2.Hafta

VERİ YAPILARI VE FONKSİYONLAR

1) Listeler, Tuple'lar, Set'ler, Dictionary

Python'da verileri düzenli bir şekilde depolamak ve yönetmek için çeşitli **veri yapıları** bulunur. Her birinin kendine özgü özellikleri ve kullanım alanları vardır.

Listeler (Lists):

Listeler, öğelerin **sıralı** (ordered) ve **değiştirilebilir** (mutable) koleksiyonlarıdır. Farklı veri tiplerindeki öğeleri bir arada tutabilirler. Köşeli parantezler [] ile tanımlanır.

- **Sıralı:** Öğelerin belirli bir dizilişi vardır ve bu diziliş korunur. (Python 3.7+ ile garantili)
- **Değiştirilebilir:** Öğeler eklenebilir, çıkarılabilir, değiştirilebilir.
- **Tekrarlayan Öğeler:** Aynı değere sahip birden fazla öğe içerebilir.
- **İndekslenebilir:** Her öğeye bir indeks numarası (0'dan başlayarak) ile erişilebilir.

Örnek:

```
ogrenciler = ["Ali", "Ayşe", "Mehmet"]
notlar = [85, 92, 78, 92]
karisik_liste = ["Elma", 5, True, 3.14]
print(ogrenciler[0])          # Ali (ilk öğe)
print(notlar[-1])             # 92 (son öğe)
ogrenciler[1] = "Fatma"       # Öğeyi değiştirme
print(ogrenciler)             # ['Ali', 'Fatma', 'Mehmet']
ogrenciler.append("Can")      # Yeni öğe ekleme
print(ogrenciler)             # ['Ali', 'Fatma', 'Mehmet', 'Can']
ogrenciler.remove("Ali")      # Öğeyi silme
print(ogrenciler)             # ['Fatma', 'Mehmet', 'Can']
```

Tuple'lar (Tuples):

Tuple'lar, öğelerin **sıralı** ve **değiştirilemez** (immutable) koleksiyonlarıdır. Listelere benzerler ancak bir kere oluşturulduktan sonra öğeleri değiştirilemez, eklenemez veya çıkarılamaz. Parantezler () ile tanımlanır. Tuple'lar genellikle fonksiyonlardan birden fazla değer döndürülürken veya değiştirilmesini istemediğiniz sabit veri kümelerini saklarken kullanılır.

- **Sıralı:** Öğelerin belirli bir dizilişi vardır ve bu diziliş korunur.
- **Değiştirilemez:** Öğeleri değiştirilemez. Bu, verilerin güvenliği açısından önemlidir.
- **Tekrarlayan Öğeler:** Aynı değere sahip birden fazla öğe içerebilir.
- **İndekslenebilir:** Her öğeye bir indeks numarası ile erişilebilir.
- **Örnek:**

```
koordinatlar = (10.5, 20.3)
gunler = ("Pazartesi", "Salı", "Çarşamba")
tek_elemanli_tuple = ("tek_eleman",) # Tek elemanlı tuple için virgül önemli
print(koordinatlar[0]) # 10.5
# gunler[0] = "Pazar" # Hata verir: TypeError: 'tuple' object does not support item assignment
```

Set'ler (Sets):

Set'ler, öğelerin **sırasız** (unordered) ve **benzersiz** (unique) koleksiyonlarıdır. Aynı değere sahip birden fazla öğe içeremezler. Küme parantezleri {} ile tanımlanır veya set() fonksiyonu ile oluşturulur.

- **Sırasız:** Öğelerin belirli bir dizilişi yoktur; bu nedenle indeksleme ile erişilemezler.
- **Değiştirilebilir:** Öğeler eklenebilir veya çıkarılabilir.
- **Benzersiz:** Sadece farklı öğeleri içerirler; tekrar eden öğeler otomatik olarak silinir.

- **Örnek:**
- rakamlar = {1, 2, 3, 4, 3, 2} **# Tekrar edenler otomatik silinir**
- print(rakamlar) **# {1, 2, 3, 4} (sıra değişebilir)**
- renkler = {"kırmızı", "mavi", "sarı"}
- renkler.add("yeşil") **# Öğe ekleme**
- print(renkler) **# {'mavi', 'sarı', 'kırmızı', 'yeşil'} (sıra değişebilir)**
- renkler.remove("kırmızı") **# Öğe silme**
- print(renkler) **# {'mavi', 'sarı', 'yeşil'}**

 # Set'ler küme işlemleri için kullanışlıdır: birleşim, kesişim, fark
- set1 = {1, 2, 3}
- set2 = {3, 4, 5}
- print(set1.union(set2)) **# {1, 2, 3, 4, 5}**
- print(set1.intersection(set2)) **# {3}**

Dictionary (Sözlükler):

Sözlükler, **anahtar-değer (key-value) çiftleri** halinde veri depolayan **sırasız** (Python 3.7+ ile eklenme sırasına göre sıralıdır) ve **değiştirilebilir** koleksiyonlardır. Her anahtar benzersiz olmalıdır. Küme parantezleri {} ile tanımlanır ve anahtarlar ile değerler iki nokta : ile ayrılır.

- **Anahtar-Değer Çiftleri:** Verilere indeks yerine benzersiz anahtarlar aracılığıyla erişilir.
- **Sırasız (eskiden):** Öğelerin belirli bir dizilişi yoktu, ancak Python 3.7 ve sonrası sürümlerde eklenme sırası korunur.
- **Değiştirilebilir:** Öğeler eklenebilir, çıkarılabilir, değiştirilebilir.
- **Benzersiz Anahtarlar:** Her anahtar sadece bir kez kullanılabilir. Değerler tekrarlanabilir.

Örnek:
- kullanici = {
- "ad": "Deniz",
- "soyad": "Ateş",
- "yas": 30,
- "sehir": "İstanbul"
- }
- print(kullanici["ad"]) **# Deniz**
- print(kullanici.get("yas")) **# 30 (Güvenli erişim, anahtar yoksa None döner)**
- kullanici["sehir"] = "Ankara" **# Değeri değiştirme**
- print(kullanici) **# {'ad': 'Deniz', 'soyad': 'Ateş', 'yas': 30, 'sehir': 'Ankara'}**
- kullanici["meslek"] = "Mühendis" **# Yeni anahtar-değer çifti ekleme**
- print(kullanici) **# {'ad': 'Deniz', ..., 'meslek': 'Mühendis'}**
- del kullanici["yas"] **# Öğeyi silme**
- print(kullanici) **# {'ad': 'Deniz', 'soyad': 'Ateş', 'sehir': 'Ankara', 'meslek': 'Mühendis'}**

 # Anahtarlar ve Değerlere Erişim

- print(kullanici.keys()) **# dict_keys(['ad', 'soyad', 'sehir', 'meslek'])**
- print(kullanici.values()) **# dict_values(['Deniz', 'Ateş', 'Ankara', 'Mühendis'])**
- print(kullanici.items()) **# dict_items([('ad', 'Deniz'), ('soyad', 'Ateş'), ...])**

2) Veri Yapılarının Metotları

Her veri yapısının, üzerinde çeşitli işlemler yapmanızı sağlayan kendi özel **metotları** vardır.

- **Listelerin Metotları:**
  - **append():** Sonuna öğe ekler.
  - **extend():** Başka bir listenin öğelerini ekler.
  - **insert(index, element):** Belirtilen indekse öğe ekler.
  - **remove(element):** Belirtilen ilk öğeyi siler.
  - **pop(index):** Belirtilen indeksteki öğeyi siler ve döndürür. (indeks verilmezse son öğeyi)
  - **clear():** Tüm öğeleri siler.
  - **index(element):** Öğenin ilk indeksini döndürür.
  - **count(element):** Öğenin kaç kez geçtiğini sayar.
  - **sort():** Listeyi sıralar.
  - **reverse():** Listeyi ters çevirir.

- **Tuple'ların Metotları:** (Değiştirilemez oldukları için daha azdır)
  - **count(element):** Öğenin kaç kez geçtiğini sayar.
  - **index(element):** Öğenin ilk indeksini döndürür.

- **Set'lerin Metotları:**
  - **add(element):** Öğeyi sete ekler.
  - **remove(element):** Öğeyi siler (yoksa hata verir).
  - **discard(element):** Öğeyi siler (yoksa hata vermez).
  - **pop ():** Rastgele bir öğeyi siler ve döndürür.
  - **clear():** Tüm öğeleri siler.
  - **union(other_set):** İki kümenin birleşimini döndürür.
  - **intersection(other_set):** İki kümenin kesişimini döndürür.
  - **difference(other_set):** Birinci kümede olup ikinci kümede olmayan öğeleri döndürür.
  - **symmetric_difference(other_set):** İki kümenin ortak olmayan öğelerini döndürür.

- **Dictionary'lerin Metotları:**
    - **keys():** Tüm anahtarları döndürür.
    - **values():** Tüm değerleri döndürür.
    - **items():** Tüm anahtar-değer çiftlerini tuple olarak döndürür.
    - **get(key, default_value):** Belirtilen anahtarın değerini döndürür, yoksa default_value'yu.
    - **pop(key, default_value):** Belirtilen anahtarı ve değerini siler ve değeri döndürür.
    - **update(other_dict):** Başka bir sözlükteki anahtar-değer çiftlerini mevcut sözlüğe ekler veya günceller.
    - **clear():** Tüm öğeleri siler.

3) Fonksiyon Tanımlama

Fonksiyonlar, belirli bir görevi yerine getiren, yeniden kullanılabilir kod bloklarıdır. Programınızı daha düzenli, okunabilir ve yönetilebilir hale getirirler. Bir fonksiyon tanımlamak için **def** anahtar kelimesi kullanılır.

Örnek:

```
def selamlama():
    """Kullanıcıyı selamlayan basit bir fonksiyon."""
    print("Merhaba!")

# Fonksiyonu çağırma
selamlama() # Merhaba!

def kare_al(sayi):
    """Verilen sayının karesini hesaplar."""
    return sayi * sayi

sonuc = kare_al(5)
print(sonuc) # 25
```

4) Parametre Türleri:

Fonksiyonlar, kendilerine gönderilen bilgilere **parametreler** aracılığıyla erişebilir. Python'da çeşitli parametre türleri bulunur:

1. Varsayılan Parametreler (Default Parameters):

Bir parametreye varsayılan bir değer atayabilirsiniz. Eğer fonksiyon çağrılırken bu parametreye bir değer verilmezse, varsayılan değeri kullanılır. Varsayılan parametreler her zaman parametre listesinin sonunda tanımlanmalıdır.

Örnek:

```
def selamlama_varsayilan(isim="Misafir"):
    print(f"Merhaba, {isim}!")

selamlama_varsayilan()          # Merhaba, Misafir!
selamlama_varsayilan("Ayşe")    # Merhaba, Ayşe!
```

2. Değişken Sayıda Konumsal Argümanlar (*args):

***args**, bir fonksiyona **değişken sayıda konumsal (sıralı)** argüman göndermenizi sağlar. Bu argümanlar, fonksiyon içinde bir **tuple** olarak alınır.

Örnek:

```
def toplama(*sayilar):
    """Değişken sayıda sayıyı toplar."""
    toplam = 0
    for sayi in sayilar:
        toplam += sayi
    return toplam

print(toplama(1, 2))          # 3
print(toplama(10, 20, 30))    # 60
print(toplama(5))             # 5
```

3. Değişken Sayıda Anahtar Kelime Argümanları (**kwargs):

****kwargs**, bir fonksiyona **değişken sayıda anahtar-değer (key-value)** çifti (keyword arguments) göndermenizi sağlar. Bu argümanlar, fonksiyon içinde bir **sözlük (dictionary)** olarak alınır.

***args** ve ****kwargs**, özellikle bir fonksiyona ne kadar veya hangi türde argüman geleceğini tam olarak bilmediğiniz durumlarda veya kütüphane/framework geliştirirken esneklik sağlar.

Örnek:

```
def kullanici_bilgisi(**bilgiler):
    """Kullanıcı bilgilerini sözlük olarak alır ve yazdırır."""
    print("Kullanıcı Bilgileri:")
    for anahtar, deger in bilgiler.items():
        print(f"{anahtar.replace('_', ' ').capitalize()}: {deger}")

kullanici_bilgisi(ad="Can", soyad="Yılmaz", yas=28, sehir="İzmir")

# Kullanıcı Bilgileri:
# Ad: Can
# Soyad: Yılmaz
# Yas: 28
# Sehir: İzmir

kullanici_bilgisi(urun="Laptop", fiyat=15000)

# Kullanıcı Bilgileri:
# Urun: Laptop
# Fiyat: 15000
```

5) return Yapısı

return ifadesi, bir fonksiyondan bir değer döndürmek için kullanılır. Fonksiyon bir **return** ifadesine ulaştığında, çalışmayı durdurur ve döndürülen değeri çağrı yapılan yere geri gönderir. Eğer bir fonksiyonda **return** ifadesi yoksa veya boş bir **return** ifadesi varsa, varsayılan olarak **None** değeri döndürülür.

Örnek:

```
def cift_mi_tek_mi(sayi):
    if sayi % 2 == 0:
        return "Çift"
    else:
        return "Tek"

sonuc1 = cift_mi_tek_mi(7)
print(sonuc1)        # Tek

sonuc2 = cift_mi_tek_mi(10)
print(sonuc2)        # Çift

def hicbir_sey_dondurmez():
    print("Sadece ekrana yazdırır.")

bos_deger = hicbir_sey_dondurmez()
print(bos_deger)          # Sadece ekrana yazdırır.
                          # None (Fonksiyon açıkça return etmediği için None döner)
```

!!! Bir fonksiyondan birden fazla değer döndürmek için virgülle ayırarak tuple olarak döndürebilirsiniz:

Örnek:

```
def hesapla(x, y):
    toplam = x + y
    carpim = x * y
    return toplam, carpim          # Tuple olarak döner
t, c = hesapla(4, 5)
print(f"Toplam: {t}, Çarpım: {c}")    # Toplam: 9, Çarpım: 20
```

6) Lambda Fonksiyonları

Lambda fonksiyonları, tek bir ifade içeren, küçük, anonim (isimsiz) fonksiyonlardır. **def** ile tanımlanan normal fonksiyonlar gibi değildirler. Genellikle kısa süreli ve tek kullanımlık işlemler için kullanılırlar. Lambda fonksiyonları, özellikle **map(), filter() ve sorted()** gibi yüksek seviyeli fonksiyonlarla birlikte kullanıldığında kodun daha kısa ve okunabilir olmasını sağlar.

```
# Normal fonksiyon
def kare_al_normal(x):
    return x * x

# Lambda fonksiyonu
kare_al_lambda = lambda x: x * x
print(kare_al_normal(7))          # 49
print(kare_al_lambda(7))          # 49
```

 # Filter fonksiyonu ile kullanım

- sayilar = [1, 2, 3, 4, 5, 6]
- cift_sayilar = list(filter(lambda x: x % 2 == 0, sayilar))
- print(cift_sayilar) **# [2, 4, 6]**

 # Map fonksiyonu ile kullanım

- kareleri = list(map(lambda x: x**2, sayilar))
- print(kareleri) **# [1, 4, 9, 16, 25, 36]**

7) Fonksiyon İç İçe Kullanımı (Nested Functions)

Python'da bir fonksiyonun içinde başka bir fonksiyon tanımlayabilirsiniz. İçteki fonksiyona **iç fonksiyon** veya **içerideki fonksiyon** denir. İç fonksiyon, yalnızca dış fonksiyonun kapsamında kullanılabilir ve dış fonksiyonun değişkenlerine erişebilir **(closure).**

İç içe fonksiyonlar, özellikle bir fonksiyonun sadece belirli bir yerde kullanılması gerektiğinde veya bir tür yardımcı fonksiyon olarak tanımlandığında yararlıdır. Ayrıca, **kapanımlar (closures)** ve **dekoratörler** gibi ileri seviye konuların temelini oluştururlar.

Örnek:

- def dis_fonksiyon(mesaj):
  # Dış fonksiyonun değişkeni
- x = "Merhaba"

- def ic_fonksiyon():
  # İç fonksiyon dış fonksiyonun değişkenine erişir
- print(f"{x}, {mesaj}!")
- ic_fonksiyon() **# İç fonksiyonu dış fonksiyon içinde çağır**
- dis_fonksiyon("dünya") **# Merhaba, dünya!**

 # ic_fonksiyon() # Hata verir: NameError: name 'ic_fonksiyon' is not defined

8) Scope (Kapsam - Global, Local)

Değişkenlerin programın hangi kısımlarından erişilebilir olduğunu belirleyen kural setine **kapsam (scope)** denir. Python'da temel olarak iki ana kapsam türü vardır:

1. Yerel Kapsam (Local Scope)

Bir fonksiyonun içinde tanımlanan değişkenler, o fonksiyona özgü **yerel değişkenlerdir**. Bu değişkenlere yalnızca tanımlandıkları fonksiyonun içinden erişilebilir. Fonksiyon sona erdiğinde yerel değişkenler bellekten silinir.

2. Küresel Kapsam (Global Scope)

Programın en üst seviyesinde, yani herhangi bir fonksiyonun veya sınıfın dışında tanımlanan değişkenler **küresel değişkenlerdir**. Bu değişkenlere programın her yerinden (hem fonksiyonların içinden hem de dışından) erişilebilir.

global Anahtar Kelimesi: Bir fonksiyon içinde küresel bir değişkeni değiştirmek isterseniz, o değişkenin küresel olduğunu belirtmek için global anahtar kelimesini kullanmanız gerekir. Aksi takdirde, Python otomatik olarak aynı isimde yeni bir yerel değişken oluşturur.

!!! Genel olarak, global anahtar kelimesinin sıkça kullanılması önerilmez, çünkü bu kodun okunabilirliğini ve hataların takibini zorlaştırabilir. Fonksiyonlar genellikle dışarıdan girdi alıp çıktı döndürerek etkileşim kurmalıdır.