**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**

asdsad