LABORATUVAR ÇALIŞMASI 10 – Özyineleme, Sözlükler

Bu Çalışmanın Amacı

Bu çalışmadaki amacımız, özyineleme ve sözlükler konularında öğrendiklerimizi pekiştirmektir.

Özyineleme

Özyineleme, "bir ifadenin kendi kendini tekrar etmesi", "bir fonksiyonun kendi içinde kendini çağırması" olarak tanımlanabilir. Örnek verecek olursak:

```
def ozyineLeme():
    ozyineLeme()
```

Yukarıdaki kodu bir betik dosyasına yazarak "ozyineLeme" fonksiyonunu çağırdığımızda bu fonksiyon defalarca kendini çağıracak ve **maksimum özyineleme sayısı** aşıldığında bu iç içe çağırma işlemi sona erecektir. Fonksiyon çalıştırıldığında görülecek sonuç, aşağıdakine benzer olacaktır:

```
  File "C:/Python25/ozyineLeme.py", line 2, in ozyineLeme
    ozyineLeme()
  File "C:/Python25/ozyineLeme.py", line 2, in ozyineLeme
    ozyineLeme()
  File "C:/Python25/ozyineLeme.py", line 2, in ozyineLeme
    ozyineLeme()
  ...
  ...
RuntimeError: maximum recursion depth exceeded
```

Başka bir örneği inceleyelim:

```
def geri_say(tavan):
    if tavan <= 0:
        print 'Basla!'
    else:
        print tavan
        geri_say (tavan-1)
```

Bir betik dosyasında yer alan bu fonksiyonu, "**tavan**" değerine **3** vererek çalıştıracak olursak şu sonucu elde ederiz:

```
>>> geri_say(3)
3
2
1
Basla!
```

Burada ilk önce "**geri_say**" fonksiyonuna **3** değeri verilmektedir. Fonksiyon, **tavan** değeri **3** olduğu için **if** bloğuna girmeyip, **else** içerisine girecektir, **tavan** değerini (3) ekrana yazdıktan sonra, "tavan – 1 (yani **2**)" değeri ile kendini çağıracaktır. Bu satırdaki işlemi "**geri_say(2)**" olarak düşünebiliriz. Daha sonra fonksiyon (aslında fonksiyonun başka bir kopyası), **2** değeri girilerek sil baştan çağrıldığı için aynı işlemler yeniden yapılacak, **if** bloğu atlanarak **else** içerisinde ekrana **tavan** değeri (2) yazılacaktır ve fonksiyon," tavan – 1 (yani **1**)" değeri ile çağrılacaktır. Bu işlemin sonunda da "**geri_say(1)**" çağrısı yapılacak, ekrana tavan değeri olan **1** yazılacak ve en sonda "**geri_say(0)**" çağrısı yapılacaktır. Ancak, **0** değeri verilerek fonksiyon tekrar çağrıldığında "**0 <= 0**" şartı sağlanacağı için bu sefer **if** bloğuna girilirken **else** bloğuna girilmeyecektir. **if** bloğuna girildiğinde ekrana "**Basla!**" yazısı yazılarak program sonlanacaktır. Programın sonlanmasının nedeni, en son olarak girmiş olduğu **if** bloğunun içerisinde kendi kendini yeniden çağırmasını söyleyen herhangi bir komut bulunmamasıdır.

Sözlük

Sözlükleri, "**indeks değerleri tamsayılardan oluşabileceği gibi, karakter dizilerinden de oluşabilen listeler**" olarak tanımlayabiliriz. Sözlüklerde de listelerde olduğu gibi, herhangi bir indekste herhangi bir türde (karakter dizisi, tamsayı, ondalıklı sayı vs.) veri tutulabilir.

Bir sözlüğün tanımlanmasını ve kullanılmasını gösteren örnek aşağıdadır:

```
>>> tur_ing = dict()
>>> tur_ing['bir'] = 'one'
>>> tur_ing['iki'] = 'two'
>>> tur_ing['uc'] = 'three'
>>> tur_ing['bes'] = 'five'
>>> tur_ing['iki']
'two'
>>> type(tur_ing['uc'])
<type 'str'>
>>> tur_ing['dort']
***Hata Mesajı (olmayan bir indekse erişimden dolayı)***
>>> tur_ing
{'bes': 'five', 'iki': 'two', 'bir': 'one', 'uc': 'three'}
>>> type(tur_ing)
<type 'dict'>
>>> len(tur_ing)
4
```

Sözlüğü tek bir satırda tanımlamak da mümkündür:

```
>>> tur_ing = dict()
>>> tur_ing = {'bir': 'one', 'iki': 'two', 'uc': 'three',
                                            'bes': 'five'}
>>> tur_ing['uc']
'three'
```

Sözlükte yer alan indislerin isimlerini (indisler içerisine atanan değerleri değil) kontrol etmek için **in** komutu kullanılabilir. Aranan indeks isminin sözlükte bulunması durumunda **True**, diğer durumda ise **False** döndürür:

```
>>> tur_ing = dict()
>>> tur_ing = {'bir': 'one', 'iki': 'two', 'uc': 'three'}
>>> 'iki' in tur_ing
True
>>> 'two' in tur_ing
False
>>> 'ik' in tur_ing
False
```

Eğer değerlerin bulunup bulunmadığını kontrol etmek istiyorsak, "**values**" isimli fonksiyondan şu şekilde yararlanabiliriz:

```
>>> tur_ing = dict()
>>> tur_ing = {'bir': 'one', 'iki': 'two', 'uc': 'three'}
>>> degerLer = tur_ing.values()
>>> 'two' in degerLer
True
>>> 'iki' in degerLer
False
>>> 'tw' in degerLer
False
```

Sözlüklerde indeks değerlerleri tamsayılardan da oluşabilir:

```
>>> kare = dict()
>>> kare = {0 : 0, 1 : 1, 2 : 4, 3 : 9, 4 : 16, 5 : 25}
>>> kare[0]
0
>>> kare['0']
***Hata Mesajı***
>>> kare[4]
16
```

Tuple Veri Türü

Sıralanmış değerler topluluklarıdır. Gösterimi aşağıdaki gibi olup, tek bir elemana sahip olsa bile tanımlama esnasında bu elemandan sonra virgül konmalıdır (**t1** ve **t2**' de olduğu gibi):

```
>>> t = ('a', 'b', 'c', 'd', 'e')
('a', 'b', 'c', 'd', 'e')
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
>>> p = tuple('python')
>>> print p
('p', 'y', 't', 'h', 'o', 'n')
>>> print p[1]
y
>>> print p[3 : 5]
('h', 'o')
>>> p[4] = 'j'
***Hata Mesajı (Tuple veri türü, atamayı desteklemiyor.)***
>>> u = ('f',) + p[1:]
>>> print u
('f', 'y', 't', 'h', 'o', 'n')
```

Yukarıdaki kutuya bakarak, tuple veri türünün atamayı desteklemediğini görebiliriz. İki değişkenin değerlerini **değiş - tokuş** yapmak için tuple yapısı büyük kolaylık sağlar:

```
>>> a = 3
>>> b = 7
>>> a, b = b, a
>>> a
7
>>> b
3
```

"**split**" fonksiyonunu kullanarak, örneğin, bir elektronik posta adresini "@" işaretinden öncesi ve sonrası olmak üzere ikiye ayırma işlemini rahatlıkla yapabiliriz:

```
>>> adres = 'python-help@python.org'
>>> kuLLanici_adi, aLan_adi = adres.split('@')
>>> print kuLLanici_adi
python-help
>>> print aLan_adi
python.org
```

NOT: split fonksiyonu değer olarak liste döndürmektedir. Buradaki atama biçimi, tuple ataması mantığına uyduğu için bu örnek verilmiştir.

Bazı fonksiyonların geri döndürdüğü değerler **tuple** türünden olabilir. Örneğin "**divmod**" fonksiyonu bir tamsayıyı diğerine böldüğünde sonuç olarak hem bölümü hem de kalanı döndürür:

```
>>> sonuc = divmod(7, 2)
>>> print sonuc
(3, 1)
>>> boLum, kaLan = divmod(7, 2)
>>> print boLum
3
>>> print kaLan
1
```

Peki, bu şekilde değer döndüren fonksiyonların tanımlanması nasıl olmalı? Aşağıdaki betik dosyasında yer alan ve verilen bir tamsayının karesini ve karekökünü döndüren fonksiyonu inceleyelim:

```
def kare_ve_karekok(tamsayi):
    if ((type(tamsayi) == int) & (tamsayi >= 0)):
        return tamsayi**2, tamsayi**0.5
    else:
        print 'Lutfen pozitif bir tamsayi giriniz.'
```

F5 ile Python Shell ekranına geldiğimizde:

```
>>> kare_ve_karekok(16)
(256, 4.0)
>>> kare, karekok = kare_ve_karekok(25)
>>> print kare
625
>>> print karekok
5.0
```

Fonksiyonların değişken sayıda argüman almasını sağlamak için "*" işareti kullanılır.

```
def hepsini_yazdir(* argumanlar):
    print argumanlar
```

Bu betik dosyasını çalıştırırsak:

```
>>> hepsini_yazdir(5, 'klm', 7.0, "pqr")
(5, 'klm', 7.0, 'pqr')
```

Yukarıdaki fonksiyona daha fazla sayıda argüman verilse de hepsini 'tuple' şeklinde ekrana yazdıracaktır.

Benzer biçimde, birden fazla sayıda argüman alan bir fonksiyona, içerisinde o fonksiyonun alacağı sayıda değer bulunduran bir 'tuple' ı argüman olarak vermek için gene "*" işareti kullanılır. İki argüman alan **divmod** fonksiyonuna argüman olarak 'tuple' geçirilmesi ile ilgili örneği inceleyelim:

```
>>> t = (7, 3)
>>> divmod(t)
***Hata Mesajı (divmod, tuple'ı tek argüman gibi görüyor)***
>>> divmod(*t)
(2, 1)
```

tuple' lar, büyüklük-küçüklük, eşitlik yönünden karşılaştırılabilirler. İki 'tuple' karşılaştırılırken ilk önce birinci elemanlarına bakılır. Birinci elemanı diğerinden büyük olan 'tuple', diğerinden daha büyüktür (Geri kalan elemanlara bakılmaz.). Eşitlik halinde ise 2. elemana, 3. elemana... bakılır:

```
>>> (0, 1, 99) < (0, 2, 0)
True
>>> (1, 2, 3) < (1, 1, 77)
False
```

Yazdığımız programlarda; **karakter dizisi**, **liste**, **tuple** gibi veri türlerinden birini bir diğerine dönüştürme ya da tamsayı, ondalıklı sayı gibi türlerdeki verileri bu yapılardan herhangi birinde saklama ihtiyacı duyabiliriz. Aşağıdaki fonksiyonu inceleyelim:

```
def sarmaLa(kaynak, yapi):
    # yapi nin turu karakter dizisi ise
    if type(yapi) == type(""):
        # degeri karakter dizisi olarak dondur
        return str(kaynak)
    # yapi nin turu liste ise
    if type(yapi) == type([]):
        # degeri liste olarak dondur
        return [kaynak]
    # yapi nin turu ikisi de degilse degeri tuple olarak dondur
    return (kaynak,)
```

Burada "kaynak" isimli argümanın taşıdığı değer, "yapi" argümanının ait olduğu türe dönüştürülmektedir. Aşağıdaki örnek kullanımları inceleyelim:

```
>>> sarmaLa(7,"")
'7'
>>> sarmaLa(7,[])
[7]
>>> sarmaLa(7,())
(7,)
>>> sarmaLa(7,0)
(7,)
>>> sayi = 21
>>> st = sarmaLa(sayi, "")
>>> st
'21'
>>> type(st)
<type 'str'>
>>> Ls = sarmaLa(sayi, [])
>>> Ls
[21]
>>> type(Ls)
<type 'list'>
>>> tp = sarmaLa(sayi, ())
>>> tp
(21,)
>>> type(tp)
<type 'tuple'>
>>> sarmaLa((2, 3, 4, 5), [])
[(2, 3, 4, 5)]
>>> sarmaLa([2, 3, 4, 5], ())
([2, 3, 4, 5],)
>>> sarmaLa([2, 3, 4, 5], "")
'[2, 3, 4, 5]'
```

Görüldüğü gibi, **sarmaLa** fonksiyonunun 2. argümanı olan **yapi**' ya karakter dizisi türünden herhangi bir değer (örnekteki `""`) verilirse, 1. argüman olan **kaynak** ile taşınan değer karakter dizisine dönüştürülmektedir. Eğer **yapi**' ya liste türünden herhangi bir değer (örnekteki boş liste) verilirse 1. argüman olan **kaynak** ile taşınan değer **liste** içerisine atılarak elde edilen bu yeni liste döndürülmektedir. Eğer **yapi**' ya bunların haricinde başka türden herhangi bir değer verilirse, 1. argüman olan **kaynak** ile taşınan değer **tuple** içerisine atılarak elde edilen bu yeni tuple döndürülmektedir.

Alıştırmalar

Alıştırma – 1

Görev

Matematikte 1' den büyük pozitif bir tamsayı ile **1** arasındaki (bahsedilen tamsayı da dahil) tüm tamsayıların çarpımına, ilgili tamsayının **faktöriyel**i denir. 0 ve 1 sayılarının faktöriyeli **1**' e eşittir. Negatif tamsayıların ise faktöriyel hesaplaması yapılamaz. Örneğin:

3' ün faktöriyeli = **3!** = 3 * 2 * 1 = **6**
7' nin faktöriyeli = **7!** = 7 * 6 * 5 * 4 * 3 * 2 * 1 = **5040**

Özyineleme kullanarak "**faktoriyeL**" isimli bir faktöriyel alma fonksiyonu yazınız ve bunu "**Lab10_faktoriyeL.py**" isimli bir betik dosyasına kaydediniz. Fonksiyonunuz argüman olarak bir tamsayı alıp, değer olarak ise bu tamsayının faktöriyelini döndürmelidir. Negatif bir tamsayı verildiğinde ise ekrana uyarı olarak "**Negatif tamsayilarin faktoriyel hesaplamasi yapilamaz**." yazmalıdır. Sizden beklenen gerçekleştirimin çalıştırılmasına ait bir örnek gösterim aşağıdaki gibidir:

```
>>> faktoriyeL(6)
720
>>> faktoriyeL(-4)
Negatif tamsayilarin faktoriyel hesaplamasi yapilamaz.
>>> faktoriyeL(0)
1
>>> faktoriyeL(1)
1
```

İpucu

"Özyineleme" bölümünü inceleyiniz. Değer döndüren bir fonksiyon içerisindeki bir **if** bloğunda değer döndürmeden fonksiyonu sonlandırmak gerekiyorsa, tek başına "**return**" komutu kullanılmalıdır ('return' komutunun sağ tarafına herhangi bir şey yazılmamalıdır.). Örneğin, kendisine verilen iki sayıdan birinciyi ikinciye bölerek sonucu döndüren, ikinci sayı 0 ise sonuç döndürmeden uyarı veren bir fonksiyon aşağıdaki gibidir:

```
def tamsay_boLme(boLunen, boLen):
    sonuc = 0
    if boLen != 0:
        sonuc = boLunen/boLen
    else:
        print 'Bolen sayi sifir olamaz.'
        return
    return sonuc
```

Çalıştırıldığında görülecek sonuç:

```
>>> tamsayi_boLme(7,3)
2
>>> tamsayi_boLme(8,0)
Bolen sayi sifir olamaz.
```

Sonuç

Gerçekleştiriminizi ve / veya karşılaştığınız problemleri raporunuza yazınız.

Alıştırma – 2

Görev

Argüman olarak bir sözlük ve değer alan, sonuç olaraksa o değerin yer aldığı **tüm indeksleri** ekrana yazdıran (eğer o değer hiçbir indekste yer almıyorsa ekrana "Indeks bulunamadi." yazdıran) ve değer döndürmeyen bir fonksiyon hazırlayınız. Fonksiyonunuza "**ters_arama**" adını vererek "**Lab10_ters_arama.py**" adlı betik dosyasına kaydediniz. Fonksiyonun kullanılmasına dair örnek aşağıdadır:

```
>>> sehirLer = {'Samsun': 'Karad.', 'Antalya': 'Akd.',
                'Tokat': 'Karad.', 'Manisa': 'Ege'}
>>> ters_arama(sehirLer, 'Karad.')
Samsun
Tokat
>>> ters_arama(sehirLer, 'Akd.')
Antalya
>>> ters_arama(sehirLer, 'Ege')
Manisa
>>> ters_arama(sehirLer, 'Manisa')
İndeks bulunamadı.
```

Sonuç

Gerçekleştiriminizi ve / veya karşılaştığınız problemleri raporunuza yazınız.

Alıştırma – 3

Görev

"**0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 ...**" şeklinde devam eden ve her elemanın, kendisinden önceki iki elemanın toplamından oluştuğu kurallı diziye "**Fibonacci dizisi**" denir (Bu dizinin sıfırıncı elemanını **0**, yedinci elemanını **13**, onuncu elemanını **55** olarak düşünelim.). "**fibonacci**" adında bir fonksiyonumuz olduğunu ve argüman olarak negatif olmayan bir tamsayı alıp, sonuç olarak Fibonacci dizisinin o tamsayıya karşılık gelen indeksindeki elemanı döndürdüğünü düşünürsek, "**fibonacci(11)**" bize **89** değerini verecektir. **fibonacci** fonksiyonunun sonucu hesaplamak için yapacağı işlemler, argüman olarak aldığı tamsayı büyüdükçe artacaktır. **Bunun için, performans açısından belli bir değere kadarki tamsayıların Fibonacci karşılıklarının program içerisinde tutulmasında yarar vardır (küçük değerlerin daha sık kullanıldığını düşünürsek).**

10 ve 10' dan küçük tamsayılar (argüman olarak fonksiyona verilen) için toplama ve benzeri hesaplama işlemi yapmadan direkt sonuç döndüren, 10' dan büyük tamsayılar için hesaplama yaparak sonuç döndüren bir Fibonacci fonksiyonu oluşturunuz. Fonksiyonunuza "**hizLi_fibonacci**" ismini vererek "**Lab10_hizLi_fibonacci.py**" isimli betik dosyasına kaydediniz. Fonksiyonun çalıştırılmasına ait örnek aşağıdadır (**doctest** olarak ekleyiniz.):

```
>>> hizLi_fibonacci(7)
13
>>> hizLi_fibonacci(11)
89
>>> hizLi_fibonacci(19)
4181
>>> hizLi_fibonacci(0)
0
>>> hizLi_fibonacci(-7)
Lutfen pozitif bir tamsayi giriniz.
>>> hizLi_fibonacci(12.7)
Lutfen pozitif bir tamsayi giriniz.
```

İpucu

"Sözlük" başlığını inceleyiniz.

Sonuç

Gerçekleştiriminizi ve / veya karşılaştığınız problemleri raporunuza yazınız.

Alıştırma – 4

Görev

Sınırsız sayıda sayının tek bir fonksiyon çağrısı ile toplanabilmesi için, sınırsız sayıda argüman kabul ederek aldığı sayıları toplayan ve ekrana toplamı yazdıran bir fonksiyon hazırlayınız. Fonksiyonunuza "**sinirsiz_topLam**" ismini vererek "**Lab10_sinirsiz_topLam.py**" isimli betik dosyasına kaydediniz. Fonksiyonun çalıştırılmasına ait örnek aşağıdadır:

NOT: Fonksiyona, sadece tamsayı ve ondalıklı sayı girileceği garanti edilmiştir. Ayrıca hata kontrolü yapmanız gerekmemektedir.

```
>>> sinirsiz_topLam(3, 5.0, 8, -0.1, 0)
15.9
>>> sinirsiz_topLam(4, -5, 6)
5.0
```

İpucu

"Tuple Veri Türü" bölümünü inceleyiniz.

Sonuç

Gerçekleştiriminizi ve / veya karşılaştığınız problemleri raporunuza yazınız.