizsgálható az `is None` és az `is not None` logikai kifejezésekkel. Ilyen módon lehetőség van arra is, hogy ezt a „nem értéket” hibajelzésre használjuk: ha a függvény a végrehajtása során hibába ütközött, akkor a `None` értéket visszaadva ezt a tényt vizsgálva a függvényhívás után, a hibát a program kezelni tudja.

Nézzük erre példaként az osztás esetét! Amennyiben egy osztást tartalmazó kifejezésben az osztó nulla, akkor a program hibaüzenettel elszáll, hiszen a nullával való osztás nem értelmezett. Ezt elkerülendő készítsünk függvényt, amely úgy végzi el az osztást, hogy ha az osztó nulla, akkor a program nem ér véget hibaüzenettel, de a függvény hívásának helyén ez leellenőrizhető.

Ilyen egyszerű műveletekre azért nem érdemes a valóságban függvényt írni, mert a függvényhíváshoz szükséges extra tevékenység sokkal több időt igényel, mint magának a függvény által végrehajtott osztásnak a végrehajtása. A függvény csak a példa kedvéért szerepel, bonyolultabb esetekben a módszer tényleg hasznos lehet.

```
18  def osztas(osztando: float, oszto: float) -> float:
19      """
20      A függvény elvégzi a valós osztást, anélkül, hogy a nullával történő osztás hibát okozna.
21      :param osztando: Az elosztandó lebegőpontos szám.
22      :param oszto: Az osztó, lebegőpontos számként.
23      :return: Az osztás eredménye lebegőpontos számként vagy None, ha az osztó 0.
24      """
25      if oszto == 0:
26          return None
27      return osztando / oszto
```

A fenti kódrészlet 26. sorában a `None` szóra a PyCharm, amelyben a kép készült, azt jelzi, hogy a kifejezés típusa nem felel meg a fejlécben a visszatérési értéknek megadott típusnak. Ha ezt a figyelmeztetést (*Warning*) el akarjuk kerülni, akkor vagy a `None` szerepeltetését kell elhagyni – akkor nincs mit más színnel kiemelve jelezni a problémát –, vagy a visszatérési érték típusát nem kell megadni.

A fenti függény először ellenőrzi az osztó paraméter értékét. Ha ez nulla, akkor nem engedi az osztást végrehajtani, hanem a `None` értéket hibajelzésként használva véget ér a függvény.

Mivel nem garantált, hogy az osztás végrehajtódott, a függvény hívását nem szabad ott elhelyezni, ahol az osztás értékére szükség van, hanem előtte ellenőrizni kell azt. Viszont hogy ne kelljen fölöslegesen kétszer meghívni ugyanazokkal a paraméterekkel a függvényt, a függvény értékét el kell tárolni egy változóban. Ezt a változót megvizsgálva azután ellenőrizhető, hogy a függvénynek van-e felhasználható értéke. Az alábbi kódrészlet a fenti függvény helyes meghívására mutat példát:

```
hanyados = osztas(a, b)
if hanyados is None:
    print("Nullával nem lehet osztani!")
else:
    print(hanyados)
```

Nagyon fontos megjegyezni, hogy a **return** utasítás bárhol szerepel is a függvény kódjában, ott véget ér a függvény végrehajtása! Amennyiben egy ciklus magjában szerepel, akkor a ciklus is azonnal véget ér, ahogyan a **break** utasítás befejezné azt, majd rögtön a függvény is véget ér. Amennyiben a függvény kódjában a **return** után is írunk utasításokat, azok soha nem fognak végrehajtódni. Ezeket az utasításokat a fejlesztőkörnyezetek esetleg más megjelenéssel jelzik is.

10.2.2. Több érték kezelése egy értékként

Készítsünk függvényt, amely a paraméterében szereplő *a*, *b*, *c* lebegőpontos számokból kiszámolja az *a*, *b*, *c* oldalú háromszög kerületét, illetve területét! Amennyiben a megadott értékekből nem szerkeszthető háromszög, akkor `None` legyen a visszaadott érték!

Első lépésben készítsük el a két értéket kiszámoló függvényt külön-külön: a kerületet számoló függvény neve **kerulet()** legyen, a területet számolóé **terulet()**.

$$K = a + b + c$$

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

$$s = \frac{a+b+c}{2}$$

10.3. ábra: Háromszög kerülete és területe (Hérón-képlet)

A számoláshoz szükséges képletek a 10.3. ábrán láthatóak. Mint az ábrán látható, a Hérón-képlethez, amely a háromszög oldalaiból tudja kiszámolni annak területét, szükség lesz a négyzetgyökvonást tartalmazó *math* modulra, így annak beimportálását a függvények definíciója előtt el kell helyezni:

```
6  import math
```

Ezután következhet a kerület kiszámítását végző függvény, ami a 10.4. ábrán látható.

```
9  def kerulet(a: float, b: float, c:float) -> float:
10     """
11     A függvény a háromszög kerületét számolja ki.
12     :param a: A háromszög egyik oldalának hossza.
13     :param b: A háromszög másik oldalának hossza.
14     :param c:: A háromszög harmadik oldalának hossza.
15     :return: A háromszög kerülete, ha megszerkeszthető a háromszög, különben None.
16     """
17     if a <= 0 or b <= 0 or c <= 0:
18         return
19     if a <= b + c or b <= a + c or c <= a + b:
20         return
21     return a + b + c
22
```

10.4. ábra: A kerulet() függvény

A 9. sorban a már megismert módon kezdődik a függvénydefiníció fejléce. Ezután a 10–16. sorban a függvény dokumentációja látható – amit a VS Code valamiért nem hajlandó megjegyzésként színezni a képen.

A 17–20. sorok tesztelik a háromszög-egyenlőtlenség teljesülését. (A három oldalnak határozottan pozitív értékűnek kell lennie úgy, hogy bármely két oldal összegének nagyobbnak kell lennie a harmadik oldal hosszánál.) Amennyiben bármelyik feltétel nem teljesül, akkor a függvény érték visszaadása nélkül kilép.

Ezután egyetlen teendő marad: kiszámolni a kerületet és visszaadni azt. Ez egyetlen utasítással megtörténik a 21. sorban.

Ezután a terület kiszámítása következik. Azonban ne feledjük, hogy a Hérón-képletben szerepel a kerület fele. Vagyis ebben a függvényben is ki kellene számolni a kerületet. Ha ehelyett meghívjuk a **kerulet()** függvényt, akkor az elvégzi a háromszög-egyenlőtlenség teljesülésére vonatkozó vizsgálatot is, így annak a kódját sem kell újra leírni. Így ha esetleg a vizsgálatot elírjuk – mint én a mintaprogram első változatánál –, akkor csak egy helyen kell azt javítani.

A t**erulet()** függvény kódját mutatja a 10.5. ábra. Ennek első utasítása, a 32. sorban elintézi a **kerulet()** függvény meghívását. Amire itt figyelni kell, hogy nem lehet a kerület eredményét egy *kerulet* nevű változóban eltárolni! Ennek az az oka, hogy ezen a néven már létezik egy függvény – történetesen az, amit meg kell hívni. Ha a változó és a függvény neve megegyezne, abból később bemutatandó problémák adódhatnának.

```
23
24  def terulet(a: float, b: float, c:float) -> float:
25      """
26      A függvény a háromszög területét számolja ki.
27      :param a: A háromszög egyik oldalának hossza.
28      :param b: A háromszög másik oldalának hossza.
29      :param c:: A háromszög harmadik oldalának hossza.
30      :return: A háromszög területe, ha megszerkeszthető a háromszög, különben None.
31      """
32      K = kerulet(a, b, c)
33      if K is None:
34          return
35      s = K / 2
36      return math.sqrt(s * (s-a) * (s-b) * (s-c))
37
```

10.5. ábra: A terulet() függvény

A következő **if** utasítás megvizsgálja, hogy a **kerulet()** függvénynek sikerült-e a kerület kiszámítása, ha nem, akkor a függvény kilép.

Ezután a kerület felének kiszámítása, majd a terület értékének kiszámítása és visszaadása van már csak hátra.

A fenti feladatban szereplő függvények helyett egy függvényt is lehetne készíteni, amely kiszámolja egyszerre a háromszög területét és kerületét is, majd mindkét értéket visszaadja eredményül. Ehhez egyetlen akadály, hogy egy függvény csak egy értéket tud visszaadni. Ám ez az érték bármilyen típusú lehet.

Az ilyen esetekre, amikor több értéket kell ott szerepeltetni, ahol csak egy szerepel, a Python ismeri a **tuple** adattípust. Ez valójában különböző értékek felsorolása egyetlen elemként. Amennyiben egyértelműsíteni kell, akkor a *tuple* részét képező értékeket egy kerek zárójelben fogjuk össze:

```
>>> a = ("alma", 2)
>>> type(a)
<class 'tuple'>
>>>
```

10.6. ábra: A tuple típus

Az `(1, 42, False)` egy értékhármas, azaz három elemű *tuple* (*triple*).

A **return** utasítás kifejezése is lehet ilyen *tuple* – a mi esetünkben két érték, azaz értékpár. Ráadásul egy ilyen *tuple* segítségével egyszerre több változónak is értéket lehet adni:

elso, masodik = (42, True)

Ennél az értékadásnál valójában mindkét oldalon egy-egy kételemű *tuple* található. A baloldalon nem szerepel a zárójel, csak a két változó listája, a jobboldalon szerepel a zárójel. De a jobboldalon is elhagyható lenne. A fenti értékadásban az első érték az első változóba, a második érték a második változóba kerül.

Tehát ha a függvény két értéket ad vissza, akkor eleve két változónak is értékül adható a függvény értéke. Ebben az esetben a változók rendre megkapják az értékeket. De lehetséges az is, hogy egy változó kapja értékül a függvény értékeit, amely ekkor *tuple* típusú lesz. Ekkor a később a listáknál megismert indexeléssel lehet elérni az értékeket. (A következő oldalon levő példából megtudod, hogy hogyan is működik ez.)

A 10.7. ábra mutatja a **kerulet_terulet()** függvény kódját, amely egyben kiszámolja a paraméterében megadott adatokból az azokkal szerkeszthető háromszög kerületét és területét is.

```
38  def kerulet_terulet(a: float, b: float, c:float) -> float:
39      """
40      A függvény a háromszög kerületét és területét számolja ki.
41      :param a: A háromszög egyik oldalának hossza.
42      :param b: A háromszög másik oldalának hossza.
43      :param c:: A háromszög harmadik oldalának hossza.
44      :return: A háromszög kerületé és területét tartalmazó tuple,
45      ha megszerkeszthető a háromszög, különben None.
46      """
47      if a <= 0 or b <= 0 or c <= 0:
48          return
49      if a <= b + c or b <= a + c or c <= a + b:
50          return
51      K = a + b + c
52      s = K / 2
53      T = math.sqrt(s * (s-a) * (s-b) * (s-c))
54      return (K, T)
55
```

10.7. ábra: A kerulet_terulet() függvény

Ebben a függvényben ismét szerepel a háromszög-egyenlőtlenség ellenőrzése, mivel nem akarjuk a kerület kiszámítását a **kerulet()** függvénnyel végeztetni. Egyrészt lehet, hogy ebben az esetben nincsenek definiálva a külön-külön számításokat végző függvények. Másrészt az már túl sok függvényhívást eredményezne a kiszámolandó értékekhez képest, ha ez a függvény meghívná külön-külön a másik két függvényt.

Az újdonság az 54. sorban van, de ennek magyarázata már szerepelt az előző bekezdésekben.

A függvény által visszaadott értéket a fentebb írtak alapján lehetne a 10.8. ábra 57. sorában látható módon, eleve két változóban eltárolni. Azonban ez programhibát fog okozni minden olyen esetben, ha a függvény paramétereivel nem lehet háromszöget szerkeszteni!

```
57  k, t = kerulet_terulet(3, 4, 5)
58  print(t)
59  print(k)
60
61  eredmeny = kerulet_terulet(3, 5, 0)
62  if eredmeny is None:
63      print("Nem szerkeszthető háromszög!")
64  else:
65      print("kerület:", eredmeny[0])
66      print("terület:", eredmeny[1])
67
```

10.8. ábra: A tuple adattípusú érték felhasználási lehetőségei

Ebben az esetben ugyanis a függvény nem egy értékpárt, hanem a `None` értéket adja vissza, amit viszont nem lehet egyszerre két változóba elhelyezni.

Az 57. sorban látható megoldás csak akkor használható, ha pontosan annyi értéket fog adni a függvény, ahány változóba kell beletenni. **Sem több, sem kevesebb érték nem lehet!**

A mi esetünkben a lehetséges hibajelzés miatt a 61–66. sorban látható megoldást kell alkalmazni. Ekkor egy *eredmeny* nevű változóba kerül a *tuple* érték. Ennek első, illetve második elemét a 65. és a 66. sorban látható módon lehet elérni.

```
>>> type((42))
<class 'int'>
>>> type((42,))
<class 'tuple'>
```

10.9. ábra

A *tuple* típusra visszatérve érdemes még megjegyezni, hogy a típus lehet egyelemű is. Ahogyan az a 10.9. ábrán látható, az – ekkor kötelező – zárójelen belül az egy érték után egy vesszőt kell tenni. Enélkül a vessző nélkül a kifejezés egy zárójelbe tett szám lenne, ami a példa esetében *int* érték lenne.

10.1. Készíts függvény, amely paraméterként megkapja a derékszögű háromszög két befogójának hosszát (valós számként) és visszaadja az átfogó hosszát! Amennyiben a kapott két paraméter nem pozitív szám, akkor a függvény adja vissza a None értéket!

10.2. Készíts függvényt, ami az elágazásoknál szerepelt feladathoz hasonlóan a paraméterül kapott egész szám alapján könyvet ajánl. A függvény neve `konyvajanlo` legyen. Az alábbiak szerint alakuljon a függvény eredménye:

1. 0–3: „Totyogóknak a kettes számrendszerről”
2. 4–6: „Hackeljük meg az óvodát!”
3. 7–14: „Felhőtechnológia a menzán”
4. 15–18: „Big data a középiskolában”
5. 19–: „Donald Knuth: A számítógép-programozás művészete”

10.3. Készíts függvényt, amely egy egész számról eldönti, hogy egy másik egész számmal eloszthatóe!

A függvény első paramétere a vizsgált szám legyen, a második paramétere pedig az a szám, amellyel való oszthatóságot kell vizsgálni. Például ha azt akarjuk vizsgálni, hogy a 15 osztható-e néggyel, akkor az `oszthato(15, 4)` függvényhívásra legyen szükség.

A függvény False értéket adjon vissza, ha nem teljesül az oszthatóság, True értéket, ha teljesül.

10.4. Készíts függvényt, amely a bemenő paramétereiben óra, perc, másodperc alakban megadott időadatról megmondja, hogy az összesen hány másodperc!

10.5. Készíts függvényt, amely a paraméterében megadott másodperceket felbontja óra, perc, másodperc alakba és ezt egy számhármasként adja vissza!

10.3. Opcionális paraméterek

Előfordulhat, hogy úgy szeretnénk definiálni egy függvényt, hogy annak bizonyos paramétereit ne legyen kötelező megadni. Erre a legjobb példa a **print()** függvény, amelynek van két ilyen paramétere, az *end* és a *sep*. Az ilyen paramétereknek mindig van egy alapértelmezett értéke arra az esetre, ha a függvény meghívásakor nem szerepelnek. Megnézve a **print()** definícióját, látható, hogy ezeket hogyan kell megadni. A paraméter neve után egy egyenlőségjel után meg kell adni az alapértelmezett értékét is. Az ilyen paramétereknek a paraméterlista végén, az alapértelmezett értékkel nem rendelkező paraméterek után kell szerepelniük.

A függvény meghívása során ezek a paraméterek bármilyen sorrendben szerepelhetnek, ugyanis ahhoz, hogy az értelmező tudja, melyik kifejezés melyik paraméter értéke lesz, a hívásnál meg kell adni a paraméter nevét és egy egyenlőségjelet a kifejezés előtt.

Igazából az alapértelmezett értékkel nem rendelkező paraméterek neve is megadható ilyen módon, így nem kell azokat sem a definíciónál megadott sorrendben megadni ezzel a módszerrel élve.

A függvény definiálásánál az alapértelmezett értékkel nem rendelkező paramétereket kell először megadni. Ezután jöhetnek olyan paraméterek, amelyeknek alapértelmezett értéke is van. Ezt az alapértelmezett értéket úgy kell megadni, hogy a paraméter neve után egy egyenlőségjelet követően szerepel az alapértelmezett érték. Ez valójában olyan, mintha egy értékadás lenne, amely így a függvény fejlécében értéket ad a paraméternek, amelyet akkor kap, ha a függvény hívása nem adott a paraméternek más értéket.

10.4. Változók láthatósága

A függvények kapcsán felmerül a kérdés, hogy egy adott nevű változó (vagy általában valamilyen objektum) a program melyik részében érhető el. Ebből a szempontból megkülönböztetünk *globális* és *lokális* változókat.

A függvény paraméterei valamint azok a változók, amelyek a függvénydefiníció blokkjában kapnak értéket, **lokális változók**. Ezek csak az adott függvény kódjának futása során léteznek, csak ott érhetőek el.

Azok a változók, amelyek minden függvényen kívül jöttek létre, a **globális változók**. Ezek elvileg mindenhol, még a függvények kódján belül is láthatóak, értékük elérhető.

Amikor a program elindul, létrejön egy *globális adatterület*. Minden objektum – változó, függvény stb. – ami a függvényeken kívül van létrehozva, ebbe a globális adatterületbe kerül. Ez a globális adatterület a program befejezéséig létezik, majd ekkor megszűnik. Ha konkrétan nem törlünk a programban egy változót a **del** utasítással, akkor a globális adatterület létezéséig az létezni fog.

> Nem érdemes nagyon elmélyedni abba, hogy hogyan történik egy függvény tényleges meghívása, legalábbis a rekurzióról szóló fejezet előtt. Azt azonban érdemes megjegyezni, hogy minden függvényhívás a memória egy *verem* nevű területén egy memóriaterület lefoglalását igényli. Ebben a memóriablokkban kerülnek a függvény paraméterei és lokális változói is elhelyezésre. Mivel ez a blokk a függvény befejezése után felszabadul, a lokális változók a függvény befejezésével fizikailag is megszűnnek létezni.

Ezzel ellentétben minden függvényhívás létrehoz egy *lokális adatterületet* – lásd a fenti megjegyzést –, amely a függvény befejezéséig létezik. A függvényhez tartozó változók és a függvény paraméterei abba lokális adatterületbe kerülnek, és az adatterület megszűnésével együtt megsemmisülnek.

Egyszerre több lokális adatterület is lehet: amennyiben az egyik függvény egy másikat hív, akkor a belső függvényhívás is új lokális adatterületet hoz létre.

1. A függvényeken kívüli kódban nem lehet használni egyetlen lokális változót sem, hiszen azok ott nem léteznek.
2. Azonban egy függvény kódjában a globális változókhoz is hozzá lehet férni.
3. Egy függvény kódja más lokális adatterületen levő változókhoz nem fér hozzá, csak a sajátjához (és a globális változókhoz).
4. Különböző adatterületeken lehet ugyanolyan nevű változót létrehozni.

A fenti pontok közül az utolsónak lehetnek nehezen felderíthető szemantikai hibát eredményező következményei!

Mivel a függvények a külvilággal csak a paraméterein és a visszatérési értékein keresztül tudnak kommunikálni – hiszen a lokális változói kívül nem elérhetőek – így kiszűrhetők az olyan rejté-

lyesnek tűnő problémák, hogy egy változó értéke minden látható ok miatt megváltozik a program futása során.

Amennyiben rászokunk arra, hogy egy függvényen belül nem használunk globális változót, csak a saját változókat, akkor még abból sem adódhat baj, ha valamelyik lokális változó neve megegyezik egy globális változóéval. Az ilyen esetekben ugyanis – bár alapból a globális változók láthatóak lennének a függvényen belül – az azonos nevű globális változót a lokális változó el fogja fedni.

Bár a globális változók látszanak a függvényen belül, azokat alapesetben nem lehet módosítani, mivel egy értékadás a globális változóval azonos nevű lokális változót hoz létre. Vannak azonban olyan esetek, amikor ez konkrétan szükséges. Például amikor egy globális változót arra használ a program, hogy egy függvény által elvégzett művelet kapcsán megszámolja, hogy az hányszor hajtódik végre (vagy lesznek még később egyéb példák is ilyen esetre). Ekkor ugyanis egy olyan értékről van szó, amit a függvény többszöri meghívásai között meg kellene őrizni.

Erre használatos a **global** utasítás, amely után annak a változónak a nevét kell megadni, amely ugyan globális, de a lokális változókhoz hasonlóan módosítania kell az értékét a függvénynek.

```
def spam():
    global eggs
    eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

A fenti programkód a „spam” szöveget fogja kiírni, mert a függvény első utasítása engedélyezi az eggs globális változó értékének módosítását a függvény további része számára.

A **global** utasítás nélkül a következő történne:

1. A függvényen belül létrejönne az *eggs* lokális változó „spam” értékkel.
2. A függvény befejezésekor megszűnik a lokális *eggs* változó.
3. Innentől ismét a globális változó válik elérhetővé, amelynek értéke „global”, amit a **print()** ki is ír.

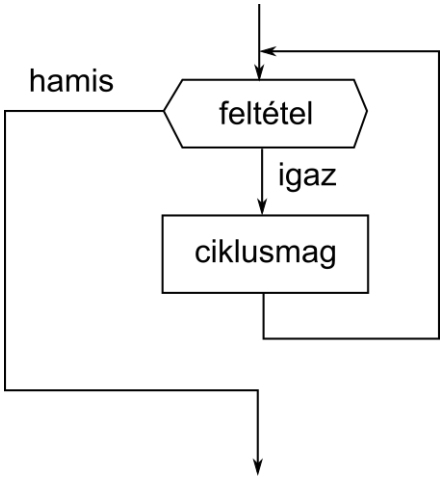
11. Ciklusok

A harmadik vezérlési szerkezet, amellyel megismerkedünk, a **ciklus**. A ciklus mindig valamilyen művelet ismételt végrehajtását jelenti. Más néven szokás *ismétlés* vagy *iteráció* néven is emlegetni. Utóbbi az angol szakkifejezéshez áll közel: *iteration*. Angolul szokták egyszerűen *loop*nak is nevezni.

Minden ciklusban van valamilyen feltételvizsgálat és van egy művelet(sor) – azaz blokk –, amelynek ismételt végrehajtása ettől a feltételvizsgálattól függ. A műveletsort, amelyet ismételten kell újra és újra végrehajtani, **ciklusmag**nak szokás nevezni. A következőkben a különböző ciklusokat vesszük sorra. Minden változatra jellemző, hogy a ciklusmag fog a feltételtől függően valahányszor végrehajtódni, majd ezt követően a szekvenciális vezérlésnek megfelelő következő utasítással folytatódik a végrehajtás.

11.1. Elöltesztelő ciklus

Az 11.1. ábra mutatja az elöltesztelő ciklus folyamatábráját. Mint látható, a ciklushoz érve egy feltételvizsgálat történik, majd ha ez a feltétel igaznak bizonyul, akkor a ciklusmag végrehajtása következik. Ezután megtörténik a feltétel újabb vizsgálata.



11.1. ábra: Elöltesztelő ciklus

A ciklusmag mindaddig fog végrehajtódni, amíg a ciklus elején levő feltétel igaz. Két fontos dolgot kell ezzel kapcsolatban megjegyezni:

1. A feltételnek nem kell a ciklusmag végrehajtása közben mindvégig igaznak lennie, csak akkor, amikor a végrehajtás éppen a feltétel vizsgálatánál tart. Amennyiben a ciklusmag végrehatása elkezdődik, akkor teljesen megtörténik annak végrehajtása, még akkor is, ha menet közben valamikor a feltétel hamis lenne.
2. Mivel először a feltétel vizsgálatára kerül sor, semmi garancia nincs rá, hogy a ciklusmag egyáltalán végrehajtódik, hiszen ha a feltétel már kezdetben is hamis, akkor a ciklusmag egyszer sem hajtódik végre. (Ám mindaddig újra és újra végrehajtódik, ameddig a feltétel a megvizsgálásakor igaz.) Ebből következően nem szabad a program további részében biztosra venni, hogy a ciklusmagban előírt tevékenység végrehajtódott. Például ha a ciklusmagban kap egy változó először értéket, akkor előfordulhat, hogy a ciklust követően a változóra való hivatkozás hibát okoz, mert ha a ciklusmag egyszer sem hajtódott végre, akkor a változó nem létezik!

Magyar nyelvű mondatszerű leírással a ciklus így néz ki:

Amíg a *feltétel* igaz, **csináld**:
  Ciklusmag utasításai
Ciklus vége

Ugyanez angolul:

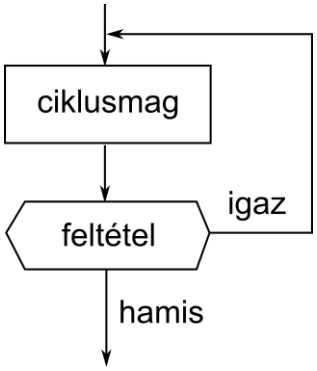
While *feltétel*, **do**:
  Ciklusmag utasításai
End of While

Amennyiben a feltétel vizsgálata hamis eredményt ad, a ciklus véget ér és a program a következő utasítással folytatódik.

A legtöbb programozási nyelvben, így a Pythonban is a **while** utasítás valósítja meg az elöltesztelő ciklust.

> A Python az összes létező ciklusfajtából összesen kétfélét ismer. Ezek egyike az elöltesztelő ciklus, amelyet a **while** utasítás valósít meg. A másik a **for** ciklus, amelyről később lesz szó.
>
> Ám ez egyáltalán nem jelent semmilyen korlátozást, ugyanis az egyes ciklusokat mind át lehet alakítani elöltesztelő ciklussá. Így tehát a Python két ciklus utasítása bőven elég bármilyen ismétléses vezérlési szerkezet megvalósítására. Azáltal, hogy csak az elöltesztelő ciklus érhető el, kevesebb megtanulandó utasítás van a nyelvben, vagyis a nyelv egyszerűbb.

11.2. Hátultesztelő ciklus



11.2. ábra: Hátultesztelő ciklus

A hátultesztelő ciklus folyamatábrája a 11.2. ábrán látható. Mint látható, először a ciklusmag végrehajtása történik meg, majd csak ezután kerül sor a ciklusfeltétel vizsgálatára. Az elöltesztelő ciklussal ellentétben tehát a hátultesztelő cikus használata esetén a ciklusmag legalább egyszer végrehajtásra kerül. Olyankor használatos, amikor a feltétel vizsgálata előtt a ciklusmagban szereplő műveletek valamelyikére szükség van.

Az egyes programnyelvekben eltérés lehet abban, hogy a ciklusmag újbóli végrehajtásához a feltételnek igaznak vagy hamisnak kell lennie. Pascalban a ciklusból kilépés történik igaz esetben, C-ben a ciklusban maradás. Mivel a Pythonban erre nincs utasítás, ez számunkra nem annyira lényeges.

Mondatszerű leírásban így néz ki a ciklus:

Csináld:
Ciklusmag utasításai
Amíg a *feltétel* igaz (bennmaradási feltétellel)

Vagy:

Csináld:
Ciklusmag utasításai
Mígnem a *feltétel* igazzá válik (kilépési feltétellel)

Ugyanez angolul:

Do:
Ciklusmag utasításai
While *feltétel*

vagy

Do:
Ciklusmag utasításai
Until *feltétel*

Ismét a második esetben lép ki a ciklus, ha a feltétel igaz.

Amennyiben elöltesztelő ciklussal akarjuk megvalósítani, akkor értelemszerűen gondoskodni kell arról, hogy a ciklusfeltétel már az első alkalommal igaz legyen...

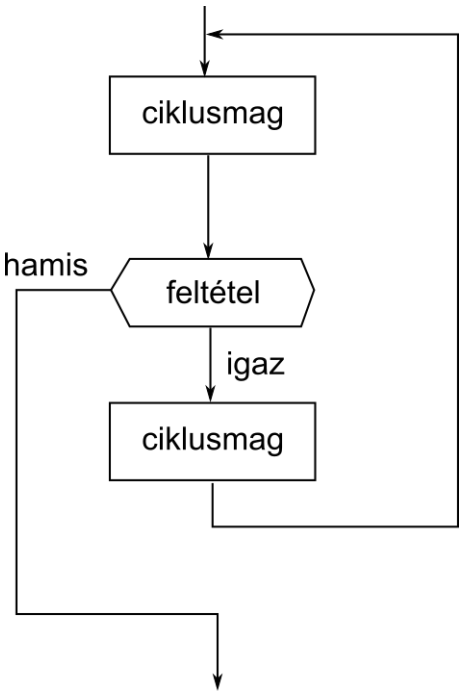
11.3. Tesztelés a ciklusmag belsejében

Bár hivatalosan ilyen ciklusfajta nem létezik, a gyakorlatban nagyon sokszor előfordul, hogy a ciklusmag belsejében kell eldönteni, hogy kell-e folytatni a ciklus végrehajtását. A 11.3. ábra ezt az esetet szemlélteti.

Az elöltesztelő ciklussal ezt kétféleképpen lehet megvalósítani.

Az egyik megoldásnál a feltétel vizsgálatánál egy elágazás szerepel, amelynek egyik ágában szerepel a ciklusmag további része, a másik ág vagy üres, vagy a ciklus elején szereplő feltétel hamisra állításáról gondoskodik. Sokáig ez számított a helyes megoldásnak, azonban a program olvashatóságát a sok egymásba ágyazás nehezíti, így a gyakorlatban inkább a másik megoldás használatos.

Ez a másik megoldás is használ elágazást, azonban a hozzá tartozó kódrészlet nagyon rövid. Mindössze azt az utasítást tartalmazza, amellyel az adott programozási nyelv lehetővé teszi a ciklusból való kiugrást (ez a Pythonban a később bemutatatott **break** utasítás). Ebben az esetben a ciklusmag további részét nem kell beágyazni az elágazásba. Ilyenkor a legtöbb esetben a ciklus elején szereplő feltételnek az igaz (True) értéket adják meg, ami látszólag végtelen ciklust eredményez.



11.3. ábra: Tesztelés a ciklusmag belsejében

Amennyiben a ciklus soha nem ér véget, akkor beszélünk végtelen ciklusról. Ez általában szemantikai hibának számít. A fenti eset, amikor az elöltesztelő ciklus belépési feltétele van konstans igaz értékre állítva, nem tekinthető végtelen ciklusnak, hiszen a ciklusmagban van egy feltételvizsgálat, amelynek eredménye alapján a ciklus véget érhet.

A tényleges végtelen ciklus az, amikor a kilépéshez szükséges feltétel sohasem teljesül, ezért a ciklus ténylegesen nem ér véget soha.

Időnként előfordul olyan helyzet is, amikor szándékosan készítünk végtelen ciklust. Például a mikrovezérlők a feladatukat végtelen ciklusban hajtják végre. De ugyanígy maga a CPU is végtelen ciklusban van: a ciklusmag egy utasítás végrehajtását jelenti, amely a gép bekapcsolásától a kikapcsolásáig ismétlődik.

11.4. ábra: Végtelen ciklus

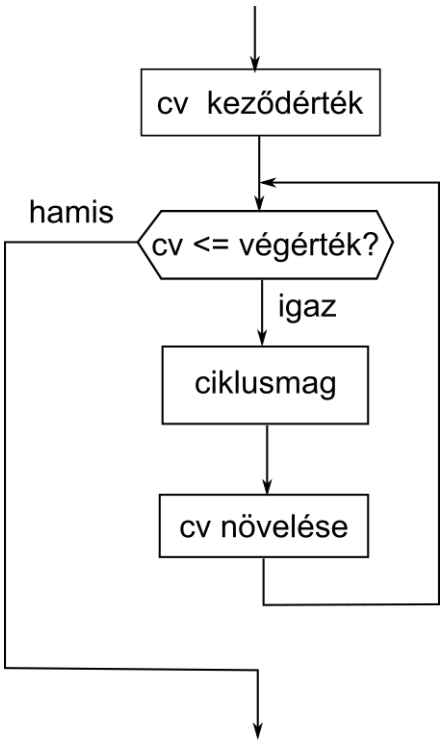
11.4. Adott elemeken végighaladó ciklus

Látszólag nincs feltétele a következő két ciklusfajtának. Az első az **adott elemeken végighaladó ciklus**. A későbbiekben több olyan adattípussal is megismerkedsz, amelyek elemein ezzel a ciklussal végig lehet lépegetni, minden elemre egyszer végrehajtva a ciklusmagot.

Az elemek sorrendje itt sem teljesen mindegy. A Python ezt úgy oldja meg, hogy definiál egy objektum-kategóriát **iterable** néven. Az ebbe a kategóriába tartozó adattípusok elemein egy adott sorrendben végig lehet menni ezzel a ciklusfajtával.

A ciklus feltételének tekinthető ilyenkor, hogy van-e még olyan elem, amellyel a ciklusmag nem futott le. Az éppen használt elem a ciklusmagban egy ciklusváltozón keresztül elérhető, de nem módosítható.

11.5. Számlálós vagy ciklusváltozós ciklus



11.5. ábra: Számlálós ciklus

Az adott elemeken végighaladó ciklus egy aleseteként is tekinthetjük azt a ciklust, amely esetében egy ciklusváltozó halad végig egy intervallumon. A 11.5. ábra a ciklus megvalósítását mutatja elöltesztelő ciklus segítségével.

Az ábrán *cv* jelöli a ciklusváltozót, amelyet egy számlálóként lehet használni a ciklusmagban. Egyes programozási nyelvekben ez a számláló elérhető a ciklusmagban, más nyelvekben nem. Egyes nyelvekben kötelezően nullától indul és egyesével növekszik, más nyelvekben a kezdőérték is szabadon megadható és akár csökkenhet is. Sőt, például a Pascal nyelvben akár karakter típusú is lehetett a ciklusváltozó, míg általában az egész típusra van korlátozva a típus.

Mivel a cikus az elöltesztelő ciklussal is megvalósítható, a fenti korlátozások a gyakorlatban semmit nem jelentenek, hiszen ha az adott nyelvi megvalósítás nem teszi lehetővé azt a megoldást, amit használni szeretnénk, egészen egyszerűen az elöltesztelő ciklust használhatjuk helyette. Éppen ezért a C-nyelv például nagyon lazán használható megvalósítást biztosít, míg a Pythonban nincs megvalósítása.

11.6. Ellenőrzött adatbevitel

Ciklus segítségével az adatbevitel ellenőrzésére is van lehetőség. Amennyiben az adatbevolvasó utasítást egy ciklus belsejébe helyezzük el, akkor a ciklus feltételében a beolvasott értéket lehet ellenőrizni. Ilyen módon amennyiben a felhasználó nem a megfelelő tartományba eső értéket írt be, akkor az adatbekérés megismétlődik.

Erre tipikusan egy hátultesztelő ciklus volna alkalmas, hiszen a feltételvizsgálatnak azután van értelme, hogy megtörtént a beolvasás. Azonban egy elöltesztelő ciklus is képes erre, ha előbb a változóba, amelynek a beolvasással értéket adnánk, egy olyan értéket adunk, amely nem fogadható el. Ezzel a ciklus belépési feltétele teljesül és megtörténik az adatbeolvasás első alkalommal is. Ezután már rendesen működik az ellenőrzés.

Másik megoldás, ha a ciklusmagban először megtörténik a beolvasás, majd a kilépés feltételének vizsgálata. Ennek a megoldásnak az előnye az előzővel szemben, hogy lehetőség van feltétel vizsgálat után a felhasználót tájékoztatni az elkövetett hibáról az adatbekérés megismétlése előtt.

11.1. Készítsetek közösen programot, amely kitalálja a felhasználó által gondolt számot! A feladat alkalmazza az intervallum-felezés módszerét! A felhasználónak az [1; 1024] zárt intervallumból kell egy egész számot választania, amit a számítógépnek kell kitalálnia.

1. Csoportokban dolgozva tervezzétek meg folyamatábrán vagy mondatszerű leírás formájában a program algoritmusát!
2. Ezután készítsétek el a programot, alkalmazva benne a lentebb szereplő információkat is a ciklus elkészítéséhez!

Az intervallum-felezés módszerével valószínűleg már találkoztál. Minden olyan esetben alkalmazható egy elem megkeresésére, ha tudjuk, hogy az egy valamilyen módon rendezett sorozat eleme (ha szerepel egyáltalán).

A módszer lényege, hogy először megnézzük az aktuális intervallum középső elemét. Ha ez nagyobb a keresett elemnél, akkor az intervallum alsó részében van; ha kisebb, akkor a felső részében az általunk keresett elem. Ha szerencsénk van, akkor éppen megtaláltuk az elemet.

Ezután az intervallumnak azt a felét, amelyben már biztosan nem lehet az elem, figyelmen kívül lehet hagyni. Az az intervallum, amelyben a keresést folytathatjuk, fele annyi elemet tartalmaz, mint eddig, ezért hívják a módszert **intervallum-felezésnek**.

Matematikai módszerekkel könnyen belátható, hogy amennyiben n elemben kell egy adott elemet megkeresni, akkor ezzel a módszerrel összesen $\log_2 n$ elemet kell megvizsgálni. Fordítva gondolkodva: m lépésben összesen 2^m elem közül kereshető meg a megfelelő.

Tehát a fenti feladatban alkalmazva a módszert, 1 és 1024 közötti szám összesen $\log_2 1024 = 10$ kérdéssel megtalálható.

11.7. A while utasítás

Az elöltesztelő ciklust a while utasítás valósítja meg a Python nyelvben. Ez a következő elemekből áll:

1. A **while** kulcsszó nyitja az utasítást;
2. ezt követi a ciklus *feltétele*;
3. majd egy kettőspont zárja az utasítás első sorát
4. A következő, bentebb kezdett *blokk* utasításai alkotják a ciklusmagot, amit mindaddig újra és újra végrehajt a számítógép, amíg a 2. pontnál megadott feltétel igaz értéket ad.

Először a while szó utáni logikai kifejezés értékének meghatározása történik. Ha a kifejezés értéke igaz, a ciklusmag végrehajtódik, majd a kifejezés értéke újra meghatározásra kerül. Amennyiben – akár már az első alkalommal – a feltétel hamis, az utasítás befejeződik, a program a ciklusmagot megadó blokkot követő utasítással folytatódik.

Például vegyük a következő kódrészletet:

```
12      while valasz not in ['k', 'n', 't']:
13          print('k: kisebb')
14          print('n: nagyobb')
15          print('t: talált')
16          print('(Más választ nem fogadok el!)')
17          valasz = input().lower()    # a választ kisbetűsítjük, hogy könnyebb legyen bánni vele
18          valasz = valasz.lstrip()[0] # az első nem szóköz karakterét tartjuk csak meg
```

A 12. sorban kezdődik a **while** utasítás[7] és a ciklusmagját a 13–18. sorok alkotják.

A feltétel ebben az esetben egy olyan kifejezést tartalmaz, amelyről még később, a listák megismerésekor lesz részletesebben szó. A ciklusmag utolsó előtti utasítása egy adatbeolvasás. Az adatbeolvasást végző **input()** függvény végén a **.lower()**, valamint a következő sorban levő **.lstrip()** egy-egy szövegkezelő művelet, amelyekről szintén később lesz részletesen szó. A megjegyzések segítenek megérteni, vajon mit is csinálnak...

A Python azon adattípusai esetében, amelyek egynél több értéket tudnak tárolni (a már részben megismert karakterlánc mellett a később megismerendő lista, halmaz, szótár például ilyen), lehetőség van arra, hogy megvizsgáljuk, egy adott értékű elemmel rendelkezik-e.

Az `in` igaz logikai értéket ad, ha benne van, a `not in` pedig akkor, ha nincs benne.

A több elemű érték jelen példában egy három elemű lista konstans.

Minderről részletesebben a listáknál lesz majd szó.

A fenti példa a korábban említett ellenőrzött adatbevitelre egy példa. A **while** utasítás előtt ebben az esetben a *valasz* változónak egy el nem fogadható értéket kell adni azért, hogy a 12. sorban levő feltétel teljesüljön (a változó értéke nincs benne a 'k', 'n' és 't' szöveg alkotta listában, aminek hatására a ciklusmag végrehajtásra kerül. A továbbiakban a feltétel már a felhasználó válaszait vizsgálja és addig nem hagyja békén a felhasználót, amíg a megfelelő karaktert nem írja be válaszként (akár nagybetűs változatban is).

Lehetőség van arra is, hogy a ciklus elhagyásakor – beleértve azt az esetet is, ha már eleve hamis a feltétel és egyszer sem hajtódik végre a ciklusmag – még valamilyen tevékenységet megadjunk. Erre az **if** utasításhoz hasonló módon az `else:` ág megadása szolgál.

```
while feltétel:
     ciklusmag
else:
     utasítások, a feltétel hamissá válása esetén
```

Az `else:` után írt utasítások minden esetben végrehajtásra kerülnek, amikor a **while** utáni feltétel hamis értéket ad. Tehát akkor is, ha a ciklus egyszer sem került végrehajtásra, mert eleve hamis volt a feltétel, és akkor is, ha a ciklusmag valahány alkalommal már végrehajtódott és utána vált a feltétel hamissá.

11.7.1. A break utasítás

Amennyiben a ciklusmagon belül adódik olyan helyzet, hogy már nem kell tovább folytatni a ciklus végrehajtását, akkor használható a ciklus elhagyására a **break** utasítás. Ahol ez az utasítás szerepel egy ciklusmagban, ott azonnal befejeződik annak a ciklusnak a végrehajtása. Amennyiben ez a ciklus egy másik ciklus ciklusmagjában szerepel, akkor fontos: **csak a legbelső ciklus fejeződik be!**

7 Mint a bentebb kezdésből sejthető, már eleve egy másik utasításhoz tartozó blokkban található, ami történetesen egy másik while utasítás (a teljes programkódot a lecke egy későbbi részén lehet majd megtalálni arra az esetre, ha önállóan nem sikerülne a feladatot megoldani).

Amennyiben a ciklus a **break** utasítás hatására fejeződik be, akkor az esetleges `else:` rész nem kerül végrehajtásra, hiszen nem a ciklusfeltétel hamisnak bizonyulása okozta a ciklus befejezését.

Az alábbi egy példa a break utasítással megvalósított „középen tesztelő ciklusra”:

```
while True:
     szam = int(input("Kérek egy pozitív egész számot!"))
     if szam > 0:
          break
     print("Ez nem pozitív!")
```

Van olyan programozási nyelv, például a Java, ahol a break utasítással nem csak a legbelső ciklusból lehet kilépni, hanem megadható az utasításnál, hogy mennyi ciklusból lépjen ki. Ez a lehetőség Python esetében nem áll rendelkezésre, itt a break mindig csak a legbelső ciklust fejezi be.

11.7.2. A continue utasítás

Olyan eset is előfordulhat a ciklusmag belsejében, hogy ugyan még nem kell feltétlenül befejezni a ciklust, de az adott ciklusmag további részének végrehajtására nincs szükség.

A **continue** utasítás egy ciklusmag adott véggrehajtását fejezi be. Ennek hatására nyilván ismét megtörténik a ciklusfeltétel megvizsgálása, amely dönt a ciklus további sorsáról. Számlálós ciklusok esetében ennek hatására a következő értékkel folytatódik azonnal a ciklus végrehajtása, ha még van érték. Értelemszerűen, ha a ciklusfeltétel hamis vagy nincs több érték, amelyre a ciklusmagot végre kell hajtani, akkor a ciklus is véget ér, de ennek már nem a **continue** utasítás az oka. Így amennyiben a ciklus véget ér, az esetleges `else:` ág végrehajtása is megtörténik.

Fontos megérteni a különbséget: a **break** hatására véget ér a ciklus végrehajtása azonnal, míg a **continue** hatására csak a ciklusmag aktuális végrehajtása fejeződik be, de a ciklus végrehajtása folytatódik onnan, ahonnan a ciklusmag végrehajtása után kell folytatódnia. A **break** esetében az *else*: rész azért nem hajtódik végre, mert a **break** azonnal *véget vet* a ciklusnak, míg a **continue** a ciklusmag további részének végrehajtását mellőzve *folytatja* a ciklust. (*break* = befejezés, *continue* = folytatás).

11.2. Arról volt szó, hogy az elöltesztelő ciklussal bármelyik ciklusfajtát meg lehet valósítani. Hogyan valósítanád meg a 11.2. ábrán látható hátultesztelő ciklust elöltesztelő ciklust használva?

11.3. Írj programot, amely kiszámítja a háromszög területét és a kerületét a három oldalának ismeretében!

A program a három oldal méretét a felhasználótól kérje be olyan módon, hogy csak olyan adatokat fogadjon el, amelyekből lehet háromszöget szerkeszteni:

- mindhárom oldal hosszának pozitívnak kell lennie (lebegőpontos számokkal dolgozunk);
- a három oldal bármelyikének rövidebbnek kell lennie a másik két oldal összegénél (háromszög-egyenlőtlenség szabálya)

Amennyiben a felhasználó által megadott adatok a fentieknek nem felelnek meg, a program kérje be az oldalhosszakat újra!

11.4. Írj programot, amely addig olvas be számokat, amíg azok nem negatívak. Amennyiben a felhasználó negatív számot ír be, a számok beolvasása véget ér. Ekkor a program írja ki a beolvasott számok összegét és átlagát (egyikbe sem beleszámolva a negatív számot, ami az adatbevitel végét okozta).

11.5. Írj programot, amely egyetlen **print()** utasítást ciklusban többször meghívva kirajzolja az alábbi ábrát:

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

11.6. Írj függvényt, amely egymás alá kiírja a paraméterében kapott *n* egész számnál nem nagyobb prímszámokat!

11.7. Írj függvényt, amely a 11.5. feladat módosításaként kapható ábrát rajzolja ki:

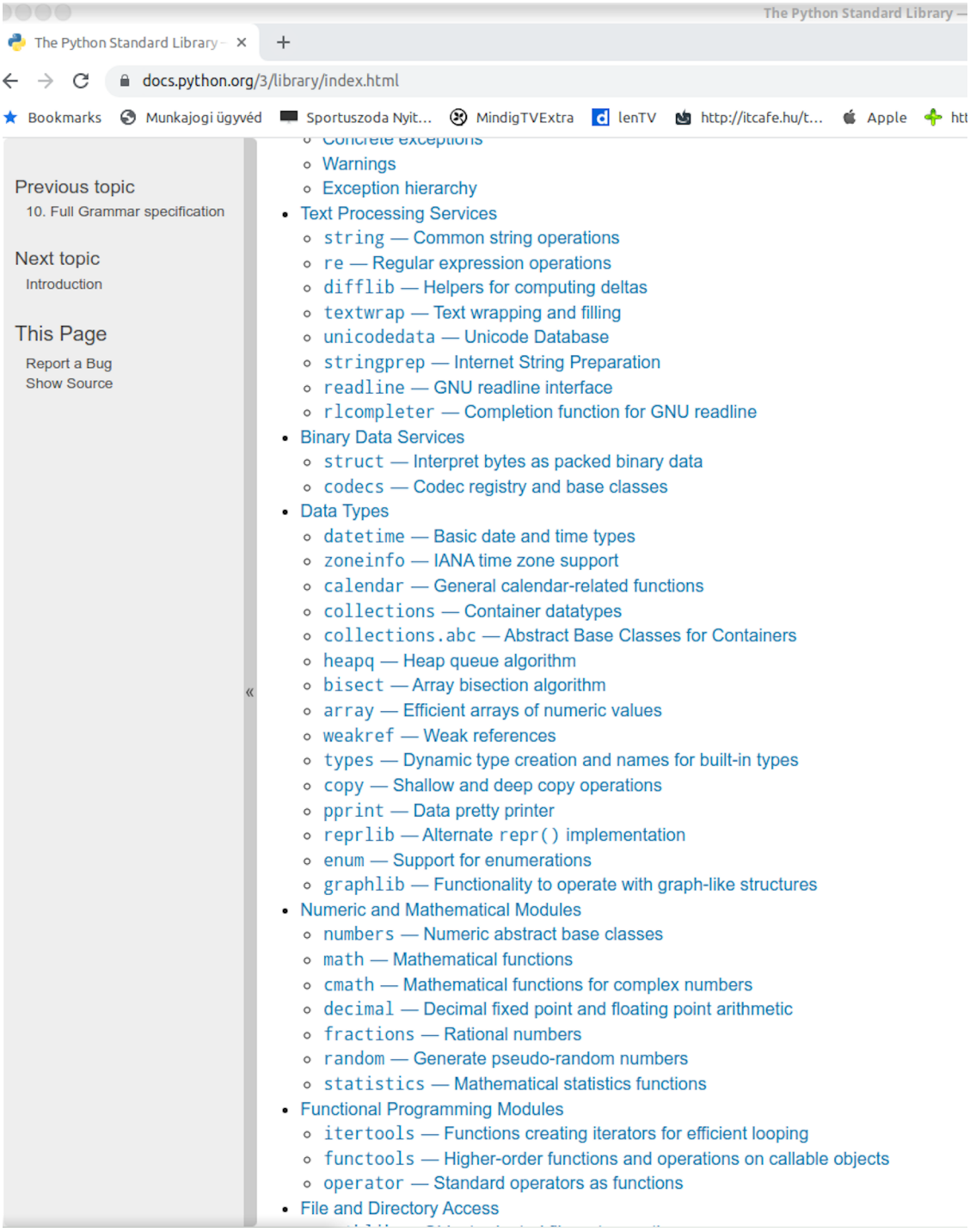
```
*   *
 * *
  **
 * *
*   *
```

A függvény paramétere határozza meg, hogy hány sorból fog állni a nagy kereszt: a paraméter a kereszt szárának elemszáma legyen, azaz hogy a középső sor előtt, illetve után hány sora legyen a keresztnek.

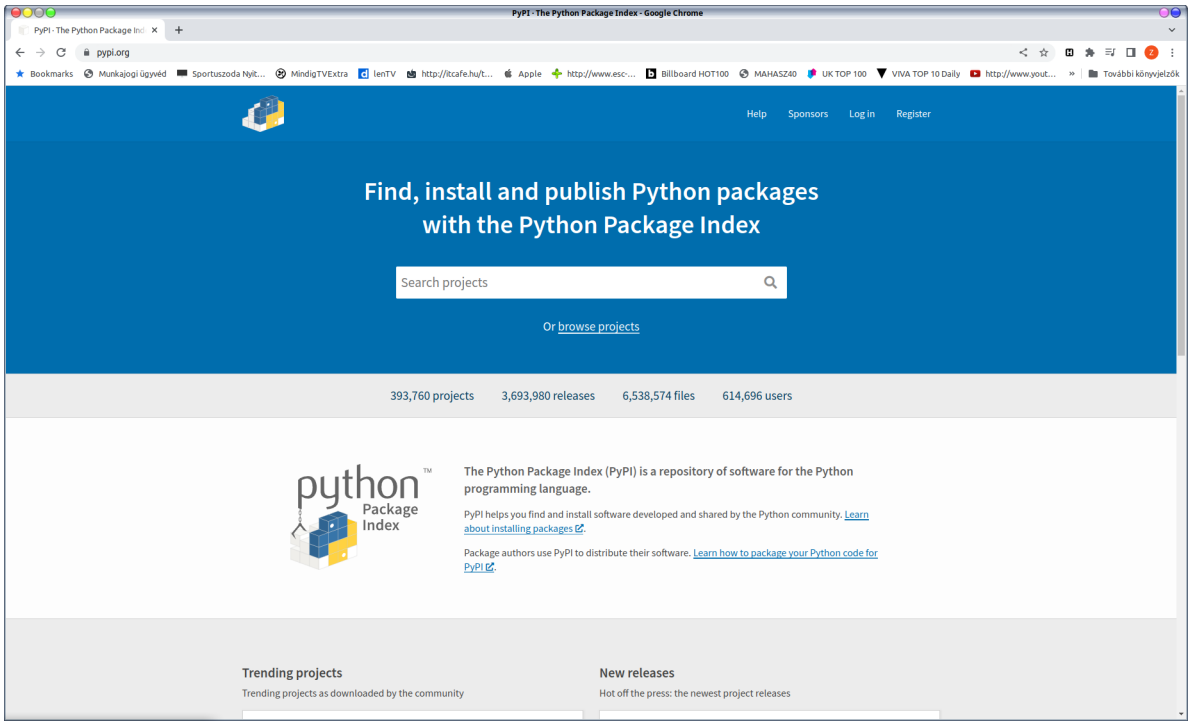
12. Modulok használata

A függvényekről szóló leckében már említett alulról felfelé fejlesztés során létrehozott függvénykönyvtárak létrehozását teszi lehetővé a Python nyelv esetében a modulok használata.

A 12.1. ábra egy részletet mutat a Python hivatalos dokumentációjából, ahol a külön telepítést nem igénylő modulok felhasználási terület szerint csoportosítva megtalálhatóak. Ezek az adott modul dokumentációját elég részletesen tartalmazzák angolul – az angol nyelv az informatikában kike-



12.1. ábra: A modulok téma szerinti listája a Python dokumentációjában



12.2. ábra: A pypi.org, a Python modulok hivatalos adatbázisának nyitóoldala

rülhetetlen. Néhány modullal ezek körül meg fogsz majd te is ismerkedni. Azonban nem minden modul szerepel ebben a dokumentációban sem a Python programokban használható modulok közül. Ezek csupán azok, amelyek egy jól feltelepített Python értelmező esetében biztosan rendelkezésre állnak.

Létezik azonban ezen felül egy hivatalos adatbázis (12.2. ábra), amelyből a Python által szabványos módon biztosított, további modulokat lehet feltelepíteni a számítógépünkre, amelyek onnantól kezdve szintén szabadon használhatóak – az adott számítógépen – egy programban.

Továbbá vannak olyan Python modulok is, amelyek nem a hivatalos adatbázison keresztül telepíthetőek, hanem más módon.

Végül van lehetőség arra is, hogy saját modulokat készítsünk, amelyeket a saját gépünkön található programok érnek csak el, vagy akár arra is lehetőség van, hogy csak egy adott programon belül legyenek elérhetőek.

Ez utóbbi lehetőséget nagyon egyszerű megvalósítani: a modul ugyanis nem más, mint egy speciális szerkezetű Python programállomány. Hogy miben speciális, arról hamarosan szó lesz.

12.1. Modulok alkalmazása

Ahhoz, hogy egy modult a programban használjunk, az **import** utasítással be kell azt importálni a programba. Az importálás tulajdonképpen úgy történik, hogy a modul tartalmát az értelmező végrehajtja az importálás helyén. Ez azt jelenti, hogy a modulban szereplő függvények az importálás helyétől tekinthetőek definiáltnak, ahogyan a modulban szereplő minden más objektum (pl. változók) is ettől kezdve léteznek. Ebből következően az importálásnak a program elején kell megtörténnie.

Bár elvi akadálya nincs, hogy ne így legyen, de alapvető szabály: **a modulok importálást rögtön a program bármely más utasítása elé kell elhelyezni**. Ez a gyakorlatban a program elején ál-

talában elhelyezett *shebang*[8] – esetleg az azt követő, az egész állomány tartalmára vonatkozó dokumentáló megjegyzés – után azonnal történik.

A 12.3. ábrán látható programrészlet egy példa arra, hogyan lehet a – később, a lecke vége felé bemutatásra kerülő – *math* modult beimportálni a programba. Az 1. sor a *shebang* néven is ismert sor, amely megadja, hol találja az operációs rendszer a Python értelmezőt, utána a jobb olvashatóság kedvéért kimarad egy sor, majd következik a 3. sorban a *math* modul beimportálása, amit illik legalább egy üres sorral megtoldani, hogy a program további részétől elkülönüljön. Az ábra az 5. sorban arra is példát mutat, hogyan lehet a modulban definiált **sqrt()** függvényt ezután használni.

```
1  #! /usr/bin/python3
2
3  import math
4
5  print(math.sqrt(4))
6
```

12.3. ábra

Mint az ábrán látható, az **import** utasítás legegyszerűbb formája, amikor egy `import` szó után megadjuk a beimportálni kívánt modul nevét (a példában ez a *math* modul). Ekkor az egész modul beimportálásra kerül olyan módon, hogy – a példában a 3. sorra érve – az értelmező a modult tartalmazó Python állományt végigolvassa és végrehajtja a benne található – modulok esetében jellemzően függvénydefiniáló – utasításokat. Az így definiált függvények és változók ettől a ponttól kezdve a program végéig úgy használhatók, mintha a program globális adatterében lettek volna definiálva. Azzal a kitétellel, hogy ahogyan a 12.3. ábra 5. sorában látható, a modulbeli függvény neve elé a modul nevét és egy pontot is ki kell tenni. Ugyanígy, ha a modulban definiált változót akarunk használni, az is így írandó. Ebből az írásmódból fogja tudni az értelmező, hogy az adott modul egy eleméről van szó.

Kicsit tudományosabban a fent leírtak így szólnának: az **import** utasítás hatására létrejön egy *névtér* (a példában *math* néven) és ebben kerül definiálásra a modul valamennyi publikus objektuma. A modul objektumainak elérésekor meg kell adni a modul névterét is a <modulnév>. előtag alkalmazásával, ahol a <modulnév> a modul importálásával létrejött névtér.

Van egy kényelmesebbnek tűnő megoldás is, amikor nem akarjuk a teljes modult beimportálni a programunkba, csupán a használni kívánt részét. Ez azonban a program saját globális objektumai és a modulból beimportált objektumok összekeveredését okozza, mivel a beimportált objektumok a program globális névterébe kerülnek, ezért nagyon körültekintően kell használni:

Az **import** utasítás hosszabb változata a `from` kulcsszóval kezdődik, amit a modul neve követ, majd jön az `import` kulcsszó után a modulból beimportálni kívánt objektum, vagy objektumok vesszővel elválasztott listája. Szélsőséges esetben az objektumok megnevezése helyett a `*` is használható.

A 12.3. és a 12.4. ábra végrehajtása ugyanazt eredményezi, de a 12.4. ábra esetén nem az egész *math* modul, hanem csak abból az **sqrt()** függvény kerül importálásra. Mint látható, ebben az utóbbi esetben nem kell alkalmazni a `math.` előtagot. Sőt, nem is szabad, mivel nem létezik a *math* névtér, az **sqrt()** függvény a program globális névterében található.

```
1  #/ /usr/bin/python3
2
3  from math import sqrt
4
5  print(sqrt(4))
6
```

12.4. ábra

Mitől veszélyesebb ez a megoldás, mint amikor az egész modult importáljuk? Azért, mert a program saját névterébe került a függvény, amit a program későbbi részében esetleg felüldefiniál egy ugyanilyen nevű függvény definíciója. Éppen ezért nem szabad – bár a nyelv lehetővé teszi – a `*` segítségével a modul minden objektumát a program névterébe beimportálni: nem lehet a program későbbi részében tudni, hogy egy modulból származó vagy egy a programban definiált függvény vagy változó szerepel-e a kódban (nem minden eleme van

8 A script nyelvekben az első sor adja meg, hogy pontosan melyik programnak kell az állomány tartalmát értelmezőként végrehajtania. Ezt az első sort nevezik *shebang*nak.

esetleg dokumentálva a modulnak). A helyzetet még inkább nehezíti, ha egy programban több modult is importálni kell. A különböző moduloknak lehetnek azonos nevű objektumai. Ha ezeket mind a `from` segítségével a program névterébe importáljuk, akkor azok egymással összeütközésbe fognak kerülni! Sokkal átláthatóbb megoldás, ha minden modul tartalma a saját névterébe kerül. Igaz, ilyenkor többet kell gépelni – főleg hosszú modulnevek esetében –, de a fejlesztőeszközök a névkiegészítés felkínálásával segítenek megspórolni az ilyen többletgépelések egy részét, cserébe viszont átláthatóbb programkódot kapunk.

Vannak esetek, amikor a modul importálása nem egyszerűen annak nevének megadásán keresztül lehetséges. Például a korábban már emlegetett Qt programkönyvtár annyira összetett, hogy a C++ könyvtár egyes elemeit elérhetővé tevő modulok egész hierarchiát alkotnak. Maga a Qt több változatban is elérhető Pythonban, ezeknek csupán az egyike – a dokumentum írásakor a legfrisebb változata – a *PySide6* modulcsomag. Csomagról van szó, azaz egy csomagon belül különböző modulok vannak, amelyeket szinte egyetlen programban sem kell együtt használni, hanem csak a csomag azon részét, amelyre éppen szükség van.

Az alábbi kép a PySide6 hivatalos dokumentációjából származik:

```
import PySide6.QtCore

# Prints PySide6 version
print(PySide6.__version__)

# Prints the Qt version used to compile PySide6
print(PySide6.QtCore.__version__)
```

Az első sorban látható a modul importálása, amely a *PySide6* csomag *QtCore* nevű modulját importálja be. Mint látható, a két név között egy pont helyezendő el ilyen esetben. A további sorok mutatják, hogy létrejött egy *PySide6* nevű névtér, azon belül pedig egy *QtCore* nevű névtér. A beimportált függvényeket és objektumokat a *PySide6.QtCore* névtérrel lehet elérni.

Ha külön nem akarjuk elérni a PySide6 névteret, akkor a fenti importálás helyett lehetett volna alkalmazni az alábbi utasítást is:

```
from PySide6 import QtCore
```

Ekkor csak egy *QtCore* névtér jön létre, amelyen keresztül ugyanúgy el lehet érni a modul valamennyi objektumát.

Valójában ez annak köszönhető, hogy a *PySide6* modulcsomag egy könyvtár, amelyben további könyvtárak, illetve az egyes modulokat tartalmazó állományok vannak. A könyvtárak és az azon belüli állományok elérésére szolgál a pont illetve a `from` megoldás használata. Amennyiben saját modulokat definiálunk egy program projektkönyvtárában, ott ugyanezt a megoldást lehet majd alkalmazni.

Végezetül jöjjön még egy példa a *PySide6* dokumentációjából arra az esetre, amikor több modult akarunk egyszerre beimportálni a csomagból, mindet önálló névtérbe:

```
import sys
import random
from PySide6 import QtCore, QtWidgets, QtGui
```

Itt az első két **import** utasítás egy-egy alapból elérhető Python modul (a *sys* és a *random*) importálását végzi. A harmadik utasítás a *PySide6* csomag három modulját, a *QtCore*, a *QtWidgets* és a *QtGui* modulokat importálja három különböző névtérbe. Így összesen öt névtér jön létre az öt importált modulra: *sys*, *random*, *QtCore*, *QtWidgets*, *QtGui*.

Az Qt programkönyvtárral ebben a dokumentumban nem foglalkozunk többet, lévén teljesen objektum-orientált megközelítést alkalmaz.

Amennyiben egy modul nevét túl hosszúnak találjuk, és szeretnénk elérni, hogy a beimportálásával létrejövő névtér neve rövidebb legyen, akkor az **import** utasítás kiegészíthető az **as** résszel, amely esetben a névtér az `as` szó utáni névvel jön létre. Például az előző oldali példában a *QtCore* név helyett *qt* névtérrel lehet importálni az alábbi utasítással a *QtCore* modult:

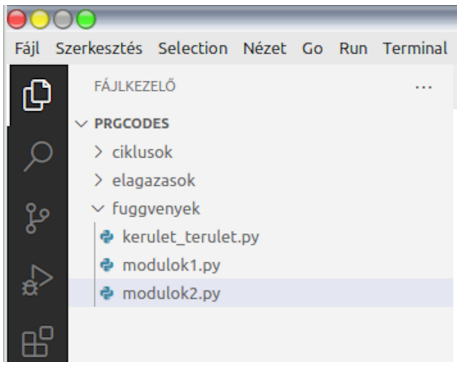
```
from PySide6 import QtCore as qt
```

12.2. Saját modulok készítése

Volt már szó arról, hogy összetettebb programokat nem ajánlatos egyetlen forrásállományban megírni, hanem a program különböző funkcióit megvalósító függvényeket külön-külön állományokba írják meg. Ezt a megközelítést a Python esetében a saját modulok írásával lehet alkalmazni.

Ekkor a program projekt könyvtárában egynél több állomány – kifejezetten összetett programok esetében több könyvtár, azokban számos állomány – jön létre a program fejlesztése során.

Ennek a dokumentumnak az elkészítéséhez tartozik egy könyvtár, amelyben levő állományok szolgálnak a dokumentumban szereplő kódok projektkönyvtáraként. Ezekből készített képernyőmentések jelennek meg a dokumentumban programkódot mutató ábraként. Az 12.5. ábra az ezen sorok írásakor ennek a könyvtárnak a tartalmát mutatja, ahogyan azt a VS Code megjeleníti az állománykezelő részben. Mint látható, jelenleg egy *ciklusok*, egy *elagazasok* és egy *fuggvenyek* nevű könyvtár van, utóbbi van kinyitott állapotban, azaz az ebben levő állományok listája jelenik meg. A VS Code egyébként ezt egy fa formájában mutatja meg, ahol az állományok felelnek meg a fa leveleinek. Amennyiben valamelyik állomány nevén duplán kattintunk, akkor azt az állományt megnyitja a program szerkesztésre. Nem látható a képen, de ha a területen van az egérmutató, akkor a jobb felső sarokban jelennek meg az új állományt, új könyvtárt létrehozni hivatott gombok is.



12.5. ábra: A dokumentum forráskódjait tartalmazó projekt könyvtára a VS Code-ban megjelenítve

Tehát egy program általában sok állományból áll, az egyes állományokban levő programrészlet pedig más állományokban definiált kódot is igénybe vehet. Ezt Python esetén szintén az **import** utasítással lehet megtenni, a projekt többi állományát modulként kezelve.

Vagyis ha például készítenék egy `haromszog.py` állományt a *fuggvenyek* könyvtárban, amelyben a `kerulet_terulet.py` állományban definiált, az előző leckében bemutatott függvényeket akarnám használni, akkor azt a `haromszog.py` állomány elején levő

```
import kerulet_terulet
```

utasítással tehetném meg. Amennyiben a `haromszog.py` állomány nem a *fuggvenyek* könyvtárban, hanem a projekt fő könyvtárában lenne, akkor azt is meg kellene adni az értelmezőnek, hogy az importáláshoz menjen be a *fuggvenyek* könyvtárba. Ez a következő két utasítás valamelyikével tehető meg:

```
import fuggvenyek.kerulet_terulet
from fuggvenyek import kerulet_terulet
```

Remélem észrevetted, hogy bár az állomány neve `.py` végződéssel rendelkezik, az import utasításban enélkül szerepel!

Amennyiben csak a területet akarjuk kiszámolni, akkor elég lenne csak a **terulet()** függvényt importálni? A válasz: igen. A

```
from fuggvenyek.kerulet_terulet import terulet
```

utasítás ezt megteszi úgy, hogy a függvény működni fog. Ebben az esetben a **kerulet()** függvény ugyan nem elérhető, de a **terulet()** függvény eléri. Visszautalva a függvényekről szóló lecke

végére, ebben az esetben a **kerulet()** a modul lokális adatterületén levő függvény, amelyet a **terulet()** függvény lát, ezért meghívhatja, de a programban a **terulet()** függvényen kívül nem látszik.

Hasonlóan, a *math* modult a `haromszog.py` állományba nem kell a **terulet()** függvény kedvéért importálni, mivel a **terulet()** függvény által elért lokális adatterületen az már elérhető. Azaz egy modul beimportálásakor nincs szükség a modulban importált további modulok importálására.

Hasonló módon lehet elrejteni bármilyen, a modulban definiált objektumot (függvényt, osztályt, változót) a modult importáló programtól, ha a nevét egy aláhúzásjellel (_) kezdjük. A modulban definiált, aláhúzásjellel kezdődő nevű objektumok csak a modulban levő függvények számára érhetőek el, a modult importáló kódban semmilyen módon nem lehet hozzájuk férni. Ezek az objektumok tehát a modul lokális objektumai.

12.3. Modul vagy főprogram?

Az előző szakaszban leírt módon létrehozott programrészek esetében fontos kérdés, hogy melyik állomány lesz a főprogram, amelyet elindítva kell a programot végrehajtani. A többi állomány vagy közvetlenül vagy közvetve ebbe a kódba lesz beimportálva modulként. Mivel egy összetettebb programot esetleg nem lehet egyben tesztelni, előfordulhat, hogy egy-egy később modulként használandó rész tesztelésére a programot önállóan kellene futtatni – ebben az esetben mintha az lenne a főprogram. Ekkor olyan kódot írnak a modul kódjába, amely a modulban definiált objektumokat használja valamilyen teszkódon keresztül.

Igen ám, de amikor a modult beimportáljuk, ez a kód is végrehajtódik! Ezt el kellene kerülni. Persze megtehető, hogy a modul tesztelése után ezt a kódot töröljük, de mi a helyzet, ha a modul fejlesztése közben már a program olyan részének a fejlesztése is zajlik párhuzamosan, amely ezt a modult használná? Ekkor nyilván még szükség van a tesztkódra, de már modulként is megtörténne az importálás, ahol viszont nem szabadna a kódnak végrehajtódnia.

Másrészt előfordulhat, hogy valaki olyan programot akar készíteni, amelyek egyes részei önállóan is végrehajthatóak, vagy egy nagyobb program részeként hajthatóak végre. Ekkor nem tesztkód

```
1  # Fibonacci számokat számító modul
2  def fib(n): # kiírja a számokat
3      a, b = 0, 1
4      while a < n:
5          print(a, end=' ')
6          a, b = b, a+b
7      print()
8
9  def fib2(n): # visszaadja a számokat egy listában
10     result = []
11     a, b = 0, 1
12     while a < n:
13         result.append(a)
14         a, b = b, a+b
15     return result
16
17 if __name__ == "__main__":
18     import sys
19     fib(int(sys.argv[1]))
20
```

12.6. Ábra: Fibonacci számokat számoló függvények modulja

kerül a kódba, hanem valóban végrehajtandó kód. Így ugyanaz az állomány önálló programként is futtatható és modulként más programba is ágyazhatóak a benne definiált függvények.

Hogyan lehet ilyen esetben elkerülni, hogy az önálló program esetén futtatandó kód a modulként importáláskor is végrehajtódjon?

A válaszhoz tudni kell, hogy a Python értelmező minden egyes modul feldolgozása során nyilvántartja, hogy éppen egy beimportált modult vagy a főprogramot dolgozza-e fel. Ezt a programkódban le is lehet kérdezni. Erre a *__name__* változó szolgál. A két aláhúzásjellel kezdődő és végződő változók az értelmező rendszerváltozói, amelyekkel az értelmező különböző állapotai kérdezhetőek le. Ez konkrétan megmondja, hogy az éppen végrehajtott kód melyik része a programnak. Amennyiben az értéke a „__main__” szöveg, akkor a főprogram.

Az 12.6. ábrán látható kód mutat példát a használatára. Ennek az elejével – a fibonacci számok számítási módjával és a listákkal – nem foglalkozunk. A 17–19. sor miatt szerepel a példa.

Itt található kód végzi el ugyanis a szükséges ellenőrzést: ha a „__main__” érték szerepel a *__name__* változóban, akkor a kód programként fut, tehát a 18–19. sorok végrehajtandóak. Modulként beimportálva a kódot, a 17. sor feltétele hamis lesz, így elkerülhető a nem kívánt kód végrehajtása.

13. Álvéletlen számok

13.1. Készítsünk programot, amely a felhasználóval úgy játssza a számkitalálós játékot, hogy a felhasználónak kell, legfeljebb tíz próbálkozásból kitalálnia a számítógép által gondolt számot!

Elsőre egyszerűen hangzik a dolog:

1. A gépnek gondolnia kell egy 1 és 1024 közti egész számot (ekkor elég a tíz kérdés).
2. Egy ciklusban legfeljebb 10 alkalommal meg kell kérdezni a felhasználótól a tippjét és kiírni válaszul, hogy a gondolt szám nagyobb vagy kisebb, esetleg a tippel megegyező-e. Ez utóbbi esetben véget kell érnie a ciklusnak akkor is, ha még nem fogyott el a tíz kérdezési lehetőség.

Jobban végiggondolva, az első lépésnél lesz gond: hogyan tud a számítógép egy számra „gondolni”?

```
1  #/ usr/bin/python3
2  """
3  szamkitalal2.py
4  Ez a program gondol egy számra, majd legfeljebb tíz lehetőséget ad a felhasználónak arra, hogy kitalálja.
5  A gondolt szám egy 1 és 1024 közötti egész szám.
6  A program a pszeudo-véletlen számok bemutatására szolgál.
7  """
8
9  print("Gondoltam egy egész számra 1 és 1024 között.")
10 print("Tíz próbálkozásból találd ki, melyik az!")
11
12 gondolt = 42    # TODO 1 és 1024 közti egész számot kell "gondolni"
13
14 # TODO Ciklus legfeljebb 10 alkalommal a felhasználó próbálkozásaira
15
```

13.1. ábra: A számkitaláló program vázlata

Hogyan lehet egy véletlenszerű eseményt, mint egy adott tartományba eső egész szám előállítása, a számítógépen előidézni? A rövid válasz: **sehogyan**! A számítógép nagyon pontosan a megadott utasításokat hajtja végre. Ezektől semmilyen módon eltérni nem képes. Véletlen számok előállítására sem.

Akkor ennyi? A programot nem lehet elkészíteni…?

Nagyon sok olyan algoritmus létezik, amely valamilyen véletlen eseményen alapszik – annak ellenére, hogy az algoritmus definíciójába a véletlen nem fér bele.

13.1. Álvéletlen számok

Léteznek bonyolult matematikai képletek, amelyek egy bemenő számból olyan kimenő számot állítanak elő, amelyből ha sorozatot alkotunk, akkor szinte lehetetlen rájönni bármilyen szabályosságra, mert a számok teljesen véletlen*szerű* értékeknek tűnnek. Az ilyen módon előállított számokat nevezzük *álvéletlen számok*nak vagy egy kicsit idegenebb hangzású néven *pszeudovéletlen számok*nak (angolul: *pseudo random number*).

13.2. ábra: Egy 600 éves, „cinkelt” dobókocka

Ezek a számok tehát nem valódi véletlen számok, de úgy viselkednek, mintha véletlen értékek lennének. Szinte minden programozási nyelv tartalmaz eszközöket, amelyekkel ilyen álvéletlen számo-

13.3. ábra: Szerepjátékban használt dobókockák. Van köztük 4, 6, 10, 12 és 20 oldalú.

kat lehet előállítani. Ezeket az eszközöket szokás (ál-)véletlenszám-generátornak nevezni.

Egy véletlenszám-generátor egy kezdőértékből kiindulva minden egyes alkalommal egy új értéket állít elő. Ez lesz a következő alkalmazásnál a kiindulási érték. Az első kiindulási értéket azonban először valamilyen módon elő kell állítani: ezt a műveletet szokás *a véletlenszám-generátor inicializálásának* nevezni.

Amennyiben a véletlenszám-generátort mindig ugyanazzal az értékkel inicializáljuk, akkor a legtöbb véletlenszám-generátor mindig ugyanazokat a számokat adja eredményül. A Python véletlenszám-generátorára is ez igaz.

Tehát mégiscsak kellene valami megoldás arra, hogy ne lehessen előre kiszámítani, hogy milyen értékeket kapunk véletlen számként. Köszönhetően az olcsó technológiának, személyi számítógépek – és alacsonyabb rangú gépek – esetén létezik valami a gépben, aminek az értékét nem lehet előre kiszámítani: a pontos idő.

Tekintve, hogy

1. a számítógépek órája nagyon pontatlanul jár;
2. két számítógépen képtelenség pontosan ugyanabban az időben elindítani egy programot (legalábbis ezredmásodperc pontosan biztosan nem);
3. mivel egy gépen egyszerre egynél több program fut, így nem mondható meg pontosan, hogy az indítás után hány másodperc telik el egy adott utasítás végrehajtásáig;

a gép órája által mutatott idővel inicializálva a véletlenszám-generátort, teljesen megjósolhatatlanná lehet tenni a kapott véletlenszámokat.

A Python véletlenszám-generátora a *random* modullal érhető el. (A modul bemutatása a Függvénytárban található.) Az idő lekérdezésére alkalmas műveleteket a time modulban találjuk (lásd ezt is a Függvénytárban). Tehát ezt a két modult kell beimportálnunk:

```
6  A program a pszeudo-véletlen számok bemutatására szolgál.
7  """
8
9  import random
10 import time
11
12 print("Gondoltam egy egész számra 1 és 1024 között.")
```

13.4. ábra: A random és a time modulok importálása

A két modul importálása a 9. és a 10. sorban történik meg.

Ezután következhet a véletlenszám-generátor inicializálása a számítógép aktuális ideje alapján (13.5. ábra, 12. sor). Mivel a PC ideje általában legalább ezredmásodperc pontosan érhető el, így gyakorlatilag nulla a valószínűsége, hogy két számítógépen a programot sikerül annyira egyszerre elindítani, hogy azonos legyen a két gépen ez az érték.

```
9  import random
10 import time
11
12 random.seed(time.time())
13
14 print("Gondoltam egy egész számra 1 és 1024 között.")
```

13.5. ábra: A véletlenszám-generátor inicializálása (12. sor)

A **random.seed()** függvény egy számot vár paraméterként, amely szám lesz a véletlenszám-generátor új bázisértéke, amelyből a következő véletlen számot előállítja. Példánkban ez a számérték a **time.time()** függvény által adott érték lesz, amely nem más, mint a számítógép belső órája által mutatott idő, ezredmásodperc pontossággal.

Ezután már lehet véletlenszámokat kérni. A véletlenszám-generátorok általában a (0; 1) nyílt intervalumba eső lebegőpontos számokat állítanak elő, amelyet a kívánt intervallumba lehet megfelelő műveletekkel konvertálni. Azonban a Python *random* modulja ezt megteszi helyettünk: a **random.randint()** függvény a paraméterében megadott két egész számmal megadott zárt intervallumba eső egész számokat ad, így a legegyszerűbb ezt használni:

```
11
12  random.seed(time.time()) # A véletlenszám-generátor inicializálása
13
14  # Gondoljunk egy számra:
15  szam = random.randint(1, 1024)
16
17  print("Gondoltam egy egész számra 1 és 1024 között.")
```

13.6. ábra: Véletlen szám 1 és 1024 között, beleértve a két határt is

Mint a példákon látható, a fenti sorok rögtön a program dokumentáló megjegyzése után kerültek beszúrásra. Bár szerepelhetne később is, érdemes hozzászokni, hogy a 13.5. ábra 9–12. sorát rögtön a program elején helyezzük el ugyanilyen formában minden olyan programban, ahol véletlen számokat akarunk használni. Természetesen ha más modulokat is használ a program, akkor azok is itt fognak szerepelni, de a szabály: a véletlenszám-generátor inicializálása lehetőleg a program első utasításainak egyike legyen!

A program további része már nem tartalmaz semmi újdonságot, így a magyarázata nem, csak maga a programkód szerepel:

```
20  i = 1
21  while i < 11:   # ciklus legfeljebb 10-szer futhat le
22      tipp = 0    # ilyen tippet nem fogadunk el, hiszen az intervallumon kívül esik
23      print(f"{i}. próbálkozás:")
24      while tipp < 1 or tipp > 1024:
25          tipp = int(input("A tipped (1 és 1024 között legyen!):"))
26      if tipp < szam:
27          print("Nagyobbra gondoltam.")
28      elif tipp > szam:
29          print("Kisebbre gondoltam.")
30      else:
31          print("Eltalátad!")
32          break   # Nincs szükség több tippelésre.
33      i += 1  # a tippek számlálójának növelése
34
35  if tipp != szam:
36      print("Próbálj valami jobb stratégiát, hátha legközelebb sikerül...")
37
```

13.7. ábra: A tippek bekérésének ciklusa

14. Szekvenciális adattípusok

Az összetett adattípusok egyik csoportja úgy tárol egyszerre több értéket, hogy azok sorrendje valamilyen módon definiálva van. Ezeket összefoglaló néven **szekvenciának** vagy *szekvenciális típusnak* nevezzük. Az angolban ezeket a típusokat az *iterable* névvel illetik, ami arra utal, hogy az egyes elemeken például egy ciklussal végig lehet menni.

Ebben a leckében ezekkel a típusokkal és az elemeiken való végighaladásra alkalmas ciklusutasítással ismerkedünk meg. Pontosabban az egyikükkel már korábban megismerkedtünk, de a részletes bemutatásuk majd csak ebben a leckében következik. Ezek a típusok a karakterláncok tárolására szolgáló **str** és rokona a **bytes** (amelyben bármely karakter szerepelhet, nem csak a nyomtatható karakterek), a lista **list** adattípusa, a *tuple* adattípus, valamint néhány *iterátor* objektum. Ezen kívül a **for** ciklusra is több példa lesz a leckében.

A Python nyelv adattípusai két kategóriába sorolhatók azok módosíthatósága szempontjából: *módosítható* (*mutable*) és *nem módosítható* (*immutable*) típusokra. Az alaptípusok mind módosítható típusok, azonban a szekvenciális típusok egy része már nem az, ahogyan a később megismerendő halmazjellegű típusoknál is vannak nem módosítható típusok. A nem módosítható típus azt jelenti, hogy a típus egyes elemei nem változtathatóak meg. Ez csupán annyit jelent, hogy az adott típusú értéket módosító bármely művelet egy új példányt hoz létre a memóriában, amely már a módosításnak megfelelő új értéket tartalmazza. A módosítható típusok értékei helyben tudnak módosulni, nem keletkezik a memóriában új példányuk. Nem módosítható típus például a karakterlánc és a tuple.

Egy objektum **iterálható**, ha képes egyszerre egy elemét elérhetővé tenni. Például iterálhatók a szekvenciák, mint az *str*, a *list* és a *tuple*, valamint néhány nem szekvenciális összetett adattípus, mint a *dict*, *állomány-objektumok* és minden olyan osztály, amely az __iter__() vagy a __getitem__() metódust megvalósítja.

14.1. A szekvenciális típusokon végezhető műveletek

A 1. táblázat azokat a műveleteket mutatja be, amelyek minden olyan típuson elvégezhetőek, amelyek több érték sorozatát tartalmazzák (szöveg, lista, stb.).

1. táblázat: Értéksorozatokon végezhető műveletek

Művelet	**Eredmény**
`x in s`	Igaz, ha egy elem *s* elemei közül megegyezik *x* értékével, különben hamis. Magyarul, igaz, ha *x* benne van elemként *s*-ben.
`x not in s`	Az előző tagadása.
`s + t`	*s* és *t* összefűzése
`s * n` vagy `n * s`	*s* összefűzése önmagával *n* alkalommal
`s[i]`	Az *i*. elem *s*-ből, ahol az első elem indexe 0.
`s[i:j]`	Szelet *s*-ből, amelynek első eleme *s[i]*, a szelet utolsó elem az *s[j]* előtti elem (lehet üres az eredmény).
`s[i:j:k]`	Szintén szelet *s*-ből az *i* indexű elemtől kezdve a *j* indexű elemet megelőző elemmel befejezve, de csak minden k-adik elemet tartalmazza.

Művelet	**Eredmény**
	Használatának inkább listáknál lehet értelme, mint szövegnél…
`len(s)`	*s* elemeinek száma
`min(s)`	A legkisebb *s*-beli elem értéke.
`max(s)`	A legnagyobb *s*-beli elem értéke.

A 2. táblázat viszont azokat a műveleteket tartalmazza, amelyek megváltoztatnák az adott típusú adatot, így csak a módosítható típusokon hajthatóak végre. A táblázatban *s* egy szekvencia (például lista), *t* egy *iterálható* objektum (lásd később), x egy bármely típusú érték, amely lehet *s* értéke. *i, j* pedig egész (*int*) érték.

2. táblázat: Módosítható szekvenciákon végezhető műveletek

Művelet	**Eredmény**
`s[i] = x`	*s i* indexű eleme *x* értéket veszi fel.
`s[i:j] = t`	*s* azon szelete, amelyet az *i* és *j* indexek meghatároznak kicserélődik a *t* szekvenciára (ha *s* lista, akkor *t* is lista).
`del s[i:j]`	Ugyanaz, mint `s[i:j] = []`, azaz a megadott szelet törlődik *s*-ből.
`s[i:j:k] = t`	Az `s[i:j:k]` által meghatározott szelet elemei a *t* elemeire cserélődnek. t-nek ugyanannyi elemből kell állnia, mind `s[i:j:k]`-nak.
`del s[i:j:k]`	Azokat az elemeket, amelyeket az `s[i:j:k]` kifejezés kiválaszt, törli a listából.
`s.append(x)`	Az *x* elemet hozzáfűzi a szekvencia végére. (Ugyanezt csinálja a következő művelet is: `s[len(s):len(s)] = [x]`)
`s.clear()`	Minden elemet eltávolít *s*-ből. (`del s[:]` művelettel azonos.)
`s.copy()`	Egy teljes másolatot készít *s*-ről a memóriában. A két változat egymástól függetlenül módosítható ezt követően. (Ugyanezt teszi: `s[:]`)
`s.extend(t)` vagy `s += t`	Kibővíti *s*-t *t* elemeivel. (Ugyanez: `s[len(s):len(s)] = t`) Másképpen fogalmazva: hozzáfűzi *s*-hez *t*-t.
`s *= n`	Kicseréli *s*-t az `s*n` kifejezés eredményére, azaz *s n*-szer való önmagával való összefűzésére.
`s.insert(i, x)`	Az *x* elemet beszúrja *s*-be az *i* indexű helyre.
`s.pop()` vagy `s.pop(i)`	Eltávolítja *s*-ből az *i* indexű elemet és visszaadja annak értékét. Amennyiben nem szerepel paraméter, akkor *i* értéke -1, azaz az utolsó elemet távolítja el.
`s.remove(x)`	Az első *x* értékű elemet eltávolítja *s*-ből. Amennyiben az elem nem szerepel *s*-ben, akkor a *ValueError* hiba keletkezik.
`s.reverse()`	Helyben megfordítja *s*-t: ami eddig az első elem volt, az lesz az utolsó stb.

Mivel a **list** egy módosítható típus, adódhatnak meglepetések! Például abban az esetben ha egy listát önmagával fűzünk össze több alkalommal:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

Ebben a példában a meglepő eredmény abból adódik, hogy az első sorban a `[[]]` kifejezés valójában egy olyan lista, amelynek egyetlen eleme egy üres lista, amelynek önmagával való 3-szorozása során ennek a belső, üres listának nem lesz három másolata, csupán három referencia kerül ugyanarra az üres listára a külső listába. Amikor ennek a listának az első elemét tartalmazó üres listához az **.append()** művelet hozzáfűz egy új elemet, akkor ennek az egy üres listának mindhárom megjelenésében megjelenik az az új elem. Ehelyett egy másik megoldással lehet három üres elemből álló listát előállítani például így:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Az első sorban szereplő **for** utasításról a következő szakaszban lesz szó...

14.2. A for utasítás

A **for** utasítást használhatjuk egy szekvencia elemein való végighaladásra. Ez az utasítás egy olyan ciklust hoz létre, amelynek ciklusmagja a szekvencia minden elemére egyszer végrehajtódik. Az adott elemet is el lehet érni a ciklusváltozón keresztül.

Az utasítás általános alakja a következő:

1. Kezdődik egy `for` kulcsszóval, amelyet egy vagy több ciklusváltozó felsorolása követ.[9]
2. Ezután az `in` kulcsszót követően egy olyan kifejezés szerepel, amely egy *iterátort* ad meg.
3. A sort egy kettőspont zárja, amelyből következik, hogy
4. a következő, bentebb kezdődő sorok alkotják a ciklusmag utasításait, amelyben a `for` kulcsszó után szereplő változók, mint ciklusváltozók használhatóak.
5. Mint a **where** utasításnál, a **for** utasításnál is követheti a ciklusmagot az `else:` rész, amely akkor hajtódik végre, ha az iterátor által megadott szekvencia minden elemére lefutott a ciklus – azaz nem igaz már, hogy van még eleme a szekvenciának.

Az utasítás ciklusmagjában a **break** és a **continue** utasítások ugyanúgy használhatóak, mint a **where** utasítás esetében.

Az alábbi példa az utasítás legegyszerűbb használatára mutat példát:

```
3  for i in range(10):
4      print(i)
5      i = 5   # ez nincs hatással a ciklusra
6
```

9 Amennyiben több változó szerepel, akkor a tuple típus szabályai érvényesek rá mintha azoknak a változóknak kellene értéket adni.

A ciklusváltozó az *i* lesz, amelynek értékei rendre a 0, 1, 2, 3, 4, 5, 6, 7 8, 9 egész számok lesznek – mivel a `range(10)` kifejezés ezeket az értékeket állítja elő, lásd a lecke egy későbbi pontján részletezve. A ciklusmag két utasítást tartalmaz, amelyek közül az első kiírja a ciklusváltozó értékeit. A második – ahogy a megjegyzés is jelzi – valójában nem csinál semmit, mivel a ciklusmag végrehajtása után az i úgyis a soron következő értéket kapja, ugyanis az új értéke nem a pillanatnyi értékétől függ, hanem attól, hogy a szekvencia (jelen esetben a 0, …, 9 számsorozat) melyik eleme következik.

A `for` kulcsszóval kezdődő sort úgy lehet tekinteni, mint egy értékadást a megadott ciklusváltozó(k) és az `in` utáni szekvencia soron következő eleme között. Ezután végrehajtódik a ciklusmag, majd ha van még eleme a szekvenciának, akkor újabb értékadás következik. Amennyiben a szekvenciának nincs további eleme, akkor az esetleg szereplő `else:` rész végrehajtása, majd a ciklus befejezése következik.

Két dolgot fontos ezzel kapcsolatban még megjegyezni, ami az értékadásból kivetkezik:

1. A ciklus befejezése után a ciklusváltozó megtartja az utoljára kapott értékét (ami a fenti példában az 5 érték lesz, egyébként a szekvencia utolsó elemének értéke lenne).
2. Ha a szekvencia üres, akkor egyszer sem történik értékadás, így ha a **for** utasítás előtt nem létezett, akkor utána sem fog ilyenkor létezni a változó!

A for utasítás használatára a leckében még sok példát fogsz találni.

14.1. Írj függvényt, amely nullától kezdve, a paraméterében megadott egész számig kiírja a páros számokat!
14.2. Írj függvényt, amely 2-től a paraméterben megadott (legfeljebb két jegyű) egész számig kiírja a prímszámokat!

14.3. Karakterláncok

Az elején, az egyszerű adattípusoknál már volt említés szintjén szó az **str** típusról. Most elérkezett az ideje, hogy alaposabban körbejárjuk.

Az *str* típus szövegek, pontosabban *karakterláncok* tárolására alkalmas. Éppen ezért már a korábbi programjainkban is rendszeresen szerepelt *str* típusú adat. Igaz, ezek minden alkalommal szövegkonstansok voltak csupán.

14.3.1. Szövegkonstansok formája

Mint a fenti bevezetőben már szerepelt, a szövegkonstansok lehetséges írási módját már ismered:

Egyszeres vagy dupla idézőjelek közé, azaz `'` vagy `"` karakterek közé írt karaktersorozat a szövegkonstans, ahol a két határolójelnek meg kell egyeznie. Amennyiben több sornyi szöveget kell megadni, akkor ennél egyszerűbb megoldás ha a határolójelet megháromszorozzuk mind az elején, mind a végén. Ez utóbbi megoldás a program vagy függvény elején dokumentáló szöveg megadására is használatos.

Azonban ennél többet is lehet ennél egy szövegkonstansban megadni. Az alábbiakban erről lesz szó.

Amennyiben egy szövegkonstans határolójele a szimpla idézőjel, akkor a dupla idézőjelet korlátozás nélkül lehet a belsejében használni; és fordítva: a szimpla idézőjelet szabadon lehet használni a dupla idézőjellel határolt szövegkonstansban. A tripla idézőjelek közé írt szövegben mindkét idézőjel használható szabadon – a tripla előfordulás esetén csak a határolójelben használttól eltérő.

Olyan eset is lehetséges azonban, ahol bárhogy válasszuk is meg a határolójelet, a szövegben azt is szerepeltetni kellene. Erre jók az *escape sequence* néven ismert karakterreferenciák. Ezek a Python esetében (is) egy fordított perjellel (`\`) kezdődnek, majd utánuk egy karakter vagy egy

számérték adja meg, hogy melyik karaktert kell ott szerepeltetni. Ilyen esetben a `\` és az utána hozzátartozó karakter vagy szám egy karakternek számít.

A teljesség igénye nélkül íme néhány ilyen karakterreferencia:

`\\`	A \ karakter maga
`\'`	A szimpla idézőjel: '
`\"`	A dupla idézőjel: "
`\n`	Az újsor karakter (ASCII kódja 13, az Enter billentyű kódja)
`\r`	A kocsivissza karakter
`\t`	Vízszintes tabulátor
`\000`	A 000 nyolcas számrendszerbeli kódú karakter
`\xhh`	A hh hexadecimális kódú karakter
`\N{név}`	A név nevű Unicode karakter

A fentieken kívül lehet még a visszaperjel után sortörést is elhelyezni, amellyel a szövegkonstanst egyszeres határolójelek használata esetén is lehet a következő sorban folytatni. Ebben az esetben a visszaperjel és az azt követő újsor karakter, valamint a következő sor elején levő szóközök nem kerülnek bele a szövegkonstansba.

Arra is van lehetőség, hogy az összefűzés műveleti jele (`+`) nélkül összefűzzünk két vagy több szövegkonstanst (de csak konstansokat, változókra ez nem igaz!): Amennyiben a konstansok között csak *whitespace*[10] karakterek vannak, akkor azokat összevonja az értelmező egyetlen szövegkonstanssá úgy, hogy nincs köztük még egy szóköz sem. Például a `"hello" 'world'` kódrészlet a `"helloworld"` karaktersorozatot jelenti.

Formázott szövegkonstansok

A Python 3.6. verziója bevezette a formázott szövegkonstansokat. Ezek olyan szövegkonstansok, amelyeknek nyitó idézőjele előtt egy `f` vagy `F` betű szerepel (f: *formatted*). A már korábban is létezett **.format()** metódussal nagyjából azonos módon működik, de azzal ellentétben sokkal rövidebben lehet leírni, hiszen nem kell külön még a szövegkonstans végére írni a metódust a paramétereivel. Bár az említett metódusról a függvénytárban szó van, érdemesebb tehát inkább ezt a megoldást használni.

A formázott szövegkonstansban a kapcsos zárójeleknek speciális funkciójuk van: Az ebbe írt kifejezés értékét az értelmező kiszámolja és behelyettesíti a szövegbe. Ráadásul a kifejezést – még a kapcsos zárójelen belül – követheti egy kettőspont, majd egy formázást előíró kód. Amennyiben a szövegben kapcsos zárójelet kellene alkalmazni, azt duplázva kell megadni (`{{` és `}}` a `{` és `}` helyett).

Például:

```
>>> name = "Fred"
>>> f"Azt mondta, a neve {name}."
"Azt mondta, a neve Fred."
>>> f"Azt mondta, a neve {name!r}."
"Azt mondta, a neve 'Fred'."
```

10 Whitespace karakternek nevezünk minden olyan karaktert vagy karaktersorozatot, amit az adott nyelven az azt értelmező programnak figyelmen kívül kell hagynia, a sorozatukat egyetlen szóközzel helyettesítve. Ilyenek a szóköz mellett az újsor jelek és a megjegyzések szoktak még lenni.

```
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
```

```
today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and
debugging
'today=January 27, 2017'
```

A fenti példákban szereplőkön felül van még egy lehetőség: a kifejezés után egy felkiáltójelet követően lehet néhány konvertáló lehetőséget is megadni.

A kifejezés egy normális Python kifejezés lehet, amelyet zárójelek közt levőnek kell érteni, néhány kivétellel: Üres kifejezés nem megengedett.

Amennyiben a kiszámolandó kifejezést magát is szeretnénk megjeleníteni, nem csak az értékét, akkor a kifejezés után írható még egy = jel is:

```
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

A kettőspont utáni formázó kifejezéssel lehet megadni a kiíráshoz használandó formázást.

Az alábbi példában az látható, hogyan lehet például a π értékét három tizedes jegy pontossággal kiírni:

```
1  import math
2  print(f"A pi értéke közelítőleg {math.pi:.3f}.")
3
```

A példában az egész hossz nincs megadva, ami azt jelenti, hogy pontosan annyi karakter legyen, amennyi az érték kiírásához szükséges.

A formátum megadására a következő betűk használhatóak:

- 's' string formátum. Az a karaktersorozat alapértelmezett formátuma, ezért elhagyható. Egész számokra:
- 'b' bináris formátum. Egész számokra alkalmazva azt kettes számrendszerben jeleníti meg.
- 'c' Egész számokra alkalmazva az annak megfelelő kódú karaktert jeleníti meg – a Unicode kódolásnak megfelelő értéket használva, természetesen.
- 'd' Egész számra alkalmazva, azt tízes számrendszerben adja meg.

- ’o’ Egész számra alkalmazva, azt nyolcas (oktális) számrendszerben írja ki.
- ’x’ hexadecimális formátum egész számok esetén. A 9-nél nagyobb értékű számjegyeket kisbetűs változatban jeleníti meg.
- ’X’ hexadecimális formátum egész számok esetén. A 9-nél nagyobb értékű számjegyeket nagybetűs változatban jeleníti meg.
- ’n’ Megegyezik a ’d’ formátummal, kivéve, hogy a rendszer helyi beállításokhoz igazítja a megjelenést.
  Lebegőpontos számokra (valamint egész számokra, amelyek a **float()** függvénnyel helyben lebegőpontos számmá alakulnak):
- ’e’ Tudományos jelölés (~normál alak) alkalmazása a megjelenítéshez.
- ’E’ Tudományos jelölés alkalmazása, nagybetűs megjelenítéssel. (0.00E-1 alakban 0.00e-1 helyett).
- ’f’ Lebegőpontos megjelenítés alkalmazása.
- ’F’ mint az ’f’, de a `nan` helyett `NAN`, az `inf` helyett `INF` jelenik meg.
- ’g’ és ’G’ esetén az érték alapján dől el, hogy lebegőpontos vagy tudományos jelölésként jelenik meg a szám.
- ’%’ százalékos megjelenés: Az érték százszorosa jelenik meg lebegőpontos alakban, mint százalékérték.

A korábbi feladatokban, példákban a formázott szövegkonstanst már többször használtuk.

14.3.2. A karakterlánc típusa, az str

Az **str** típus valójában karakterek sorozata, más néven tömbje. A számítógép memóriájában egymás utáni byte-okban elhelyezkedő értékekként kell elképzelni. Valószínűleg ez utóbbi is az oka annak, hogy hivatalosan az értéke nem változtatható meg, csak az egész karaktersorozat érték cserélhető le egy másik értékre.

Új szöveges érték bármely másik típusú értékből létrehozható az **str()** függvénnyel – valójában a típus konstruktorával. Mivel ezt alkalmazza a **print()** függvény is, az így kapott szöveg megegyezik azzal, ahogyan a **print()** írná ki az adott értéket.

14.3.3. Byte-ok sorozata: bytes

Az **str** típushoz nagyon hasonlóan működik, de gyakorlatilag bármilyen byte-érték tárolására alkalmas a **bytes** típus. Amennyiben konstansként akarjuk megadni, akkor az *str*-nél megszokott formát alkalmazhatjuk, de a kezdő idézőjel elé egy `b` vagy `B` betűt kell írni: `b"alma"` egy **bytes** típusú érték lesz, nem **str** típusú.

14.3.4. Szöveg indexelése

Mivel tömbről van szó, a karaktersorozatot alkotó egyes karakterek külön is elérhetőek indexeléssel. Ennek módja, hogy az **str** típusú kifejezés után szögletes zárójelekben az elérni kívánt elem indexét írjuk. Az index egy egész érték, amely alapesetben nemnegatív számként, nullától indulva az elem sorszámát jelenti.

Régóta létező és a mai processzorok gyakorlatilag mindegyike által támogatott címzési módszer, amikor az utasítás kap egy memóriacímet és egy offset értéket. Ez az offset egy egész szám, amely megmondja, hogy az eredeti memóriacímhez mennyit kell hozzáadni, hogy a ténylegesen keresett byte címét kapja a processzor. Az újabb processzorok, amelyek több byte méretű adatokat tudnak egyszerre címezni, az offsetet esetleg már nem is byte-ban mérik, hanem nagyobb egységeket is ismernek.

Az offset tehát egy hosszabb adatsor – például karakterek sorozata – valamelyik elemét tudja megadni, az adatsor elejéhez képest. Amennyiben értéke 0, akkor az első elemet lehet elérni, pozitív értékek pedig az ezt követő elemekre mutatnak.

A fenti címzési módot indexelésnek nevezük. Ennek a gépi kódú műveletnek a magasszintű programozási nyelvekben a megfelelői a tömbök, mint a Python karakterláncai.

Tehát ha *s* egy **str** típusú változó, annak első karakterét az `s[0]` kifejezéssel lehet elérni. Amennyiben van legalább 6 karakter benne, akkor az `s[5]` a hatodik karakterét jelenti. Viszont ha olyan indexet adunk meg, amely már túlmutat a karakterek elemszámán, annak hibaüzenet lesz az eredménye:

```
>>> s = "alma"
>>> s[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Ugyanakkor lehetőség van negatív indexek használatára is. Ekkor a szöveg utolsó karakterétől visszafelé indul az indexelés, -1 értékkel kezdve. Természetesen ekkor is igaz, hogy nem létező indexű elemre hivatkozás a fenti hibaüzenetet okozza.

14.1. Ábra: Az indexek jelentése

A 14.1. ábra mutatja a „Python” szövegen, hogy az indexek mikor melyik karakterre mutatnak.

Mivel a szöveg tekinthető az azt alkotó karakterek tömbjeként (az indexelés is ezért lehetséges), így alkalmazható a **len()** függvény is, amely ha a paramétere egy *str* típusú érték, akkor megadja annak karaktereinek számát, mint a szöveg hosszát.

14.3.5. Szöveg szeletelése

Angolul *slice*, magyar fordításban *szelet* a neve annak a kifejezésnek, amely a karakatrelánc egy részláncát eredményezi egy adott indextől egy másik indexig terjedően.

Ezt a szeletelést az indexeléshez hasonlóan az *str* típusú kifejezés után írt szögletes zárójellel lehet megadni, viszont ebben az esetben két egész számmal, kettősponttal elválasztva őket. A két egész szám egy-egy index a következő jelentésekkel:

- Az első érték a szelet kezdő elemének indexe.
- A második érték a szelet utolsó elemét követő elem indexe.

Az egyes elemek el is hagyhatóak:

- Ha a kettőspont előtti index hiányzik, akkor a szelet a karakterlánc elejétől kezdődik.
- Ha a kettőspont utáni index hiányzik, akkor a szelet a karakterlánc végéig tart.
- A fenti kettőből következően ha mindkét index hiányzik – de a kettőspont szerepel –, akkor a teljes karakterlánc másolatát tartalmazza a szelet.

Példa a fentiekre:

```
ital = 'almabor'
ital[0:4]      # 'alma'
ital[4:-1]     # 'bo'
ital[:4]       # 'alma'
ital[4:]       # 'bor'
ital[:]        # 'almabor'
```

A szeletelés során megadott index túlmutathat a karakterlánc elemein, nem okoz hibát. Így például a fenti példában az utolsó előtti kifejezéssel azonos eredményt adna bármely 7-nél nagyobb index megadása: `ital[4:10]`.

14.3.6. A karakterlánc elemein végigmenő lista

Mivel a karakterlánc is egy szekvencia, így a for ciklussal végig lehet haladni az elemein, azaz a benne levő karaktereken:

```
7  string = 'banánfaló majmok'
8  for karakter in string:
9      print(karakter, end=' ')
10
```

A 9. sorban a print() második paramétere gondoskodik róla, hogy a kiírás végén ne új sor kezdődjön, hanem csak egy szóköz válassza el az egymást követő kiírásokat egymástól. (Ilyenkor vigyázni kell: a ciklus után nem fog a kiírás automatikusan új sorba kerülni, arról egy új sor jel kiírásával külön gondoskodni kell, ami ebben a példakódban nem szerepel!) A ciklus eredményeként kapott kiírásban a sztring változóban levő szöveg karakterei úgy szerepelnek, hogy köztük van egy-egy extra szóköz is elhelyezve.

14.4. Feladatok karakterláncokra

14.3. Írjunk programot, amely beolvas egy karakterláncot, majd

1. kiírja a hosszát;
2. kiírja megfordítva;
3. számoljuk meg, hány szóköz van benne;
4. az első 'A' betűtől kezdve írjuk ki 10 karakterét;
5. az első 'jaj' szövegrészt cseréljük ki a 'hajaj' szövegrészre és innen kezdve írjuk ki az egész karakterláncot;
6. tegyük a karakterlánc végére, hogy ' Hello';
7. végül vegyük el belőle a szóközöket!

Minden olyan pontnál, ahol nem szerepel, végezzük el a keletkezett eredmény kiírását.

A feladat megoldásához szükség lehet néhány szövegkezelő függvényre, amelyek rövid magyar nyelvű leírása a Függvénytáblázatban megtalálható, illetve angolul a hivatalos Python dokumentációban.

14.4. Írj programot, amely bekéri a felhasználótól két részletben a nevét: először a vezetéknevét, majd a keresztnevét. Ezután írja ki a program az illető monogramját. Például Kíváncsi Ödön esetén a K.Ö. monogramot kell adnia. A program működjön helyesen akkor is, ha a felhasználó a neve előtt szóközöket is begépel, illetve ha kisbetűvel írja be a nevét, akkor is nagybetűs legyen a monogram!
14.5. Olvassunk be egy szöveget a programba, valamint azt a két karakterláncot, amit és amire ki kell az először beolvasott szövegben cserélni, majd végezzük el a cserét és írjuk ki a kapott új karakterláncot!
14.6. Egy beolvasott karakterláncban keressük meg az első '<<<' és '>>>' közé zárt szövegrészt és írjuk ki azt a képernyőre!
14.7. Olvassunk be egy szöveget. A szövegben levő első számot növeljük meg 5-tel és az így módosított szöveget írjuk a képernyőre!
14.8. Olvassunk be egy dátumot ÉÉÉÉ-HH-NN alakban! A dátumot ismételten kérjük be, ha az nem helyes (formai hiba van benne, hibás hónap vagy nap sorszám szerepel benne)! Elég az első hibát felderíteni. Bontsuk szét a dátumot a három komponensére, amelyeket egy-egy egész típusú változóban tárolja el a program, majd írja ki azokat.

14.5. Listák

A listák olyan módosítható (*mutable*) szekvenciák, amelyeket tipikusan egyforma értékek gyűjteményének tárolására használunk.

Más elterjedtebb nyelvekben igazából nincs is lista típus. Helyette tömb típus van, amely sokkal megkötöttebb a listánál: a tömb elemei a memória egy folyamatos részén helyezkednek el egymás után, míg a lista elemei bárhol lehetnek a memóriában egymáshoz képest. Ebből az eltérésből is adódik, hogy míg a lista egyes elemei különböző méretű memóriaterületet is igénybe vehetnek, a tömb esetében a létrejöttekor ismertnek kell lennie az elemek memóriaigényének. Ezért a tömbök esetében minden elem típusának meg kell egyeznie. Mivel a Python a tömbműveleteket is a lista adattípusra biztosítja, így ez a megkötés nem érvényes.

A típus neve: **list**. Ebből talán már következik, hogy különböző típusú értékekből a **list()** függvény képes listát létrehozni. De egy lista többféleképpen is létrehozható.

Egy szögletes zárójelpár megadásával egy üres lista hozható létre: `[]`.

Szögletes zárójelek között értékeket vesszővel elválasztva adhatók meg a lista elemei.

Például ha az első öt négyzetszámot akarjuk egy listában tárolni, az így néz ki:[11]

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

További lehetőség az elemeket egy **for** kifejezéssel legenerálni: `[x for x in iterable]`.

Ezen felül használható a fent már említett **list()** konstruktorfüggvény is. Amennyiben nincs paramétere, akkor üres listát hoz létre, ha szerepel egy szekvencia a paraméterében, akkor annak elemei lesznek a létrejövő lista elemei.

11 Az angol nyelvű példák a Python dokumentációból származnak: https://docs.python.org/3/tutorial/introduction.html#lists

Mint a szövegek és a többi szekvenciális típus, amit a Python alapból ismer, a lista indexelhető és szeletelhető:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Minden szeletelő kifejezés egy új listát hoz létre. Ebből következően az eredeti *sqares* lista másolatát kapjuk az alábbi utasítással:

```
squares[:]
```

Ez a memóriában teljesen új másolatot jelent, amely az eredetitől függetlenül változtatható meg. Ennek a jelentősége a listán végighaladó és azt módosító ciklusoknál, illetve függvényhívásoknál lehet: Amennyiben a ciklus vagy a függvény módosítja a listát, amelyen végighalad, annak előre nem látható következményei lehetnek. Ennek elkerülése érdekében ilyenkor az eredeti lista egy másolatát kell a ciklusnak vagy a függvénynek odaadni.

A szövegekkel ellentétben azonban egy-egy elem értéke megváltoztatható. A listára az összefűzés is alkalmazható, amely szintén egy új listát hoz létre.

A listával végezhető még sok további művelet függvényekkel és a lista metódusaival, amelyek felsorolása a Függvénytárban megtalálható.

14.5.1. Lista kifejezések

A lista kifejezések segítségével egy listát bizonyos esetekben könnyebben lehet létrehozni. Ilyen eset például amikor valamilyen művelet eredményeként létrejövő értékekkel akarunk egy új listát létrehozni.

Például a következő kódrészlettel lehet a négyzetszámok listáját létrehozni:

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

Azonban a fenti kódrészlet hatására melléktermékként az *x* változóban a ciklus után ott lesz a legutolsó érték (9). Ez kerülhető el az alábbi lista kifejezés alkalmazásával:

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

vagy ennél olvashatóbban:

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

Egy lista kifejezés szögletes zárójelben egy kifejezésből áll, amit a **for-kifejezés** követ, majd nulla vagy több for- vagy if-kifejezés. Az eredmény egy új lista, amelynek elemei a megadott kifejezésből lesznek kiszámolva, amelyben a **for** utasításnál megszokott módon képzett ciklusváltozó értéke kerül felhasználásra. Amennyiben **if-kifejezés** is szerepel, akkor a ciklusváltozó értékének felhasználását feltételhez lehet kötni.

Az alábbi példa egy elég bonyolult kifejezés, amely egy olyan listát állít elő másik két listából, amelyben a két eredeti lista elemei szerepelnek, *ha azok nem azonos értékűek*:

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

A fenti kifejezés ekvivalens az alábbi programkóddal:

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

A lista kifejezések egymásba is ágyazhatóak.

14.5.2. A del utasítás listaelemekre

A **del** utasítás nem csak egész változó törlésére alkalmas, hanem egy lista egy részének törlésére is. Akár egy elemet, akár a lista egy elemét is törölni lehet az utasítással:

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

Amennyiben a listából elemeket törlünk – akár a **del** utasítással, akár más módon –, a lista újraindexelése automatikusan megtörténik, ugyanúgy, mint elemek beszúrásakor.

14.5.3. Ciklus a lista elemeire

A **for** ciklust listákra többféleképpen is alkalmazhatjuk.

Amennyiben csak az elemek értéke szükséges a ciklusmagban, akkor elég a lista elemein végighaladni:

```
11 szavak = ['macsek', 'ablak', 'demonstráció']
12 for szo in szavak:
13     print(szo, len(szo))
```

A fenti példában a 13. sor a lista egyes elemeinek értékét és a karaktereinek számát írja ki egy-egy sorban.

Azokban az esetekben, amikor olyasmit kell a ciklusban csinálni, amihez több, egymás közelében levő elemet is el kell érni, az egyes elemek indexéhez is hozzá kell férni. Ekkor nem az elemeken, hanem azok indexén kell végigmenni. Erre az alábbi megoldás a legegyszerűbb:

```
15  for index in range(len(szavak)):
16      print(index, szavak[index])
17
```

A 15. sorban az `in` utáni kifejezésben a **len()** az előző példában szerepelt lista elemszámát adja, így a **range()** a 0, 1, 2 értékeket adja egymás után, így az *index* változó pont a lista indexeit fogja rendre felvenni. Amennyiben túl hosszú lenne az elem értékét adó kifejezés, akkor talán kényelmesebben írható a fenti kód az alábbi módosítással:

```
17
18  for index, szo in enumerate(szavak)
19      print(index, szo)
20
```

Ez a kód példa arra az esetre, amikor nem egy ciklusváltozó van, hanem a következő szakaszban bemutatásra kerülő *tuple* típust használva több, jelen esetben kettő ciklusváltozó szerepel az értékadás szabályait betartva a „baloldalon”. A „jobboldalon” akkor egy olyan iterátornak kell szerepelnie, amely pontosan annyi értékből álló *tuple* értéket szolgáltat, amennyi az `in` szó baloldalán szerepel. Ebben az esetben ezt az értékpárokat szolgáltató **enumerate()** függvény eredményezi. Ez a függvény a paraméterében szereplő szekvencia elemeit adja egy kételemű *tuple* formájában, amelynek első értéke az elem indexe, a második pedig az értéke. Így, bár egyszerűbben írható a 19. sor, ez a kódrészlet az előző példáéval azonos eredményt ad.

Vigyázni kell azonban a szekvenciákon végighaladó for ciklussal! Amennyiben a ciklus esetleg módosíthatja az adott szekvenciát, akkor az a ciklus viselkedésére nem várt hatással lehet. Ezt elkerülendő mindig a szekvencia egy másolatára kell alkalmazni a ciklust:

```
20
21  a = ['alma', 'körte', 'barack', 'banán', 'kiwi']
22  for gyumolcs in a.copy():
23      if gyumolcs == 'kiwi':
24          a.append('narancs')
25
```

Amennyiben a 22. sorban csak az *a* lista szerepelne, nem annak másolata, akkor a ciklusmag a ’narancs’ elemre is végrehajtódna. Ez még nem lenne feltétlenül baj, de képzeljük el, mi történik egy elem törlésének vagy két elem felcserélésnek hatására!

14.6. A tuple adattípus

Vannak olyan – nem is olyan ritka esetek –, amikor ott, ahol egyébként egy érték szerepelhet, több értéket kellene szerepeltetni. Erre alkalmas a **tuple** típus, amely értékek nem módosítható sorozata. Vannak olyan függvények (mint például az **enumerate()**), amelyek eleve ilyen *tuple* típusú értékeket adnak eredményül. Emellett több esetben kell ilyen értéksorozatokat egy-egy függvényben paraméterként is megadni.

A *tuple* létrehozható a szokásos konstruktorfüggvénnyel:

A **tuple()** függvény egy nulla elemű tuple-t hoz létre. Ellenben ha a paraméterében bármilyen szekvenciális típusú érték szerepel, akkor annak elemeivel jön létre egy *tuple*.

Ezen felül még az alábbi módszerekkel lehet *tuple* értéket létrehozni.

- Egy üres kerekzárójelpár egy nulla elemű *tuple*-t hoz létre: `()`
- Egy érték után egy vesszőt elhelyezve egy elemű *tuple* hozható létre. Ez kerülhet opcionálisan kerek zárójelbe is: `a,` vagy `(a,)` egyaránt egy elemű *tuple*, amelynek *a* az eleme. Viszont az `(a)` nem lesz *tuple*.
- Több elemet vesszővel elválasztva: `a, b, c` vagy `(a, b, c)`.

Ott, ahol egyértelmű, hogy miről van szó, a kerek zárójelek elhagyhatóak, de nem okoz hibát sehol azok szerepeltetése. Így ha nem vagyunk benne biztosak, hogy elhagyható-e, akkor tegyük ki, abból nem lehet baj.

Például `f(a, b, c)` egy függvényhívás három paraméterrel, míg `f((a, b, c))` egy függvényhívás, amelynek egyetlen paramétere egy három elemű tuple.

14.7. Iterátorok

Az *iterátor* szóra nehéz jó magyar elnevezést találni. A Python esetében az iterátorok olyan objektumok, amelyek minden alkalommal újabb értéket szolgáltatnak valamilyen szabály szerint. Ilyen iterátor például a **Range()** függvénnyel érhető el, de van más iterátor is.

A lényege mindegyiknek, hogy olyan módon adnak értéksorozatot, hogy minden alkalommal csak egy értéket adnak, majd a következő alkalommal adják a következő értéket. Ez a listáknál a memória szempontjából gazdaságosabb, mivel nem kell az értéksorozat minden értékét egyszerre a memóriában tárolni, hanem csupán a következő értéket valamint az azt követő érték előállítását végző képletet.

Az iterátorokra jellemző, hogy egy sorozatnak egyszerre mindig csak egy elemét adják, amelyet a **.__next__()** metódussal lehet megkapni. Ez a metódus nem csak a soron következő elemet adja eredményül, hanem a sorozat következő elemére is továbblép, hogy a következő meghívásakor már azt tudja adni. Amennyiben elfogyott a sorozat, tehát nem tud több elemet adni, akkor a `StopIteration` kivétel keletkezik minden további meghívása esetén.

Valójában a **for** utasítás is a fent leírtak alapján halad végig a megadott értékeken. Ennek érdekében minden szekvenciális típushoz – és még néhány egyéb összetett típushoz – tartozik egy iterátor, amellyel valamilyen sorrendben az elemeihez hozzá lehet jutni. A **for** utasítás ezt az iterátort használja. Ebből az is következik, hogy a **for** utasítást használhatjuk minden olyan típussal, amelyhez tartozik – vagy definiálunk[12] – iterátort.

14.7.1. Range

A **range()** függvény maga is egy típust reprezentál, a **range** típust, azaz ez is egy konstruktorfüggvény. A *range* típus egy iterátor, egy nem módosítható számsorozat, amit rendszeresen használunk például a **for** utasításban. A legegyszerűbb használat esetében a sorozat nullától indul egy pozitív egész számig. De előállítható más egész számokból álló sorozat is.

A **range()** függvény háromféleképpen paraméterezhető:

1. Egy paraméter megadása esetén a létrejövő iterátor nullától indul és a paraméternél egyel kisebb egész számig tart: `Range(n)` eredménye a 0, … *n*-1 számsorozat lesz.
2. Két paraméter megadása esetén az első paraméter a sorozat kezdőértékét adja meg, a második paraméter ugyanúgy a végét, mint az egy paraméteres változatban. Tehát a `range(start, stop)` kifejezés a *start*, …, *stop*-1 egész számokat adó iterátor lesz.
3. Három paraméter esetén az első két paraméter a kétparaméteres változatnak felel meg, míg a harmadik paraméter a számok lépésközét adja meg (elhagyása esetén ez 1 lenne).

Az alábbiakban néhány példa következik. Ezekhez elöljáróban egy kis magyarázat: Mivel a *range* egy önálló típus, annak értékét kiírva nem kapunk túl beszédes választ. Ellenben, ha a **list()** segítségével a range típusú iterátort listává alakítjuk, akkor már látni fogjuk az iterátor által előállított értékek sorozatát. Ezért vannak a példában az iterátorok listává alakítva.

12 Ez utóbbi már nagyon az objektum-orientált programozás körébe tartozik, ezért ezzel nem fogunk foglalkozni.

```
>>>list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>>list(range(0, 10, 3))
[0, 3, 6, 9]
>>>list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>>list(range(0))
[]
>>>list(range(1, 0))
[]
```

A *range* típusú értékekre alkalmazható az összes nem módosítható szekvenciákra alkalmazható művelet (lásd 1. táblázat), kivéve az összefűzést és az ismétlést (+ és * műveletek).

A *range* típus előnye a listához vagy a *tuple*-hoz képest, hogy a range mindig ugyanazt a kis memóriaterületet foglalja le a tényleges elemszámtól függetlenül, hiszen csak a kezdő, befejező értéket és a lépésközt tárolja, minden mást szükség esetén számol ki a gép.

A következőkben néhány példa látható ezekre a műveletekre. Elsőként mindjárt azt is lehet látni, miért nem érdemes közvetlenül a *range* értékét kiíratni.

```
>>>r = range(0, 20, 2)
r
range(0, 20, 2)
>>>11 in r
False
>>>10 in r
True
>>>r.index(10)
5
>>>r[5]
10
>>>r[:5]
range(0, 10, 2)
>>>r[-1]
18
```

Az egyenlőségvizsgálat *range* értékekre az általuk reprezentált szekvenciák összehasonlításaként történik, így egyenlő lehet két olyan *range*, amelyeknek nem azonos a kezdő- és záróértékük vagy a lépésközük, ha ugyanazokat az értékeket adják.

Például a `range(0) == range(2, 1, 3)` igaz, mivel mindkettő üres sorozatot jelent, vagy `range(0, 3, 2) == range(0, 4, 2)`.

Mivel a **len()** függvény a paraméterében szereplő szekvencia elemszámát adja, a **len()** és a **range()** kombinálásával végig lehet menni egy lista vagy karaktersorozat indexein:

A `range(len(l))` kifejezés az *l* lista indexeit adja egymás után, így ezt egy **for** ciklusnak adva, a lista elemein az indexein keresztül lehet végiglépkedni.

14.7.2. Enumarate

Az **enumerate()** függvény paramétere is egy tetszőleges szekvenciális típus lehet. Az előállított iterátor két elemű *tuple*-kat tartalmaz. Minden *tuple* első eleme az aktuális elem indexe az eredeti szekvenciában, míg a második annak az elemnek az értéke. Ez nagyon hasznos lehet például az olyan **for** ciklusokban, ahol egyszerre szükség van az elem indexére és értékére is (lásd a listák elemeire vonatkozó ciklus utolsó példáját a 14.5.3. szakaszban).

14.8. Feladatok listákra és ciklusokra

14.8.1. Lottóhúzó program

Készítsetek lottóhúzó programot! A program legyen képes bármilyen fajta lottó-változat szimulálására: induláskor kérje be a felhasználótól, hogy összesen hány számból mennyi számot kell kihúzni.

Magyarországon a törvény szerint 18 éves kor alatt tilos a szerencsejáték!

A szerencsejátékok közül a legnépszerűbb a Lottó, amelynek ma már több változata is létezik. Nevét az olasz *lotto* (magyarul: *sors*, *nyeremény*) szóból kapta. Magyarországon sokáig *lutri* néven is emlegették.

Hazánkban a ma futó változatai közül a legrégebbi az Ötöslottó, ahol 90 számból húznak ki 5 számot. Kicsit fiatalabb a Hatoslottó, ahol 45 számból húznak ki 6+1 számot (6 számot kell eltalálni, de van egy pótszám arra az esetre, ha valaki ötöt eltalál, így a pótszámmal is még nyerhet). Léteznek további változatok, mint például a Skandináv Lottó.

A húzás hagyományosan egy urnából történik, amelyben 1-től folyamatosan számozott golyók találhatóak, a legnagyobb szám tehát annyi, ahány golyó összesen van. A kihúzott számokat a végén sorba rendezik, tehát a kihúzás sorrendje nem számít.

Először is, mivel sorsolásról van szó, szükség lesz a véletlenszám-generátorra, így a *time* és a *random* modult importálni kell a programba:

```
1  #! /usr/bin/python3
2
3  """
4  Ez a program a lottóhúzást szimulálja. Először megkérdezi a felhasználótól, hogy összesen hány számból
5  mennyi számot kell kihúzni, kihúzza és sorba rendezve kiírja a képernyőre.
6  """
7
Testing  import random, time
```

Ezután be kell kérni a két adatot: mennyi számból összesen hány számot kell kihúzni. Amennyiben az adatbekérésnél ellenőrzést is akarunk végezni, akkor arra is figyelni kell, hogy az urnában határozottan több golyónak kell lennie, mint amennyit ki kell húzni. Amennyiben többet kellene kihúzni, akkor nyilván az nem lenne lehetséges, mert túl hamar kiürülne az urna, míg ha ugyanannyit, akkor mindet kihúzva gyakorlatilag nem lenne semmi kiszámíthatatlan a húzásban, hiszen a húzás sorrendje nem számít.

```
10  # Az urna számosságának bekérése adatellenőrzéssel:
11  urna_merete = 0
12  while urna_merete < 1:
13      urna_merete = int(input("Hány golyó legyen az urnában? "))
14  # a kihúzandó számok számának lekédezése adatellenőrzéssel:
15  kihuzando = 0
16  while kihuzando < 1 or kihuzando >= urna_merete:
17      kihuzando = int(input("Hány számot kell kihúzni? "))
18
```

Az urna esetében azonban nem egyszerűen a mérete kell, hanem konkrétan azok a golyók kellenek valamilyen módon tárolva, amelyeket az urnából majd ki lehet húzni. Hasonló módon a kihúzott golyókat is tárolni kell majd valahol. Valami halmaz-szerű adattípusra van tehát szükség.

Van ugyan a Python nyelvnek halmaz típusa is, de az alapvetően nem módosítható típus – majd tizenegyedikben lesz róla szó. Viszont a lista is jól használható arra, hogy elemeket távolítsunk el belőle és helyezzünk bele.

Hozzunk tehát létre az urna elemei számára egy listát, amelybe rögtön bele is tesszük a kihúzható számokat (amelyeket a valóságban egy-egy golyó reprezentálna):

```
19  # Az urna feltöltése:
20  urna = list(range(urna_merete+1))[1:] # a lista első elemét (ami nulla) elhagyjuk
21
```

A fenti utasításban több minden szerepel egyben, fejtsük meg, mik ezek!

Először is a **list()** a lista típus konstruktora, tehát az *urna* változóba egy lista fog bele kerülni. Paramétere a **range()**, amely egy paraméter esetén az ott szereplő egész számnál kisebb nem negatív egész számokat adja, ezt fogja a **list()** ezen számok listájává alakítani. De nekünk 1) a nulla nem kell 2) az *urna_merete* által megadott szám is kell, vagyis igazából egyel meg kellene növelni minden, a **range()** által adott számértéket. Ebből a 2)-re megoldás a **range()** paraméterében az *urna_merete* változó megnövelése egyel. Így egy olyan lista keletkezik, amely a [0; *urna_merete*] zárt intervallumba eső számokat adja, amelyből a legelső érték, a 0 felesleges. Ezt tünteti el a **list()** függvény eredményeként kapott listán végrehajtandó `[1:]` szeletelő kifejezés, amely a lista másolatát adja a 0 indexű elem, azaz a 0 érték nélkül, megoldva az 1) problémát is. Tehát az utasítás hatására az *urna* változó azt a listát fogja tartalmazni, amelyben az [1; *urna_merete*] zárt intervallumba eső valamennyi egész szám fog egyszer szerepelni – mégpedig növekvő sorrendben, de ez számunkra most nem lesz lényeges.

Megvan tehát az urnánk, amiből a számokat reprezentáló golyók kihúzása történik. Hogyan történjen a számhúzás?

Kell egy másik lista is a kihúzott számok tárolására:

```
24  # A számok khúzása:
25  kihuzottak = []     # üres lista létrehozása
```

Annyi számot kell kihúzni, amennyit a felhasználó kér, tehát kell egy ciklus, aminél tudjuk, hogy hányszor kell végrehajtani a ciklusmagot. Tehát egy **for** ciklus a **range()** által adott értékekre. A függvény paramétere pedig az a szám legyen, ahány számot ki kell húzni, tehát a *kihuzando* változó értéke.

Mit kell a ciklusban csinálni?

„*Kihúzni*” egy számot az *urna* listából és azt a számot eltárolni a *kihuzottak* listában. Ehhez véletlenszerűen ki kell választani az *urna* egy elemét, a kiválasztott elem értékét új elemként a *kihuzottak* listához adni és törölni az elemet az *urna* listából. Ezáltal az urna lista a ciklusmag minden

végrehajtása során egy elemmel rövidül, azaz az adott elem kiválasztásánál minden esetben figyelni kell arra, hogy a listának pillanatnyilag hány eleme van. Vagyis az elem indexét abból a zárt intervallumból kell választani, amelynek legkisebb eleme 0 (első elem indexe), az utolsó eleme az utolsó elem indexe, amelyet a `len(urna)-1` kifejezéssel kaphatunk meg. Ezt az értéket a **random.randint()** függvénnyel kaphatjuk meg. A kapott indexű elemet a **list.pop(index)** metódussal kivehetjük a listából úgy, hogy az elem értékét a metódus visszatérési értéke meg fogja adni. Ezt az értéket közvetlenül eltehetjük a **list.append(érték)** metódussal a kihuzottak listába. Mindezeket végzi el az alábbi ciklus:

```
26 for i in range(kihuzando):
27     szam = random.randint(0, len(urna)-1)   # kihúzunk egy golyót az urnából (az indexét kapjuk)
28     # a kihúzott számot eltároljuk és egyben töröljük az urnából:
29     kihuzottak.append(urna.pop(szam))
```

A fenti utasításban az *i* változót a ciklusmag nem használja, így akár a _ szimbólumot is lehetne használni, amivel a Python azt adja meg, hogy a ciklusváltozó értékére nincs szükség, így nem kell tárolni, de minden lehetséges értékre a ciklusmagot végre kell hajtani.

Ezután a ciklus után gyakorlatilag megvannak a kihúzott számok, de a programnak azokat növekvő sorrendbe kellene tennie. Később – tizenegyedik évfolyamon – majd fogsz olyan algoritmusokkal találkozni, amelyekkel egy lista elemeit rendezni lehet. Most viszont kihasználjuk, hogy a lista adatszerkezetre alkalmazható műveletek között szerepel az elemek sorbarendezése is, amelyet a **list.sort()** metódussal lehet elérni. Ez a művelet helyben cseréli meg úgy a lista elemeit, hogy azok növekvő sorrendben legyenek, amennyiben az elemekre van definiálva valamilyen rendező elv. Ilyen rendező elv számokra a < művelet szerinti sorrend, szövegre az ábécé sorrend.

Van egy **sorted()** nevű függvény is, amely viszont a paraméterébe tett lista elemeinek sorba rendezésével kapott listát adja eredményül. Ez egy új, a paraméterben szereplőtől független lista lesz. Mivel ez a függvény listát ad eredményül, amin a for utasítással végig lehet menni, jelen esetben talán jobb megoldás ennek a használata:

```
31 # növekvő sorrendben kiírjuk a kihúzott számokat:
32 for szam in sorted(kihuzottak):
33     print(szam, end=" ")
34 print()
```

A fenti kód végigmegy a kihúzott számok listájából kapott rendezett listán, és szóközzel elválasztva kiírja azokat. Mivel a ciklus végén a kiírás az utolsó kiírt számot követő szóköz után folytatódna, a 34. sorban levő **print()** függvény még egy újsor jelet is kiír, hogy a kiírás új sorban tudjon folytatódni.

A további, listák vagy a **for** utasítás (vagy mindkettő) használatát igénylő feladatok megoldása már nem fog szerepelni, azok megoldását vagy tanároddal közösen, vagy pár fős csoportokban, vagy önállóan kell kitalálnod, és a megoldást adó programot elkészítened. Hogy melyik feladatnál hogyan, azt a tanárod fogja meghatározni.

14.8.2. Véletlen-statisztika

A statisztikusok sokat használnak véletlenszámokat. Azonban csak olyan véletlenszám előállító algoritmus jó számukra, amellyel előállított számok *egyenletes eloszlásúak*. Ez azt jelenti, hogy az egyes lehetséges értékek megjelenésének a valószínűsége azonos: például a [0; 10] intervallumba eső egész számok mindegyike ugyanannyiszor jön ki, ha elég nagy számú mintát állítunk elő.

Teszteljük le, hogy vajon a Python *random* modulja ilyen egyenletes eloszlású véletlenszámokat adó algoritmust használ-e![13]

Első lépésként készítsünk egy pénzfeldobás-szimulátort, majd fejlesszük azt tovább:

1. A gép írja ki, hogy fejet vagy írást „dobott”. Erre használjuk a **random.randint()** függvényt!
2. Készítsünk statisztikát, amely egymillió pénzfeldobást végez és kiírja, ebből mennyi lett fej és mennyi írás!
3. Írjuk át a programot úgy, hogy kockadobásokat szimuláljon! Nézzük meg itt is, hogy egymillió kockadobás esetén melyik érték hány százalékkal szerepel. (Ha mindegyik azonos vagy majdnem azonos százalékkal szerepel, akkor nevezhetjük egyenletesnek az eloszlást.)
4. Készítsünk cinkelt kockát: A hatos szám jöjjön ki kétszer akkora eséllyel, mint a többi szám!

14.8.3. További feladatok

14.9. Írjunk szorzótáblát a kicsiknek! A várt kimenet:

```
 1 *  1 =   1
 1 *  2 =   2
…
 6 *  6 =  36
 6 *  7 =  42
…
10 *  9 =  90
10 * 10 = 100
```

14.10. Írj programot, amely egy-egy listába bekéri három leves, főétel és desszert nevét, majd kiír három menüt, mindegyikben egy levessel, főétellel és desszerttel!

14.11. Írd meg az előző feladat olyan változatát, amely véletlenszerűen összekeveri az egyes menükbe kerülő ételeket, de minden menüben más-más ételt ad meg – azaz a harmadik menüre már az előző két menüből kimaradt ételek kerülnek!

14.12. Szimuláljunk tízmillió kockadobást és az eredményeket tároljuk listában! A programunk számolja meg, hogy hányszor dobtunk hatost!
Számoljuk meg azt is, hány helyen előzi meg a hatos dobást ötös dobás? Hány helyen van egymás után két hatos dobás?

13 A modul leírásában szerepel egy figyelmeztetés, miszerint statisztikai számításokhoz nem elég egyenletes eloszlásúak a *random* modul által előállított számok.

14.13. Írjunk olyan programot, amely egy verseny résztvevőinek célba érkezés szerinti névsorát kéri a felhasználótól, majd kiírja a dobogósokat és a sereghajtót (az utolsót)! A program minden újabb versenyző bekérése előtt írja ki az épp aktuális névsort!

14.14. Állítsunk elő harmincelemű, nulla és kilenc közötti véletlen számokat tartalmazó listát! A számok egy útvonal magassági adatait jelentik. Meredek az útszakasz, ha legalább kettővel magasabb az aktuális hely, mint az előző. Írja ki a program a listát (hogy az eredményt ellenőrizni lehessen) és azt, hogy hány meredek szakasz van az úton. Írja ki a program a meredek útszakaszok hosszát visszafelé is!

14.15. Delfinek

Állítsunk elő nyolcvanelemű, -5 és 3 közötti egész számokból álló listát! A számok egy úszó palackorrú delfin magasságát jelentik. A delfink ki-kiugrál a vízből, ilyenkor pozitív a magassága. Nulla a magasság, amikor a felszínen úszik, negatív, amikos a víz alatt tartózkodik. Írjunk programot, ami választ ad az alábbi kérdésekre!

a) Az út mekkora részét tette meg a delfin a vízben (felszínen vagy az alatt), illetve a víz alatt? A válaszok megadhatók törtszámként és százalékként is, de legfeljebb két tizedes jeggyel.
b) Víz alatt vagy viz felett volt-e többet a delfin? A vízfelszínen való úszás ezek egyikébe sem számít bele.
c) Hányszor törte át a vízfelszínt, azaz hányszor követ a listában negatív számot pozitív vagy fordítva?
d) Mély merülésnek számít, ha a delfin -4 vagy -5 mélységben van. Az út során hányszor merült mélyre? Figyeljünk arra, hogy például a 4, -2, -4, -5 3 útvonal csak egy mély merülésnek számít!
e) Milyen hosszú volt a leghosszabb kiugársa? Az út hányadik pontján kezdődött?

14.16. Olvassunk be számokat nulláig egy maximum 100 elemű listába!

Ezután írjuk ki a számokat a beolvasás sorrendjében, majd fordítva. Egy sorba tíz szám kerüljön vesszővel és szóközzel elválasztva, de az utolsó szám után ne legyen szóköz!

Írjuk ki az elemek közül a legkisebbet és a legnagyobbat az indexükkel együtt!

Olvassunk be egy számot és keressük meg az a tömbben!

A 14.15. és 14.16. feladatra tizenegyedik osztályban még vissza fogunk térni, amikor már konkrét algoritmusokkal fogsz megismerkedni.

Táblázatkezelés

Tárgymutató