1. hét / Prológus

In [2]:
```
import sys
print(sys.version)
```

```
3.10.9 | packaged by Anaconda, Inc. | (main, Mar  1 2023, 18:18:15) [MSC v.1916 64 b
it (AMD64)]
```

A mai órán a következőkről lesz szó:

- Rövid köszöntő, a filozófiánk (~10-15 perc)
- Bevezetés az algoritmuselméletbe (~30 perc)
- Aritmetikai műveletek, operátorok (~30 perc)
- Tömbműveletek (~15 perc)

1. hét / I. Köszöntés

Kedves Hallgató!

Mindenekelőtt nagyon sok szeretettel köszöntelek minden kollégám nevében, akikkel ezt a kurzust tartjuk. Nagyon nagyra értékeljük, hogy érdeklődsz a számítástudományok irányt, és megtiszteltél minket azzal, hogy feltehetően az első sor kódjaidat, és ezzel együtt első komolyabb Python-projekteted velünk együtt fogod elkészíteni! Emellett azt is reméljük, hogy lesz egy csepp szabad perced, hogy az elkövetkezendő bevezetőt elolvasd a kurzusról. Ez egy általános képet fog arról adni, hogy mire is számíts a félév során, milyen lesz a kurzus struktúrája, és mi az amit remélünk, hogy később elteszel majd belőle!

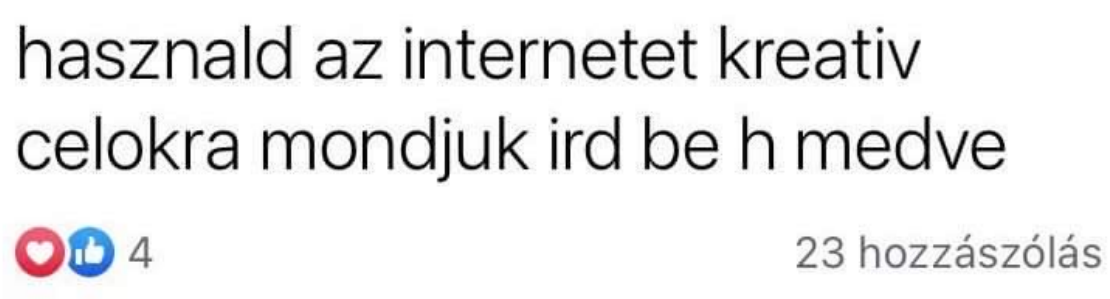
Programozni megtanulni korántsem egyszerű, de azért vagyunk, hogy segítsünk!

Először is elevenítsük fel a tárgy nevét: *Python mérnöki alkalmazásai*. Amit itt elsőkézben látni kell az nem más, hogy a félév végére szeretnénk megmutatni nektek számtalan érdekes - a mérnöki életben előforduló - példát, melyek megoldásához elengedhetetlen valamilyen programozói megoldás. Lehet szó egy egyszerű statika feladatmegoldó programról, vagy egy szenzoradatokat feldolgozó egységről, de akár beszélhetünk mozgásdetektálásról, képszegmentációról, emberi arcfelismerésről a gépi tanulás */machine learning/* forradalmi erejéig akármiről! *(És megjegyzem: minderről lesz is szó a félév során!)*. Tehát a **programozást, mint a mérnök gyakorlatban elterjedt eszközt** szeretnénk nektek bemutatni és nem azon lesz a hangsúly, hogy a számítástudomány és algoritmuselmélet, mint matematikai tudományágak mélyére evezzünk. A fő hangsúlyt a **gyakorlati problémamegoldásra** és az **általános imeretbővítésre** fektetjük.

A kritikus gondolkodás kialakításához elengedhetetlen, hogy minél többet kérdezzetek és minél intenzívebben keressétek a **miért**eket! Ezét mindenkit buzdítok arra, hogy **kérdezzen sokat** és még annál is többet! Akkor is ha akár jelentéktelennek, akár nagyon földtől elrugaszkodottnak tűnik a kérdés! Elvégre rossz kérdés nincs egy sem, legfeljebb nem tudunk válaszolni, *ha megkérdezitek a P=NP probléma megoldását*.

Természetesen egy - a legtöbbetek számára - teljesen idegen vízre evezni kihívásokkal teli, viszont valamennyien azért gondozzuk a tárgyat, hogy segítsük és támogassuk a fejlődéseteket! Bármilyen kérdés, probléma esetén nyugodtan keressetek meg minket privát elérhetőségeinken, közérdekű témában nyugodtan használjátok a tárgy hivatalos Teams csatornáját! A kurzus nagyjából úgy van összeállítva, hogy átlagosan **legfeljebb heti 4-5 órát** (ebbe beleértve az órai jelenlétet) **vegyen igénybe a kurzus teljesítése! Ha ennél lényegesen több időre van szükségetek, akkor ezt mindenképpen jelezzétek nekünk!**

Bátorítunk titeket az önálló tanulásra és szorgalmazzuk, hogy utánanézzetek mindannak ami érdekel!



Az önálló utánanézés kimondottan sok forrásból eredhet, végsősoron az mindennek a célja, hogy *tanulj, fejlődj, láss minél több érdekeset* és így **ismételd át a főbb koncepciókat**, mialatt **gyakorolsz, gyakorolsz és még többet gyakorolsz**. A kutatásod sok különböző formában végezheted:

- Legegyszerűbb, ha egy hibaüzenetet, problémás kérdést (angolul) bedobsz a Google keresőnek! *Nagyon kicsi a valószínűsége annak, hogy valaha olyan problémával fogsz találkozni, amivel előtted még soha senki!* Egyébként a leggyakrabban a Stack Overflow oldallal fogtok találkozni, egy példa: SO - Slicing in Python

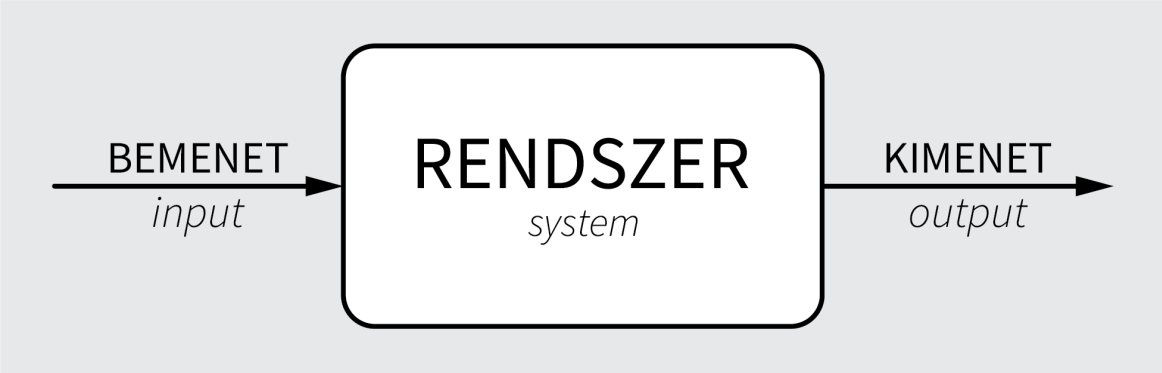
- Az AI térhódításával együtt megjelent a mindenki által jól ismert ChatGPT! Kisebb kódok írására (20-30 sor + kommentek) tökéletesen alkalmas, ráadásul az összes elterjedt programozási nyelvre elérhető nem csak Pythonra!

- Sajnos az utóbbi időkben egyre kevésbé elterjedt, de (különösen ritkábban használt packagek esetén) mindenképpen érdemes a *fejlesztői dokumentációkat* böngészni! Például itt elérhető a mai anyaghoz kapcsolódó Numpy Dokumentáció. Kötelességemnek érzem, hogy kiemeljem: hosszútávon és haladói programozói gyakorlatban a dokumentációk alapos áttekintése vezet majd el gyakorlatilag minden kérdés megválaszolásához, ugyanis ezek nagyon részletes leírásokat és számtalan példakódot tartalmaznak!

- Ez a legtöbb embernek újdonság varázsával hat, de az interneten teljesen igyenes hozzáférést garantálnak az oktatási anyagaikhoz a világ legrangosabb egyetemei (MIT, Caltech, Harvard...)! Akár hivatalos honlapjaikon (MIT Opencourse), vagy különböző kurzusportálokon. *Csak kezdjük el a Google keresőbe bepötyögni, hogy "MIT Python course..."*

1. hét / II. "Hello World"

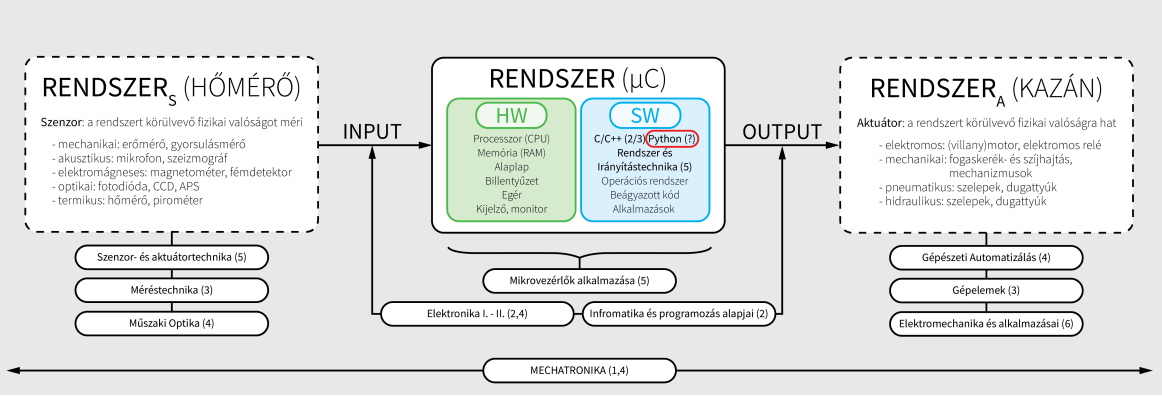
A bevezető szellemében kezdjük is az órát egy *nagyon fontos miérttel*: **Miért akarunk egyáltalán programozni?** Erre a kérdésre önmagában nagyon sok válasszal lehetne szolgálni, viszont megragadom az alkalmat, hogy egy *mechatronikai megközelítéssel* élve járjam körbe a kérdést. Elevenítsük fel a *Mechatronika alapjai* tárgyon felvázolt rendszermodellt:



Leegyszerűsítve a rendszerek azt a funkciót töltik be, hogy egy megadott **bemenet**re egy elvárt **kimenet**tel szolgáljanak. Akár értelmezhetjük ezt úgy is, mintha a rendszer egy egyértelmű hozzárendelés, egy **függvény** lenne a bemenet és kimenet között! Azt a *folyamat*ot, vagy *szabályok véges sorozatá*t, melynek célja jellemzően egy specifikus problémaosztály megoldása, **algoritmus**nak nevezzük. Minden algoritmus felépíthető a következő 3 elemből: *utasítás*ok (function), *elágazás*ok (if-statement) és *ciklus*ok (loop).

Példa Fűtésszezon van, Marika néni azt szeretné, hogy a lakásában a hőmérséklet 21 [°C]. Ezért egy hőmérő van telepítve.

- `INPUT` : A hőmérőből származó adat: $T \in [-273, \infty) \subseteq \mathbb{R}$
- `OUTPUT` : Menjen a fűtés, vagy sem: `true` / `false`
- `SYSTEM` : A program egy: $T \to \{\texttt{true}, \texttt{false}\}$ függvény lesz, azaz minden hőmérsékletértékre *egyértelműen eldönthető*, hogy kell-e mennie a fűtésnek avagy sem. Ha a hőmérséklet kisebb, mint 21 [°C], akkor menjen a fűtés, ha nagyobb, mint 21 [°C], akkor ne menjen a fűtés.
- `PROGRAM` : A hőmérő szenzor jelét rákötjük egy mikrovezérlőre. A mikrovezérlő folyamatosan figyeli a hőmérsékletet, ha `T>21` $\to$ `false` , ha pedig `T<21` $\to$ `true` jelet ad ki (0, vagy 1) a kazán/fűtésrendszer felé. A fűtésrendszer a `true` bement/input esetén be van kapcsolva, `false` esetén pedig ki van kapcsolva... de az már egy másik rendszer!



Most írjuk meg ehhez a kódot!

In [3]:
```
# INPUT: T - a hőmérőből származó adat
T = 12
# Ha a hőmérséklet kisebb, mint 21 °C:
if T < 21:
    print("Kapcsoljuk be a fűtést, mert Marika néni megfagy!")      # OUTPUT_1: {tru
# Egyébként minden más esetben:
else:
    print("Kapcsoljuk ki a fűtést, már elég meleg van.")            # OUTPUT_2: {fal
```

```
Kapcsoljuk be a fűtést, mert Marika néni megfagy!
```

1. hét / III. Változók és aritmetikai műveletek

Ahogy az előzőekben is láttuk *adatokat eltárolhatunk* **változók** segítségével. Minden változó az alábbiakkal rendelkezik:

- *Név*: célszerűen *beszédes neveket* megadni! Nem valószínű, hogy a `_6dwRζpwe3_` névből kitalálja valaki, hogy ezt a változót az Allee Intersparban egyszerre maximálisan megvásárolható zsemlék számára hoztam létre. A `maxZsemle` eggyel beszédesebb! Egyébként (Pythonban!) szinte bármilyen nevet adhatunk a változóknak, tartalmazhat akár görög, vagy japán karaktereket is, gyakorlatilag tetszőleges hosszúságúak is lehetnek. Kulcsszavakat (`def` , `if` , ...) nem adhatunk meg, nem kezdődhetnek számmal, illetve bizonyos fordítók a `!` használatát sem szeretik.

- *Érték* és *típus*: későbbiekben látni fogjuk de a *Python* nyelvben szinte akármit eltárolhatunk változókban: egész számokat, tizedes törteket, szöveget, de akár teljes grafikonokat, képeket, vagy komplett neurális háló modelleket is! Ennek megfelelően egy változónak lehet típusa: `int` , `float` , `double` , `string` ... Ezt a Python *lazán kezeli* (kicsit precízebben a Python ún. *dynamically typed language*), egy változó deklarálásakor nem kell megadnunk a típusát. Típuskonverzióknál viszont fontos, hogy észben tartsuk a típusokat.

- *Memóriaterület*: az egyes változók a *memóriában vannak eltárolva*, ezekhez úgynevezett *(memória)címek* tartoznak. A Pythonban nem is igazán, inkább C/C++ nyelvekben érdekes róluk beszélni...

Ha csak lehetséges (és ésszerű), akkor minden mennyiséghez rendeljünk egy változót! Nagyban megkönnyítik a kód átírását és az átlátását!

In [4]:
```
a = 5
print("Az a változó értéke:", a, "és típusa:", type(a))

b = -7.2
print("Az a változó értéke:", b, "és típusa:", type(b))

c = [1,2,3]
print("Az a változó értéke:", c, "és típusa:", type(c))

d = "szöveg"
print("Az a változó értéke:", d, "és típusa:", type(d))
```

```
Az a változó értéke: 5 és típusa: <class 'int'>
Az a változó értéke: -7.2 és típusa: <class 'float'>
Az a változó értéke: [1, 2, 3] és típusa: <class 'list'>
Az a változó értéke: szöveg és típusa: <class 'str'>
```

Aritmetikai (int, float) műveletek

Az első és legfontosabb úgynevezett `operator`, az *értékeadás operátor*, amely nem meglepő módon a Python leggyakrabban alkalmazott operátora. Segítségével változóknak értéket adhatunk. Általános szintaktikája: `<var1> = <val>`.

Fontos! Tegyük fel, hogy `a=3`. Amikor azt látjuk, hogy `a = a + 2`, akkor erre ne úgy gondoljunk, mint egy egyenlet, amit meg kell oldani `a`-ra, elvégre ennek ilyen formában semmi értelme nincs! Az előbbi kifejezés úgy fordítható le, mint *az* `a` *változó új értéke legyen a jelenlegi értékénél* `2` *-vel nagyobb szám!* Tehát az `a = a + 2` művelet után az `a`-ban eltárolt érték `5` lesz!

A gyorsabb kódolásért és a félreértések elkerülése végett a programozói gyakorlatban elterjed, *összevont alakot* is gyakran használjuk:

- `a = a + 2` → `a += 2`
- `a = a - 5` → `a -= 5`
- ...

In [5]:
```
# Értékadás operátor
a = 5
print("Az a változó értéke:",a)
print(f"Az a változó értéke: {a}")

# Érték növelése 2-vel
a = a + 2               # Az a változó új értéke legyen egyenlő az előző érték + 2-v
print(f"Az a változó új értéke",a)

# Kicsit kompaktabban
a += 2
print(f"Az a változó még újabb értéke",a)
```

```
Az a változó értéke: 5
Az a változó értéke: 5
Az a változó új értéke 7
Az a változó még újabb értéke 9
```

Tekintsük át az alapvető aritmetikai műveleteket: *összeadást, kivonást, szorzást, osztást*, illetve a *hatványozást* és a *maradékos osztást*!

In [6]:
```
# Hozzunk létre még egy változót!
b = 2

# Összeadás
c = a+b
print("Az a+b művelet eredménye:",c)        # 9 + 2 = 11
print(f"Az a+b művelet eredménye: {c}")

# Kivonás
c = a-b
print(f"Az a-b művelet eredménye: {c}")     # 9 - 2 = 7

# Szorzás
```

```
c = a*b
print("Az a*b művelet eredménye:",c)        # 9 * 2 = 18

# Osztás
c = a/b
print(f"Az a/b művelet eredménye: {c}")     # 9 / 2 = 4.5

# Hatványozás
c = a**b
print("Az a^b művelet eredménye:",c)        # 9^2 = 81

# Maradékos osztás
c = a%b
print(f"Az a%b művelet eredménye: {c}")     # 9 = 4*2 + 1

# Kicsit más számokra:
d = 5.2
print(f"Az a^b művelet eredménye: {d**b}")
print(f"Az a^b művelet eredménye: {round(d**b,2)}")
```

```
Az a+b művelet eredménye: 11
Az a+b művelet eredménye: 11
Az a-b művelet eredménye: 7
Az a*b művelet eredménye: 18
Az a/b művelet eredménye: 4.5
Az a^b művelet eredménye: 81
Az a%b művelet eredménye: 1
Az a^b művelet eredménye: 27.040000000000003
Az a^b művelet eredménye: 27.04
```

Összetettebb aritmetikai műveletek (Numpy bevezető)

Láthatjuk, hogy alapvetően sok műveletet támogat az alapvető Python könyvtár is, viszont ha szeretnénk *logaritmust*, vagy *trigonometrikus függvényeket* számolni (és *szeretnénk* majd a statika házi feladatban trigonometrikus függvényekkel dolgozni), akkor bővítenünk kell az eszköztárunkat! Ezekhez a műveletekhez a `numpy` könyvtárat kell segítségül hívnunk, melyet telepíthetünk (helyi számítógépünkre) az alábbi módon:

In [7]:
```
%pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\mbenc\anaconda3\lib\site-packages
(1.23.5)
Note: you may need to restart the kernel to use updated packages.
```

A `NumPy` egyébként egy az egyben tartalmazza a MATLAB programozási nyelv szinte összes függvényét. A MATLAB egy kiemelt (C-hez nagyon hasonló) programozási nyelv, különösen a mérnöki gyakorlatban, ugyanis nagyon széles matematikai eszköztárral rendelkezik. Viszont a MATLAB használatához komoly összegekért kell liszenszt vásárolni, ezért a `NumPy` készítői gyakorlatilag egy az egyben lemásolták az összes benne található függvényt és *ingyenesen elérhetővé tették mindenki számára*!

A könyvtár használatához *importnálni kell* a programunkba, ezt az `import` kulcsszóval tehetjük meg. Ezt elég egyszer megtenni, általában a *file legtetején szokott szerepelni az import lista*, ahol össze van gyűjtve minden szükséges, úgynevezett *dependency*, azaz olyan csomagok, amelyek elengedhetetlenek a program futásához.

In [8]:
```
import numpy as np      # Importáljuk a numpy könyvtárat a futtatókörnyezetünkbe
```

Innentől kezdve egy sokkal bővebb matematikai eszköztár áll rendelkezésünkre! Nézzünk is meg néhány példát abszolút értékekre, egészrész függvényekre és gyökvonásra!

In [9]:
```
x = -6

# Abszolút érték
print(f"Az x = {x} abszolút értéke {np.abs(x)}")

x = 5.5

# Alsó és felső egészrész
print(f"Az x = {x} alsó egészrésze {np.floor(x)}")
print(f"Az x = {x} felső egészrésze {np.ceil(x)}")

# Gyökvonás
print(f"Az x = {x} négyzetgyöke {np.sqrt(x)}")
print(f"Az x = {x} négyzetgyöke {np.power(x,(1/3))}")
```

```
Az x = -6 abszolút értéke 6
Az x = 5.5 alsó egészrésze 5.0
Az x = 5.5 felső egészrésze 6.0
Az x = 5.5 négyzetgyöke 2.345207879911715
Az x = 5.5 négyzetgyöke 1.7651741676630315
```

A következőkben tekintsük át a trigonometrikus függvényeket!

In [10]:
```
x = np.pi/6

# Trigonometrikus függvények
print(f"A sin(x) értéke: {np.sin(x)}")       # ~ 1/2
print(f"A cos(x) értéke: {np.cos(x)}")       # ~ sqrt(3)/2
print(f"A tan(x) értéke: {np.tan(x)}\n")     # ~ sqrt(3)/3

# Inverz trigonometrikus függvények
y = 0.5
print(f"Az arcsin(y) értéke: {np.arcsin(y)}")                        # ~ pi/6 radián

# Radián átváltása fokra
print(f"Az arcsin(y) értéke fokban: {np.rad2deg(np.arcsin(y))}")    # pi/6 [rad] = 3
```

```
A sin(x) értéke: 0.49999999999999994
A cos(x) értéke: 0.8660254037844387
A tan(x) értéke: 0.5773502691896257

Az arcsin(y) értéke: 0.5235987755982989
Az arcsin(y) értéke fokban: 30.000000000000004
```

Ezt követően nézzünk néhány példát logaritmikus függvényekre!

In [11]:
```
x = np.e**5
print(x)

# Természetes alapú logaritmus
print(f"Az ln(x) értéke: {np.log(x)}")      # 5

# 10-es alapú logaritmus
print(f"Az lg(x) értéke: {np.log10(x)}")    # ~ 2.1 - 2.2
```

```
148.41315910257657
Az ln(x) értéke: 5.0
Az lg(x) értéke: 2.171472409516259
```

Feladat: Próbáljuk meg kiszámolni `x` tetszőleges alapú logaritmusát!

Megjegyzés: amikor összeállítottam ezt az órai anyagot, fogalmam sem volt ennek a függvénynek a szintaktikájáról. Őszintén kicsit meg is lepődtem, hogy bonyolultabb elérni, mint a többit... **Nem attól lesz valaki jó programozó, hogy minden függvény szintaktikáját, bemeneti paramétereit, kimeneteit kívülről fújja, sőt! A véges agykapacitásunkat használjuk inkább arra, hogy fejlesszük a problémamegoldó képességünket**; mintsem arra, hogy értelmetlen mennyiségű lexikális tudást felhalmozzunk!

Megoldás: Beírtam a Google keresőbe, hogy *numpy logarithm with arbitrary base*, első találat: stackoverflow - NumPy: Logarithm with base n (2023. 07. 16.), *click*. Lentebb görgetünk, és kimásoljuk *u:dwitvliet* 2014-es válaszát:

In [12]:
```
# Tetszőleges alapú logaritmust kicsit bonyolultabb írni:
base = 5
print(f"A x={x} 5 alapú logaritmusa: {np.emath.logn(base,x)}")     # ~ 3
```

```
A x=148.41315910257657 5 alapú logaritmusa: 3.1066746727980594
```

1. hét / IV. Tömbök

Gyakran előfordul, hogy adatok egy *halmazát* szeretnénk együtt, egy *konténerben* belül kezelni. Például egy képzeljük el, hogy 15 percenként megmérjük a hőmérsékletet és ezeket az adatokat szeretnénk eltárolni egy *tömb*ben, és ezt követően szeretnénk ezt feldolgozni. Erre a Python nyelv lehetőséget kínál, sőt különböző *adatstruktúrákat* is biztosít, hogy mindig a céljainknak megfelelő konténerben gyűjtsük össze az adatainkat:

- **List** `[]`: legegyszerűbb adatstruktúra, az egyes adatok egymást követően (rendezetten) vannak elhelyezve ebben az adatstruktúrában. Tetszőleges adatot tartalmazhat (akár különböző adattípust is: számokat és szöveget). Elemei közvetlenül elérhetőek, azaz *indexelhető* a `<lista>` változó `i`. indexű elemére a `<list>[i]` szintaktikával hivatkozhatunk. Fontos megjegyezni, hogy a lista első elemének indexe a `0`!

- **Tuple** `()`: A listához hasonló adatstruktúra, ugyanúgy képes különböző típusú adatok rendezett tárolására, emellett ugyanúgy indexelhető is. Viszont a tuple *elemei nem módosíthatóak* (*immutable* object), ezért új elemeket hozzáadni nehézkes (de nem lehetetlen).

- **Set** `{}`: Az előzőekkel ellentétben ez egy *rendezetlen* adatstruktúra (azaz nem indexelhető), melyben *nem szerepelhet többször ugyanaz az elem*! Emiatt gyakran használják *duplicate element* (ismétlődő elemek) szűrésre, vagy tagság tesztelésére (azaz megvizsgálni, hogy egy adott elem benne van-e egy halmazban). Emellett értelmezhetőek rajta a *matematikai halmazműveletek*, mint: metszet, unió, különbség, szimmetrikus különbség...

- **Dictionary** `{}`: Ugyancsak *rendezetlen* adatstruktúra, mely alapvetően hozzárendelések és leképezések tárolására alkalmas. Például ha statika házi feladatot szeretnénk megoldani, akkor a létrehozhatjuk az `adatok` *dictionary-t, melyben az adatokhoz hozzárendeljük a számértékeket: `F` → `30` [kN], `a` → `200` [mm], `b` → `150` [mm] ...

Tömbműveletek

Ugyan magasfokú kényelmet garantálnak az egyes adattárolók, mint mindennek, ennek is megvannak a maga hátrányai. Tömbökben feldolgozni az adatot mindig egy lépéssel összetettebb feladat lesz, ezért az alábbi fontos feladatokkal mindenképpen meg kell ismerkednünk:

- Deklarálás és definiálás
- Tömbök címzése: indexelés
- Fontos függvények: len(), max(), min(), index()
- Tömbök rendezése

Példa: Dupla szakítás

Marika néni unokája, *Petike* az Anyagismeret laborban acél rudak szakítóvizsgálati mérését végezte el (L2-es labor). Az adatok kiértékeléshez Petike összegyűjtötte a `terheles[]` tömbbe a vizsgált terheléseket, viszont mialatt éppen nem figyelt, Petike most már ex-barátnője, *Suhanó Szabó* (közismertebb nevén *Taylor Swift*) adatszemetet csúsztatott a tömbbe.

Feladatok:

1. Írassuk ki a tömböt elemeit a hibás adatokkal együtt.
2. Írassuk ki külön a legelső és a legutolsó mérési eredményt is.
3. Tudjuk, hogy az utolsó mérési eredményt írta át *Suhanó Szabó*. Módosítsuk ezt az elemet úgy, hogy 270 [MPa] terhelésnek feleljen meg ez a pont!

In [13]:
```
terheles = [0.1, 0.2, 0.5, 2, "kétszázhúszfelett"]    # [MPa]

# A print() függvénnyel könnyen kiíratható
print(f"A szakítóvizsgálat során az alábbi terheléseket vizsgáltuk: {terheles}")

# Az elemek elérése - indexelés
print(f"Az első terhelés: {terheles[0]} [MPa]")         # Az indexelés mindig 0-val

# Elem módosítása
terheles[4] = 270
print(f"A szakítóvizsgálat helyes terhelései: {terheles}")
```

```
A szakítóvizsgálat során az alábbi terheléseket vizsgáltuk: [0.1, 0.2, 0.5, 2, 'kéts
zázhúszfelett']
Az első terhelés: 0.1 [MPa]
A szakítóvizsgálat helyes terhelései: [0.1, 0.2, 0.5, 2, 270]
```

Most, hogy kijavítottuk az adatokat, folytathatjuk az adatok feldolgozását!

1. Először számoljuk össze hány mérést végeztünk!
2. Petike utólag észrevette, hogy 7 mérést kell feltüntetni a jegyzőkönyvben. Adjunk hozzá két új mérést a tömbhöz! Az egyik legyen a 0.7, másik legyen az 5!
3. Keressük meg a maximumát az újonnan kapott tömbnek!
4. Rendezzük a tömböt!

In [14]:
```
# Tömbök hossza
print(f"Összesen {len(terheles)} mérést végzett Petike!")
```

```
# Elemek hozzáadása tömbökhöz
terheles.append(0.7)            # hozzáfűzés .append() metódussal
terheles += [5]                 # hozzáfűzés az '+' operátorral: listát kell hozzáfű
print(f"Az újonnan kapott terhelések tömbje: {terheles}")
```

```
Összesen 5 mérést végzett Petike!
Az újonnan kapott terhelések tömbje: [0.1, 0.2, 0.5, 2, 270, 0.7, 5]
```

In [20]:
```
# Maximum/minimum megkeresése
maxTerheles = max(terheles)
maxHely = terheles.index(maxTerheles)
print(f"A legnagyobb terhelés a {maxHely+1}. mérésnél volt, {maxTerheles} [MPa]")

# Tömbök rendezése
terheles.sort()
print(terheles)
```

```
A legnagyobb terhelés a 7. mérésnél volt, 270 [MPa]
[0.1, 0.2, 0.5, 0.7, 2, 5, 270]
```

Stringműveletek

A `string`-ek feldolgozása központi feladat a programozási gyakorlat során, bár hozzá kell tenni, hogy túlzottan komoly szükségünk nem lesz ezekre a félév során. Mindenesetre gyakorlatilag kötelező, hogy ezekről mindenki halljon és az alapvető metódusok létezésével tisztában legyen!

A `"string"` adattípus az `'` és `"` karakterek segítségével válnak elérhetővé, többsoros szöveg esetén a `""" <szöveg> """` szintaxist használjuk. A tömbökhöz hasonlóan indexelhetőek, speciális tulajdonságuk, hogy rendezettek (és legyenek is azok)! Felülírni viszont csak a `.replace()` metódussal lehet.

Fontos metódusok:

- `.capitalize()` - A string első karakterét nagybetűvé, minden más karakterét kisbetűvé írja át.
- `.count()` - Megszámolja egy adott karakter/substring előfordulását az adott stringben.
- `.find()` - Megkeres egy adott karaktert/substringet és visszatér a helyével.
- `.lower()` - A teljes stringet kisbetűvel írja át.
- `.replace()` - Kicseréli a string egy adott részét egy másik stringre.
- `.split()` - Szétszed egy stringet megadott elválasztó karakter alapján.
- `.upper()` - A teljes stringet nagybetűvel írja át.

In [16]:
```
# Definiáljuk a kezdeti stringet
szoveg = "kübekcsakra"
print(szoveg)
```

```
kübekcsakra
```

Indexelés stringekben

A tömbökhöz hasonlóan, a stringek is indexelhetőek, ami ugyanúgy `0`-val kezdődik. Amit érdemes itt áttekinteni (révén a stringeknél az egyik leglátványosabb, de a rendes tömbökre is működik) az az úgynevezett *slicing*. A tömbökből kiválaszthatok adott részeket az alábbi

módon: `tomb[<kezd> : <veg> : <lepes>]`. Azaz a `<kezd>` indextől a `<veg>` indexig (és azt már nem beleértve) kiválaszthatjuk adott `<lepes>` lépésközzel az egyes betűket:

- `"nukleárizs"[:]` → `"nukleárizs"`
- `"nukleárizs"[1:4]` → `"ukl"`
- `"nukleárizs"[0:6:2]` → `"nke"`

Ezzel a módszerrel egyébként könnyen megfordíthatjuk a stringeket!

- `"alma"[::-1]` → `"amla"`

In [38]:
```
# Adott karakter elérése
print(szoveg[0])

# Slicing
print(szoveg[1:3])
print(szoveg[:])
print(szoveg[::2])

# String megfordítása
print(szoveg[::-1])

# Felülírni viszont nem lehet!
# szoveg[1] = "k"
print(szoveg.replace("k", "K"))
print(szoveg.replace("k", "K", 1))
```

```
k
üb
kübekcsakra
kbkska
arkasckebük
KübeKcsaKra
Kübekcsakra
```

1.hét / Epilógus

Szorgalmi feladatok

Az órai anyaghoz kapcsolódó gyakorló feladatok a Teamsen elérhetőek a *PMA_gyak1.pdf* dokumentumban! Ahhoz, hogy minél jobban rögzüljenek az órán tanultak, feltétlen javaslom, hogy gyakoroljatok amennyit csak időtök engedi! Némi útmutatás a feladatgyűjteményhez: három csoportba vannak sorolva a feladatok:

- **Bevezető feladatok**: rövid (2-3 sor kód) és nagyon egyszerű példák, melyek az órai anyag áttekintését segítik elő. Minden különálló anyagrészhez (aritmetikai műveletek, összetettebb aritmetikai műveletek (numpy), tömbműveletek...) tartozik egy bevezető feladat. Körülbelül *2-3 perc* időt vesznek igénybe egyenként, és tartalmaznak *megoldásokat*!

- **Gyakorló feladatok**: az a bevezető feladatoknál egy lépéssel összetettebb feladatok, melyekben több különböző anyagrészt is fel kell használni. Ha valaki ezeket a feladatokat megoldja, egy komolyabb megértést tud szerezni az anyagról! Ezek a feladatok is tartalmaznak *megoldásokat*!

- **Ajánlott feladatok**: komolyabb kihívást jelentő feladatok, melyek jellemzően túl is mutatnak az órai anyagon. Megoldásukhoz több ötletelés, internetes kutakodás szükséges. Ezekhez *nem tartozik megoldás*, viszont ha valaki próbálkozott megoldani a feladatokat, akkor kérés esetén nagyon szívesen bemutatom a saját megoldásaimat!

Kérlek benneteket, hogy különösen az első hetekben a **bevezető példákat mindenképpen nézzétek át óráról órára**, különben könnyen lemaradhattok és a következő órákon elveszve érezhetitek magatokat. Másik jó tanács, hogy *használjátok a megoldókulcsot*, de ne intézzétek el annyival a problémát, hogy *"hát értem mi van oda írva, tehát kész a feladat"*! A **kódot mindenképpen gépeljétek be és próbáljátok ki**, akár több bemenetre is!

Hasznos anyagok:

- Dokumentációk
  - Python hivatalos dokumentációja: https://docs.python.org/3/
  - PEP 8 Style Guide for Python Code - Melyek a jó és rossz programozási praktikák
  - NumPy hivatalos dokumentációja: https://numpy.org/doc/1.25/
- Tankönyvek
  - Dive Into Python 3
  - Dive into Deep Learning - Interaktív tankönyv Deep Learninghez
  - Fluent Python: Clear, Concise, and Effective Programming by Luciano Ramalho - Haladóbb szemléletű Python programozás
- Útmutatók
  - The Official Python Tutorial - Self-explanatory?
  - Foglalt Keyword lista - Ezeket ne használd változónévnek!
  - Codecademy - Interaktív (fizetős) online tutorial
  - CheckIO - Tanulj Pythont játékfejlesztésen kersztül
- Competitive Programming
  - Codewars
  - CodeForces

Elérhetőség

Bármilyen kérdés, kérés vagy probléma esetén keressetek minket az alábbi elérhetőségeken:

- Monori Bence - m.bence02@outlook.hu
- Wenesz Dominik - weneszdominik@gmail.com

Illetve anonim üzenetküldésre is lehetőséget biztosítunk, ezt az alábbi linken tudjátok elérni: https://forms.gle/6VtGvhja3gq6CTT66