**Veri Tipleri ve Veri Girişi**

**input() Fonksiyonu**

Kullanıcıdan veri almamızı sağlar. Alınan veriler her zaman **string** tipindedir. Sayısal olarak kullanacağımız bir veri alıyor ve aritmetik işlemler yapacaksak **int()** ‘a dönüştürmeliyiz.

**Veri Tiplerine Dönüştürmek**

**int() :**  Uygun olan verileri sayısal , integer veri tipine dönüştürmek için kullanılır.

**str() :** Verileri string, yani metin veri tipine dönüştürmek için kullanılır.

**float() :** Tamsayı, noktalı sayısal değerleri olacak şekilde dönüştürmek için kullanılır.

**complex() :** Karmaşık sayılara ait veri tpine dönüştürmemizi sağlar.

Her sayı bir karakter dizisine dönüştürülebilir fakat, karakterler sayısal değerlere dönüştürülemez.

**len()**, String türündeki verilerin uzunluğu, karakter sayısını verir.

String değerleri **split()** metodu ile bölebiliriz. Bölüm sonrası çıktı **list()** türünde olacaktır.

>> kardiz = "İstanbul Büyükşehir Belediyesi"

>> print(kardiz.split())

['İstanbul', 'Büyükşehir', 'Belediyesi']

**print() Fonksiyonu**

Ekrana değer yazdırmak için kullanılır.

**format() Metodu**

Kullanıcıya yansıtılacak değişkenlerin daha nizami bir şekilde kod içerisine yazılmasına olanak sağlar. Örneğin doldurulması gereken yerleri olan bir dilekçe taslağında ilgili yerlere **{}** işareti koyarak, sırasıyla girilen verileri, yine sırasıyla ilgili yerlere ekler.

>> print(" {} ve {} iyi bir ikilidir".format("Python", "Django"))

>> metin = "{} ve {} iyi bir ikilidir"

>> metin.format("Python", "Django")

Python ve Django iyi bir ikilidir

**eval() ve exec() Fonksiyonları**

**eval()** : Kullanıcının girdiği her veriyi bir Python komutu olarak algılar ve bu veriyi işleme sokar. Kullanılacağı durumlarda sıkı bir kontrol mekanizmasının olması gerekmektedir.

**exec()** : eval() ile yapamadığımız bazı şeyleri yapamızı sağlar. Bu fonksiyon yardımıyla karakter dizileri içerisinde çok kapsamlı Python kodları çalıştırabiliriz. Örneğin eval() içerisinde değişken tanımlaması yapamazken, exec() içerisinde bunu yapabiliriz.

**Liste ve Demetler**

**Liste**

**list(),** içinde birden fazla veri barındıran fonksiyonlardır. Barındırdığı verilere **index** üzerinden, **for** ya da **map** fonksiyonları ile erişebiliriz. Map’ ın çıktısı objectir. İç içe liste eklemesi yapabiliriz.

>> liste = ["öğe1", "öğe2", "öğe3"]

Listenin elemanlarına erişmek için **for** döngüsünü kullanırız.

>> meyveler = ["elma","armut","karpuz"]

>> for i in range(len(meyveler)):

>>    print("{}. {}".format(i,meyveler[i]))

1. elma
2. armut
3. karpuz

Öğe değişikliği yapmak istersek, liste içerisinden ilgili index değerine atama yaparız.

>> meyveler = ["elma","armut","karpuz"]

>> meyveler[0] = "çilek"

>> print(meyveler)

['çilek', 'armut', 'karpuz']

**len()** fonksiyonu liste içi veri tiplerinin sayısını söyler.

>> diller = ["İngilizce", "Fransızca", "Türkçe", "İtalyanca"]

>> print(len(diller))

Listeden öğe silmek için **del** ifadesini kullanırız.

>> meyveler = ["elma","armut","karpuz"]

>> del meyveler[-1]

**Liste Üreteçleri (****List Comprehensions)**, kodları satır satır yazmak yerine tek satırda daha kısa halletmemizi sağlar. İç içe ne kadar çok kod yazarsak, işlev ve okunaklığını kaybeder.

>> liste = [i for i in range(1000) if i % 2 == 0]

Normalde yazmak isteseydik şöyle olacaktı.

>> liste = []

>> for i in range(1000):

>>    if i % 2 == 0:

>>        liste += [i]

**append()**, listeye öğe eklemek için kullanırız. Her seferinde tek öğe eklenebildiği için çoklu eklemelerde **for** döngüsü kullanmak gerekir.

>> yeni\_meyveler = ["kavun","kiraz"]

>> for i in yeni\_meyveler:

>>    meyveler.append(i)

**extend()**, değişkeni parçalara ayırarak listeye ekler.

>> meyveler.extend("ayva")

>> print(meyveler)

['elma', 'armut', 'kavun', 'kiraz', 'a', 'y', 'v', 'a']

**insert()**, listenin sonuna ya da belirtilen yere öğe ekler.

>> yeni\_meyveler.insert(0,"Muşluma")

**remove()**, belirttiğimiz öğeyi listeden siler.

>> yeni\_meyveler.remove("Muşluma")

**reversed()**, listeyi tersten tekrar oluşturur. Çıktısı **ist\_reverseiterator** nesnesidir. İçeriğine erişmek için farklı yöntemler mevcuttur. En basit yöntem [::**-1]** kullanımıdır.

>> reversed(meyveler)

>> print(\*reversed(meyveler))

a v y a kiraz kavun armut elma

>> print(list(reversed(meyveler))

['a', 'v', 'y', 'a', 'kiraz', 'kavun', 'armut', 'elma']

**reversed(),** yerine **reverse()** de kullanılır.

**pop()**, parametresiz kullanılırsa listenin son öğesini siler.

**sort()**, parametreye bağlı olarak liste içi sıralama yapar.

**index()**, istenilen öğenin liste içerisindeki konumunu verir.

**count()**, öğenin liste içerisindeki tekrar sayısını verir.

**copy()**, listeler arası etkileşim olmadan kopyalama yapar.

**clear()**, listenin tüm öğelerini siler.

**List Comprehesion**

Liste oluşturmanın farklı bir yoludur.

>> num = [(x,y) for x in range(3) for y in range(2)]

**range()**, belirtilen aralıktaki sayıları listeler. 3. değer olarak atlama değeri alır.

**Demetler**

**tuble()**, liste ile aynı yapıda olup, aynı fonksiyonları kullanırlar. Listeler değiştirilebilir ( **mutable** ) bir veri tipi iken, demetler değiştirilemez ( **immutable** ) bir veri tipidir. Demet verilerine erişim yine index ile yapılır.

>> demet = ("ahmet", "mehmet", 23, 45)

>> demet = ('ahmet',)

>> demet = 'ahmet',

**index()**, demet içerisinde bulunan öğenin konumunu söyler.

>> demet.index("ahmet")

**count()**, öğenin demet içerisindeki tekrar sayısını verir.

>> demet.count("ahmet")

**Sözlük**

**dict**, veriyi **key** ve **value** şeklinde verileri saklayabileceğimiz bir veri yapısıdır. Sözlük öğelerine erişmek için anahtar verilerini kullanırız.

>> sözlük = {}

>> kelimeler = {"kitap": "book"}

>> print(kelimeler["kitap"])

book

Sözlük yapısında iç içe farklı veri tiplerinin barındığı yapılar oluşturabiliriz.

>> kişiler = {"Ahmet Özkoparan": {"Memleket": "İstanbul","Meslek"  : "Öğretmen",

                                "Yaş"     : 34},

           "Mehmet Yağız": {"Memleket": "Adana","Meslek"  : "Mühendis",

                                "Yaş"     : 40}}

>> print(kişiler["Mehmet Yağız"]["Memleket"])

Kullanıcıdan alınan bilgi ile verilere erişmek istersek, sözlük değerlerine erişilmesi gerekmektedir.

>> arama = input("isim")

>> ayrıntı = input("Memleket/Meslek/Yaş")

>> print(kişiler[arama][ayrıntı])

Sözlük yapılarında öğe eklemek istersek, anahtar seçimi yapılarak ona ekleme yaparız. Aynı durum değişiklik içinde geçerlidir.

>> kisiler["Ali"]["Memleket"] = "Kocaeli"

>> notlar["Ahmet"] = 65

**Sözlük Metotları**

**keys()**, sözlüğün sadece anahtarlarını almak istediğimiz de kullanırız.

>> print(kişiler.keys())

Memleket/Meslek/Yaş

Sözlük anahtarlarını karakter dizisine dönüştürmek için **join()** metodunu kullanmalıyız.

>> kardiz = ', '.join(sözlük.keys())

**values()**, sözlüğün değerlerine erişmemizi sağlar. Çıktıları başka veri tiplerine dönüştürebiliriz. Değerler **int** ise karakter dizisine dönüştürme sırasında hata alabiliriz.

>> print(sözlük.values())

>> kardiz = ", ".join([str(i) for i in sözlük.values()])

**items()**, sözlüğün hem anahtarlarını hem de değerlerini aynı anda almamızı sağlar.

>> sözlük.items()

>> for anahtar, değer in sözlük.items():

>>    print("{} = {}".format(anahtar, değer))

**get()**, sözlük içerisinde istenilen verinin olup olmadığını kullanıcıya döndürür. İki adet argüman alır. Birinci argüman sorgulamak istediğimiz sözlük öğesidir. İkinci argüman ise bu öğenin sözlükte bulunmadığı durumda kullanıcıya hangi mesajın gösterileceğini belirtir.

>> ing\_sözlük = {"dil": "language", "elma": "apple", "masa": "table"}

>> sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

>> print(ing\_sözlük.get(sorgu, "Bu kelime veritabanımızda yoktur!"))

**clear()**, sözlükte bulunan öğeleri temizler.

**copy()**, sözlüğü kopyalayıp, yedeğini oluşturur.

**fromkeys()**, mevcut sözlük üzerinde işlem yapmaz. Görevi yeni sözlük oluşturmaktır. Yeni sözlüğün anahtarlarını da önceki sözlükten alır.

>> elemanlar = "Ahmet", "Mehmet", "Can"

>> adresler = dict.fromkeys(elemanlar, "Kadıköy")

**pop()**, parantez içerisine yazılan öğeyi siler ve öğenin değerini ekrana yansıtır. Eğer silmeye çalıştığımız anahtar sözlükte yoksa Python bize bir hata mesajı gösterir.

>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>> sepet.pop("meyveler")

>> sepet.pop("tatlılar", "Silinecek öğe yok!")

**setdefault()**, sözlük içerisinde yeni bir anahtar oluşturacağımız zaman, aynı isimde anahtarın olup olmadığını kontrol eder ve varsa uyarı verir. Yoksa da oluşturma işlemini tamamlar.

>> sepet = {"meyveler": ("elma", "armut")}

>> sepet.setdefault("içecekler", ("su", "kola"))

**update()**, oluşturduğumuz sözlükleri yeni verilerle günceller.

>> stok = {"elma": 5, "armut": 10, "peynir": 6}

>> yeni\_stok = {"elma": 3, "armut": 20, "peynir": 8, "sucuk": 6}

>> stok.update(yeni\_stok)

**Koşullu Durumlar**

**if**, eğer anlamındadır. Ardından gelen koşul sağlanırsa, işlemi gerçekleştirir.

**elif**, **if** değimi ile brikte kullanılır. Belirtilen koşullara göre işlem gerçekleştirmektedir.

**else**, kendinden önce gelen şartlar sağlanmazsa gerçekleşecek olan deyim, işlemdir.

>> boy = int(input("boyunuz kaç cm?"))

>> if boy < 170:

>>    print("boyunuz kısa")

>> elif boy < 180:

>>    print("boyunuz normal")

>> else:

>>    print("boyunuz uzun")

**elif** yerine hepsini **if** yazsaydık, girdi değeri hepsinde ayrı ayrı değerlendirilecek ve uygun olan koşulların hepsi çalışacaktı. **elif** sayesinde sadece doğru olan koşul çalışacaktır.

**Döngüler**

**While**

Belirtilen koşul sağlandığı sürece devam eder.

>> while True:

>>    soru = input("Çıkmak için q ya basınız")

>>    if soru == "q":

>>        print("Çıkış yapılıyor")

>>        break

**For**

While ile benzer yapıdadır. Belirtilen döngü içerisinde işlemleri yapar.

>> sayılar = "123456"

>> for i in sayılar:

>>    if int(i) > 3:

>>        print(i)

**Pass**

Kod **pass** değerine geldiğinde görmezden gelir ve devam eder. Genelde taslak aşamasında olan kod bloklarında kullanılır.

>> if not parola:

>>    pass

**Break**

Devam eden süreci kesintiye uğratır. Her zaman bir döngünün içinde yer almalıdır.

>> for val in "pythontr":

>>    if val == "t":

>>        break

>>    print(val)

>> print("Bitti")

p

y

Bitti

**Continue**

Ardından gelen her şeyin es geçilip, döngüyü devam ettirir.

>> for val in "hontr":

>>    if val == "t":

>>        continue

>>    print(val)

h

o

n

r

**Hata Ayıklama**

**try… except…**

>> try:

>>    hata verebileceğini bildiğimiz kodlar

>> except HataAdı: #HataAdı yazmadan da geçebiliriz.

>>    hata durumunda yapılacak işlem

**try… except… as…**

Kullanıcıya olası bir hata durumunda hem kendi yazdığınız hata mesajını, hem de özgün hata mesajını göstermek istiyorsak kullanırız..

>> try:

>>    print("Sayı")

>> except ValueError as hata:

>>    print("Sadece sayı girin!")

>>    print("orijinal hata mesajı: ", hata)

**try… except… else…**

Herhangi bir hata oluşmadıysa yürütülecek bir kod bloğu tanımlamak için else anahtar sözcüğünü kullanabiliriz.

>> try:

>>  print("Merhaba")

>> except:

>>  print("Bir şeyler yanlış gitti ")

>> else:

>>  print("Hiçbir şey ters gitmedi ")

**try… except… finally…**

En önemli özelliği, programın çalışması sırasında herhangi bir hata gerçekleşse de gerçekleşmese de işletilecek olmasıdır. Eğer yazdığınız programda mutlaka ama mutlaka işletilmesi gereken bir kısım varsa, o kısmı **finally** bloğu içine yazabilirsiniz.

>> try:

>>  print(x)

>> except:

>>  print("Bir şeyler yanlış gitti ")

>> finally:

>>  print("”try except” bitti ")

**raise**

Kullanıcının yaptığı bir işlem normal şartlar altında hata vermeyecek olsa bile biz ona ‘Python tarzı’ bir hata mesajı göstermek isteyebiliriz.

>> if bölünen == 23:

>>    raise Exception("Bu programda 23 sayısını görmek istemiyorum!")

**assert**

Bazen programımızda bir hata yaptığımızda bu hatayı bulmakta zorlanabiliriz. Böyle durumlarda **assert** ifadesini kullanabiliriz. **assert** ifadesi aynı zamanda hata yükseltmenin kısa bir yoludur. Ancak **assert** ifadesini kullanarak sadece **AssertionError** türünde bir hata yükseltebiliriz. Normalde **raise** kullanmamız daha doğru olacaktır. Dediğimiz gibi **assert** ifadesi hızlı bir şekilde kodumuzdaki hataları belirlemek için kullanılır.

>> assert ifade , mesaj

Burada ifade bir **bool** yani **True** veya **False** olabileceği gibi, (aynı if’deki gibi) bool fonksiyonu ile birlikte kullanılabilecek bir nesne de olabilir. Sonuç olarak eğer ifade’nin değeri True ise **assert** ifademiz çalışmayacak, False ise çalışacaktır.

**Temel Dosya İşlemleri**

**Dosya Oluşturmak**

**open()** fonksiyonu ile sağlanır. Farklı parametreleri de mevcuttur. Yazma kipinde açmak için w kullanılır ve içi boş bir dosya oluşturur.

>> f = open(dosya\_adı, kip)

>> with open("falanca.txt", "r+") as f:

Hangi dizinde bulunuyorsa, oraya dosyayı oluşturacaktır. Farklı bir dizinde oluşturmak için konumu belirtmemiz gerekir. Dizin bilgilerini girerken, ters taksim (**\**) yerine düz taksim (**/**) kullanmalıyız. Veya ters taksim işaretini çiftleyebiliriz.

**Dosyaya Yazmak**

Oluşturduğumuz boş dosyanın içerisine yazı yazalım. Dosyaya yazı yazmadan önce açmayı ve işlem bitince de kapatmayı unutmayın.

>> ths = open("tahsilat\_dosyası.txt", "w")

>> ths.write("Halil Pazarlama: 120.000 TL")

>> ths.close()

**Dosya Okumak**

Dosyayı okuma kipinde açmak için **r** harfi kullanılır. Harf belirtmeden de sadece adını yazarsak, Python okuma kipinde açtığımızı anlayacaktır.

>> fihrist = open("fihrist.txt", "r")

>> fihrist = open("fihrist.txt")

Okuma kipi için üç ayrı metot bulunmaktadır.

**read()**, direkt dosya içini okur.

**readline()**, sadece ilk satırı okur.

**readlines()**, satır satır okuma yapar. Tekrar çalıştırmak istediğimiz zaman okuyacağı satır kaldığı için boş veri döndürecektir.

>> print(fihrist.read())

>> print(fihrist.readline ())

>> print(fihrist.readline())

**Dosyaları Otomatik Kapatma**

**try**... **except**... **finally**... bloklarından yararlanmak

>> try:

>>    dosya = open("dosyaadı", "r")

>>    #...burada dosyayla bazı işlemler yapıyoruz...

>>    #...ve ansızın bir hata oluşuyor...

>> except IOError:

>>    print("bir hata oluştu!")

>> finally:

>>    dosya.close()

**with** kullanımı

>> with open("dosya\_adı", "r") as dosya:

>>    print(dosya.read())

Dosyayı İleri-Geri Sarmak

Dosyayı okurken imlecin sona kadar devam edip, başa dönmediğini biliyoruz. Dosyayı tamamen okuduktan sonra başa dönmek istersek eğer **seek()** metodunu kullanmamız gerekmektedir.

**seek()** içerisine yazacağımız değer hangi noktadan itibaren geri döneceğimizi adresler.

>> f = open("python.txt")

>> f.read()

>> f.seek(0)

>> f.read()

İmlecin hangi konumda olduğunu bilmiyor ve öğrenmek istiyorsak, **tell()** metodunu kullanırız.

>> f.tell()

**Dosyaya Erişme Kipleri**

* **r** - Okuma, varsayılan değerdir, dosyayı okumak için açar. Dosya yoksa hata verir.
* **a** - Ekleme, dosyayı veri eklemek için açar, dosya yoksa yeni bir tane oluşturur.
* **w** - Yazma, dosyayı yazmak için açar, dosya yoksa yeni bir tane oluşturur.
* **x** - Oluşturma, zaten bir dosya varsa hata verir.

**Fonksiyonlar**

Proje içerisinde tekrar eden kodları, fonksiyon haline getirebiliriz. Bu sayede kullanımı kolay ve tekrar etmeden projemizi oluştururuz.

Fonksiyon tanımı için **def** kullanılır. Fonksiyon ile oluşan sonucu, değişkene atayabilmemiz için fonksiyon içerisinde tanım yapılması gerekmektedir. Bunun için **return** kullanılır

>> def facto(n):

>>    if (n == 0):

>>        return 1

>>    return n \* facto(n-1)

>> print(facto(5))

120

>> def indirim\_yap(fiyat, yuzde = 20):

>>      indirim\_miktarı = fiyat \* (yuzde / 100)

>>      indirimli\_tutar = fiyat - indirim\_miktarı

>>      print (f"İndirimli tutar: {indirimli\_tutar}")#str ile sayısalı kullanabiliriz.

>> indirim\_yap(50)

İndirimli tutar: 40.0

Üstteki örnekte değişkene bir değer atadık. Fonskiyon çağrıldığında buna atıf olmazsa, atamadaki değer dönecektir.

**Argüman**

Fonksiyonun alacağı parametre sayısı belirsiz ise argüman kullanmamız gerekir.

>> def topla(\*a):

>>      sonuc = 0

>>      for i in a :

>>            sonuc = sonuc + i

>>      return sonuc

>> print(topla(2,3,4,5))

>> print(topla(2))

14

2

**Global**

Fonskiyonun dışında tanımlanan değişkenler, global değişken olarak tanımlanır. Bu değişkene fonksiyon içerisinde değer ataması yaparsakta, global etiketi vermediğimiz sürece içeride tanımlı kalır.

>> c = 10

>> def carp(a,b):

>>      global c

>>      c = 5

>>      print("fonksiyonun içinde c : " ,c)

>>      print(a\*b)

>> carp(2,9)

>> print("fonksiyonun dışında c :" ,c)

fonksiyonun içinde c : 5

18

fonksiyonun dışında c : 5

**Lambda**

Lambda fonksiyonu sayesinde, yapacağımız tanımlamayı tek satırda bitirebiliriz. Kullanımı şu şekildedir.

fonksiyon = **lambda** degiskenler: donecekDeger

>> tek\_mi = lambda sayi: sayi % 2 == 1

**Modüller**

Gömülü fonksiyon, metot ve sınıflardan oluşan Python dosyasıdır. Yapacağınız programda birden fazla alan varsa bunların hepsini ayrı ayrı modüllerde kodlamak müdahaleyi daha kolay hale getirir.

**hesaplar.py** adında modüllerin olduğu bir dosya oluşturacağız. Ardından asıl dosyamızda bunu çağırarak işlemleri tamamlayacağız.

**hesaplar.py**

>> def topla (a,b):

>>    return a+b

>> def cikar (a,b):

>>    return a-b

>> def carp (a,b):

>>    return a\*b

>> def böl (a,b):

>>    return a/b

**program.py**

>> from hesaplar import carpma, topla

>> from hesaplar import \*

>> topla(3,10)

>> import hesaplar

>> hesaplar.topla(3,5)

**Class (Sınıf)**

Sınıflar sayesinde birden fazla özelliği bir arada tutabiliriz.

Basit sınıf tanımı ile başlayalım.

>> class Araba():

>>    marka = "Renault"

>>    model = "Clio"

>>    fiyat = 575.000

>>    renk = "Kirmizi"

>>    def bilgileriYazdir(self):

>>            print(self.marka,self.model,self.fiyat,self.renk,sep="\n")

>> araba = Araba() #Sınıf çağırılmalıdır.

>> araba.bilgileriYazdir()

Bu örnekte tanımlamaları kendimiz yaptık. Tanımlama işlemini de kullanıcıya bırakabiliriz. Bunun içinde sınıf içerisinde **\_\_init\_\_** fonksiyonunu çağırıp, içerisine tanımlama yapacağız.

>> class Araba():

>>    def \_\_init\_\_(self,marka,model,fiyat,renk):

>>        self.gelenMarka = marka

>>        self.gelenModel = model

>>        self.gelenFiyat = fiyat

>>        self.gelenRenk = renk

>>    def bilgileriYazdir(self):

>>            print(self.gelenMarka,self.gelenModel, self.gelenFiyat,self.gelenRenk)

>>    def renkDegistir(self,renk):

>>            self.gelenRenk = renk

>> araba1 = Araba("Ford","Fiesta",256.000,"Mavi")

>> araba1.renkDegistir("Turuncu")

>> araba1.bilgileriYazdir()

>> araba2 = Araba("Opel","Astra",656.000,"Sarı")

Ford Fiesta 256.0 Turuncu

**print()** fonksiyonunu çağırdığımız zaman çıkmasını istediğimiz değerleri, **\_\_str\_\_** fonksiyonunun altında tanımlarız.

>> class Araba():

>>    def \_\_init\_\_(self,marka,model,fiyat,renk):

>>        self.gelenMarka = marka

>>        self.gelenModel = model

>>        self.gelenFiyat = fiyat

>>        self.gelenRenk = renk

>>    def \_\_str\_\_ (self): #String objesi olduğundan return olur.

>>            return self.bilgileriYazdir()

>>    def bilgileriYazdir(self):

#İçeriğin hepsi string dönecek, sayısal değere str "string" kullan”

>>            return self.gelenMarka + " " + self.gelenModel + " "  +  str(self.gelenFiyat) + " " + self.gelenRenk

>>    def renkDegistir(self,renk):

>>            self.gelenRenk = renk

>> araba1 = Araba("Ford","Fiesta",256.000,"Mavi")

>> araba1.renkDegistir("Turuncu")

>> print(araba1) #str yi tetikledik.

Ford Fiesta 256.0 Turuncu

**Kalıtım**

Kalıtım yolu ile oluşturulan sınıf, öncekine ait bütün özellikleri kendisinde barındırır. Örneğin, bir şirket içerisinde kıdem fark etmeden çalışanlar arasında ortak özellikler mevcuttur. Bunu kod üzerinde ayrıntılı inceleyelim.

>> class Personel():

>>    def \_\_init\_\_ (self,ad,soyad,yas,cinsiyet,maas):

>>        self.ad = ad

>>        self.soyad = soyad

>>        self.yas = yas

>>        self.cinsiyet = cinsiyet

>>        self.maas = maas

>> #     def bilgileriYazdir(self): #Böyle olursa erişmek için personel.bilgileriYazdir()

>> #         print("""

>> # {} {} Bilgileri şunlardır:

>> # Yaş: {}

>> # Cinsiyet: {}

>> # Maaş: {}

>> #             """.format(self.ad,self.soyad,self.yas,self.cinsiyet,self.maas))

>>    def bilgileriYazdir(self): #Böyle yazazarsak eğer erişmek için print(personel)

>>        return self.ad+ " " +self.soyad+ " " + "Bilgileri şunlardır: \n" + "Yaş : "+str(self.yas)+ "\nCinsiyet :" +self.cinsiyet+"\nMaaş :"+str(self.maas)+"\n"

>>    def \_\_str\_\_(self):

>>        return self.bilgileriYazdir()

>> class Yonetici(Personel):

>>    def \_\_init\_\_ (self,ad,soyad,yas,cinsiyet,maas):

>>            super().\_\_init\_\_(ad,soyad,yas,cinsiyet,maas)

>>            #Bu sayede üst yapıyı direkt alabildik.

>> def maasArttir(self,pObject,arttirmaMiktari=1000):

>>         #Personelde olmayan özellik.

>>         pObject.maas += arttirmaMiktari

>>         #Sorun çıkmaması için Object değer gönderdik.

>> personel = Personel("Berkay","Yürür","30","E",20000)

>> #personel.bilgileriYazdir()

>> print(personel)

>> yonetici = Yonetici("Serra","Kurhan","35","K",50000)

>> #yonetici.bilgileriYazdir()

>> print(yonetici)

>> yonetici.maasArttir(personel)

>> #personel.bilgileriYazdir()

>> print(personel)

**Yararlı Gömülü Fonksiyon/Sınıflar**

**map()**

Parametre olarak aldığı fonksiyona, parametre olarak aldığı listenin her elemanını sırasıyla gönderir.

>> def carp(sayi1,sayi2):

>>    return sayi1 \* sayi2

>> sayilar1 = range(1,10)

>> sayilar2 = range(11,20)

>> print (\*map(carp, sayilar1, sayilar2))

11 24 39 56 75 96 119 144 171

**zip()**

İki dizinin öğelerini birbirleriyle eşleştiren zip objesi oluşturur.

>> listeSayi = [1,2,3,4]

>> listeHarf = ["a","b","c","d"]

>> print(\*zip(listeSayi, listeHarf))

(1, 'a') (2, 'b') (3, 'c') (4, 'd')

**Enumerate Sınıfı**

**Itere** edilebilir bir objenin(list, string, tuple vb) itemlarına birer **indeks** numarası verir. Birinci parametre itere edilecek objedir yani bir iterabledır. Örneğin bir list, tuple, yada string gibi. İkinci parametre ise **start** parametresidir ve indekslenmenin kaçtan başlayacağını belirler.

>> isimler = ['Oğuzhan', 'Enes', 'Selim']

>> print(list(enumerate(isimler, 1)))

[(1, 'Oğuzhan'), (2, 'Enes'), (3, 'Selim')]

**filter()**

Çağırılan fonksiyonun döndürdüğü değerin **True** olduğu durumlara göre liste döndürür. Fonksiyona gönderilen değerler içerisinden istediklerimizi alarak yeni liste oluşturmamızı elimizdeki liste üzerinde filtreleme yapmamızı sağlar.

>> def carp(sayi1,sayi2):

>>    return sayi1 \* sayi2

>> def tek\_sayi(number):

>>    return number % 2 == 1

>> liste1 = range(1, 10)

>> liste2 = range(11, 20)

>> liste3 = map(carp, liste1, liste2)

>> print (\*filter(tek\_sayi, liste3))

11 39 75 119 171

**all()**

Bir dizi içinde bulunan bütün değerler **True** ise **True** değeri, eğer bu değerlerden en az biri **False** ise de **False** değeri döndürmektir.

>> liste = [0, 1, 2, 3, 4]

>> print(all(liste))

False #Çünkü 0’ın değeri False

**any()**

Dizi içindeki bütün değerlerden en az biri **True** ise **True** çıktısı vermektir.

>> liste = [0, 1, 2, 3, 4]

>> print(any(liste))

True

**SET(Küme) Sınıfı**

Elemanları tekrar etmeyen, sıralı yapılardır. Tekrar edenler varsa onları tek eleman olarak kabul eder. Elemanlar anahtar küme içermez. Sözlükler gibi sıralı veri tipi değildir. Bu yüzden indeksleme desteklemez

>> kume={}

>> kume= {"Python",'c',4,"Cahit"}

**Add Metodu**

Kümeye eleman, öğe eklememizi sağlar.

>> kume ={1,2,3,4,5}

>> kume.add(6)

>> print(kume)

{1, 2, 3, 4, 5, 6}

**Clear**

Kümeyi temizlemeye yarar ve boş değer döndürür.

**Copy**

Kümeyi kopyalamamıza yarar. Bir kümeyi kopyalamadan da başka bir değişkene atayabiliriz ama biz bir küme üzerinde değiştirme işlemi yaptığımızda diğer küme de değişiyor bu yüzden **copy()** metodunu kullanıyoruz.

>> kume ={1,2,3,4,9,6}

>> kume\_yedek =set()

>> kume\_yedek=kume.copy()

>> print(kume\_yedek)

{1, 2, 3, 4, 6, 9}

**Update**

Kümeyi yenilememizi sağlar.

>> kume ={1,2,3,4,5,6}

>> kume\_2={7,8,9,10}

>> kume.update(kume\_2)

>> print(kume)

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

**Remove**

Kümeden eleman silmemizi sağlar. Küme içinde ilgili eleman yoksa bize hata verir.

>> kume ={1,2,3,4,5,6}

>> kume.remove(4)

>> print(kume)

{1, 2, 3, 5, 6}

**Discard**

**remove()** metodu gibi kümede istenilen elemanı silme yarar ama bu silmek istenilen eleman eğer kümenin içinde yoksa hata vermez.

>> kume ={1,2,3,4,5,6}

>> kume.discard(8)

>> print(kume)

{1, 2, 3, 4, 5, 6}

**Pop**

Kümenin ilk elemanını silmeye yarar.

>> kume ={1,2,3,4,5,6}

>> kume.pop()

>> print(kume)

{2, 3, 4, 5, 6}

**Union**

İki kümenin birleşmesini sağlayan metottur. Birleşim kümesi oluşturur.

>> A = {1,2,3,4,9}

>> B = {3,4,5,6,10}

>> C = {4,7,8,9,10}

>> print(A.union(B))

{1, 2, 3, 4, 5, 6, 9, 10}

**Intersection**

İki kümenin kesişimini sağlayan metottur. Kesişim kümesi oluşturur.

>> print(A.intersection(C))

{9, 4}

**Intersection\_update**

Bu metot da **intersection** metodu gibi iki metodun kesişimini alır ve güncelleme işlemini yapar.

>> A.intersection\_update(C)

>> print(A)

{9, 4}

**Difference**

İki küme arasındaki farkı bulmaya yarar.

>> print(A.difference(B))

{1, 2, 9}

**Differnce\_update**

Hem iki küme arasındaki farkı bulur hem de kümeyi günceller.

>> A.difference\_update(B)

>> print(A)

{1, 2, 9}

**Issubset**

Bir kümenin başka bir kümenin alt kümesi olup olmadığını sorgular.

>> A = {1,2,5,8,9,12}

>> B = {1,2,5}

>> print(B.issubset(A))

True

**Issuperset**

**Issubset** metodunu tersini yapar. Bize bir kümenin başka bir kümeyi kapsayıp kapsamadığını sorgulayan metottur.

>> print(A.issuperset(B))

True

A kümesi B kümesini kapsar dedik ve **True** değeri döndü. Çünkü B kümesi A kümesinin alt kümesidir. Tam tersini yaparsak eğer **False** değeri dönecektir.

**Frozenset(Kısıtlanmış Küme)**

Python, veri tiplerini **mutable** (değiştirilebilir) ve immutable (değiştirilmez) olarak ayırır. Bu veri tipi de **immutable** (değiştirilemez) veri tipidir. Bu yüzden kısıtlanmış küme denir ve içerisinde herhangi bir işlem yapılamaz.

>> kisitli\_kume = frozenset(["Python",3.14,5,'A'])

>> print(kisitli\_kume)

>> frozenset({5, 3.14, 'Python', 'A'})

>> kisitli\_kume.remove("Python")

AttributeError: 'frozenset' object has no attribute 'remove'

Görüldüğü gibi metot, herhangi bir değişikliğe, silmeye izin vermemektedir. O yüzden hata döndürdü.

**SqlLite Veri Tabanı**

Veri tabanı işlemlerinde, önce bağlantı kurmamız ardından da tablo oluşturarak işlemleri yapmamız gerekmektedir. İşlemler için de **imleç** oluşturmalı, işimiz bitince de veri tabanı bağlantısını kesmeliyiz.

>> import sqlite3

>> baglanti = sqlite3.connect("kisiler.db")

>> imlec = baglanti.cursor()

>> imlec.execute("CREATE TABLE IF NOT EXISTS ekip(isim TEXT, yas INT, cinsiyet TEXT)")

Daha önceden oluşturduğumuz aynı isme sahip tablo yoksa komut direkt tabloyu oluşturur. Yoksa hata verir.

Veri eklemek için “**INSERT INTO <tablo\_adi> VALUES(...,..,..)**” kullanılır. Girdiğimiz verileri işlemek için **.commit()** komutu çalıştırılır.

>> imlec.execute('INSERT INTO ekip VALUES("ilkin",22,"Erkek")')

>> baglanti.commit()

>> baglanti.close()

Veri çekme işlemi için, “**SELECT <\* veya sütün\_adi/adlari> FROM < tablo\_adi > KOŞULLAR**”

>> with sqlite3.connect("kisiler.db") as baglanti:

>>    imlec = baglanti.cursor()

>>    imlec.execute("SELECT isim,yas FROM ekip ")

>>    imlec.execute('SELECT \* FROM ekip WHERE yas == 22 AND cinsiyet == "Erkek"')

>>    for veri in imlec.fetchall():

>>        print(veri)

>>    baglanti.commit()

**with** ile bağlantı oluşturduğumuz zaman, veri tabanı işlem sonrasında otomatik kapanmaktadır.

Verinin güncellenmesi “**UPDATE < tablo\_adi > SET <..> KOSULLAR**”

>> with sqlite3.connect("kisiler.db") as baglanti:

 >>   imlec = baglanti.cursor()

 >>   imlec.exetuce ('UPDATE ekip SET yas == 25 WHERE isim == "eva" ')

 >>   baglanti.commit()

Veriyi silmek istersek **“DELETE FROM < tablo\_adi > KOSULLAR”**

>> with sqlite3.connect("kisiler.db") as baglanti:

>>    imlec = baglanti.cursor()

>>    imlec.exetuce ('DELETE FROM ekip WHERE isim == "mert" ')

>> baglanti.commit()

**İteratör (Iterator)**

Sayılabilir sayıda değer içeren bir objedir. Yani tüm değerler arasında geçiş yapabilirsiniz. Teknik olarak bir yineleyici, **\_\_iter\_\_()** ve **\_\_next\_\_()** yöntemlerinden oluşan yineleyici protokolünü uygular.

Tüm sınıfların **\_\_init\_\_()** adlı bir işlevi vardır; bu işlev, nesne oluşturulurken bazı başlatma işlemleri yapmanıza olanak tanır.

**\_\_iter\_\_()** yöntemi benzer şekilde davranır, işlemler yapabilirsiniz (başlatma vb.), ancak her zaman yineleyici nesnenin kendisini döndürmesi gerekir.

**\_\_next\_\_()** yöntemi ayrıca işlemler yapmanıza izin verir ve dizideki bir sonraki öğeyi döndürmesi gerekir.

>> class MyNumbers:

>>  def \_\_iter\_\_(self):

>>    self.a = 1

>>    return self

>>  def \_\_next\_\_(self):

>>    x = self.a

>>    self.a += 1

>>    return x

>> myclass = MyNumbers()

>> myiter = iter(myclass)

>> print(next(myiter))

>> print(next(myiter))

>> print(next(myiter))

>> print(next(myiter))

1

2

3

4

5

**Generatör (Generators)**

Generator'lar ise, farklı bir çeşit iterable'dır. Bunların diğer iterable'lardan farklarından biri, bunların tek kullanımlık olmasıdır. Örneğin, bir listeyi istediğiniz kadar for döngüsünde kullanabilirsiniz, ancak, bir generator'u yalnız bir kere for döngüsünde kullanabilirsiniz. Bunların bir diğer önemli farkı ise, tüm elemanların hafızada tutulmaması. Generatorlar, sırası gelen elemanı üretip döndürür, daha sonra da bu elemanı unuturlar.

>> generator = (x\*x\*x for x in range(5))

>> for k in generator:

>>    print(k)

0

1

8

27

64

>> for k in generator:

>>    print(k)

#Ekrana hiçbir şey basılmaz, çünkü generator'u bir kere kullandık ve bitti

**yield**

**return** deyimi gibi fonksiyonlarda kullanılır, ancak, fonksiyon bir generator döndürür.

>> def creategeneratorSquare(l):

>>    for x in l:

>>        yield x \* x

>> generator = creategeneratorSquare([1,2,3,4,5])

>> for k in generator:

>>    print (k)

1

4

9

16

25

Bu fonksiyonu çağırdığımızda, normal fonksiyonlardan beklediğimiz gibi, fonksiyonun gövdesi çalışmıyor, bunun yerine fonksiyon bir generator döndürüyor. 6. satırdaki döngü, bir sonraki elemanı istedikçe, beklemedeki kod bloğu tekrar yield görene kadar çalışıp, yield gördüğünde sıradaki elemanı döndürüyor. Böylece, bu kod bloğu tamamlanıncaya kadar, 6. satırdaki for döngüsü k'ya farklı değerler atayıp, bunları ekrana bastırıyor.