Python'da Fonksiyonlar

Python'da **"fonksiyonlar"**, belirli bir görevi yerine getirmek için tanımlanan kod bloklarıdır. Tekrar eden işlemleri fonksiyonlar ile düzenleyerek daha okunabilir, temiz ve yeniden kullanılabilir kodlar yazılabilir. Python'da fonksiyonlar, **"def"** anahtar kelimesi ile tanımlanır ve genel olarak üç tür fonksiyon türü vardır;

- **yerleşik (built-in) fonksiyonlar**
- **kullanıcı tanımlı fonksiyonlar**
- **anonim fonksiyonlar (lambda fonksiyonları)**

Fonksiyon Tanımı ve Kullanımı

Bir fonksiyon tanımlamak için **"def"** anahtar kelimesi kullanılır ve ardından fonksiyonun adı yazılır. Fonksiyonun görevini yerine getirecek kod bloğu, tanım satırından sonra girintili bir şekilde yazılır. Fonksiyonlar, fonksiyon ismiyle birlikte aç-kapa parantezler ve varsa parametreler verilerek çağrılır.

```
def  fonksiyon ismi ( parametreler ):
    .
    . fonksiyon işlemleri
    .
    return (değerler)
```

Her fonksiyon tanımında parantezler içinde parametreler olması zorunlu değildir. **"fonksiyon_ismi()"** şeklinde kullanım fonksiyonun her hangi bir girdi parametresi olmadığı anlamına gelir. Fonksiyonların geri dönüş değeri ya da değerleri **"return"** kelimesiyle sağlanır ve **"return parametre"** şeklinde kullanılır. Geri dönen parametre tuple, list ya da değişken değerlerden oluşabilir.

Parametresiz fonksiyon

Parametresiz fonksiyonlar çoğunlukla bir programın genel akışı içinde olmayan ancak bilgi vermek ya da belirli mesajları göstermek için kullanılmaktadır. Bu tür fonksiyonlarda her hangi bir girdi parametresi bulunmaz. Ancak bazı durumlarda ana program içinde tanımlanmış genel değişkenler kullanılabilir. Kullanım amacına göre geri dönüş değerleri de olmaktadır. En basit parametresiz fonksiyon şu şekilde tanımlanabilir;

```
def merhaba():
    print("Merhaba, hoş geldiniz!")

# Fonksiyonu çağırma
merhaba()
```

Bu örnekte **"merhaba"** adlı bir fonksiyon tanımlandı ve bu fonksiyon çağrıldığında ekrana **"Merhaba, hoş geldiniz!"** mesajını yazdırıyor.

Parametreli fonksiyon

Fonksiyonların en genel kullanım şekli parametreli kullanımıdır. Bir fonksiyonun parametreleri değişken değerler olabildiği gibi list, tuple veri tipi değişkenleri de olabilir. Fonksiyon parametreleri sadece fonksiyon içinde tanımlı olurlar. Başka fonksiyonlar içinde kullanılamazlar En basit parametreli fonksiyon şu şekilde verilebilir.

```
def topla(a, b):
    return a + b

# fonksiyon kullanımı
sonuc = topla(4, 8)
print(sonuc)       # çıktı: 12
```

Burada **"topla"** adlı , **"a"** ve **"b"** olmak üzere iki parametre alır ve bu iki sayıyı toplayarak sonucu döndürür.

Parametreli fonksiyon: dizi örnekleri

Bir liste tipi olarak verilen ve tamsayı ya da ondalık değerlerden oluşan verilerin ortalama değerlerini hesaplayan bir fonksiyonu dikkate alalım. Bu durum fonksiyonun parametresi liste tipi olacaktır ve fonksiyon içindeki işlemler bu parametre tipine göre ayarlanması gereklidir.

```
def ortalama(veri):
    adet = 0
    toplam = 0

    for x in veri:
        adet += 1
        toplam += x

    ortlm = toplam / adet

    return ortlm

# fonksiyon kullanımı
liste = [3, 6, 8, 9, 13, 18, 19, 21]
sonuc = ortalama(liste)
print(sonuc)          # çıktı: 12.125
```

```
def ortalama(veri):
    adet, toplam = 0, 0
    notlar = veri.values()

    for x in notlar
        adet += 1
        toplam += x

    ortlm = toplam / adet
    return ortlm

liste = {"Ahmet":87, "Mert":68, "Fatma":92, "Canan":82, "Samet":75}
sonuc = ortalama(liste)
print(sonuc)               # çıktı: 80.8
```

Sözlük tipi olarak öğrenci adı ve başarı notu şeklinde verilen listeden öğrencilerin ortalama başarı notu hesaplayan bir fonksiyonu düşünelim.

Yerleşik Fonksiyonlar (Built-in Functions)

Python'daki **"yerleşik fonksiyonlar"**, dilin temel özelliklerinden biridir ve herhangi bir ek modül yüklemeye gerek kalmadan doğrudan kullanılabilen hazır fonksiyonlardan oluşur. Bu fonksiyonlar, Python'un veri işleme, listeleme, matematiksel hesaplamalar ve tip dönüşümleri gibi birçok işleminde hızlı ve pratik bir çözüm sunar. Yerleşik fonksiyonların bazı örnekleri şunlardır:

∗ Matematiksel Fonksiyonlar

- **abs()** : Bir sayının mutlak değerini döner. Negatif sayılar pozitif yapılır. Örnek: print(abs(-5)) # Çıktı: 5
- **round()** : Bir ondalıklı sayıyı en yakın tam sayıya yuvarlar. İkinci bir argüman olarak basamak sayısı belirtilebilir. Örnek: print(round(5.678, 2)) # Çıktı: 5.68
- **pow()** : Bir sayının belirtilen üssünü alır. pow(x, y) ifadesi, x üzeri yyi hesaplar. Örnek: print(pow(2, 3)) # Çıktı: 8
- **max()** : Verilen sayılar arasında en büyük olanı döner. Liste veya dizi gibi bir koleksiyonun en büyük elemanını da döndürebilir. Örnek: print(max(3, 7, 2, 9)) # Çıktı: 9
- **min()** : Verilen sayılar arasında en küçük olanı döner. Liste veya dizi gibi bir koleksiyonun en küçük elemanını da döndürebilir. Örnek: print(min(3, 7, 2, 9)) # Çıktı: 2
- **sum()** : Bir liste veya dizi gibi bir koleksiyonun elemanlarını toplar. Örnek: (sum([1, 2, 3, 4])) # Çıktı: 10
- **divmod()**: Bir sayıyı diğerine böler ve bölüm ile kalanı bir demet (tuple) olarak döner. Örnek: print(divmod(10, 3)) # Çıktı: (3, 1)
- **sqrt()** : Bir sayının karekökünü hesaplar. math.sqrt(x) şeklinde kullanılır. Örnek: print(math.sqrt(16)) # Çıktı: 4.0
- **degrees()**: Radyan cinsinden verilen bir açıyı dereceye çevirir. Örnek: print(math.degrees(math.pi)) # Çıktı: 180.0
- **radians()** : Derece cinsinden verilen bir açıyı radyana çevirir. Örnek: print(math.radians(180)) # Çıktı: 3.1415
- **sin()** : Radyan cinsinden bir açının sinüs değerini hesaplar. Örnek:print(math.sin(math.radians(30))) # Çıktı: 0.5

Yerleşik Fonksiyonlar (Built-in Functions)

∗ Tip Dönüştürme Fonksiyonları

- **int()** : Bir değeri tam sayıya dönüştürür. Ondalıklı sayıların tam kısmını alır; eğer string (yazı) veriliyorsa, bu string'in sayısal bir değeri temsil etmesi gerekir. Örnek: int(3.7) sonucu 3 döner. int("5") sonucu 5 döner.
- **float()** : Bir değeri ondalıklı sayıya dönüştürür. Tam sayılar ve sayısal değere sahip string ifadeler ondalıklı sayıya çevrilebilir. Örnek: float(3) sonucu 3.0 döner. float("7.2") sonucu 7.2 döner.
- **str()** : Bir değeri string (yazı) haline dönüştürür. Sayılar, listeler veya diğer veri türleri bu fonksiyonla metin formatına çevrilebilir. Örnek: str(25) sonucu "25" döner. str(3.14) sonucu "3.14" döner.
- **bool()** : Bir değeri Boolean (True veya False) türüne dönüştürür. Python'da sıfır, boş değerler veya None değeri False olarak kabul edilirken, geri kalan tüm değerler True olarak değerlendirilir. Örnek: bool(0) sonucu False döner. bool(5) sonucu True döner.
- **list()** : Bir değeri liste türüne dönüştürür. String veya başka bir iterable (örneğin bir tuple veya set) listeye çevrilebilir. Örnek: list("abc") sonucu ['a', 'b', 'c'] döner. list((1, 2, 3)) sonucu [1, 2, 3] döner.
- **tuple()** : Bir değeri demet (tuple) türüne dönüştürür. Özellikle bir liste veya başka bir iterable (tekrarlanabilir) veri yapısını tuple'a çevirmek için kullanılır. Örnek: tuple([1, 2, 3]) sonucu (1, 2, 3) döner. tuple("xyz") sonucu ('x', 'y', 'z') döner.
- **set()** : Bir değeri küme (set) türüne dönüştürür. Aynı elemanlardan birden fazla varsa sadece bir tanesini alır, yani tekrar eden elemanları çıkarır. Örnek: set([1, 2, 2, 3]) sonucu {1, 2, 3} döner. set("hello") sonucu {'h', 'e', 'l', 'o'} döner.
- **dict()** : Anahtar-değer çiftlerinden oluşan bir diziyi sözlük (dictionary) türüne dönüştürür. Genellikle liste veya tuple olarak verilen anahtar-değer çiftlerini dict formatına çevirir. Örnek: dict([("a", 1), ("b", 2)]) sonucu {'a': 1, 'b': 2} döner.

Yerleşik Fonksiyonlar (Built-in Functions)

∗ Dizi ve Listeleme Fonksiyonları

- **len()** : Bir dizinin veya koleksiyonun uzunluğunu döndürür. Örneğin, len("Python") çıktısı 6 olur.
- **sorted()** : Bir diziyi sıralar ve sıralanmış yeni bir liste döner. Örnek: sorted([3, 1, 4, 2]) sonucu [1, 2, 3, 4] döner. sorted([3, 1, 4, 2], reverse=True) sonucu [4, 3, 2, 1] döner.
- **enumerate()** : Bir diziyi numaralandırır ve her elemanın indeksini ve değerini içeren bir iterator döner. Listeye çevirmek için list() ile kullanılabilir. Örnek: list(enumerate(["elma", "armut", "çilek"])) sonucu [(0, 'elma'), (1, 'armut'), (2, 'çilek')] döner.
- **zip()** : İki veya daha fazla diziyi birleştirir ve her bir dizinin aynı sıradaki elemanlarını bir tuple olarak döner. Listeye çevirmek için list() ile kullanılabilir. Örnek: list(zip([1, 2, 3], ["a", "b", "c"])) sonucu [(1, 'a'), (2, 'b'), (3, 'c')] döner.

∗ Girdi ve Çıktı Fonksiyonları

- **print()** : Ekrana çıktı vermek için kullanılır. Örnek: print("Merhaba", "Dünya") sonucu Merhaba Dünya döner.
- **input()** : Kullanıcıdan veri almak için kullanılır. Girdi her zaman string olarak alınır. Örnek: isim = input("Adınız: ") sonucu olarak kullanıcı "Adınız: " ifadesini görür ve bir giriş yapabilir.
- **open()** :Dosya açmak için kullanılır. Varsayılan olarak okuma modunda ("r") açar. Örnek: with open("dosya.txt", "w") as dosya: içinde dosya.write("Merhaba!") ifadesi ile dosya.txt dosyasına Merhaba! yazılır.

∗ Yardımcı Fonksiyonlar

- **help()** : Belirtilen fonksiyon, sınıf ya da modül hakkında yardım bilgisi verir. Örneğin, help(str) ile "str" sınıfının açıklamasına ulaşabilirsiniz.
- **type()** : Bir değerin veri türünü döndürür. Örneğin, type(5) çıktısı <class 'int'> olur.

Kullanıcı Tanımlı Fonksiyon

Kullanıcı tanımlı fonksiyonlar (user-defined functions), belirli görevleri gerçekleştirmek veya tekrarlayan işlemleri düzenlemek amacıyla kullanıcı tarafından tanımlanan özel fonksiyonlardır. Bu fonksiyonlar, kodun daha düzenli, yeniden kullanılabilir ve anlaşılır hale getirilmesine olanak verir. Kullanıcı tanımlı fonksiyonların yazım düzeni genel fonksiyonu tanımına uyması gereklidir. Örnek olarak iki sayının ortalamasını hesaplayan bir fonksiyon şöyle yazılabilir:

```
def ortalama_bul(sayi1, sayi2):
    hesap = (sayi1 + sayi2) / 2
    return hesap

# fonksiyon kullanımı
ortlm = ortalama_bul(15, 25)
print("Ortalama hesabı: ", ortlm)   # çıktı: Ortalama hesabı: 20
```

Varsayılan Parametreli Kullanıcı Tanımlı Fonksiyonlar

Bir fonksiyonda parametrelere başlangıç ya da ön tanım değerler atanabilir. Bu şekilde fonksiyonlar daha esnek hale getirilebilir ve belirli parametrelerin girilmesi gerekmez. Aşağıdaki örnek, varsayılan bir parametre ile kullanıcıya merhaba demek için kullanılır.

```
def selamla(isim="Ziyaretçi"):
    print("Merhaba, {}!".format(isim))

# fonksiyon kullanımı
selamla()          # Çıktı: Merhaba, Ziyaretçi!
selamla("Ahmet")   # Çıktı: Merhaba, Ahmet!
```

Bu örnekte, **"isim"** parametresine bir değer verilmezse, fonksiyon otomatik olarak "Ziyaretçi" kelimesini kullanır.

Fonksiyonlarda esnek parametreler

Python fonksiyonlarında varsayılan parametrelerin haricinde esnek parametre kullanabilme özelliği de bulunmaktadır. Bir fonksiyonun parametrelerini tek tek belirtmeden değişkenlerin girilebilmesi özelliği esnek parametre olarak adlandırılır. Bunu içi iki tip argüman ***args** ve ****kwargs** ifadeleri kullanılır. ***args** ile tuple, list tipi veri girişi sağlanırken ****kwargs** ile de sözlük tipi veri girişi sağlanmaktadır.

***args** için aşağıdaki 2 örnek fonksiyonu dikkate alalım:

verilen tüm sayıların çarpımı bulunur

```
def carpim(*args):
    sonuc = 1
    for sayi in args:
         sonuc *= sayi
    print(sonuc)

# fonksiyonu çalıştıralım
carpim(2,3,4)            # 24
carpim(2,3,4,5,6)        # 720
```

Farklı sayıda girilen isimler yazılabilir

```
def yaz(ilk, *args):
    print(f"İlk eleman : {ilk}")
    for kelime in args:
         print(f"Esnek eleman : {kelime}")

# fonksiyonu çalıştıralım
yaz("Ahmet")
         İlk eleman: Ahmet
yaz("Ahmet", "Hasan")
         İlk eleman: Ahmet
         Esnek eleman: Hasan
yaz("Ahmet", "Hasan", "Hülya")
         İlk eleman: Ahmet
         Esnek eleman: Hasan
         Esnek eleman: Hülya
```

Parametre sayısındaki bu esnekliği sağlayan **"args"** kelimesi değildir, bu işlevi "*" ifadesi sağlamaktadır. **"*"** dan sonra her hangi bir isim konulabilir.

Fonksiyonlarda esnek parametreler

***kwargs** için ise aşağıdaki örnek fonksiyonu dikkate alalım:

```
def bilgi_yaz(**kwargs):
    for anahtar, deger in kwargs.items():
        print(f"{key} : {anahtar}")


# fonksiyonu çalıştıralım
bilgi_yaz(Ad="Fatih", Soyad="Göktürk")
          Ad : Fatih
          Soyad : Göktürk

bilgi_yaz(Ad="Fatih", Soyad="Göktürk", Meslek="Berber")
          Ad : Fatih
          Soyad : Göktürk
          Meslek : Berber
```

Python'da fonksiyonların 1'den fazla değeri geri dönüş değeri olarak ataması mümkündür. Bunun için geri dönüş değeri tuple ya da list veri tipi olarak ayarlamak yeterlidir.

```
def us_alma(x, a, b):
    us1 = x ** a
    us2 = x ** a
    return (us1, us2)
```

```
# fonksiyonu çalıştıralım
print(us_alma(3, 2, 4))          # (9, 81)

t, k = us_alma(3, 2, 4)
print(t, k)                      # 9, 81
```

Anonim fonksiyonlar (lambda fonksiyonları)

Python'daki anonim fonksiyonlar veya lambda fonksiyonları, hızlı ve kısa işlemler için tanımlanan, **adı olmayan** (anonim) fonksiyonlardır. Lambda fonksiyonları, özellikle bir fonksiyona geçici olarak ihtiyaç duyulduğunda veya basit işlemler gerçekleştirilirken tercih edilir. **"lambda"** anahtar kelimesi ile tanımlanır ve genellikle tek satırlık işlevleri yerine getirir. Lambda fonksiyonları, diğer fonksiyonlara parametre olarak geçirilebildiği gibi, hızlı hesaplamalar ve veri işleme süreçlerinde de oldukça kullanışlıdır. Lambda fonksiyonlarının genel yapısı şu şekildedir;

```
lambda  parametre_1 , parametre_2 ,... : işlem
```

"lambda" anahtar kelimesinden sonra parametreler gelir ve ardından **":"** işaretinden sonra bir işlem tanımlanır. Lambda, tanımlandığı anda çalışır ve sonucu döndürür. Bazı lamda fonksiyonu örnekleri:

Bir fonksiyon oluşturma

```
fonksiyon = lambda x: x**2 + 2*x + 3
print("f(3) = ", fonksiyon(3))   # çıktı: f(3) = 18
```

özel işlem tanımlama

```
hesap = lambda a,b,c: a + b / c
print(hesap(5,3,2))   # çıktı: 6.5
```

Liste İçinde Lambda Kullanımı

Lambda fonksiyonları, **map()**, **filter()** ve **sorted()** gibi fonksiyonlarla birlikte kullanılabilir. Bu örnekte, bir listedeki sayıların tek tek karesini almak için **map()** fonksiyonuyla birlikte lambda kullanılmıştır:

```
sayilar = [1, 2, 3, 4, 5]
kareler = list(map(lambda x: x**2, sayilar))
print(kareler)   # çıktı: [1, 4, 9, 16, 25]
```

Lambda fonksiyon örnekleri

Lambda fonksiyonlarıyla birlikte kullanılan diğer bir kullanışlı fonksiyon **filter()** fonksiyonudur. Adından da anlaşılacağı üzere veriler üzerinde filtreleme yapmak için kullanılmaktadır. Diğer bir deyişle bir veri içinde belirli bir şartı sağlayan değerleri elde etmek için kullanılır. **filter()** fonksiyonu **map()** fonksiyonu gibi girdi olarak iki parametre almaktadır. ilk parametre, bir koşulun sonucunu True/False olarak belirten bir fonksiyondur, ikinci parametre ise incelenecek dizidir.

Şimdi tamsayı değerlerine sahip verilen bir liste içindeki tek sayıları bulacak bir fonksiyonu düşünelim. Burada koşulumuz tek sayılardır ve bunu 2'ye bölündüğünde kalanı 1 olan sayılar kontrol etmekle elde edeceğiz.

```
sayilar = [5, 6, 11, 17, 20, 22, 27, 31, 38, 45]
tekler = list(filter(lambda x: x%2 == 1, sayilar))
print(tekler)

[5, 11, 17, 27, 31, 45]
```

Diğer bir örnek olarak da içinde birkaç şehir isimlerinin olduğu metin türü bir liste göz önüne alalım ve isim uzunluğu 7 harften büyük şehirleri bulmaya çalışıyor olalım. Bu durum koşulumuz 7 harften büyük olmaktır. Bir metin değişkenin uzunluğu len() fonksiyonu ile kontrol edilir. Buna göre yazılacak kod şöyle olacaktır.

```
sehirler = ["Adıyaman", "Kars", "Ankara", "Çanakkale", "Balıkesir"]
secim = list(filter(lambda x: len(x) > 7, sehirler))
print(secim)

['Adıyaman', 'Çanakkale', 'Balıkesir']
```

İç-içe fonksiyon yapısı

İç içe fonksiyonlar, bir görevin yalnızca belirli bir fonksiyonun içinde gerçekleştirilmesi gerektiğinde oldukça kullanışlıdır. Bu yapı, bir ana fonksiyonun içinde geçici olarak görev alacak, daha dar kapsamlı bir işlev oluşturmanıza olanak tanır. İç içe fonksiyonlar sayesinde, yalnızca ana fonksiyonun işlemesi gereken görevleri ayrı bir fonksiyon olarak tanımlayabilirsiniz. Bu da yalnızca o fonksiyonun içinde çalıştırılabilen bir yardımcı işlev oluşturarak, kodun daha düzenli bir hale gelmesini sağlar.

```python
def yemek_tarifi(yemek_adi):
    print(f"{yemek_adi} tarifi başlıyor...")

    # İç fonksiyon tanımlanır
    def malzemeleri_hazirla():
        print("Malzemeler hazırlanıyor...")

    # İç fonksiyon çağrılır
    malzemeleri_hazirla()
    print(f"{yemek_adi} hazırlanıyor...")
```

```python
# Dış fonksiyonu çağırdığımızda iç fonksiyon da çalışır
yemek_tarifi("Karnıyarık")
# Çıktı:
Karnıyarık tarifi başlıyor...
Malzemeler hazırlanıyor...
Karnıyarık hazırlanıyor...
```

çıktısı

Bu örnekte **yemek_tarifi** ana fonksiyonu, yemek tarifinin genel işleyişini gösterirken **malzemeleri_hazirla** ise yalnızca tarif içinde çağrılabilecek şekilde yapılandırılmış bir iç fonksiyondur. **malzemeleri_hazirla** fonksiyonu dışarıdan erişilemez.

İç içe fonksiyonlar, ana fonksiyon dışında bir yerde kullanılmasını istemediğiniz işlemler için ideal bir çözümdür ve aynı zamanda dış fonksiyonun değişkenlerine ve parametrelerine erişim sağladığından, dış fonksiyonun kapsamına özgü işlemler yapmanıza imkan tanır.

Bir iç-içe fonksiyon örneği

Hemen hemen herkesin bildiği Kelvin, Celsius ve Fahrenheit sıcaklık ölçüleri arasındaki dönüşüm formülleri düşünelim. Bu sıcaklık dereceleri belirli bağıntılarla birbirine dönüştürülebilmektedir. Bu bağıntılar;

```
celsius = kelvin - 273
fahrenheit = 1.8 * celcius + 32
```

olarak verilmektedir. Şimdi Kelvin cinsinden verilen bir sıcaklığı Fahrenheit'e çevirecek bir fonksiyonu oluşturalım. Yukarıdaki bağıntıları dikkate aldığımızda görüleceği üzere Kelvin'den Fahrenheit'e direkt bir çevirim söz konusu değildir. İlk adımda Kelvin Celsius'a ardından Celsius Fahrenheit'e çevrilme işlemlerinin yapılması gerekmektedir.

```
# ana fonksiyon
def kelvin_fahrenheit(sicaklik):

    # iç fonksiyon
    def kelvin_celsius(kelvin_sicaklik):
        celsiuc_sicaklik = kelvin_sicaklik - 273
        return celsiuc_sicaklik

    # İç fonksiyon çalıştırılır
    fahrenheit = 1.8 * kelvin_celsius(sicaklik) + 32
    return fahrenheit
```

Bu fonksiyonu 303 Kelvin derecesi için "d = kelvin_fahrenheit(303)" şeklinde test edersek , "d = 86.0" fahrenheit derecesi değerini elde ederiz. Burada "kelvin_celsius" fonksiyonu bir iç fonksiyondur ve ana fonksiyon "kelvin_fahrenheit" dışında başka yerden çağrılması mümkün değildir.

Üreteç Fonksiyonları

Python'da **üreteç (generator) fonksiyonlar**, bellek dostu ve verimli bir şekilde bir dizi değer üretmek için kullanılan özel bir fonksiyon türüdür. Üreteçler, bir dizi değeri tek seferde bellekte tutmak yerine, her bir değeri ihtiyaç duyulduğunda üretir. Bu, özellikle büyük veri kümeleriyle çalışırken bellek kullanımını optimize etmek için çok faydalıdır.

Üreteç Fonksiyonlarının Temel Özellikleri:

- **yield Kullanımı:** Üreteç fonksiyonları, return yerine **yield** ifadesini kullanır. yield, fonksiyonun durumunu korur ve bir sonraki çağrıldığında kaldığı yerden devam eder.
- **Durumunu Korur:** Üreteçler, değerleri ihtiyaç duyuldukça üretir. Bu, tüm değerlerin önceden hesaplanmasına gerek olmadığı anlamına gelir. Bu sayede sonsuz diziler veya büyük veri akışları ile çalışmak mümkündür.
- **Tekrar Kullanımı:** Bir üreteç fonksiyon bir kez tüketildiğinde sıfırlanmaz. Yeni bir örnek oluşturmak gerekir.
- **Bellek Verimliliği:** Üreteçler veriyi tek seferde üretip tükettiği için büyük veri kümeleriyle çalışırken bellekte gereksiz yer kaplamaz. Tüm veriyi bir liste içinde saklamak yerine, ihtiyaç duyuldukça bir sonraki öğeyi hesaplar.

Üreteç Fonksiyonu Örneği

```
def sayi_uret(baslangic, bitis):
    while baslangic < bitis:
        yield baslangic
        baslangic += 1

# Üreteç kullanımı
uret = sayi_uret(1, 5)

for sayi in uret:
    print(sayi)
```

```
1
2
3
4
```

Bu örnekte, sayi_uret fonksiyonu bir üreteçtir. yield ifadesi, fonksiyonun her çağrıldığında bir sonraki sayıyı üretir. for döngüsü ile üreteç üzerinde gezinerek her bir değeri yazdırıyoruz.

Üreteç fonksiyon örnekleri

Üreteçler, özellikle büyük veri setleriyle çalışırken bellek verimliliği sağlar. Örneğin, bir dosyayı satır satır okumak için bir üreteç kullanmak bellek verimliliği bakımından daha uygundur.

```
def dosya_okuyucu(dosya_adi):
    with open(dosya_adi, 'r') as dosya:
        for satir in dosya:
            yield satir

# Üreteci kullanma
dosya_uretec = dosya_okuyucu('buyuk_dosya.txt')

for satir in dosya_uretec:
    print(satir)
```

Bu örnekte, dosya_okuyucu fonksiyonu dosyayı satır satır okur ve her satırı bir üreteç olarak döndürür. Bu, dosyanın tamamını belleğe yüklemek yerine, sadece bir satırı bellekte tutar.

Sonsuz Üreteç Örneği

Üreteçler, sonsuz döngülerle birlikte kullanıldığında da oldukça kullanışlıdır. Örneğin, sonsuz bir sayı dizisi üreten bir üreteç şu şekilde tanımlanabilir.

```
def sonsuz_sayilar():
    sayi = 0
    while True:
        yield sayi
        sayi += 1

# Üreteci kullanma
sonsuz_uretec = sonsuz_sayilar()

print(next(sonsuz_uretec))  # Çıktı: 0
print(next(sonsuz_uretec))  # Çıktı: 1
print(next(sonsuz_uretec))  # Çıktı: 2
# Bu şekilde devam eder...
```

Üreteç İfadeleri (Generator Expressions)

Daha önce döngüler konusu anlatılırken satır içi döngüler kavramından bahsedilmişti. Tek satır döngüler kullanılarak liste veriler oluşturulabilmektedir. Şimdi tekrar hatırlatmak amacıyla bir listedeki sayıların küplerini hesaplayan satır içi bir döngüyü göz önüne alalım. Bu işlemi yapan kod şöyle olacaktır.

```
sayilar = [2, 5, 6, 8, 10, 3]
sayi_kup = [x ** 3 for x in sayilar]
print(sayi_kup)
[8, 125, 216, 512, 1000, 27]
```

Burada kullanılan liste oldukça küçüktür yani az elemandan oluşmaktadır. Eğer eleman sayı çok fazla ise ya da binlerce elemandan oluşan bir liste söz konuş olduğunda bu işlem oldukça uzun sürecektir ve üretilen sonucun hafızada tutulma problemleri oluşacaktır. Bu sorunu aşmanın en kolay yolu üreteç ifadeleri (generator Expressions) kullanmaktır. Bir üreteç ifadesi geri dönen bir liste veri tipi oluşturmaz, onun yerine sayılabilir bir obje yani nesne üretir. Bir üreteç ifadesini yazım düzeni satır içi liste döngülerle aynıdır. Ancak liste döngüsündeki köşeli parantez **"[]"** yerine, tuple verilerinde olduğu gibi normal parantezler **"()"** kullanılır. Şöyle ki;

(işlem for eleman in dizi)

Şimdi yukarıdaki küp alma işlemini üreteç fonksiyonu kullanarak tekrar düzenlediğimizde elde edilen sonuç bir nesne olması sebebiyle terminale herhangi bir sonuç yazılmamaktadır.

```
sayilar = [2, 5, 6, 8, 10, 3]
sayi_kup = (x ** 3 for x in sayilar)
print(sayi_kup)
<generator object <genexpr> at 0x0000012D3B0182B0>
```

Üreteç ifadeleri ile işlemler

Bir üreteç fonksiyonu ile elde edilen bir nesne sayılabilir elemanlara sahiptir. Nesne olması itibariyle içindeki elemanlara erişmek için ya "**list**" fonksiyonu ya da "**for-in**" döngüsü ya da "**next**" fonksiyonu kullanılarak işlem yapılması gerekmektedir.

list fonksiyon ile

```
sayilar = [2, 5, 6, 8, 10, 3]
sayi_kup = (x ** 3 for x in sayilar)
print(list(sayi_kup))
[8, 125, 216, 512, 1000, 27]
```

next fonksiyon ile

```
sayilar = [2, 5, 6, 8, 10, 3]
sayi_kup = (x ** 3 for x in sayilar)
print(next(sayi_kup))
8
print(next(sayi_kup))
125
print(next(sayi_kup))
216
```

for-in döngüsü ile

```
sayilar = [2, 5, 6, 8, 10, 3]
sayi_kup = (x ** 3 for x in sayilar)
for x in sayi_kup: print(x)


8
125
216
512
1000
27
```

Üreteç fonksiyonları için bilinmesi gereken en önemli özellik, fonksiyonlar direkt olarak bir sonuç üretmiyorlar, sadece üzerinde işlem yapılabilecek bir nesne üretmektedir. Bunun en büyük faydası büyük hacimli veriler üzerinde daha kolaylıkla işlem yapabilmesini sağlamasıdır.

Modül-Paket Kavramı ve Faydaları

Bir program hazırlarken o programa özgü bir çok fonksiyon oluşturabilmektedir. Ancak başka bir program hazırlarken daha önce hazırlamış olduğumuz fonksiyonların tekrar kullanılması gerekmektedir. Bu durumda önceki programdan bu fonksiyonlar kopyalanması lazımdır. Fakat her defasında fonksiyonları böyle aktarmak, özellikle çok sayıda fonksiyon bulunuyorsa zorluklar yaratacaktır. İşte böyle sıklıkla kullanılan fonksiyonların bir araya getirilerek **".py"** uzanımlı dosyalarda tutulmasına Python'da **"modül"** adı verilmektedir.

Modüller ve içerdiği fonksiyonlar, değişkenler veya sınıflar başka Python dosyalarına kolayca dahil edilebilir. Python'da **"import"** anahtar kelimesi ile bir modül çağrılarak, modüldeki fonksiyonlar ve veriler kullanılabilir hale getirilir. Örneğin, Python'un yerleşik **"math"** modülü, matematiksel işlemleri kolaylaştırmak için birçok hazır fonksiyon içerir.

```
import math
sayi = math.factorial(4)

print(sayı)   # çıktı: 24
```

Görüldüğü üzere modül **"import"** anahtarı ile programa aktarıldı ve modül içindeki fonksiyonlar modül isminin yanına **"."** nokta konularak çağrılmaktadır.

Benzer şekilde Python için geliştirilmiş ve bütün Python kullanıcıların kullanımına sunulmuş çok sayıda modül bulunmaktadır. Böyle ortak kullanıma sunulmuş modüllere **"paket-package"** adı verilir. Python'da numpy, pandas, matplotlib, scikit-learn, scipy gibi birçok paket bulunmaktadır. Bir paketin kullanılabilmesi içim Python çalışma klasörlerine kurulması gereklidir. Paket kurulum işlemi **"pip"** komutu ile yapılır ve yazım düzeni şöyledir:

```
pip install <paket ismi>
```

Modül kullanım şekilleri

Python'da modüller, **import** anahtar kelimesi ile içe aktarılır ve içindeki fonksiyonlar ve değişkenler kullanılır. Modülleri bir programa aktarıp kullanmanın birkaç yolu vardır:

1. import anahtar kelimesi ile modül içe aktarma

Bu yöntem, bir modülü programınıza dahil etmek ve modüldeki işlevleri kullanabilmek için kullanılır. Bu sayede, modülde tanımlanmış olan fonksiyonlar, sınıflar ve değişkenler programınıza dahil edilir ve bunları kendi kodunuzda doğrudan kullanabilirsiniz.

```
import math  # Python'un yerleşik bir modülü

# Modülün bir fonksiyonunu kullanma
sonuc = math.sqrt(16)

print(sonuc)
4.0
```

2. Modülden belirli bir fonksiyonu veya değişkeni içe aktarma

Bu yöntem, tüm modülü içeri aktarmak yerine, yalnızca ihtiyaç duyduğunuz belirli öğeyi almanıza olanak tanır. Bu, kodunuzu daha hafif ve okunabilir hale getirir, çünkü yalnızca gerekli olan işlevler veya değişkenler üzerinde çalışırsınız. Ancak eğer içe aktarılan fonksiyon veya değişken, mevcut programdaki bir isimle çakışırsa, bu çakışma istenmeyen sonuçlara yol açabilir. Bu nedenle isim çakışmalarına dikkat edilmesi gerekir.

```
from math import sqrt

# Direkt olarak fonksiyonu kullanma
sonuc = sqrt(25)

print(sonuc)
5.0
```

Modül Kullanım Şekilleri

3. Modülü kısaltma ile içe aktarma

Bir modülü daha kısa ve kullanışlı bir adla içe aktarmak için **"as"** anahtar kelimesi kullanılır. Bu yöntem, özellikle büyük veya karmaşık modüllerle çalışırken kodu daha temiz ve okunabilir hale getirir. **import ... as ...** yapısı sayesinde modülün orijinal uzun adını her seferinde yazmak yerine, daha kısa bir takma ad (alias) ile çalışabilirsiniz. Bu yaklaşım, kodun daha kompakt olmasını sağlar ve tekrar eden uzun isim yazımlarını önler.

```
import pandas as pd  # pandas modülünü "pd" olarak kısalttık

# DataFrame oluşturma
veri = {"isim": ["Ali", "Ayşe"], "yas": [25, 22]}
df = pd.DataFrame(veri)
print(df)
```

4. Tüm içeriği içe aktarma (TAVSİYE EDİLMEZ)

Bu yöntem, bir modüldeki tüm fonksiyonları, değişkenleri ve sınıfları doğrudan kullanılabilir hale getirir, böylece her bir öğeyi tek tek içe aktarmak zorunda kalmazsınız. Kodunuzu daha kısa hale getirmesi açısından avantajlı görünse de, bu yöntem kodda isim çakışmaları yaratma potansiyeline sahiptir. Çünkü hangi fonksiyonun veya değişkenin hangi modülden geldiğini belirsiz hale gelebilmektedir.

```
from math import *

# Tüm fonksiyonları içe aktardık
sonuc = cos(0)  # cos fonksiyonunu direkt kullanabiliyoruz
print(sonuc)
1.0
```

Modüllerde yaygın metotlar ve kullanımlar

Python'da modüllerle çalışırken bazı yaygın metotlar ve fonksiyonlar, modüllerin içeriğini keşfetmek ve kullanmak için oldukça faydalıdır. Bu metotlar, modüllerin sunduğu fonksiyonlar, sınıflar ve değişkenler hakkında bilgi almanızı sağlar ve kod geliştirme sürecini kolaylaştırır.

1. dir() fonksiyonu

Bir modül içinde tanımlanmış olan fonksiyonların, değişkenlerin ve diğer ögelerin listesini verir. Modülün içeriğini keşfetmek için kullanılır.

```
import math
liste = dir(math) # math modülündeki tüm fonksiyonlar ve sabitlerin listesini döndürür
print(liste)

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e',
'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

2. help() fonksiyonu

Bir modül veya bileşenin ne işe yaradığını ve nasıl kullanıldığını anlamak için yararlıdır. Özellikle modüllerdeki fonksiyonların ne yaptığını ve hangi parametreleri kabul ettiğini görmek için kullanılır.

modüldeki bir fonksiyon için

```
import math

help(math.sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(x, /)

    Return the square root of x.
```

Modüllerde yaygın metotlar ve kullanımlar

Eğer **help()** fonksiyonu bir modül ya da paket için kullanılırsa paket hakkında kısa bir bilgiden sonra pakettin içindeki tüm fonksiyonlara ilişkin bilgiler de verilecektir.

modülün kendisi için

```
import math

print(help(math))
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical
    functions defined by the C standard.
```

```
FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.
        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.
        The result is between -pi/2 and pi/2.

    ...
```

3. __name__ özel değişkeni

Bir Python dosyasının ana program olarak mı çalıştırıldığını yoksa başka bir modül tarafından mı içe aktarıldığını kontrol eder.

```
# mymodule.py
if __name__ == "__main__":
    print("Bu dosya doğrudan çalıştırılıyor.")
else:
    print("Bu dosya bir modül olarak içe aktarılıyor.")
```

4. __all__ özel değişkeni

Bir modülden from ... import * ile hangi öğelerin içe aktarılabileceğini sınırlamak için kullanılır. Sadece belirtilen öğelerin kullanılabilir olmasını sağlar ve gizli işlevlerin dışarıdan görünmesini engellemesi açısından faydalıdır.

```
# mymodule.py
__all__ == ['fonksiyon1', 'fonksiyon2']
def fonksiyon1(): pass
def fonksiyon2(): pass
def gizli_fonksiyon(): pass
```

Özel modül oluşturma ve kullanma

Bir veya birden fazla Python dosyasında tanımlanmış fonksiyonlar, değişkenler veya sınıfların başka projelerde kullanılabilir hale getirilmesi işlemi özel modül oluşturma anlamına gelir. İstenilen kodların bir dosya içine konularak **".py"** uzantıyla kaydedilmesi yeterlidir. Örneğin, matematik.py adlı bir dosya oluşturabilir ve bu dosyada çeşitli matematiksel fonksiyonlar tanımlayabilirsiniz. Bu dosya, bir modül olarak işlev görür ve istediğiniz zaman başka bir Python dosyasında içe aktarılabilir.

1. Modül oluşturma

Belirli fonksiyonların ve değişkenlerin oluşturularak bir dosyada kaydedilme işlemidir. Örneğin **hesaplama.py** isimli bir dosyada aşağıdaki fonksiyonların ve değişkenin olduğunu varsayalım.

```
# hesaplama.py
sabit = 12.765
def topla(a, b): return a + b
def carp(a, b): return a * b
```

2. Kendi modülünüzü içe aktarma

Daha önceden hazırlanmış ve özel bir dosyaya kaydedilmiş fonksiyonların ve değişkenlerin başka programdan çağrılma işlemdir. Şimdi hesaplama.py modülü/paketini başka program içine aktaralım.

```
# anaprogram.py
import hesaplama as hsp

sonuc = hsp.topla(5, 9)
print(sonuc)
14
```

```
katkı = 2 * hsp.sabit + 13.0
scarp = hsp.carp(12, 3)

print(katkı, scarp)
38.53, 36
```

Yerleşik ve üçüncü taraf modüller

Python'un kurulumu ile gelen ve herhangi bir ek kurulum gerektirmeden doğrudan kullanılabilen modüller yerleşik modül olarak isimlendirilir. Üçüncü taraf modüller ise genellikle başka kod geliştiricilerin oluşturdukları ve Python Paket İndeksi (PyPI) gibi kaynaklardan indirilebilen nodüllerdir.

Yerleşik modüller

Genellikle yaygın ve temel işlevleri gerçekleştirmek için kullanılan ve Python'un güçlü ve esnek bir dil olmasını sağlayan temel araçlardır. Kod geliştiricilerin birçok yaygın problemi çözmesine katkı sağlayan hazır fonksiyonlar sunmaktadır. Örneğin, **math** modülü matematiksel işlemler için, **datetime** modülü tarih ve saat işlemleri için, **os** modülü işletim sistemi ile ilgili işlemler için kullanılır.

```
import math
# Pi sayısı
print(math.pi)
# Faktöriyel hesabı
print(math.factorial(5))
```

```
from datetime import datetime
su_an = datetime.now()
print(su_an.strftime("%Y-%m-%d %H:%M:%S"))
```

```
import os
print(os.getcwd())
# Geçerli çalışma dizinini alır
```

Üçüncü taraf modüller

Python'un standart kitaplığının dışında kalan, ancak geliştiriciler tarafından ek işlevler sağlamak amacıyla oluşturulmuş modüllerdir. Bu modüller **pip** komutu benzer paket yükleyici komutlar ile kurulurlar. Örneğin, numpy bilimsel hesaplamalar için, pandas veri analizi için, requests ise web istekleri yapmak için yaygın olarak kullanılan üçüncü taraf modüllerdir.

```
import numpy as np
import pandas as pd
```

```
import requests
response = requests.get('https://api.example.com/data')
print(response.json())
```

Çalışma soruları

Soru 1. Verilen bir kelimedeki harfleri tersten yazan bir fonksiyon yazın. Aynısını bir liste için yapın.

Soru 2. Verilen iki kelimeyi büyük harfe çevirerek birleştiren bir fonksiyon yazın. Ancak büyük harfe çevirme işlemi bir iç fonksiyon olarak ayarlasın.

Soru 3. 1 ila 100 arasındaki çift sayıları bulan bir üreteç fonksiyonu yazın. Oluşan nesneni ilk 3 elemanın next fonksiyonu ile yazdırın.