Programozás 2 Jegyzet  
Szemán László

[Programozási nyelvek osztályozása (1.1) 3](#_Toc181955480)

[Az objektum-orientált paradigma alapfogalmai és jellemzői (1.2) 3](#_Toc181955481)

[Objektum viselkedése és élettartama (1.3) 4](#_Toc181955482)

[Osztályattribútum és Osztálymetódus (1.4) 5](#_Toc181955483)

[Aszimmetria az Öröklődésben (1.5) 6](#_Toc181955484)

[Absztrakt Osztályok és Metódusok (1.6) 9](#_Toc181955485)

[Paraméterezett Osztályok (1.7) 11](#_Toc181955486)

[Példányosítás Pythonban (1.8) 13](#_Toc181955487)

[Az információrejtés Pythonban (1.9) 14](#_Toc181955488)

[Konstruktorok és Attribútumok Pythonban (2.1) 14](#_Toc181955489)

[Pont Minősítés (2.2) 15](#_Toc181955490)

[Osztály és példány kapcsolat (2.3) 15](#_Toc181955491)

[Metódusok (2.4) 15](#_Toc181955492)

[Öröklődés Pythonban (2.5) 17](#_Toc181955493)

[Típus és Osztály a Pythonban (2.6) 18](#_Toc181955494)

[Objektumorientált Programozás (OOP) (2.7) 18](#_Toc181955495)

[Objektumorientáltság célja és előnyei (2.8) 19](#_Toc181955496)

[Pass Kulcsszó (2.9) 20](#_Toc181955497)

[Dir() metódus (2.10) 20](#_Toc181955498)

[Konstruktorok és Attribútumok Pythonban (3.1) 20](#_Toc181955499)

[Osztály és példány kapcsolat (3.2) 21](#_Toc181955500)

[Metódusok (3.3) 22](#_Toc181955501)

[Osztály definiálása – Python Standard Metódusok (3.4) 22](#_Toc181955502)

[Mi az operátor túlterhelés? (3.5) 25](#_Toc181955503)

[\_\_str\_\_ vs \_\_repr\_\_ Pythonban (3.6) 26](#_Toc181955504)

[\_\_radd\_\_ metódus (3.7) 27](#_Toc181955505)

[Osztály és példány kapcsolata (3.8) 27](#_Toc181955506)

[Objektumok Hatóköre (3.9) 28](#_Toc181955507)

[Öröklődés: Erő és Komplexitás (4.1) 29](#_Toc181955508)

[A Beépített Típusok is Objektumok (4.2) 29](#_Toc181955509)

[Miért Használjuk a super()-t? (4.3) 30](#_Toc181955510)

[Öröklődés áttekintése (4.4) 30](#_Toc181955511)

[Absztrakt Osztályok a Pythonban (4.5) 32](#_Toc181955512)

[Belső Osztályok (Inner Classes) Pythonban (4.6) 33](#_Toc181955513)

[Miért Fontos a Tesztelés? (5.1) 33](#_Toc181955514)

[Hibák Típusai (5.2) 34](#_Toc181955515)

[Tesztelési Típusok (5.3) 36](#_Toc181955516)

[Mit Teszteljünk? (5.4) 36](#_Toc181955517)

[Doctest Modul a Pythonban (5.5) 38](#_Toc181955518)

[Unittest Modul a Pythonban (5.6) 39](#_Toc181955519)

[Mintaillesztés (6.1) 40](#_Toc181955520)

[Brute force algoritmus (6.2) 40](#_Toc181955521)

[**Knuth–Morris–Pratt (KMP) algoritmus (6.3)** 41](#_Toc181955522)

[Boyer-Moore algoritmus (6.4) 41](#_Toc181955523)

[**Rabin–Karp algoritmus (6.5)** 41](#_Toc181955524)

[A re modul a Pythonban (6.6) 42](#_Toc181955525)

[Re modul függvényeinek összegzése (6.7) 43](#_Toc181955526)

# Programozási nyelvek osztályozása (1.1)

A programozási nyelvek több kategóriába sorolhatók, főként az alapján, hogyan kezelik a problémamegoldást és milyen programozási szemléletet követnek:

**1. Imperatív nyelvek**

* **Jellemzői**: Az imperatív, azaz parancsoló nyelvek parancs-orientáltak, vagyis lépésről lépésre adnak utasításokat a számítógépnek.
* **Algoritmikus jellegűek**: A program szekvenciákból, utasításokból áll, amelyek algoritmust definiálnak. A processzor az algoritmus lépéseit végrehajtva működik.
* **Változók használata**: A változók a legfontosabb eszközök ezekben a nyelvekben.
* **Kapcsolat a Neumann-architektúrával**: Az imperatív nyelvek általában szorosan kötődnek a Neumann-féle architektúrához, ahol a memória és a végrehajtási egységek elkülönülnek.
* **Alcsoportjai**:
  + *Eljárásorientált nyelvek*: pl. C, Pascal.
  + *Objektumorientált nyelvek*: pl. Python, Java, C++.

**2. Deklaratív nyelvek**

* **Jellemzői**: A deklaratív nyelvek nem algoritmikusak, vagyis a programozó itt nem ad meg részletes algoritmust, csak a megoldandó problémát írja le.
* **Függetlenség a Neumann-architektúrától**: Ezek a nyelvek kevésbé kötődnek a hagyományos számítógép-architektúrához.
* **Problémamegoldás**: A deklaratív nyelvekben a nyelv implementációja tartalmazza a problémamegoldás módját, nem a programozó specifikálja azt lépésről lépésre.
* **Memóriaműveletek korlátozottsága**: A programozó csak korlátozottan fér hozzá a memóriakezeléshez.
* **Alcsoportjai**:
  + *Funkcionális nyelvek*: pl. Haskell, Lisp.
  + *Logikai nyelvek*: pl. Prolog.

**3. Egyéb nyelvek**

* Ide tartoznak a programozási nyelvek, amelyek nem illenek tisztán sem az imperatív, sem a deklaratív kategóriába, vagy több paradigmát is támogatnak egyszerre (pl. Python, amely mind az imperatív, mind a funkcionális paradigmákat támogatja).

# Az objektum-orientált paradigma alapfogalmai és jellemzői (1.2)

1. **Adat- és eljárásmodell egyesítése**:
   * Az OO nyelvek az adatokat és a hozzájuk kapcsolódó műveleteket (eljárásmodellt) egy objektumban egyesítik, ami közelebb hozza a programozást a valósághoz. Ez a megközelítés segíti az újrafelhasználhatóságot és a modularitást.
   * Az OO szemlélet alapvetően imperatív eszközrendszerre épül, tehát algoritmusokat kell kódolni a működés biztosítására.
2. **Objektum**:
   * Az objektum az eljárásorientált nyelvek változó fogalmának kiterjesztett verziója, amely a belső állapotot és a viselkedést is magában foglalja.
   * Az objektum jellemzői:
     + **Attribútum**: Az objektum adatrésze, amely összetett adatszerkezeteket tartalmazhat. Az attribútumok az objektum statikus részét alkotják, és minden attribútumhoz egy egyedi, jól definiált memóriacím tartozik, ahol az értéke tárolódik.
     + **Állapot**: Az objektum állapota az attribútumok aktuális értékeinek kombinációját jelenti, amit bitsorozatok reprezentálnak a memóriában.
3. **Metódus (method)**:
   * A metódus az objektum viselkedését leíró eljárás vagy függvény. Nyelvi szinten a metódusok határozzák meg, hogyan reagál az objektum különböző helyzetekre, és hogyan manipulálja az attribútumait.
4. **Azonosság**:
   * Minden objektum egyedi azonossággal rendelkezik, amit egy egyedi azonosító (OID, object identifier) biztosít. Ez az azonosító minden objektumot megkülönböztet minden más objektumtól, még programok között is.
   * Az OID hasonló a változókhoz társított nevekhez, azonban az OID valóban egyedi minden egyes objektum számára, míg a változónév csak adott hatókörön belül egyedi.

# Objektum viselkedése és élettartama (1.3)

1. **Objektum viselkedése**:
   * Az objektum állapota idővel változhat, ami a metódusokon keresztül történik.
   * A metódusok csoportjai:
     + *Lekérdező metódusok*: Az objektum aktuális állapotát lekérdező funkciók, amelyek visszaadják az attribútumok értékeit.
     + *Módosító metódusok*: Az objektum állapotának módosítására szolgáló funkciók, amelyek beállítják vagy megváltoztatják az attribútumok értékeit.
2. **Objektumok élettartama**:
   * Az objektumot létre kell hozni, és a létrejötte után addig marad fenn, amíg el nem törlődik.
   * Az objektum megszüntetése történhet automatikusan a nyelvi rendszer (pl. Pythonban a szemétgyűjtés) által, vagy a programozó explicit törlési utasítása révén.
   * Az objektum egyedi azonosítója (OID) minden szinten él, az objektum fennállásának teljes ideje alatt létezik.

**Osztályok és példányok**

1. **Az osztály fogalma**:
   * Az osztály absztrakt eszköz, amely az eljárásorientált nyelvek típusfogalmának általánosítása, gyakran típusként hivatkozunk rá.
   * Egy osztály azonos attribútumokkal és metódusokkal rendelkező objektumok csoportja.
   * Az objektumok mindig egy osztályhoz kapcsolódnak, és az osztályból származtathatóak.
2. **Példány (instance)**:
   * Az osztály egy konkrét objektuma, amely a példányosítás során jön létre.
   * Minden objektum tudja, hogy melyik osztály példánya, így minden példány ugyanazokkal az attribútumokkal és metódusokkal rendelkezik, mint az osztálya.
   * A metódusokat mindig egy konkrét példányon hajthatjuk végre, és ezek az adott példányra vonatkoznak.
   * Minden példány létrehozása során új adatszerkezet jön létre a memóriában, azonban a metódusok nem duplikálódnak, azokat az osztály szintjén tárolja a rendszer.

# Osztályattribútum és Osztálymetódus (1.4)

1. **Osztályattribútumok**:
   * Az osztályattribútumok nem az egyes példányok állapotát és viselkedését jellemzik, hanem magához az osztályhoz tartoznak.
   * Példa: Egy osztály attribútuma lehet például, hogy hány példány létezik az osztályból (ez az osztály "kiterjedését" mutatja).
   * Az osztályattribútumok nem többszöröződnek, tehát az osztály minden példányához csak egy osztályattribútum tartozik.
2. **Osztálymetódusok**:
   * Az osztálymetódusok olyan metódusok, amelyek szintén nem az egyes objektumok viselkedését határozzák meg, hanem az osztályhoz kapcsolódnak.
   * Ezen metódusok használata az osztály szintjén történik, és nem függ a példányok állapotától.

**Osztály és példányosítás**

1. **Osztály létrehozása**:
   * Először létre kell hozni egy osztályt, amely meghatározza, hogy a hozzá tartozó objektumoknak milyen attribútumaik és metódusaik lesznek.
   * Az osztály definiálása után lehet az osztályhoz tartozó objektumokat létrehozni.
2. **Példányosítás**:
   * Az osztály példányosítása után az osztály egyedi példányairól beszélhetünk.
   * Minden egyes példány saját állapottal és viselkedéssel rendelkezik, de az osztályattribútumok és osztálymetódusok közösek a példányok között.

**Öröklődés (Inheritance)**

1. **Öröklődés alapjai**:
   * Az öröklődés lehetővé teszi az osztályok közötti kapcsolatokat, amelyeket aszimmetrikus viszonyok jellemeznek.
   * Az öröklődés révén egy osztály (szuperosztály) "származtatott" osztályokat (alosztályokat) hozhat létre.
   * A szuperosztályból származó alosztályok automatikusan megkapják a szuperosztály attribútumait és metódusait.
2. **Szuperosztály és alosztály**:
   * **Szuperosztály (Superclass)**: Az az osztály, amelyet más osztályok örökölnek.
   * **Alosztály (Subclass)**: Az a származtatott osztály, amely az öröklés révén hozzáfér a szuperosztály attribútumaihoz és metódusaihoz.
   * Az alosztály átveszi azokat az attribútumokat és metódusokat, amelyek nem lettek eltávolítva vagy elrejtve a szuperosztályban.
3. **Alosztály kiterjesztése**:
   * Az alosztály nemcsak örökli a szuperosztály összes jellemzőjét, hanem további lehetőségei is vannak:
     + Új attribútumokat és metódusokat vezethet be.
     + Megváltoztathatja (újraimplementálhatja) a szuperosztályban lévő metódusokat.
     + Eltávolíthatja a szuperosztályban lévő attribútumokat és metódusokat.
     + A láthatósági szabályokat módosíthatja, például felfüggesztheti vagy újraértelmezheti azokat.
     + Átnevezheti az attribútumokat vagy metódusokat.
     + Duplikálhat attribútumokat vagy metódusokat.

# Aszimmetria az Öröklődésben (1.5)

1. **Szuperosztály és alosztály közötti aszimmetria**:
   * A szuperosztály **nem látja** és **nem manipulálhatja** az alosztályokat.
   * Az alosztály viszont **teljes mértékben látja** a szuperosztályt, tehát hozzáférhet annak attribútumaihoz és metódusaihoz.
   * **Objektum kapcsolat**: Ha egy objektum egy alosztály példánya, akkor az automatikusan a szuperosztály objektumának is tekinthető. Fordítva, egy szuperosztály objektuma **nem tekinthető az alosztály objektumának**.
2. **Példa**:
   * Ha egy szuperosztály egy példányát használjuk, akkor annak bármely alosztályának példánya is alkalmazható a helyére, mivel az alosztály örökli a szuperosztály viselkedését.
   * Fordítva, egy alosztály példányát nem lehet szuperosztályként használni, mivel az alosztály nem biztos, hogy tartalmazza a szuperosztály összes attribútumát és metódusát.

**Öröklődés típusai**

1. **Szuperosztály és alosztály kapcsolata**:
   * Egy osztályból tetszőleges számú alosztály származtatható.
   * **Egyszeres öröklődés**: Egy alosztály csak egy szuperosztálytól örököl.
   * **Többszörös öröklődés**: Az alosztály több szuperosztálytól is örököl.
2. **Problémák a többszörös öröklődésben**:
   * **Névütközés**: Ha két szuperosztályban ugyanaz a nevű attribútum vagy metódus található, akkor névütközés léphet fel. Ezt a problémát a rendszernek kell kezelnie, de hogy miként történik, az a programozási nyelv implementációjától függ.

**Öröklődési Hierarchia**

1. **Hierarchikus felépítés**:
   * Az alosztályok más alosztályokat is származtathatnak, így egy **öröklési hierarchia** jön létre.
   * **Egyszeres öröklődés esetén**: Az öröklési hierarchia fa struktúrát alkot.
   * **Többszörös öröklődés esetén**: Az öröklési kapcsolatokat egy **aciklikus gráf** formájában ábrázolhatjuk, ahol egy osztály több szuperosztályból is örökölhet.

**Bezárás (Encapsulation)**

1. **Definíció és jelentőség**:
   * A bezárás (encapsulation) az objektumorientált (OO) nyelvek egyik alapfogalma, amely gyakran félreértett. Ez a fogalom az objektumok adatainak és viselkedésüknek a védelmét szolgálja, és biztosítja, hogy azok csak meghatározott módon érhetők el.
   * Az eljárásorientált programozásban alkalmazott hatáskör fogalmának OO-ban való kiterjesztése.
2. **Adattípus és interfész**:
   * Az osztály egy **absztrakt adattípus**, amelynek **interfész** és **implementációs** része van.
   * Az osztály objektumaihoz **csak az interfészen keresztül** férhetünk hozzá, az implementációhoz nem, ezzel biztosítva a **korlátozott hozzáférést**.
   * Ez a **"információrejtés"** elve, ami lehetővé teszi, hogy az osztályok belső működése el legyen rejtve a külvilág elől.
3. **Nyilvános és privát rész**:
   * Az osztály metódusait és attribútumait két fő részre osztják:
     + **Nyilvános rész**: Olyan elemek, amelyek bárki számára hozzáférhetők (pl. kliens osztályok).
     + **Privát rész**: Olyan elemek, amelyek kívülről nem láthatóak és nem érhetők el, csak az osztályon belül.
4. **Szabályozott hozzáférés**:
   * Az objektumok elérhetősége és láthatósága szigorúan szabályozott, a programozónak lehetősége van meghatározni, hogy mi látható és kinek.

**Polimorfizmus**

1. **Objektum polimorfizmus**:
   * Egy objektum saját osztályán kívül is szerepelhet olyan helyzetekben, ahol annak szuperosztályának példánya is megjelenhet.
   * Az objektumok polimorfikus viselkedése az öröklődési hierarchiában azáltal valósul meg, hogy minden objektum egyben az összes ősosztály példányaként is viselkedhet.
2. **Metódus polimorfizmus (Overriding)**:
   * Az alosztályok lehetőséget adnak arra, hogy egy örökölt metódust **újraimplementáljanak** (override), miközben a metódus **specifikációja** változatlan marad.
   * Példa: Az alosztályban új implementációval rendelkező **terület()** vagy **kerület()** metódusok, amelyek felülírják az alosztályban örökölt metódusokat.

**Kötés (Binding)**

1. **Kötési típusok**:
   * A polimorfizmus egyik aspektusa a metódusok és függvények különböző implementációihoz való hozzárendelés.
     + **Statikus (korai) kötés**: A kód fordítási időben kapcsolódik a metódushoz (az OO rendszerek többsége ilyen típusú).
     + **Dinamikus (késői) kötés**: A kód futásidőben kapcsolódik a metódushoz, lehetőséget adva arra, hogy az aktuális objektum típusától függően más-más metódus implementáció legyen végrehajtva.
2. **Kötési példák**:
   * **Statikus kötés**: A metódusok előre, fordítási időben vannak hozzárendelve a megfelelő osztályokhoz.
   * **Dinamikus kötés**: A futásidőben az aktuális objektum osztályának megfelelő kódot kapcsolja a rendszer, így az objektum típusa határozza meg a végrehajtott kódot.

**Üzenet (Message)**

1. **Smalltalk filozófia**:
   * Az üzenetküldés a Smalltalk programozási nyelv alapvető filozófiája. Az objektumokat üzenetek segítségével kezeljük, ami azt jelenti, hogy ha egy objektumtól valamit kérünk, akkor üzenetet küldünk neki.
   * Az üzenetküldés folyamata az alábbiak szerint zajlik:
     + Az objektum **megkapja az üzenetet**.
     + A programozó nem tudja, hogy mi történik pontosan a háttérben, csak azt, hogy az objektum megfelelően válaszol.
     + Az objektum **válaszol** az üzenetre, és végrehajtja a kérésnek megfelelő műveletet.

# Absztrakt Osztályok és Metódusok (1.6)

1. **Absztrakt osztályok**:
   * Az **absztrakt osztály** olyan osztály, amely nem tartalmaz példányokat, tehát nem lehet példányosítani.
   * Célja, hogy más osztályok örököljék, és ezekből az örökölt osztályokból **konkrét, példányosítható osztályok** származzanak.
   * Az absztrakt osztályokban nem találhatóak teljes implementációk, és gyakran **absztrakt metódusokat** is tartalmaznak.
2. **Absztrakt metódusok**:
   * Az **absztrakt metódusok** olyan metódusok, amelyek csak a **specifikációjukat** tartalmazzák, de nincs hozzájuk implementáció.
   * A metódusok implementációját az őket öröklő osztályokban kell megadni. Az absztrakt metódusok arra szolgálnak, hogy meghatározzák az elvárt viselkedést, amelyet az alosztályoknak implementálniuk kell.
3. **Absztrakció szerepe**:
   * Az absztrakció célja, hogy elválassza a **megjelenítés** és a **megvalósítás** részét, így egy magasabb szintű, könnyebben kezelhető és bővíthető rendszert eredményezzen.
   * A rendszerfejlesztési ciklusban az absztrakció segíthet az **alkalmazás tervezésében** és az **implementáció megvalósításában** is.

**Konténer Osztályok**

1. **Konténer osztályok**:
   * A konténer osztályok olyan **adatszerkezetek**, amelyek **objektumokat** tartalmaznak, például **tömbök, láncolt listák, vermek** és **sorok**.
   * Ezek az osztályok lehetővé teszik, hogy különböző típusú adatokat tároljunk egyetlen struktúrában, és az adatok kezelésére különböző műveleteket végezhessünk.
   * Néhány programozási nyelvben a konténer osztályokat a nyelv biztosítja (pl. Java Collections Framework), más nyelvekben a programozónak kell ezeket az adatstruktúrákat megvalósítania.
2. **Példák**:
   * **Tömb**: Fix méretű adatstruktúra, amely azonos típusú elemeket tárol.
   * **Láncolt lista**: Dinamikusan bővülő adatstruktúra, amely egymástól független elemeket tartalmaz, minden elem egy következő elemre mutat.
   * **Verem (stack)**: LIFO (Last In, First Out) alapú adatstruktúra, ahol az utolsóként betett elem kerül ki először.
   * **Sor (queue)**: FIFO (First In, First Out) alapú adatstruktúra, ahol az elsőként betett elem kerül ki először.

**Kollekciók**

1. **Kollekciók az objektum-orientált adatbázisokban**:
   * Az objektum-orientált adatbázisokban a konténer osztályok helyett a **kollekciók** fogalmát használják.
   * A kollekciók azokat az objektumokat jelentik, amelyek egyetlen struktúrában tárolhatók és rendszerezhetők, és az azokkal kapcsolatos műveletek könnyen kezelhetők.
2. **Iterátor**:
   * Az **iterátor** olyan eszközt biztosít, amely lehetővé teszi, hogy az elemeket egyenként végigiteráljuk a kollekciókon. Az iterátor segít abban, hogy anélkül férhessünk hozzá a kollekció elemeihez, hogy közvetlenül kezelni kellene az adatstruktúra implementációját.

# Paraméterezett Osztályok (1.7)

1. **Paraméterezett osztályok (Template-ek)**:
   * A paraméterezett osztályok olyan osztályok, amelyek **tipust paraméterként** fogadnak el, és ezáltal lehetővé teszik, hogy az osztályokat különböző típusokkal dolgoztassuk.
   * A **C++** terminológiájában ezeket **template-eknek** nevezik. Ada nyelven a megfelelő kifejezés a **generikus osztályok**.
   * A paraméterezett osztályok segítségével **újrahasznosítható, rugalmas kódot** hozhatunk létre, amely különböző adattípusokkal működik, anélkül hogy az osztályokat minden esetben újra kellene definiálni.

**Objektumok Élettartama**

1. **Objektumok élettartama és példányosítás**:
   * Az objektumok **példányosítása** mindig explicit tevékenység, tehát az objektumot a programozó hozza létre.
   * A különböző nyelvek kezelhetik az objektumok **élettartamát** különböző módon:
     + **Explicit törlés**: Néhány nyelv (pl. **C++**) esetén a programozónak kell gondoskodnia az objektumok **törléséről** is, ha már nincs rájuk szükség.
     + **Automatikus törlés (Garbage Collection)**: Más nyelvek, például **Java** vagy **Python**, alkalmaznak automatikus **garbage collection** mechanizmust, amely az objektumokat **aszerint törli**, hogy mikor már nincs rájuk hivatkozás, és a **memória felszabadul**.
     + Az automatikus törlés segíti a programozót, hogy ne kelljen manuálisan törölnie az objektumokat, ami kényelmesebbé teszi a programozást és elkerüli a memória szivárgásokat.

**Egységesség a Nyelvekben**

1. **Tisztán OO és Hibrid Nyelvek**:
   * **Tisztán OO nyelvek**: Olyan nyelvek, amelyek csak **objektum-orientált** eszközöket tartalmaznak, és nem használnak más paradigmákat. Minden elem osztály és objektum, például **Smalltalk** és **Eiffel**.
   * A tisztán OO nyelvekben az **osztályhierarchia** egyetlen szinten jelenik meg, és a programozás lényege az osztályok definiálása és azok hierarchiájába való elhelyezése.
   * **Hibrid nyelvek**: Olyan nyelvek, amelyek **objektum-orientált** eszközöket tartalmaznak, de más paradigmák eszközeit (pl. eljárás-orientált, funkcionális, logikai) is használják. Ilyenek például a **C++**, **Object Pascal**, és **OO COBOL**.
     + A hibrid nyelvek lehetővé teszik, hogy a programozók **objektum-orientált** módon programozzanak, de ugyanakkor más típusú kódokat (pl. eljárásokat) is használjanak. Az osztályok és az objektumok nem minden esetben vannak jelen, így **több osztályhierarchia** is létezhet.

**Terminológia**

1. **Objektum-alapú nyelvek**:
   * Az **objektum-alapú nyelvek** (pl. **Ada**) olyan nyelvek, amelyek tartalmazzák az **objektumokat** és a **bezárást**, de nem tartalmaznak osztályokat vagy öröklődést.
2. **Osztály-alapú nyelvek**:
   * Az **osztály-alapú nyelvek** (pl. **CLU**) olyan nyelvek, amelyek tartalmaznak **osztályokat**, **bezárást** és **objektumokat**, de nem rendelkeznek öröklődési mechanizmussal.
3. **Objektum-orientált nyelvek**:
   * Az **objektum-orientált nyelvek** (pl. **Java**, **C++**) minden fontos OO fogalmat tartalmaznak, így az **osztályokat**, **objektumokat**, **öröklődést** és **bezárást** is.
   * Ezek a nyelvek **imperatív** jellegűek, és a programok általában **fordítóprogramokkal** működnek.
4. **Speciális OO nyelvek**:
   * Léteznek olyan **speciális OO nyelvek** is, amelyek nem tartalmaznak **osztály fogalmat**, de más OO alapú eszközöket tartalmaznak, például a **Self** vagy a **Eiffel** egyes verziói.

**Objektumorientáltság és Python**

1. **Objektumorientált megközelítés Pythonban**:
   * A Pythonban **MINDEN** objektum, így nemcsak az osztályok példányai, hanem a **beépített típusok** is (pl. **egészek**, **stringek**, **szótárak**, stb.) objektumok, melyek osztályokból jönnek létre.
   * Az objektumok lehetnek:
     + **Beépített osztályok**: mint az alapvető adatstruktúrák (pl. listák, szótárak).
     + **Felhasználó által definiált osztályok**: melyek a programozó által meghatározott osztályok példányosításával jönnek létre.
2. **Python osztályok**:
   * A Pythonban osztályokat bárhol definiálhatunk a programban, nincs speciális helyhez kötve.
   * Alapértelmezés szerint minden attribútum és metódus **publikus**, tehát bárki hozzáférhet hozzájuk.
   * Pythonban **privát attribútumokat és metódusokat** is definiálhatunk, de a nyelv nem biztosít formális védelmet ezekhez (a privát attribútumokat egyszerűen úgy jelöljük, hogy egy **aláhúzást** teszünk az attribútum neve elé, például: \_privat), de ezek még így is hozzáférhetők kívülről.

**A self paraméter szerepe**:

* A metódusok első paramétere mindig az aktuális objektumra vonatkozik, amelyet általában **self** névvel jelölnek.
* A **self** kulcsszó nem kötelező, de ez a hagyományos elnevezés.

# Példányosítás Pythonban (1.8)

**Példányosítás**:

* A Pythonban nincs szükségünk új operátorra az objektumok példányosításához, egyszerűen az osztály nevét hívjuk meg, mintha egy függvényt hívnánk.

**Példány létrehozása**:

* A példányosításkor, ha az osztálynak van konstruktora (pl. \_\_init\_\_), akkor azt is meghívja a rendszer.
* A példányok **rendelkeznek az osztály összes attribútumával és metódusával**. Az osztályban definiált metódusokat és attribútumokat minden példány hozzáférheti.

**Példányok attribútumai**

1. **Attribútumok létrehozása**:
   * Pythonban egyszerű **értékadással** hozhatunk létre új attribútumokat egy példányban.
   * Ha egy attribútumot hozzáadunk egy példányhoz, az csak arra az egy példányra vonatkozik, nem érinti a többi példányt.

**Példány szintű attribútumok**:

* A **példány szintű attribútumok** azok, amelyek csak az adott példányhoz tartoznak, és nem befolyásolják más példányokat.
* A példányok egyedi attribútumait bárki hozzáadhatja a program futása közben, mivel Python rugalmas az objektumok szerkezetének módosításában.

# Az információrejtés Pythonban (1.9)

1. **Információrejtés (Encapsulation)**:
   * A Python alapvetően nem kínál explicit **információrejtést** (encapsulation), mint más programozási nyelvek (pl. Java vagy C++).
   * Pythonban **minden attribútum** és **minden metódus alapértelmezés szerint publikus**.
   * Ugyanakkor lehetőség van az attribútumokat védetként és privátként kezelni, ha azokat egy \_ vagy \_\_ előtaggal nevezzük el, de ezek csak konvenciók, és nem valódi védelmet biztosítanak.
     + Az \_ és \_\_ prefixek nem jelentik a valódi rejtést, hanem csak azt jelzik, hogy az attribútumokat nem szabad közvetlenül módosítani.

# Öröklődés Pythonban (2.1)

**Öröklődés alapjai**:

* Pythonban az **öröklődés** lehetőséget biztosít arra, hogy egy osztály (az alosztály) örökölje a másik osztály (szuperosztály) attribútumait és metódusait.

**Többszörös öröklődés**:

* Python támogatja a **többszörös öröklődést**, ami azt jelenti, hogy egy osztály több szuperosztálytól is örökölhet.

**Többszörös öröklődés konfliktusai**:

* Ha ugyanazt az attribútumot vagy metódust találjuk több szuperosztályban, akkor a Python **mélységi, majd balról-jobbra** keresési stratégiát használ a konfliktusok feloldására:
  + **Mélységben kezdünk**: A keresés először a példányosított osztályban történik, majd a szuperosztályokban rekurzívan.
  + **Balról jobbra**: Ha egy osztályt több szuperosztály örököl, akkor a deklaráció sorrendjét követve, balról jobbra történik az attribútum keresés.
  + Ez a viselkedés megoldja a többszörös öröklődés által okozott problémákat, de a keresési sorrend mindig meghatározott.

# Típus és Osztály a Pythonban (2.2)

A **típus** és az **osztály** koncepciói szoros kapcsolatban állnak egymással a Pythonban, de fontos megérteni a különbségeket és hasonlóságokat.

1. **Típus és osztály közötti hasonlóság**:
   * A Pythonban az **osztályok** egy olyan mechanizmust biztosítanak, amivel **új adattípusokat** hozhatunk létre. Az osztályok **típusok** is, mivel meghatározzák, hogy milyen adatokat és metódusokat (üzeneteket) tartalmaznak.
   * A típusok, mint a lista, szótár, vagy sztring, ugyanúgy definiálhatók osztályokban, mint bármilyen egyéb komplex adattípus a programozásban. Az osztályok lehetővé teszik az adatok és azok kezelésére szolgáló metódusok összekapcsolását egyetlen entitásban.
2. **A típusok szerepe**:
   * Pythonban, ha egy új objektumot hozunk létre egy osztályból, akkor annak **típusa** az adott osztály lesz. A típus tehát azt határozza meg, hogy milyen adatokat tartalmazhat az objektum, és milyen műveletek hajthatók végre rajta.
   * Például a beépített Python típusok, mint a list, dict vagy str, mind különböző osztályok, amelyek különböző attribútumokkal és metódusokkal rendelkeznek.

# Objektumorientált Programozás (OOP) (2.3)

1. **Mi is az objektumorientált programozás?**  
   Az **objektumorientált programozás** (OOP) egy programozási paradigma, amely lehetővé teszi, hogy az **objektumok** köré szervezzük a programunkat. Ezek az objektumok tartalmazzák az adatokat és az azokkal kapcsolatos metódusokat (funkciók).
   * Az OOP lehetővé teszi számunkra, hogy a programunkat ne csak utasítások sorozataként, hanem egy olyan **összehangolt objektumok hálózata**-ként gondoljuk el, amelyek kommunikálnak egymással.
   * Az objektumok az adatokat és a velük végzett műveleteket egyesítik egy egységben, és **üzeneteket** küldenek egymásnak, hogy reagáljanak a külső változásokra.
2. **Miért fontos az OOP?**
   * Az OOP segít a **szoftver fejlesztésében** és a **kód hosszú távú fenntartásában**. Az OOP alapú kód könnyebben érthető és karbantartható, mert a programozók a világot objektumok és azok interakcióinak segítségével modellezhetik.
   * Az OOP előnye, hogy segít a **kód modularizálásában**, mivel egyes osztályok és objektumok különálló részként kezelhetők, és könnyen újrahasználhatók más programokban.
3. **Az OOP kulcsfogalmai:**
   * **Objektumok**: Az objektumok az OOP legfontosabb építőkövei. Ezek az osztályok példányai, amelyek adatokkal és metódusokkal rendelkeznek. Az objektumok **állapotot** (adattípusokat) és **viselkedést** (metódusokat) reprezentálnak.
   * **Attribútumok**: Az objektumokhoz kapcsolódó adatok. Az attribútumok változók, amelyek a példányok egyedi állapotát tárolják.
   * **Metódusok**: Az objektumokhoz tartozó függvények, amelyek az objektum adataival dolgoznak és műveleteket végeznek rajtuk.

# Objektumorientáltság célja és előnyei (2.4)

1. **A kód formájának javítása**: Az objektumorientált programozás segítségével a kód szervezettebbé és könnyebben érthetővé válik, mivel az objektumok egymással interakcióba lépve hajtják végre a feladatokat. Az OOP segíti a komplex rendszerek kezelését, mivel lehetővé teszi azok részletesebb és modulárisabb megértését.
2. **Kölcsönhatás objektumok között**: Az OOP-ban az objektumok **üzenetekkel** kommunikálnak egymással, amelyek lehetővé teszik, hogy egy objektum reagáljon más objektumok állapotváltozásaira. Ezzel egy magas szintű leírást adhatunk arról, hogyan működik a program.
3. **Fenntarthatóság és újrafelhasználhatóság**: Az objektumorientált programozás egyik legnagyobb előnye, hogy az objektumok újrahasználhatók más projektekben. A jól megtervezett osztályok és objektumok egyszerűsíthetik a kód karbantartását, mivel a módosításokat könnyen végrehajthatjuk egyetlen osztályban anélkül, hogy az egész programot módosítani kellene.

# Pass Kulcsszó (2.5)

A pass kulcsszó a Pythonban egy olyan művelet, amely **nem csinál semmit**. Általában akkor használjuk, ha egy blokkot (például osztályt vagy függvényt) szintaktikai okokból szeretnénk definiálni, de nem akarunk benne semmilyen kódot elhelyezni. A pass hasznos lehet olyan esetekben, amikor még nem akarunk implementálni egy funkciót vagy osztályt, de a szintaxisnak érvényesnek kell lennie. Ha például egy osztályt kezdünk el írni, de egyelőre nem akarjuk, hogy tartalmazzon bárminemű kódot, akkor a pass kulcsszóval jelezhetjük, hogy a blokk üres.

# Dir() metódus (2.6)

A dir() metódus egy beépített Python függvény, amely a megadott objektum vagy osztály attribútumait listázza ki. Az attribútumok közé tartoznak a változók, metódusok és az objektumhoz kapcsolódó egyéb elemek. Ha egy osztályt vagy objektumot adunk át a dir() metódusnak, akkor az összes elérhető tulajdonságot és metódust tartalmazó lista fog visszajönni.

# Konstruktorok és Attribútumok Pythonban (3.1)

**Konstruktor Pythonban**:

* Pythonban az osztályok konstruktorát az \_\_init\_\_ nevű metódus képviseli.
* A **konstruktor** a példányosításkor automatikusan meghívódik, és lehetőséget ad arra, hogy beállítsuk az osztály attribútumait a példányosítás során.

**Attribútumok létrehozása**:

* Az attribútumokat egyszerű **értékadással** hozhatjuk létre a Pythonban, nem szükséges őket előre deklarálni, mint más programozási nyelvekben.
* Az attribútumokat az objektum példányában is definiálhatjuk.
* Az objektumhoz **nem szükséges előre deklarálni** az attribútumokat, és ezek elérhetők a példányosított objektumból.

**Attribútumok osztály és példány szinten**:

* Az attribútumok létrehozása történhet **osztály szinten**, ha minden példánynak és alosztálynak ugyanazt az értéket kell osztania.
* Vagy létrehozhatók **példány szinten**, ha egy adott objektumhoz akarunk egyedi attribútumot rendelni.
* Az **osztály szintű** attribútumot minden példány megosztja, míg a **példány szintű** attribútumot csak az adott objektumban érhetjük el.

**A Konstruktor**

* Amikor egy osztályt definiálunk, automatikusan létrejön egy függvény is, amelynek neve megegyezik az osztály nevével.
* Ezt a függvényt hívjuk konstruktornak.
  + A konstruktor segítségével hozhatjuk létre az osztály új példányait.
* Ezt a folyamatot mi is befolyásolhatjuk, de alapértelmezetten a Python magától is kezeli a példányosítást.

**Pont Minősítés**

* Amikor egy objektum (példány) valamely mezőjére szeretnénk hivatkozni, a **„pont minősítést”** (dot notation) használjuk.
* Annak eldöntésére, hogy változóról vagy metódusról van szó, a **zárójelezés** segít. Ha zárójelek vannak, akkor metódusról van szó, ha nincsenek, akkor változóról.

**Hogyan hozzunk létre objektum-specifikus értéket**

* Miután egy objektumot létrehoztunk, az adatok ugyanúgy kerülnek hozzárendelésre, mint bármely más Python helyzetben, azaz értékadással
* Így bármely objektumot bővíthetünk egy új változóval.

# Osztály és példány kapcsolat (3.2)

* Mivel minden egyes példány típusa az az osztály, amelyből létrejött, a példány "emlékszik" az osztályára.
* Ezt gyakran **példány-osztály relációként** emlegetik.
* Az osztályt a példányban a \_\_class\_\_ attribútumban tárolja a Python.

**Scope (Hatókör)**

* Az osztályrendszerben a **scope** másképp működik, mivel kihasználja a példány-az-osztály kapcsolatot.

**Az objektum scope szabálya**

Az objektum scope-jában az első két szabály:

* Először az objektumon belül kell keresni.
* Ha az attribútumot nem találjuk meg, akkor fel kell nézni az objektum osztályába, és ott is keresni kell az attribútumot.

# Metódusok (3.3)

**Metódus vs. függvény**

* Mint már említettem, a metódus és a függvény szoros kapcsolatban állnak. Mindkettő "kis program", amely paramétereket fogad, végrehajt egy műveletet, és (esetleg) értéket ad vissza.
* A fő különbség, hogy a metódusok olyan függvények, amelyek egy adott objektumhoz vannak kötve.
* Ez azt jelenti, hogy az objektum, amelyen a metódust hívják, mindig implicit módon paraméterként kerül átadásra.

**Különbség a definícióban**

* A metódusokat az osztály törzsén belül definiáljuk.
* A metódusok mindig az első paramétert az objektumra kötik, amelyik meghívta a metódust.
* Ezt a paramétert bárminek elnevezhetjük, de hagyományosan self-nek hívják.

**További információk a self-ről**

* A self egy fontos változó. Minden metódusban az objektumra van kötve, amelyik meghívja a metódust.
* A self-en keresztül hozzáférhetünk ahhoz az példányhoz, amely meghívta a metódust (és annak összes attribútumához).

**A self automatikus kötése**

* Amikor egy pont (.) szintaxisú metódushívást végzünk, az objektum, amelyik meghívja a metódust, automatikusan hozzárendelődik a self változóhoz.
* A self segítségével emlékezhetünk és hivatkozhatunk a hívó objektumra.
* Bármelyik része az objektumnak, amelyet szeretnénk elérni, mindig a self-el kell, hogy kezdődjön.
* A metódusok generikus módon is megírhatók, úgy, hogy a hívó objektumot a self segítségével kezeljük.

# Osztály definiálása – Python Standard Metódusok (3.4)

A Python számos beépített standard metódust biztosít, amelyeket az osztály tervezője megadhat, hogy az osztály normál "Pythonos" módon viselkedjen.

* Sok közülük dupla aláhúzásokkal van ellátva a nevük elején és végén.
* Ezeknek a metódusoknak a használatával az osztály „beleilleszkedik” a szokásos Python működésbe.

**Standard Metódus: Konstruktor**

* A konstruktor akkor hívódik meg, amikor egy példányt létrehozunk, és lehetőséget ad az osztály tervezőjének, hogy a példányokat változók beállításával inicializálja.

**Konstruktor hívása**

* Ahogy említettük, a konstruktort úgy hívjuk meg, hogy az osztály nevét függvényhívásként használjuk, zárójelekkel

**Konstruktor definiálása**

* Az osztály egyik speciális metódusneve a konstruktor neve, amely \_\_init\_\_.
* Az értékek a konstruktorban történő hozzárendeléssel biztosítják, hogy minden példány ugyanazokkal a változókkal kezdődjön.
* Az osztály konstruktorának az \_\_init\_\_ metódusába argumentumokat is átadhatunk.

**Default Konstruktor**

* **Ha nem adunk meg konstruktort**, akkor a rendszer automatikusan biztosít egy alapértelmezett konstruktort.
* **Alapértelmezett konstruktor**: Az alapértelmezett konstruktor csak a példány létrehozásához szükséges rendszeres feladatokat végzi el, de nem hajt végre egyéb műveleteket.
* **Nem lehet argumentumot átadni** az alapértelmezett konstruktor számára.

**Minden osztálynak rendelkeznie kell \_\_init\_\_-vel**

* A konstruktor megadásával biztosítjuk, hogy minden példány legalább a példányosítás pontján azonos tartalommal jöjjön létre.
* Ez lehetőséget ad arra, hogy kontrolláljuk, hogy a példányok kezdeti értékei miként alakuljanak.

**\_\_str\_\_ és kiírás**

* Amikor meghívjuk print(my\_inst)-et, Python úgy értelmezi, mintha a példányt egy szöveggé próbálnánk konvertálni. Erre a célra az \_\_str\_\_ metódus szolgál.
* Az \_\_str\_\_ metódusban a my\_inst az objektumra (self) vonatkozik, és az objektum kiírása ezen keresztül történik.
* Az \_\_str\_\_ metódusnak mindig **sztringet kell visszaadnia**.

**Kódolási Csoportok**

A kódunkban három csoportot különíthetünk el:

* **Felhasználó** (user): Az, aki a kész osztályt és objektumot használja.
* **Programozó, osztály felhasználó** (programmer, class user): Az a programozó, aki az osztályt használja, de nem tervezte meg.
* **Programozó, osztály tervező** (programmer, class designer): Az a programozó, aki az osztályt megtervezte és készítette.

**Osztály Tervező**

* Az **osztály tervezője** olyan kódot készít, amit más programozók fognak használni.
* Ezzel az osztálytervező gyakorlatilag egy **könyvtárat** hoz létre, amit más fejlesztők kihasználhatnak.

**Az objektum-orientált programozás segíti a szoftverfejlesztést**

* A **szoftverfejlesztés** (software engineering) az a tudományág, amely a kód kezelésével foglalkozik, hogy biztosítsa annak hosszú távú használhatóságát.
* Ezt **refaktorálás** (refactoring) segíti, ami azt jelenti, hogy a meglévő kódot módosítjuk úgy, hogy az átfogó kód egyszerűbbé és könnyebben érthetővé váljon, anélkül hogy a funkcionalitás változna. Csak a **formát** változtatjuk, nem a működést.

**További refaktorálás**

* Az üzenet tartalmának elrejtése lehetővé teszi, hogy változásokat hajtsunk végre az objektumon, miközben az üzenetek áramlása (és azok eredményei) változatlanok maradnak.
* Így az üzenet implementációja változhat, de annak szándékolt hatása ugyanaz marad.
* Ez az **enkapszuláció** (encapsulation).

**OOP elvek**

* **Enkapszuláció**: Az implementációs részletek elrejtése, hogy a program könnyebben olvasható és később könnyebben módosítható legyen.
* **Modularitás**: Az objektumok "önállóvá" tétele, így újra felhasználhatók más kontextusokban (mint a math modul).
* **Öröklődés**: Új objektum létrehozása azzal, hogy számos objektumjellemzőt öröklünk (mint apa és fia között), miközben létrehozhatjuk vagy felülírhatjuk azokat az új objektumban.
* **Polimorfizmus**: Lehetővé teszi, hogy egy üzenetet bármely objektumnak elküldjünk, és az megfelelően reagáljon annak típusa alapján.

**Még mindig az enkapszulációval foglalkozunk**

* Az enkapszuláció azt jelenti, hogy elrejtjük az implementációs részleteket, hogy a program könnyebben olvasható és írható legyen.
* A **modularitás** azt biztosítja, hogy egy objektum újra felhasználható más kontextusokban.
* Az **interfész** (a metódusok) biztosítja, hogy a megfelelő módon kezeljük az osztályt.

**Privát változók**

* **Pythonban** minden objektumnak és modulnak van egy saját névtére, amely valójában egy szótár.
* Ez a szótár a speciális \_\_dict\_\_ változóhoz van kötve.
* A \_\_dict\_\_ tartalmazza az összes helyi attribútumot (változókat, függvényeket) az objektumban.

**Privát változók egy példányban**

* Sok objektum-orientált programozási megközelítés lehetővé teszi, hogy egy változót vagy függvényt priváttá tegyünk egy példányban.
* **A privát** változó azt jelenti, hogy az a változó nem elérhető az osztály felhasználói számára, csak az osztály fejlesztője férhet hozzá.
* A privát változók használata előnyös lehet, mivel lehetőséget adunk arra, hogy kontrolláljuk, ki férhet hozzá az egyes példányértékekhez.

**Adatvédelmi megoldás Pythonban**

* Python a "Mi mind felnőttek vagyunk" megközelítést alkalmazza. Nincsenek szigorú korlátozások.
* Azonban **nevezéktan** biztosít a véletlen hibák elkerülésére. Ha egy változót **\_\_** (két alulvonás) karakterekkel jelölünk, akkor az elnevezés megváltozik úgy, hogy tartalmazza az osztály nevét is.
* Például: **\_\_var** átneveződik **\_class\_\_var**-ra.
* A változó továbbra is teljes mértékben elérhető, és a **\_\_dict\_\_** segítségével nyilvánvaló, hogy hogyan van elnevezve.

**Introspekció**

* A Python nem rendel típust egyetlen változóhoz sem, mivel minden változó bármilyen objektumra hivatkozhat.
* Azonban bármely változót lekérdezhetjük, hogy jelenleg milyen típusú objektumra hivatkozik.
* Ezt a folyamatot **introspekciónak** nevezzük. Vagyis a program futása közben meghatározhatjuk, hogy egy változó milyen típusú objektumra hivatkozik.

# Mi az operátor túlterhelés? (3.5)

Az **operátor túlterhelés** (operator overloading) egy fontos és hasznos technika az objektum-orientált programozásban, különösen olyan nyelvekben, mint a Python. Lényegében lehetővé teszi, hogy ugyanazt az operátort különböző típusú objektumokkal eltérő módon alkalmazzuk.

Alapvetően az operátor túlterhelés azt jelenti, hogy egy adott operátor, mint például +, -, \*, == stb., különböző típusú operándusok esetén más-más műveletet hajtson végre. Pythonban például a + operátort alapértelmezetten az **összeadás** művelethez használjuk, de ugyanazt az operátort használhatjuk más típusok, például két string összeillesztésére (konkatenációjára) is.

Az operátor túlterhelés segítségével saját osztályokban is meghatározhatjuk, hogy az operátorok hogyan viselkedjenek az általunk definiált típusok esetén.

**Miért fontos az operátor túlterhelés?**

1. **Szemantikai tisztaság**: Az operátor túlterhelés lehetővé teszi, hogy az operátorok a típusok természetes viselkedését tükrözzék. Például egy **racionális számok** osztálya esetén az + operátor összeadja a két racionális számot, éppen úgy, ahogy az matematikailag is működik.
2. **Kódolási hatékonyság**: Az operátorok túlterhelésével kódunk olvashatóbbá válik. Az operátorok használatával elkerülhetjük a hosszú és bonyolult függvényhívásokat. Például a a + b kifejezés használata intuitívabb, mint mondjuk add(a, b).
3. **Modularitás és újrafelhasználhatóság**: Az operátor túlterhelés lehetővé teszi, hogy az egyes típusokat egyre inkább a Python beépített típusaihoz hasonlóan kezeljük, így azokat könnyen kombinálhatjuk más típusokkal anélkül, hogy extra kódot írnánk.

**Hogyan működik az operátor túlterhelés Pythonban?**

Pythonban a túlterhelhető operátorokat **speciális metódusok** segítségével definiáljuk. Minden operátor egy bizonyos metódust igényel, amelyet felülírhatunk. Ezeket a metódusokat általában két aláhúzással jelöljük: \_\_ (például \_\_add\_\_, \_\_sub\_\_, \_\_mul\_\_ stb.).

# \_\_str\_\_ vs \_\_repr\_\_ Pythonban (3.6)

Az \_\_str\_\_ és a \_\_repr\_\_ két fontos speciális metódus, amelyeket a Python osztályokban definiálhatunk. Mindkettő célja, hogy objektumokat karakterlánccá alakítson, de eltérő célokra és helyzetekre szolgálnak.

**\_\_repr\_\_**

* A \_\_repr\_\_ a "hivatalos" reprezentációját adja az objektumnak. Ennek az a célja, hogy egyértelmű és értelmezhető módon bemutassa az objektumot, gyakran úgy, hogy a kimenet elegendő információval rendelkezzen ahhoz, hogy az objektumot újra létre lehessen hozni a kódon keresztül.
* A \_\_repr\_\_ metódust akkor hívja meg a Python, amikor egy objektumot kiíratunk a parancssorba (például interaktív módon) vagy amikor az objektumot egy sorban jelenítjük meg.
* A \_\_repr\_\_ visszatérési értékének tipikusan úgy kell kinéznie, hogy a Python értelmezője képes legyen újra létrehozni az objektumot. Például ha a Rational osztály példányának egy string reprezentációját akarjuk, akkor azt úgy kell formáznunk, hogy az objektum újra létrejöjjön ugyanazzal az értékkel.

**\_\_str\_\_**

* Az \_\_str\_\_ a "felhasználói" reprezentáció, vagyis a megjelenítésre szánt karakterláncot adja vissza.
* Ez a metódus akkor kerül meghívásra, amikor az objektumot nyomtatjuk ki a print() függvénnyel, vagy amikor a str() metódust alkalmazzuk rajta.
* Az \_\_str\_\_ célja az, hogy az objektumot olvasható és értelmezhető formában jelenítse meg, a felhasználó számára barátságosabb módon, mint a \_\_repr\_\_.

**Hogyan kapcsolódik a kettő?**

* Az \_\_str\_\_ és **\_\_repr\_\_** metódusok közötti kapcsolat az, hogy gyakran a \_\_repr\_\_ metódus belsőleg meghívja az \_\_str\_\_ metódust, ha nincs kifejezetten implementálva a saját \_\_repr\_\_ metódusunk.
* Ezért gyakran az \_\_str\_\_ metódust definiáljuk, és a \_\_repr\_\_ metódusban a self.\_\_str\_\_() hívást alkalmazzuk.

# \_\_radd\_\_ metódus (3.7)

A \_radd\_\_ metódus egy speciális Python osztály metódus, amely az operátor túlterhelés részeként van jelen. Az operátor túlterhelés célja, hogy egy adott osztály példányai viselkedjenek úgy, mint a beépített típusok (pl. számok, listák), lehetővé téve számukra, hogy aritmetikai és egyéb műveletekhez is hozzáférjenek.

A \_radd\_\_ metódus szerepe az, hogy kezeli a **reverz műveleteket** – vagyis azt az esetet, amikor a bal oldali operandus nem tudja végrehajtani a kívánt műveletet, de a jobb oldali operandus képes rá.

**Hogyan működik a \_\_radd\_\_?**

A \_radd\_\_ metódust akkor hívja meg Python, amikor a bal oldali operandus nem képes kezelni a műveletet. A \_radd\_\_ tehát a "reverse add" (fordított összeadás) névre hallgat, mivel akkor lép életbe, amikor a jobb oldali operandus tudja kezelni a műveletet, de a bal oldali operandus nem.

**Hogyan működik a \_radd\_\_ metódus?**

* **Ha a bal oldali operandus nem képes végrehajtani a műveletet (pl. int típusú)**, akkor a Python megpróbálja meghívni a jobb oldali operandus \_\_radd\_\_ metódusát.
* A \_radd\_\_ metódus lehetővé teszi a jobb oldali operandus számára, hogy felcserélje a helyet, és végrehajtsa a kívánt műveletet.
* A \_radd\_\_ metódus végül meghívhatja az \_\_add\_\_ metódust a felcserélt operandusokkal, így biztosítva, hogy a művelet végrehajtódjon.

**Miért fontos az \_\_radd\_\_?**

* A \_radd\_\_ segít biztosítani, hogy az operátorok (mint az összeadás) **kommutatívak** legyenek, azaz az operandusok sorrendje ne befolyásolja a művelet eredményét.
* A kommutatív műveletek olyan matematikai műveletek, amelyek nem változtatják meg az eredményt a operandusok sorrendjének cserélésével (pl. a + b = b + a).

# Osztály és példány kapcsolata (3.8)

Az osztályok és példányok kapcsolata alapvető az objektum-orientált programozásban:

* **Osztály**: Az osztály egy sablon vagy tervrajz, amely meghatározza, hogy a példányoknak milyen attribútumaik (adatok) és metódusaik (funkciók) lesznek. Egy osztálynak sok példánya lehet.
* **Példány**: A példány egy konkrét objektum, amely az osztályból készül. A példányok a saját attribútumaikkal rendelkeznek, de az osztályuk metódusait osztják meg.

A Pythonban az osztályok és példányok közötti kapcsolatot a következőképpen értelmezhetjük:

* Egy osztálynak sok példánya lehet, és ezek a példányok mind ugyanazokat a metódusokat hívhatják, amiket az osztály definiál.
* Az osztály metódusai mind a példányok által közösen használhatóak, tehát az osztály szintjén vannak definiálva.

**Osztályok közötti kapcsolat (öröklődés)**

A Pythonban az osztályok öröklődése lehetővé teszi, hogy egy osztály (szülőosztály) kódját újra felhasználjuk más osztályokban (al-osztályok). Ezáltal elkerülhetjük a kód duplikálódását, és biztosíthatjuk, hogy az alosztályok hozzáférjenek a szülőosztály metódusaihoz és attribútumaihoz.

A Python osztályhierarchiáját fához hasonlíthatjuk, ahol a gyökérosztály az object, amely minden osztály ősosztálya. Minden új osztály, amelyet létrehozunk, örökli az object osztályt, ha nincs más specifikus szülőosztály megadva.

**Az öröklődés előnyei**

Az öröklődés egyik legfontosabb előnye, hogy lehetővé teszi a **kód újrahasználhatóságát**:

* A szülőosztály kódja **általános** funkcionalitást biztosít, amelyet az alosztályok örökölnek. Ezáltal nem kell minden alosztályban újraimplementálni a közös funkciókat.
* Az alosztályok képesek **specializálni** a szülőosztály kódját, és ha szükséges, **felülírhatják** a szülőosztály metódusait, hogy jobban illeszkedjenek az alosztály specifikus igényeihez.

# Objektumok Hatóköre (3.9)

Amikor egy attribútumot keresünk egy objektumban Pythonban, az alábbi lépéseket követi a rendszer:

1. **Keresés az objektumban**: Először az objektumban magában keresünk, hogy találunk-e ilyen attribútumot.
2. **Keresés az objektum osztályában**: Ha az attribútum nem található meg az objektumban, akkor az osztályától keresünk, amely az objektum viselkedését definiálja.
3. **Keresés az osztály hierarchiájában**: Ha az attribútum az osztályban sincs, akkor felfelé keressük a hierarchiában, az osztály szülőosztályában, és így tovább.
4. **Az object osztály elérése**: Ha elérjük az object osztályt és ott sem találjuk meg az attribútumot, akkor az attribútum nem létezik az objektumban.

Ez a folyamat az úgynevezett **"find the attribute"** játék, ahol az interpreter követi az attribútumok keresésének hierarchikus szabályait.

# Öröklődés: Erő és Komplexitás (4.1)

* **Öröklődés**: Az objektum-orientált programozás egyik alapvető ereje az, hogy lehetővé teszi a kód újrahasználatát. Az öröklődés révén a szülőosztályok közös viselkedéseit a gyermekosztályok örökölhetik.
* **Komplexitás**: Azonban az öröklődés nem mindig egyszerű, és bonyolultabb hierarchiák esetén nehezebbé válhat a kód kezelés és karbantartás. Egyes öröklődési struktúrák használata bonyolult, különösen ha sok szintű öröklődést alkalmazunk.

# A Beépített Típusok is Objektumok (4.2)

* A Pythonban a **beépített típusok** (mint a lista, számok, stb.) szintén objektumok. Ez lehetőséget ad arra, hogy ezeket a típusokat kiterjesszük, és egyedi viselkedést adjunk hozzá az öröklés segítségével.
* A beépített típusok tulajdonságainak öröklése és testreszabása egyszerűsíti a kód újrahasználatát, és rugalmasabbá teszi a programokat.

**Metódusok Specializálása**

* A normál metódushívásokat **kötött metódusoknak** (példánymetódus) nevezzük. A kötött metódusok úgy működnek, hogy a metódushívás előtt egy példány szerepel, és a Python automatikusan átadja neki a self paramétert.

**Nem Kötött Metódusok (osztálymetódus vagy statikus metódus)**

* Lehetőség van **nem kötött metódusok** hívására is, ahol a self paramétert explicit módon kell átadni a metódusnak.
* Az ilyen metódusokat az osztályon keresztül hívjuk meg, és a példányt (self) kézzel kell átadnunk

**A super() használata**

* Nézzünk egy példát, ahol egy új osztályt szeretnénk létrehozni, amely örökli a listát. Az új osztály példányait megfelelően szeretnénk inicializálni, ezért meg kell hívnunk a szülőosztály \_\_init\_\_ metódusát. Ha nem hívjuk meg explicit módon a szülő osztály \_\_init\_\_ metódusát, akkor az alosztály örökölhet a szülő osztályból, de nem biztos, hogy az inicializálás megfelelően történik meg.

# Miért Használjuk a super()-t? (4.3)

* Ha nem hívjuk meg explicit módon a szülő osztály \_\_init\_\_ metódusát, a gyermekosztály örökölheti a szülő osztály viselkedését, de fontos, hogy a megfelelő kontextusban hívjuk meg a szülő osztály inicializálását.

**Explicit Hívás a super() Használatával**

* A szülő osztály konstruktorának meghívása **nem kötött metódusként** történik, mivel a super()-t használjuk, amely az alosztály konstruktorában explicit módon hívja meg a szülő osztály metódusát.

Miért nem használunk kötött metódust?

* + A kötött metódusok a példányhoz kapcsolódnak, és automatikusan átadják a self-et. Azonban a super() használata esetén a self átadása explicit módon történik, mivel a metódust az osztály szintjén hívjuk meg.

**Öröklés és a kód újra felhasználása**

* Az **öröklődés** lehetővé teszi a kód **specializálását**: az alosztály örökli a szülőosztály kódját, de az alosztály testre szabhatja azt.
* Az **override** lehetősége azt jelenti, hogy a gyermekosztály **megváltoztathatja** a szülőosztály viselkedését, hogy az jobban megfeleljen a specifikus igényeiknek.
* Az **újrahasználható kód** előnye, hogy nem kell újra megírni a szülőosztály funkcióit, hanem azokhoz hozzáadhatunk saját specifikus viselkedést a gyermekosztályban.

# Öröklődés áttekintése (4.4)

**Öröklődés alapjai:**

* A Python-os projektek és csomagok bővítése során fontos, hogy alkalmazd az osztályokat és kövesd az "DRY" (Don't Repeat Yourself) elvet.
* Az **osztály öröklődés** egy nagyszerű módja annak, hogy egy új osztályt hozzunk létre egy másik osztály alapján, így elkerülhetjük a kód ismétlését.

**Mi az osztály öröklődés?**

* Az osztály öröklődés ugyanúgy működik, mint a genetika: a gyermek osztály (Child) örökli a szülő osztály (Parent) attribútumait és metódusait.

**Miért használjuk a super()-t?**

* A **super()** használata lehetővé teszi, hogy a gyermek osztályban meghívjuk a szülő osztály konstruktorát anélkül, hogy explicit módon kellene a szülő osztály nevét használnunk. Ez segít abban, hogy könnyebben módosíthassuk az öröklődési hierarchiát anélkül, hogy minden metódushívásnál meg kellene változtatnunk a szülőosztály nevét.

**Előnyök és hátrányok a super() használatában:**

**Előnyök:**

* A **karbantarthatóság** szempontjából, ha a gyermek osztály öröklődési mintázata változik, nem szükséges minden egyes szülőosztályra hivatkozó helyet megkeresni és módosítani, mivel a super() automatikusan kezeli ezt.

**Hátrányok:**

* Az egyik érv a super() ellen, hogy a kód **kevésbé explicitté** válik. Az "Explicit is better than implicit" (Az explicitek jobbak, mint az implicitek) elv szerint a super() használata helyett inkább egy konkrét hívás (pl. Parent.\_\_init\_\_(self)) lehet a jobban átlátható megoldás.

**Többszörös Öröklődés és a super() Használata**

**Mi az a többszörös öröklődés?**

* Az eddigi példák egyetlen szülőosztályból származó gyermekosztályok öröklődését mutatták be. **Többszörös öröklődés** esetén egy gyermekosztály több szülőosztályból is örökölhet. Ez azt jelenti, hogy egy osztály 2, 3 vagy akár 10 különböző szülőosztályból is örökölhet.
* A többszörös öröklődés esetén a **super()** előnyei még világosabban megmutatkoznak, mivel segít a szülőosztályok közötti konfliktusok kezelésében.

**Többszörös Öröklődés Példa**

Ha két szülőosztályban ugyanaz a metódus neve szerepel, akkor felmerül a kérdés, hogy Python hogyan dönt, hogy melyik szülő metódusát hívja meg. Erre a problémára létezik egy olyan koncepció, mint a **többszörös feloldás sorrendje (MRO - Method Resolution Order)**.

**MRO és a super():**

A **MRO** azt határozza meg, hogy Python milyen sorrendben keresi meg a metódust az öröklődési hierarchiában, és melyik metódust hívja meg, ha konfliktus lép fel. Az MRO szabályozza azt is, hogy amikor egy osztálynak több szülője van, hogyan oldja meg a rendszer az öröklési láncot és a metódusok elérését.

**A super() és a "Gyémánt Probléma" (Diamond Problem):**

A **gyémánt probléma** akkor merül fel, amikor egy osztály többszörösen örököl egy közös szülőt, és a metódusok konfliktusa lép fel a különböző szülőosztályok között. A Python ezt a problémát az MRO és a super() segítségével kezeli.

**MRO és super() használata a gyémánt problémában:**

Pythonban az MRO meghatározza az öröklődési sorrendet. Ha a gyermek osztály több szülőtől is örököl, akkor a super() a hierarchia előre meghatározott sorrendje szerint keres a metódusok között, hogy elkerülje a konfliktusokat.

# Absztrakt Osztályok a Pythonban (4.5)

1. **Mi az Absztrakt Osztály?**
   * Az **absztrakt osztály** alapvetően egy tervrajzként szolgál más osztályok számára. Lehetővé teszi, hogy meghatározz egy sor metódust, amelyeket minden, ebből az osztályból származó gyermekosztálynak implementálnia kell.
   * Az **absztrakt osztály** olyan osztály, amely legalább egy **absztrakt metódust** tartalmaz. Az absztrakt metódusok olyan metódusok, amelyeknek van deklarációjuk, de nincs implementációjuk.
   * Az **absztrakt osztályokat** nem lehet példányosítani (nem hozhatunk létre objektumot belőlük). A gyermekosztályoknak kell biztosítaniuk az absztrakt metódusok implementálását, amelyek az absztrakt osztályokban vannak meghatározva.
2. **Absztrakt Osztályok Használata:**
   * Az **absztrakt osztályok** lehetővé teszik, hogy részben implementálj osztályokat. Ha egy osztály teljesen implementálja az összes metódust, akkor az interfésznek számít.
   * Az **absztrakt osztályok** lehetővé teszik, hogy alapértelmezett funkcionalitást biztosíts a gyermekosztályok számára. Egy absztrakt alap osztály (Abstract Base Class, **ABC**) segítségével közös alkalmazásprogramozási interfészt (API) definiálhatsz egy osztálycsoport számára.
3. **Python és az Absztrakt Osztályok:**
   * Python alapértelmezetten nem támogatja az absztrakt osztályokat, de a Python rendelkezik egy modullal, amely lehetővé teszi az absztrakt alap osztályok (ABC) meghatározását. Ez a modul az **ABC** névre hallgat.
   * Az absztrakt osztályok **hiányosak**, mivel olyan metódusokat tartalmaznak, amelyeknek nincs implementációjuk. Ha Python lehetővé tenné egy absztrakt osztály példányosítását, és valaki meghívná az absztrakt metódust, akkor hiba lépne fel, mivel nincs valódi implementáció.
   * Az absztrakt osztályok **sablonként** szolgálnak. A szükségleteknek megfelelően bővíthetjük őket, és építhetjük őket, mielőtt használni tudnánk őket.

# Belső Osztályok (Inner Classes) Pythonban (4.6)

1. **Mi az a Belső Osztály (Nested Class)?**
   * A **belső osztály** vagy **beágyazott osztály** egy olyan osztály, amely egy másik osztályon belül van meghatározva. Az ilyen típusú osztályok struktúrája jól látható az alábbi példában, ahol a Engine osztály a Car osztály belső osztálya.
2. **Miért Használjunk Belső Osztályokat?**
   * **Osztályok csoportosítása:** Egy jó példa erre, amikor például egy Car (autó) osztályhoz egy Engine (motor) osztály tartozik. Mivel minden autóhoz tartozik egy motor, de a motor nem fog önállóan működni autó nélkül, ezért célszerű a Engine osztályt a Car osztály belső osztályaként definiálni.
   * **Kód egyszerűsítése:** Az ilyen típusú csoportosítással csökkenthetjük a szükséges kód mennyiségét.
   * **Kód elrejtése:** A beágyazott osztályok segítségével elrejthetjük a kódot a külvilág elől, így csak azokat az osztályokat láthatjuk kívülről, amelyek valóban szükségesek.
   * **Könnyű érthetőség:** A belső osztályok közvetlenül kapcsolódnak a külső osztályhoz, így a kód könnyebben érthető, és nem szükséges keresgélnünk külön fájlokban vagy osztályokban.
3. **Belső Osztályok Használata:**
   * Bár a belső osztályok nem a leggyakrabban használt funkciók Pythonban, hasznosak lehetnek, ha kódot kell szerveznünk.
   * A belső osztályokat könnyen elérhetjük a külső osztályból a self kulcsszó használatával. Ez lehetővé teszi, hogy gyorsan létrehozzunk egy példányt a belső osztályból, és végezzünk műveleteket a külső osztályban.
   * Fontos, hogy **nem férhetünk hozzá a külső osztályhoz** a belső osztályból közvetlenül. A belső osztály csak a külső osztályban lévő példányokra tud hivatkozni.

# Miért Fontos a Tesztelés? (5.1)

1. **A Programok Védelme a Hibáktól:**
   * Korábban már beszéltünk arról, hogyan védhetjük meg programjainkat a felhasználók által beadott hibás adatokról.
   * De ki védi meg a programokat azoktól a hibáktól, amelyeket mi, fejlesztők véletlenül beléjük építünk?
   * A válasz: nekünk, programozóknak kell megvédenünk magunkat. Ezért vagyunk felelősek, hogy ellenőrizzük, teszteljük saját munkánkat.
2. **A "Jó" Programok és a Tesztelés Szerepe:**
   * A programozóknak a lehető legjobban kell integrálniuk a tesztelést a fejlesztési folyamatba.
   * Hogyan lehetünk biztosak abban, hogy a program helyesen működik, ha nem végzünk teszteket a hibák feltárására?
   * Léteznek olyan fejlesztési módszerek és megközelítések, amelyek előírják, hogy a fejlesztők először teszteket írjanak meg, még mielőtt bármilyen kódot írnának. Az egyik ilyen módszer a **tesztvezérelt fejlesztés (TDD)**, amely azt javasolja, hogy a tesztek írása előzze meg a kódolási folyamatot.
3. **A Tesztelés Előnyei:**
   * **Hibák Felfedezése:** A tesztek segítenek a hibák korai felismerésében, még mielőtt a programot használatra bocsátanánk.
   * **Kód Minőségének Növelése:** A rendszeres tesztelés hozzájárul a kód megbízhatóságához és karbantarthatóságához.
   * **Önbizalom Növelése:** Ha egy program sikeresen teljesít minden tesztet, a fejlesztők magabiztosabbak lehetnek abban, hogy a kódjuk helyesen működik.
4. **Különböző Hibák és Tesztelési Típusok:**
   * Különböző hibák léteznek (pl. logikai, szintaktikai, futásidejű hibák), amelyek mind más-más tesztelési módszert igényelhetnek.
   * A tesztelésnek több fajtája van, mint például az **egységtesztelés**, **integrációs tesztelés**, és **rendszertesztelés**, amelyek mind hozzájárulnak a program különböző részeinek ellenőrzéséhez.

# Hibák Típusai (5.2)

A hibák különböző kategóriákba sorolhatók a jellegük és a keletkezési módjuk alapján. Az alábbiakban a leggyakoribb hibafajtákat tekintjük át:

1. **Szintaktikai Hibák** (*Syntax Errors*):
   * A programozási nyelv szabályainak megsértése. Ilyenkor a program nem képes futni, mivel Python maga észleli és jelzi a hibát. Ezek gyakran a helytelen parancsstruktúrából vagy elírásokból erednek.
2. **Szemantikai Hibák** (*Semantic Errors*):
   * A kód szintaktikailag helyes, de olyan utasítást adunk a programnak, amit nem képes végrehajtani. Például 0-val osztás, vagy egy szöveg típusú adat átalakítása egész szám típusúvá („a” átalakítása integerré). Ezek a hibák futás közben jelentkeznek.
3. **Tervezési Hibák** (*Design Errors*):
   * A program szintaktikailag és szemantikailag is helyes, de nem az elvárásoknak megfelelően működik. Például rossz eredményeket adhat vagy hiányos eredményekhez vezethet. Ezek a hibák gyakran a program alapvető logikájának tervezési hiányosságaiból erednek.

**A Tesztelés Segítsége a Hibák Elkerülésében**

* **Tesztelés**: A tesztelés segít a szemantikai és tervezési hibák felismerésében, mielőtt a programot véglegesen elkészítjük. Ezzel időben azonosíthatjuk a problémákat és gyorsabban javíthatjuk azokat a fejlesztési folyamat korai szakaszában.
* **Gyorsabb Hibajavítás**: Ha a tesztelés során találunk hibákat, azok kijavítása sokkal gyorsabb és egyszerűbb, mintha csak a fejlesztés után, a végleges program használata közben derülnének ki a problémák.

**A Programozási Folyamat: Tervezés – Tesztelés – Programozás**

Egy jó programozó követi a következő lépéseket:

1. **Tervezés**: A program minden aspektusának megtervezése a kódolás megkezdése előtt, beleértve a funkciókat, osztályokat, és a felhasználói felületet.
2. **Tesztelési Programok Írása**: Írja meg azokat a tesztprogramokat, amelyeket a fő programnak teljesítenie kell, hogy biztosítsa a helyes működést.
3. **Kód Írása**: Végül következik a kódolás, amely a tervezés és tesztelés során felállított keretek szerint történik.

A megfelelő tervezés és tesztelés nagyban megkönnyíti a kódolást, tisztábbá és kevésbé hibaképessé téve a fejlesztési folyamatot.

**A "Bug" Kifejezés Eredete**

* A programozásban a "bug" kifejezés hibát vagy működési zavart jelent. Érdekes módon a kifejezés eredete nem a számítástechnikából származik:
  + **Edison** is használta a kifejezést különféle hibás működésű találmányaival kapcsolatban.
  + **Távírók és speciális berendezések** használói is említik a hibás működéseket "bug"-ként.
  + **Admirális Grace Hopper** egy valódi "bug"-ra hivatkozott: a számítógépek korai korszakában egy lepke került egy relébe, ami miatt a rendszer hibásan működött, és így vált a kifejezés a programhibák szinonimájává.

A "bug" tehát mára a szoftverfejlesztés állandó elemévé vált, jelölve minden olyan hibát, amelyet a programokban találunk, legyen az szintaktikai, szemantikai vagy tervezési hiba.

# Tesztelési Típusok (5.3)

**Miért Olyan Nehéz a Tesztelés?**

* A tesztelés **bonyolult és széleskörű terület**, amely számos különböző aspektust foglal magában.
* Bár nehéz átfogó képet adni róla röviden, itt egy áttekintést nyújtunk a legfontosabb alapokról és típusokról, hogy ösztönözzük a mélyebb vizsgálódást.

**Statikus és Dinamikus Tesztelés (1)**

1. **Statikus Tesztelés**: A programozás előtti, tervezési fázis, amelyben megértjük a problémát és a követelményeket, majd ez alapján készítünk tervet a kódhoz. Ezt statikus tesztelésnek hívják, mivel elviekben nem változik idővel (bár a valóságban ez gyakran változik).
2. **Dinamikus Tesztelés**: A kódírás utáni tesztelés, amely során a terv megvalósítása alapján javítjuk a kódot. Mivel a kód folyamatosan frissül, ez a folyamat dinamikus.

**Előzetes és Utólagos Tesztelés (2)**

* **Utólagos Tesztelés**: A hagyományos megközelítés, amikor a kód megírása után teszteljük, hogy megfelelően működik-e.
* **Előzetes Tesztelés**: Modern megközelítés, ahol a teszteket a kód megírása előtt készítjük el. Ennek célja, hogy a kódot mindig az előre megírt tesztekhez igazítsuk, ezzel biztosítva, hogy minden fejlesztés során a meglévő funkciók stabilan működjenek.

**Tesztelési Szintek**

1. **Egységtesztelés**: Az egyes alapegységek (mint pl. függvények, osztályok) külön-külön történő tesztelése.
2. **Integrációs Tesztelés**: Két vagy több egység együttműködésének vizsgálata annak biztosítására, hogy azok megfelelően működnek együtt.
3. **Rendszertesztelés**: Az egész rendszer vizsgálata annak megállapítására, hogy az a kívánt módon működik-e.

# Mit Teszteljünk? (5.4)

A **tesztelés céljainak meghatározása** az egyik legnehezebb feladat, hiszen sok különböző szempontból megközelíthetjük a program ellenőrzését. Az alábbiakban felsoroljuk a főbb területeket, amelyekre figyelmet kell fordítani a tesztelés során:

**1. Helyesség**

* **Kérdés**: Az eredmények valóban helyesek-e?
* **Lényege**: Az ellenőrzés célja annak megállapítása, hogy a program minden esetben a kívánt eredményeket állítja-e elő. Ehhez fontos ismernünk az összes lehetséges esetet és helyes eredményüket.

**2. Teljesség**

* **Kérdés**: Minden lehetséges esetet lefedünk-e?
* **Lényege**: A program minden típusú bemenetet, felhasználói műveletet, adatformátumot és adattípust kezel-e, amit várhatóan használnak majd? A tesztelés során különböző forgatókönyveket kell tesztelni, hogy minden eset lefedett legyen.

**3. Biztonság**

* **Kérdés**: A program védett-e a külső támadásokkal szemben?
* **Lényege**: Annak biztosítása, hogy a program biztonságosan működik és megvédhető mindenféle kártékony behatolástól, például az interneten keresztüli támadásoktól.

**4. Felhasználói Felület**

* **Kérdés**: A felület használható-e és teljes-e?
* **Lényege**: A felhasználói felület egyszerűen érthető és használható legyen. Fontos ellenőrizni, hogy minden funkció megfelelően működik, és a felhasználók a várt élményt kapják.

**5. Terhelhetőség**

* **Kérdés**: Hogyan viselkedik a program nagy terhelés esetén?
* **Lényege**: A program teljesítményének vizsgálata akkor, ha sok felhasználó, nagy adathalmazok, vagy intenzív hálózati forgalom terheli.

**6. Erőforrás-kezelés**

* **Kérdés**: A program optimálisan használja az erőforrásokat?
* **Lényege**: A memória, processzor, hálózati sávszélesség felhasználásának figyelemmel kísérése. Fontos ellenőrizni, hogy a program nem pazarolja ezeket az erőforrásokat.

**7. Válaszképesség**

* **Kérdés**: A program megfelelő időn belül reagál?
* **Lényege**: A program válaszidejének tesztelése annak biztosítása érdekében, hogy a válaszok ésszerű időn belül megérkezzenek, különösen ha időkorlátokat is figyelembe kell venni.

A **tesztelési stratégia** összeállítása során fontos ezeket a területeket figyelembe venni, hogy a program megfelelően működjön és minden szempontból megbízható legyen.

**Felhasználói Hibák Kezelése**

A program biztonságát és használhatóságát növelhetjük, ha a felhasználói hibákat előre kezeljük. Ezt Pythonban kivételkezeléssel (exceptions) tehetjük meg, amely lehetővé teszi, hogy a program a váratlan bemenetekre is helyesen reagáljon.

**Fejlesztői Hibák Kezelése**

A saját fejlesztésünk közben fellépő hibák ellen is védekezhetünk, mivel a hibák nagy része a programozás során íródik be a kódban.

1. **Logikai tesztek beszúrása**: Olyan ellenőrzéseket adhatunk a kódhoz, amelyek megakadályozzák a nem kívánt állapotok kialakulását.
2. **Automatikus ellenőrző tesztek**: Ezeket azért adhatjuk a kódhoz, hogy a fejlesztés alatt automatikusan értesüljünk a hibás működésről.

**Az assert Utasítás**

Az **assert** utasítással ellenőrizhetünk olyan feltételeket, amelyeknek mindig igaznak kell lenniük, ha a kód helyesen működik. Az assert után írt logikai feltétel igaz kell, hogy legyen; ha hamis, a program megszakad, és egy AssertionError hibát dob. Megadhatunk egy üzenetet is a hiba mellett, amelyet a hiba részletezésére használhatunk, megkönnyítve ezzel a hiba azonosítását.

**Az assert Hatékonysága**

Az **assert** utasítás hasznos eszköz lehet a fejlesztés során fellépő hibák elhárítására, és biztosítja, hogy a kód nem fut végig helytelen logikai feltételekkel. Az assert állítások fejlesztési környezetben aktívak, de később, a kész programban, letilthatóak, hogy ne befolyásolják a program teljesítményét.

# Doctest Modul a Pythonban (5.5)

A doctest modul a Pythonban egy egyszerű eszköz, amely lehetővé teszi a kód példáinak beágyazását és automatikus tesztelését a dokumentáció részeként. Ez a módszer segít a fejlesztőknek ellenőrizni a kódot, valamint megmutatja a felhasználóknak a várt használati példákat.

**doctest Modul Előnyei**

* **Példák ellenőrzése**: A doctest lehetőséget nyújt a dokumentációban található példák futtatására és ellenőrzésére, hogy a kód működik-e a leírtak szerint.
* **Folyamatos tesztelés**: Minden kódsor módosítása után lefuttathatjuk a teszteket, hogy ellenőrizzük, nem okoztak-e hibát.
* **Dokumentáció mint teszt**: Az olvasó számára is bemutatja a kód tervezett működését, így segít megérteni, hogyan használható az adott funkció vagy osztály.

**doctest Felépítése a Docstring-ben**

A doctest példákat speciális formátumban adhatjuk meg a kód docstring-jében:

1. **Példasor**: Minden teszt >>> karakterekkel kezdődik.
2. **Elvárt eredmény**: A kimeneti értéket közvetlenül a példa után tüntetjük fel.
3. A doctest.testmod() függvényt kell meghívni, hogy az összes tesztet lefuttassa a modulban található docstring-ek alapján.

# Unittest Modul a Pythonban (5.6)

Az unittest modul a Pythonban egy átfogó tesztelési keretrendszer, amely lehetővé teszi a különféle tesztelési folyamatok létrehozását és automatizálását, különösen az **integrációs** és **rendszer tesztelés** során.

**unittest Előnyei és Funkciói**

Az unittest egy külön fájlban fut, nem beágyazva a forrásfájlba, így a kód áttekinthetőbb és strukturáltabb. Ez különösen hasznos nagyobb tesztrendszerek kialakításához, és összetettebb funkcionalitást kínál a doctest-hez képest.

**unittest.TestCase Használata**

Az unittest keretrendszerben az összes tesztet a TestCase osztályból származtatott osztályban definiáljuk. Minden olyan metódus, amely test-tel kezdődik, automatikusan tesztként fut le.

**Fontos Metódusok és Ellenőrzések**

* **setUp**: Minden teszt előtt fut le, beállítja a kezdeti környezetet.
* **tearDown**: Minden teszt után fut le, és eltakarítja a tesztelés során használt erőforrásokat.
* **assertEqual(a, b)**: Ellenőrzi, hogy a és b egyenlőek-e.
* **assert\_(condition)**: Ellenőrzi, hogy a feltétel igaz-e.
* **assertRaises**: Ellenőrzi, hogy a megadott kivétel bekövetkezik-e.

**Összetett Tesztelési Funkciók**

Az unittest modul lehetőséget nyújt arra, hogy átfogó rendszertesztelési feladatokat végezzünk, és az eredményeket gyorsan értékeljük. Ez különösen hasznos bonyolultabb alkalmazások esetén, ahol a hibakeresés fontos a rendszer stabilitása szempontjából.

# Mintaillesztés (6.1)

A **mintaillesztés** (más néven **mintavételezés** vagy **pattern matching**) egy olyan programozási technika, amely lehetővé teszi, hogy egy adatstruktúrát összehasonlítsunk egy előre meghatározott mintával, és annak megfelelően elvégezzünk bizonyos műveleteket. A mintaillesztés különböző formákban létezik, és számos programozási nyelvben használják a kód olvashatóságának és karbantarthatóságának javítására.

# Brute force algoritmus (6.2)

A **mezítlábas (brute force)** algoritmus egy egyszerű, de gyakran nem hatékony megközelítése a problémák megoldásának. Az alapötlete az, hogy minden lehetséges megoldást vagy lehetőséget kipróbálunk, amíg meg nem találjuk a keresett választ. A brute force algoritmusok nem használják fel a problémák szerkezetét vagy egyéb optimalizálási technikákat, hanem egyszerűen mindent kipróbálnak, amíg a megoldásra nem bukkannak.

**Jellemzők:**

1. **Teljes keresés**: A brute force algoritmusok minden lehetséges megoldást próbálnak végig, nem hagynak ki semmit. Ez azt jelenti, hogy a megoldás minden egyes lehetséges kombinációját, értékét vagy lehetőségét kipróbálják.
2. **Egyszerűség**: A brute force megoldások gyakran nagyon egyszerűek, mivel nem igényelnek bonyolult matematikai vagy algoritmus-elméleti ismereteket. Az algoritmus csak végrehajtja a feladatot úgy, ahogyan az van, anélkül, hogy optimalizálná azt.
3. **Nem hatékony**: A brute force megoldások gyakran rendkívül lassúak, mivel az összes lehetséges megoldást próbálják végig, ami nagyon időigényes lehet, különösen, ha nagy számú lehetséges megoldásról van szó.

**Miért használják mégis?**

Bár a brute force nem mindig hatékony, van néhány olyan eset, amikor használata indokolt lehet:

1. **Egyszerűség**: Ha a problémát gyorsan kell megoldani, és nincs szükség bonyolult optimalizálási technikákra, akkor a brute force egy egyszerű megoldás lehet.
2. **Kisebb adathalmazok**: Ha a bemenet viszonylag kicsi, akkor a brute force algoritmusok még gyorsan végrehajthatóak, és nem jelentkeznek a sebességi problémák.
3. **Prototípusok és tesztelés**: A brute force algoritmusok kiválóan alkalmasak arra, hogy prototípusokat készítsünk vagy gyors tesztelést végezzünk egy probléma megoldásához, mielőtt bonyolultabb algoritmusokat implementálnánk.
4. **Garantált eredmény**: Mivel a brute force minden lehetséges megoldást kipróbál, biztosak lehetünk benne, hogy megtaláljuk a helyes megoldást, ha van ilyen.

# **Knuth–Morris–Pratt (KMP) algoritmus (6.3)**

A **Knuth–Morris–Pratt (KMP) algoritmus** egy hatékony keresési algoritmus, amelyet karakterláncok (stringek) keresésére használnak egy másik karakterláncban. Az algoritmus célja, hogy minimalizálja a karakterek összehasonlításának számát a szövegben történő keresés során, így gyorsabbá téve a keresést, mint a brute-force módszerek, amelyek minden karaktert külön-külön összehasonlítanak. A KMP algoritmus lényege, hogy a már megtalált részstringek segítségével kihasználja azokat az információkat, hogy a keresés folytatásakor ne kelljen újra átvizsgálni a már összehasonlított karaktereket. Ez egy **előfeldolgozó lépést** igényel, amely meghatározza a "hátralépési szabályokat" a minták kereséséhez.

# Boyer-Moore algoritmus (6.4)

A **Boyer-Moore algoritmus** a karakterláncok keresésére használt egyik legismertebb és leggyorsabb algoritmus, különösen akkor, ha a keresendő minta viszonylag hosszú. Ez az algoritmus egy speciális mintaillesztési eljárás, amely a mintát nem az elejétől, hanem a végétől kezdve hasonlítja össze a szöveggel, és ha nem egyezik, a mintát gyorsan eltolja a szövegben. Ezt a sebességet több trükk segítségével éri el, mint például a **hátrálási szabályok** és az **ugrási** stratégiák.

**A Boyer-Moore algoritmus alapelve**

A Boyer-Moore algoritmus két fő ötletet alkalmaz a mintaillesztés hatékonyságának növelésére:

1. **Jó illeszkedés esetén az algoritmus nem csak a következő karakterekkel próbálkozik**, hanem a minta egy része alapján úgy tolja el a mintát, hogy biztosan nem veszít el fontos karaktereket.
2. **Előre meghatározott szabályok segítségével** a mintát a szöveghez képest gyorsan eltolhatja, anélkül hogy minden karaktert újra összehasonlítana.

# **Rabin–Karp algoritmus (6.5)**

A **Rabin–Karp algoritmus** egy másik hatékony mintaillesztési algoritmus, amely különösen akkor hasznos, ha több minta keresésére van szükség egy szövegben. Az algoritmus a **hashing** (titkosítás) elvét alkalmazza, hogy gyorsan ellenőrizze, hogy egy minta illeszkedik-e a szöveg egy részéhez, és ha nem, gyorsan eltolja a keresést.

* A Rabin-Karp algoritmus különösen akkor hasznos, amikor egy szövegben több mintát keresünk, mivel lehetővé teszi, hogy több mintát is egyszerre ellenőrizzünk.
* A Rabin–Karp algoritmus a **hashing** technikát alkalmazza annak érdekében, hogy gyorsan összehasonlítsa a minta hash értékét a szöveg egy-egy részének hash értékével. Ha két hash érték megegyezik, akkor az algoritmus összehasonlítja a karaktereket is, hogy megbizonyosodjon róla, hogy valóban illeszkedés történt-e.

# A re modul a Pythonban (6.6)

A **reguláris kifejezések** (röviden **regex**) egy rendkívül erőteljes eszközt kínálnak a szövegek keresésére és manipulálására. A reguláris kifejezések segítségével komplex mintákat tudunk keresni, például a karakterek, számok, szavak vagy más egyedi minták előfordulásait a szövegekben.

A Pythonban a **re** modul biztosítja a reguláris kifejezésekkel kapcsolatos funkciókat. Az alábbiakban bemutatok néhány alapvető szabályt és metakaraktert, amelyek a reguláris kifejezéseken belül használhatók.

**Alapvető metakarakterek:**

* \*: A megelőző karakter **nulla vagy több** előfordulása.  
  Például: ab\* illeszkedik az a, ab, abb, abbb stb. szavakra.
* +: A megelőző karakter **egy vagy több** előfordulása.  
  Például: ab+ illeszkedik az ab, abb, abbb stb. szavakra, de nem illeszkedik az a egyedül.
* ?: A megelőző karakter **nulla vagy egy** előfordulása.  
  Például: s?a illeszkedik az a vagy sa szavakra.
* .: **Bármilyen karakter** (kivéve új sor karaktert).  
  Például: e.h illeszkedik az eh, exh, eah stb. szavakra.
* ^: **Sztring eleje**.  
  Például: ^if illeszkedik olyan szövegekre, amelyek if-el kezdődnek.
* $: **Sztring vége**.  
  Például: \“:$ illeszkedik olyan szövegekre, amelyek : karakterrel végződnek.
* {}: **Számított előfordulás**.  
  Például: a{2} illeszkedik az aa szóra, a{2,4} illeszkedik aa, aaa, aaaa szavakra.
* |: **Választás**, azaz "ez vagy az".  
  Például: a(b|c) illeszkedik az ab vagy ac szavakra.
* (): **Csoportosítás**.  
  Például: (ab)?b illeszkedik az ab vagy b szavakra.

**Speciális karakterek:**

* \d: **Számjegy** (0–9).  
  Például: \d{3} illeszkedik bármely három számjegyű számra, mint például 123.
* \D: **Nem számjegy** (bármely karakter, ami nem számjegy).  
  Például: \D{2} illeszkedik két nem számjegyű karakterből álló szövegre.
* \s: **Fehér karakter** (térköz, tabulátor, új sor).  
  Például: \s+ illeszkedik egy vagy több fehér karakterre.
* \S: **Nem fehér karakter**.  
  Például: \S+ illeszkedik bármely nem fehér karakterből álló szövegre.
* \w: **Szó karakter** (betű, számjegy, aláhúzás).  
  Például: \w+ illeszkedik egy vagy több szóbeli karakterből álló szövegre, mint például abc123.
* \W: **Nem szó karakter**.  
  Például: \W+ illeszkedik bármely nem szóbeli karakterből álló szövegre, mint például !@#.
* \b: **Szó határ** (szó eleje vagy vége).  
  Például: \bin.\*in\b illeszkedik olyan szavakra, amelyek in-nel kezdődnek és végződnek.
* \B: **Nem szó határ**.  
  Például: \Bin.\*\Bin illeszkedik olyan szavakra, amelyek nem kezdődnek és nem végződnek szó határon.
* \A: **Sztring eleje** (az egész szöveg eleje).  
  Például: \Aif illeszkedik csak akkor, ha a szöveg az if szóval kezdődik.
* \Z: **Sztring vége** (az egész szöveg vége).  
  Például: \Z$ illeszkedik a szöveg utolsó karakterére.

**Karakter halmazok:**

* [arn]: Illeszkedik a a, r, vagy n karakterek egyikére.
* [a-n]: Illeszkedik az ASCII kódtábla a és n közötti összes karakterére.
* [a-ck-m]: Illeszkedik bármelyik a, b, c, k, l, vagy m karakterre.
* [^arn]: Illeszkedik bármely karakterre, **kivéve** a, r, vagy n.
* [0-9]: Illeszkedik bármely számjegyre (0–9).
* [a-zA-Z]: Illeszkedik bármely angol betűre (kis- és nagybetűs).
* [0-5][0-9]: Illeszkedik a 00 és 59 közötti kétjegyű számokra.
* [+]: Illeszkedik a + karakterre. Ha karakter halmazban szerepel, akkor már nem speciális jelentéssel bír.

# Re modul függvényeinek összegzése (6.7)

* **re.search()**: Megkeresi az első illeszkedést a szövegben és visszaad egy Match objektumot, ha van illeszkedés.
* **re.findall()**: Visszaadja az összes illeszkedő mintát egy listában.
* **re.split()**: A reguláris kifejezésnek megfelelő illeszkedési pontokon felbontja a szöveget.
* **re.sub()**: Cseréli az illeszkedő mintákat egy másik szövegre.