



# PYTHON 101

## TEMEL PYTHON



# Kullanım Alanları

## a. Veri Bilimi ve Makine Öğrenimi

Python, veri analizi ve yapay zeka için en çok kullanılan dildir. Pandas, NumPy, TensorFlow gibi kütüphaneleri kullanarak veri analizi yapabilir, hatta yapay zeka modelleri geliştirebilirsiniz.

## b. Web Geliştirme

Python'un Django ve Flask gibi güçlü web geliştirme çerçeveleriyle, profesyonel düzeyde web siteleri ve uygulamalar oluşturabilirsiniz. Mesela Instagram, Netflix gibi platformların bazı bölümleri Python kullanılarak geliştirilmiştir.

## c. Oyun Geliştirme

Python, özellikle küçük oyunlar yapmak için idealdir. Pygame gibi kütüphaneler, oyun yapımını kolaylaştırır.

## d. Otomasyon ve Scripting

Günlük rutinlerinizi otomatikleştirmek için Python'u kullanabilirsiniz. Örneğin, bir dosyayı her gün bir yere taşımak veya e-posta göndermek gibi işlemleri otomatikleştirebilirsiniz.

# PYTHON NEDEN YAVAS ?

Python, yorumlanan bir dil olduğu için düşük seviyeli dillere (C, C++, Assembly) kıyasla daha yavaştır.

Ancak Python, çoğu durumda yeterince hızlıdır ve özellikle **iş geliştirme hızı** sağlar; karmaşık bir projeyi kısa sürede tamamlamaya olanak tanır.

Performansın kritik olduğu yerlerde, Python kütüphaneleri (NumPy gibi) genellikle C veya diğer yüksek performanslı dillerde yazılmış çekirdeklerle çalışır. Bu sayede Python'un yavaşlığı büyük ölçüde telafi edilir.

# INTERPRETER - COMPILER NEDİR ?

Derleyici (compiler) ve yorumlayıcı (interpreter) **birer yazılımdır**. Donanımın üzerinde çalışır ve yazdığınız insan tarafından anlaşılabilir kodu (örneğin Python, C veya başka bir dildeki kod) makine tarafından çalıştırılabilir hale getirirler.

Bir **compiler** (derleyici), yazdığımız kodu alır ve bilgisayarın anlayabileceği makine diline (0 ve 1'ler) çevirir. Bu işlem tamamlandıktan sonra programı çalıştırabiliriz. **Kod, önceden tamamen çevrilmiştir.**

Bir **interpreter** (yorumlayıcı), kodu satır satır alır, okur ve çalıştırır. **Kod tamamen çevrilmez, yorumlayıcı her satırı okur ve anında çalıştırır.**

# TEMEL VERİ TURLERİ VE DEĞİSKENLER

Değişken, bir veriyi saklamak için kullandığımız **isimlendirilmiş bir kutu** gibidir. Python'da değişkenler, bir değeri tutmak ve bu değere daha sonra kolayca erişmek için kullanılır. Sadece harf, rakam ve alt çizgi (\_).

Veri Türü	Örnek	Açıklama
<b>int</b> (Tam sayı)	<code>yas = 20</code>	Tam sayılar için kullanılır.
<b>float</b> (Ondalık)	<code>pi = 3.14</code>	Ondalıklı sayılar için kullanılır.
<b>str</b> (Metin)	<code>isim = "Ali"</code>	Yazılar ve metinler için kullanılır.
<b>bool</b> (Mantıksal)	<code>aktif = True</code>	Doğru veya yanlış değerlerini tutar.
<b>list</b> (Liste)	<code>sayilar = [1, 2, 3]</code>	Birden fazla değer içeren bir liste tutar.

# GOMULU FONKSİYONLAR

Python, programcılarının işlerini kolaylaştırmak için önceden tanımlanmış çok sayıda yerleşik fonksiyon sunar. `print()`, `input()`, `len()`, `type()` 'da bu kategoride yer alır.

`print("Hello world")` #ekrana(komut istemi) tırnak içine yazdığımız ifadeyi basar.

`input("Sayı giriniz")` #ekrana sayı giriniz ifadesini basıp klavyeden bir şey girmemizi bekler.

`alınan=input("Sayı giriniz")` #girdiğimiz şeyi "alınan" isminde bir değişkene atar.

`len(alınan)` #alınan ismindeki değişkenin(girdiğimiz ifadenin) uzunluğunu gösterir.


`type(alınan)` #değişkenin veri tipini gösterir.

# KARMASIK VERI TURLERI 1: LISTELER

## . list (Liste)

- **Tanım:** Birden fazla öğeyi bir arada saklayan, sıralı ve değiştirilebilir (mutable) bir veri yapısı.
- **Örnek:**

python

 Kodu kopyala

```
meyveler = ["elma", "armut", "muz"]  
sayilar = [1, 2, 3, 4, 5]  
karisik = [1, "Ali", True]
```

- **Kullanım:** Birden fazla veriyi bir arada tutmak.

# İNDEKSLEME

## . Pozitif İndeksleme

- Python'da **indeksler sıfırdan başlar**.
- İlk elemanın indeksi **0**, ikinci elemanın indeksi **1**, üçüncü elemanın indeksi **2** şeklindedir.

## Negatif İndeksleme

- Python, listenin sonundan başlamak için **negatif indeksleme** sunar.
- Son elemanın indeksi **-1**, sondan bir önceki elemanın indeksi **-2**, vb.

python

```
meyveler = ["elma", "armut", "muz", "çilek"]
```

# İndeks	0	1	2	3
----------	---	---	---	---

Elemanlara erişim:

python

```
print(meyveler[0]) # Çıktı: elma
```

```
print(meyveler[2]) # Çıktı: muz
```



# İNDEKSLEME NEDEN KULLANILIR ?

## a. Belirli Bir Elemanı Erişmek

İndeks, sıralı bir veri yapısından belirli bir elemanı seçmek için kullanılır:

```
sayilar = [10, 20, 30, 40, 50]
```

```
print(sayilar[1]) # Çıktı: 20
```

```
print(sayilar[-3]) # Çıktı: 30
```

## b. Elemanı Değiştirmek

Bir listenin belirli bir indeksindeki elemanı değiştirebilirsiniz:

```
sayilar[2] = 35
```

```
print(sayilar) #Çıktı: [10, 20, 35, 40, 50]:
```

## c. Elemanı Silmek

`del` anahtar kelimesiyle belirli bir indeksteki elemanı silabilirsiniz:

```
del sayilar[3]
```

```
print(sayilar) # Çıktı: [10, 20, 35, 50]
```

# İNDEKS İLE DİLİMLEME (SLICING)

İndeksleme ile bir liste veya string'in belli bir kısmını seçmek için **dilimleme** kullanabilirsiniz. Dilimleme, iki nokta üst üste (:) ile yapılır.

## Temel Dilimleme

```
meyveler = ["elma", "armut", "muz", "çilek", "portakal"]  
print(meyveler[1:4]) # Çıktı: ['armut', 'muz', 'çilek']
```

## Başlangıç veya Bitiş Belirtilmezse

```
print(meyveler[:3]) # Çıktı: ['elma', 'armut', 'muz'] (0'dan başlar)  
print(meyveler[2:]) # Çıktı: ['muz', 'çilek', 'portakal'] (2'den başlar)
```

## Adımlı Dilimleme

- Üçüncü bir parametre ekleyerek, elemanları belirli bir adımla seçebilirsiniz:

```
print(meyveler[::2]) # Çıktı: ['elma', 'muz', 'portakal'] (2 adımla ilerler)
```

# LISTELERİN METODLARI

## `append()` Metodu

- Bir listeye **tek bir eleman eklemek** için kullanılır.
- Eleman her zaman listenin **sonuna** eklenir.

## `extend()` Metodu

- Listeye birden fazla eleman eklemek için kullanılır.
- Listeyi bir başka liste veya iterable (örneğin string) ile genişletir.

## `insert()` Metodu

- Listeye **belirli bir konuma** eleman eklemek için kullanılır.
- İlk parametre: Eklenecek konum (indeks).
- İkinci parametre: Eklenecek eleman.

## `remove()` Metodu

- Listedden **belirli bir değeri** çıkarır.
- İlk bulunduğu değeri çıkarır, eğer değer listede yoksa hata verir.

## `pop()` Metodu

- Listedden bir eleman **indeksine göre** çıkarır.
- İndeks belirtilmezse, **son elemanı çıkarır**.
- Çıkardığı elemanı döndürür.

## `del` Anahtar Kelimesi

- Belirli bir indksi veya bir dilimi silebilir.
- Listeyi tamamen boşaltabilir.

## `clear()` Metodu

- Listenin **tüm elemanlarını** siler.

# DEĞİSTİRİLEMİYEN LİSTE: DEMET

## 1. tuple (Demet)

- **Tanım:** Listeye benzer, ancak **değiştirilemezdir** (immutable).
- **Örnek:**

python

 Kodu kopyala

```
sabit_veriler = (1, 2, 3)
tek_eleman = (5,) # Tek elemanlı tuple için virgöl gerekir.
```


- **Kullanım:** Değiştirilmeyecek veriler için uygundur.

# BASKA BİR VERİ TIPI: SOZLUK

**Tanım:** Anahtar-değer çiftlerinden oluşan veri yapısı.

**Örnek:**

python

 Kodu kopyala

```
ogrenci = {  
    "isim": "Ali",  
    "yas": 21,  
    "dersler": ["Matematik", "Fizik"]  
}
```


**Kullanım:** Karmaşık ilişkileri modellemek için.

# ESSİZ DEĞERLERİ SAKLAMAK: KUME

**Tanım:** Sırasız ve eşsiz (unique) öğelerden oluşan bir veri yapısı.

**Örnek:**

python

 Kodu kopyala

```
sayilar = {1, 2, 3, 4, 5}
tekrarli = {1, 2, 2, 3} # Çıktı: {1, 2, 3}
```

**Kullanım:** Benzersiz değerleri saklamak, kümelerle işlemler yapmak.

# Degiskenlerin Dinamik Yapisi

Python'da bir deęişken bir türde veri tutarken, sonradan başka bir türde veri saklayabilir. Bu esneklik, Python'un dinamik bir dil olmasını sağlar.

```
x = 10 # x şimdi bir tam sayı
```

```
x = "Merhaba" # x şimdi bir metin
```

```
x = 3.14 # x şimdi bir ondalıklı sayı
```

```
a = b = c = 10
```

# OPERATORLER

**Aritmetik Operatörler**

**Karşılaştırma (İlişkisel) Operatörleri**

**Mantıksal Operatörler**

**Atama Operatörleri**



# ARİTMETİK OPERATORLER

Operatör	İşlem	Örnek	Sonuç
+	Toplama	5 + 3	8
-	Çıkarma	5 - 3	2
*	Çarpma	5 * 3	15
/	Bölme	5 / 3	1.666..
//	Tam sayı bölme	5 // 3	1
%	Mod (kalan bulma)	5 % 3	2
**	Üs alma	5 ** 3	125

# KARŞILAŞTIRMA OPERATORLERİ (BOOL ONEM TASIR)

Operatör	Anlam	Örnek	Sonuç
==	Eşit mi?	5 == 3	False
!=	Eşit değil mi?	5 != 3	True
>	Büyük mü?	5 > 3	True
<	Küçük mü?	5 < 3	False
>=	Büyük veya eşit mi?	5 >= 3	True
<=	Küçük veya eşit mi?	5 <= 3	False

# MANTIKSAL OPERATORLER

Karşılaştırma ifadelerinin sonuçlarını birleştirir.

$(x > 5)$  and  $(y < 10)$

Operatör	Anlam	Örnek	Sonuç
and	ve (her iki ifade doğruysa)	$(5 > 3)$ and $(2 > 1)$	True
or	veya (biri doğruysa)	$(5 > 3)$ or $(2 < 1)$	True
not	değil (tersini alır)	not( $5 > 3$ )	False

# ATAMA OPERATORLERİ

Operatör	Anlam	Örnek	Sonuç
=	Atama	x = 5	x = 5
+=	Toplayıp ata	x += 3	x = x + 3
-=	Çıkarıp ata	x -= 3	x = x - 3
*=	Çarpıp ata	x *= 3	x = x * 3
/=	Bölüp ata	x /= 3	x = x / 3
//=	Tam sayı bölüp ata	x //= 3	x = x // 3
%=	Mod alıp ata	x %= 3	x = x % 3
**=	Üs alıp ata	x **= 3	x = x ** 3

# KOSULLU İFADELER, DALLANMALAR

Python'da **koşullu ifadeler**, bir durumun doğru (**True**) veya yanlış (**False**) olmasına göre farklı kod parçalarını çalıştırmamızı sağlar. Bu, programın akışını kontrol etmek için kullanılan temel bir yapıdır. Koşullu ifadeler, karar verme mekanizmasını kodlarımıza ekler.

Normalde yukarıdan aşağı akan kod, if-else bloğu eklendiğinde dallanır. If-Elif,Else girdiğimizi varsayalım, 3 işlem için kod dallanacaktır.

# Basit Bir Kosul

```
python
```

```
sayi = 10
```

```
if sayi > 0:  
    print("Sayı pozitifdir.")
```

### 3 Dala Ayrilmis Kod Parçasi

```
sayi = 0

if sayi > 0:
    print("Sayı pozitifdir.")
elif sayi < 0:
    print("Sayı negatiftir.")
else:
    print("Sayı sıfırdır.")
```

# Karsilastirma- Mantiksal Operatorlerini Kullanalim

```
sayi = 15
```

```
if sayi > 10 and sayi < 20:  
    print("Sayı 10 ile 20 arasında.")
```



# DONGULER

Döngü, bir işlemi **tekrar tekrar yapmak** için kullanılan programlama yapısıdır. Aynı kodu birden fazla kez yazmak yerine, döngüleri kullanarak bu işlemi otomatik hale getirebiliriz. Python'da iki temel döngü vardır:

1. **for döngüsü**
2. **while döngüsü**

Döngüler, bir listedeki elemanlar üzerinde işlem yapmak, belirli bir koşulu kontrol ederek işlemleri tekrarlamak veya bir sayıyı artırarak birden fazla kez işlem yapmak gibi birçok senaryoda kullanılır.

# for Dongusu

**for** döngüsü, **bir koleksiyonun (liste, dizi, string, vb.) elemanları üzerinde sırayla dolaşır** ve her eleman için bir işlem yapar.

```
for eleman in koleksiyon:  
    # Döngüde yapılacak işlem
```

# ÖRNEK

Meyveler listesindeki her elemana (0'dan başlayarak) bir isim ata (bu örnekte "meyve" ismini) ve bu isimle işlemler yap.

```
meyveler = ["elma", "armut", "muz"]  
  
for meyve in meyveler:  
    print(meyve)
```

# Örnek 2: Belirli Bir Aralıkta Döngü

Python'un `range()` fonksiyonu, bir aralık oluşturur. Döngüyü bu aralık üzerinde çalıştırabiliriz.

`range(0,6,1)` #0 dahil 6 dahil değil 1'er art. (Dilimleme sırası ile aynı:  
"Baş,Son,ArtışMiktarı")

```
for i in range(5): # 0'dan 4'e kadar (5 dahil değil)
    print(i)
```

# while Dongusu

**while** döngüsü, **belirli bir koşul doğru olduğu sürece** çalışır. Koşul yanlış olduğunda döngü sona erer.

```
sayi = 0

while sayi < 5:
    print(sayi)
    sayi += 1  # sayi'yi 1 artır
```

# Ornek: Kullanici Girdiyse Calisan Dongu

```
girdi = ""
```

```
while girdi != "tamam":
```

```
    girdi = input("Çıkmak için 'tamam' yazın: ")
```

# Dongulerde Kullanilan Bazi Anahtar Kelimeler

## 1. **break**: Döngüyü Sonlandırır

Döngü belirli bir koşulda hemen sona erdirilir.

```
for i in range(10):  
    if i == 5:  
        break # Döngüyü durdur  
    print(i)
```

## 2. **continue**: Döngünün Bir Sonraki

Adımına Geçer

```
for i in range(5):  
    if i == 2:  
        continue # 2'yi atla  
    print(i)
```

# for ile while Arasindaki Farklar

Özellik	for Döngüsü	while Döngüsü
Kullanım Amacı	Koleksiyonlar ve belirli aralıklar için.	Belirli bir koşul doğru olduğu sürece.
Kapsam	Liste, string, aralık gibi veri yapılarında gezinmek.	Şart bazlı döngüler.
Kullanım Kolaylığı	Daha basit ve okunabilir.	Koşula bağlıdır, bazen daha karmaşık.



# Sonsuz Dongu

```
while True:  
    print("Bu sonsuz bir döngü!")  
    break # Sonsuz döngüyü sonlandırmak için
```

# FONKSIYONLAR - Gömülü Fonksiyonlar

## Veri Türü ve Tip Dönüşüm Fonksiyonları

Fonksiyon	Açıklama
<code>type()</code>	Bir değerin veri türünü döner.
<code>int()</code>	Veriyi tamsayıya dönüştürür.
<code>float()</code>	Veriyi ondalıklı sayıya dönüştürür.
<code>str()</code>	Veriyi metne (string) dönüştürür.
<code>list()</code>	Veriyi listeye dönüştürür.
<code>tuple()</code>	Veriyi demete (tuple) dönüştürür.
<code>set()</code>	Veriyi kümeye dönüştürür.
<code>dict()</code>	Veriyi sözlüğe dönüştürür.
<code>bool()</code>	Veriyi True veya False değerine dönüştürür.

# Matematiksel Fonksiyonlar

Fonksiyon	Açıklama
<code>abs()</code>	Sayının mutlak değerini döner.
<code>pow()</code>	Üs alma işlemi yapar.
<code>round()</code>	Ondalıklı sayıyı yuvarlar.
<code>max()</code>	Bir listedeki en büyük değeri döner.
<code>min()</code>	Bir listedeki en küçük değeri döner.
<code>sum()</code>	Bir listedeki tüm değerlerin toplamını döner.

# DOSYA ISLEME FONKSIYONLARI

Fonksiyon	Açıklama
<code>open()</code>	Bir dosya açar veya oluşturur.
<code>read()</code>	Bir dosyayı okur.
<code>write()</code>	Bir dosyaya veri yazar.
<code>close()</code>	Dosyayı kapatır.

## Örnek:

python

 Kodu kopyala

```
dosya = open("deneme.txt", "w")
dosya.write("Merhaba Python!")
dosya.close()
```

# FONKSİYON TANIMLAMAK

Python'da **fonksiyon**, bilgisayara bir işi nasıl yapacağını söylediğimiz bir "talimatlar paketi"dir. Fonksiyon, sizin yazdığınız veya Python'un hazır sunduğu bir grup koddur. Öyle düşünün ki, bir işi tekrar tekrar yapmak yerine o işi "bir düğmeye basarak" halletmenizi sağlar.

## Neden Fonksiyon Kullanırız?

1. **Tekrarı Önleriz:** Aynı kodu defalarca yazmak yerine, bir kere yazar ve istediğimiz kadar kullanırız.
2. **Kod Daha Düzenli Olur:** Fonksiyonlar, karmaşık işleri küçük parçalara böler ve kodu daha anlaşılır hale getirir.
3. **İşleri Hızlandırırız:** İşimizi kolaylaştırır ve zamandan tasarruf ederiz.

# FONKSİYON TANIMLAMAK

## Fonksiyon Tanımlamak

Fonksiyon tanımlarken şunları yaparız:

1. **def** yazılır: Bu, "Ben bir fonksiyon tanımlıyorum" anlamına gelir.
2. Bir **isim** verilir: Fonksiyonunuzu çağırmak için bir isim belirlenir.
3. Parantez açılır: Fonksiyonun parametreleri (isteğe bağlı) parantez içine yazılır.
4. Fonksiyonun yapacağı işler tanımlanır (kod bloğu).

```
def selam_ver():  
    print("Merhaba!")
```

```
selam_ver() # Çıktı: Merhaba!
```

# Fonksiyonlara Veri Vermek - Parametreler

Bazen bir fonksiyona, ne yapacağını anlaması için bilgi vermeniz gerekir. Bu bilgilere **parametre** denir.

```
def selam_ver(isim):  
    print(f"Merhaba, {isim}!")
```

# Parametre Konusuna Bir Örnek

print() fonksiyonundaki "sep" parametresi, ayrı ayrı girdiğimiz her değer arasına istediğimiz ifadeyi koymamızı sağlar.

```
print("Elma", "Armut", "Muz", sep=" | ")
```

```
Elma | Armut | Muz
```



# RETURN İLE DEĞER DONDURMEK

Fonksiyonların en önemli özelliği olan return konusuna geldik. Bazı fonksiyonlar bir işlem yapar ve sonucu size geri verir. Bu geri verilen sonuca **"return değeri"** denir.

Bir kahve otomatını düşünelim. Kahve otomatı bir **fonksiyon** gibi çalışır.

Kahve otomatına kahve türünü seçersiniz ve kahve hazır olduğunda size geri döner.

"Size geri dönen kahve" fonksiyonun **return ettiği şeydir.**

# RETURN NASIL CALISIR ?

Topla fonksiyonunu oluştururuz, oluşturacağımız değişkene atamak istediğimiz değerin başına “return” ifadesini koyarız ve artık o değer döndürülmeye hazırdır.

```
def topla(a, b):  
    return a + b
```

```
sonuc = topla(3, 5)  
print(sonuc) # Çıktı: 8
```

# Return Konusuna Bir Örnek

input() fonksiyonunu kullanırken, klavyeden aldığımız değeri bir değişkene atayabiliriz. Bu da demek oluyor ki input fonksiyonu bizim klavyeden aldığımız değeri kendi içinde return ediyor.

```
isim = input("Adınızı girin: ")  
print("Merhaba, {}! Hoş geldiniz.".format(isim))
```

```
Adınızı girin: Ayşe  
Merhaba, Ayşe! Hoş geldiniz.
```

# SINIFLAR - NESNEYE YONELIK PROGRAMLAMA (OOP)

Nesneye Yönelik Programlama, gerçek hayattaki nesneleri programlara modellememizi sağlayan bir programlama yöntemidir. Bu yöntemle, bir nesneyi tanımlarken:

- **Özellikleri (Attributes):** Nesnenin sahip olduğu özellikler.
- **Davranışları (Methods):** Nesnenin yapabildiği işler.

Gerçek dünyayı düşünerek program yazmamızı sağlar. **Amaç:** Kodları daha düzenli, yeniden kullanılabilir ve anlaşılır hale getirmektir.

# Gerçek Hayattan Bir Örnek

Bir **araba** düşünelim:

- Özellikleri:
  - Marka: Toyota
  - Renk: Siyah
  - Motor Gücü: 1.6
- Davranışları - Metodlar:
  - Hızlanmak
  - Fren yapmak
  - Kontak açmak

Programlamada bu arabayı bir **sınıf** (class) olarak tanımlarız ve her araba modelini bir **nesne** (object) olarak oluştururuz.

# SINIF (CLASS) NEDİR ?

Sınıf, nesneleri tanımlayan bir şablondur. Şablon diyebiliriz çünkü bu sınıfı kullanarak birçok nesne üretebiliriz.

**Örnek:** "Araba" sınıfı, her araba modelini tanımlayabileceğimiz genel bir şablondur. Her sınıftan farklı özelliklere sahip nesneler oluşturabiliriz (örneğin, siyah bir Toyota ya da kırmızı bir BMW).

# SINIF TANIMLAMA

```
class Araba:  
    pass # Şimdilik boş bir sınıf
```

```
araba1 = Araba()  
araba2 = Araba()
```

# Özellikler

Bir nesnenin sahip olduğu özellikleri tanımlamak için `__init__` metodu kullanılır. Bu metod, nesne oluşturulurken çalışır ve nesneye özellikler ekler. Parametre olarak bu özellikleri ekleriz.

```
class Araba:
    def __init__(self, marka, renk):
        self.marka = marka # Arabanın markası
        self.renk = renk   # Arabanın rengi
```

```
araba1 = Araba("Toyota", "Siyah")
araba2 = Araba("BMW", "Kırmızı")

print(araba1.marka) # Çıktı: Toyota
print(araba2.renk)  # Çıktı: Kırmızı
```



# Davranislar (Metodlar)

Bir sınıfa davranış eklemek için, fonksiyonlar yazılır. Örneğin, arabayı hızlandırmak gibi.

```
class Araba:
    def __init__(self, marka, renk):
        self.marka = marka
        self.renk = renk

    def hizlan(self):
        print(f"{self.marka} hızlanıyor!")
```

```
araba1 = Araba("Toyota", "Siyah")
araba1.hizlan() # Çıktı: Toyota hızlanıyor!
```

# Ödev Takip Sistemi Örneği

**Ödev Takip Sistemi:** Bir öğrenciyi bir nesne olarak düşünebiliriz.

- **Özellikleri:** Adı, Numarası, Ders Listesi
- **Davranışları:** Ders ekleme, ortalama hesaplama

```
class Ogrenci:
    def __init__(self, isim, numara):
        self.isim = isim
        self.numara = numara
        self.dersler = []

    def ders_ekle(self, ders):
        self.dersler.append(ders)
        print(ders + " dersi eklendi!")

    def dersleri_listele(self):
        print(self.isim + " adlı öğrencinin dersleri: " + ', '.join(self.dersler))
```

```
ogrenci1 = Ogrenci("Ali", 12345)
ogrenci1.ders_ekle("Matematik")
ogrenci1.ders_ekle("Fizik")
ogrenci1.dersleri_listele()
```

Matematik dersi eklendi!

Fizik dersi eklendi!

Ali adlı öğrencinin dersleri: Matematik, Fizik

# PYTHON'DA HATA YAKALAMAK (EXCEPTION HANDLING)

Python'da bir program çalışırken bazı hatalarla karşılaşabilir. Örneğin:

- Kullanıcı beklenmeyen bir veri girerse,
- Dosya bulunamazsa,
- Sıfıra bölme gibi matematiksel hatalar olursa.

Bu gibi durumlarda, program **çökmemesi** için hataları **yakalamalı** ve uygun bir şekilde işlem yapmalıdır. Python'da bu işlem **hata yakalama (exception handling)** olarak adlandırılır ve bunu yapmak için **try**, **except** gibi yapılar kullanılır.

# TRY-EXCEPT BLOKLARI

try bloğu içinde yazdığımız kod çalışır, herhangi bir hatada o blok durur ve except bloğu başlar.

Bu kısım ise belirli hataları yakalayıp ona göre işleme devam etmemizi sağlar.

```
try:
    x = int(input("Bir sayı girin: ")) # Kullanıcı string girerse hata oluşur
    print("Girdiğiniz sayının karesi:", x ** 2)
except:
    print("Bir hata oluştu! Lütfen geçerli bir sayı girin.")
```

```
try:
    x = int(input("Bir sayı girin: "))
    y = int(input("Bölmek için bir sayı girin: "))
    sonuc = x / y
    print("Sonuç:", sonuc)
except ValueError:
    print("Lütfen geçerli bir sayı girin!")
except ZeroDivisionError:
    print("Bir sayı sıfıra bölünemez!")
```

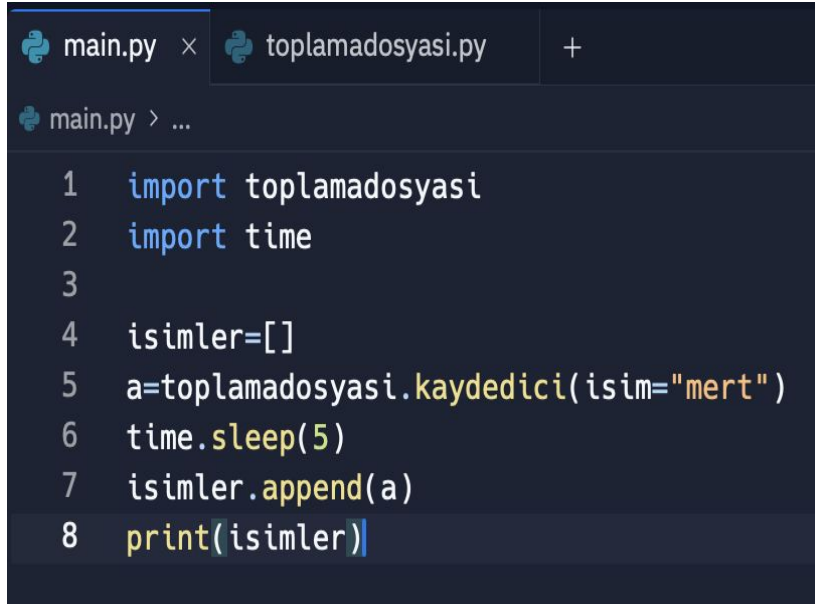
# IMPORT

Python'da `import` anahtar kelimesi, başka bir dosyadaki kodları kendi programınıza dahil etmek için kullanılır. Bu, yazdığınız fonksiyonları veya sınıfları farklı dosyalarda organize ederek tekrar kullanılabilir hale getirir. Şimdi, bunu adım adım açıklayalım ve örneklerle gösterelim.

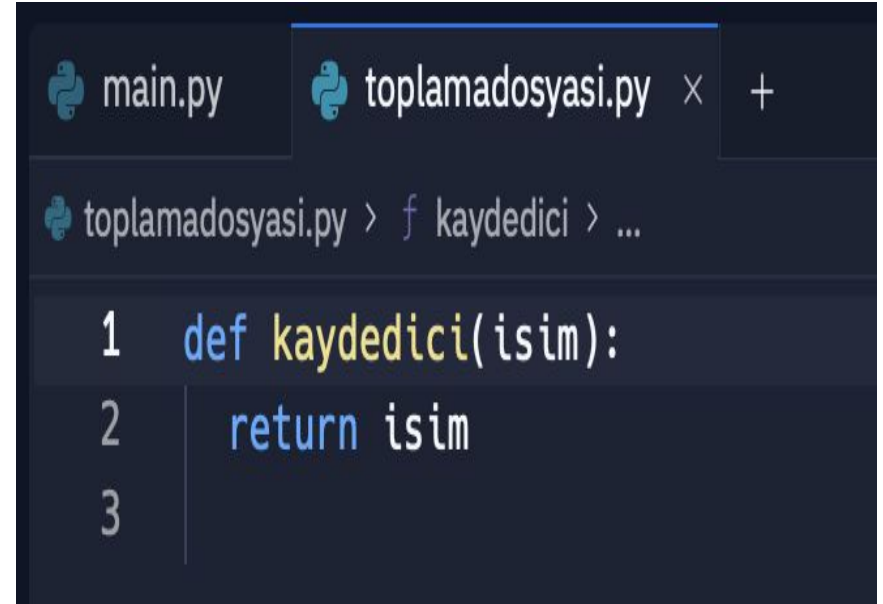
Aynı dizinde 2 dosyamız olduğunu düşünelim. Bunlardan ilki, `main.py` isimli ana kod dosyamız, ikincisi ise `toplamadosyasi.py` ismindeki kaydedici işlemi fonksiyonunu içeren bir dosya.

Öncelikle kendi yazdığımız `toplamadosyasi.py` dosyasını ve python'da hazır yüklenmiş `time` kütüphanesini `import` ediyoruz. (`import toplamadosyasi / from toplamadosyasi import kaydedici`)

# IMPORT ORNEGI



```
1 import toplamadosyasi
2 import time
3
4 isimler=[]
5 a=toplamadosyasi.kaydedici(isim="mert")
6 time.sleep(5)
7 isimler.append(a)
8 print(isimler)
```



```
1 def kaydedici(isim):
2     return isim
3
```

# IMPORT ORNEGI

Burada, `kütüphane_adi.fonksiyon` ifadesi ile kaydedici fonksiyonumuz erişiyor, parametresini veriyor, verdiğimiz parametreyi return etmesini sağlıyor ve bunu `a` değişkenine atıyoruz. Ardından diğer bir kütüphane olan `time` kütüphanesinin `sleep()` fonksiyonu sayesinde 5 saniye bekliyoruz, listelerin bir metodu olan `append` metodu ile `a` değişkenini içi boş listemize ekliyor ve yazdırıyoruz.

# VIRTUAL ENVIRONMENT (VENV) NEDİR ?

Sanal ortam, projeleriniz için bağımsız bir Python ortamı oluşturmanızı sağlar. Bu ortamlar sayesinde:

- Bir projede Python'un belirli bir sürümünü veya kütüphanelerin farklı sürümlerini kullanabilirsiniz.
- Farklı projelerde oluşabilecek uyumsuzlukları engellersiniz.
- Sadece 4 adım !



# NEDEN SANAL ORTAM KULLANIRIZ ?

Python projelerinde bazen farklı kütüphane sürümleri gerekebilir. Örneğin:

- Proje A, NumPy 1.21 sürümünü isterken,
- Proje B, NumPy 1.19 sürümüne ihtiyaç duyabilir.

Sanal ortamlar sayesinde her projeye özel bir ortam yaratarak bu tür sorunlardan kaçınırsınız.

# ANACONDA HAKKINDA

Anaconda, **Python'un özel bir dağıtımıdır** ve özellikle veri bilimi, makine öğrenimi ve büyük veri analizinde kullanılır. Python ile çalışırken, farklı kütüphaneleri ve araçları kurmak bazen zor olabilir. Anaconda, bu süreci kolaylaştıran ve ihtiyacınız olan her şeyi bir arada sunan bir platformdur.

Python dağıtımı, temel Python diline ek olarak belirli kütüphaneler ve araçları bir araya getiren bir paket anlamına gelir.

# SADECE 4 ADIM

1 - Projeni belirle.

2 - Hangi kütüphanelerle çalışacağını araştır.

3 - Çalışacağın kütüphanelerin birbiriyle uyumlu sürümlerini bul. (tercihen ChatGPT bunun için iyi bir araç).

4 - Tüm kütüphanelerin sürümüyle uyumlu python sürümünü bul ve buna uygun bir VENV kur.

# DINLEDIGINIZ ICIN TESEKKURLER

