

Functions in Python

Aşağıda belirtilen yerleşik işlevler için resmi Python belgelerine hızlı bir şekilde bakmak büyük bir fayda sağlar.

Bazıları, içinde bulunan koşullu algoritmaya göre bool türünü döndürür. Örneğin; `all(yinelemeli)`, `any(iterable)` ve `callable(object)`.

Bazıları veri türlerini birbirine dönüştürmenize yardımcı olur. Örneğin; `bool()`, `float()`, `int()` ve `str()`.

Bazıları koleksiyon türlerini oluşturmanıza ve işlemenize izin verir. Gibi: `dict()`, `list()`, `tuple()`, `set()`, `len()`, `frozenset()`, `zip()`, `filter(function, iterable)` ve `enumerate(iterable)`.

Bazıları sayılarla uğraşır. : `max()`, `min()`, `sum()` ve `round()` gibi.

Diğerleri özel amaçlar için inşa edilmiş. Bazı karmaşık uygulamalar yapıyorlar. Mesela: `map(function, iterable, ...)`, `eval(expression[, globals[, locals]])`, `sorted(iterable)`, `open()`, `dir([object])`, `hash()`, and `help([object])`

S: Python işlevlerini açıklayın.

Y: İşlev, programın bir bölümü veya bir kez yazılmış ve programda gerektiğinde yürütülebilen bir kod bloğudur. İşlev, geçerli bir ada, parametre listesine ve gövdeye sahip bağımsız deyimler bloğudur. İşlevler, modüler görevleri gerçekleştirmek için programlamayı daha işlevsel ve modüler hale getirir. Python, görevleri tamamlamak için çeşitli yerleşik işlevler sağlar ve ayrıca bir kullanıcının yeni işlevler oluşturmaya izin verir. İki tür işlev vardır: Yerleşik İşlevler: `copy()`, `len()`, `count()` bazı yerleşik işlevlerdir. Kullanıcı tanımlı işlevler: Kullanıcı tanımlı işlevler olarak bilinen bir kullanıcı tarafından tanımlanan işlevler.

The basic formula syntax of user-defined function is :

```
def function_name(arguments) :
```

```
    execution body
```

```
In [2]: def first_function(argument_1, argument_2) :  
        print(argument_1**2 + argument_2**2)  
        # Tanımladığımız bu fonksiyon, argümanların karelerinin  
        # toplamını verir. Arayıp kullanalım.
```

```
In [4]: first_function(2, 3)  
        # here, the values (2 and 3) are allocated to the arguments  
  
13
```

```
In [6]: first_function(5,6)  
        # görüldüğü üzere yukarıda birkere fonksiyonumuzu tanımladık.  
        # burda sadece fonksiyonumuzu yazarak kolayca sonuca ulaşabildik.  
        # fonksiyonlar bu şekilde işimizi kolaylaştırıyorlar.  
  
61
```

```
In [25]: def number(a, b):  
         print(a*b+a//b)  
         # burada kendimiz bir fonksiyon oluşturduk.değişken tanımlar gibi fonksiyonumuzun adını(number örneğin) tanımladık.  
         # daha sonra parantez içinde parametreleri tanımladık(yani aslında değişken).
```

```
In [23]: number(10,5)  
         # kendi oluşturduğumuz fonksiyonu çağırarak sonuca ulaştık.  
         # number yazarak oluşturduğumuz fonksiyonu çağırdık.  
         # 10 ve 5 te argümanlarımızdır yani a ve b parametrelerine(değişkenlerine) atayarak işlemimizi gerçekleştirdik.  
  
52
```

```
In [9]: # şimdi de çarpma fonksiyonu tanımlayalım.  
def multiply(a, b) :  
    print(a * b)  
  
multiply(3, 5)  
multiply(-1, 2.5)  
multiply('amazing ', 3) # it's really amazing, right?  
  
15  
-2.5  
amazing amazing amazing
```

```
In [12]: # herhangi bir argüman kullanmadan bir fonksiyon tanımlayabiliriz.  
def motto() :  
    print("Don't hesitate to reinvent yourself!")  
  
motto() # it takes no argument  
  
Don't hesitate to reinvent yourself!
```

```
In [18]: # return keywords:

def multiply_1(a, b) :
    print(a * b) # it prints something

def multiply_2(a, b) :
    return a * b # returns any numeric data type value

multiply_1(10, 5)
print(multiply_2(10, 5))
# return u kullandığımızda sonuca ulaşmak için print kullanmamız gerekiyor.

50
50
```

Fark ettiğiniz gibi, çıktılar aynı. O zaman fark nedir? Pekala, ilk fonksiyon sadece girdiğiniz bazı verileri yazdırır. İkincisi, sayısal bir tür değeri üretir. Türlerini kontrol ederseniz şunları görürsünüz:

```
In [15]: print(type(multiply_1(10, 2)))
print(type(multiply_2(10, 5)))

20
<class 'NoneType'>
<class 'int'>
```

Yani programımızda ihtiyaç duyduğumuzda ilk fonksiyonun sonucunu NoneType verisi olduğu için kullanamıyoruz. Ancak ikincisi, gelecekte ihtiyaç duyduğumuzda kullanabileceğimiz tamsayı verilerdir. Python'un en iyi bilinen fonksiyonunu kullanarak bu konuya bir göz atalım.

```
In [16]: shadow_var = print("It can't be assigned to any variable")
print(shadow_var) # NoneType value can't be used
# Yukarıdaki örnekte print() fonksiyonunun sonucunu bir değişkene atayamayız.

It can't be assigned to any variable
None
```

```
In [ ]: def my_function(a,b):
        area = a*b
        return area
print(my_function(3, 4))
```

print anahtar sözcüğünü kullanarak bir işlevin çıktısını oluşturursanız, sonuç bir NoneType olacaktır. Ancak return anahtar sözcüğünü kullanırsanız, sonucun bir veri türü olacaktır.

```
In [20]: def my_function(a, b):
        hypotenuse = (a**2 + b**2)**0.5
        return hypotenuse

print(my_function(6,8))
# hipotenüsü bulma kodu

10.0
```

```
In [22]: def longer(a, b):
        if len(a) >= len(b):
            return a
        else:
            return b

print(longer('Richard', 'John'))
# verilen argümanları uzunluk bakımından karşılaştırma kodu.
# ikisi eşitsede ilk argümanı döndüren kod.

Richard
```

```
In [27]: def who(first, last) : # 'first' ve 'last' parametreler(veya değişkenlerdir)
        print('Adınız :', first)
        print('Soyadınız :', last)

who('Guido', 'van Rossum') # 'Guido' ve 'van Rossum' argümanlardır
print()
who('Marry', 'Bold') # 'Marry' ve 'Bold' da argümanlardır.

Adınız : Guido
Soyadınız : van Rossum

Adınız : Marry
Soyadınız : Bold
```

parametreler yeni bir fonksiyon tanımlanana kadar sabit kalır.

ama argümanlar her işlemde değişebilir. yani parametreler farklı argümanlar alabilir.

```
In [29]: who("joseph")
# görüldüğü üzere iki parametrelili oluşturduğumuz fonksiyona
# tek argüman verirsek hata veriyor.
```

```
-----
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_3656/4265310354.py in <module>
----> 1 who("joseph")
      2 # görüldüğü üzere iki parametrelili oluşturduğumuz fonksiyona
      3 # tek argüman verirsek hata veriyor.

TypeError: who() missing 1 required positional argument: 'last'
```

```
In [31]: def age(a):
        print("your age is :", a)
age(15)
# ama tek parametrelili fonksiyon oluşturursak hata vermez.
# bunda da tek parametreye iki argüman verirsek hata verir.
```

your age is : 15

Konumsal argümanlarla bir fonksiyon çağırırken, soldan sağa sırayla iletilmelidirler.

```
In [1]: def pos_args(a, b):
        print(a, 'is the first argument')
        print(b, 'is the second argument')
```

```
pos_args(3,4)
print()
pos_args(4,3)
```

3 is the first argument
4 is the second argument

4 is the first argument
3 is the second argument

Bir işlevi çağırdığınızda argümanların dizilerinin/pozisyonlarının sizi kısıtlamasına izin vermek istemiyorsanız, bu argümanları anahtar kelimelerle de çağırabilirsiniz. Yaygın ve geleneksel olarak, kwargs, anahtar kelime bağımsız değişkenlerinin kısaltması olarak kullanılır.

The formula syntax is : kwargs=values.

```
In [2]: def who(first, last) : # same structure as the previous one
        print('Your first name is :', first)
        print('Your last name is :', last)

who(first='Guido', last='van Rossum') # işlevi çağırmak farklıdır.
# değerleri fonksiyona geçirmek için kwargs kullandık.
```

Your first name is : Guido
Your last name is : van Rossum

```
In [3]: def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
        print("-- This parrot wouldn't", action, end=' ')
        print("if you put", voltage, "volts through it.")
        print("-- Lovely plumage, the", type)
        print("-- It's", state, "!")

# bir gerekli bağımsız değişkeni (voltage) ve üç isteğe bağlı bağımsız değişkeni (durum, eylem ve tür) kabul eder.
# Bu işlev aşağıdaki yollardan herhangi biriyle çağrılabilir:
```

```
In [4]: parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

```
-- This parrot wouldn't vroom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't vroom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
-- This parrot wouldn't vroom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

Voltaaj ve eylem parametrelerinin konumlarını fark ettiyseniz, anahtar kelime argümanlarını kullanırken sıralar veya konumlar önemli değildir.

```
In [6]: # Tanımlanan işlevler göz önüne alındığında, aşağıdaki tüm çağrılar geçersiz olacaktır:
parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

```
File "/tmp/ipykernel_6483/1778684260.py", line 3
    parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
                    ^
```

SyntaxError: positional argument follows keyword argument

```
In [9]: # Hiçbir argüman birden fazla değer alamaz. İşte bu kısıtlama nedeniyle başarısız olan bir örnek:
def function(a):
    pass # actually, 'pass' does nothing. it just moves to the next line of code
# aslında, 'geçmek' hiçbir şey yapmaz. sadece bir sonraki kod satırına geçer
function(0, a=0)
```

```
-----
TypeError                                Traceback (most recent call last)
/tmp/ipykernel_6483/2492535521.py in <module>
      3     pass # actually, 'pass' does nothing. it just moves to the next line of code
      4 # aslında, 'geçmek' hiçbir şey yapmaz. sadece bir sonraki kod satırına geçer
----> 5 function(0, a=0)
```

TypeError: function() got multiple values for argument 'a'

```
In [11]: def çift(a,b="çift"):
        if a % 2 ==0:
            print(a,b)
        else:
            print(a,b)
çift(3,b ="tek")
```

3 tek

Arbitrary Number of Arguments

```
In [1]: # Varsayılan değerlere sahip parametreler tarafından tanımlanan bir işlevi çağırırken,
# işleve herhangi bir argüman iletme zorunluluğu yoktur.
def city(capital, continent='Europe'):
    print(capital, 'in', continent)

city('Atina') # 'continent'
city('Ulaanbaatar', continent= 'Asia') # varsayılan değeri kwargs ile değiştirebiliriz.
city('Cape Town', 'Africa') # varsayılan değeri konumsal argümanlarla değiştirebiliriz.
```

Atina in Europe
Ulaanbaatar in Asia
Cape Town in Africa

args and *kwargs

The formula syntax is : *args.

```
In [3]: # Örneğin iki çeşit meyveyi alıp yazdıran bir fonksiyon tanımlayalım.
def fruiterer(fruit1, fruit2) :
    print('I want to get', fruit1, 'and', fruit2)

fruiterer('orange', 'banana')
```

I want to get orange and banana

```
In [5]: # Kullanıcı iki çeşitten fazla meyve almak isterse ne olur? Her kullanıcının kaç çeşit meyve
# gireceğini bilmediğimiz için,
# 'rastgele sayıda argüman' kullanmak en akıllı yöntemdir. *args yönetimin kullanırssak;
def fruiterer(*fruit) :
    print('I want to get :')
    for i in fruit :
        print('-', i)

fruiterer('orange', 'banana', 'melon', 'ananas')
```

I want to get :

- orange
- banana
- melon
- ananas

==>Yukarıda görebileceğiniz gibi, bir meyve (argüman) listesini bir parametreye (meyve) aktardık. çok kullanışlı değil mi!

==>Keyfi anahtar kelime argümanları (**kwargs) kullanmayı tercih etmeniz gerekiyorsa, aynı şekilde kullanabilirsiniz.

```
In [11]: # the formula syntax is : **kwargs.
def animals(**kwargs):
    for key, value in kwargs.items():
        print(value, "are", key)

animals(Carnivores="Lions", Omnivores="Bears", Herbivores="Deers", Nomnivores="Human")
# burada farkettiysek keyword kullanımında dictionary olarak çağırıyoruz.
```

Lions are Carnivores
Bears are Omnivores
Deers are Herbivores
Human are Nomnivores

==> Örnekte görebileceğiniz gibi, bu argüman türünde (**kwargs), argümanların sayısını ve atanan değer çiftlerini kendimiz belirleyebiliriz. Bu fonksiyonun bir sonraki çağrısında, yukarıdaki argüman çiftlerinden hem sayı hem de değer olarak farklı argümanlar kullanabiliriz.

```
In [8]: def animals(**kwargs):
        for key, value in kwargs.items():
            print(value, "are", key)

animals(Carnivores="Lions", Omnivores="Bears", animal = "monkey")
```

Lions are Carnivores
Bears are Omnivores
monkey are animal

==> Birden çok konum parametresi tarafından tanımlanan bir işlevi çağırırken, *arg sözdizimini parantez içinde kullanarak, tüm argümanları tek bir değişkenle işleve geçirebiliriz. Aynı şekilde; Birden çok anahtar kelime argümanı ile tanımlanmış bir fonksiyonu çağırırken, *kwargs sözdizimini parantez içinde kullanarak, sözlük formundaki tüm argümanları tek bir değişkenle fonksiyona geçirebiliriz. Aşağıdaki örnekleri dikkatlice inceleyin:*

```
In [10]: def brothers(bro1, bro2, bro3):
        print('Here are the names of brothers :')
        print(bro1, bro2, bro3, sep='\n')

family = ['tom', 'sue', 'tim']
brothers(*family)
# burada listenin elemanlarını kullanmış olduk. burada dikkat edilmesi gereken parametrelerin sayısı ile eleman sayısının
```

Here are the names of brothers :
tom
sue
tim

```
In [12]: def gene(x, y): # defined by positional args
        print(x, "belongs to Generation X")
        print(y, "belongs to Generation Y")

        dict_gene = {'y' : "Marry", 'x' : "Fred"}
        gene(**dict_gene) # we call the function by a single argument(variable)
        # burada da yine sayıladikkat etmek gerekiyor.
```

Fred belongs to Generation X
Marry belongs to Generation Y

```
In [13]: def gene(x='Solomon', y='David'): # defined by kwargs (default values assigned to x and y)
        print(x, "belongs to Generation X")
        print(y, "belongs to Generation Y")

        dict_gene = {'y' : "Marry", 'x' : "Fred"}
        gene(**dict_gene)
```

Fred belongs to Generation X
Marry belongs to Generation Y

==> Bir işleve kaç bağımsız değişkenin iletileceğinden emin olmadığımızda veya bir işleve depolanmış bir liste ya da bağımsız değişken grubu iletmek istediğimizde *args kullanırız*. *kwargs, bir işleve kaç tane anahtar kelime bağımsız değişkeni geçirileceğini bilmediğimizde kullanılır veya bir sözlüğün değerlerini anahtar kelime bağımsız değişkenleri olarak iletme için kullanılabilir. args ve kwargs tanımlayıcıları bir kuraldır, *bob* ve *billy de kullanabilirsiniz, ancak bu akıllıca olmaz.

```
In [15]: # Örnekte görebileceğiniz gibi, değişkenin adı (my_var) hem işlevde (func_var) hem de ana program akışının
        # en üstünde kullanılmıştır.
        # (func_var) işlevini çağırdığınızda veya doğrudan değişkeni (my_var) yazdırdığınızda,
        # muhtemelen aynı değişkenin farklı çıktılar ürettiğini fark etmişsinizdir.
        # Bunun nedeni, o değişkenin konumu (boşluğu), yani program akışında nerede veya hangi boşlukta tanımlandığıdır.
        my_var = 'outer variable'

        def func_var():
            my_var= 'inner variable'
            print(my_var)
```

```
func_var()
print(my_var)
# fonksiyonun dışında değişkeni tanımladığımız için değişkenimiz hep aynı kalcaktır.
```

inner variable
outer variable

Lambda Functions

The formula syntax is : lambda parameters : expression

Lambda'nın en önemli avantajları ve kullanımları şunlardır:

Parantez kullanarak kendi sözdizimi ile kullanabilirsiniz,

Ayrıca bir değişkene atayabilirsiniz,

Birkaç yerleşik işlevde kullanabilirsiniz,

Kullanıcı tanımlı işlevler (def) içinde faydalı olabilir.

```
In [34]: # Fonksiyonu parantez içine alarak:
        (lambda x: x**2)(2) # squares '2'
```

Out[34]: 4

```
In [35]: # Veya aynı sözdizimini kullanarak birden çok bağımsız değişken kullanabilirsiniz:
        print((lambda x, y: (x+y)/2)(3, 5)) # takes two int, returns mean of them
```

4.0

```
In [36]: # Ayrıca parantez içindeki lambda ifadesini bir değişkene atayabilirsiniz:
        average = (lambda x, y: (x+y)/2)(3, 5)
        print(average)
```

4.0

```
In [37]: # Bir değişkene bir işlev nesnesi atayarak:
        average = lambda x, y: (x+y)/2
        print(average(3, 5)) # we call
```

4.0


```
In [38]: echo_word = lambda x,y : x*y
print(echo_word('hello', 3))
```

hellohellohello

Lambda within Built-in (map()) Functions

Lambda within map() function :

The basic formula syntax is : map(function, iterable)

!!!==> map(), verilen işlevi liste, tanımlama grubu vb. gibi belirli bir yinelenebilir nesnenin her ögesine uyguladıktan sonra çıktıların bir listesini döndürür.

```
In [40]: iterable = [1, 2, 3, 4, 5]
result = map(lambda x:x**2, iterable)
print(type(result)) # it's a map type

print(list(result)) # we've converted it to list type to print

print(list(map(lambda x:x**2, iterable))) # you can print directly
```

```
<class 'map'>
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

```
In [42]: def square(n): # at least two additional lines of code
        return n**2

iterable = [1, 2, 3, 4, 5]
result = map(square, iterable)
print(list(result))
# görebileceğiniz gibi, en az iki ek kod satırı vardır. Ayrıca square işlevini tekrar kullanmayacağız
# çünkü onu sadece map() işlevi içinde kullanmamız gerekiyor.
```

```
[1, 4, 9, 16, 25]
```

```
In [44]: # Şimdi Lambda fonksiyonunda map() kullanarak birden fazla argüman içeren bir örnek vermeye çalışalım:
letter1 = ['o', 's', 't', 't']
letter2 = ['n', 'i', 'e', 'w']
letter3 = ['e', 'x', 'n', 'o']
numbers = map(lambda x, y, z: x+y+z, letter1, letter2, letter3)

print(list(numbers))
# Yukarıdaki örnekte Lambda tanımında <math>+</math> operatörünü kullanarak üç karakter dizisini birleştirdik.
```

```
['one', 'six', 'ten', 'two']
```

==> Ayrıca map() ögesinin yinelenebilir nesnelerden her öğeyi birer birer ve sırayla aldığını unutmayın.

```
In [46]: number_list = [1, 2, 3, 4, 5]

result = list(map(lambda x : x*3 , number_list))
print(result)
```

```
[3, 6, 9, 12, 15]
```

Lambda within filter() function :

The basic formula syntax is : filter(function, sequence)

!!!=> filter(), dizideki her öğenin doğru olup olmadığını test eden bir işlev (lambda) yardımıyla verilen diziyi (yinelenebilir nesneler) filtreler.

```
In [47]: # Listedeki çift sayıları filtreleyeceğimiz bir örnekle konuyu kavrayalım.
first_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

even = filter(lambda x:x%2==0, first_ten)
print(type(even)) # it's 'filter' type,
                 # in order to print the result,
                 # we'd better convert it into the list type

print('Even numbers are :', list(even))
```

```
<class 'filter'>
Even numbers are : [0, 2, 4, 6, 8]
```

==> Filter() ögesinin, işlevin sonucunun True veya False olmasına bağlı olarak, yinelenabilir nesnedeki her öğeyi filtrelediğini unutmayın.

```
In [49]: # Bu sefer, listedeki ilk 10 harfin sesli harflerini filtreleyeceğiz.
vowel_list = ['a', 'e', 'i', 'o', 'u']
first_ten = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

vowels = filter(lambda x: True if x in vowel_list else False, first_ten)

print('Vowels are :', list(vowels))
# Bu örnekte kullandığımız lambda tanımının sonuç olarak
# yalnızca True veya False verdiği bu konuya dikkatinizi çekiyoruz.
```

Vowels are : ['a', 'e', 'i']

```
In [50]: number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

result = list(filter( lambda x : True if x >= 6 in number_list else False, number_list))
print(result)
```

[6, 7, 8, 9, 10]

Lambda within def :

Kullanıcı tanımlı bir işlevde lambda deyimi kullanmak bize yararlı fırsatlar sağlar. Program akışımızda daha sonra kullanabileceğimiz bir işlev grubu tanımlayabiliriz. Aşağıdaki örneğe bir göz atın:

```
In [53]: def modular_function(n):
        return lambda x: x ** n

power_of_2 = modular_function(2) # def'den türetilen ilk alt işlev
power_of_3 = modular_function(3) # second sub-function derived from def
power_of_4 = modular_function(4) # third sub-function derived from def

print(power_of_2(2)) # 2 to the power of 2
print(power_of_3(2)) # 2 to the power of 3
print(power_of_4(2)) # 2 to the power of 4
# burada görüldüğü üzere önce n i tanımladık sonra x e değer gönderdik.
```

4
8
16

==> Modüler_function, n sayısı olan bir bağımsız değişken alır ve verilen herhangi bir x sayısının o n'ye göre kuvvetini alan bir işlev döndürür.

==> Bu kullanım, bir fonksiyonu esnek olarak kullanmamızı sağladı. Lambda sayesinde tek bir def'i istediğimiz argümanlarla farklı şekillerde kullanabildik. Tek bir tanımdan türetilen üç alt fonksiyon yarattık. Bu esneklik!

```
In [54]: # Bir önceki örnekle aynı mantıkla, içine geçirilen stringi tekrarlayan bir fonksiyon tanımlayabiliriz.
def repeater(n):
    return lambda x: x * n

repeat_2_times = repeater(2) # repeats 2 times
repeat_3_times = repeater(3) # repeats 3 times
repeat_4_times = repeater(4) # repeats 4 times

print(repeat_2_times('alex '))
print(repeat_3_times('lara '))
print(repeat_4_times('linda '))
```

alex alex
lara lara lara
linda linda linda linda

örnekler

```
In [55]: def modular_function(n):
        return lambda x: x ** n

power_of_3 = modular_function(3)
print(power_of_3(5))
```

125

```
In [56]: print((lambda x: x**3)(5))
```

125

```
In [57]: mean = lambda x, y: (x+y)/2
print(mean(8, 10))
```

9.0


```
In [58]: multiply = lambda x: x * 4  
add = lambda x, y: x + y  
print(add(multiply(10), 5))
```

45

```
In [59]: number_list = [1, 2, 3, 4]  
result = map(lambda x:x**3, number_list)  
print(list(result))
```

[1, 8, 27, 64]

```
In [60]: number_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
divisible_list = filter(lambda x:x%3==0, number_list)  
print(list(divisible_list))
```

[0, 3, 6, 9, 12]

The matter of arguments extract

Function(the matter of arguments)

Kullanım durumu, işlev tanımında hangi parametrelerin kullanılacağını belirleyecektir:

```
def a_function(pos1, pos2, /, pos_or_kwarg, *, kwarg1, kwarg2, **kwargs) :
```

özet olarak:

Parametrelerin adının kullanıcı tarafından kullanılmamasını istiyorsanız, yalnızca konumsal kullanın. Bu, parametre adlarının gerçek bir anlamı olmadığına, işlev çağrıldığında argümanların sırasını zorlamak istediğinizde veya bazı konumsal parametreler ve rastgele anahtar sözcükler almanız gerektiğinde kullanışlıdır.

Anahtar kelimeyi yalnızca adların bir anlamı olduğunda ve işlev tanımı adlarla açık olarak daha anlaşılır olduğunda veya kullanıcıların iletilen argümanın konumuna güvenmesini önlemek istediğinizde kullanın.

İşlevinizin kaç bağımsız değişkene ihtiyacı olduğunu belirleyemiyorsanız, rastgele sayıda bağımsız değişken (*args) kullanın. *args, işlevinizdeki konumsal bağımsız değişkenlerin bir listesiyle birlikte çalışabilirliğe sahip olmanızı sağlar.

İşlevinizdeki anahtar kelime bağımsız değişkenlerinin tam sayısını bilmiyorsanız **kwargs kullanabilirsiniz. **kwargs, bir anahtar/değer çiftleri sözlüğü ile birlikte çalışabilirliğe sahip olmanızı sağlar.

Fonksiyonu tanımlarken kullandığınız parametrelerin sırası, fonksiyonu çağırırken ilettiğiniz argümanların sırası kadar önemlidir.

Scope of the Variables (Optional)

Python'daki ad alanı ve kapsam terimleri hakkında size bazı teorik açıklamalar verelim. Bu konuyu net bir şekilde anlamanızı sağlamak için yine ilgili Python belgelerine bağlı kalacağız. Ayrıca bu Python belgelerini incelemenin size büyük fayda sağlayacağını düşünüyoruz.

Ad Alanı nedir?

Ad alanı, Python'daki her nesnenin ayrı bir ada sahip olduğu bir sistemdir. Bir nesne bir yöntem veya bir değişken olabilir. Başka bir deyişle, ad alanı, adlardan nesnelere bir eşlemedir. Çoğu ad alanı şu anda Python sözlükleri olarak uygulanmaktadır, ancak bu normalde hiçbir şekilde fark edilmez (performans dışında) ve gelecekte değişebilir.

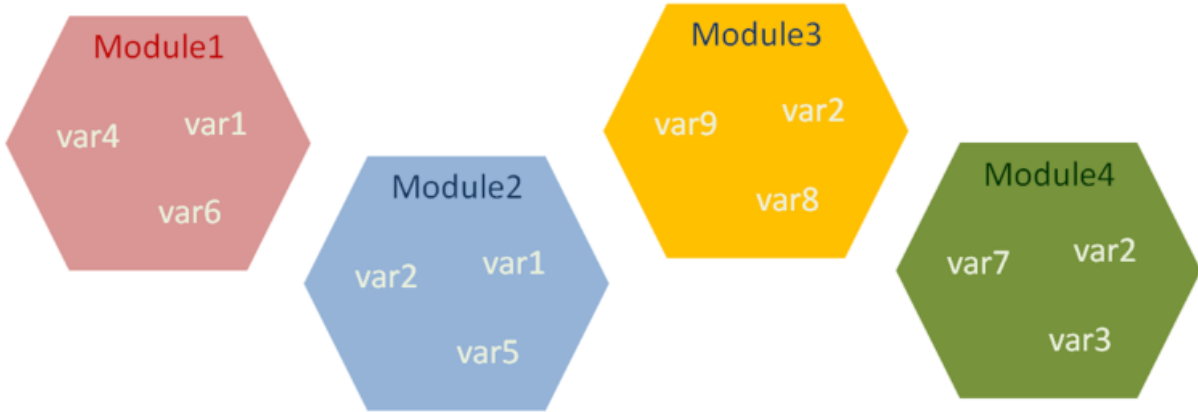
Örneğin: bilgisayarınızda hem C sürücüsünde hem de D sürücüsünde 'my_python' adlı iki dosyanız olduğunu düşünün. Dosya yolu sistemi ile istediğiniz dosyaya kolayca ulaşabilirsiniz. Dosya yollarına bakarak hangi dosyanın hangi sürücüde olduğu kolayca anlaşılabilir.

Program akışında, Python yorumlayıcısı, ad alanlarına bağlı olarak kodda hangi belirli yöneme veya değişkene erişmeye çalıştığını anlar.

Ad alanları farklı anlarda oluşturulur ve farklı ömürleri vardır. Yerleşik adları içeren ad alanı, Python yorumlayıcı başlatıldığında oluşturulur ve asla silinmez.

Aşağıdaki şekilde, aynı ada sahip bazı değişkenlerin aynı anda farklı modüllerde (ad alanlarında) olduğunu görebilirsiniz. Bu sözdizimini kullanarak istediğiniz değişkenle çalışabilirsiniz: `module.variable`. Bu rakamdan yola çıkarak `Module2`'de `var1`'i şöyle çağırabiliriz: `module2.var1`

Şimdilik modülün ne olduğunu ve tanımını bilmenize gerek yok. Sonraki derslerde detaylı olarak inceleyeceksiniz.



Kapsam nedir?

Kapsam, program akışında değişkenlerin nerede veya hangi alanda tanımlandığını açıklayan bir kavramdır. Bu kavram programlamada önemli bir yere sahiptir. Başka bir deyişle, kapsam, bir Python programının bir ad alanına doğrudan erişilebildiği metinsel bir bölgesidir. Burada 'doğrudan erişilebilir', bir ada yapılan koşulsuz referansın, adı ad alanında bulmaya çalışması anlamına gelir.

Kapsam terimi daha çok iç içe işlevler, modüller ve değişkenlerin kullanımına göre ana program akışı ile ilgilidir. Bir değişkenin erişilebilirliğini ve varlığını açıklar.

Kapsam, adları nesnelerle (değişkenler) eşleştirmek için değişken adlarının bulunması gereken hiyerarşik sırayı tanımlar.

Şimdi tüm bu tanımları basit bir örnekle uygulamaya koyalım:

input :

```
my_var = 'outer variable'

def func_var():
    my_var= 'inner variable'
    print(my_var)

func_var()
print(my_var)
```

output :

```
inner variable
outer variable
```

Örnekte görebileceğiniz gibi, değişkenin adı (`my_var`) hem işlevde (`func_var`) hem de ana program akışının en üstünde kullanılmıştır. (`func_var`) işlevini çağırdığınızda veya doğrudan değişkeni (`my_var`) yazdığınızda, muhtemelen aynı değişkenin farklı çıktılar ürettiğini fark etmişsinizdir. Bunun nedeni, o değişkenin konumu (boşluğu), yani program akışında nerede veya hangi boşlukta tanımlandığıdır.

Kapsam kavramının teorik olarak ne olduğunu öğrendikten sonra global ve yerel değişkenleri inceleyelim.

Q: What is the namespace in Python?

A: Ad alanı, büyük projelerde daha kullanışlı olan kodu yapılandırmak ve düzenlemek için temel bir fikirdir. Ad alanı, bir programdaki adları kontrol etmek için basit bir sistem olarak tanımlanır. Adların benzersiz olmasını ve herhangi bir çakışmaya yol açmamasını sağlar. Ayrıca Python, sözlük biçiminde ad alanlarını uygular ve adların anahtar, nesnelerin de değer olarak işlev gördüğü ad-nesne eşlemeyi sürdürür.

Global and Local Variables

Python program akışında bir değişken tanımladığınızda, tanımlandığı alana bağlı olarak global veya yereldir.

Küresel değişken

Tanımladığınız değişken bir modülün en üst seviyesinde ise o değişken global hale gelir. Böylece, bu global değişkeni programınızdaki herhangi bir yerde bir kod bloğunda kullanma özgürlüğüne sahipsiniz.

Global değişkenler, fonksiyonlar arasında bazı etkileşimler yapmamızı sağlar. Örneğin, Clarusway'e başvuran bir öğrencinin kimlik bilgilerini global bir değişkende sakladığımızı varsayalım. Bu global değişkeni ders etkinlikleri ile ilgili tanımladığımız 3 farklı fonksiyonda defalarca kullandığımızı varsayalım. Global değişken, kişinin kimlik bilgileri değiştiğinde bize kolaylık sağlar. Yalnızca bu global değişkendeki bilgileri yeniden düzenlediğimizde, tüm fonksiyonlardaki değişkenlerimiz yeniden düzenlenecektir.

Yerel değişken

Bir fonksiyon gövdesinde tanımladığınız değişkenler yereldir. Bu değişkenin adı bu nedenle yalnızca bulunduğu işlev gövdesinde geçerlidir. Yerel değişkenler, global değişkenlerin neden olabileceği bazı karışıklık risklerini ortadan kaldırır.

Global ve yerel değişkenler arasındaki farkı kavramak için bu örneğe bir göz atalım:

input :

```
text = "I am the global one"

def global_func():
    print(text) # we can use 'text' in a function
                # because it's a global variable

global_func() # 'I am the global one' will be printed
print(text)  # it can also be printed outside of the function

text = "The globals are valid everywhere "

global_func() # we changed the value of 'text'
# 'The globals are valid everywhere' will be printed

def local_func():
    local_text = "I am the local one"
    print(local_text) # local_text is a local variable

local_func() # 'I am the local one' will be printed as expected


print(local_text) # NameError will be raised
# because we can't use local variable outside of its function
```

output :

```
I am the global one
I am the global one
The globals are valid everywhere
I am the local one
-----
NameError: name 'local_text' is not defined
```

Yukarıdaki örnekte, global bir değişkene sadece modülün üst seviyesinden değil, aynı zamanda fonksiyonun gövdesinden de erişilebileceğini gördük. Öte yandan, yerel bir

değişken yalnızca tanımlandığı işlevin gövdesinde geçerlidir. Böylece en yakın kapsam seviyesinden içeriden erişilebilir ve dışarıdan erişilemez.

 **Tips: Bu sorunları nerede kullanmanız gerekeceği konusunda bir sorunuz olabilir. Ancak, nispeten uzun bir algoritma yazıyorsanız, sonunda iç içe işlevler ve modüllerle çalışmanız gerekecektir.**

Q: What are local variables and global variables in Python?

A: Bir fonksiyonun dışında veya global uzayda bildirilen değişkenlere global değişkenler denir. Bu değişkenlere programdaki herhangi bir fonksiyon tarafından erişilebilir. Bir fonksiyon içinde tanımlanan herhangi bir değişken yerel değişken olarak bilinir. Bu değişken, global uzayda değil, yerel uzayda mevcuttur. Yerel değişkene işlevin dışında erişmeye çalıştığınızda hata verir.

LEGB Ranking Rule

Bir nesneyi (yöntem veya değişken) çağırdığınızda, yorumlayıcı adını aşağıdaki sırayla arar:

Locals: İlk aranan boşluk, bir fonksiyon gövdesinde tanımlanan yerel isimleri içerir.

Enclosing: En yakın çevreleyen kapsamdan başlayarak (içten dışa doğru) aranan herhangi bir çevreleyen işlevin kapsamı, yerel olmayan, aynı zamanda genel olmayan adları da içerir.

Globals : Geçerli modülün genel adlarını içerir. Modülünün en üst seviyesinde tanımlanan değişkenler.

Built-in: En dıştaki kapsam (en son aranan), yerleşik adları içeren ad alanıdır.

Yukarıda verilen sıralama LEGB Sıralama Kuralı olarak bilinir. Bir örnekte nasıl çalıştığını görelim:

input :

```
variable = "global"

def func_outer():
    variable = "enclosing outer local"
    def func_inner():
        variable = "enclosing inner local"
        def func_local():
            variable = "local"
            print(variable)
```

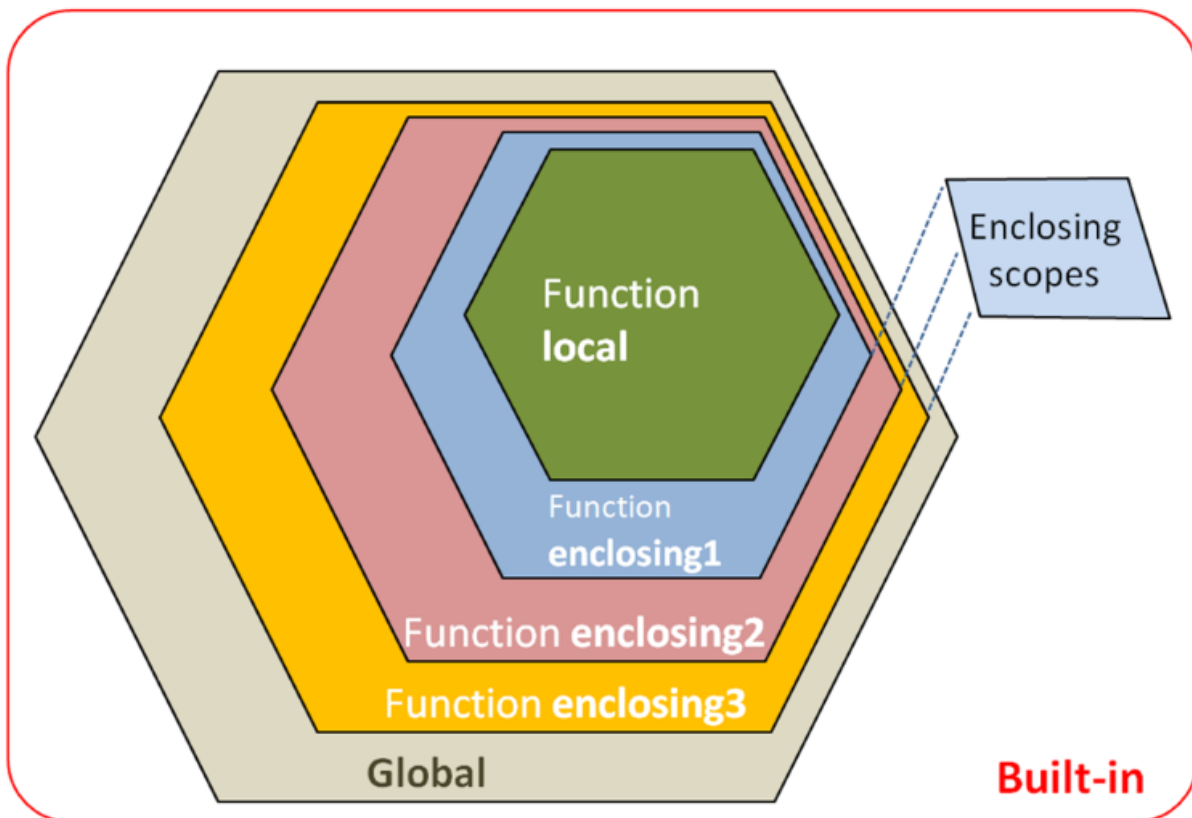
```
func_local()
func_inner()

func_outer() # prints 'local' defined in the innermost function
print(variable) # 'global' level variable holds its value
```

output :

```
local
global
```

Bu örnekte, kod satırlarının yürütülmesi sırasında yorumlayıcının 'değişken' adını çözmesi gerekir. Değişken isimlerinin arama sırası şu şekilde olacaktır: func_local'da 'local', func_inner'da 'including internal local', func_outer'da 'enclosing external local', globals ve yerleşik isimler. LEGB Kuralını aşağıdaki resimde inceleyebilirsiniz.



'global' and 'nonlocal'

Önceki dersten, bir fonksiyon gövdesinde tanımlanan bir değişkenin yerel hale geldiğini biliyorsunuz. Bazı durumlarda, fonksiyon gövdesinde global kapsam olarak tanımlanan değişkenlerle çalışmak isteriz. Normalde küresel olarak algılanırlar ve buna göre işlenirler.

Veya fonksiyon gövdesindeki yerel olmayan değişkenlerle çalışmamız gerekebilir. Global ve nonlocal anahtar kelimeleri bizi bu kısıtlamalardan kurtarıyor.

Keyword 'global'

Bir fonksiyon içinde global olarak tanımlanmış bir değişkene atanan değeri değiştiremezsiniz. Bunu yapmak için global anahtar kelimesini kullanıyoruz. Aşağıdaki örneği incellerseniz daha iyi anlayacaksınız.

input :

```
count = 1

def print_global():
    print(count)

print_global()

def counter():
    print(count)
    count += 1 # we're trying to change its value

print() # just empty line
counter()
```

output :

```
1

Traceback (most recent call last):
  File "code.py", line 11, in <module>
    counter()
  File "code.py", line 8, in counter
    print(count)
UnboundLocalError: local variable 'count' referenced before assignment
```

Yukarıdaki örnekte de görebileceğiniz gibi, bir fonksiyon içerisinde global bir değişkene yerel değişken ifadeleri içeren bir değer atamaya çalışırsanız, `UnboundLocalError` yükselcektir. Sayı değişkenini içeren bir ifade kullanarak sayım değişkenine bir değer atamaya çalıştık. Bunun nedeni, yorumlayıcının bu değişkeni yerel kapsamda bulamamasıdır. Öyleyse, bu sorunu çözmek için global anahtar kelimesini kullanalım.

input :

```
count = 1

def counter():
    global count # we've changed its scope
    print(count) # it's global anymore
    count += 1

counter()
counter()
counter()
```

output :

```
1
2
3
```

Bir önceki programdaki hatanın nedeni, değiştirmeye çalıştığımız değişkenin (`count`) yerel kapsamda yorumlayıcı tarafından bulunamamasıdır. Bunun nedeni, yerel kapsamda genel bir değişken kullanmamızdır.

Keyword 'nonlocal'

Öte yandan, yerel değişkenin kapsamını bir üst kapsama genişletmek için `nonlocal` anahtar sözcüğünü kullanabilirsiniz. Yerelleştirmeme örneklerini göz önünde bulundurun:

input :

```
def func_enclosing1():
    x = 'outer variable'
    def func_enclosing2():
        x = 'inner variable'
        print("inner:", x)
```

```
func_enclosing2()
print("outer:", x)

func_enclosing1()
```

output :

```
inner: inner variable
outer: outer variable
```

x değişkenini yerel olmayan hale getireceğiz, böylece onun iç değerini dış fonksiyonda (kapsam) kullanabiliriz. Görelim:

input :

```
def enclosing_func1():
    x = 'outer variable'
    def enclosing_func2():
        nonlocal x # its inner-value can be used in the outer scope
        x = 'inner variable'
        print("inner:", x)
    enclosing_func2()
    print("outer:", x)

enclosing_func1()
```

output :

```
inner: inner variable
outer: inner variable
```

💡 İpuçları: **Açıkçası, bu anahtar kelimeler programlamada yaygın olarak kullanılmazlar, ancak tartışmaya değer.**