[Gömülü Fonksiyonlar](https://python-istihza.yazbel.com/gomulu_fonksiyonlar.html)

Gömülü Fonksiyonlar

Bu bölümde, daha önce de birkaç kez bahsettiğimiz ve çokça örneğini gördüğümüz bir kavramdan söz edeceğiz. Bu kavramın adı ‘gömülü fonksiyonlar’.

Esasında biz buraya gelene kadar Python’da pek çok gömülü fonksiyon gördük. Dolayısıyla aslında görünüş olarak bunların neye benzediğini biliyoruz. Örneğin daha önceki derslerimizde gördüğümüz print() gömülü bir fonksiyondur. Aynı şekilde open(), type(), len(), pow(), bin() ve şimdiye kadar tanıştığımız öteki bütün fonksiyonlar birer gömülü fonksiyondur.

Gömülü fonksiyonlar İngilizcede *builtin functions* olarak adlandırılır. Bu fonksiyonlar gerçekten de dile gömülü vaziyettedirler. Bildiğiniz gibi, bir fonksiyonu kullanabilmemiz için o fonksiyonu tanımlamamız gerekir. İşte gömülü fonksiyonlar, bizim tanımlamamıza gerek kalmadan, Python geliştiricileri tarafından önceden tanımlanıp dile gömülmüş ve hizmetimize sunulmuş faydalı birtakım araçlardır.

İşte bu bölümde biz de bu gömülü fonksiyonları tek tek ve ayrıntılı olarak inceleyeceğiz. Dediğimiz gibi, bunlardan bir kısmını halihazırda görmüştünüz. Ama biz bütünlük açısından, önceden ele almış olduğumuz bu fonksiyonlara da kısaca değinmeden geçmeyeceğiz. Böylelikle hem yeni fonksiyonlar öğrenmiş olacağız hem de önceden öğrendiğimiz fonksiyonlarla birlikte yeni fonksiyonları da derli toplu bir şekilde görme imkanımız olacak.

Bu bölümde elbette birtakım fonksiyonları salt art arda sıralamakla yetinmeyeceğiz. Python’daki gömülü fonksiyonları incelerken bir yandan da Python programlama dilindeki çok önemli bazı kavramları ele alacağız.

İlk olarak abs() adlı bir fonksiyonla başlıyoruz gömülü fonksiyonları incelemeye…

abs()

İngilizcede ‘mutlak’ anlamına gelen *absolute* adlı bir kelime bulunur. İşte bu fonksiyonun adı da bu kelimeden gelir. Fonksiyonumuzun görevi de isminin anlamına yakındır. abs() fonksiyonunu bir sayının mutlak değerini elde etmek için kullanıyoruz.

Peki ‘mutlak değer’ ne anlama geliyor. Esasında siz bu kavrama matematik derslerinden aşinasınız. Ama bilmeyenler veya unutmuş olanlar için tekrar edelim. ‘Mutlak değer’ bir sayının 0’a olan uzaklığıdır. Örneğin 20 sayısının 0 sayısına olan uzaklığı 20’dir. Dolayısıyla 20 sayısının mutlak değeri 20’dir. Aynı şekilde -20 sayısının da 0 sayısına uzaklığı 20’dir. Yani, -20 sayısının da mutlak değeri 20’dir.

İşte abs() fonksiyonu bize bir sayının mutlak değerinin ne olduğunu söyler:

**>>>** abs(-20)

20

**>>>** abs(20)

20

**>>>** abs(20.0)

20.0

Mutlak değer kavramı yalnızca tamsayılar ve kayan noktalı sayılar için değil, aynı zamanda karmaşık sayılar için de geçerlidir. Dolayısıyla abs() fonksiyonunu kullanarak karmaşık sayıların da mutlak değerini hesaplayabiliriz:

**>>>** abs(20+3j)

20.223748416156685

Gördüğünüz gibi bu fonksiyon yalnızca tek bir parametre alıyor ve bu parametrenin mutlak değerini döndürüyor.

round()

round() fonksiyonu bir sayıyı belli ölçütlere göre yukarı veya aşağı doğru yuvarlamamızı sağlar. Basit birkaç örnek verelim:

**>>>** round(12.4)

12

**>>>** round(12.7)

13

Gördüğünüz gibi bu fonksiyon, kayan noktalı sayıları en yakın tam sayıya doğru yuvarlıyor.

Ancak burada dikkat etmemiz gereken bir nokta var.

Şu örnekleri bir inceleyelim:

**>>>** round(1.5)

2

**>>>** round(12.5)

12

Gördüğünüz gibi, fonksiyonumuz 1.5 sayısını yukarı doğru, 12.5 sayısını ise aşağı doğru yuvarladı. Bunun sebebi, kayan noktalı bir sayının üst ve alt tam sayılara olan uzaklığının birbirine eşit olduğu durumlarda Python’ın çift sayıya doğru yuvarlama yapmayı tercih etmesidir. Mesela yukarıdaki örneklerde 1.5 sayısı hem 1 sayısına, hem de 2 sayısına eşit uzaklıkta bulunuyor. İşte Python bu durumda, bir çift sayı olan 2 sayısına doğru yuvarlamayı tercih edecektir.

round() fonksiyonu toplam iki parametre alır. İlk parametre, yuvarlanacak sayının kendisidir. Yuvarlama hassasiyetini belirlemek için ise ikinci bir parametreden yararlanabiliriz.

Örneğin 22 sayısını 7’ye böldüğümüzde normalde şöyle bir çıktı elde ederiz:

**>>>** 22/7

3.142857142857143

round() fonksiyonunu tek parametre ile kullandığımızda bu fonksiyon yukarıdaki sayıyı şu şekilde yuvarlayacaktır:

**>>>** round(22/7)

3

İşte biz round() fonksiyonuna ikinci bir parametre daha vererek, yuvarlama hassasiyetini kontrol edebiliriz.

Aşağıdaki örnekleri dikkatlice inceleyin:

**>>>** round(22/7)

3

**>>>** round(22/7, 0)

3.0

**>>>** round(22/7, 1)

3.1

**>>>** round(22/7, 2)

3.14

**>>>** round(22/7, 3)

3.143

**>>>** round(22/7, 4)

3.1429

Gördüğünüz gibi, round() fonksiyonuna verdiğimiz ikinci parametre, yuvarlama işleminin ne kadar hassas olacağını belirliyor.

all()

*All* kelimesi Türkçede ‘hepsi’ anlamına gelir. Bu fonksiyonun görevi de bu anlamı çağrıştırır. all() fonksiyonunun görevi, bir dizi içinde bulunan bütün değerler True ise True değeri, eğer bu değerlerden herhangi biri False ise de False değeri döndürmektir.

Örneğin elimizde şöyle bir liste olduğunu varsayalım:

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

Şimdi all() fonksiyonunu bu liste üzerine uygulayalım:

**>>>** all(liste)

True

Bildiğiniz gibi, 0 hariç bütün sayıların bool değeri True’dur. Yukarıdaki listede False değeri verebilecek herhangi bir değer bulunmadığından, all() fonksiyonu bu liste için True değerini veriyor. Bir de şuna bakalım:

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

**>>>** all(liste)

False

Dediğimiz gibi, all() fonksiyonu ancak dizi içindeki bütün değerlerin bool değeri True ise True çıktısı verecektir.

Son bir örnek daha verelim:

**>>>** liste = ['ahmet', 'mehmet', '']

**>>>** all(liste)

False

Listede False değerine sahip bir boş karakter dizisi bulunduğu için all() fonksiyonu False çıktısı veriyor.

Bu fonksiyonu her türlü kodun bool değerlerini test etmek için kullanabilirsiniz. Mesela bu fonksiyonu kullanarak, bir nesnenin listelenen özelliklerin hepsine sahip olup olmadığını denetleyebilirsiniz:

**>>>** a = 3

**>>>** t1 = a == 3 *#sayı 3 mü?*

**>>>** t2 = a < 4 *#sayı 4'ten küçük mü?*

**>>>** t3 = a % 2 == 1 *#sayı bir tek sayı mı?*

**>>>** all([t1, t2, t3]) *#sayı bu özelliklerin hepsine sahip mi?*

True

Eğer sayımız bu özelliklerin birine bile sahip değilse, all() fonksiyonu False çıktısı verecektir.

any()

*Any* kelimesi İngilizcede ‘herhangi bir’ anlamına gelir. İşte any() fonksiyonunun görevi de, bir dizi içindeki bütün değerlerden en az biri True ise True çıktısı vermektir.

Örneğin:

**>>>** liste = ['ahmet', 'mehmet', '']

**>>>** any(liste)

True

any() fonksiyonunun True çıktısı verebilmesi için listede yalnızca bir adet True değerli öğe olması yeterlidir. Bu fonksiyonun False çıktısı verebilmesi için dizi içindeki bütün öğelerin bool değerinin False olması gerekir:

**>>>** l = ['', 0, [], (), set(), dict()]

**>>>** any(l)

False

İçi boş veri tiplerinin bool değerinin False olduğunu biliyorsunuz.

Tıpkı all() fonksiyonunda olduğu gibi, any() fonksiyonunu da, bir grup nesnenin bool değerlerini denetlemek amacıyla kullanabilirsiniz.

ascii()

Bu fonksiyon, bir nesnenin ekrana basılabilir halini verir bize. Dilerseniz bu fonksiyonun yaptığı işi tanımlamak yerine bunu bir örnek üzerinden anlatmaya çalışalım:

**>>>** a = 'istihza'

**>>>** print(ascii(a))

'istihza'

Bu fonksiyonun, print() fonksiyonundan farklı olarak, çıktıya tırnak işaretlerini de eklediğine dikkat edin.

ascii() fonksiyonunun tam olarak ne yaptığını daha iyi anlamak için herhalde şu örnek daha faydalı olacaktır.

Dikkatlice bakın:

**>>>** print('**\n**')

Bu komutu verdiğimizde, n kaçış dizisinin etkisiyle yeni satıra geçileceğini biliyorsunuz.

Bir de şuna bakın:

**>>>** print(ascii('**\n**!'))

'\n'

Gördüğünüz gibi, ascii() fonksiyonu, satır başı kaçış dizisinin görevini yapmasını sağlamak yerine bu kaçış dizisinin ekrana basılabilir halini veriyor bize.

Ayrıca bu fonksiyon, karakter dizileri içindeki Türkçe karakterlerin de UNICODE temsillerini döndürür. Örneğin:

**>>>** a = 'ışık'

**>>>** print(ascii(a))

'\u0131\u015f\u0131k'

Bunu daha net şu şekilde görebiliriz:

**>>> for** i **in** a:

**...**  print(ascii(i))

**...**

'\u0131'

'\u015f'

'\u0131'

'k'

Gördüğünüz gibi, ascii() fonksiyonu ASCII olmayan karakterlerle karşılaştığında bunların karakter temsilleri yerine UNICODE temsillerini (veya onaltılık sayma düzenindeki karşılıklarını) veriyor.

Son olarak şu örneğe bakalım:

**>>>** liste = ['elma', 'armut', 'erik']

**>>>** temsil = ascii(liste)

**>>>** print(temsil)

['elma', 'armut', 'erik']

Burada listemiz ascii() fonksiyonuna parametre olarak verildikten sonra artık liste olma özelliğini yitirip bir karakter dizisi haline gelir. Bunu denetleyelim:

**>>>** print(type(temsil))

<class 'str'>

**>>>** temsil[0]

'['

Gördüğünüz gibi, ascii() fonksiyonu listeyi alıp, bunu ekrana basılabilir bir bütün haline getiriyor. Elbette bunun için de, kendisine verilen parametreyi bir karakter dizisine dönüştürüyor.

repr()

repr() fonksiyonunun yaptığı iş, biraz önce gördüğümüz ascii() fonksiyonunun yaptığı işe çok benzer. Bu iki fonksiyon, ASCII olmayan karakterlere muameleleri açısından birbirinden ayrılır.

Hatırlarsanız ascii() fonksiyonu ASCII olmayan karakterlerle karşılaştığında bunların UNICODE (veya onaltılık) temsillerini gösteriyordu:

**>>>** ascii('şeker')

"'\\u015feker'"

repr() fonksiyonu ise ASCII olmayan karakterlerle karşılaşsa bile, bize çıktı olarak bunların da karakter karşılıklarını gösterir:

**>>>** repr('şeker')

"'şeker'"

Geri kalan özellikleri bakımından repr() ve ascii() fonksiyonları birbiriyle aynıdır.

bool()

Bu fonksiyon bir nesnenin bool değerini verir:

**>>>** bool(0)

False

**>>>** bool(1)

True

**>>>** bool([])

False

bin()

Bu fonksiyon, bir sayının ikili düzendeki karşılığını verir:

**>>>** bin(12)

'0b1100'

Bu fonksiyonun verdiği çıktının bir sayı değil, karakter dizisi olduğuna dikkat etmelisiniz.

bytes()

Bu fonksiyon bytes türünde nesneler oluşturmak için kullanılır. Bu fonksiyonu ‘bayt’ adlı veri tipini incelerken ayrıntılı olarak ele almıştık. Gelin isterseniz burada da bu fonksiyona şöyle bir değinelim.

Dediğimiz gibi, bytes() adlı fonksiyon, bytes türünde veriler oluşturmaya yarar. Bu fonksiyon işlev olarak, daha önce öğrendiğimiz list(), str(), int(), set(), dict() gibi fonksiyonlara çok benzer. Tıpkı bu fonksiyonlar gibi, bytes() fonksiyonunun görevi de farklı veri tiplerini ‘bayt’ adlı veri tipine dönüştürmektir.

Bu fonksiyon, kendisine verilen parametrelerin türüne bağlı olarak birbirinden farklı sonuçlar ortaya çıkarır. Örneğin eğer bu fonksiyona parametre olarak bir tam sayı verecek olursanız, bu fonksiyon size o tam sayı miktarınca bir bayt nesnesi verecektir. Gelin isterseniz bu durumu örnekler üzerinde göstermeye çalışalım:

**>>>** bytes(10)

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

Yukarıdaki komut bize, her bir öğesinin değeri 0 olan 10 baytlık bir veri döndürdü:

**>>>** a = bytes(10)

**>>>** a

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

**>>>** a[0]

0

**>>>** a[1]

0

**>>>** a[2]

0

Gördüğünüz gibi, bytes(10) komutuyla oluşturduğumuz a değişkeni içinde toplam 10 adet bayt var ve bu baytların her birinin değeri 0.

Yukarıda, bytes() fonksiyonuna bir tam sayı değerli parametre verdiğimizde nasıl bir sonuç alacağımızı öğrendik. Peki biz bu fonksiyona parametre olarak bir karakter dizisi verirsek ne olur?

Hemen görelim:

**>>>** bytes('istihza')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: string argument without an encoding

Bu fonksiyona karakter dizilerini doğrudan parametre olarak veremeyiz. Eğer verirsek yukarıdaki gibi bir hata alırız. Peki acaba bu hatayı almamızın nedeni ne olabilir?

Dediğimiz gibi, bytes() fonksiyonu, çeşitli veri tiplerini bayta dönüştürmeye yarar. Ancak bildiğiniz gibi, bayta dönüştürme işlemi her kod çözücü tarafından farklı biçimde yapılır. Örneğin:

**>>>** 'ışık'.encode('utf-8')

b'\xc4\xb1\xc5\x9f\xc4\xb1k'

**>>>** 'ışık'.encode('cp857')

b'\x8d\x9f\x8dk'

**>>>** 'ışık'.encode('cp1254')

b'\xfd\xfe\xfdk'

Dolayısıyla, bytes() fonksiyonunun bir karakter dizisini bayta çevirirken nasıl davranması gerektiğini anlayabilmesi için, bayta dönüştürme işlemini hangi kod çözücü ile yapmak istediğimizi açıkça belirtmemiz gerekir:

**>>>** bytes('ışık', 'utf-8')

b'\xc4\xb1\xc5\x9f\xc4\xb1k'

**>>>** bytes('ışık', 'cp1254')

b'\xfd\xfe\xfdk'

**>>>** bytes('ışık', 'cp857')

b'\x8d\x9f\x8dk'

Gördüğünüz gibi, bytes() fonksiyonuna parametre olarak bir karakter dizisi verebilmek için, bu karakter dizisi ile birlikte bir kod çözücü de belirtmemiz gerekiyor. Böylece bytes() fonksiyonu kendisine verdiğimiz karakter dizisini, belirttiğimiz kod çözücünün kurallarına göre bayta dönüştürüyor.

Bu arada, çıktıda görünen ‘b’ harflerinin, elimizdeki verinin bir bayt olduğunu gösteren bir işaret olduğunu biliyorsunuz.

Ayrıca, bytes() fonksiyonuna verdiğimiz ikinci parametrenin isminin encoding olduğunu ve bu parametreyi isimli bir parametre olarak da kullanabileceğimizi belirtelim:

**>>>** bytes('istihza', encoding='ascii')

Bu noktada size şöyle bir soru sorayım: Acaba bytes() fonksiyonuna ilk parametre olarak verdiğimiz karakter dizisi, ikinci parametrede belirttiğimiz kod çözücü tarafından tanınmazsa ne olur?

Cevabı tahmin edebilirsiniz: Böyle bir durumda elbette Python bize bir hata mesajı gösterir:

**>>>** bytes('şeker', 'ascii')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

UnicodeEncodeError: 'ascii' codec can't encode character '\u015f' in position 0:

ordinal not in range(128)

… veya:

**>>>** bytes('€', 'cp857')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "C:\Python33\lib\encodings\cp857.py", line 12, in encode

**return** codecs.charmap\_encode(input,errors,encoding\_map)

UnicodeEncodeError: 'charmap' codec can't encode character '\u20ac' in position

0: character maps to <undefined>

‘ş’ harfi ‘ASCII’ karakter kümesinde; ‘€’ işareti ise ‘CP857’ adlı karakter kümesinde tanımlanmamış birer karakter olduğu için, ilgili kod çözücüler bu karakterleri çözüp bayta dönüştüremiyor. Yazdığımız kodların bu tür durumlarda tamamen çökmesini engellemek için, önceki derslerimizde de çeşitli vesilelerle öğrenmiş olduğumuz errors adlı bir parametreden yararlanabiliriz:

**>>>** bytes('ışık', encoding='ascii', errors='replace')

b'???k'

**>>>** bytes('şeker', encoding='ascii', errors='replace')

b'?eker'

**>>>** bytes('€', encoding='cp857', errors='replace')

b'?'

**>>>** bytes('€', encoding='cp857', errors='ignore')

b''

**>>>** bytes('€', encoding='cp857', errors='xmlcharrefreplace')

b'&#8364;'

**>>>** bytes('şeker', encoding='cp857', errors='xmlcharrefreplace')

b'\x9feker'

Gördüğünüz gibi, errors parametresine verdiğimiz çeşitli değerler yardımıyla, bytes() fonksiyonunun, encoding parametresinde belirtilen kod çözücü ile çözülemeyen karakterlerle karşılaştığında nasıl davranacağını belirleyebiliyoruz.

errors parametresine verdiğimiz bütün bu değerleri önceki derslerimizde öğrenmiştik. Dolayısıyla yukarıda gösterdiğimiz kodları rahatlıkla anlayabilecek kadar Python bilgisine sahibiz.

Son olarak, bytes() fonksiyonuna parametre olarak 0-256 arası sayılardan oluşan diziler de verebiliriz:

**>>>** bytes([65, 10, 12, 11, 15, 66])

b'A\n\x0c\x0b\x0fB'

Bu yapı içinde Python, 0 ile 128 arası sayılar için standart ASCII tablosunu, 128 ile 256 arası sayılar için ise Latin-1 karakter kümesini temel alarak sayıları birer bayta dönüştürecektir.

bytearray()

Bildiğiniz gibi baytlar değiştirilemeyen bir veri tipidir. Dolayısıyla bir bayt veri tipi üzerinde herhangi bir değişiklik yapamayız. Örneğin bir baytın herhangi bir öğesini başka bir değerle değiştiremeyiz:

**>>>** a = bytes('istihza', 'ascii')

**>>>** a[0]

105

**>>>** a[0] = 106

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'bytes' object does not support item assignment

Ama eğer hem baytlarla çalışmak, hem de bu baytların üzerinde değişiklik yapabilmek isterseniz baytlar yerine bayt dizileri ile çalışabilirsiniz. İşte bunun için bytearray() adlı bir fonksiyondan yararlanıyoruz.

Yaptıkları iş bakımından bytearray() ve bytes() fonksiyonları birbirlerine çok benzer. Bu ikisi arasındaki tek fark, bytearray() ile oluşturulan veri tipinin, bytes() ile oluşturulan veri tipinin aksine, değiştirilebilir nitelikte olmasıdır:

**>>>** a = bytearray('adana', 'ascii')

**>>>** a

bytearray(b'adana')

**>>>** a[0] = 65

**>>>** a

bytearray(b'Adana')

chr()

Bu fonksiyon, kendisine parametre olarak verilen bir tam sayının karakter karşılığını döndürür. Örneğin:

**>>>** chr(10)

'\n'

Bildiğiniz gibi 10 sayısının karakter karşılığı satır başı karakteridir. Bir de şuna bakalım:

**>>>** chr(65)

'A'

65 sayısının karakter karşılığı ise ‘A’ harfidir.

Bu fonksiyon sayıları karakterlere dönüştürürken ASCII sistemini değil, UNICODE sistemini temel alır. Dolayısıyla bu fonksiyon ile 128 (veya 255) üstü sayıları da dönüştürebiliriz. Örneğin:

**>>>** chr(305)

'ı'

list()

Bu fonksiyon iki farklı amaç için kullanılabilir:

1. Liste tipinde bir veri oluşturmak
2. Farklı veri tiplerini liste adlı veri tipine dönüştürmek

Birinci amaç için bu fonksiyonu şu şekilde kullanıyoruz:

**>>>** l = list()

Böylece liste tipinde bir veri oluşturmuş olduk.

Dediğimiz gibi list() fonksiyonunu, farklı tipteki verileri listeye dönüştürmek için de kullanabiliriz. Örneğin:

**>>>** list('istihza')

['i', 's', 't', 'i', 'h', 'z', 'a']

Burada ‘istihza’ adlı karakter dizisini bir listeye dönüştürdük.

Elbette bu fonksiyonu kullanarak başka veri tiplerini de listeye dönüştürebiliriz. Örneğin bir sözlüğü, bu fonksiyon yardımıyla kolayca listeye dönüştürebiliriz:

**>>>** s = {'elma': 44, 'armut': 10, 'erik': 100}

**>>>** list(s)

['armut', 'erik', 'elma']

Bir sözlük listeye dönüştürülürken, elbette sözlüğün anahtarları dikkate alınacaktır. Eğer siz sözlüğün anahtarlarından değil de değerlerinde bir liste oluşturmak isterseniz şöyle bir kod yazabilirsiniz:

**>>>** list(s.values())

[10, 100, 44]

set()

set() fonksiyonu list() fonksiyonuna çok benzer. Bu fonksiyon da tıpkı list() fonksiyonu gibi, veri tipleri arasında dönüştürme işlemleri gerçekleştirmek için kullanılabilir. set() fonksiyonunun görevi farklı veri tiplerini kümeye dönüştürmektir:

**>>>** k = set()

Burada boş bir küme oluşturduk. Şimdi de mesela bir karakter dizisini kümeye dönüştürelim:

**>>>** i = 'istihza'

**>>>** set(i)

{'t', 's', 'z', 'a', 'i', 'h'}

tuple()

tuple() fonksiyonu da, tıpkı list(), set() ve benzerleri gibi bir dönüştürücü fonksiyondur. Bu fonksiyon farklı veri tiplerini demete dönüştürür:

**>>>** tuple('a')

('a',)

frozenset()

Bu fonksiyonu kullanarak farklı veri tiplerini dondurulmuş kümeye dönüştürebilirsiniz:

**>>>** s = set('istihza')

**>>>** df = frozenset(s)

**>>>** df

frozenset({'t', 's', 'a', 'z', 'i', 'h'})

complex()

Sayılardan söz ederken, eğer matematikle çok fazla içli dışlı değilseniz pek karşılaşmayacağınız, ‘karmaşık sayı’ adlı bir sayı türünden de bahsetmiştik. Karmaşık sayılar, bir gerçek, bir de sanal kısımdan oluşan sayılardır.

Karmaşık sayılar Python’da ‘complex’ ifadesiyle gösteriliyor. Mesela şu bir karmaşık sayıdır:

**>>>** 12+0j

İşte eğer herhangi bir sayıyı karmaşık sayıya dönüştürmeniz gerekirse complex() adlı bir fonksiyondan yararlanabilirsiniz. Örneğin:

**>>>** complex(15)

(15+0j)

Böyle bir kod yazdığımızda, verdiğimiz parametre karmaşık sayının gerçek kısmını oluşturacak, sanal kısım ise 0 olarak kabul edilecektir. Elbette isterseniz sanal kısmı kendiniz de belirleyebilirsiniz:

**>>>** complex(15, 2)

(15+2j)

float()

Bu fonksiyonu, sayıları veya karakter dizilerini kayan noktalı sayıya dönüştürmek için kullanıyoruz:

**>>>** float('134')

134.0

**>>>** float(12)

12.0

int()

Bu fonksiyon birkaç farklı amaç için kullanılabilir. int() fonksiyonunun en temel görevi, bir karakter dizisi veya kayan noktalı sayıyı (eğer mümkünse) tam sayıya dönüştürmektir:

**>>>** int('10')

10

**>>>** int(12.4)

12

Bunun dışında bu fonksiyonu, herhangi bir sayma sisteminde temsil edilen bir sayıyı onlu sayma sistemine dönüştürmek için de kullanabiliriz. Örneğin:

**>>>** int('12', 8)

10

Burada, sekizli sayma sistemine ait sayı değerli bir karakter dizisi olan ‘12’`yi onlu sayma sistemine dönüştürdük ve böylece `10 sayısını elde ettik.

int() fonksiyonunu sayma sistemleri arasında dönüştürme işlemlerinde kullanabilmek için ilk parametrenin bir karakter dizisi olması gerektiğine dikkat ediyoruz.

Bu arada, int('12', 8) komutununun 12 sayısını sekizli sayma sistemine dönüştürmediğine dikkat edin. Bu komutun yaptığı iş sekizli sayma sistemindeki 12 sayısını onlu sayma sistemine dönüştürmektir.

int() fonksiyonunun bu kullanımıyla ilgili bir örnek daha verelim:

**>>>** int('4cf', 16)

1231

Burada da, onaltılı sayma sistemine ait bir sayı olan 4cf’yi onlu sayma sistemine çevirdik ve 1231 sayısını elde ettik. 4cf sayısını int() fonksiyonuna parametre olarak verirken bunu karakter dizisi şeklinde yazmayı unutmuyoruz. Aksi halde Python bize bir hata mesajı gösterecektir.

str()

Bu fonksiyonun, farklı veri tiplerini karakter dizisine dönüştürmek için kullanıldığını biliyorsunuz. Örneğin:

**>>>** str(12)

'12'

Burada 12 sayısını bir karakter dizisine dönüştürdük. Şimdi de bir baytı karakter dizisine dönüştürelim:

**>>>** bayt = b'istihza'

Bayt nesnemizi tanımladık. Şimdi bunu bir karakter dizisine dönüştürelim:

**>>>** kardiz = str(bayt, encoding='utf-8')

**>>>** print(kardiz)

istihza

Gördüğünüz gibi, bir baytı karakter dizisine dönüştürmek için str() fonksiyonuna encoding adlı bir parametre veriyoruz. Fonksiyonumuz, bu parametrede hangi kodlama biçimi belirtildiyse, baytları bu kodlama biçiminin kurallarına göre bir karakter dizisine dönüştürüyor.

Tahmin edebileceğiniz gibi, belirttiğiniz kodlama biçiminin herhangi bir baytı karakter dizisine dönüştüremediği durumlara karşı bir errors parametresi de verebiliriz str() fonksiyonuna. Örneğin elimizde bayt tipinde şöyle bir veri olduğunu varsayalım:

**>>>** bayt = bytes('kadın', encoding='utf-8')

**>>>** print(bayt)

b'kad\xc4\xb1n'

Şimdi bu bayt veri tipini bir karakter dizisine dönüştürmeye çalışalım:

**>>>** kardiz = str(bayt, encoding='ascii')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 3: ordinal

not in range(128)

ASCII adlı kod çözücü, b'kadın' içindeki baytlardan birini tanıyamadığı için bize bir hata mesajı gösterdi. Bildiğiniz gibi ASCII 128’den büyük baytları dönüştüremez. İşte bu tür durumlara karşı errors parametresinden yararlanabilirsiniz:

**>>>** kardiz = str(bayt, encoding='ascii', errors='ignore')

**>>>** print(kardiz)

kadn

errors parametresine verdiğimiz ‘ignore’ değeri sayesinde Python bize hata mesajı göstermek yerine, ASCII ile çözülemeyen baytı görmezden geldi. errors parametresinin hangi değerleri alabileceğini önceki derslerimizden hatırlıyor olmalısınız.

dict()

Bu fonksiyon, farklı veri tiplerinden sözlükler üretmemizi sağlar. Örneğin bu fonksiyonu kullanarak boş bir sözlük oluşturabiliriz:

**>>>** s = dict()

Bu fonksiyon, değişkenlerden sözlükler oluşturmamızı da sağlar:

**>>>** s = dict(a=1, b=2, c=3)

{'a': 1, 'b': 2, 'c': 3}

dict() fonksiyonuna parametre olarak iç içe geçmiş listeler veya demetler vererek de sözlük üretebiliriz:

**>>>** öğeler = (['a', 1], ['b', 2], ['c', 3])

**>>>** dict(öğeler)

{'a': 1, 'b': 2, 'c': 3}

callable()

Bu fonksiyon, bir nesnenin ‘çağrılabilir’ olup olmadığını denetler. Peki hangi nesneler çağrılabilir özelliktedir. Mesela fonksiyonlar çağrılabilir nesnelerdir. Değişkenler ise çağrılabilir nesneler değildir.

Birkaç örnek verelim bununla ilgili:

**>>>** callable(open)

True

Python’ın open() adlı bir fonksiyonu olduğu için, doğal olarak callable() fonksiyonu True çıktısı veriyor.

Bir de şuna bakalım:

**>>> import** **sys**

**>>>** callable(sys.version)

False

Burada da sys modülü içindeki version adlı nesnenin çağrılabilir özellikte olup olmadığını sorguladık. Daha önceki derslerimizde de gördüğünüz gibi, sys modülü içindeki version adlı araç bir fonksiyon değil, değişkendir. Dolayısıyla bu değişken callable(sys.version) sorgusuna False yanıtı verir.

ord()

Bu fonksiyon, bir karakterin karşılık geldiği ondalık sayıyı verir. Örneğin:

**>>>** ord('a')

97

**>>>** ord('ı')

305

oct()

Bu fonksiyon, bir sayıyı sekizli düzendeki karşılığına çevirmemizi sağlar:

**>>>** oct(10)

'0o12'

hex()

Bu fonksiyon, bir sayıyı onaltılı düzendeki karşılığına çevirmemizi sağlar:

**>>>** hex(305)

'Ox131'

Yalnız hem oct() hem de hex() fonksiyonlarında dikkat etmemiz gereken şey, bu fonksiyonların parametre olarak bir sayı alıp, çıktı olarak bir karakter dizisi veriyor olmasıdır.

eval(), exec(), globals(), locals(), compile()

Bu bölümde beş farklı fonksiyonu bir arada inceleyeceğiz. Bu fonksiyonları birlikte ele almamızın nedeni bunların birbiriyle yakından bağlantılı olması.

Burada işleyeceğimiz bu beş fonksiyon şunlardan oluşuyor:

1. eval()
2. exec()
3. globals()
4. locals()
5. compile()

Ancak bu fonksiyonlardan söz etmeye başlamadan önce Python’daki iki önemli kavramı açıklığa kavuşturmamız gerekiyor: Bu kavramlar şunlar:

1. ifade
2. deyim

Öncelikle ‘ifade’ kavramından başlayalım.

İngilizcede *expression* denen ‘ifadeler’, bir değer üretmek için kullanılan kod parçalarıdır. Karakter dizileri, sayılar, işleçler, öteki veri tipleri, liste üreteçleri, sözlük üreteçleri, küme üreteçleri, fonksiyonlar hep birer ifadedir. Örneğin:

**>>>** 5

**>>>** 23 + 4

**>>>** [i **for** i **in** range(10)]

**>>>** len([1, 2, 3])

İngilizcede *statement* olarak adlandırılan ‘deyimler’ ise ifadeleri de kapsayan daha geniş bir kavramdır. Buna göre bütün ifadeler aynı zamanda birer deyimdir. Daha doğrusu, ifadelerin bir araya gelmesi ile deyimler oluşturulabilir.

Deyimlere birkaç örnek verelim:

**>>>** a = 5

**>>> if** a:

**...**  print(a)

**>>> for** i **in** range(10):

**...**  print(i)

Python programlama dilinde deyimlerle ifadeleri ayırt etmenin kolay bir yolu da eval() fonksiyonundan yararlanmaktır. Eğer deyim mi yoksa ifade mi olduğundan emin olamadığınız bir şeyi eval() fonksiyonuna parametre olarak verdiğinizde hata almıyorsanız o parametre bir ifadedir. Eğer hata alıyorsanız o parametre bir deyimdir. Çünkü eval() fonksiyonuna parametre olarak yalnızca ifadeler verilebilir.

Birkaç örnek verelim:

**>>>** eval('a = 5')

File "<string>", line 1

a = 5

^

SyntaxError: invalid syntax

Gördüğünüz gibi, eval() fonksiyonu bize bir hata mesajı verdi. Çünkü a = 5 kodu bir deyimdir. Unutmayın, Python’da bütün değer atama işlemleri birer deyimdir. Dolayısıyla eval() fonksiyonu bu deyimi parametre olarak alamaz.

Bir de şuna bakalım:

**>>>** eval('5 + 25')

30

Bu defa hata almadık. Çünkü eval() fonksiyonuna, olması gerektiği gibi, parametre olarak bir ifade verdik. Bildiğiniz gibi, 5 + 25 kodu bir ifadedir.

Dediğimiz gibi, eval() fonksiyonu deyimleri parametre olarak alamaz. Ama exec() fonksiyonu alabilir:

**>>>** exec('a = 5')

Bu şekilde, değeri 5 olan a adlı bir değişken oluşturmuş olduk. İsterseniz kontrol edelim:

**>>>** print(a)

5

Gördüğünüz gibi, exec() fonksiyonu, mevcut isim alanı içinde a adlı bir değişken oluşturdu. Yalnız elbette mevcut isim alanı içinde yeni değişkenler ve yeni değerler oluştururken dikkatli olmamız gerektiğini biliyorsunuz. Zira mesela yukarıdaki komutu vermeden önce mevcut isim alanında zaten a adlı bir değişken varsa, o değişkenin değeri değişecektir:

**>>>** a = 20

Elimizde, değeri 20 olan a adlı bir değişken var. Şimdi exec() fonksiyonu yardımıyla a değişkeninin de içinde yer aldığı mevcut isim alanına müdahale ediyoruz:

**>>>** exec('a = 10')

Böylece a değişkeninin eski değerini silmiş olduk. Kontrol edelim:

**>>>** print(a)

10

Bu tür durumlarda, exec() ile oluşturduğunuz değişkenleri global isim alanına değil de, farklı bir isim alanına göndermeyi tercih edebilirsiniz. Peki ama bunu nasıl yapacağız?

Python programlama dilinde isim alanları sözlük tipinde bir veridir. Örneğin global isim alanı basit bir sözlükten ibarettir.

Global isim alanını gösteren sözlükte hangi anahtar ve değerlerin olduğunu görmek için globals() adlı bir fonksiyonu kullanabilirsiniz:

**>>>** globals()

Bu fonksiyonu çalıştırdığımızda şuna benzer bir çıktı alırız:

{'\_\_doc\_\_': None, '\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>,

'\_\_name\_\_': '\_\_main\_\_', '\_\_package\_\_': None, '\_\_builtins\_\_': <module 'builtins'>}

Gördüğünüz gibi, elimizdeki şey gerçekten de bir sözlük. Dolayısıyla bir sözlük ile ne yapabilirsek bu sözlükle de aynı şeyi yapabiliriz…

‘globals’ adlı bu sözlüğün içeriği, o anda global isim alanında bulunan nesnelere göre farklılık gösterecektir. Örneğin:

**>>>** x = 10

şeklinde 10 değerine sahip bir x nesnesi tanımladıktan sonra globals() fonksiyonunu tekrar çalıştırırsanız global isim alanına bu nesnenin de eklenmiş olduğunu görürsünüz.

Dediğimiz gibi, globals() fonksiyonundan dönen nesne bir sözlüktür. Bu sözlüğe, herhangi bir sözlüğe veri ekler gibi değer de ekleyebilirsiniz:

**>>>** globals()['z'] = 23

Bu şekilde global isim alanına z adlı bir değişken eklemiş oldunuz:

**>>>** z

23

Yalnız, Python programlama dili bize bu şekilde global isim alanına nesne ekleme imkanı verse de, biz mecbur değilsek bu yöntemi kullanmaktan kaçınmalıyız. Çünkü bu şekilde sıradışı bir yöntemle değişken tanımladığımız için aslında global isim alanını, nerden geldiğini kestirmenin güç olduğu değerlerle ‘kirletmiş’ oluyoruz.

Bildiğiniz gibi, Python’da global isim alanı dışında bir de lokal isim alanı bulunur. Lokal isim alanlarının, fonksiyonlara (ve ileride göreceğimiz gibi sınıflara) ait bir isim alanı olduğunu biliyorsunuz. İşte bu isim alanlarına ulaşmak için de locals() adlı bir fonksiyondan yararlanacağız:

**def** fonksiyon(param1, param2):

x = 10

print(locals())

fonksiyon(10, 20)

Bu fonksiyonu çalıştırdığınızda şu çıktıyı alacaksınız:

{'param2': 20, 'param1': 10, 'x': 10}

Gördüğünüz gibi, locals() fonksiyonu gerçekten de bize fonksiyon() adlı fonksiyon içindeki lokal değerleri veriyor.

globals() ve locals() fonksiyonlarının ne işe yaradığını incelediğimize göre exec() fonksiyonunu anlatırken kaldığımız yere dönebiliriz.

Ne diyorduk?

Elimizde, değeri 20 olan a adlı bir değişken vardı:

**>>>** a = 20

exec() fonksiyonu yardımıyla a değişkeninin de içinde yer aldığı mevcut isim alanına müdahale edelim:

**>>>** exec('a = 3')

Bu şekilde a değişkeninin varolan değerini silmiş olduk:

**>>>** print(a)

3

Dediğimiz gibi, bu tür durumlarda, exec() ile oluşturduğunuz değişkenleri global isim alanı yerine farklı bir isim alanına göndermeyi tercih etmemiz daha uygun olacaktır. Python’da isim alanlarının basit bir sözlük olduğunu öğrendiğimize göre, exec() ile oluşturduğumuz değişkenleri global isim alanı yerine nasıl farklı bir isim alanına göndereceğimizi görebiliriz.

Önce yeni bir isim alanı oluşturalım:

**>>>** ia = {}

Şimdi exec() ile oluşturacağımız değerleri bu isim alanına gönderebiliriz:

**>>>** exec('a = 3', ia)

Böylece global isim alanındaki a değişkeninin değerine dokunmamış olduk:

**>>>** a

20

Yeni oluşturduğumuz değer ise ia adlı yeni isim alanına gitti:

**>>>** ia['a']

3

copyright()

Bu fonksiyon yardımıyla Python’ın telif haklarına ilişkin bilgilere erişebilirsiniz:

**>>>** copyright()

Copyright (c) 2001-2012 Python Software Foundation.

All Rights Reserved.

Copyright (c) 2000 BeOpen.com.

All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.

All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.

All Rights Reserved.

credits()

Bu fonksiyon, Python programlama diline katkıda bulunanlara teşekkür içeren küçük bir metni ekrana çıktı olarak verir:

**>>>** credits()

Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a cast of thousands

for supporting Python development. See www.python.org for more information.

license()

Bu fonksiyon yardımıyla Python’ın lisansına ilişkin epey ayrıntılı metinlere ulaşabilirsiniz.

dir()

Eğer dir() fonksiyonunu parametresiz olarak kullanırsak, mevcut isim alanındaki öğeleri bir liste halinde elde ederiz:

**>>>** dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_']

Bu bakımdan dir() fonksiyonu globals() ve locals() fonksiyonlarına benzer. Ancak onlardan farkı, dir() fonksiyonunun çıktı olarak bir liste, globals() ve locals() fonksiyonlarının ise birer sözlük vermesidir.

Ayrıca dir() fonksiyonunu kullanarak nesnelerin metot ve niteliklerini içeren bir listeye ulaşabileceğimizi de biliyorsunuz. Örneğin bu fonksiyonu kullanarak farklı veri tiplerinin metot ve niteliklerini listeleyebiliriz:

**>>>** dir('')

**>>>** dir([])

**>>>** dir({})

divmod()

Bu fonksiyonun işlevini bir örnek üzerinden göstermeye çalışalım:

**>>>** divmod(10, 2)

(5, 0)

Gördüğünüz gibi divmod(10, 2) komutu bize iki öğeli bir demet veriyor. Bu demetin ilk öğesi, divmod() fonksiyonuna verilen ilk parametrenin ikinci parametreye bölünmesi işleminin sonucudur. Demetimizin ikinci öğesi ise, ilk parametrenin ikinci parametreye bölünmesi işleminden kalan sayıdır. Yani demetin ilk parametresi bölme işleminin ‘bölüm’ kısmını, ikinci öğesi ise ‘kalan’ kısmını verir.

Bu fonksiyonun bölme işlemininin sonucunu tamsayı cinsinden verdiğine dikkat ediyoruz:

**>>>** divmod(10, 3)

(3, 1)

10 sayısı 3 sayısına bölündüğünde tamsayı cinsinden sonuç 3’tür. Bu bölme işleminin kalanı ise 1’dir.

enumerate()

İngilizcede *enumerate* kelimesi ‘numaralandırmak’ anlamına gelir. enumerate() fonksiyonunun görevi de kelimenin bu anlamıyla aynıdır. Yani bu fonksiyonu kullanarak nesneleri numaralandırabiliriz.

Bu fonksiyon bize bir ‘enumerate’ nesnesi verir:

**>>>** enumerate('istihza')

<class 'enumerate'>

Bu nesnenin içeriğine nasıl erişebileceğimizi biliyorsunuz:

Nesneyi bir listeye çevirebiliriz:

**>>>** list(enumerate('istihza'))

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]

veya:

**>>>** [i **for** i **in** enumerate('istihza')]

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]

print() fonksiyonuna yıldızlı parametre olarak verebiliriz:

**>>>** print(\*enumerate('istihza'))

(0, 'i') (1, 's') (2, 't') (3, 'i') (4, 'h') (5, 'z') (6, 'a')

veya nesne üzerinde bir döngü kurabiliriz:

**>>> for** i **in** enumerate('istihza'):

**...**  print(i)

**...**

(0, 'i')

(1, 's')

(2, 't')

(3, 'i')

(4, 'h')

(5, 'z')

(6, 'a')

Gördüğünüz gibi, ‘enumerate’ nesnesi bize her koşulda iki öğeli demetler veriyor. Bu demetlerin her bir öğesine nasıl ulaşabileceğimizi de biliyor olmalısınız:

**>>> for** sıra, öğe **in** enumerate('istihza'):

**...**  print("**{}**. **{:>2}**".format(sıra, öğe))

**...**

0. i

1. s

2. t

3. i

4. h

5. z

6. a

Örneklerden de gördüğünüz gibi, enumerate() fonksiyonu bize bir dizi içindeki öğelerin sırasını ve öğenin kendisini içeren bir demet veriyor. Dikkat ettiyseniz, her zaman olduğu gibi, Python burada da saymaya 0’dan başlıyor. Yani enumerate() fonksiyonunun ürettiği öğe sıralamasında ilk öğenin sırası 0 oluyor. Elbette eğer isterseniz enumerate() fonksiyonunun saymaya kaçtan başlayacağını kendiniz de belirleyebilirsiniz:

**>>> for** sıra, öğe **in** enumerate('istihza', 1):

**...**  print("**{}**. **{:>2}**".format(sıra, öğe))

**...**

1. i

2. s

3. t

4. i

5. h

6. z

7. a

enumerate() fonksiyonuna verdiğimiz ikinci parametre saymaya kaçtan başlanacağını gösteriyor. Eğer bu fonksiyonu ikinci parametre olmadan kullanırsak Python bizim saymaya 0’dan başlamak istediğimizi varsayacaktır.

exit()

Bu fonksiyon, o anda çalışan programdan çıkmanızı sağlar. Eğer bu komutu etkileşimli kabukta verirseniz o anda açık olan oturum kapanacaktır.

help()

help() fonksiyonu gömülü fonksiyonlar içinde en faydalı fonksiyonların başında gelir. Bu fonksiyonu kullanarak Python programlama diline ait öğelere ilişkin yardım belgelerine ulaşabiliriz. Örneğin:

**>>>** help(dir)

Bu komutu verdiğimizde dir() fonksiyonunun ne işe yaradığını gösteren İngilizce bir belgeye ulaşırız. Gördüğünüz gibi, hakkında bilgi edinmek istediğimiz öğeyi help() fonksiyonuna parametre olarak vererek ilgili yardım dosyasına erişebiliyoruz.

Eğer bu fonksiyonu parametresiz olarak kullanırsak ‘etkileşimli yardım’ denen ekrana ulaşırız:

**>>>** help()

Welcome to Python 3.3! This is the interactive help utility.

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at http://docs.python.org/3.3/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing

Python programs and using Python modules. To quit this help utility and

return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",

"keywords", or "topics". Each module also comes with a one-line summary

of what it does; to list the modules whose summaries contain a given word

such as "spam", type "modules spam".

help>

Bu ekranda, hakkında bilgi edinmek istediğiniz öğeyi help> ibaresinden hemen sonra, boşluk bırakmadan yazarak öğeye ilişkin bilgilere ulaşabilirsiniz:

help> dir

Etkileşimli yardım ekranından çıkmak için ‘q’ harfine basabilirsiniz.

id()

Python’da her nesnenin bir kimliğinin olduğunu biliyorsunuz. Kimlik işleçlerini incelediğimiz konuda bundan bir miktar bahsetmiş ve orada id() adlı bir fonksiyondan söz etmiştik.

Orada şöyle bir örnek vermiştik:

**>>>** a = 50

**>>>** id(a)

505494576

**>>>** kardiz = "Elveda Zalim Dünya!"

**>>>** id(kardiz)

14461728

Orada söylediğimiz ve yukarıdaki örneklerden de bir kez daha gördüğünüz gibi, Python’daki her nesnenin kimliği eşşiz, tek ve benzersizdir.

input()

Bu fonksiyonun ne işe yaradığını gayet iyi biliyorsunuz. input() adlı bu gömülü fonksiyonu kullanarak kullanıcı ile veri alışverişinde bulunabiliyoruz.

format()

Bu gömülü fonksiyonun görevi, daha önce karakter dizilerini işlerken, karakter dizilerinin bir metodu olarak öğrendiğimiz format() metoduna benzer bir şekilde, karakter dizilerini biçimlendirmektir. Ancak format() fonksiyonu, daha önce öğrendiğimiz format() metoduna göre daha dar kapsamlıdır. format() metodunu kullanarak oldukça karmaşık karakter dizisi biçimlendirme işlemlerini gerçekleştirebiliriz, ama birazdan inceleyeceğimiz format() gömülü fonksiyonu yalnızca tek bir değeri biçimlendirmek için kullanılır.

Basit bir örnek verelim:

**>>>** format(12, '.2f')

‘12.00’

Yukarıdaki ifadeyi daha önce gördüğümüz format() metodu ile şu şekilde yazabiliriz:

**>>>** '**{:.2f}**'.format(12)

'12.00'

filter()

Bu gömülü fonksiyon yardımıyla dizi niteliği taşıyan nesneler içindeki öğeler üzerinde belirli bir ölçüte göre bir süzme işlemi uygulayabiliriz. Dilerseniz filter() fonksiyonunun görevini bir örnek üzerinden anlamaya çalışalım.

Diyelim ki elimizde şöyle bir liste var:

**>>>** l = [400, 176, 64, 175, 355, 13, 207, 298, 397, 386, 31, 120, 120, 236, 241, 123, 249, 364, 292, 153]

Amacımız bu liste içindeki tek sayıları süzmek.

Daha önce öğrendiğimiz yöntemleri kullanarak bu görevi şu şekilde yerine getirebiliriz:

**>>> for** i **in** l:

**...**  **if** i % 2 == 1:

**...**  print(i)

**...**

175

355

13

207

397

31

241

123

249

153

Hatta eğer istersek liste üreteçlerini kullanarak aynı işlemi daha kısa bir yoldan da halledebiliriz:

**>>>** [i **for** i **in** l **if** i % 2 == 1]

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]

İşte Python, yukarıdaki işlemi yapabilmemiz için bize üçüncü bir yol daha sunar. Bu üçüncü yolun adı filter() fonksiyonudur. Dikkatlice bakın:

**def** tek(sayı):

**return** sayı % 2 == 1

print(\*filter(tek, l))

Dilerseniz bu kodları daha iyi anlayabilmek için filter() fonksiyonuna biraz daha yakından bakalım…

filter() fonksiyonu toplam iki parametre alır. Bu parametrelerden ilki ölçütü belirleyen fonksiyon, ikincisi ise bu ölçütün uygulanacağı öğedir. Yukarıdaki örneğe baktığımızda, tek() adlı fonksiyonun, l adlı öğe üzerine uygulandığını görüyoruz.

Yukarıdaki örnekte ilk olarak tek() adlı bir fonksiyon tanımladık:

**def** tek(sayı):

**return** sayı % 2 == 1

Bu fonksiyonun görevi, kendisine parametre olarak verilen bir sayının tek sayı olup olmadığını sorgulamak. Eğer verilen parametre bir tek sayı ise fonksiyonumuz True değerini, tek sayı değilse False değerini döndürecektir. İsterseniz fonksiyonumuzu test edelim:

print(tek(12))

12 sayısı bir tek sayı olmadığı için fonksiyonumuz bize False çıktısı verir.

Bir de şuna bakalım:

print(tek(117))

117 sayısı ise bir tek sayıdır. Bu nedenle fonksiyonumuz bize True değerini verecektir.

İşte biz bu fonksiyonu, filter() fonksiyonu yardımıyla şu liste üzerine uygulayacağız:

l = [400, 176, 64, 175, 355, 13, 207, 298, 397, 386, 31,

120, 120, 236, 241, 123, 249, 364, 292, 153]

Dediğimiz gibi, filter() fonksiyonu, dizi özelliği taşıyan nesneler üzerinde belli bir ölçüte göre filtreleme işlemi yapmamızı sağlar. Biz de biraz önce tanımladığımız tek() adlı fonksiyonu l adlı bu listeye uygulayarak liste içindeki tek sayıları filtreleyeceğiz.

filter() fonksiyonunu çalıştıralım:

**>>>** filter(tek, l)

Burada filter() fonksiyonuna ilk parametre olarak tek() fonksiyonunu verdik. İkinci parametremiz ise bu fonksiyonu uygulayacağımız l adlı liste. Amacımız, l adlı liste üzerine tek() fonksiyonunu uygulayarak, bu liste içindeki öğelerden True çıktısı verenleri (yani tek sayıları) ayıklamak.

Şimdi de yukarıdaki koddan aldığımız çıktıya bakalım:

<filter object at 0x00F74F30>

Gördüğünüz gibi, bu fonksiyonu bu şekilde kullandığımızda elde ettiğimiz şey bir ‘filtre nesnesi’. Bu nesne içindeki öğeleri görebilmek için ne yapabileceğimizi biliyorsunuz:

**>>>** list(filter(tek, l))

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]

veya:

**>>>** print(\*filter(tek, l))

175 355 13 207 397 31 241 123 249 153

ya da:

**>>>** [i **for** i **in** filter(tek, l)]

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]

Gördüğünüz gibi, gerçekten de l adlı liste içindeki bütün tek sayılar süzüldü.

Gelin isterseniz filter() fonksiyonunu biraz daha iyi anlayabilmek için basit bir çalışma yapalım.

Elimizde bir sınıftaki öğrencilerin Matematik sınavından aldığı notları içeren bir sözlük var:

notlar = {'Ahmet' : 60,

'Sinan' : 50,

'Mehmet' : 45,

'Ceren' : 87,

'Selen' : 99,

'Cem' : 98,

'Can' : 51,

'Kezban' : 100,

'Hakan' : 66,

'Mahmut' : 80}

Okulda kullanılan not sistemine göre yukarıdaki notları şu şekilde sınıflandırabiliyoruz:

**def** not\_durumu(n):

**if** n **in** range(0, 50): **return** 'F'

**if** n **in** range(50, 70): **return** 'D'

**if** n **in** range(70, 80): **return** 'C'

**if** n **in** range(80, 90): **return** 'B'

**if** n **in** range(90, 101): **return** 'A'

Buna göre mesela print(not\_durumu(54)) gibi bir kod yazdığımızda bu notun karşılık geldiği ‘D’ sayısını alabiliyoruz. Peki biz bu notları belli bir ölçüte göre süzmek, ayıklamak istersek ne yapabiliriz? Örneğin notu 70’ten yukarı olan öğrencileri listelemek istersek nasıl bir kod yazabiliriz?

İşte böyle bir durumda filter() adlı gömülü fonksiyonu kullanabiliriz:

notlar = {'Ahmet' : 60,

'Sinan' : 50,

'Mehmet' : 45,

'Ceren' : 87,

'Selen' : 99,

'Cem' : 98,

'Can' : 51,

'Kezban' : 100,

'Hakan' : 66,

'Mahmut' : 80}

**def** süz(n):

**return** n >= 70

print(\*filter(süz, notlar.values()))

Gördüğünüz gibi, filter() fonksiyonu, süz() adlı fonksiyon ile belirlediğimiz ölçütü notlar adlı sözlüğün değerleri üzerine tek tek uygulamamızı sağlıyor.

hash()

Bu fonksiyon, belirli türdeki nesnelere bir karma değeri vermemizi sağlar. Örneğin:

**>>>** hash('istihza')

-840510580

**>>>** hash('python')

212829695

Ancak bu fonksiyonun ürettiği çıktı aynı nesne için bütün sistemlerde aynı olmayabilir. Yani örneğin yukarıdaki hash('istihza') komutu 32 bitlik ve 64 bitlik işletim sistemlerinde birbirinden farklı sonuçlar verebilir. Ayrıca bu fonksiyonun ürettiği karma değerlerinin birbiriyle çakışma ihtimali de yüksektir. Dolayısıyla bu fonksiyonu kullanarak, mesela parola girişleri için karma değeri üretmek doğru olmaz.

isinstance()

Hatırlarsanız daha ilk derslerimizde öğrendiğimiz type() adlı bir fonksiyon vardı. Bu fonksiyonu bir nesnenin hangi veri tipinde olduğunu tespit etmek için kullanıyorduk:

**>>>** type('istihza')

<class 'str'>

İşte buna benzer şekilde, tip denetimi için kullanabileceğimiz bir fonksiyon daha var. Bu fonksiyonun adı isinstance().

Bu fonksiyonu şöyle kullanıyoruz:

**>>>** isinstance('istihza', str)

True

Gördüğünüz gibi 'istihza' gerçekten bir karakter dizisi (str) olduğu için komutumuz True çıktısı veriyor.

Bir de şuna bakalım:

**>>>** isinstance('istihza', list)

False

'istihza' bir liste (list) olmadığı için komutumuz bu kez False çıktısı verdi.

len()

Bu fonksiyon yardımıyla nesnelerin uzunluklarını hesaplayabileceğimizi biliyorsunuz:

**>>>** len('istihza')

7

**>>>** l = [1, 4, 5, 3, 2, 9, 10]

**>>>** len(l)

7

map()

Diyelim ki elimizde şöyle bir liste var:

**>>>** l = [1, 4, 5, 4, 2, 9, 10]

Amacımız bu liste içindeki her öğenin karesini hesaplamak. Bunun için şöyle bir yol izleyebiliriz:

**>>> for** i **in** l:

**...**  i \*\* 2

**...**

1

16

25

16

4

81

100

Böylece, istediğimiz gibi, bütün öğelerin karesini bulmuş olduk. Bu tür bir işlemi yapmanın bir başka yolu da map() adlı bir gömülü fonksiyondan yararlanmaktır. Dikkatlice bakın:

**>>> def** karesi(n):

**...**  **return** n \*\* 2

**...**

Burada bir n sayısının karesini hesaplayan bir fonksiyon tanımladık. Şimdi bu fonksiyonu l listesinin bütün öğeleri üzerine uygulayacağız:

**>>>** list(map(karesi, l))

[1, 16, 25, 16, 4, 81, 100]

max()

max() gömülü fonksiyonunun görevi, bir dizi içindeki en büyük öğeyi vermektir. Bu fonksiyon birkaç farklı parametre alır ve verdiği çıktı, aldığı parametrelerin türüne ve sayısına bağlı olarak değişiklik gösterebilir.

Bu fonksiyonu en basit şu şekilde kullanabilirsiniz:

**>>>** max(1, 2, 3)

3

max() fonksiyonu yukarıdaki şekilde çalıştırıldığında, kendisine verilen parametreler arasında en büyük olanı bulacaktır. Yukarıdaki parametrelerden en büyüğü 3 olduğu için de yukarıdaki komut 3 çıktısı verecektir.

Yukarıdaki kodların sağladığı etkiyi şu şekilde de elde edebiliriz:

**>>>** liste = [1, 2, 3]

**>>>** max(liste)

3

max() fonksiyonu yukarıda gösterildiği gibi birtakım isimsiz parametrelerle birlikte key adlı isimli bir parametre de alır. Bu parametre yardımıyla max() fonksiyonunun ‘en büyük’ kavramını hangi ölçüte göre seçeceğini belirleyebiliriz. Örneğin:

**>>>** isimler = ['ahmet', 'can', 'mehmet', 'selin', 'abdullah', 'kezban']

**>>>** max(isimler, key=len)

'abdullah'

max() fonksiyonu öntanımlı olarak, ‘en büyük’ kavramını sayısal büyüklük üzerinden değerlendirir. Yani herhangi bir key parametresi kullanılmadığında, bu fonksiyon otomatik olarak bir dizi içindeki en büyük sayıyı bulur. Ancak eğer biz istersek, yukarıdaki örnekte olduğu gibi, ‘en büyük’ kavramının uzunluk cinsinden değerlendirilmesini de sağlayabiliriz.

Yukarıdaki örnekte elimizde şöyle bir liste var:

**>>>** isimler = ['ahmet', 'can', 'mehmet', 'selin', 'abdullah', 'kezban']

Amacımız bu liste içindeki isimler arasından, en fazla harf içerenini bulmak. Bildiğiniz gibi Python’da bir karakter dizisinin uzunluğunu belirlemek için len() adlı bir fonksiyondan yararlanıyoruz. İşte aşağıdaki kod yardımıyla da max() fonksiyonunun ‘en büyük’ ölçütünü len() fonksiyonu üzerinden değerlendirmesini sağlıyoruz:

**>>>** max(isimler, key=len)

Bu arada key fonksiyonuna len() fonksiyonunu parantezsiz olarak verdiğimize dikkat edin.

Gelin isterseniz max() fonksiyonunu biraz daha iyi anlamak için ufak bir çalışma yapalım.

Diyelim ki elimizde şöyle bir sözlük var:

askerler = {'ahmet' : 'onbaşı',

'mehmet' : 'teğmen',

'ali' : 'yüzbaşı',

'cevat' : 'albay',

'berkay' : 'üsteğmen',

'mahmut' : 'binbaşı'}

Amacımız bu sözlük içindeki en yüksek askeri rütbeyi bulmak. İşte bunun için max() fonksiyonundan yararlanabiliriz.

Bildiğiniz gibi, max() fonksiyonu ölçüt olarak sayısal büyüklüğü göz önüne alıyor. Elbette askeri rütbeleri böyle bir ölçüte göre sıralamak pek mümkün değil. Ama eğer şöyle bir fonksiyon yazarsak işler değişir:

**def** en\_yüksek\_rütbe(rütbe):

rütbeler = {'er' : 0,

'onbaşı' : 1,

'çavuş' : 2,

'asteğmen' : 3,

'teğmen' : 4,

'üsteğmen' : 5,

'yüzbaşı' : 6,

'binbaşı' : 7,

'yarbay' : 8,

'albay' : 9}

**return** rütbeler[rütbe]

Burada, rütbelerin her birine bir sayı verdik. En küçük rütbe en düşük sayıya, en yüksek rütbe ise en büyük sayıya sahip. Fonksiyonumuz bir adet parametre alıyor. Bu parametrenin adı rütbe. Yazdığımız fonksiyon, kendisine parametre olarak verilecek rütbeyi rütbeler adlı sözlükte arayıp, bu rütbeye karşılık gelen sayıyı döndürecek.

Bu bilgileri kullanarak kodlarımızın son halini düzenleyelim:

**def** en\_yüksek\_rütbe(rütbe):

rütbeler = {'er' : 0,

'onbaşı' : 1,

'çavuş' : 2,

'asteğmen' : 3,

'teğmen' : 4,

'üsteğmen' : 5,

'yüzbaşı' : 6,

'binbaşı' : 7,

'yarbay' : 8,

'albay' : 9}

**return** rütbeler[rütbe]

askerler = {'ahmet': 'onbaşı',

'mehmet': 'teğmen',

'ali': 'yüzbaşı',

'cevat': 'albay',

'berkay': 'üsteğmen',

'mahmut': 'binbaşı'}

Artık max() fonksiyonunu askerler adlı sözlük üzerinde uygulayabiliriz:

print(max(askerler.values(), key=en\_yüksek\_rütbe))

Böylece askerler adlı sözlüğün değerleri en\_yüksek\_rütbe() fonksiyonunun sunduğu ölçüt üzerinden ele alınacak ve en büyük sayı değerine sahip olan rütbe çıktı olarak verilecektir.

Yukarıdaki kodlar problemimizi çözüyor. Peki ama ya biz rütbe ile birlikte bu rütbeyi taşıyan askerin adını da öğrenmek istersek nasıl bir kod yazacağız?

Bunun için de şöyle bir kod yazabiliriz:

**for** k, v **in** askerler.items():

**if** askerler[k] **in** max(askerler.values(), key=en\_yüksek\_rütbe):

print(v, k)

Eğer isterseniz burada in işleci yerine == işlecini de kullanabilirsiniz:

**for** k, v **in** askerler.items():

**if** askerler[k] == max(askerler.values(), key=en\_yüksek\_rütbe):

print(v, k)

min()

min() fonksiyonu max() fonksiyonunun tam tersini yapar. Bildiğiniz gibi max() fonksiyonu bir dizi içindeki en büyük öğeyi buluyordu. İşte min() fonksiyonu da bir dizi içindeki en küçük öğeyi bulur. Bu fonksiyonun kullanımı max() ile aynıdır.

open()

Bu fonksiyon herhangi bir dosyayı açmak veya oluşturmak için kullanılır. Eğer dosyanın açılması veya oluşturulması esnasında bir hata ortaya çıkarsa IOError türünde bir hata mesajı verilir.

Bu fonksiyonun formülü şudur:

**>>>** open(dosya\_adi, mode='r', buffering=-1, encoding=**None**,

**...**  errors=**None**, newline=**None**, closefd=**True**, opener=**None**)

Gördüğünüz gibi, bu fonksiyon pek çok farklı parametre alabiliyor. Biz şimdiye kadar bu parametrelerin yalnızca en sık kullanılanlarını işlemiştik. Şimdi ise geri kalan parametrelerin ne işe yaradığını da ele alacağız.

Yukarıdaki formülden de görebileceğiniz gibi, open() fonksiyonunun ilk parametresi dosya\_adi’dır. Yani açmak veya oluşturmak istediğimiz dosya adını bu parametre ile belirtiyoruz:

**>>>** open('falanca\_dosya.txt')

Elbette eğer açmak istediğiniz dosya, o anda içinde bulunduğunuz dizinde değilse dosya adı olarak, o dosyanın tam adresini yazmanız gerekir. Mesela:

**>>>** open('/home/istihza/Desktop/dosya.txt')

Bu arada, dosya adresini yazarken ters taksim yerine düz taksim işaretlerini kullanmak daha doğru olacaktır. Bu taksim türü hem Windows’ta hem de GNU/Linux’ta çalışır. Ancak eğer ters taksim işaretlerini kullanacaksanız, dosya yolu içindeki sinsi kaçış dizilerine karşı dikkatli olmalısınız:

**>>>** f = open('test**\n**isan.txt')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

OSError: [Errno 22] Invalid argument: 'test\nisan.txt'

Burada problemin nisan.txt adlı dosyanın ilk harfi ile, bundan önce gelen ters taksim işaretinin birleşerek tesadüfen bir kaçış dizisi oluşturması olduğunu biliyorsunuz. Bu tür hatalara karşı ters taksim yerine düz taksim işaretlerini kullanabileceğiniz gibi r adlı kaçış dizisinden de yararlanabilirsiniz:

f = open(r'test\nisan.txt')

open() fonksiyonunun ikinci parametresi olan mode’un da ne olduğunu biliyorsunuz. Bu parametre yardımıyla, herhangi bir dosyayı hangi kipte açmak istediğimizi belirtebiliyoruz.

Bildiğiniz gibi, eğer mode parametresine herhangi bir değer vermezseniz Python ilgili dosyayı okuma kipinde açacaktır.

Bu parametreye verebileceğiniz değerleri şöyle özetleyebiliriz:

|  |  |
| --- | --- |
| Karakter | Anlamı |
| ‘r’ | Okuma kipidir. Öntanımlı değer budur. |
| ‘w’ | Yazma kipidir. Eğer belirtilen adda dosya zaten varsa o dosya silinir. |
| ‘x’ | Yeni bir dosya oluşturulup yazma kipinde açılır. |
| ‘a’ | Dosya ekleme kipinde açılır. Bu kip ile, varolan bir dosyanın sonuna eklemeler yapılabilir. |
| ‘b’ | Dosyaları ikili kipte açmak için kullanılır. |
| ‘t’ | Dosyaları metin kipinde açmak için kullanılır. Öntanımlı değerdir. |
| ‘+’ | Aynı dosya üzerinde hem okuma hem de yazma işlemleri yapılmasını sağlar. |

open() fonksiyonunun alabileceği bir başka parametre de buffering parametresidir. Bildiğiniz gibi, open() fonksiyonuyla bir dosyayı açıp bu dosyaya veri girdiğimizde bu veriler önce tampona alınacak, dosya kapandıktan sonra ise tamponda bekletilen veriler dosyaya işlenecektir. İşte bu buffering parametresi yardımıyla bu tampona alma işleminin nasıl yürüyeceğini belirleyebiliriz.

Eğer dosyaya işlenecek verilerin tampona alınmadan doğrudan dosyaya işlenmesini isterseniz buffering değerini 0 olarak belirlersiniz. Yalnız bu değer sadece ikili kipte etkindir. Yani bir dosyayı eğer metin kipinde açıyorsanız buffering parametresinin değerini 0 yapamazsınız.

Eğer dosyaya veri işlerken tampona alınan verilerin satır satır dosyaya eklenmesini isterseniz buffering değerini 1 olarak belirlersiniz. Bunun nasıl çalıştığını anlamak için şu örneği dikkatlice inceleyin:

**>>>** f = open('ni.txt', 'w', buffering=1)

**>>>** f.write('birinci satır**\n**')

14

**>>>** f.write('ikinci satır**\n**')

13

**>>>** f.write('aaa')

3

**>>>** f.write('**\n**')

1

Burada her write() komutundan sonra ni.txt adlı dosyayı açıp bakarsanız, şu durumu görürsünüz:

* f.write('birinci satır\n') komutuyla dosyaya bir satırlık veri ekledik ve bu veri dosyaya anında işlendi.
* f.write('ikinci satır\n') komutuyla dosyaya bir satırlık başka bir veri daha ekledik ve bu veri de dosyaya anında işlendi.
* f.write('aaa') komutuyla eklenen veri satır değil. Çünkü satır sonuna işaret eden satır başı kaçış dizisini kullanmadık.
* f.write('\n') komutuyla satır başı kaçış dizisini eklediğimiz anda bir önceki karakter dizisi ('aaa') de dosyaya eklenecektir.

Ancak buffering parametresi bu 1 değerini yalnızca metin kipinde alabilir. Bu kısıtlamayı da aklımızın bir kenarına not edelim…

0 ve 1 dışında buffering parametresine 1’den büyük bir değer verdiğinizde ise tampon boyutunun ne kadar olacağını kendiniz belirlemiş olursunuz.

Yalnız çoğu durumda buffering parametresine herhangi bir özel değer atamanız gerekmeyecektir. Bu parametreye herhangi bir değer atamadığınızda, kullandığınız işletim sistemi tampona alma işlemlerininin nasıl yürütüleceğine ve tampon boyutuna kendisi karar verecektir. İşletim sisteminin sizin yerinize verdiği bu karar da çoğunlukla istediğiniz şey olacaktır… Eğer kendi sisteminizde öntanımlı tampon boyutunun ne olduğunu merak ediyorsanız şu komutları kullanabilirsiniz:

**>>> import** **io**

**>>>** io.DEFAULT\_BUFFER\_SIZE

Çoğu sistemde bu değer 4096 ve 8192 bayt olacaktır.

open() fonksiyonunun alabileceği bir başka parametre de encoding parametresidir. Bu parametre, dosyanın hangi karakter kodlaması ile açılacağını belirler. Örneğin bir dosyayı ‘UTF-8’ karakter kodlaması ile açmak için şu komutu kullanıyoruz:

**>>>** f = open('dosya', encoding='utf-8')

Üzerinde işlem yaptığınız dosyalarda özellikle Türkçe karakter sorunları yaşamak istemiyorsanız, bir dosyayı açarken mutlaka encoding parametresinin değerini de ayarlamanızı tavsiye ederim.

Bir dosyayı açarken veya okurken herhangi bir karakter kodlama hatası ile karşılaştığınızda Python’ın ne tepki vermesi gerektiğini ise errors adlı parametre yardımıyla belirleyebilirsiniz.

Eğer bu parametreye strict değerini verirseniz karakter kodlama hataları programınızın ValueError türünde bir hata vererek çalışmayı kesmesine neden olacaktır. Bu parametreye herhangi bir değer vermediğinizde de Python sanki strict değerini vermişsiniz gibi davranır.

Eğer errors parametresine ignore değerini verirseniz kodlama hataları görmezden gelinecek, bu hataya sebep olan karakter silinecektir. Yalnız bu değerin veri kaybına yol açma ihtimalini de göz önünde bulundurmalısınız.

Eğer errors parametresine replace değerini verirseniz kodlama hatasına yol açan karater ‘?’ veya ‘ufffd’ karakterleri ile değiştirilecektir.

open() fonksiyonunun kabul ettiği bir başka parametre de newline adlı parametredir. Peki bu parametre ne işe yarar?

Windows ve GNU/Linux işletim sistemleri satır sonlarını birbirlerinden farklı şekilde gösterir. GNU/Linux’ta yazılmış dosyalarda satır sonları \n karakteri ile gösterilirken, Windows’ta yazılmış dosyalarda satır sonunda \r\n karakterleri bulunur. Eğer Windows ve GNU/Linux sistemleri arasında dosya alışverişi yapıyorsanız kimi durumlarda bu farklılık çeşitli sorunların ortaya çıkmasına yol açabilir. İşte dosyalarınızın hangi satır sonu karakterine sahip olacağını yukarıda bahsettiğimiz newline adlı parametre ile belirleyebilirsiniz. Örneğin:

**>>>** f = open('dosya', newline='**\n**')

Bu şekilde dosyanız hangi işletim sisteminde olursa olsun satır sonlarında \n karakterine sahip olacaktır.

Dosyaların metotlarını incelerseniz o listede fileno() adlı bir metodun olduğunu göreceksiniz. Bu metot, bize bir dosyanın ‘dosya tanımlayıcısını’ (*file descriptor*) verir. Dosya tanımlayıcıları, dosyaya işaret eden pozitif tam sayılardır. 0, 1 ve 2 sayıları standart girdi, standart çıktı ve standart hata dosyalarına ayrılmış olduğu için, sizin açtığınız ve üzerinde işlem yaptığınız dosyaların tanımlayıcıları 2 sayısından büyük olacaktır.

Bir örnek verelim:

**>>>** f = open('ni.txt')

**>>>** f.fileno()

3

İşte burada gördüğünüz sayı, ni.txt adlı dosyanın ‘dosya tanımlayıcısıdır. Her dosyanın dosya tanımlayıcısı benzersizdir:

**>>>** g = open('zi.txt')

**>>>** g.fileno()

4

Python’da bir dosyayı open() fonksiyonuyla açarken dosya adını vermenin yanı sıra, dosyanın tanımlayıcısını da kullanabilirsiniz:

**>>>** z = open(4)

veya:

**>>>** z = open(g.fileno())

Bu sayede, eğer isterseniz, elinizdeki dosyalarla daha ileri düzeyli işlemler yapabilirsiniz. Bir örnek verelim.

Dediğimiz gibi, bir dosyanın tanımlayıcısı tek ve benzersizdir. Farklı dosyalar aynı tanımlayıcılara sahip olmaz:

**>>>** a = open('aaa.txt')

**>>>** a.fileno()

3

**>>>** b = open('bbb.txt')

**>>>** b.fileno()

4

Şimdi şu örneklere bakın:

**>>>** c = open(b.fileno(), closefd=**False**)

Bu şekilde b adlı dosyanın tanımlayıcısını kullanarak, aynı dosyayı bir de c adıyla açtık. Ancak burada kullandığımız closefd=False parametresine dikkat edin. Normalde dosyayı kapattığımızda dosyanın tanımlayıcısı serbest kalır ve başka bir dosya açıldığında bu tanımlayıcı yeni dosyaya atanır. Ama closefd parametresine False değeri verdiğimizde dosya kapansa bile, o dosyaya ait dosya tanımlayıcısı varolmaya devam edecektir.

pow()

Daha önceki derslerimizde pek çok kez örneklerini verdiğimiz bu fonksiyon İngilizcedeki *power* (kuvvet) kelimesinin kısaltmasından oluşur. Adının anlamına uygun olarak, bu fonksiyonu bir sayının kuvvetlerini hesaplamak için kullanıyoruz.

Bu fonksiyon en temel şekilde şöyle kullanılır:

**>>>** pow(2, 3)

8

Bu komutla 2 sayısının 3. kuvvetini hesaplamış oluyoruz.

pow() fonksiyonu toplamda üç farklı parametre alır. İlk iki parametrenin ne olduğunu yukarıda örnekledik. Üçüncü parametre ise kuvvet hesaplaması sonucu elde edilen sayının modülüsünü hesaplayabilmemizi sağlar. Yani:

**>>>** pow(2, 3, 2)

0

Burada yaptığımız şey şu: Öncelikle 2 sayısının 3. kuvvetini hesapladık. Elde ettiğimiz sayı 8. Ardından da bu sayının 2’ye bölünmesi işleminden kalan sayıyı elde ettik. Yani 0. Çünkü bildiğiniz gibi 8 % 2 işleminin sonucu 0’dır. Dolayısıyla yukarıdaki komut şuna eşdeğerdir:

**>>>** (2 \*\* 3) % 2

0

Ancak önceki derslerimizde de söylediğimiz gibi, pow() fonksiyonu çoğunlukla yalnızca ilk iki parametresi ile birlikte kullanılır:

**>>>** pow(12, 2)

144

print()

print() fonksiyonunu artık gayet iyi tanıyoruz. Bu fonksiyonu, bildiğiniz gibi, kullanıcılarımıza birtakım mesajlar göstermek için kullanıyoruz.

Kullanımını daha önce ayrıntılı bir şekilde anlatmış olduğumuz bu fonksiyonu şu şekilde formüle edebiliriz:

print(deg1, deg2, deg3, ..., sep=' ', end='**\n**', file=sys.stdout, flush=**False**)

Burada;

**degx**

Çıktı verilecek değerlerin ne olduğunu belirtir. Buraya 256 adete kadar değer yazabilirsiniz.

**sep**

Çıktı verilirken değerlerin arasına hangi karakterin yerleştirileceğini belirtir. Bu değer öntanımlı olarak boşluk karakteridir.

**end**

Çıktı verilecek son değerin ardından hangi karakterin iliştirileceğini belirtir. Bu değer öntanımlı olarak satır başı (\n) karakteridir.

**file**

Çıktıların hangi dosyaya yazılacağını belirtir. Öntanımlı olarak bu parametrenin değeri sys.stdout’tur. Yani print() fonksiyonu çıktılarını öntanımlı olarak standart çıktı konumuna gönderir.

**flush**

Bildiğiniz gibi, herhangi bir dosyaya yazma işlemi sırasında dosyaya yazılacak değerler öncelikle tampona alınır. İşlem tamamlandıktan sonra tampondaki bu değerler topluca dosyaya aktarılır. İşte bu parametre, değerleri tampona almadan doğrudan dosyaya gönderebilmemizi sağlar. Bu parametrenin öntanımlı değeri False’tur. Yani değerler dosyaya yazılmadan önce öntanımlı olarak öncelikle tampona gider. Ama eğer biz bu parametrenin değerini True olarak değiştirirsek, değerler doğrudan dosyaya yazılır.

quit()

Bu fonksiyonu programdan çıkmak için kullanıyoruz. Eğer bu fonksiyonu etkileşimli kabukta verecek olursanız etkileşimli kabuk kapanacaktır.

range()

Bu fonksiyonu belli bir aralıktaki sayıları listelemek için kullanıyoruz. Yani mesela 0 ile 10 arası sayıların listesini almak istersek şöyle bir komut yazabiliriz:

**>>>** l = range(0, 10)

Ancak burada dikkat etmemiz gereken bir özellik var: Bu fonksiyon aslında doğrudan herhangi bir sayı listesi oluşturmaz. Yukarıda l değişkenine atadığımız komutu ekrana yazdırırsak bunu daha net görebilirsiniz:

**>>>** print(l)

range(0, 10)

Bir de bu verinin tipine bakalım:

**>>>** type(l)

<class 'range'>

Gördüğünüz gibi, elimizdeki şey aslında bir sayı listesi değil, bir ‘range’ (aralık) nesnesidir. Biz bu nesneyi istersek başka veri tiplerine dönüştürebiliriz. Mesela bunu bir listeye dönüştürelim:

**>>>** list(l)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

veya bir demete:

**>>>** tuple(l)

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

ya da bir kümeye veya dondurulmuş kümeye:

**>>>** set(l) *#küme*

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

**>>>** frozenset(l) *#dondurulmuş küme*

frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

Bu ‘range’ nesnesini istediğiniz veri tipine dönüştürdükten sonra, dönüştürdüğünüz veri tipinin kuralları çerçevesinde elinizdeki veriyi işleyebilirsiniz.

range() fonksiyonundan elde ettiğiniz ‘range’ nesnesinin içeriğini elde etmek için bunu başka bir veri tipine dönüştürmenin yanısıra, bu nesne üzerinde bir for döngüsü de kurabilirsiniz:

**>>> for** i **in** range(10):

**...**  print(i)

**...**

0

1

2

3

4

5

6

7

8

9

Ya da yıldızlı parametreler yardımıyla bu nesneyi print() fonksiyonuna göndererek, bu nesneyi istediğiniz gibi evirip çevirebilirsiniz:

**>>>** print(\*range(10), sep=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Esasında, yukarıda nasıl kullanılacağına dair bazı örnekler verdiğimiz bu range() fonksiyonunu temel olarak şu şekilde formüle edebiliriz:

range(başlangıç\_değer, bitiş\_değeri, atlama\_değeri)

Aşağıdaki örneği tekrar ele alalım:

**>>>** range(0, 10)

Burada 0 başlangıç değeri, 10 ise bitiş değeridir. Buna göre oluşturulacak sayılar 0 ile 10 arasında olacaktır. Yalnız burada üretilecek sayı listesinde 0 sayısının dahil, 10 sayısının ise hariç olduğunu unutmuyoruz. Yani bu komutun bize vereceği ilk sayı 0; son sayı ise 9 olacaktır.

range() fonksiyonunda başlangıç değerinin öntanımlı değeri 0’dır. Dolayısıyla istersek biz yukarıdaki komutu şöyle de yazabiliriz:

**>>>** range(10)

Böylece Python bizim range(0, 10) komutunu kastettiğimizi varsayacaktır. Elbette eğer başlangıç değerinin 0 dışında bir değer olmasını istiyorsanız bunu özellikle belirtmeniz gerekir:

**>>>** range(10, 100)

Bu komut bize 10 ile (10 dahil) 100 arası (100 hariç) sayıları içeren bir ‘range’ nesnesi verecektir.

Yukarıda verdiğimiz formülden de göreceğiniz gibi, başlangıç\_değer ve bitiş\_değer dışında range() fonksiyonu üçüncü bir parametre daha alabiliyor. Bu parametreye atlama\_değeri adı verdik. Bu parametreyi şöyle kullanıyoruz:

**>>>** list(range(0, 10, 2))

[0, 2, 4, 6, 8]

Gördüğünüz gibi, range() fonksiyonuna üçüncü parametre olarak verdiğimiz 2 sayısı, 0 ile 10 arası sayıların ikişer ikişer atlanarak üretilmesini sağladı.

reversed()

Diyelim ki elimizde şöyle bir liste var:

**>>>** isimler = ['ahmet', 'mehmet', 'veli', 'ayşe', 'çiğdem', 'ışık']

Eğer bu listedeki isimleri ters çevirmek, yani şöyle bir liste elde etmek isterseniz:

['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']

… ne yapmanız gerektiğini biliyorsunuz. Bu amaç için liste dilimleme yöntemlerinden yararlanabilirsiniz:

**>>>** isimler[::-1]

['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']

İşte aynı işlevi reversed() adlı bir fonksiyon yardımıyla da yerine getirebilirsiniz:

**>>>** reversed(isimler)

<list\_reverseiterator object at 0x00EB9710>

Gördüğünüz gibi, tıpkı range() fonksiyonunda olduğu gibi, reversed() fonksiyonu da bize ürettiği öğelerin kendisi yerine, bir ‘nesne’ veriyor. Ama tabii ki bu bizim için bir sorun değil. Biz bu nesnenin içeriğini nasıl elde edebileceğimizi gayet iyi biliyoruz:

**>>>** list(reversed(isimler))

['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']

range() fonksiyonunu anlatırken sözünü ettiğimiz içerik elde etme yöntemlerini reversed() fonksiyonuna da uygulayabilirsiniz.

sorted()

Bu metot, daha önceki derslerimizden de bildiğiniz gibi, bir dizi içindeki öğeleri belirli bir ölçüte göre sıraya dizmemizi sağlıyor. Bununla ilgili çok basit bir örnek verelim:

**>>>** sorted('ahmet')

['a', 'e', 'h', 'm', 't']

Bu kodlar yardımıyla ahmet adlı karakter dizisi içindeki harfleri alfabe sırasına dizdik.

Elbette bu fonksiyonu sadece karakter dizileri üzerine uygulamıyoruz. sorted() adlı fonksiyon, dizi özelliği taşıyan her türlü nesne üzerine uygulanabilir. Mesela demetlerin ve listelerin bir dizi olduğunu biliyoruz. Dolayısıyla:

**>>>** sorted(('elma', 'armut', 'kiraz', 'badem'))

['armut', 'badem', 'elma', 'kiraz']

**>>>** sorted(['elma', 'armut', 'kiraz', 'badem'])

['armut', 'badem', 'elma', 'kiraz']

sorted() fonksiyonuna hangi türde bir veri tipi verirseniz verin, aldığınız çıktı her zaman bir liste olacaktır. Bunu unutmayın.

Gördüğünüz gibi, sorted() fonksiyonu nesneler üzerinde bir sıralama işlemi gerçekleştiriyor. Ancak bu fonksiyonun bir problemi var.

Dikkatlice bakın:

**>>>** isimler = ['ahmet', 'çiğdem', 'ışık', 'şebnem', 'zeynep', 'selin']

**>>>** sorted(isimler)

['ahmet', 'selin', 'zeynep', 'çiğdem', 'ışık', 'şebnem']

Bu fonksiyon, Türkçe karakter içeren öğeleri düzgün sıralayamaz.

Bu sorunu *kısmen* çözebilmek için locale adlı bir modül içindeki strxfrm() adlı bir fonksiyondan yararlanabilirsiniz:

**>>> import** **locale**

Henüz modülleri öğrenmemiş de olsak, bir modülü kullanabilmek için öncelikle o modülü ‘içe aktarmamız’ gerektiğini artık biliyorsunuz. Bu işlemi import adlı bir komut yardımıyla yaptığımızı da biliyorsunuz.

Şimdi de yerelimizi (*locale*) ‘Türkçe’ olarak ayarlayalım:

**>>>** locale.setlocale(locale.LC\_ALL, 'tr\_TR') *#GNU/Linux*

**>>>** locale.setlocale(locale.LC\_ALL, 'Turkish\_Turkey.1254') *#Windows*

Bu işlemleri yaptıktan sonra, sorted() fonksiyonunun key adlı bir parametresini kullanarak ve yine locale modülünün strxfrm() adlı fonksiyonundan faydalanarak Türkçe karakterler içeren listemizi sıralamayı deneyebiliriz:

**>>>** sorted(isimler, key=locale.strxfrm)

['ahmet', 'çiğdem', 'ışık', 'selin', 'şebnem', 'zeynep']

locale modülü içinde bulunan strxfrm() adlı fonksiyon, karakter dizilerinin, o anda etkin yerel neyse, ona göre muamele görmesini sağlar. Biz yukarıda yerelimizi Türkçe olarak ayarladığımız için strxfrm() fonksiyonu, sorted() ile yapılan alfabe sırasına dizme işleminin Türkçenin kurallarına göre yapılmasını sağlıyor.

Ancak bu yöntemin de sorunlu olduğunu bir süre sonra kendiniz de farkedeceksiniz. Mesela şu örneği inceleyin:

**>>>** sorted('afgdhkıi', key=locale.strxfrm)

['a', 'd', 'f', 'g', 'h', 'i', 'ı', 'k']

Gördüğünüz gibi, listede ‘i’ harfi ‘ı’ harfinden önce geliyor. Ama aslında bunun tersi olmalıydı.

İşte böyle bir durumda, kendi sıralama mekanizmamızı kendimiz icat etmeliyiz. Peki ama nasıl?

Bilgisayarlar farklı dillerdeki karakterleri her zaman doğru sıralayamasa da, sayıları her zaman doğru sıralar:

**>>>** sorted([10, 9, 4, 14, 20])

[4, 9, 10, 14, 20]

Bilgisayarların bu özelliğinden ve Python’daki sözlük veri tipinden yararlanarak kendi sıralama mekanizmamızı rahatlıkla icat edebiliriz.

Öncelikle harflerimizi yazalım:

**>>>** harfler = "abcçdefgğhıijklmnoöprsştuüvyz"

Burada Türk alfabesini oluşturan harfleri sırasına göre dizdik. Şimdi bu harflerin her birine bir sayı vereceğiz:

**>>>** çevrim = {'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4,

**...**  'e': 5, 'f': 6, 'g': 7, 'ğ': 8, 'h': 9,

**...**  'ı': 10, 'i': 11, 'j': 12, 'k': 13,

**...**  'l': 14, 'm': 15, 'n': 16, 'o': 17,

**...**  'ö': 18, 'p': 19, 'r': 20, 's': 21,

**...**  'ş': 22, 't': 23, 'u': 24, 'ü': 25,

**...**  'v': 26, 'y': 27, 'z': 28}

Yalnız böyle her harfe karşılık gelen sayıyı elle yazmak yorucu olacaktır. Bu işlemi daha kolay bir şekilde yapabilmek için farklı teknikleri kullanabilirsiniz. Mesela daha önce öğrendiğimiz sözlük üreteçlerinden yararlanabilirsiniz:

**>>>** çevrim = {i: harfler.index(i) **for** i **in** harfler}

Bu şekilde harfler değişkeni içindeki her bir harfin bir anahtar; bu harflerin harfler değişkeni içindeki sırasını gösteren her bir sayının ise bir değer olduğu bir sözlük oluşturmuş olduk.

Şimdi isim listemizi alalım karşımıza:

isimler = ["ahmet", "ışık", "ismail",

"çiğdem", "can", "şule"]

Normal bir sorted() işleminin yanlış sonuç döndüreceğini biliyoruz:

**>>>** sorted(isimler)

['ahmet', 'can', 'ismail',

'çiğdem', 'ışık', 'şule']

Aynı şekilde key parametresine locale.strxfrm değerinin verilmesi de yetersiz kalacaktır:

**>>>** sorted(isimler, key=locale.strxfrm)

['ahmet', 'can', 'çiğdem', 'ismail',

'ışık', 'şule']

Ama biraz önce oluşturduğumuz çevrim anahtarını kullanırsak durum bambaşka olacaktır:

**>>>** sorted(isimler, key=**lambda** x: çevrim.get(x[0]))

['ahmet', 'can', 'çiğdem', 'ışık', 'ismail', 'şule']

Ancak aslında burada da oldukça sinsi bir problem var. Bu metot ile kelime listesini oluşturan kelimeleri yalnızca ilk harflerine göre sıralıyoruz (x[0]). Peki ya aynı liste içinde ilk harfleri aynı olup, sonraki harflerde farklılaşan kelimeler varsa ne olacak? Yani mesela bu metot acaba ‘ismail’ ve ‘iskender’ kelimelerini doğru bir şekilde sıralayabilir mi? Bakalım:

harfler = "abcçdefgğhıijklmnoöprsştuüvyz"

çevrim = {i: harfler.index(i) **for** i **in** harfler}

isimler = ["ahmet", "ışık", "ismail", "çiğdem",

"can", "şule", "iskender"]

print(sorted(isimler, key=**lambda** x: çevrim.get(x[0])))

Buradan şu çıktıyı alıyoruz:

['ahmet', 'can', 'çiğdem', 'ışık', 'ismail', 'iskender', 'şule']

Gördüğünüz gibi ‘ismail’ ve ‘iskender’ kelimeleri doğru bir şekilde sıralanmadı; ‘iskender’ kelimesinin ‘ismail’ kelimesinden önce gelmesi gerekiyordu…

Demek ki şimdiye kadar kullandığımız bütün sıralama yöntemlerinin bir eksiği varmış. O halde başka bir metot bulmaya çalışalım.

Dikkatlice bakın:

harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'

çevrim = {i: harfler.index(i) **for** i **in** harfler}

**def** sırala(kelime):

**return** ([çevrim.get(kelime[i]) **for** i **in** range(len(kelime))])

isimler = ['ahmet', 'can', 'iskender', 'cigdem',

'ismet', 'ismail', 'ismit', 'çiğdem',

'ismıt', 'ışık', 'şule']

print(\*sorted(isimler, key=sırala), sep='**\n**')

Gelin bu kodları biraz inceleyelim.

Burada ilk gördüğümüz kodlar şunlar:

harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'

çevrim = {i: harfler.index(i) **for** i **in** harfler}

Esasında siz bu kodların anlamını biliyorsunuz. Önceki derslerimizde de aynı kodları birkaç kez kullanmıştık. Yalnız biz burada, örnek olması açısından, harfler değişkeni için değer olarak yalnızca küçük harfleri kullandık. Bu kodları daha kapsamlı bir program içinde kullanacaksanız bu değişkenin uygun yerlerine mesela büyük harfleri ve noktalama işaretleriyle sayıları da eklemek isteyebilirsiniz.

Sonraki satırlarda sırala() adlı bir fonksiyon tanımladık:

**def** sırala(kelime):

**return** ([çevrim.get(kelime[i]) **for** i **in** range(len(kelime))])

Burada liste üreteçlerinden yararlandığımıza dikkatinizi çekmek isterim:

[çevrim.get(kelime[i]) **for** i **in** range(len(kelime))]

Bu kod yardımıyla kelime içinde geçen her bir harfi çevrim adlı sözlükte sorgulayarak, sözlükte ilgili harfe karşılık gelen sayıyı buluyoruz.

Aslında bu kodları daha iyi anlayabilmek için Python’daki sorted() fonksiyonunun mantığını biraz daha derinlemesine incelememiz gerekiyor. Gelin şimdi bu inceleme işini yapalım:

Diyelim ki elimizde şöyle bir liste var:

elemanlar = [('ahmet', 33, 'karataş'),

('mehmet', 45, 'arpaçbahşiş'),

('sevda', 24, 'arsuz'),

('arzu', 40, 'siverek'),

('abdullah', 30, 'payas'),

('ilknur', 40, 'kilis'),

('abdurrezzak', 40, 'bolvadin')]

Bu liste, her biri ‘isim’, ‘yaş’ ve ‘memleket’ bilgilerini içeren üç öğeli birer demetten oluşuyor. Eğer biz bu liste üzerine sorted() fonksiyonunu uygularsak:

print(\*sorted(elemanlar), sep='**\n**')

Python elemanları demetlerin ilk öğesine göre sıralayacaktır. Yani isme göre.

Peki ya biz bu elemanları yaşa göre sıralamak istersek ne yapacağız?

Bu amacı gerçekleştirmek için şöyle bir kod yazabiliriz:

**def** sırala(liste):

**return** liste[1]

elemanlar = [('ahmet', 33, 'karataş'),

('mehmet', 45, 'arpaçbahşiş'),

('sevda', 24, 'arsuz'),

('arzu', 40, 'siverek'),

('abdullah', 30, 'payas'),

('ilknur', 40, 'kilis'),

('abdurrezzak', 40, 'bolvadin')]

print(\*sorted(elemanlar, key=sırala), sep='**\n**')

Bu örnek bize key parametresinin de ne işe yaradığını açık seçik gösteriyor. Eğer Python’ın kendi sıralama yönteminin dışında bir sıralama yöntemi uygulayacaksak, bu sıralama yönteminin ne olduğunu bir fonksiyon yardımıyla tarif edip bunu key parametresine değer olarak veriyoruz. Örneğin biz yukarıdaki Python’ın elemanlar adlı listeyi ilk sütuna (‘isim’ sütunu) göre değil, ikinci sütuna (‘yaş’ sütunu) göre sıralamasını istedik. Bunun için de şöyle bir fonksiyon yazdık:

**def** sırala(liste):

**return** liste[1]

Bu fonksiyon, kendisine parametre olarak verilen nesnenin ikinci öğesini döndürüyor. İşte biz sorted() fonksiyonunun key parametresine bu fonksiyonu verdiğimizde Python sıralama işleminde elemanlar listesinin ikinci öğesini dikkate alacaktır. Eğer Python’ın sıralama işleminde mesela üçüncü sütunu dikkate almasını isterseniz sırala() fonksiyonunu şöyle yazabilirsiniz:

**def** sırala(liste):

**return** liste[2]

Gördüğünüz gibi, elemanlar listesinin ikinci sütununda değeri aynı olan öğeler var. Mesela ‘arzu’, ‘ilknur’ ve ‘abdurrezzak’ 40 yaşında. Python bu öğeleri sıralarken, bunların listede geçtiği sırayı dikkate alacaktır. Ama bazen biz sıralamanın böyle olmasını istemeyebiliriz. Mesela bizim istediğimiz şey, değeri aynı olan öğeler için üçüncü sütunun (veya birinci sütunun) dikkate alınması olabilir. İşte bunun için de sırala() fonksiyonunu şu şekilde tanımlayabiliriz:

**def** sırala(liste):

**return** (liste[1], liste[2])

Gördüğünüz gibi burada sırala() fonksiyonu bize iki öğeli bir demet döndürüyor.

Kodlarımız tam olarak şöyle görünecek:

**def** sırala(liste):

**return** (liste[1], liste[2])

elemanlar = [('ahmet', 33, 'karataş'),

('mehmet', 45, 'arpaçbahşiş'),

('sevda', 24, 'arsuz'),

('arzu', 40, 'siverek'),

('abdullah', 30, 'payas'),

('ilknur', 40, 'kilis'),

('abdurrezzak', 40, 'bolvadin')]

print(\*sorted(elemanlar, key=sırala), sep='**\n**')

Kodlarımızı böyle yazdığımızda Python listeyi ilk olarak ikinci sütundaki ‘yaş’ değerlerine göre sıralar. Değeri aynı olan öğelerle karşılaştığında ise üçüncü sütundaki ‘memleket’ değerlerine bakar ve sıralamayı ona göre yapar.

Bütün bu açıklamalardan sonra yukarıdaki şu kodları daha iyi anlıyor olmalısınız:

harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'

çevrim = {i: harfler.index(i) **for** i **in** harfler}

**def** sırala(kelime):

**return** ([çevrim.get(kelime[i]) **for** i **in** range(len(kelime))])

isimler = ['ahmet', 'can', 'iskender', 'cigdem',

'ismet', 'ismail', 'ismit', 'çiğdem',

'ismıt', 'ışık', 'şule']

print(\*sorted(isimler, key=sırala), sep='**\n**')

Biz yine de her şeyin iyiden iyine anlaşıldığından emin olmak için durumu kısaca açıklayalım. Öncelikle ilgili fonksiyonu önümüze alalım:

**def** sırala(kelime):

**return** ([çevrim.get(kelime[i]) **for** i **in** range(len(kelime))])

Burada yaptığımız şey biraz önce yaptığımız şeyle tamamen aynı aslında. Tek fark, Python’ın sıralamada kullanmasını istediğimiz öğeleri tek tek elle yazmak yerine, bunları bir liste üreteci yardımıyla otomatik olarak belirlemek.

Eğer yukarıdaki kodları şöyle yazsaydık:

**def** sırala(kelime):

**return** (çevrim.get(kelime[0]))

Bu durumda Python sıralamada kelimelerin yalnızca ilk harflerini dikkate alacaktı. İlk harfi aynı olan kelimeleri ise bu yüzden düzgün sıralayamayacaktı. Elbette Python’ın önce ilk harfe, sonra ikinci harfe, sonra da üçüncü harfe bakmasını sağlayabiliriz:

**def** sırala(kelime):

**return** (çevrim.get(kelime[0]), çevrim.get(kelime[1]), çevrim.get(kelime[2]))

Ancak bu yöntemin uygulanabilir ve pratik olmadığı ortada. Kendi kendinize bazı denemeler yaparak bunu kendiniz de rahatlıkla görebilirsiniz.

Python’ın, sıralama yaparken kelimelerin önce ilk harflerini, sonra ikinci, sonra üçüncü, vb. harflerini karşılaştırmasını sağlamanın en uygun yolu şu olacaktır:

**def** sırala(kelime):

**return** ([çevrim.get(kelime[i]) **for** i **in** range(len(kelime))])

Gördüğünüz gibi, burada kelimelerdeki harflerin sırasını tek tek elle yazmak yerine, bunu bir for döngüsü içinde otomatik olarak yaptırıyoruz. Dolayısıyla sırala() fonksiyonuna verilen parametrenin mesela ahmet olduğu bir durumda yukarıdaki fonksiyon şu demeti döndürüyor:

**def** sırala('ahmet'):

**return** (çevrim.get('ahmet'[0]),

çevrim.get('ahmet'[1]),

çevrim.get('ahmet'[2]),

çevrim.get('ahmet'[3]),

çevrim.get('ahmet'[4]))

Mesela ‘can’ için ise şunu:

**def** sırala('can'):

**return** (çevrim.get('can'[0]),

çevrim.get('can'[1]),

çevrim.get('can'[2]))

Böylece Python, hangi uzunlukta bir isimle karşılaşırsa karşılaşsın, sıralama işlemini düzgün bir şekilde gerçekleştirebiliyor.

Bu bölümde Python’da sıralama konusunu epey ayrıntılı bir şekilde ele aldık.

*‘Sıralama’ konusuna ilişkin bir tartışma için*[*https://web.archive.org/web/20161017211231/www.istihza.com/forum/viewtopic.php?f=25&t=1523*](https://web.archive.org/web/20161017211231/www.istihza.com/forum/viewtopic.php?f=25&t=1523)*adresindeki konuyu inceleyebilirsiniz.*

slice()

Bildiğiniz gibi, birtakım öğelerden oluşan bir nesnenin yalnızca belli kısımlarını ayırıp alma işlemine ‘dilimleme’ adı veriliyor. Örneğin elimizde şöyle bir liste olduğunu düşünelim:

**>>>** l = ['ahmet', 'mehmet', 'ayşe', 'senem', 'salih']

5 öğeli bu listenin yalnızca ilk iki öğesini almak, yani dilimlemek için şu yapıyı kullanıyoruz:

**>>>** l[0:2]

['ahmet', 'mehmet']

Dilimleme işleminin şöyle bir formülden oluştuğunu biliyoruz:

l[başlangıç:bitiş:atlama\_değeri]

Başlangıç parametresinin öntanımlı değeri 0 olduğu için yukarıdaki kodu şöyle de yazabilirdik:

**>>>** l[:2]

['ahmet', 'mehmet']

Aynı listenin, ilk öğeden itibaren sonuna kadar olan bütün öğelerini almak için ise şunu yazıyoruz:

**>>>** l[1:]

Eğer bu listeyi, öğelerini ikişer ikişer atlayarak dilimlemek istersek de şu yolu takip ediyoruz:

**>>>** l[::2]

['ahmet', 'ayşe', 'salih']

Bu örnekte başlangıç ve bitiş parametrelerinin öntanımlı değerlerini kullandık. O yüzden o kısımları boş bıraktık. Öğeleri ikişer ikişer atlayabilmek için ise atlama\_değeri olarak 2 sayısını kullandık.

İşte yukarıdakine benzer dilimleme işlemleri için slice() adlı bir gömülü fonksiyondan da yararlanabiliriz. Dikkatlice bakın:

Listemiz şu:

**>>>** l = ['ahmet', 'mehmet', 'ayşe', 'senem', 'salih']

Bir ‘dilimleme’ (*slice*) nesnesi oluşturuyoruz:

**>>>** dl = slice(0, 3)

Bu nesneyi liste üzerine uyguluyoruz:

**>>>** l[dl]

['ahmet', 'mehmet', 'ayşe']

Gördüğünüz gibi, slice() fonksiyonunu yukarıda iki parametre ile kullandık. Tahmin edebileceğiniz gibi, bu fonksiyonunu formülü şu şekildedir:

slice(başlangıç, bitiş, atlama)

sum()

Bu fonksiyonun temel görevi, bir dizi içindeki değerlerin toplamını bulmaktır. Örneğin:

**>>>** l = [1, 2, 3]

**>>>** sum(l)

6

Bu fonksiyon genellikle yukarıdaki gibi tek parametreyle kullanılır. Ama aslında bu fonksiyon ikinci bir parametre daha alır. Dikkatlice bakın:

**>>>** l = [1, 2, 3]

**>>>** sum(l, 10)

16

Gördüğünüz gibi, Python sum() fonksiyonuna verilen ikinci parametreyi, birinci parametredeki toplam değerin üzerine ekliyor.

type()

type() fonksiyonunun görevi bir nesnenin hangi veri tipine ait olduğunu söylemektir. Bu fonksiyonu artık yakından tanıyorsunuz:

**>>>** type('elma')

<class 'str'>

zip()

Gelin isterseniz bu fonksiyonu bir örnek üzerinden açıklamaya çalışalım.

Diyelim ki elimizde şöyle iki farklı liste var:

**>>>** a1 = ['a', 'b', 'c']

**>>>** a2 = ['d', 'e', 'f']

Eğer bu listelerin öğelerini birbirleriyle eşleştirmek istersek zip() fonksiyonundan yararlanabiliriz.

Dikkatlice bakın:

**>>>** zip(a1, a2)

<zip object at 0x00FD0BE8>

Gördüğünüz gibi, yukarıdaki kod bize bir ‘zip’ nesnesi veriyor. Bu nesnenin öğelerine nasıl ulaşabileceğinizi biliyorsunuz:

**>>>** print(\*zip(a1, a2))

('a', 'd') ('b', 'e') ('c', 'f')

**>>>** list(zip(a1, a2))

[('a', 'd'), ('b', 'e'), ('c', 'f')]

**>>> for** a, b **in** zip(a1, a2):

**...**  print(a, b)

**...**

a d

b e

c f

Yukarıdaki çıktıları incelediğimizde, ilk listenin ilk öğesinin, ikinci listenin ilk öğesiyle; ilk listenin ikinci öğesinin, ikinci listenin ikinci öğesiyle; ilk listenin üçüncü öğesinin ise, ikinci listenin üçüncü öğesiyle eşleştiğini görüyoruz.

Bu özellikten pek çok farklı şekilde yararlanabilirsiniz. Örneğin:

**>>>** isimler = ['ahmet', 'mehmet', 'zeynep', 'ilker']

**>>>** yaşlar = [25, 40, 35, 20]

**>>> for** i, y **in** zip(isimler, yaşlar):

**...**  print('isim: **{}** / yaş: **{}**'.format(i, y))

**...**

isim: ahmet / yaş: 25

isim: mehmet / yaş: 40

isim: zeynep / yaş: 35

isim: ilker / yaş: 20

Burada isimler ve yaşlar adlı listelerin öğelerini zip() fonksiyonu yardımıyla birbirleriyle eşleştirdik.

vars()

Bu fonksiyon, mevcut isim alanı içindeki metot, fonksiyon ve nitelikleri listeler. Eğer bu fonksiyonu parametresiz olarak kullanırsak, daha önce gördüğümüz locals() fonksiyonuyla aynı çıktıyı elde ederiz:

**>>>** vars()

{'\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_name\_\_': '\_\_main\_\_',

'\_\_package\_\_': None, '\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>,

'\_\_doc\_\_': None}

Bu fonksiyonu, nesnelerin metotlarını ve niteliklerini öğrenmek için de kullanabilirsiniz:

**>>>** vars(str)

**>>>** vars(list)

**>>>** vars(dict)

Yukarıda sırasıyla karakter dizilerinin, listelerin ve sözlüklerin metotlarını listeledik. Bu yönüyle vars() fonksiyonu dir() fonksiyonuna benzer.

Böylece Python’daki gömülü fonksiyonları tek tek incelemiş olduk. Bu bölümde incelemediğimiz gömülü fonksiyonlar şunlar:

1. memoryview
2. iter
3. next
4. object
5. property
6. staticmethod
7. super
8. getattr
9. hasattr
10. delattr
11. classmethod
12. issubclass
13. setattr
14. \_\_import\_\_

Bu fonksiyonları, ilerleyen derslerle birlikte Python bilgimiz biraz daha arttığında ele alacağız.