Yorumlayıcı (Interpreter)	Derleyici (Compiler)
Herhangi bir hata olana kadar programı çalıştırır. İlk hata gördüğü yerde durur. Bu nedenle hata ayıklama kolaydır.	Hatayı bütün kodu tamamladıktan sonra gösterir. Bu nedenle hata ayıklama nispeten zordur.
Kaynak kodu analiz etmekle zaman harcamaz. Ancak genel yürütme süresi daha yavaştır.	Kaynak kodun analizi için büyük zaman harcar. Ancak genel yürütme süresi daha hızlıdır.
Programı satır satır işler.	Tüm programı tarar ve bir bütün olarak makine koduna çevirir.
Python, Java	C, C++

Imperative (Emirsel)	Declarative (Bildirimsel)
Imperative nasıl yapacağını anlattığın programlama şeklidir.	Declarative ne yapacağını anlattığın programlama şekldir.
Imperative yöntemde açıklayıcı emirlerle işlemi detaylı bir şekilde gerçekleştiririz.	Declarative yöntemde sadece yapacağınız şeyi anlatırsınız.

PYTHON 2	PYTHON 3
`print "Merhaba Dünya"`	`print ("Merhaba Dünya")`
5/2=2	5/2=2,5

Dinamik Tipleme	Statik Tipleme
Çalıştırma esnasında veri tipi değişir.	Çalıştırma esnasında “int a” veri tipi sonuna kadar “int a” veri tipi olarak kalır.
Javascript	C

DİLLERİN YÖNLERİ	
Syntax	
"hi"5	sözdizimsel olarak geçerli değil
3.2*5	sözdizimsel olarak geçerli
Static Semantics (Anlambilim)	
"I are an engineer"	sözdizimsel olarak geçerli ancak statik anlamsal hata
1.9*0.5	sözdizimsel olarak geçerli
22/"hi"	statik anlam hatası
Dynamic Semantics (Anlambilim)	
Konuşma dilinde farklı anlamlar çıkabilir ancak esasında sadece tek bir doğru anlamı vardır. I saw a man on a hill with a telescope.	
Teleskopla bir tepede bir adam gördüm.	DOĞRU
Tepede teleskopla bir adam gördüm.	YANLIŞ
Teleskoplu bir tepede adam gördüm.	YANLIŞ
Anlamsal hata yok ama programcının amaçladığından farklı anlamlar çıkabilir bunlar; – Program çöküyor, çalışmayı durduruyor – Program sonsuza kadar çalışır. – Program yanıt veriyor ancak beklenenden farklı sonuç çıkabilir.	

PYTHON'A GİRİŞ

Python ile yazılan kodlar derleme işlemine tabi tutulmaz.

Python dilinde veri türleri **dinamik**tir. Python ile program yazarken kullanacağınız değişkenlerin tiplerini belirtmenize gerek yoktur. Üzerine atanan değere göre Python değişkenin türünü otomatik olarak algılar. **a=3** soldaki tanımlamada, a'nın bir tamsayı olduğunu Python otomatik olarak algılar.

Nesneler **scalar** yani alt operatörlere bölünmez (int, float, bool) ve **non-scalar** yani alt operatörlere bölünür (strings, tuples, list, arrays, dictionaries) olmak üzere ikiye ayrılır.
int() alt alanlara ayrılamazken, string ayrılır. Yani string'in içinde gezebiliyoruz ancak int'ın içinde gezemiyoruz, istediğimiz değeri seçip alamıyoruz gibi.

Uygulama içindeki hatalara **bug** denilir. Hata ayıklama işlemine de **debug** denilir.

Tek satırlık **yorum satırı** eklemek için # işareti kullanılır. Birden fazla satırdan oluşan yorum satırlarını üç adet çift tırnak (""" """) arasına almalıyız.

Bilgisayarın, kodunuzu her seferinde bir satır olacak şekilde sırayla çalıştırmasına **sequence (sıralama)** denir. Örneğin 1. satırda başlayacak, ardından 2. satırı, ardından 3. satırı yürütecek ve programınızın son satırına ulaşana kadar devam edecektir.

Komut İstemi'ne "py" yazdığımızda karşımıza çıkan ekrana "**Etkileşimli Kabuk**" diyoruz. Ekileşimli Kabuk'ta Python komutlarını çalıştırırız. Çıkmak için "**exit()** ya da **CTRL+Z**" komutlarını kullanabiliriz. Etkileşimli kabuk ekranındaki >>> işareti, o anda etkileşimli kabuk ortamında olduğumuzu ve Python'un da bizden komut almaya hazır olduğunu gösteriyor. Etkileşimli kabukta komutları ve tanımlamaları sadece tek bir satıra yazıp çalıştırabiliriz ancak **Script Mod**'da (Herhangi bir programlama dili ile oluşturulmuş hazır kod bloklarıdır.) komutları ve tanımlamaları ard arda yazabiliriz. Python Komut İstemi'nin diğer adı **prompt**'tur.

Değişken Tanımlama

Aynı anda birden fazla değişkeni tek satırda tanımlayabiliriz.

```
a, b, c = 3, 5, 7
print(a,b,c)
```

Veri Tipleri

Python'da verilerin tipini **type()** adlı bir fonksiyon yardımıyla sorgulayabiliriz.

`n = 5.5` `type(n)`	`veri = input("Veri:")` `print(type(veri))`	`type(True)`	`type(None)`
<class 'float'>	<class 'str'> !!DİKKAT!	<class 'bool'>	<class 'Nonetype'>

OPERATÖRLER

İşlem	Sembol	Örnek	Sonuç
Toplama	+	3+5	8
Çıkarma	-	7-2	5
Çarpma	*	3*2	6
Ondalıklı Bölme	/	3/2	1.5
Tam Bölme	//	5//3	1
Üs Alma	**	3**2	9
Kalan Bulma (mod)	%	9%2	1

Aşağıdaki tablo için a değişkeninin mevcut değerinin 8 olduğunu varsayarsak:

Operatör	Anlamı	Kısa Kullanımı	Uzun Kullanımı	a'nın Yeni Değeri
=	Atama yapar	a=8	a=8	8
+=	Arttır ve ata	a+=2	a=a+2	10
-=	Azalt ve ata	a-=2	a=a-2	6
*=	Çarp ve ata	a*=2	a=a*2	16
/=	Ondalıklı böl ve ata	a/=2	a=a*2	4.0
//=	Tam böl ve ata	a//=2	a=a//2	4
%=	Kalan bul ve ata	a%=2	a=a%2	0
**=	Üs al ve ata	a**=2	a=a**2	64

Aşağıdaki tablo için a değişkeninin mevcut değerinin 8, b değişkeninse 10 olduğunu varsayarsak:

Operatör	Açıklama	İşlem	Sonuç
==	İki operandın değeri birbirine eşit midir?	a==b	False
!=	İki operandın değeri birbirinden farklı mıdır?	a!=b	True
<	Soldaki operand sağdakinden küçük müdür?	a<b	True
>	Soldaki operand sağdakinden büyük müdür?	a>b	False
<=	Soldaki operand sağdakinden küçük ya da eşit midir?	a<=b	True
>=	Soldaki operand sağdakinden büyük ya da eşit midir?	a>=b	False
and	ve operatörü	(8<10) and (6>5)	True
		(8>5) and (8>10) and (6>5)	False
or	veya operatörü	(5==5) or (6==5)	True
		(8>5) or (8>10) or (6>5)	True
not	değil operatörü	not (5==5)	False

Mantıksal Veri Türü (Boolean)

Mantıksal bir değişkenin alabileceği iki değerden biri True diğeri ise False'tur. **Python dilinde 0 ya da boş veriler mantıksal olarak False değerine karşılık gelir.**

`print(bool(1))`	True	`print(bool(-5))`	True
`print(bool("a"))`	True	`print(bool(0))ve print(bool(0.0))`	False
`print(bool(""))`	False	`print(bool(" "))`	True

Üyelik (Identity) Operatörü

Herhangi bir elemanın bir koleksiyona ait olup olmama durumunu kontrol eden operatörlerdir.

İki adet aitlik operatörü vardır: **in** ve **not in**.

```
print("a" in "merhaba")      True
```

Burada "a" karakteri "merhaba" karakter dizisi içinde var mı durumu kontrol ediliyor ve merhaba ifadesinde a karakteri bulunduğu için ekrana True yazar.

```
print("abc" in "merhaba")    False
```

Yukarıdaki örnekte ise merhaba ifadesi içerisinde abc ifadesi bulunmadığı için (bir bütün olarak, ayrı ayrı karakterler olarak değil) False sonucu elde edilir.

Kimlik (Membership) Operatörü

İki nesnenin **bellek adreslerini** yani **id**'lerini karşılaştıran operatörlerdir.

İki adet kimlik operatörü vardır: **is** ve **is not**.

```
a=20    b=30    print(a is b)     False
```

is operatörü id(a) ve id(b) aynı sonucu üretiyorsa yani bellek adresleri aynı ise True, değilse False sonuç verir. Burada a ve b değişkenlerinin adreslerini ayrı ayrı ekrana yazdırdırarak farklı olduklarını görebilirsiniz. Bu durumda "a is b"ifadesi False sonuç üretir.

```
print(a is not b)                 True
```

Bu durumda a ve b değişkenlerinin bellek adresleri yani id'leri aynı olmadığı için True'yu elde ederiz.

Operatör Önceliği

1- Parantezler en yüksek önceliğe sahiptir.	2- Üs alma operatörü bir sonraki en yüksek önceliğe sahiptir
3- Çarpma ve bölme, toplama ve çıkarma operatörlerine göre daha önceliklidir.	4- Aynı önceliğe sahip operatörler soldan sağa doğru değerlendirilir (Üs alma operatörü hariç).

Expressions ve Statements

Statement, programlama dili tarafından işlenmesi gereken, en küçük birim olarak tanımlanıyor.

Expression ise bir ya da daha fazla değişkenden, operatörden (örneğin +) veya fonksiyondan meydana gelen ve programlama dili tarafından yorumlandığında bir değer döndüren ifadelerdir.

Örnek verecek olursak; "int a;" bir **statement** iken "1 + 1" ifadesi bir **expression**'dır. Çünkü ilk ifade bir değer döndürmezken ikincisi 2 değerini döndürmektedir.

Kaçış Dizileri (\)

Buradaki hatanın sebebi de karakter dizisini başlatıp bitiren tırnak işaretleriyle, Ali'nin sözünü aktarmamızı sağlayan tırnak işaretlerinin birbirine karışmasıdır. Bu hatayı karakter dizisini önüne \ işaretini yerleştirerek Python'un bu işaretlere takılmasını önlüyoruz.

`print("Ali, "Gidiyorum." dedi")`	`print("Ali, \"Gidiyorum.\" dedi")`
SyntaxError: invalid syntax	Ali, "Gidiyorum." dedi

Karakter Dizileri

Ekrana yazdırma **print()** fonksiyonu ile yapılır. print(" ") = print("" "") = print(""" """) aynı işlevi görmektedir.

Python programlama dilinde print(" ") fonksiyonundaki tırnak içinde gösterilen bütün değerlere **karakter dizisi** yani **string** adı verilir.

```
d = input("Sayı: ")
print(d*3)
```

-veri girişi 5'tir-

555

555 yazmasının sebebi kullanıcıdan aldığımız verinin string tipinde olmasıdır.

```
e = int(input("Sayı:"))
print(e*3)
```

-veri girişi 5'tir-

15

Bunu çözmek için kullanıcıdan aldığımız verinin tipini başta belirtmemiz gerekir.

Kullanıcıdan okunacak veriler **input()** fonksiyonu ile alınır. input'tan alınan bütün veriler **string** tipindedir, **casting** yaparak int'e ya da float'a çevirebiliriz.

Tam sayıyı ondalıklı sayıya çevirmek için **float()** komutunu, ondalıklı sayıyı tam sayıya çevirmek için **int()** komutunu son olarak sayıyı stringe çevirmek için **str()** komutunu kullanırız.

```
a = "43" ya da a = 43
a = float(a)
print(a)          43.0
```

```
a = 54.12
a = int(a)
print(a)          54
```

```
a = 31.60
b = str(a)
print(b)          31.6
print(len(b))     5
```

```
print("2+2")    2+2
print(2.0+2)    4.0
```

Sol tarafta olduğu gibi **float + int = float** tipinde sonucumuzu alırız.

```
urun1, urun2 = 50, 71                  # int tipinde
toplam = urun1 + urun2                 # int tipinde
print("Ürün toplamı: ",toplam)             Ürün toplamı: 121
print("Ürün toplamı: ",urun1 + urun2)      Ürün toplamı: 121
```

```
a = input("Bir sayı giriniz:")
print("Kullanıcının Değeri: ",a)
```

---71 girdiğimizi varsayalım---
Kullanıcının Değeri: 71

```
a = 5
b = 3
print(a, "ile", b, "'ü çarparsak", a * b, "buluruz.")
```

5 ile 3 çarparsak 15 buluruz.

Parametreler (Argümanlar)

print() fonksiyonunun, parantezleri içine yazdığımız her bir değere **argüman** deriz. Fonksiyonu birden fazla argüman ile de çağırabiliriz örneğin;

```
print(34,"Python", "Java", "C++")                    34 Python Java C++
```

print() fonksiyonunun parantezleri içine yazdığımız argümanların her birini birbirinden birer virgül ile ayırıyoruz. Eğer virgül yerleştirmezsek, print() bu argümanları birbiriyle birleştirecektir örneğin;

`print("bin"+"beş"+"yüz")`	`print("bin""beş""yüz")`	`a = "Elif"`
		`b = "Polat"`
		`c = "Kız Kulesi"`
		`print(a+b+c)`

binbeşyüz	binbeşyüz	ElifPolatKız Kulesi

Argümanlar arasında sayılar varsa arasına virgül koymayı kesinlikle ihmal etmemeliyiz.

```
print("Yaşınız:", 40)    Doğru: Yaşınız: 40
print("Yaşınız:" 40)     Yanlış: SyntaxError: invalid syntax.
```

NOT: String ile string'i toplayabiliriz ancak string ile int'i toplayamayız ayrıca string ile int'i çarpabiliriz.

`print("Cim"+"Bom"*3)`	`print("25" + "50")`
CimBomBomBom	2550

```
print("Sonuç: " + 10)
TypeError: can only concatenate str (not "int") to str
```

1- sep = " " Parametresi

Python, **sep=" "** argümanının **öntanımlı değerini boşluk** karakteri olarak belirlemiştir.

```
print("Polis", "Akademisi")                  Polis Akademisi
```

Ancak eğer biz istersek argümanların arasına yerleştirilecek karakteri değiştirebiliriz.

```
print("www", "google", "com", sep=".")     www.google.com
```

Bu argümanın değerini boş bir karakter dizisi yaparak öteki argümanlar arasındaki boşlukları tamamen kaldırabiliriz de.

```
print("afyon", "kara", "hisar", sep="")   afyonkarahisar
```

2- end = " " Parametresi

end=" " argümanın görevi argümanların sonuna hangi karakterin geleceğini belirlemektir.

```
print("Merhaba", "Zalim", "Dünya", end="**")    Merhaba Zalim Dünya**
```

3- file = Parametresi

Bu parametrenin görevi, print() fonksiyonunun içindekilerin nereye yazılacağını belirtmektir.

Bu parametrenin öntanımlı değeri **sys.stdout**'tur **(standart çıktı konumu)**.

Eğer yazdığınız bir programı komut satırında çalıştırıyorsanız, üretilen çıktılar komut satırında görünür, etkileşimli kabuktaysanız etkileşimli kabukta görünür. Dolayısıyla Python'ın **standart çıktı konum**u etkileşimli kabuk veya komut satırıdır.

Eğer istersek print() fonksiyonunun, çıktılarını ekrana değilde, bir dosyaya yazdırmasını da sağlayabiliriz. Örneğin aşağıdaki örnekte print() fonksiyonunun içinde yer alanlar, deneme.txt adlı bir dosyaya çıktı verdi.

```
dosya = open("deneme.txt", "w")
print("Ben Python, Monty Python!", file=dosya)
dosya.close()
```

4- flush = " " Parametresi

Yazdığımız programda, dosyaya yazmak istediğiniz bilgilerin hiç bekletilmeden doğrudan dosyaya aktarılmasını istiyorsak bu parametreyi kullanabiliriz.

Bu parametrenin **True** ve **False** olmak üzere iki değeri vardır, **öntanımlı değeri False**'tur. Yani herhangi bir değer belirtmezsek Python bu parametrenin değerini False olarak kabul edecek ve bilgilerin dosyaya yazılması için dosyanın kapatılmasını bekleyecektir.

```
f = open("kişisel_bilgiler.txt", "w")
```

Dosyamızı oluşturduk. Şimdi bu dosyaya bazı bilgiler ekleyelim:

```
print("Merhaba Dünya!", file=f, flush=True)
```

Parametrelerin Dağıtılması

print() fonksiyonunda yer alan karakter dizilerini * işareti yardımıyla ögelerine ayırabiliriz.

```
print(*"KOMUTANLIĞI")
K O M U T A N L I Ğ I
print(*"Özel", *"Kuvvetler", "Komutanlığı")
Ö z e l K u v v e t l e r Komutanlığı

print(*"Özel ", "Kuvvetler ", "Komutanlığı", sep="-")
Ö-z-e-l-Kuvvetler-Komutanlığı
```

* işaretini integer ile birlikte kullanırsak hata alırız çünkü int alt alanlara ayrılan bir değer değil.

```
print(*12345)
TypeError: print() argument after * must be an iterable, not int
```

Eğer bu işareti, sayı değerli verilerle birlikte kullanmak istiyorsak o sayı değerli veriyi öncelikle karakter dizisine dönüştürmemiz gerekir. Bunun için o sayıyı tırnak içine almamız yeterli olacaktır.

```
print(*"12345")          1 2 3 4 5
```

MODÜLLER (KÜTÜPHANELER)

Python'da iki tür modül vardır. Bunlar **standart kütüphane modülleri** ve **harici modüllerdir**. Standart kütüphane modüllerinin özelliği, bunların Python programlama dilinin bir parçası olmasıdır. Yani bu tür modülleri kullanabilmek için herhangi bir ek yazılım kurmamıza gerek yoktur. Bu modülleri **import** komutunu kullanarak programlarımıza dâhil edebiliyoruz. Harici modüller ise Python programlama dilinin bir parçası değildir. O yüzden bu modülleri kullanabilmek için bunları öncelikle sistemimize kurmamız gerekir.

Eğer istersek birkaç farklı modülü tek bir import komutuyla da içe aktarabiliriz yani ekleyebiliriz.

```
import random, datetime, webbrowser
```

sys Modülü

sys.exit()	Programınızın işleyişini durdurur.	**sys.prefix**	Python'ın hangi dizine kurulduğunu gösterir.
sys.argv	Yazdığımız program çalıştırılırken kullanılan parametreleri bir liste halinde tutar.	**sys.ps1**	Etkileşimli kabuktaki '>>>' işaretini tutar.

format() Metotu

Bazı yerlerde bi stringin içinde daha önceden tanımlı string, float, int vs. değerleri yerleştirmek isteyebiliriz böyle durumlarda **.format()** metotunu kullanırız.

```
print("{:.2f} ve {} iyi bir ikilidir!".format(1.3412, "Elif"))
```

1.34 ve Elif iyi bir ikilidir!

Süslü parantezlerin içinde :.2f ifadesi ondalıklı kısmın ilk iki basamağını almamızı sağlar.
Yukarıdaki işlemi f'li karakter dizileriyle şu şekilde gerçekleştirebiliriz

```
print(f"{'Polat'} ve {'Elif'} iyi bir ikilidir!")
```

Polat ve Elif iyi bir ikilidir!

format() metotunu değişken atayarak kullanabilirsiniz.

```
ad, soyad = "Elif", "Tek"
print("Ad: {n}\nSoyad: {s}".format(n=ad, s=soyad))
```

Ad: Elif Soyad: Tek

İsterseniz küme parantezleri içine birer sayı yazarak karakter dizisi dışındaki değerlerin hangi sırayla kullanılacağını belirleyebilirsiniz. Örneğin;

```
print("{0} {1} ({1} {0})".format("Fırat", "Özgül"))
```

'Fırat Özgül (Özgül Fırat)'

```
sayi = 200/700
print("Sonuç:{n:10.3}".format(n=sayi))
```

Sonuç: 0.286

Sonuç yazısından itibaren 0.286'ya kadar 10 karakterlik yer kapladık (boşluklar da dahil), virgülden sonra da 3 basamak yazdık.

eval() Fonksiyonu

Kullanıcıdan gelen **karakter dizisi** şeklindeki veriyi **eval()** fonksiyonu yardımıyla değerlendirmeye tabi tutuyoruz. Yani **kullanıcının girdiği komutları işleme sokuyor**uz.

Örneğin, kullanıcı 46 / 2 gibi bir veri girdiyse, biz eval() fonksiyonu yardımıyla bu 46 / 2 komutunu işletiyoruz. Bu işlemin sonucunu da hesap adlı başka bir değişken içinde depoluyoruz.

```
veri = input("İşleminiz: ")
hesap = eval(veri)              - 23 - 3 girdiğimizi varsayalım -
print(hesap)                  20
```

eval() fonksiyonu bir karakter dizisi içindeki değişken tanımlama işlemini yerine getiremez.

```
eval("a = 45")
```

NOT: Bir modül içindeki fonksiyonları çağırırken **parantezleri** kullandığımıza ama **özellikleri** çağırırken parantez kullanmadığımıza dikkat edin. Modüller içindeki nitelikler, değişkenlere çok benzer. Değişkenleri nasıl parantezsiz kullanıyorsak özellikleri de aynı şekilde **parantezsiz** olarak kullanıyoruz.

```
import datetime
şuan = datetime.datetime.now()
gün = şuan.day
ay = şuan.month
print(f"{gün}/{ay}")              26/3
```

********** **********

********** **********

1- f'li Karakter Dizileri

Öncelikle karakter dizisinin en başına bir " f " harfi yerleştirdiğimize, değişkenleri ise süslü parantezler ({}) içinde gösterdiğimize dikkat edelim.

```
a = 5
b = 3
print(f"{a} * {b} = {a * b} 'dir ")
print(f"{5} * {3} = {5 * 3} 'dir.")
5 * 3 = 15 'dir.
```

```
ad = "Elif"
soyad = "Tek"
print(f"{ad:} - {soyad:}")

Elif - Tek
```

2- Değişken Değiştirme

İkinci örnekte, aynı satırda tanımladığımız değişkenleri birbiriyle değiştirdik.

```
birinci, ikinci = "CSS", "C++"
print(birinci)
print(ikinci)
```

CSS
C++

```
birinci, ikinci = "Python", "Java"
birinci, ikinci = ikinci, birinci
print(birinci)
print(ikinci)
```

Java
Python

KOŞULA BAĞLI DURUMLAR (SELECTION)

Tek şart varsa örneğin if (sayi==100) buna **one armed condition**, iki şart varsa örneğin if(sayi>100) ise ve bu değer true ise sayi = sayi+1 olsun eğer false ise (else) sayi = sayi -1 olsun buna da **two armed condition** denir.

if, **elif** ve **else**'den sonrakileri bir tab boşluk bırakarak yazmalıyız aksi takdirde aynı hizzada olurlarsa program bunu algılamaz.

```
sayı = 100
if sayı == 100:
    print("sayı 100'dür")
if sayı <= 150:
    print("sayı 150'den küçüktür")
if sayı > 50:
    print("sayı 50'den büyüktür")
if sayı <= 100:
    print("sayı 100'den küçük veya
100'e eşittir")
```

sayı 100'dür
sayı 150'den küçüktür
sayı 50'den büyüktür
sayı 100'den küçük veya 100'e eşittir

```
sayı = 100
if sayı == 100:
    print("sayı 100'dür")
elif sayı <= 150:
    print("sayı 150'den küçüktür")
elif sayı > 50:
    print("sayı 50'den büyüktür")
elif sayı <= 100:
    print("sayı 100'den küçüktür
veya 100'e eşittir")
```

sayı 100'dür.

Yukarıdaki örnekte gördüğünüz gibi programımızı elif deyimini kullanarak yazarsak Python, belirtilen koşulu karşılayan **ilk sonucu ekrana yazdıracak ve orada duracaktır.**

```
notunuz = input("Notunuz: ")
notunuz = int(notunuz)
if notunuz not in range(0, 101):
    print("Notunuz 0 ile 101 arasında olmalı")
    quit()
elif notunuz in (range(90, 100) or range(80, 90)):
    puan = "AA"
    print("Puanınız: ", puan)
elif notunuz in range(70, 80) and notunuz>0:
    puan = "BB"
    print("Puanınız: ", puan)
else:
    puan = "FF"
    print("Puanınız: ", puan)

- 75 girdiğimizi varsayalım -
Notunuz: 75
Puanınız:  BB
```

TEKRARLAMA YAPILARI (DÖNGÜLER) (ITERATION)

İki farklı tekrarlama yapısı bulunmaktadır.

1- Sayı Kontrollü (Counted / Count Controlled) (For)
- Belirli sayıdaki tekrarlar için kullanılır.
- Örneğin; 1'den 10'a kadar olan sayıları topla.

2- Şart Kontrollü (Uncounted / Condition Controlled) (While)
- Verilen şart sağlanana kadar tekrarlama işlemine devam edilir.
- Örneğin; Girilen sayı negatif olana kadar sayıları topla.

While Döngüsü (Un-Bounded Iteratıon / Sınırsız Yineleme) (Mantığı Şart Kontrollüdür)

```
while koşul:
    koşula bağlı tekrarlanacak ifade(ler)
```

```
a = 0
while a < 100:
    a = a + 1
    print(a)
```

Bu kodu çalıştırdığımızda, 1'den 100'e kadar olan sayıların ekrana yazdırıldığını görürüz.

```
durum = "devam"
while durum == "devam":
    soru = input("Bir veri girin: ")
    print(soru)
    if soru == "q":
        durum = "yeter"
        print(durum)
```

Klavyeden "q" değeri girilinceye kadar bu döngü devam eder, "q" değeri girildiğinde ise ekrana "yeter" yazacaktır.

```
while True:
 seçenek1 = "(1) toplama"
 seçenek2 = "(2) çıkarma"
 print(seçenek1)
 print(seçenek2)
 soru = input("Yapılacak işlemin numarasını girin: ")
 if soru == "1":
    sayı1 = int(input("Toplama için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama için ikinci sayıyı girin: "))
    print(f"{sayı1} + {sayı2} = {sayı1 + sayı2}")
 if soru == "2":
    sayı3 = int(input("Çıkarma için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma için ikinci sayıyı girin: "))
    print(f"{sayı3} - {sayı4} = {sayı3 - sayı4}")
```

İlk önce programın en başına **while True:** ifadesini ekledik. Bu sayede programımıza şu komutu vermiş olduk: "Doğru olduğu müddetçe aşağıdaki komutları çalıştırmaya devam et! " Yani bir nevi, "Aksi belirtilmediği sürece aşağıdaki komutları çalıştırmaya devam et! " emrini yerine getiriyor. Dolayısıyla **sonsuza kadar devam etmesini istediğiniz döngülerde while True:** 'yu kullanırız.

For Döngüsü (Bounded Iteration / Sınırlı Yineleme) (Mantığı Sayaç Kontrollüdür)

```
for döngü_değikeni in üzerinde_dolaşılacak_veri:
    döngü_içi_işlemler
```

```
sayılar = 143
print("sayı \t karesi \t küpü")
for sayı in str(sayılar):
    print(f"{sayı} \t\t {int(sayı)**2}  \t\t {int(sayı)**3}")
```

sayı	karesi	küpü
1	1	1
4	16	64
3	9	27

Burada **sayılar değişkeni üzerinde döngü kurabilmek için öncelikle bunu str() fonksiyonuyla bir karakter dizisine çevirdik**. Ardından bu değişken içindeki her bir ögeye sayı adını verdik. Ayrıca burada sayı adını verdiğimiz her bir değişken üzerinde aritmetik işlem yapabilmek için **int() fonksiyonuyla sayıya dönüştürdüğümüze** de dikkat edin.

```
toplam = 0
liste = [1,2,5,4]
for eleman in liste:
    toplam = toplam + eleman
    print("Toplam {} Eleman: {}".format(toplam,eleman))
print("Toplam: {}".format(toplam))
```

```
Toplam 1 Eleman: 1
Toplam 3 Eleman: 2
Toplam 8 Eleman: 5
Toplam 12 Eleman: 4
Toplam: 12
```

```
toplam = 0
liste = [13,32,34,56,55]
for eleman in liste:
    if eleman%2==0:
        print(eleman)
```

```
32
34
56
```

```
ulke = "IRAK"
for karakter in ulke:
    print(karakter)
```

```
I
R
A
K
```

```
ulke = "IRAK"
for karakter in ulke:
    print(karakter*3)
```

```
III
RRR
AAA
KKK
```

for ve while Arasındaki Farklar Nelerdir?

<table>
<tr><th>FOR</th><th>WHILE</th></tr>
<tr><td>Bir listenin öğeleri üzerinde yürüyerek işlem yapmamız gerekiyorsa; for döngüsünü kullanırız:

```
liste = [1, 2, 3]
for öğe in liste:
    print(öğe)
```

1
2
3</td><td>Eğer döngü belirli bir koşula sahipse; while'ı kullanmak daha mantıklıdır:

```
sayı = 3
while sayı > 0:
    print(sayı)
    sayı -= 1
```

3
2
1</td></tr>
<tr><td>For döngüsü, programcının belirli sayıda kez yürütmesi gereken bir döngüyü verimli bir şekilde yazmasına olanak tanıyan bir tekrar kontrol yapısıdır.</td><td>Sonsuza dek sürecek bir döngüyü while ile kurmak çok daha kolaydır:

```
while True:
    print("Merhaba Dünya!")
```

</td></tr>
<tr><td></td><td>Döngü değişkeni döngünün dışında initilaze edilir.</td></tr>
</table>

pass Deyimi

Bu deyim, herhangi bir işlem yapmadan geçeceğimiz durumlarda kullanılır.

```
def deneme():
    liste = []
    while True:
        a = input("Giriniz: ")
        if a == "0":
            pass
        elif a == "iptal":
            break
        else:
            liste.append(a)
            print(liste)
deneme()
```

Bir program yazdığımızı ve bir fonksiyon tanımladığımızı varsayalım. Fonksiyonun isminin ne olacağına karar verdiniz, ama fonksiyon içeriğini nasıl yazacağınızı düşünmediniz. Eğer program içinde sadece fonksiyonun ismini yazıp bırakırsanız programınız çalışma sırasında hata verecektir. İşte böyle bir durumda **pass** deyimi imdadınıza yetişir. Bu deyimi kullanarak şöyle bir şey yazabilirsiniz:

```
def bir_fonksiyon():
    pass
```

_ (Alt Çizgi) Karakteri

<table>
<tr><td>Etkileşimli kabukta son işlem gören değeri tutar.

```
>>> 19
19
>>> _ + 3
22
>>> _ * 5 + 40
150
```

</td><td>Döngü değişkeni göz ardı edilerek yerine kullanılabilir.

```
for _ in range(5):
    print('Merhaba Python')
```

```
Merhaba Python
Merhaba Python
Merhaba Python
Merhaba Python
Merhaba Python
```

</td></tr>
</table>

range() Fonksiyonu

Bu fonksiyon belirli aralıkta bulunan sayıları göstermek için kullanılır. Öntanımlı başlangıç değeri 0'dır.

```
print(*range(5))
```
0 1 2 3 4
- Aralarında boşluk var -

```
for a in range(3):
    print(a)
```
0
1
2

```
print(*range(15, 18))
```
15 16 17
- Aralarında boşluk var -

```
for b in range(15, 18):
    print(b)
```
15
16
17

```
print(*range(30, 36, 2))
```
30 32 34

```
print(*range(3,0,-1))
```
3 2 1

```
for sayi in range(1,6):
    print("*" *sayi)
Burdaki *sayi'da yer alan
* işareti çarpma anlamındadır.
```
*
**
***
****
*****

Break ve Continue Komutu

break deyimi, bir döngüyü sona erdirmek için kullanılır.

```
kullanıcı_adı = "hasan"
parola = "hasan123"
while True:
    soru1 = input("Kullanıcı adı: ")
    soru2 = input("Parola: ")
    if soru1 == kullanıcı_adı and soru2 == parola:
        print("Kullanıcı adı ve parolanız onaylandı.")
        break
    else:
        print("Kullanıcı adınız veya parolanız yanlış.")
        print("Lütfen tekrar deneyiniz!")
```
Break komutu yardımıyla, kullanıcı adı ve parola doğru girildiğinde program durduruluyor.

continue ise döngü içinde kendisinden sonra gelen her şeyin es geçilip döngünün en başına dönülmesini sağlar.

```
while True:
    sayi = input("Bir sayı girin: ")
    if sayi == "iptal":
        break
    elif len(sayi) <= 3:
        continue
    print("En fazla üç haneli bir sayı girin.")
```
Kullanıcı "iptal" yazarsa programdan çıkılacaktır. Girdiği sayıdaki hane, üçten fazlaysa ekrana "En fazla üç haneli bir sayı girin. " cümlesi yazdırılacak ve döngünün başına gidilecektir.

KARAKTER DİZİLERİ " "

len() Fonksiyonu

len() fonksiyonunu sayılar üzerinde kullanılamaz. Dolayısıyla bir sayının uzunluğunu len() fonksiyonu yardımıyla öğrenebilmek için öncelikle o sayıyı bir karakter dizisine dönüştürmemiz gerekiyor.

```
n = 123456
print(len(n))
```

TypeError: object of type 'int' has no len()

```
n = 123456
sayi = str(n)
print(len(sayi))
```

6

```
str = "TEST"
i = 0
while i<len(str):
    print(str[i])
```

Sonsuz döngüye girer ve ekrana sürekli alt alta T yazdırır.

```
str = "TEST"
i = 0
while i<len(str):
    print(str[i])
    i +=1
```

T
E
S
T

```
str = "TEST"
for i in str:
    print(i)
```

T
E
S
T

```
str = "1905"
for i in str:
    print(i*2)
```

11
99
00
55

index Numaraları - Dilimleme

Bir karakter dizisinin belirli bir değerine ulaşmak için **index** kullanılır. **Dilimleme** ile karakter dizisinin belli bir bölümündeki değerler elde edilebilir.

[başlangıç : bitiş : adım_sayısı] Öntanımlı adım sayısı = 1'dir.

```
ad = "Hasan"
mesaj = "Benim adım " + ad
print(mesaj[0])
print(mesaj[-1])
print(mesaj[-5])
```

B
n
H

```
print(mesaj[-17])
```

IndexError: string index out of range

```
mesaj = "Benim adım Elif"
print(mesaj[0:8])       Benim ad
print(mesaj[:8])        Benim ad
print(mesaj[4:8])       m ad
print(mesaj[8:])        ım Elif
print(mesaj[::])        Benim adım Elif
print(mesaj[-11:-1])    m adım Eli
print(mesaj[0:14:2])    Bnmaı l
print(mesaj[::2])       Bnmaı lf
print(mesaj[0::2])      Bnmaı lf
print(mesaj[::-2])      fl ıamnB
print(mesaj[4:-1])      m adım Eli
print(mesaj[-11:-1])    m adım Eli
```

Yukarıdaki örnekte belirttiğimiz index'deki karakterleri ekrana yazdırmış olduk. Sağdaki 7. satırda 0 ile 11. index arasındaki karakterleri 2'şer atlayarak ekrana yazdırmamızı istiyor.

Karakter Dizilerinin Üzerinde Değişiklik

`site1 = "www.google.com"`
`site2 = "www.yahoo.com"`
 	
`for i in site1, site2:`
`  print("http://", i[4:], sep="")`
	http://google.com
	http://yahoo.com
	site1 ve site2 değişkenlerinden www ifadesini atıp yerine http:// ifadesini yerleştirdik.

Karakter dizileri üzerinde yapılan değişikliklerin kalıcı olmamasını nedeni, karakter dizilerinin değiştirilemeyen (immutable) bir veri tipi olmasıdır.

İki tür veri tipi bulunur; **değiştirilemeyen (immutable (int, string, tuple, bool, float))** ve **değiştirilebilen (mutable (list, dict, set, user-defined type))**.

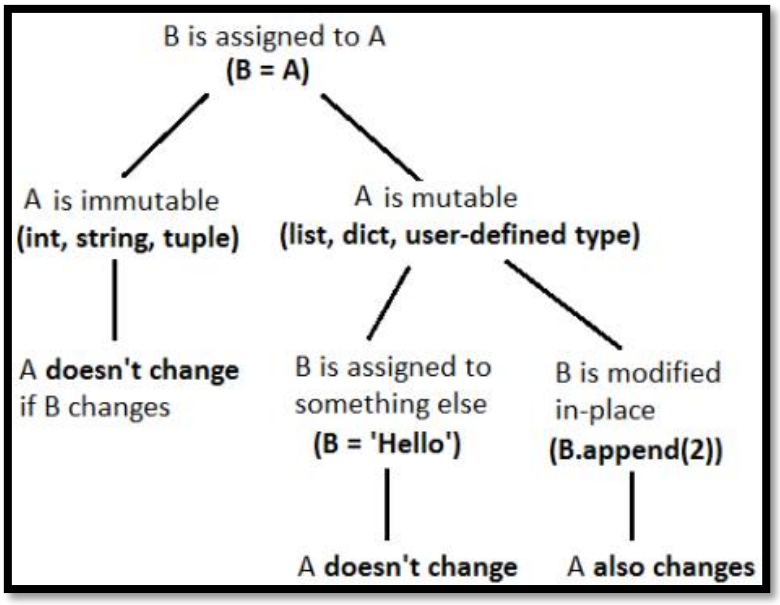
Python'da bir karakter dizisini bir kez tanımladıktan sonra bu karakter dizisi üzerinde artık değişiklik yapamazsınız. Eğer bir karakter dizisi üzerinde değişiklik yapmanız gerekiyorsa, yapabileceğiniz tek şey o karakter dizisini yeniden tanımlamaktır.

`mesaj = "Hello world"`
`mesaj = "Hello" + "W" + mesaj[7:11]`
`print(mesaj)`
	`mesaj = "Hello world"`
	`print(id(mesaj))`
	`mesaj = "Hello" + "W" + mesaj[7:11]`
	`print(id(mesaj))`

Hello World w ile W karakterini değiştirdik.	2223342795312 2223342867184

Burada yaptığımız şey mesaj değişkeninin değerini değiştirmek değildir. Biz burada aslında bambaşka bir mesaj değişkeni daha tanımlıyoruz. Gördüğünüz gibi, ilk mesaj değişkeni ile sonraki mesaj değişkeni farklı kimlik numaralarına sahip.

`liste = ["Ali",2 ,"22"]`
`print(liste)`
`print(id(liste))`
`liste.pop()`
`print(liste)`
`print(id(liste))`
	['Ali', 2, '22']
	1482665554816
	['Ali', 2]
	1482665554816



Proces abstraction, arka planda ne olduğunu bilmeden kullandıklarımızına denir, örneğin input fonksiyonu. **Data abstraction,** lazım olan verileri alırız ve bilgisayara aktarırız.

enumerate() Fonksiyonu

Bu fonksiyonu ögeleri numaralandırmamızı sağlar, içerikleri görmek için * işaretini kullanmalıyız.

```
isimler = ["ahmet", "ışık", "ismail"]
print(*enumerate(isimler))
```

(0, 'ahmet') (1, 'ışık') (2, 'ismail')

```
print(*enumerate("Python"))
```

(0, 'P') (1, 'y') (2, 't') (3, 'h') (4, 'o') (5, 'n')

```
isimler = ["ahmet", "ışık", "ismail", "çiğdem"]
for sıra, öğe in enumerate(isimler):
    print(sıra, öğe)
```

0 ahmet
1 ışık
2 ismail
3 çiğdem

```
isimler = ["ahmet", "ışık", "ismail", "çiğdem"]
for sıra, öğe in enumerate(isimler, 1):
    print(sıra, öğe)
```

1 ahmet
2 ışık
3 ismail
4 çiğdem

zip() Fonksiyonu

Bu fonksiyonu, farklı dizilerin ögelerini birbiriyle birleştirmek için kullanıyoruz.

```
liste1 = [1, 2, 3]
liste2 = ["bir", "iki", "üç"]
print(*zip(liste1, liste2))
```

(1, 'bir') (2, 'iki') (3, 'üç')

```
liste1 = [1, 2, 3]
liste2 = ["bir", "iki", "üç"]
for a, b in zip(liste1, liste2):
    print(a, b)
```

1 bir
2 iki
3 üç

id() Fonksiyonu

Her nesnenin bir kimliği **(eşsizdir)** vardır. Kimlik o nesnenin bellekteki adresini temsil eder.

```
a = 100
print(id(a))
print(id(100))
b = a
print(id(b))
```

2653007908176
2653007908176
2653007908176

```
mesaj = "ey edip pide ye"
ters_Mesaj = mesaj[::-1]
print(mesaj == ters_Mesaj)   True
print(mesaj is ters_Mesaj)   False
print(id(mesaj))
print(id(ters_Mesaj))
```

2163962001648
2163962073584

reversed() Fonksiyonu

String'in değerini ya da listeyi tersine çevirir. İçerikleri görmek için * işaretini kullanmalıyız.

```
mesaj = "Python"
print(reversed(mesaj))
print(*reversed(mesaj))
```

<reversed object at 0x00000224E6C263E0>
n o h t y P

dir() Fonksiyonu

Bu metot bize Python'daki bir nesnenin metotları hakkında bilgi edinme imkanı sağlar.

```
print(dir(str))
```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__' ...]

KARAKTER DİZİLERİ METOTLARI

Metotlar **değişken_Ismi.metot_Adi(Parametreler)** şeklinde bir syntax'a sahiptir. Nokta ile çağrılır.

replace() Metotu

Bir karakter dizisi içindeki karakterleri başka karakterlerle değiştirebileceğiz.

```
ad = "elif"
print(id(ad))
ad = ad.replace("e", "E")
print(ad.replace("e", "E"))
print(id(ad))
```

```
1581830355504
Elif
1581830355632
```

split() Metotu

Bu metotun görevi karakter dizilerini belli noktalardan bölmektir.

```
text = "Kara Harp Okulu".split()
print(text[1])
print(text[0][0])
text = "Kara Harp Okulu"
print(text.split())

kurum = input("Kurum adını girin: ")
for i in kurum.split():
    print(i[0], end="")
```

```
Harp
K
['Kara', 'Harp', 'Okulu']

- Özel Kuvvetler Komutanlığı – yazdığımızı varsayalım.
ÖKK
```

lower() Metotu - upper() Metotu

Karakter dizisindeki büyük harfleri küçük harflere dönüştürürken lower() metotunu, Karakter dizisindeki küçük harfleri büyük harflere dönüştürürken upper() metotunu kullanırız.

```
mesaj = "Necmettin Erbakan Üniversitesi"
print(mesaj.upper())
print(mesaj.lower())
```

```
NECMETTIN ERBAKAN ÜNIVERSITESI
necmettin erbakan üniversitesi
```

islower() Metotu – isupper() Metotu

islower() metotu bir karakter dizisinin tamamının küçük harflerden oluşup oluşmadığını denetleme imkânı sağlar, isupper() metotu ise karakter dizilerinin tamamının büyük harflerden oluşup oluşmadığını denetlememizi sağlar.

```
mesaj = "necmettin erbakan üniversitesi"
print(mesaj.isupper())
print(mesaj.islower())
```

```
False
True
```

capitalize() Metotu

capitalize() metotunun görevi karakter dizilerinin yalnızca ilk harfini büyütmektir.

```
mesaj = "necmettin erbakan üniversitesi"
print(mesaj.capitalize())
```

```
Necmettin erbakan üniversitesi
```

endswith() Metotu

Bu metotla bir karakter dizisinin hangi karakterle veya karakter dizisi ile bittiğini sorgulayabiliriz.

```
isim = "Elif"
print(isim.endswith("f"))
```

True

```
d1 = "filanca.ogg"
d2 = "falanca.mp3"
d3 = "dosya.avi"
d4 = "c++.mp3"
for i in d1, d2, d3, d4:
    if i.endswith(".mp3"):
        print(i)
```

falanca.mp3
c++.mp3

```
d1 = "filanca.ogg"
d2 = "falanca.mp3"
d3 = "dosya.avi"
d4 = "c++.mp3"
for i in d1, d2, d3, d4:
    if i[-4:len(i)] == ".mp3":
        print(i)
```

falanca.mp3
c++.mp3

startswith() Metotu

Bu metot ise bir karakter dizisinin hangi karakter veya karakterlerle başladığını denetler.

```
isim = "Elif"
print(isim.startswith("e"))
```

False

```
d1 = "filanca.ogg"
d2 = "falanca.mp3"
d3 = "dosya.avi"
d4 = "c++.mp3"
for i in d1, d2, d3, d4:
    if i.startswith("f"):
        print(i)
```

filanca.ogg
falanca.mp3

```
d1 = "filanca.ogg"
d2 = "falanca.mp3"
d3 = "dosya.avi"
d4 = "c++.mp3"
for i in d1, d2, d3, d4:
    if i[0] == "f":
        print(i)
```

filanca.ogg
falanca.mp3

strip() Metotu - lstrip() Metotu - rstrip() Metotu

Karakter dizisinin sağında ve solunda bulunan boşlukları silmek için strip() metotu kullanılır.

```
isim = "  Elif  "
print(isim.strip())
```

Elif

```
isim = "kazak"
print(isim.strip("k"))
```

aza

```
isim = "kazaK"
print(isim.strip("k"))
```

azaK

lstrip metotu bir karakter dizisinin sol tarafındaki gereksiz karakterlerden kurtulmamızı sağlar, rstrip ise sağ tarafındaki gereksiz karakterlerden kurtulmamızı sağlar.

```
kiyafet = "kAzak"
print(kiyafet.rstrip("k"))
print("kabak".lstrip("k"))
```

kAza
abak

join() Metotu

Karakter dizilerini tekrar birleştirmek için bu metotu kullanırız.

```
OKK = "Özel Kuvvetler Komutanlığı"
bolunmus = OKK.split()
print(bolunmus)
print("".join(bolunmus))
print(" - ".join(bolunmus))
birlestir = " "
print(birlestir.join(bolunmus))
```

```
['Özel', 'Kuvvetler', 'Komutanlığı']
ÖzelKuvvetlerKomutanlığı
Özel – Kuvvetler – Komutanlığı
Özel Kuvvetler Komutanlığı
```

count() Metotu

Bu metotun görevi bir karakter dizisi içinde belli bir karakterin kaç kez geçtiğini sorgulamaktır.

```
sehir = "Kahramanmaraş"
print(sehir.count("a"))
```

5

```
sehir = "adanalalılar"
print(sehir.count("a",1,5))
```

Burada, 1. karakter ile 5. karakter arasında kalan ‘a’ harflerini saymış olduk.

```
sehir = "adana"
print(sehir.count("a",1))
```

2

Saymaya "adana" karakter dizisinin 1. sırasından başlanılacak. Dolayısıyla 0. sıradaki "a" harfini saymayacağımız için toplam "a" sayısı 2 olacaktır.

index() Metotu - rindex() Metotu

Karakterlerin, bir karakter dizisi içinde hangi sırada bulunduğunu öğrenmek için index() adlı bir metottan yararlanabiliriz. index() ve rindex() metotlarının birbirinden tek farkı, index() metotunun karakter dizilerini soldan sağa, rindex() metotunun ise sağdan sola doğru okumasıdır.

```
sehir = "adana"
print(sehir.index("n"))
```

3

```
sehir = "adanalılar"
print(sehir.rindex("a"))
print(sehir.rindex("r"))
```

8
9

```
sehir = "adanalılar"
print(sehir.index("a",3))
```

4

index() metotunun ikinci parametresi, Python’ın aramaya kaçıncı sıradan itibaren başlayacağını gösteriyor.

```
sehir = "adanalılar"
print(sehir.index("l",3,9))
```

5

sorgulama işlemini hangi sıra aralıklarından gerçekleştireceğini gösterir

find() Metotu - rfind() Metotu

find() ve rfind() metotlarının görevi de karakter dizisi içindeki bir karakterin konumunu sorgulamaktır. Peki **index() - rindex()** ve **find() - rfind()** metotları arasında ne fark var? index() ve rindex() metotları karakter dizisi içindeki karakteri sorgularken, eğer o karakteri bulamazsa bir ValueError hatası verir. Ama find() ve rfind() metotları böyle bir durumda -1 çıktısı verir.

center() Metotu

center() metotunu karakter dizilerini ortalamak için kullanabiliriz.

```
mesaj = "adana"
print(mesaj.center(10))
```

adana

```
mesaj = "adana"
print(mesaj.center(10,"-"))
```

--adana---

center() metotuna verilen genişlik parametresi aslında bir karakter dizisinin toplam kaç karakterlik bir yer kaplayacağını gösteriyor. Kaplanacak yere karakter dizisinin kendisi de dahildir. Yani 10 olarak belirttiğimiz boşluk adedinin 6 'sı 'python' kelimesinin kendisi tarafından işgal ediliyor.

rjust() Metotu - ljust() Metotu

Bu metotlar da tıpkı bir önceki center() metotu gibi karakter dizilerini hizalama vazifesi görür. rjust() metotu bir karakter dizisini sağa yaslarken, ljust() metotu karakter dizisini sola yaslar. Metota verdiğimiz "." karakterini görebilmemiz için, verdiğimiz sayı, en az karakter dizisinin boyutunun bir fazlası olması gerekir.

```
mesaj = "adana"
print(mesaj.ljust(10,"-"))
```

adana-----

```
for i in "elma", "patlıcan":
    print(i.ljust(10, "."))
```

elma.....
patlıcan..

isalpha() Metotu

Bu metot yardımıyla bir karakter dizisinin 'alfabetik' olup olmadığını denetleyeceğiz. Peki 'alfabetik' ne demek? Eğer bir karakter dizisi içinde yalnızca alfabe harfleri ('a', 'b', 'c' gibi...) varsa o karakter dizisi için 'alfabetik' diyoruz.

```
mesaj = "adana"
print(mesaj.isalpha())
```

True

```
mesaj = "adana5"
print(mesaj.isalpha())
```

False

isdigit() Metotu

Bu metotla bir karakter dizisinin sayısal olup olmadığını denetleyebiliriz. Sayılardan oluşan karakter dizilerine 'sayı değerli karakter dizileri' adı verilir.

```
mesaj = "55"
print(mesaj.isdigit())
```

True

```
mesaj = "adana5"
print(mesaj.isdigit())
```

False

isalnum() Metotu

Bu metot, bir karakter dizisinin 'alfanümerik' olup olmadığını denetlememizi sağlar. Sayı ve harflerden oluşan karakter dizilerine alfanümerik karakter dizileri adı verilir.

```
mesaj = "adana5"
print(mesaj.isalnum())
```

True

isdecimal() Metotu

Bu metot yardımıyla bir karakter dizisinin ondalık sayı cinsinden olup olmadığını denetliyoruz.

`mesaj1 = "123"`
`print(mesaj1.isdecimal())`
True
	`mesaj2 = "0.123"`
	`print(mesaj2.isdecimal())`
	False
	Bu sayı kayan noktalı sayı (floating number) olduğu için False cevabını alıyoruz.

isidentifier() Metotu

Neyin tanımlayıcı olup neyin tanımlayıcı olamayacağını denetlememizi sağlar. Örneğin, değişken adları bir sayı ile başlayamıyordu. Dolayısıyla sayı ile başlayan bir değişken tanımlayamıyoruz.

title() Metotu

title() metotu karakter dizilerinin ilk harfini büyütür. Ama capitalize() metotundan farklı olarak bu metot, birden fazla kelimeden oluşan karakter dizilerinin her kelimesinin ilk harflerini büyütür.

`mesaj = "özel kuvvetler komutanlığı"` `print(mesaj.title())`	Özel Kuvvetler Komutanlığı

********** NOT **********

Metot ve Fonksiyon Arasındaki Fark Nedir?

Metotlar "." ile kullanılır ve bir obje üzerindeyken çağrılır. Örneğin; mesaj.title()

Fonksiyonları direk kullanabiliriz yani bir obje üzerinden çağırmamıza gerek yok. Örneğin; range()

Python programlama dilinde sadece aynı tür verileri birbirleriyle birleştirebiliriz.

LİSTELER (LISTS) [ ]

Öğeleri birbirinden virgülle ayırıp, bunların hepsini köşeli parantezler içine alarak listeleri elde ederiz.

```
liste = ["Ali", "Veli", ["Ayşe", "Nazan"], 37, 65, 5.8]
print(type(liste))      <class 'list'>
print(len(liste))       6
bos = []
print(type(bos))        <class 'list'>
```

Sağdaki kodlardan gördüğünüz gibi, boş bir liste oluşturmak için liste = [] koduna alternatif olarak list() fonksiyonundan da yararlanabilirsiniz.

```
li = list()
print(li)
```
[]

```
sayılar = [[0, 7], [6, 10], [12, 15]]
for i in sayılar:
    print(*range(*i)
```
0 1 2 3 4 5 6
6 7 8 9
12 13 14

list() Fonksiyonu

list() fonksiyonu, kendisine verilen obje eğer uygun veri tipinde ise listeye dönüştürür.

```
isimler = "elma, armut, çilek"
print(isimler.split(", "))
```
['elma', 'armut', 'çilek']

```
alfabe = "abcdiklmnvyz"
print(alfabe.split("i"))
```
['abcd', 'klmnvyz']

```
alfabe = "abcdiklmnz"
harf_listesi = list(alfabe)
print(harf_listesi)
```
['a', 'b', 'c', 'd', 'i', 'k', 'l', 'm', 'n', 'z']

Listelerin Öğelerine Erişmek

Hem listelerde hem de karakter dizilerinde Python saymaya 0 'dan başlar.

```
mesaj = "Dur Polis"
print(mesaj[0])
```
D

```
mesaj = "Dur Polis"
liste = mesaj.split()
print(liste[0])
```
Dur

```
mesaj = ["Dur", "Hak"]
print(mesaj[0])
```
Dur

```
meyveler = ["elma", "armut", "çilek"]
for sıra, öğe in enumerate(meyveler, 1):
    print("{}. {}".format(sıra, öğe))
```
1. elma
2. armut
3. çilek

```
meyveler = ["elma", "armut",["1","2"], "çilek"]
print(meyveler[-1])          çilek
print(meyveler[0:2])         ['elma', 'armut']
print(meyveler[::-1])        ['çilek', ['1', '2'], 'armut', 'elma']
print(meyveler[:-2])         ['elma', 'armut']
print(meyveler[-3:-1])       ['armut', ['1', '2']]
print(meyveler[1:2])         ['armut']
print(meyveler[:])           ['elma', 'armut', ['1', '2'], 'çilek']
print(meyveler[0:3:1])       ['elma', 'armut', ['1', '2']]
```

İç içe geçmiş listenin içindeki listeden öğe almak için; **gömülü listenin önce ana listedeki konumunu, ardından da almak istediğimiz öğenin gömülü listedeki konumunu belirtmektir.**

```
liste = ["Ali", "Veli", ["Nazan", "Zeynep"], 34, 79, 5.6]
print(liste[2][1])      Zeynep
```

Listelerin Öğelerini Değiştirmek

Bir liste üzerinde değişiklik yapabilmek için o listeyi **yeniden tanımlamamıza gerek yok.**

```
renkler = ["kırmızı", "sarı", "mavi"]
print(renkler)
renkler[0] = "siyah"
print(renkler)
```

['kırmızı', 'sarı', 'mavi']
['siyah', 'sarı', 'mavi']

```
liste = [1, 2, 3]
liste[0 : len(liste)] = 5, 6, 7
print(liste)
```

[5, 6, 7]

Listeye Öğe Eklemek - Listeden Bir Öğe Çıkarmak – Listeyi Tekrarlatmak

Python'da + işareti kullanarak bir listeye öğe ekleyecekseniz, eklediğiniz öğenin de liste olması gerekiyor. Mesela bir listeye doğrudan karakter dizilerini veya sayıları ekleyemezsiniz.

```
liste = [2, 4, 5]
print(liste + [8])
```

[2, 4, 5, 8]

```
liste = [2, 4, 5]
print(liste + 8)
```

TypeError: list (not "int") to list

```
liste = [2, 4, 5]
print(liste + "8")
```

TypeError: list (not "str") to list

```
list = [1,2,["re"]*3]
print(list)
```

[1, 2, ['re', 're', 're']]

Listeleri Silmek

Bir listeden öğe silmek için **del** adlı ifadeden yararlanabiliriz.

```
liste = [5, 3, 2, 9]
del liste
print(liste)
```

NameError: name 'liste' is not defined. Did you mean: 'list'?
liste adında bir değişken olmadı için hata verdi.

```
liste = [5, 3, 2, 9]
del liste[-1]
print(liste)
```

[5, 3, 2]

Listeleri Birleştirmek

Karakter dizilerinde olduğu gibi, listelerde de birleştirme işlemleri için + işlecinden yararlanabiliriz.

```
derlenen_diller = ["C", "C++"]
yorumlanan_diller = ["Python", "HTML"]
programlama_dilleri = derlenen_diller + yorumlanan_diller
print(programlama_dilleri)
```

['C', 'C++', 'Python', 'HTML']

```
liste = []
alfabe = "abcdef"
for harf in alfabe:
    liste += harf
print(liste)
```

['a', 'b', 'c', 'd', 'e', 'f']
list() fonksiyonu da tam olarak böyle çalışır. Yani bir karakter dizisi üzerinde döngü kurarak, o karakter dizisinin her bir öğesini tek tek bir listeye atar

```
print(list("abcdef"))
```

['a', 'b', 'c', 'd', 'e', 'f']
list() fonksiyonu da ancak, üzerinde döngü kurulabilen nesneler üzerinde çalışabilir.

```
notlar = []
for i in range(3):
    veri = int(input("{}. not: ".format(i+1)))
    notlar += list(veri)
print("Girdiğiniz notlar: ", *notlar)
```
TypeError: 'int' object *is not* iterable

Kullanıcıdan gelen veri değerini int() fonksiyonuyla sayıya dönüştürdüğümüz için ve sayılar da üzerinde döngü kurulabilen yani alt alanlara bölünebilen bir veri tipi olmadığı için list() fonksiyonuna parametre olarak atanamaz.
Peki kullanıcıdan gelen veri değerini sayıya dönüştürmeden, karakter dizisi biçiminde list() fonksiyonuna parametre olarak verirsek ne olur? Bu durumda list() fonksiyonu çalışır, ama istediğimiz gibi bir sonuç vermez.

```
notlar = []
for i in range(3):
    veri = input("{}. not: ".format(i+1))
    notlar += list(veri)
print("Girdiğiniz notlar: ", *notlar)
```
1. not: 3
2. not: 45
3. not: 89
Girdiğiniz notlar: 3 4 5 8 9

Tek haneli sayılar düzgün bir şekilde listeye eklenir, ancak çift ve daha fazla haneli sayılar ise listeye parça parça eklenir, bu da bizim istemeyeceğimiz bir şey.

```
liste = []
while True:
    sayı = input("Bir sayı girin (çıkmak için q): ")
    if sayı == "q":
        break
    sayı = int(sayı)
    if sayı not in liste:
        liste += [sayı]
        print(liste)
    else:
        print("Bu sayıyı daha önce girdiniz!")
print("Girdiğiniz sayılar: ",liste)
```
Bir sayı girin (çıkmak için q): 14
[14]
Bir sayı girin (çıkmak için q): 56
[14, 56]
Bir sayı girin (çıkmak için q): 1
[14, 56, 1]
Bir sayı girin (çıkmak için q): q
Girdiğiniz sayılar: [14, 56, 1]

q karakterini girmediğimiz sürece bu döngü sonsuza kadar sürecektir.

Listeleri Kopyalamak (Aliasing – Örtüşme)

Bir değişkenin bir nesneyle ilişkilendirilmesine **referans** denir.

```
a = [1,2]
b = [1,2]
print(a is b)
```

False

```
li1 = ["elma", "armut", "erik"]
li2 = li1
print(li1)                          ['elma', 'armut', 'erik']
print(li2)                          ['elma', 'armut', 'erik']
li1[0] = "karpuz"
print(li1)                          ['karpuz', 'armut', 'erik']
print(li2)                          ['karpuz', 'armut', 'erik']
```

Biz biraz önce li1 üzerinde değişiklik yapmıştık, ama görünüşe göre bu değişiklikten li2'de etkilenmiş. Hatırlarsanız, **listeler değiştirilebilir (mutable)** bir veri tipiydi. Ama karakter dizileri değil. Zira biraz önce li1 ve li2 üzerinde yaptığımız işlemin bir benzerini karakter dizileri ile yaparsak farklı bir sonuç alırız.

```
Li1 = ['elma', 'armut', 'erik']
li2 = li1
li2.append('karpuz')
print(li2)                      ['elma', 'armut', 'erik', 'karpuz']
print(li1)                      ['elma', 'armut', 'erik', 'karpuz']
```

```
meyve = "elma"
sebze = meyve
print(meyve)                        elma
print(sebze)                        elma
meyve = "E" + meyve[1:]
print(meyve)                        Elma
```

Burada yaptığımız şey bir "değişiklik" değil. Çünkü biz burada varolan meyve adlı değişken üzerinde bir değişiklik yapmak yerine, yine meyve adı taşıyan başka bir değişken oluşturuyoruz.

```
print(sebze)                        elma
```

Gördüğünüz gibi, bu değişiklik sebze dizisini etkilememiş. Bunun sebebi, karakter dizilerinin **değiştirilemeyen (immutable)** bir veri tipi olmasıdır.

```
print(id(meyve))                    1914315854768
print(id(sebze))                    1914315782704
```

Bu sonuç bize, bu iki karakter dizisinin bellekte farklı konumlarda saklandığını gösteriyor. Dolayısıyla **Python, bir karakter dizisini kopyaladığımızda bellekte ikinci bir nesne daha oluşturuyor. Bu nedenle birbirinden kopyalanan karakter dizilerinin biri üzerinde yapılan herhangi bir işlem öbürünü etkilemiyor. Ama listelerde (mutable) durum farklı.**

```
list1 = ['sarı','mor']
list2 = ['siyah']
list3 = [list1]
list3.append(list2)
print(list3)                    [['sarı', 'mor'], ['siyah']]
list2.append('beyaz')
print(list2)                    ['siyah', 'beyaz']
print(list3)                    [['sarı', 'mor'], ['siyah', 'beyaz']]
```

```
liste1 = ["Ali", "Elif", "Sefa"]
liste2 = liste1
print(liste2)                       ['Ali', 'Elif', 'Sefa']
print(id(liste1))                   1944059835264
print(id(liste2))                   1944059835264
```

Gördüğünüz gibi, liste1 ve liste2 adlı listeler aynı kimlik numarasına sahip. Yani bu iki nesne birbiriyle aynı. Dolayısıyla birinde yaptığınız değişiklik öbürünü de etkiler. Eğer birbirinden kopyalanan listelerin birbirini etkilemesini istemiyorsanız, önünüzde birkaç seçenek var.

1-

```
liste1 = ["Ali", "Elif", "Sefa"]
liste2 = liste1[:]
liste1[0] = "Esra"
print(liste1)                       ['Esra', 'Elif', 'Sefa']
print(liste2)                       ['Ali', 'Elif', 'Sefa']
print(id(liste1))                   2160438735744
print(id(liste2))                   2160438702400
```

2- list() fonksiyonunu kullanarak:

```
liste1 = ["Ali", "Elif", "Sefa"]
liste2 = list(liste1)
liste2[0] = 'Alp'
print(liste1)                       ['Ali', 'Elif', 'Sefa']
print(liste2)                       ['Alp', 'Elif', 'Sefa']
print(id(liste1))                   2241336439681
print(id(liste2))                   2241336669376
```

Mutation and Iteration

```
l1 = [1, 2, 3, 4]
l2 = [1, 2, 5, 6]
for i in l1:
    if i in l2:
        l1.remove(i)
print(l2)                           [1, 2, 5, 6]
```

```
l1 = [1, 2, 3, 4]
l2 = [1, 2, 5, 6]
l3 = l1[:]
for e in l3:
    if e in l2:
        l1.remove(e)
print(l1)                           [3, 4]
print(l2)                           [1, 2, 5, 6]
print(l3)                           [1, 2, 3, 4]
```

l3'ün bütün ögeleri içermesinin sebebi; döngü en başta oluştuğu için index numaraları yani sıradaki elemanın numarası değişmez bu yüzden sonuçlar l3'ün içindeki elemanlar silinmemiş olarak gözüküyor.

Liste Üreteçleri (List Comprehensions)

Liste üreteçlerinin görevi liste üretmektir.

```
liste = [i for i in range(10)]
print(liste)
```

Burada 0'dan 10'a kadar olan sayıları tek satırda bir liste haline getirdik. Yukardaki kod sağdaki şekildede yazılabilir.

Her iki geçici değişkenin ismi aynı olmak zorundadır mesela bu örnek için bu değişkenimizin isimleri i'dir.

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

```
liste = []
for i in range(10):
    liste += [i]
print(liste)
```

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

```
liste = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9],
         [10, 11, 12]]
tümü = []
for i in liste:
    for z in i:
        tümü += [z]
print(tümü)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

```
liste = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9],
         [10, 11, 12]]
tümü = [z for i in liste for z in i]
print(tümü)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

********** NOT **********

```
liste = [[8,5,-1],[3,7,2],[-4,9,6]]
yeniListe = [number for i in liste for number in i if number>0]
print(yeniListe)
[8, 5, 3, 7, 2, 9, 6]
```

```
liste = [1, '4', 9, 0, 'a']
yeniListe = [x**2 for x in liste if type(x)==int]
print(yeniListe)
[1, 81, 0]
```

LİSTELERİN METOTLARI

append() Metotu

Bu metotu, bir listeye öğe eklemek için kullanırız.

`liste = ["elma", "armut", "çilek"]`
`liste.append("erik")`
`print(liste)`
	['elma', 'armut', 'çilek', 'erik']

+ işleci ile append() metotu arasındaki fark; append() metotunun yalnızca tek bir parametre alabilmesidir. Yani bu metotu kullanarak bir listeye birden fazla öğe ekleyemezsiniz.

Eğer bu metotu kullanarak bir listeye yine bir liste eklemeye çalışırsanız, eklediğiniz liste tek bir öğe olarak eklenecektir.

`li1 = [1, 3, 4]`
`li2 = [10, 11, 12]`
`li1. append(li2)`
`print(li1)`
	[1, 3, 4, [10, 11, 12]]

extend() Metotu

extent() metotu listeleri genişletir.

`li1 = [1, 3, 4]`
`li2 = [10, 11, 12]`
`li1. extend(li2)`
`print(li1)`
	[1, 3, 4, 10, 11, 12]
	li1 değişti ve li1+li2'nin değerini aldı ama aşağıda + ile yapılan örnek bu şekilde değil.

`marka = ["Wolkswagen", "Audi", "BMW"]`
`model = ["Polo", "Golf", "Passat"]`
`print(marka + model)`
	['Wolkswagen', 'Audi', 'BMW', 'Polo', 'Golf', 'Passat']

`marka = ["Wolkswagen", "Audi", "BMW"]`
`model = ["Polo", "Golf", "Passat"]`
`for i in model:`
`  marka.append(i)`
`print(marka)`
	['Wolkswagen', 'Audi', 'BMW', 'Polo', 'Golf', 'Passat']

`marka = ["Wolkswagen", "Audi", "BMW"]`
`model = ["Polo", "Golf", "Passat"]`
`marka.extend(model)`
`print(marka)`
	['Wolkswagen', 'Audi', 'BMW', 'Polo', 'Golf', 'Passat']

insert() Metotu

insert() metotu, öğeleri listenin istediğimiz bir konumuna yerleştirir.

`liste = ["elma", "armut", "çilek"]`
`liste.insert(0, "erik")`
`print(liste)`
	['erik', 'elma', 'armut', 'çilek']

remove() Metotu

Bu metot listeden öğe silmemizi sağlar.

```
Liste = ["elma", "armut", "çilek"]
liste.remove("elma")
print(liste)
```

['armut', 'çilek']

reverse() Metotu

String değerini ya da listeyi tersine çevirir. İçerikleri görmek için * işaretini kullanmalıyız.

```
Meyveler = ["elma", "armut", "çilek", "kiraz"]
print(*reversed(meyveler))
```

kiraz çilek armut elma

pop() Metotu

Bu metot da bir listeden öğe silmemizi sağlar. remove() metotundan biraz farklı olarak pop() metotunu kullanarak bir liste öğesini sildiğimizde, silinen öğe ne ise ekrana basılacaktır.

Eğer index belirtilmezse, listedeki son elemanı çıkarır.

```
Liste = ["elma", "armut", "çilek"]
print(liste.pop(0))
```

elma

sort() Metotu

sort() metotu bir listenin öğelerini alfabetik olarak ya da sayısal olarak sıraya dizmemizi sağlar.

```
Üyeler = ['Ali', 'Ceylan', 'Mahmut', 'Zeynep', 'Kadir', 'Tolga']
üyeler.sort()
print(üyeler)
['Ali', 'Ceylan', 'Kadir', 'Mahmut', 'Tolga', 'Zeynep']

sayılar = [1, 0, -1, 4, 10, 3, 6]
sayılar.sort()
print(sayılar)
[-1, 0, 1, 3, 4, 6, 10]
```

clear() Metotu

Bu metotun görevi bir listenin içeriğini tamamen silmektir.

```
Liste = [1, 2, 3, 5, 10, 20, 30, 45]
liste.clear()
print(liste)
```

[]
Boş küme çıkıyor karşımıza.

count() Metotu

count() metotu bir öğenin o veri tipi içinde kaç kez geçtiğini söyler.

```
Liste = ["elma", "armut", "elma", "çilek"]
print(liste.count("elma"))
```

2

TUPLE (DEMETLER) ( )

Tuple Tanımlamak

Nasıl karakter dizilerinin ayırt edici özelliği tırnak işaretleri, listelerin ayırt edici özelliği ise köşeli parantez işaretleri ise, demetlerin ayırt edici özelliği de normal parantez işaretleridir. Tuple **değiştirilemez (immutable)** veri tipindedir.

```
demet = ("ahmet", "mehmet", 23, 45)
print(type(demet))
demet = "ahmet", "mehmet", 23, 45
print(type(demet))
```

```
<class 'tuple'>
<class 'tuple'>
```

Demet oluşturmak için veya listeyi tuple'a dönüştürmek için **tuple()** isimli **fonksiyondan** yararlanabiliriz.

`print(tuple('abcde'))`	('a', 'b', 'c', 'd', 'e')
`print(tuple(["ali", "cix", 45]))`	('ali', 'cix', 45)

Burada, ["ali", "cix", 45] adlı bir listeyi tuple() fonksiyonu yardımıyla demete dönüştürdük.

Tek Öğeli bir Demet Tanımlamak

Eğer tek öğeye sahip bir demet oluşturacaksak şöyle bir şey yazmalıyız:

```
demet = ('ahmet',)    veya    demet = 'ahmet',
```

Demetlerin Öğelerine Erişmek

```
demet = ('elma', 'armut', 'kiraz')
print(demet[0])
print(demet[-1])
print(demet[:2])
```

```
elma
kiraz
('elma', 'armut')
```

Demetlerle Listelerin Birbirinden Farkı

Listeler değiştirilebilirken **(mutable)**, demetler değiştirilemez **(immutable)** veri tipindedir.

```
demet = ('elma', 'kiraz')
demet[0] = 'karpuz'
```

```
TypeError: 'tuple' object does not support item assignment
```

Demetin herhangi bir öğesini değiştirmeye çalıştığımızda Python bize bir hata mesajı gösteriyor.

NOT: index() ve count() metotlarını tuple'da kullanabiliriz.

SÖZLÜKLER (DICTIONARIES) {}

Sözlük Tanımlamak

Sözlüklerin ayırt edici özelliği süslü parantezleridir. Sözlük en basit haliyle şöyle görünür:

```
sozluk = {}
print(type(sozluk))
```

```
<class 'dict'>
```

Sözlüklerin Python programlama dilindeki teknik karşılığı **dict** ifadesidir.

İki nokta üst üste işaretinin solundaki karakter dizisine **anahtar (key)**, sağındaki karakter dizisine ise **değer (value)** adı verilir. **Anahtarlar eşsiz (unique) ve immutable olmak zorundadır. Değerler ise birden fazla aynı değeri içerebilir ayrıca immutable ya da mutable olabilir.**

```
kelimeler = {"kitap": "book"}
print(len(kelimeler))
```

```
1
```

Yukarıdaki sözlüğün 2 öğeden oluştuğu yanılgısına düşebilirsiniz ama bu yanlış. Sözlüklerde de birden fazla öğeyi birbirinden ayırmak için **virgül** işaretlerinden yararlanacağız.

Sözlük Öğelerine Erişmek

Sözlük öğelerine erişmek için: **değişken_ismi[sözlük_öğesinin_anahtar_adı]** kullanırız.

```
sözlük = {"kitap" : "book",
          1 : "computer",
          "programlama": "programming",}
print(sözlük["kitap"])
print(sözlük[1])
```

```
book
computer
```

Sözlük içinde iki nokta üst üste işaretinin sol tarafında görünen öğeleri köşeli parantez içinde yazarak, iki nokta üst üste işaretinin sağ tarafındaki değerleri elde edebiliyoruz. Eğer bir sözlük içinde bulunmayan bir öğeye erişmeye çalışırsak Python bize **KeyError** tipinde bir hata mesajı verecektir.

Sözlüklerin Yapısı

Sözlük içinde hem sayıları hem karakter dizilerini hem de listeleri kullanabiliriz.

```
sözlük = {"sıfır": 0,
          "bir" : 1,
          "iki" : 2,}
```

```
sözlük = {"Ali Kral": ["Konya", "Asker", 34],
          "Alp Yağız" : ["Adana", "Polis", 40],
          "Seda Bal" : ["Hatay", "Doktor", 30]}
```

İstersek sözlükleri, içlerinde başka sözlükleri barındıracak şekilde de tanımlayabiliriz:.

```
kişiler = {"Ahmet Öz": {"Memleket": "İstanbul",
                        "Meslek" : "Öğretmen",
                        "Yaş" : 34},
           "Mehmet Yağız": {"Memleket": "Adana",
                            "Meslek" : "Mühendis",
                            "Yaş" : 40}}
isim = input("Bilgi edinmek istediğiniz kişinin adı: ")
ayrıntı = input("Hangisi seçin?(Memleket/Meslek/Yaş): ")
print(kişiler[isim][ayrıntı])
```

```
-Ahmet Öz-
yazdığımızı
varsayalım.
-Memleket-
yazdığımızı
varsayalım.

İstanbul
```

Sözlükteki öğeler açısından 'sıra' diye bir kavram yoktur. Örneğin;

```
sözlük = {'elma': 'apple',
          'armut': 'pear',
          'çilek': 'strawberry'}
print(sözlük[0])
```

```
KeyError: 0
```

Sözlüklere Öğe Eklemek

Sözlüğe öğe eklemek için şöyle bir formül kullanacağız: **sözlük[anahtar] = değer**

```
personel = {"Mehmet Öz": "AR-GE Müdürü",
            "Samet Söz": "Genel Direktör",
            "Sedat Gün": "Proje Müdürü"}
personel["Turgut Özben"] = "Mühendis"
print(personel["Turgut Özben"])
```

Mühendis

İstediğimiz öğe sözlüğe eklenmiş. Ancak bu öğenin sözlüğün en sonuna değil, sözlük içine **rastgele bir şekilde** yerleştirilmiştir. Çünkü, **sözlükler sırasız bir veri tipidir.**

```
sözlük = {}
sözlük = {'a': 1}
sözlük = {'a': (1,2,3)}
sözlük = {[1,23]: 'kardiz'}      TypeError: unhashable type: 'list'
sözlük = {'a': [1,2,3]}
```

Sözlükler değer olarak her türlü veri tipini kabul ediyor. Ama durum sözlük anahtarları açısından böyle değildir. Yani sözlüklere anahtar olarak her veri tipini atayamayız. Bir değerin 'anahtar' olabilmesi için, o öğenin **değiştirilemeyen (immutable)** bir veri tipi olması gerekir. **Dolayısıyla bir sözlüğe ancak şu veri tiplerini ekleyebiliriz: Demetler, Sayılar ve Karakter Dizileri.**

Sözlük Öğeleri Üzerinde Değişiklik Yapmak

```
notlar = {'Seda': 98, 'Ege': 95, 'Zeynep': 100, 'Ahmet': 45}
notlar["Ahmet"] = 88
print(notlar)
{'Seda': 98, 'Ege': 95, 'Zeynep': 100, 'Ahmet': 88}
```

Sözlük Üreteçleri (Dictionary Comprehensions)

Tıpkı liste üreteçlerinde olduğu gibi, sözlük üreteçleri sayesinde tek satırda ve hızlı bir şekilde sözlükler üretebiliriz.

```
harfler = 'abcçdevyz'
sözlük = {}
for i in harfler:
    sözlük[i] = harfler.index(i)
print(sözlük)
```

{'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4, 'e': 5, 'v': 6, 'y': 7, 'z': 8}

```
harfler = 'abcçdevyz'
sözlük = {}
for i in range(len(harfler)):
    sözlük[harfler[i]] = i
print(sözlük)
```

{'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4, 'e': 5, 'v': 6, 'y': 7, 'z': 8}

```
harfler = 'abcçdevyz'
sözlük = {}
sözlük = {i: harfler.index(i) for i in harfler}
print(sözlük
```

{'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4, 'e': 5, 'v': 6, 'y': 7, 'z': 8}

SÖZLÜK METOTLARI

keys() Metotu

Bir sözlüğü normal yollardan ekrana yazdırırsanız size hem anahtarları hem de bunlara karşılık gelen değerleri verecektir. Ama eğer bir sözlüğün sadece anahtarlarını almak isterseniz keys() metotundan yararlanabilirsiniz.

`sözlük = {"a": 0,`
`"b": 1,`
`"c": 2,`
`"d": 3}`
`print(sözlük.keys())`
	dict_keys([‘a’, ‘b’, ‘c’, ‘d’])

Bu nesneyi programınızda kullanabilmek için isterseniz, bunu listeye, demete veya karakter dizisine dönüştürebilirsiniz.

`S = {"a": 0,`
`"b": 1,`
`"c": 2,`
`"d": 3}`
`liste = list(s.keys())`
`print(liste)`
	`s = {"a": 0,`
	`"b": 1,`
	`"c": 2,`
	`"d": 3}`
	`demet = tuple(s.keys())`
	`print(demet)`
		`s = {"a": 0,`
		`"b": 1,`
		`"c": 2,`
		`"d": 3}`
		`kD="".join(s.keys())`
		`print(kD)`

[‘a’, ‘b’, ‘c’, ‘d’]	(‘a’, ‘b’, ‘c’, ‘d’)	abcd

Eğer sözlük anahtarlarını str() fonksiyonu yardımıyla karakter dizisine dönüştürmeye kalkışırsanız beklemediğiniz bir çıktı alırsınız bu yüzden join’i kullandık.

values() Metotu

Bir sözlüğün değerlerini values() metotu verir.

`sözlük = {'b': 1, 'c': 2, 'a': 0, 'd': 3}` `print(sözlük.values())`	dict_values([1, 2, 0, 3])

`sözlük = {'b': 1, 'c': 2, 'a': 0, 'd': 3}`
`liste = list(sözlük.values())`
`print(liste)`
	[1, 2, 0, 3]

`sözlük = {'b': 1, 'c': 2, 'a': 0, 'd': 3}`
`demet = tuple(sözlük.values())`
`print(demet)`
	(1, 2, 0, 3)

Bu verileri karakter dizisine dönüştürmeye çalıştığımızda hata alırız. Bunun sebebi, sözlükteki değerlerin int tipinde olmasıdır. Bildiğiniz gibi, sadece aynı tip verileri birbiriyle birleştirebiliriz. Eğer birleştirmek istediğimiz veriler birbirinden farklı tipte ise, bunları birleştirmeden önce bir dönüştürme işlemi yapmamız gerekir.

`sözlük = {'b': 1, 'c': 2, 'a': 0, 'd': 3}`
`kD = "".join([str(i) for i in sözlük.values()])`
`print(kD)`
	1203

items() Metotu

Bu metot, bir sözlüğün hem anahtarlarını hem de değerlerini aynı anda almamızı sağlar.

`sözlük = {'b': 1, 'c': 2, 'a': 0, 'd': 3}` `print(sözlük.items())`	dict_items([(‘b’, 1), (‘c’, 2), (‘a’, 0), (‘d’, 3)])

get() Metotu

Sözlüklerin get() adlı metotu, parantez içinde iki adet argüman alır. Birinci argüman sorgulamak istediğimiz sözlük öğesidir. İkinci argüman ise bu öğenin sözlükte bulunmadığı durumda kullanıcıya hangi mesajın gösterileceğini belirtir.

```
soru = input("Şehrinizin adını yazın: ")
cevap = {"istanbul": "gök gürültülü ve sağanak yağışlı",
         "ankara": "açık ve güneşli", "izmir": "bulutlu"}
print(cevap.get(soru, "Bu şehre ilişkin bilgi bulunmamaktadır."))
```

– istanbul – yazdığımızı varsayalım.
gök gürültülü ve sağanak yağışlı

clear() Metotu

Görevi, sözlük **içinde yer alan** bütün öğeleri temizlemektir.

```
lig = {'şampiyon': 'Adana',
       'ikinci': 'Mersin ',
       'üçüncü': 'Hatay'}
lig.clear()
print(lig)
```

{}

Ama tabii ki bu şekilde sözlüğü silmiş olmadık. Boş da olsa bellekte hala lig adlı bir sözlük duruyor. Eğer siz lig'i ortadan kaldırmak isterseniz **"del"** adlı bir parçacıktan yararlanmanız gerekir:

```
lig = {'şampiyon': 'Adana',
       'ikinci': 'Mersin ',
       'üçüncü': 'Hatay'}
del lig
print(lig)
```

NameError: name 'lig' is not defined

copy() Metotu

```
hava_durumu = {"İstanbul": "Kar",
               "Adana": "Güneş",}
yedek_hava_durumu = hava_durumu
print(hava_durumu)
print(yedek_hava_durumu)
hava_durumu["Mersin"] = "Sis"
print(hava_durumu)
print(yedek_hava_durumu)
```

{'İstanbul': 'Kar', 'Adana': 'Güneş'}
{'İstanbul': 'Kar', 'Adana': 'Güneş'}
{'İstanbul': 'Kar', 'Adana': 'Güneş', 'Mersin': 'Sis'}
{'İstanbul': 'Kar', 'Adana': 'Güneş', 'Mersin': 'Sis'}

Varolan bir sözlüğü veya listeyi başka bir değişkene atadığımızda aslında yaptığımız şey bir kopyalama işleminden ziyade bellekteki aynı nesneye gönderme yapan iki farklı isim belirlemekten ibaret.

copy() metotunu kullanarak varolan bir sözlüğü gerçek anlamda kopyalayabilir, yani yedekleyebiliriz.

```
hava_durumu = {"İstanbul": "Kar",
               "Adana": "Güneş",}
yedek_hava_durumu =
hava_durumu.copy()
print(hava_durumu)
print(yedek_hava_durumu)
hava_durumu["Mersin"] = "Sis"
print(hava_durumu)
print(yedek_hava_durumu)
```

{'İstanbul': 'Kar', 'Adana': 'Güneş'}
{'İstanbul': 'Kar', 'Adana': 'Güneş'}
{'İstanbul': 'Kar', 'Adana': 'Güneş', 'Mersin': 'Sis'}
{'İstanbul': 'Kar', 'Adana': 'Güneş'

fromkeys() Metotu

fromkeys()'in görevi yeni bir sözlük oluşturmaktır.

```
elemanlar = "Ahmet", "Mehmet", "Can"
adresler = dict.fromkeys(elemanlar, "Kadıköy")
print(adresler)
```

{'Ahmet': 'Kadıköy', 'Mehmet': 'Kadıköy', 'Can': 'Kadıköy'}

Gördüğünüz gibi öncelikle "elemanlar" adlı bir demet tanımladık. Daha sonra da "adresler" adlı bir sözlük tanımlayarak, fromkeys() metotu yardımıyla anahtar olarak "elemanlar" demetindeki öğelerden oluşan, değer olarak ise Kadıköy'ü içeren bir sözlük meydana getirdik.

pop() Metotu

Normalde pop() metotunu index numarası belirtmeden kullanabiliriz ama burada pop() metotunu argümansız bir şekilde kullanamıyoruz.

```
sepet = {"meyveler": ("elma", "armut"),
         "sebzeler": ("pırasa", "fasulye"),
         "içecekler": ("su", "kola", "ayran")}
sepet.pop("meyveler")
print(sepet)
```

{'sebzeler': ('pırasa', 'fasulye'), 'içecekler': ('su', 'kola', 'ayran')}

popitem() Metotu

Bu metot bir sözlükten rastgele öğeler silmek için kullanılır. pop() metotu parantez içinde bir parametre alırken, popitem() metotunun parantezi boş, yani parametresiz olarak kullanılır.

3.6 ve sonraki sürümlerde son item sözlükten çıkarılır. 3.6'dan önceki sürümlerde rastgele item çıkarılır.

setdefault() Metotu

Bu metot yardımıyla bir sözlük içinde arama yapabiliyor, eğer aradığımız anahtar sözlükte yoksa, setdefault() metotu içinde belirttiğimiz özellikleri taşıyan yeni bir anahtar-değer çifti oluşturabiliyoruz.

```
sepet = {"meyveler": ("elma", "armut"),
         "sebzeler": ("pırasa", "fasulye")}
sepet.setdefault("içecekler", ("su",
"kola"))
print(sepet)
```

{'meyveler': ('elma', 'armut'), 'sebzeler': ('pırasa', 'fasulye'), 'içecekler': ('su', 'kola')}

update() Metotu

Bu metot yardımıyla oluşturduğumuz sözlükleri yeni verilerle güncelleyeceğiz.

```
stok = {"armut": 10,
        "peynir": 6,
        "sosis": 15}
yeni_stok = {"armut": 20,
             "peynir": 8,
             "sosis": 4,
             "sucuk": 6}
stok.update(yeni_stok)
print(stok)
```

{'armut': 20, 'peynir': 8, 'sosis': 4, 'sucuk': 6}

SETS (KÜMELER)

Bu veri tipi, matematikteki kümelerin sahip olduğu bütün özellikleri taşır. Yani matematikteki kümelerden bildiğimiz kesişim, birleşim ve fark gibi özellikler Python'daki kümeler için de geçerlidir.

Listeler, demetler ve sözlüklerin aksine kümelerin ayırt edici bir işareti yoktur.

Kümeler sıralı (ordered) yapıda olmadıkları için indeks işlemleri kullanılamaz.

Küme oluşturmak için **set()** adlı özel bir fonksiyondan yararlanıyoruz.

Kümeler de, tıpkı listeler ve sözlükler gibi, **değiştirilebilir (mutable)** bir veri tipidir. Boş bir kümeyi şöyle oluşturuyoruz:

```
boş_küme = set()
print(boş_küme)
```

set()

```
küme = set(["elma", "armut", "kebap"])
```

Dikkat ederseniz, küme oluştururken listelerden faydalandık. Gördüğünüz gibi set() fonksiyonu içindeki öğeler bir liste içinde yer alıyor. Dolayısıyla yukarıdaki tanımlamayı şöyle de yapabiliriz:

```
liste = ["elma", "armut", "kebap"]
küme = set(liste)
```

İstersek demetleri, karakter dizilerini ve sözlükleri küme haline getirebiliriz ancak sayılardan küme oluşturamayız.

```
demet =("a", "b")
küme = set(demet)
```

```
mesaj = "Python"
küme = set(mesaj)
```

```
bilgi =
{"işletim": "GNU",
"sistem": "Linux",}
küme = set(bilgi)
```

```
n = 10
küme = set(n)
```

TypeError: 'int' object is not iterable

```
bilgi = {"işletim sistemi": "GNU",
         "sistem çekirdeği": "Linux",}
küme = set(bilgi)
liste = [(anahtar, değer) for anahtar, değer in bilgi.items()]
küme = set(liste)
```

Bir sözlüğü kümeye çevirdiğinizde, sözlüğün yalnızca anahtarları kümeye eklenecektir. Eğer bir sözlüğü kümeye çevirirken hem anahtarları hem de değerleri korumak istiyorsak yukardakini yazarız.

```
küme = {'Python', 'C++', 'PHP'}
print(type(küme))
```

DOĞRU

<class 'set'>

```
küme = {}
```

YANLIŞ

Aslında sözlüklerin ayırt edici işareti olan süslü parantezleri kullanarak ve öğeleri birbirinden virgülle ayırarak küme veri tipini elde edebiliriz. Ancak bu yapıyı kullanarak boş bir küme oluşturamayız.

<pre>kume1 = {1,9,0,5}
kume2 = {2,1,8,5}
print(kume1 & kume2)   #Kesişim
print(5 in kume1)   #Üyelik
print(9 in kume2)   #Üyelik
print(kume1 | kume2)   #Birleşim
print(kume1 - kume2)   #Fark</pre>
	{1, 5}
	True
	False
	{0, 1, 2, 5, 8, 9}
	{0, 9}

<pre>liste1 = [1,9,0,5]
liste2 = [2,1,8,5]
kesisim = list()
for i in liste1:
    if i in liste2:
        kesisim.append(i)
print(kesisim)</pre>[1, 5]
	<pre>kume1 = {1,9,0,5}
	kume2 = {2,1,8,5}
	birlesim = kume1.copy()
	for i in kume2:
	    if i not in kume1:
	        birlesim.append(i)
	print(birlesim)</pre>[1, 9, 0, 5, 2, 8]

Kümelerin Yapısı

```
mesaj = "Python Programlama"
küme = set(mesaj)
print(küme)
```

{'h', 'P', 'o', 'l', 'g', ' ', 'a', 'r', 't', 'n', 'y', 'm'}

Bir öğeyi küme olarak tanımlayıp ekrana yazdırdığımızda elde ettiğimiz çıktı, o öğe içindeki her bir alt öğeyi tek bir kez içeriyor. Yani mesela "Python Programlama" içinde iki adet "P" karakteri var, ama çıktıda bu iki "P" karakterinin yalnızca biri görünüyor. Buradan anlıyoruz ki, **kümeler aynı öğeyi birden fazla kez tekrar etmez.** Aynı durum karakter dizisi dışında kalan öteki veri tipleri için de geçerlidir.

Kümelerin önemli bir özelliği de, tıpkı sözlükler gibi, herhangi bir şekilde "öğe sırası" kavramına sahip olmamasıdır. Bu durumun bir yansıması olarak, küme öğelerine sıralarına göre de erişemezsiniz.

Küme Üreteçleri (Set Comprehensions)

Küme üreteçlerini kullanarak tek satırda ve hızlı bir şekilde kümeler de üretebiliriz.

Birbirinin aynı olmayan sayılardan oluşmuş bir listeyi elde etmek istiyorsanız küme üreteçlerini kullanabiliriz.

```
import random
liste = [random.randint(0, 10000) for i in range(1000)]
küme = {i for i in liste if i < 100}
print(küme)
```

{32, 1, 33, 37, 71, 75, 15, 80, 49, 53, 21, 88}

NOT: Küme elemanları aynı verileri ekrana yazdırmaz. !!

```
liste = ["elma", "armut", "kebap", "elma"]
küme = set(liste)
print(küme)
```

{'kebap', 'elma', 'armut'}

KÜMELERİN METOTLARI

clear() Metotu

Bu metotu kullanarak kümenin içindeki bütün öğeleri sileriz.

copy() Metotu

Listeler ve sözlükleri incelerken copy() adlı bir metot öğrenmiştik. Bu metot aynı zamanda kümelerle birlikte de kullanılabilir. Üstelik işlevi de aynıdır.

add() Metotu

Bu metot yardımıyla kümelerimize yeni öğeler ilave edebileceğiz. Eğer kümede zaten varolan bir öğe eklemeye çalışırsak kümede herhangi bir değişiklik olmayacaktır. Ayrıca bir kümeye herhangi bir öğe ekleyebilmemiz için, o öğenin değiştirilemeyen (immutable) bir veri tipi olması gerekiyor.

Küme metotları ile sadece kümenin kendisi değişir. Küme elemanları immutable oldukları için değiştirilemez.

`39üme = {1,9,0,5}`
`39üme.add('a')`
`print(39üme)`
`39üme.add[6,7]`
	{0, 1, 5, 9, 'a'}
	TypeError: 'builtin_function_or_method' object is not subscriptable

difference() Metotu

Bu metot iki kümenin farkını almamızı sağlar. İsterseniz uzun uzun difference() metotunu kullanmak yerine sadece eksi (-) işaretini kullanarak da aynı sonucu elde edebilirsiniz.

`K1 = set([1, 2, 3, 5])`
`k2 = set([3, 4, 2, 10])`
`print(k1.difference(k2))`
`print(k1 - k2)`
	{1, 5}
	{1, 5}

difference_update() Metotu

Bu metot, difference() metotundan elde edilen sonuca göre bir kümenin güncellenmesini sağlar.

`K1 = set([1, 2, 3])`
`k2 = set([1, 3, 5])`
`k1.difference_update(k2)`
`print(k1)`
	{2}

Gördüğünüz gibi, bu metot k1'in k2'den farkını aldı ve bu farkı kullanarak k1'i yeniden oluşturdu.

discard() Metotu

discard() metotu kümeden öğe silmemizi sağlar.

`Hayvanlar = set(["kedi", "inek", "deve"])`
`hayvanlar.discard("kedi")`
`print(hayvanlar)`
`hayvanlar.discard("yılan")`
	{'deve', 'inek'}

Bu metotun en önemli özelliği olmayan bir öğeyi silmeye çalıştığımızda hata vermiyor olması.

remove() Metotu

Eğer bir kümeden öğe silmek istersek remove() metotunu da kullanabiliriz. **Eğer remove() metotunu kullanarak, kümede olmayan bir öğeyi silmeye çalışırsak, discard() metotunun aksine, hata mesajı alırız.**

intersection() Metotu

intersection() metotu bize iki kümenin kesişim kümesini verecektir. İki kümenin kesişimini bulmak için "&" işaretinden de yararlanabiliriz.

```
K1 = set([1, 2, 3, 4])
k2 = set([1, 3, 5, 7])
print(k1.intersection(k2))
print(k1&k2)
```

```
{1, 3}
{1, 3}
```

intersection_update() Metotu

intersection() metotundan elde edilen sonuca göre kümenin güncellenmesini sağlamaktır.

```
K1 = set([1, 2, 3])
k2 = set([1, 3, 5])
k1.intersection_update(k2)
print(k1)
print(k2)
```

```
{1, 3}
{1, 3, 5}
```

isdisjoint() Metotu

isdisjoint() metotunu kullanarak iki kümenin kesişim kümesinin boş olup olmadığı sorgulayabiliriz.

issubset() Metotu

Bu metot yardımıyla, bir kümenin bütün elemanlarının başka bir küme içinde yer alıp yer almadığını sorgulayabiliriz.

```
A = set([1, 2, 3])
b = set([0, 1, 2, 3, 4, 5])
print(a.issubset(b))
```

```
True
```

issuperset() Metotu

```
a = set([1, 2, 3])
b = set([0, 1, 2, 3, 4, 5])
print(b.issuperset(a))
```

```
True
```

Burada ise, "b kümesi a kümesini kapsar," sonucunu elde ediyoruz. Yani b kümesi a kümesinin bütün elemanlarını içinde barındırıyor.

update() Metotu

Bir kümeyi, başka bir küme ile birleştirerek güncelleme işlemi yapar.

```
Küme = set(["elma", "kebap"])
yeni = [1, 2, 3]
küme.update(yeni)
print(küme)
```

```
{1, 2, 3, 'kebap', 'elma', 'armut'}
```

union() Metotu

union() metotu iki kümenin birleşimini almamızı sağlar. union() metotu yerine “|” işaretini de kullanabiliriz.

```
a = set([2, 4, 6, 8])
b = set([1, 3, 5, 7])
print(a.union(b))
print(a|b)
```

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

symmetric_difference() Metotu

Bu metot kümelerin ikisinde de bulunan öğeleri aynı anda almamızı sağlar.

```
a = set([1, 2, 5])
b = set([1, 4, 5])
print(a.symmetric_difference(b))
```

{2, 4}

symmetric_difference_update() Metotu

```
a = set([1,2, 5])
b = set([1,4, 5])
a.symmetric_difference_update(b)
print(a)
```

{2, 4}

Gördüğünüz gibi, a kümesinin eski öğeleri gitti, yerlerine symmetric_difference() metotuyla elde edilen çıktı geldi. Yani a kümesi, symmetric_difference() metotunun sonucuna göre güncellenmiş oldu.

pop() Metotu

Herhangi bir elemanı rasgele olarak kümeden çıkarır ve ekrana yazdırır.

Dondurulmuş Kümeler (Frozenset)

Eğer öğeleri üzerinde değiştirilemeyen **(immutable)** bir küme oluşturmak isterseniz set() yerine frozenset() fonksiyonunu kullanabiliriz.

FONKSİYONLAR

Fonksiyonlar, karmaşık işlemleri bir araya toplayarak, bu işlemleri tek adımda yapmamızı sağlamaktır.

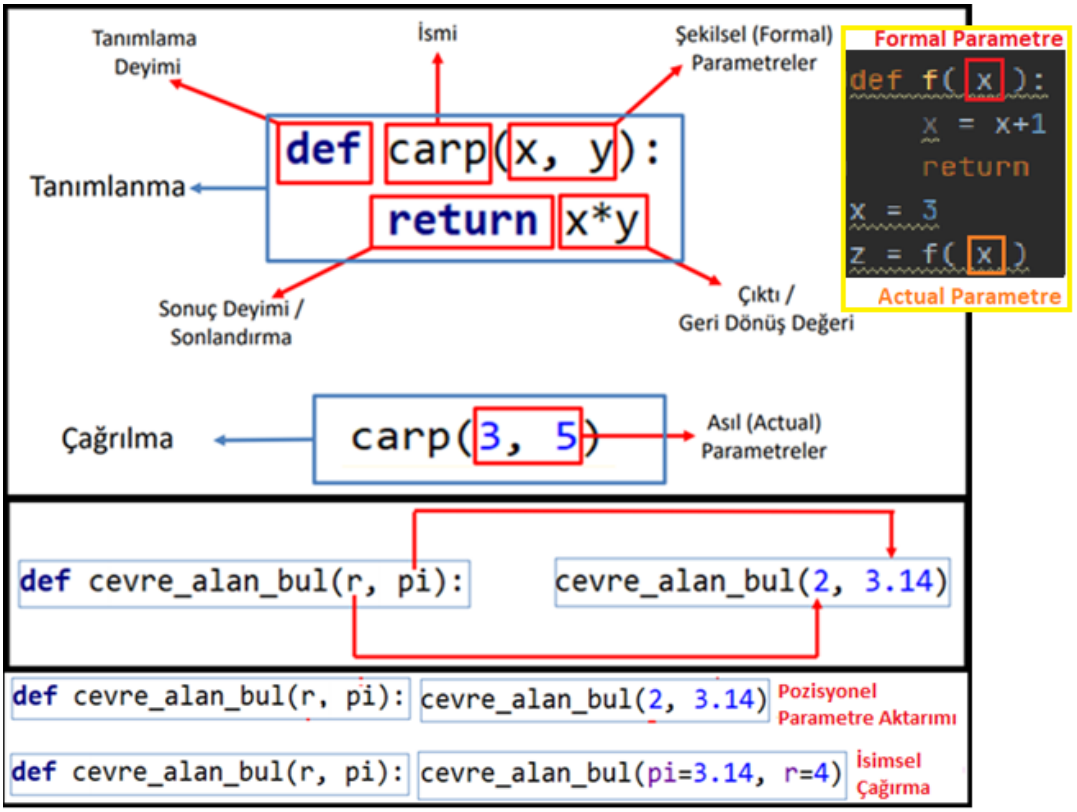
1. Python'da 4 tip fonksiyon bulunur. Bunlar **gömülü fonksiyonlar**, **kullanıcı tanımlı fonksiyonlar, lambda** ve **özyinelemeli (recursive)**.
2. Gömülü fonksiyonlar; Python geliştiricileri tarafından tanımlanıp dilin içine gömülmüş olan print(), open(), type(), str(), int() vb. fonksiyonlardır.
3. Fonksiyon tanımlamak için **def** adlı bir ifadeden yararlanıyoruz. Bu ifadeden sonra, tanımlayacağımız fonksiyonun adını belirleyip iki nokta üst üste işareti koyuyoruz. İki nokta üst üste işaretinden sonra gelen satırlar girintili olarak yazılıyor.girintili olarak yazdığımız bütün kodlar **fonksiyonun gövdesini** oluşturur. Girintinin dışına çıkıldığı anda fonksiyon tanımı da sona erer.

Fonksiyonun Avantajları	**İyi Bir Programlama**
- Diğer programlarda da kullanılabilen kodlar elde ederiz. - Kodun gereksiz yere büyümesini engeller. - Okunabilirliği arttırarak algılamayı ve debug yapmayı kolaylaştırır.	- Fazla kod satırı iyi bir şey değildir. - Fonksiyonları tanıtmalıyız.

Abstraction, arka planda ne olduğunu bilmeden kullandıklarımıza denir. **(Soyutlama Durumu)**
Decomposition, küçük fonksiyonların birleştirilerek büyütülmesine denir. **(Ayrıştırma Durumu)**

Python'da her fonksiyonun;	
- Bir ismi	- Parametresi (0 veya daha fazla)
- Fonksiyon belge dizisi (docstring)	- Kod bölümü (body)
- Çıktısı (return) bulunmaktadır.	

Fonksiyon Tanımlamak ve Çağırmak



Fonksiyonun yaşam döngüsü iki aşamadan oluşur: **Fonksiyon tanımı** ve **fonksiyon çağrısı.**

Bir fonksiyonu tanımlarken belirlediğimiz adlara **parametre** (diğer bir deyişle fonksiyonların girdilerine parametre denir), aynı fonksiyonu çağırırken belirlediğimiz adlara ise **argüman** deniyor.

```
def kopyala(kaynak_dosya, hedef_dizin):
    çıktı = "{} adlı dosya {} adlı dizin içine kopyalandı!"
    print(çıktı.format(kaynak_dosya, hedef_dizin))
kopyala("deneme.txt", "/Desktop")
```

deneme.txt adlı dosya /Desktop adlı dizin içine kopyalandı!

Burada gördüğünüz "deneme.txt" ve "/Desktop" değerlerine **argüman,** "kaynak_dosya" ve "hedef_dizin" değerlerine **parametre** adı verilir.

Parametrelerin sırası büyük önem taşır, veriliş sırası önem taşıyan bu tür parametrelere **"sıralı parametreler"** veya **"isimsiz parametreler"** adı verilir.
Parametreleri isimleri ile birlikte kullanmaya **"isimli parametreler"** adı verilir.

```
kayıt_yap(soyisim="Öz", isim="Ahmet", işsiz="Debian", şehir= "Ankara")
```

Ancak burada dikkat etmemiz gereken bir nokta var. Python'da isimli bir parametrenin ardından sıralı bir parametre gelemez. Yani şu kullanım yanlıştır:

```
kayıt_yap(soyisim="Öz", isim="Ahmet", "Debian", "Ankara")
```

Varsayılan Değerli Parametreler

```
def alan_cevre(pi=3, r=4):
    alan = pi*r**2
    cevre = 2*pi*r
    return alan, cevre
print(alan_cevre())
print(alan_cevre(4))
```

(48, 24)
(64, 32)

Gördüğünüz gibi, alan_cevre() fonksiyonunun pi ve r adlı iki parametresi var. Biz fonksiyonu tanımlarken, bu parametreye bir varsayılan değer atadık (pi=3, r=4). Böylece alan_cevre() fonksiyonu parametresiz olarak çağrıldığında (48, 24) sonucunu bize üretti. Ama tek bir paramere atadığımızda (4) pi'nin yerini tuttu, r'nin değeri sabit kaldı ve sonuç olarak bize (64, 32)'yi üretti.

Rastgele Sayıda İsimsiz Parametre Belirleme (*args)

```
def topla(*sayilar):
    print(sayilar)
    toplam = 0
    for i in sayilar:
        toplam += i
    return toplam
print("Toplam: {}".format(topla(1,2,0,2)))
```

(1, 2, 0, 2)
Toplam: 5

Gördüğünüz gibi, fonksiyon tanımı içinde kullandığımız * işareti sayesinde fonksiyonumuzun pratik olarak sınırsız sayıda parametre kabul etmesini sağlayabiliyoruz. Bu arada, bu tür fonksiyonların alabileceği parametre sayısı pratikte sınırsızdır ama teknik olarak bu sayı 256 adedi geçemez.

Yukarıdaki kodların verdiği çıktının bir **tuple (demet)** veri tipi olduğuna dikkatinizi çekmek isterim.

Rastgele Sayıda İsimli Parametre Belirleme (**kwargs)

```
def listele(**personel):
    print(personel)
    for key, value in personel.items():
        print("{} : {}".format(key, value))
listele(Ad="Kamber", Soyad="Ata")
```

```
{'Ad': 'Kamber', 'Soyad': 'Ata'}
Ad : Kamber
Soyad : Ata
```

Gördüğünüz gibi, fonksiyon tanımı içinde kullandığımız ** işareti sayesinde fonksiyonumuzun pratik olarak sınırsız sayıda parametre kabul etmesini sağlayabiliyoruz.

Yukarıdaki kodların verdiği çıktının bir **dict (sözlük)** veri tipi olduğuna dikkatinizi çekmek isterim.

```
def myFunc(a,b,c,*args,**kwargs):
    print(a)
    print(b)
    print(c)
    print(args)
    print(kwargs)

myFunc(10,20,30,40,50,60,key1="value 1", key2 = "value 2")
```

```
10
20
30
(40, 50, 60)
{'key1': 'value 1', 'key2': 'value 2'
```

Fonksiyon Belge Dizisi (Docstring)

Fonksiyon içindeki yorum satırında ulaşmak için **__doc__** komutu kullanılır.

```
def cift(x):
    """
    sayi çift mi
    hayır
    """
    return x%2==0
print(cift.__doc__)
```

```
    sayi çift mi
    hayır
```

return Deyimi

Python'da her fonksiyonun bir dönüş değeri vardır. Burada döndürmekten kastımız, bir işlemin sonucu olarak ortaya çıkan değeri vermektir. Mesela "Bu fonksiyonun dönüş değeri bir karakter dizisidir." veya "Bu fonksiyon bir karakter dizisi döndürür." dediğimiz zaman kastettiğimiz şey, bu fonksiyonun çalışması sonucu ortaya çıkan değerin bir karakter dizisi olduğudur.

```
def cift_mi(x):
    x % 2 == 0
print(cift_mi(4))
```

None

Eğer bir fonksiyon return ifadesi içermiyorsa, Python **None** değerini döndürür.

```
def fonk(n):
    if n < 0:
        return 'Negatif sayı!'
    else:
        return 'Pozitif sayı!'
        print("Bu satır gözükmez")
print(fonk(5))
```

Pozitif sayı!

return deyimini kullandığınız satırdan sonra gelen hiçbir kod çalışmaz.

Bir fonksiyon birden çok ara sonuç üretebilir ve bunların ayrı ayrı geri döndürülmesi gerekebilir.

```
def islem(a,b):
    toplam=a+b
    carpim=a*b
    return toplam, carpim
print(islem(3,5))
```

(8, 15)

Kapsam - Yaşam Süresi - global ve lokal Deyimi – Namespace

Herhangi bir kod bölgesinde (bu bir fonksiyonun içindeki değişkenler olabilir, if-for blokunun içindeki değişkenler olabilir) geçerli olan değişkenlerin tutulduğu alana **"isim alanı (namespace)"** denir.

Bir değişkenin kullanılabilir, erişilebilir ve görünür olduğu (visible), yere **"kapsam"** denir.

Bir değişkenin bellek bölgesine bağlanması ve ayrılması arasındaki geçen süreye **"yaşam süresi (lifetime)"** denir. return fonksiyonu sayesinde ya da fonksiyonun bitmesiyle yaşam alanı sona erer.

Her bir fonksiyonun ya da her bir blokun (if, else, for gibi) arasında yer alan değişkenlere **"lokal"** değişkenler adı verilir ve bu değişkenlere fonksiyon dışından erişilemezler. Bunun nedeni ise lokal değişkenlerin **yaşam süresi** fonksiyon dışında henüz başlatılmamış olmasıdır.

Koddaki bütün fonksiyonların ve bütün blokların (if, else, for gibi) dışında yer alan değişkenlere **"global"** değişkenler adı verilir. global anahtar kelimesi, lokal alanda tanımlanan (fonksiyonun içi ya da herhangi bir blokun içi) bir değişkene global alanda (fonksiyonun dışından) erişebilmek amacıyla kullanılır.

nonlocal komutu sayesinde fonksiyonun ya da blokun bir üst seviyede (bir üst fonksyionda, bir üst bloklar arasında) varlığının olup olmadığını kontrol eder. **global** komutu sayesinde de en dış katmanda yani global'de varlığının olup olmadığını kontrol eder.

İç İçe Fonksiyonlar Neden Kullanılır ? (Nested Functions)

1- Sarma / Kuşatma (Encapsulation) : İç içe fonksiyonlar, fonksiyon dışında gerçekleşen her şeyden korunurlar, yani global scope'dan gizlenirler.
2- Kendini tekrar etmeden korur.
3- Kapatma ve Fabrika Fonksiyonları (Closure and Factory Functions)

İç İçe Fonksiyonlarda Kapsam

Python statik kapsama uygun olarak çalışır. Statik kapsam, gördüğü değişkeni ilk olarak bulunduğu mevcut namespace içinde arar. Bulamazsa bir üstteki namespace içinde arar ve böylece global namespace'e kadar ilerler. Hiçbir namespace içinde değişkeni bulamaz ise hata verir.

```
x = 0
def sayi(a):
    print(x+1)
sayi(x)
print(x)
```

1
0

print() ile x'in değerini sorguladığımızda Python öncelikle sayi() adlı fonksiyonun isim alanına baktı. Orada x'i bulamayınca bu kez **global** alana yönelip, orada bulduğu x'in değerini yazdırdı.

```
x = 1
def outer():
    x = 3
    def inner():
        print("Inner: ", x)
    inner()
    print("Outer: ",x)
outer()
print("Global: ",x)
```

Inner: 3
Outer: 3
Global: 1

Sağdaki kod bloğuna bakacak olursak; 3. satırda yer alan x=3 sayesinde x'in yaşam alanı başladı, 8. satırda yer alan outer() sayesinde x'in yaşam alanı sona ermiş oldu.

```
liste = [1,2,3,4]
print("Listenin İlk Hali: ", liste)
def ekle_liste():
    liste = []
    liste.append(5)
    print("Fonksiyon İçi =", liste)
    return liste
ekle_liste()
print("Fonksiyon Dışı =", liste)
```

Listenin İlk Hali: [1, 2, 3, 4]
Fonksiyon İçi = [5]
Fonksiyon Dışı = [1, 2, 3, 4]

```
liste = [1,2,3,4]
print("Listenin İlk Hali: ", liste)
def ekle_liste():
    #liste = []
    liste.append(5)
    print("Fonksiyon İçi =", liste)
    return liste
ekle_liste()
print("Fonksiyon Dışı =", liste)
```

Listenin İlk Hali: [1, 2, 3, 4]
Fonksiyon İçi = [1, 2, 3, 4, 5]
Fonksiyon Dışı = [1, 2, 3, 4, 5]

```
liste = [1,2,3,4]
def ekle_liste():
    liste += [5]
    print("Fonks. İçi = ", liste)
    return liste
ekle_liste()
print("Fonksiyon Dışı = ", liste)
```

UnboundLocalError: local variable 'liste' referenced before assignment

```
liste = [1,2,3,4]
def ekle_liste():
    global liste
    liste += [5]
    print("Fonks. İçi = ", liste)
    return liste
ekle_liste()
print("Fonks. Dışı = ", liste
```

Fonks. İçi = [1, 2, 3, 4, 5]
Fonks. Dışı = [1, 2, 3, 4, 5]

```
name = 'global string'
def greeting():
    name = 'Çınar'
    def hello():
        name = 'Ada'
        print(name)
    hello()
greeting()
```

Ada

```
name = 'global string'
def greeting():
   #name = 'Çınar'
    def hello():
       #name = 'Ada'
        print(name)
    hello()
greeting()
```

global string

Kodu incelediğimizde 3 farklı name değişkeninin olduğunu görüyoruz. hello() fonksiyonun kapsamı dahilinde (yani hello() fonksiyonunun içinde) name değişkeni olduğu için Ada kelimesine ulaştık. Eğer hello() fonksiyonunun kapsamı dahilinde name isminde bir değişken olmasaydı bir üst kategoriye (seviyeye) yani greeting() isimle fonksiyona bakacaktık eğer ordada name isminde bir değişken olmasaydı global'e yani en dış katmana bakmamız gerekecekti.

```
isim = 'Kamber'
def outer():
    global isim
    isim = 'Helin'
    print(isim)          # 1 Helin
    def inner():
        global isim
        isim = 'Melisa'
        print(isim)      # 2 Melisa
    inner()
    return isim
print(outer())           # 3 Melisa
print(isim)              # 4 Melisa
```

```
isim = 'Kamber'
def fon():
   #isim = "Helin" !!! Hata verir.
    global isim
    isim = "Helin"
    isim = isim + ' ATA'
    return isim
print(fon())
```

Helin ATA

Fonksiyonlar iç içe değilse yani aynı seviyeyse ya da bloklar aynı seviyeyse diğer bir deyişle biri diğerini kapsamıyorsa bir üst bloka ya da fonksiyona bakmamız lazım.

Aşağıdaki örnekte phone() ve hello() fonksiyonları aynı seviye yani biri diğerinin içinde değil ya da biri diğerini kapsamıyor. Bundan dolayı phone() fonksiyonu bir üst seviyeye yani greeting() isimli fonksiyona baktı ve oradaki name isimli değişkeni aldı.

```
name = 'global string'
def greeting():
    name = 'Çınar'
    def hello():
        name = 'Kıbrıs'
        print(name)    # Kıbrıs    5
    print(name)        # Çınar     2
    def phone():
        print(name)    # Çınar     4
    print(name)        # Çınar     3
    phone()
    hello()
print(name)       # global string 1 !!
greeting()
```

global string
Çınar
Çınar
Çınar
Kıbrıs

```
name = 'global string'
def greeting():
    name = 'Çınar'
    def hello():
        name = 'Kıbrıs'
        print(name)      # Kıbrıs 2
    hello()
    print(name)          # Çınar  3
    name = 'Kilise'
    def phone():
        name = 'Mavi'
        print(name)      # Mavi   5
    print(name)          # Kilise 4
    phone()
print(name)      # global string  1
greeting()
```

global string
Çınar
Kıbrıs
Çınar
Kilise
Mavi

```
def outer():
    global a
    a = 20
    print("a: ",a)
    def inner():
        global a
        a = 30
        print("b: ", a)
    inner()                 # 2
    print("c: ", a)         # 3
outer()                     # 1
print("d: ", a)             # 4 !!!!!!
```

a: 20
b: 30
c: 30
d: 30

```
a = 50
def outer():
    global a
    print("a: ", a)
    def inner():
        global a
        a = 30
        print("b: ", a)
        def ininner():
            global a
            a = 40
            print("c: ", a)
        ininner()               # 3
        print("d: ", a)         # 4
    inner()                     # 2
    print("e: ", a)             # 5
outer()                         # 1
print("f: ", a)                 # 6
```

a: 50
b: 30
c: 40
d: 40
e: 40
f: 40

Eğer en dış katmanda a adında bir değişken olmasaydı program hata verecekti.

```
x = 50
def outer():
    x = 10
    print("outer I: ",x)
    def inner():
        nonlocal x
        x = x+1
        print("inner: ",x)
        def fear():
            nonlocal x
            x = 20
            print("fear: ",x)
            def factor():
                nonlocal x
                print("factor: ",x)
            factor()
        fear()
    inner()
    print("outer II: ",x)
outer()
print("global: ",x)
```

outer I: 10
inner: 11
fear: 20
factor: 20
outer II: 20
global: 50

```
def outer_function():
    global a
    a = 20
    def inner_function():
        global a
        a = 30
        print("a = ", a)
    inner_function()
    print("a = ", a)
a = 10
outer_function()
print("a = ", a)
```

a = 30
a = 30
a = 30

```
def external():
    x = 10
    def internal():
        global x
        x += 1
        print(x)
    internal()
external()
```

x += 1 NameError: name 'x' is not defined
(Global'de yani en dış katmanda x değişkeni yok.)

```
def func1():
    x = 3
    def func2():
        print("x1=", x)
        y = 4
        def func3():
            nonlocal x
            print("x2=", x)
            print("y1=", y)
            z = 5
            print("z1=", z)
            x = 10
        func3()
    func2()
    print("x3=",x)
func1()
```

x1= 3
x2= 3
y1= 4
z1= 5
x3= 10

```
def func1():
    x = 3
    def func2():
        print("x1=", x)
        y = 4
        def func3():
            nonlocal x
            print("x2=", x)
            print("y1=", y)
            z = 5
            print("z1=", z)
            x = 30
            def func4():
                global y
                y = 40
                print("y2=", y)
                def func5():
                    print("x3=", x)
                    print("y3=", y)
                    print("z2=", z)
                func5()
            func4()
        func3()
        def func6():
            global x
            x = 300
            print("x4=", x)
        func6()
    func2()
    print("x5=", x)
func1()
print("x6=", x)
print("y4=", y)
#print("z3=", z)
```

!!! Hata verir.

x1= 3
x2= 3
y1= 4
z1= 5
y2= 40
x3= 30
y3= 40
z2= 5
x4= 300
x5= 30
x6= 300
y4= 40

Lambda Fonksiyoları

lambda, fonksiyon tanımlamamızı sağlayacak, tıpkı **def** gibi bir ifadedir.

lambda fonksiyonlarını, bir fonksiyonun işlevselliğine ihtiyaç duyduğumuz, ama konum olarak bir fonksiyon tanımlayamayacağımız veya fonksiyon tanımlamanın zor ya da meşakkatli olduğu durumlarda kullanabiliriz.

```
def fonk(sayi1, sayi2):
    return sayi1 + sayi2
print(fonk(2,4))
```

6

```
fonk = lambda sayi1, sayi2: sayi1 + sayi2
print(fonk(3,4))
```

7

```
def çift_mi(sayı):
    return sayı % 2 == 0
print(çift_mi(100))
```

True

```
çift_mi = lambda sayı: sayı % 2 == 0
print(çift_mi(100))
```

True

Sözün özü:

```
lambda PARAMETRE : RETURN
lambda x: x + 10
```

Türkçede şu anlama gelir:
'x' adlı bir parametre alan bir lambda fonksiyonu tanımla. Bu fonksiyon, bu 'x parametresine 10 sayısını eklesin.

Özyinelemeli (Recursive) Fonksiyonlar

Özyinelemli fonksiyonların amacı; büyük bir problemin çözülebilmesi için o problemin, problemin bütününü temsil eden daha küçük bir parçası üzerinde işlem yapabilmemizi sağlayan fonksiyonlardır.

Fibonacci serisinin üretimi özyinelemeli yapıya uygundur. Fibonacci'nin kuralı; serinin ilk iki elemanının değeri 1'dir, bundan sonraki her bir eleman kendinden önceki iki elemanın değerlerinin toplamıdır. 1 1 2 3 5 8 13 21 34 55 … Bu serinin herhangi bir sıradaki elemanını hesaplayıp geri döndüren özyineli bir fonksiyon tasarlayalım. İlk bakılması gereken nokta fonksiyonun kendi içinde kendini tekrar eden bir kuralının olup olmadığıdır. Burada, her elemanın kendinden önceki iki elemanın toplamı olması kendini tekrar etme kuralını sağlamaktadır. İkinci kural fonksiyonun yerine getireceği tekrarlı iş için bir bitiş ya da başlangıç koşulunun belirlenmesidir. Buradaki koşul ilk iki elemanın değerinin 1 olarak belirlenmesidir. Buna göre şöyle bir fonksiyon tasarımı mümkündür:

```
def Fibonacci(sirano):
    if sirano==1 or sirano==2:
        return 1
    return Fibonacci(sirano-1)+Fibonacci(sirano-2
```

Fonksiyona parametre olarak gelen sıra numarası 1 ya da 2 ise fonksiyon 1 sonucunu üretir. Diğer durumlarda ise sıranın 1 eksiği ve 2 eksiğindeki elemanın toplamı sonucunu üretir. Burada her bir fonksiyon çağrımı kendi içinde kendine iki kez daha çağrıda bulunmaktadır.

Özyinelemeli Fonksiyon konusunu özetlersek:

1- **Özyineli fonksiyon yazabilmek için algoritmanın yani iş adımlarının kendi içinde kendini tekrar ediyor olması gerekir. Her işlem için özyineleme uygulanamaz.**
2- **Özyineli fonksiyon yazmak için tekrar eden iş adımı kuralı ortaya konmalıdır.**
3- **Tekrar eden iş adımının sonsuza kadar devam etmemesi için bir başlangıç ya da bitiş kuralının belirlenmiş olması gereklidir.**

yetki_sorgula diye bir fonksiyon tanımlanmış ve bunun içersinde de inner diye bir fonksiyon var.

yetki_sorgu fonksiyonu page diye bir değişken istiyor çalışmak için, inner ise role diye bir değişken.

```
def yetki_sorgula(page):
    def inner(role):
        if role == 'Admin':
            return "{0} rolü {1} sayfasına ulaşabilir.".format(role,page)
        else:
            return "{0} rolü {1} sayfasına ulaşamaz.".format(role,page)
    return inner
```

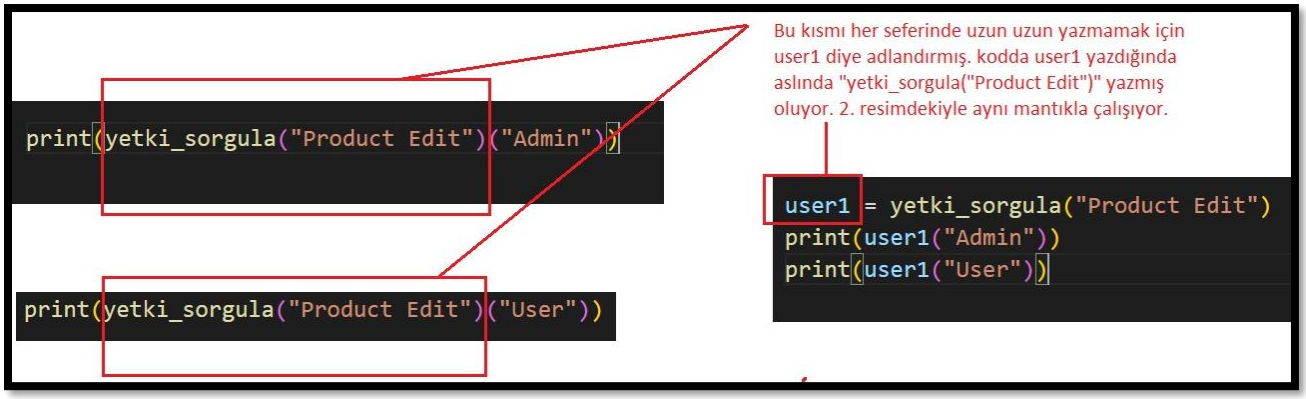
yetki_sorgu fonksiyonunu çağırdığımızda, "return inner" yani inner fonksiyonunun sonucunu döndür komutu olduğu için içersindeki inner fonksiyonu da çalışıyor.

```
def yetki_sorgula(page):
    def inner(role):
        if role == 'Admin':
            return "{0} rolü {1} sayfasına ulaşabilir.".format(role,page)
        else:
            return "{0} rolü {1} sayfasına ulaşamaz.".format(role,page)
    return inner


print(yetki_sorgula("Product Edit")("Admin"))
```

yetki_sorgula('...')('...') kalıbıyla ilk parantez içersindeki page değişkenine ikinci parantez içersindeki ise role değişkenine atanır. Yani print le yukarıda yazdığım gibi fonksiyonu çağırırsan aşağıdaki gibi çalışır.

```
PS C:\Users\aykut\Desktop> & 'C:\python\python.exe' 'c:\Users\aykut\.vsco
py'
Admin rolü Product Edit sayfasına ulaşabilir.
PS C:\Users\aykut\Desktop>
```



Aşağıdaki kod yukarıdaki kodun daha anlaşılır hali.

```
def yetki_sorgula(page, role):

    def inner(role):
        if role == 'Admin':
            print("{0} rolü {1} sayfasına ulaşabilir.".format(role,page))
        else:
            print("{0} rolü {1} sayfasına ulaşamaz.".format(role,page))

    inner(role)


yetki_sorgula("Product Edit","Admin")
```

sorted() Metotu

Bir **dizi** içindeki öğeleri belirli bir ölçüte göre sıraya dizmemizi sağlıyor.

```
print(sorted('ahmet'))
```
['a', 'e', 'h', 'm', 't']
Bu kodlar yardımıyla ahmet adlı karakter dizisi içindeki harfleri alfabe sırasına dizdik.
```
print(sorted(('elma', 'armut', 'kiraz', 'badem')))
```
['armut', 'badem', 'elma', 'kiraz']
NOT: Bu metot, Türkçe karakter içeren öğeleri düzgün sıralayamaz.

```
sayilar = [1,53,45,67,97,5,7]
sonuc = sorted(sayilar)
print(sonuc)
sonuc = sorted((1,53,45,67,97,5,7)) # Tuple
print(sonuc)
```
[1, 5, 7, 45, 53, 67, 97]
[1, 5, 7, 45, 53, 67, 97]

DOSYA İŞLEMLERİ

Dosyalara genel olarak iki ayrı grupta değerlendirilir.

1- Metin (Text) Dosyalar (Notepad, Gedit, Kwrite vb.)
2- İkili (Binary) Dosyalar (Resim, müzik, video dosyaları, MS Office dosyaları vb.)

Metin (Text) Dosyaları	İkili (Binary) Dosyalar
1- Bir metin dosyasındaki ufak değişiklikler dosyanın okunamaz hale gelmesine yol açmaz.	1- Ancak ikili dosyalarda ufak değişiklikler dosyanın tümden bozulmasına yol açabilir.
2- Okunan kayıtları tekrar okumak için dosya kapatılıp yeniden açılmalıdır.	2- Okunan kayıtları tekrar okumak için dosyanın kapatılıp tekrar açılmasına gerek yoktur.
3- Metin dosyası insan tarafından okunabilir çünkü her şey metin olarak saklanıyor.	3- İkili dosyada her şey 0 ve 1 cinsinden yazılır, dolayısıyla ikili dosyayı insan okuyamaz.
4- Metin dosyasında, metin ve karakterler bayt başına bir karakter olarak saklanır. Örneğin, 1245 değeri ikili dosyada 2 bayt yer kaplar ancak metin dosyasında 5 bayt yer kaplar.	4- İkili dosyada, 1245 tamsayı değeri, dosyada olduğu gibi bellekte de 2 baytlık yer kaplar. 5- Farklı veri türlerini (resim, ses, metin) tek bir dosyada saklayabilir.

Dosya Oluşturmak

Dosya oluşturmak için **open()** fonksiyonundan yararlanacağız. **Bir dosyayı açarken ya da oluştururken kip belirtmediğimizde Python bizim o dosyayı okuma kipinde (r) açmak istediğimizi varsayacaktır.** Oluşturulacak olan dosyanın varsayılan değeri .txt dosyasıdır.

```
tahsilat_dosyası = open("tahsilat_dosyası.txt", "w")
```

Burada "tahsilat_dosyası.txt" ifadesi dosyamızın adını belirtiyor. **"w"** harfi ise bu dosyanın yazma kipinde açıldığını söylüyor.

Yukarıdaki komutu çalıştırdığınızda, o anda hangi dizin altında bulunuyorsanız o dizin içinde tahsilat_dosyası.txt adlı boş bir dosyanın oluştuğunu göreceksiniz.

Bu arada, dosya adını yazarken, dosya adı ile birlikte o dosyanın hangi dizin altında oluşturulacağını da belirleyebilirsiniz. Örneğin:

```
open("C:\aylar\nisan\toplam masraf\masraf.txt", "w")
```

```
OSError: [Errno 22] Invalid argument: 'C:\x07ylar\nisan\toplam masraf\\masraf.txt'
```

Bunun sebebi, bildiğiniz gibi, Python'ın \a, \n ve \t ifadelerini birer kaçış dizisi olarak algılamasıdır. Bu durumdan kaçabilmek için, dizin adlarını ters taksim işareti ile ayırmanın dışında, r adlı kaçış dizisinden de yararlanabilirsiniz.

```
open(r"D:\DERSLER\NEÜ\2. Yıl Bahar Dönemi\Algoritma ve Programlama II\PycharmProjects\UDEMY\masraf.txt", "w")
```

ya da

```
open("D:\\DERSLER\\NEÜ\\2. Yıl Bahar Dönemi\\Algoritma ve Programlama II\\PycharmProjects\\UDEMY\\masraf.txt", "w")
```

Bu şekilde, eğer bilgisayarınızda **D:\DERSLER\NEÜ\2. Yıl Bahar Dönemi\Algoritma ve Programlama II\PycharmProjects\UDEMY** adlı bir dizin varsa, o dizin içinde masraf.txt adlı bir dosya oluşturulacaktır.

Dosya Yolu

Dosya yolunu verirken **'\'** ile yan yana gelecek bu karakterler hataya sebep olabilir. Bu durumu önlemek için aşağıdaki seçenekleri kullanabilir:

1- '\' karakteri yerine '\\' kullanmak.

```
file = open("D:\\DERSLER\\NEÜ\\PycharmProjects\\UDEMY\\deneme.txt", "r")
```

2- '\' karakteri yerine '/' kullanmak.

```
file = open("D:/DERSLER/NEÜ/PycharmProjects/UDEMY/deneme.txt", "r")
```

3- Dosya yolu başına r eklemek.

```
file = open(r"D:/DERSLER/NEÜ/PycharmProjects/UDEMY/deneme.txt", "r")
```

Dosyaya Yazmak

Yazma kipinde **(w), (a), (x)** açtığımız bir dosyaya bir veri yazabilmek için dosyaların **write()** adlı metotundan yararlanacağız.

```
ths = open("tahsilat_dosyası.txt", "w")
ths.write("Halil Pazarlama: 120.000 TL"),
ths.close()
```

Bu kodlarda sırasıyla şu işlemleri gerçekleştirdik:

1- tahsilat_dosyası adlı bir dosyayı yazma kipinde açarak, bu adda bir dosya oluşturulmasını sağladık
2- write() metotunu kullanarak bu dosyaya bazı bilgiler girdik.
3- Dosyamıza yazdığımız bilgilerin dosyaya işlendiğinden emin olmak için close() metotuyla programımızı kapattık.

NOT: Python'da bir dosyayı "w" kipinde açtığımızda, eğer o adda bir dosya ilgili dizin içinde zaten varsa, Python bu dosyayı sorgusuz sualsiz silip, yerine aynı adda başka bir boş dosya oluşturacaktır. Yani mesela yukarıda tahsilat_dosyası.txt adlı dosyayı oluşturup içine bir şeyler yazdıktan sonra bu dosyayı yine "w" kipinde açmaya çalışırsanız, Python bu dosyanın bütün içeriğini silip, yine tahsilat_dosyası.txt adını taşıyan başka bir dosya oluşturacaktır.

'w' (Write)

- Belirtilen konumda verilen isimde dosya yoksa oluşturulur, varsa dosya içeriği tamamen silinir. Yani eski veriler silinir onun yerine yeni eklenen veriler yazılır.

'a' (Append)

- Belirtilen konumda verilen isimde dosya yoksa oluşturulur, varsa dosya içeriği korunur.
- Yeni veri yazılması durumunda mevcut içeriğin sonuna eklenir.

'x' modu

- Belirtilen konumda verilen isimde dosya varsa hata verir, yoksa o isimde dosya oluşturulur.

flush = " " Parametresi

Yazdığımız programda, dosyaya yazmak istediğiniz bilgilerin hiç bekletilmeden doğrudan dosyaya aktarılmasını istiyorsak bu parametreyi kullanabiliriz.

Bu parametrenin **True** ve **False** olmak üzere iki değeri vardır, **öntanımlı değeri False**'tur. Yani herhangi bir değer belirtmezsek Python bu parametrenin değerini False olarak kabul edecek ve bilgilerin dosyaya yazılması için dosyanın kapatılmasını bekleyecektir.

Dosya Okumak

Bir dosyayı okuma kipinde açmak için **"r"** harfini kullanacağız eğer dosya konumda yoksa hata verir.
Bir dosyayı okumak için ise **read(), readline() ve readlines()** adlı üç farklı metottan yararlanacağız.

Bir dosya üzerinde hem okuma yapmak hem de yazma yapmak istiyorsak "r+" kipinden yararlanmalıyız. Eklenen veriler dosyanın en başına yazılır. Dosya belirtilen konumda yoksa hata alırız.

Öncelikle içeriği aşağıdaki gibi olan, deneme.txt adlı bir dosyamızın olduğunu varsayalım:

```
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
```

```
deneme = open("deneme.txt")
print(deneme.read())
```

Ahmet Obudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
read() metotu, dosyanın bütün içeriğini bir karakter dizisi olarak veriyor.

```
deneme = open("deneme.txt")
deneme = open("deneme.txt")
print(deneme.readline())
print(deneme.readline())
print(deneme.readline())
```

Ahmet Obudak : 0533 123 23 34

Mehmet Sulun : 0532 212 22 22

readline() metotu, .txt dosyasındaki tek bir satır veriyor.
Son satırı da okuduktan sonra, readline() metotunu tekrar çalıştırırsak (5. satır) hiçbir çıktı alamayız çünkü okunacak satır kalmadı.

```
deneme = open("deneme.txt")
print(deneme.readlines())
```

['Ahmet Ozbudak : 0533 123 23 34\n', 'Mehmet Sulun : 0532 212 22 22\n']
readlines() metotu, çıktıyı liste olarak veriyor.

```
with open("deneme.txt", "r") as deneme:
    print(deneme.read())
```

Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22

```
with open("deneme.txt", "r") as deneme:
    print(deneme.readline())
```

Ahmet Ozbudak : 0533 123 23 34

```
with open("deneme.txt", "r") as deneme:
    print(deneme.readlines())
```

['Ahmet Ozbudak : 0533 123 23 34\n', 'Mehmet Sulun : 0532 212 22 22\n']

Dosya Kapatma

Açtığımız bir dosyayı **close()** fonksiyonu ile kapatabiliriz. Eğer açılan bir dosyanın otomatik olarak kapanmasını istiyorsak **with** adlı deyimi kullanmalıyız.

```
with open("deneme.txt", "r") as dosya:
    print(dosya.read())
```

with deyimini içeren bloktan çıkıldığı anda dosya kapatılacaktır. Ayrıca bir close() satırı yazmamıza gerek yok.

Dosya Üzerinde Hareket

- Dosya bir kez okunduktan sonra imleç otomatik olarak dosyanın başına dönmemektedir.
- read() metouyla dosyayı bir kez okuduktan sonra, dosya tekrar okunursa boş bir karakter dizisi elde edilir, çünkü dosya okunduktan sonra okunacak başka bir satır kalmamış, imleç dosya sonuna ulaşmış ve otomatik olarak da başa dönmemiştir.
- Dosya üzerinde imleci (cursor) hareket ettirebilmek için **seek()** metotu kullanılır.
- Eğer o anda dosyanın hangi bayt (cursor-imleç) konumunda bulunduğunu öğrenmek istersek **tell()** adlı metottan yararlanabiliriz.

```
deneme = open("deneme.txt", "r")
print(deneme.readline())
deneme.seek(6)
print(deneme.readline())
print(deneme.tell())
```

Ahmet Ozbudak : 0533 123 23 34

Ozbudak : 0533 123 23 34

32
seek(6) metotu sayesinde, dosyanın 6. baytına, yani Ahmet'ten sonraki kısma ulaşmış olduk.

```
with open("deneme.txt", "r+") as deneme:
    veri = deneme.read()
    basaEkle = "Kamber Ata : 0533 123 23 34 \n"
    deneme.seek(0)
    deneme.write(basaEkle + veri)
```

Kamber Ata : 0533 123 23 34
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22

```
with open("deneme.txt", "r") as deneme:
    for i in range(4):
        print(deneme.readline(), end="")
```

Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22

```
with open("deneme.txt", "r") as deneme:
    line = deneme.readline()
    while line != "":
        print(line)
        line = deneme.readline()
```

Ahmet Ozbudak : 0533 123 23 34

Mehmet Sulun : 0532 212 22 22

Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
deneme.txt dosyasında yukarıdaki verilen olduğunu varsayalım.

```
with open("deneme.txt", "r+") as deneme:
    veri = deneme.readlines()
    veri.insert(3, "Görsel Programlama \n")
    deneme.seek(0)
    for ders in veri:
        deneme.write(ders)
        print(deneme.tell())
    deneme.writelines(veri)
```

32
61
82

Yukarıdaki kod çalıştıktan sonra deneme.txt dosyasının içeriği aşağıdaki gibi olur.
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22Görsel Programlama
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22Görsel Programlama

insert() metotu, öğeleri listenin istediğimiz bir konumuna yerleştirir. Ancak indexlediğimiz satırın sonuna ekler. Yani 3. satırın en sonuna eklenmiş oldu.
writelines() adlı metot bize dosyaya liste tipinde verileri yazma imkanı verir. seek() karakter karakter veri içinde gezinmemizi sağlar.

```
with open("deneme.txt", "r+") as deneme:
    deneme.seek(8)
    print(deneme.read())
```

budak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22Görsel Programlama
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22Görsel Programlama

Başlangıçta dosya içeriğimiz şu şekilde olsun:
1-Bmw
2- Audi
4-Honda
Sağdaki kodu çalıştırdığımızda şu çıktıyı elde ederiz:
1-Bmw
2- Audi
3-Renault
4-Honda
5-Nissan

```
with open("markalar.txt","a") as file:
    file.write("\n5-Nissan")

with open("markalar.txt","r+") as file:
    markalar = file.readlines()
    markalar.insert(2,"3-Renault\n")
    file.seek(0) #imleç başa gitti
    file.writelines(markalar)

with open("markalar.txt") as file:
    print(file.read())
```

```python
def dosya_kopyala(dosya_ismi, yeni_dosya_ismi):
    with open(dosya_ismi) as file:
        icerik = file.read()
    with open(yeni_dosya_ismi, "w") as new_file:
        new_file.write(icerik)
dosya_kopyala("deneme.txt","denemeYeni.txt")
```

Bu kodu çalıştırığımızda denemeYeni.txt dosyası oluşacak ve içeriği
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
olacaktır.

deneme.txt dosyasının içeriğinin
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
olduğunu varsayalım.

```python
def ters_cevir(dosya_ismi, yeni_dosya_ismi):
    with open(dosya_ismi) as file:
        icerik = file.read()
    with open(yeni_dosya_ismi, "w") as new_file:
        new_file.write(icerik[::-1])
ters_cevir("deneme.txt","denemeYeni.txt")
```

Bu kodu çalıştırığımızda denemeYeni.txt dosyası oluşacak ve içeriği
22 22 212 2350 : nuluS temheM
43 32 321 3350 : kadubzO temhA
olacaktır.

deneme.txt dosyasının içeriğinin
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
olduğunu varsayalım.

```python
def bilgilendir(dosya_ismi):
    with open(dosya_ismi) as file:
        satirlar = file.readlines()
    sonuc = {
        "satir_sayisi": len(satirlar),
        "kelime_sayisi": sum(len(satir.split(' ')) for satir in satirlar),
        "karakter_sayisi": sum(len(satir) for satir in satirlar)
    }
    return sonuc
print(bilgilendir("deneme.txt"))
```

deneme.txt dosyasının içeriğinin
Ahmet Ozbudak : 0533 123 23 34
Mehmet Sulun : 0532 212 22 22
olduğunu varsayalım. Çıktı olarak aşağıdaki satırı elde ederiz.
{'satir_sayisi': 2, 'kelime_sayisi': 14, 'karakter_sayisi': 60}

```python
def urun_ekle(ad, fiyat):
    with open("urunler.txt" ,"a") as file:
        file.write(f"ad: {ad} fiyat: {fiyat}\n")
urun_ekle("samsung s10", 5000)
```

urunler.txt dosyası oluşturulup içine 4. satırdaki bilgiler eklendi.
ad: samsung s10 fiyat: 5000

```python
def bul_ve_degistir(dosya_ismi, eski_kelime, yeni_kelime):
    with open(dosya_ismi, "r+") as file:
        text = file.read()
        yeni_text = text.replace(eski_kelime, yeni_kelime)
        file.seek(0)
        file.write(yeni_text)
        file.truncate()
bul_ve_degistir("urunler.txt" ,"s10" ,"s12")
```

Bu kodları çalıştırdığımızda s10 yerine s12 gelecektir.
ad: samsung s12 fiyat: 5000

Dosya Metotları ve Nitelikleri

1- closed Niteliği

Bu nitelik, bir dosyanın kapalı olup olmadığını sorgulamamızı sağlar.

`deneme = open("deneme.txt", "r")`
`deneme.close()`
`deneme.closed`
	True

2- readable() Metotu

Bu metot bir dosyanın okuma yetkisine sahip olup olmadığını sorgulamamızı sağlar. Eğer bir dosya "r" gibi bir kiple açılmışsa, yani o dosya 'okunabilir' özellikle ise bu metot bize True çıktısı verir.

3- writable() Metotu

Bu metot bir dosyanın yazma yetkisine sahip olup olmadığını sorgulamamızı sağlar. Eğer bir dosya "w" gibi bir kiple açılmışsa bu metot bize True çıktısı verir.

4- truncate() (Kırpma) Metotu

Bu metot yardımıyla dosyalarımızı istediğimiz boyuta getirebiliyoruz. Eğer parametresiz olarak kullanırsak (imleç dosyanın başında ise) tüm veri silinir.

`with open("deneme.txt", "r+") as deneme:`
`  deneme.truncate(10)`
Ahmet Ozbu
	deneme.txt adlı dosyanın ilk 10 baytı dışındaki bütün veriler silinir.
	Dosya içeriğimizin aşağıdaki gibi olduğunu varsayalım.
	Ahmet Ozbudak : 0533 123 23 34
	Mehmet Sulun : 0532 212 22 22

1- mode Niteliği

Bu nitelik, bize bir dosyanın hangi kipte açıldığına dair bilgi verir.

2- name Niteliği

Bu nitelik, bize bir dosyanın adını verir.

3- encoding Niteliği

Bu nitelik, bize bir dosyanın hangi dil kodlaması ile kodlandığını söyler.

TEST ETME VE HATA YAKALAMA

Test etme, bir programın istenildiği gibi çalışıp çalışmadığını deneme ve belirleme için bir programın çalıştırılma sürecidir. Amaç hata oluşturan şeyler bulmaktır.

Hata ayıklama, istenildiği gibi çalışmayan bir programı düzeltmeye çalışma sürecidir.

Kara Kutu (Black Box) Testi	Şeffaf Kutu (Glass Box) Testi
Yazılımın iç yapısı ve tasarımı bilinmeden testlerin tasarlandığı bir yazılım test tekniğidir. 1- Kod erişimi zorunlu değildir. 2- Başkaları tarafından yapılabilir. 3- Her güncelleme sonrası yeniden kullanılabilir. 4- Implementation (uygulanışını) gösterir. 5- Daha az zamanımızı alır. 6- Programlama bilgisi gerekli değildir. 7- Sistemin iç yapısı bilinmediği için kaynak kod içerisinde kümelenmiş hataların bulunması zorlaşır.	Yazılımın iç yapısı ve tasarımı bilinerekten testlerin tasarlandığı bir yazılım test tekniğidir. 1- Kod erişimi zorunludur. 2- Uygulayıcıdan bağımsız yapılamaz. 3- Veri seti değişebilir. 4- Implementation (uygulanışını) göstermez. 5- Kapsamlı ve zaman alıcıdır. 6- Programlama bilgisi gereklidir. **Dezavantajları:** - Birçok kez keyfi olarak döngülerden geçebilir . - Eksik/Gözden kaçırılan yollar. **NOT:** Koddaki her potansiyel yol en az bir kez test edilirse **path-complete** olarak adlandırılır.

Program içerisinde doğruluğunu iddia ettiğimiz belirli durumlar olabilir. Bu durumların yanlış olması halinde programın hata oluşturmasını isteyebiliriz. Doğruluğunu iddia ettiğimiz durumu **assert** deyimi ile tanımlarız ve eğer bu durum yanlış ise program otomatik olarak **AssertionError** üretir.
İddialar, koşulların doğru olup olmadığını kontrol eden boole ifadeleridir. Doğruysa, program hiçbir şey yapmaz ve bir sonraki kod satırına geçer. Ancak, yanlışsa, program durur ve bir hata oluşturur. assert deyiminin if deyiminden farkı, koşulun yanlış olması halinde assert deyiminde otomatik olarak hata üretilir.
Savunma programlama, hataların olacağını varsaymak ve bunlara karşı önlem almaktır.

```
k = input("Kullanıcı adı: ")
assert k=="python"
s=input("Şifre: ")
assert s=="123"
print("Programa hoşgeldiniz...")
```

Kullanıcı adı: python
Şifre: 123
Programa hoşgeldiniz...

```
k = input("Kullanıcı adı: ")
assert k=="python"
s=input("Şifre: ")
assert s=="123"
print("Programa hoşgeldiniz...")
```

Kullanıcı adı: hakan
  assert k=="python"
AssertionError

```
sicaklik=int(input("Sıcaklık kaç derece:"))
assert sicaklik>=20,"Hava soğuk"
print("Güzel bir gün ölmek için")
```

Sıcaklık kaç derece:21
Güzel bir gün ölmek için

```
sicaklik=int(input("Sıcaklık kaç derece:"))
assert sicaklik>=20,"Hava soğuk"
print("Güzel bir gün ölmek için")
```

Sıcaklık kaç derece:19
  assert sicaklik>=20,"Hava soğuk"
AssertionError: Hava soğuk

Önce kodu yazıp, sonra testleri yazarsak uygulamayı hatırlarız. Ancak bu şekilde uygulamada aynı hataları yapma ihtimaliniz vardır.	Önce testleri yazarsanız, hakkında hata yapmış olabileceğiniz bir davranışın bazı yönlerini fark edebilirsiniz.

Hata Ayıklama Araçları

- Python hata mesajları	- Assert (iddia) deyimi	- print
- Python yorumlayıcısı	- Python destek web siteleri	- Python

BÖCEK (BUG)	
Açık (Overt) ve Gizli (Covert) **1- Açık**, bir hatanın belirgin bir göstergesi vardır. Program çöküyor veya çalışması gerektiğinden daha uzun sürüyor. **2- Gizli**, bir böceğin (bugun) belirgin bir göstergesi yoktur. Program, yanlış bir cevap vermek dışında sorunsuz bir şekilde sonuçlanabilir.	**Devamlı (Persistent) ve Kesintili (Intermittent)** **1-** Devamlı bir böcek, program aynı girdilerle her çalıştırıldığında ortaya çıkar. **2-** Kesintili bir böcek ise, program aynı girdiler üzerinde ve görünüşte aynı koşullar altında çalıştırıldığında bile, yalnızca bazı zamanlar oluşur.

1- Sözdizim (Syntax) Hataları

Mesaj size programın neresinde oluştuğunu bildirir.

```
print "Merhaba Python
```

SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?

`SyntaxError`, bu hatalar, programlama diline ilişkin bir özelliğin yanlış kullanımından veya programcının yaptığı yazım hatalarından kaynaklanır.

NameError, tanımlanmamış bir değişken kullanımı. TypeError, bir fonksiyona veya metota yanlış sayıda parametre veriyorsunuz. AttributeError, var olmayan bir özniteliğe veya yönteme erişmeye çalışıyorsunuz.	IndexError, yanlış index numarası. ValueError, hatalı tip kullanımı. KeyError, sözlüğün içermediği bir anahtarı kullanarak sözlüğün bir öğesine erişmeye çalışıyorsunuz.

2- Çalışma Zamanı (Runtime) Hataları

1. Program çalışırken bir şeyler ters giderse, çalışma zamanı hataları yorumlayıcı tarafından üretilir.
2. Programım kesinlikle hiçbir şey yapmıyor.
   - Bu durum genellikle sınıflar ve fonksiyonlardan oluşan kodlarda hiçbir çağrım yapılmaması nedeniyle oluşur.
3. Programım askıda kalıyor.
   - Sonsuz döngü veya sonsuz özyineleme (recursion)
4. Programımı çalıştırdığım zaman bir hata alıyorum.
   - Çalışma sırasında bir şeyler ters giderse, Python sorunun adını, sorunun oluştuğu programın satırını ve geri izleme içeren bir ileti yazdırır.
   - Geri izleme, sizi bulunduğunuz yere götüren işlev çağırma dizisini izler.

3- Anlamsal (Semantic) Hatalar

Hata mesajı üretmeden çalışan, ancak doğru olanı yapmayan bir programdaki sorunlardır. Örneğin bir fonksiyon, bölme işlemini yapabilir ancak yanlış bir sonuç üretir.

```
sayı1 = input("İlk sayıyı girin: ")
sayı2 = input("İkinci sayıyı girin: ")
print(sayı1, "+", sayı2, "=", sayı1+sayı2)
```

-12 ve 13 girdiğimizi varsayalım-

12 + 13 = 1213

Bu programda kullanıcı veri girdiği zaman, programımız toplama işlemi değil karakter dizisi birleştirme işlemi yapacaktır. Böyle bir program çalışma sırasında hata vermeyeceği için buradaki sorunu tespit etmek, özellikle büyük programlarda çok güçtür. Yani sizin düzgün çalıştığını zannettiğiniz program aslında gizliden gizliye bir bug barındırıyor olabilir.

try… except…

```
try:
    #....birtakım işler...
except:
    #...hata mesajı...
```

Burada herhangi bir hata adı belirtmedik. Böylece Python, yazdığımız programda **hangi hata** oluşursa oluşsun hepsini yakalayabilecektir.

```
try:
    # Hata verebileceğini bildiğimiz kodlar
except HataAdı:
    # Hata durumunda yapılacak işlem
```

Burada ise özel olarak hatanın adını yazdık, o hata karşımıza çıktığında kod bloğunun aşağısındaki işlemler yapılacaktır.

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")
try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError:
    print("Lütfen sadece sayı girin!")
```

int(ilk_sayı) ve **int(ikinci_sayı)** kodları, kullanıcının gireceği veri türüne göre (harf girerse) hata üretme potansiyeline sahiptir. O yüzden, burada hata vereceğini bildiğimiz o kodları **try** bloğu içine aldık.

Yine bildiğimiz gibi, veri dönüştürme işlemi sırasında kullanıcının uygun olmayan bir veri girmesi halinde üretilecek hata bir **ValueError**'dır. Dolayısıyla **except** bloğu içine yazacağımız hata türünün adı da ValueError olacaktır.

Bu kodlarla Python'a şu emri vermiş olduk:

- Eğer try bloğu içinde belirtilen işlemler sırasında bir ValueError ile karşılaşırsan bunu görmezden gel ve normal şartlar altında kullanıcıya göstereceğin hata mesajını gösterme. Onun yerine kullanıcıya Lütfen sadece sayı girin! uyarısını göster.

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")
try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
```

Gördüğünüz gibi, Python'ın **ZeroDivisionError** vereceğini bildiğimiz durumlara karşı bu hata türünü yakalama yoluna gidiyoruz. Böylece kullanıcıya anlamsız ve karmaşık hata mesajları göstermek ve daha da kötüsü, programımızın çökmesine sebep olmak yerine daha anlaşılır mesajlar üretiyoruz.

Dikkat ederseniz yukarıdaki kodlar aslında bir değil iki farklı hata üretme potansiyeline sahip. Eğer kullanıcı sayı değerli veri yerine harf değerli bir veri girerse **ValueError**, eğer bir sayıyı 0 'a bölmeye çalışırsa da **ZeroDivisionError** hatası alıyoruz. Peki aynı kodlarda iki farklı hata türünü nasıl yakalayacağız?

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")
try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
except ValueError:
    print("Lütfen sadece sayı girin!")
```

```python
while True:
    try:
        num1 = int(input("Bir sayı giriniz: "))
        num2 = int(input("Bir sayı giriniz: "))
        print(num1/num2)
        break
    except ZeroDivisionError:
        print("0'a bölme hatası.")
    except ValueError:
        print("Geçersiz değer girildi.")
    except:
        print("Bilinmeyen bir hata oluştu.")
```

Bir sayı giriniz: 12 Bir sayı giriniz: a Geçersiz değer girildi. Bir sayı giriniz:	Bir sayı giriniz: 12 Bir sayı giriniz: 0 0'a bölme hatası. Bir sayı giriniz:	Bir sayı giriniz: 12 Bir sayı giriniz: 3 4.0 Bir sayı giriniz:

except ifadesi ile kullanılan hataların sırası önem arz etmektedir. Bir hata deyimi çalıştırılsa diğerleri atlanır. Bu nedenle, diğer hataları kapsayacak daha geniş bir hata adı ile en üstte hatalar yakalanırsa, diğer except ifadeleri göz ardı edilmiş olur.

```python
def bolme(x,y):
    print("Bölme fonks. girildi.")
    bol = x/y
    print("Bölme fonks. çıkıldı.")
    return bol
try:
    print("Bölme fonks. öncesi.")
    print(bolme(5, a))
    print("Bölme fonks. sonrası.")
except ZeroDivisionError:
    print("0'a bölme hatası.")
except ValueError:
    print("Geçersiz değer girildi.")
except:
    print("Bilinmeyen hata oluştu.")
```

Bölme fonksiyonu öncesi.
Bilinmeyen bir hata oluştu.

```python
def bolme(x,y):
    print("Bölme fonks. girildi.")
    bol = x/y
    print("Bölme fonks. çıkıldı.")
    return bol
try:
    print("Bölme fonks. öncesi.")
    print(bolme(5, 2))
    print("Bölme fonks. sonrası.")
except ZeroDivisionError:
    print("0'a bölme hatası.")
except ValueError:
    print("Geçersiz değer girildi.")
except:
    print("Bilinmeyen hata oluştu.")
```

Bölme fonks. öncesi.
Bölme fonks. girildi.
Bölme fonks. çıkıldı.
2.5
Bölme fonks. sonrası.

Hata yakalama bloğu sayesinde;

- **Programın çalışmasına devam etmesi,**
- **Hata mesajının kullanıcıya anlaşılır bir şekilde verilmesi,**
- **Her hatanın ayrı ayrı tespit edilerek geçerli hatanın belirtilmesi sağlanmaktadır.**

Python' da Error ve Exception terimleri birbirleri yerine kullanılabilir.

- **Exception (Özel Durumlar) işlemler üzerinde olabilecek bir hatayı temsil ederken (Örn: Aritmetik işlemler, int yerine string vb.)**
- **Error (Programcı Hataları) işlevsel sorunlar ile ilgili hatalar için söylenebilir (Örn: Dosya bulunamaması, sözdizim hataları vb.)**
- **Bug(Program Kusurları) Siz ne istiyorsunuz? Program ne yapıyor?**

try... except... as...

Python bir programın çalışması esnasında hata üretirken çıktıda hata türünün adıyla birlikte kısa bir hata açıklaması veriyor. Yani mesela şöyle bir çıktı üretiyor:

ValueError: invalid literal for int() with base 10: 'f'

Burada 'ValueError' hata türünün adı, 'invalid literal for int() with base 10: "f" ' ise hatanın açıklamasıdır. Eğer istersek, yazdığımız programda bu hata açıklamasına erişebiliriz.

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")
try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError as hata:
    print(hata)
```

ilk sayı: 12
ikinci sayı: Elif
invalid literal for int() with base 10: 'Elif'
Gördüğünüz gibi, bu defa çıktıda hata türünün adı (**ValueError**) görünmüyor. Onun yerine sadece hata açıklaması var.

Diyelim ki kullanıcıya olası bir hata durumunda hem kendi yazdığınız hata mesajını, hem de özgün hata mesajını göstermek istiyorsunuz.

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")
try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError as hata:
    print("Sadece sayı girin!")
    print("orijinal hata mesajı: ", hata)
```

ilk sayı: 12
ikinci sayı: Elif
Sadece sayı girin!
orijinal hata mesajı: invalid literal for int() with base 10: 'Elif'

Özeltemek gerekirse;

- Burada '**except falancaHata as filanca**' yapısını kullanarak falancaHata'yı filanca olarak isimlendiriyor ve daha sonra bu ismi istediğimiz gibi kullanabiliyoruz. Böylece bütün hata türleri için hem kendi yazdığınız mesajı görüntüleyebiliyor, hem de özgün hata mesajını çıktıya eklemiş oluyoruz.

try... except... else...

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)
except ValueError:
    print("Lütfen sadece sayı girin!")
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
```

Burada bütün kodlarımızı tek bir **try...** bloğu içine tıkıştırıyoruz. Bu blok içinde gerçekleşen hataları da daha sonra tek tek **except...** blokları yardımıyla yakalıyoruz. Ama eğer biz istersek bu kodlarda verilebilecek hataları gruplamayı da tercih edebiliriz:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
except ValueError:
    print("Lütfen sadece sayı girin!")
else:
    try:
        print(bölünen/bölen)
    except ZeroDivisionError:
        print("Bir sayıyı 0'a bölemezsiniz!")
```

1- İlk olarak **try... except...** bloğu yardımıyla öncelikle **int(input())** fonksiyonu ile kullanıcıdan gelecek verinin sayı olup olmadığını denetliyoruz.

2- Ardından bir **else...** bloğu açarak, bunun içinde ikinci **try... except...** bloğumuzu devreye sokuyoruz. Burada da bölme işlemini gerçekleştiriyoruz. Kullanıcının bölme işlemi sırasında 0 sayısını girmesi ihtimaline karşı da except ZeroDivisionError ifadesi yardımıyla olası hatayı göğüslüyoruz.

3- Böylelikle her blok içinde sadece almayı beklediğimiz hatayı karşılıyoruz. Mesela yukarıda ilk **try...** bloğu içindeki dönüştürme işlemi yalnızca ValueError hatası verebilir. **else:** bloğundan sonraki **try...** bloğunda yer alan işlem ise ancak **ZeroDivisionError** verecektir. Biz yukarıda kullandığımız yapı sayesinde her bir hatayı tek tek ve yeri geldiğinde karşılıyoruz.

4- Bu durumun aksine, sayfanın başında verdiğimiz **try... except** bloğunda hem *ValueError* hem de **ZeroDivisionError** hatalarının gerçekleşme ihtimali bulunuyor. Dolayısıyla biz orada bütün hataları tek bir **try...** bloğu içine sıkıştırmış oluyoruz. İşte **else:** bloğu bu sıkışıklığı gidermiş oluyor.

```
try :
    #...Hata aranacak kod bloğu...
except:
    #...Hata yakalanınca yapılacak işlemler...
else:
    #...Eğer hiçbir hata oluşmadıysa yapılacak işlemler...
```

try... except... finally...

```
try:
    #...bir takım işler...
except birHata:
    #...hata alınınca yapılacak işlemler...
finally:
    #...hata olsa da olmasa da yapılması gerekenler...
```

Eğer yazdığınız programda mutlaka ama mutlaka çalışması gereken bir kısım varsa, o kısmı **finally...** bloğu içine yazarız.

```
try:
    dosya = open("yeni.txt", "w+")
    dosya.write("Yeni Dosya")
    dosya.seek(0)
    print(dosya.readline())
except IOError:
    print("Dosya işlemleri hatası.")
finally:
    print("Finally Blok")
    dosya.close()
```

Yeni Dosya
Finally Blok

- Ayrıca dosyamızın içine "Yeni Dosya" yazdırdı. -

try... except... else... finally...

```
try:
    dosya = open("yeni1.txt", "r")
    dosya.write("Yeni Dosya")
    dosya.seek(0)
    print(dosya.readline())
except IOError:
    print("Dosya işlemleri hatası.")
else:
    print("Her şey yolunda.")
finally:
    print("Finally Blok")
    try:
        dosya.close()
    except NameError as isimHatasi:
        print("Dosya göstergesi bulunamadı: ",isimHatasi)
```

Dosya işlemleri hatası.
Finally Blok
Dosya göstergesi bulunamadı: name 'dosya' is not defined

- yeni1.txt isminde bir dosya olmadığı için çalıştırdığımızda bu hatayı aldık. -

raise

raise deyimi ile Python yorumlayıcısının hata oluşturmayacağı istenilen özel durumlar için hata oluşturulabilir.

```
bölünen = int(input("bölünecek sayı: "))
if bölünen == 23:
    raise Exception("Bu programda 23 sayısını görmek istemiyorum!")
bölen = int(input("bölen sayı: "))
print(bölünen/bölen)
```

-23 girdiğimizi varsayalım-

```
  raise Exception("Bu programda 23 sayısını görmek istemiyorum!")
Exception: Bu programda 23 sayısını görmek istemiyorum!
```

Burada eğer kullanıcı 23 sayısını girerse, kullanıcıya bir hata mesajı gösterilip programdan çıkılacaktır. Biz bu kodlarda Exception adlı genel hata mesajını kullandık. Burada Exception yerine her istediğimizi yazamayız. Yazabileceklerimiz ancak Python'da tanımlı hata mesajları olabilir. Örneğin NameError, TypeError, ZeroDivisionError, IOError, vb...

```
def colorize(text, color):
    colors = ("blue","red","white","black","orange")
    if type(text) is not str:
        raise TypeError("text str tipinde olmalıdır.")
    if color not in colors:
        raise ValueError("geçersiz bir renk ismi.")
    print(f"{text} {color} olarak yazdırıldı.")
colorize("selam","yellow")
```

```
   raise ValueError("geçersiz bir renk ismi.")
ValueError: geçersiz bir renk ismi.
```

```
tr_karakter = "şçğüöıİ"
parola = input("Parolanız: ")
try:
    for i in parola:
        if i in tr_karakter:
            raise TypeError("Parolada Türkçe karakter var.")
except TypeError:
    print("Parolada Türkçe karakter kullanılamaz.")
else:
    print("Parola kabul edilidi.")
```

Parolanız: **aliş** Parolada Türkçe karakter kullanılamaz.	Parolanız: **test** Parola kabul edilidi.

İç İçe try Blokları

try bloğu eşleşecek bir istisna bulamadıysa, bir üstteki try bloğunun except cümlelerinde bu hatayı arar.

- Eğer üstte bir eşleşme bulunursa, o except bloğu yürütülür.
- Eğer üstteki try bloğunda eşleşmezse, program akışı bir eşleşme bulana kadar bir üstteki try bloğunun except cümlelerini araştırır.
- Tüm durum sonunda bir eşleşme bulunmazsa, hata Python hata yakalayıcısı tarafından ele alınır.

```
pencil: kalem
ilk finally cümleciği
pencil sharpner: kalem tıraş
üçüncü finally cümleciği
```

```python
sozluk = {
    "notebook": 'defter',
    "pencil": 'kalem',
    "eraser": 'silgi',
    "pencil sharpner": 'kalem tıraş'
}

try:
    try:
        print("pen:", sozluk["pen"])
    except:
        print("pencil:", sozluk["pencil"])
    finally:
        print("ilk finally cümleciği")
except KeyError:
    try:
        print("eraser:", sozluk["eraser"])
    except:
        print("sözlükte eraser kelimesinin karşılığı yok'")
    finally:
        print("ikinci finally cümleciği")
finally:
    try:
        print("pencil sharpner:", sozluk["pencil sharpner"])
    except:
        print("sözlükte pencil sharpner kelimesinin karşılığı yok")
    finally:
        print("üçüncü finally cümleciği")
```

Sözlükte pen olmadığı için hata verir ve except e girer

sözlükte pencil olduğu için sözlükten karşılığını yazdırır

try veya except başarılı çalışırsa finally kısmı da çalışır

try kısmında rxcept ve finally çalıştığı için en dıştaki excepte girmez ve

dıştaki finally çalışır

try içersindeki kelime sözlükte olduğu için try çalışır ve finally kısmıda çalışır

```
ilk finally cümleciği
eraser: silgi
ikinci finally cümleciği
pencil sharpner: kalem tıraş
üçüncü finally cümleciği
```

```python
sozluk = {
    "notebook": 'defter',
    "pencil": 'kalem',
    "eraser": 'silgi',
    "pencil sharpner": 'kalem tıraş'
}

try:
    try:
        print("pen:", sozluk["pen"])
    except:
        print("pencil:", sozluk["book"])
    finally:
        print("ilk finally cümleciği")
except KeyError:
    try:
        print("eraser:", sozluk["eraser"])
    except:
        print("sözlükte eraser kelimesinin karşılığı yok'")
    finally:
        print("ikinci finally cümleciği")
finally:
    try:
        print("pencil sharpner:", sozluk["pencil sharpner"])
    except:
        print("sözlükte pencil sharpner kelimesinin karşılığı yok")
    finally:
        print("üçüncü finally cümleciği")
```

try hata verecek

except de sözlükte yok o da hata verecek

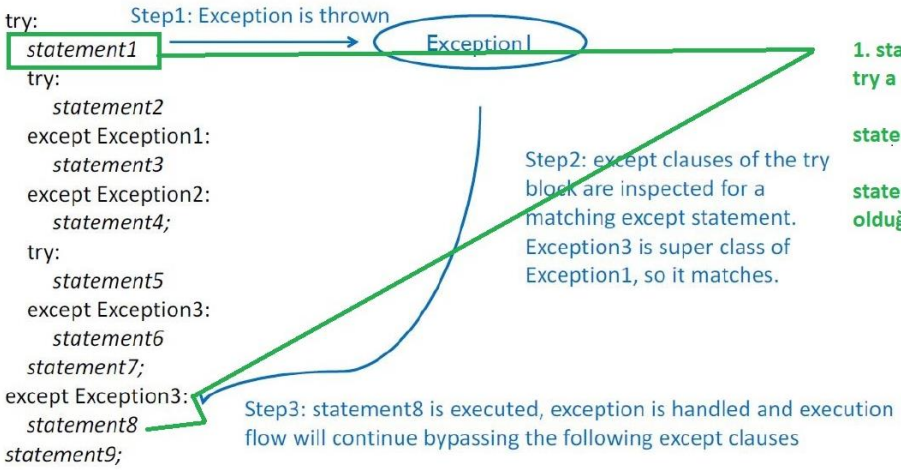
finally cümlecikleri her durumda çalışır

Bu nedenle en dıştaki try içersi hatalı olarak dönecek ve dıştaki except çalışacak

dıştaki except çalışacak ve içinde yer alan try ve finally çalışacak

dıştaki finally çalışacak ve içinde yer alan try ve finally çalışacak

Scenario: statement1 throws Exception1

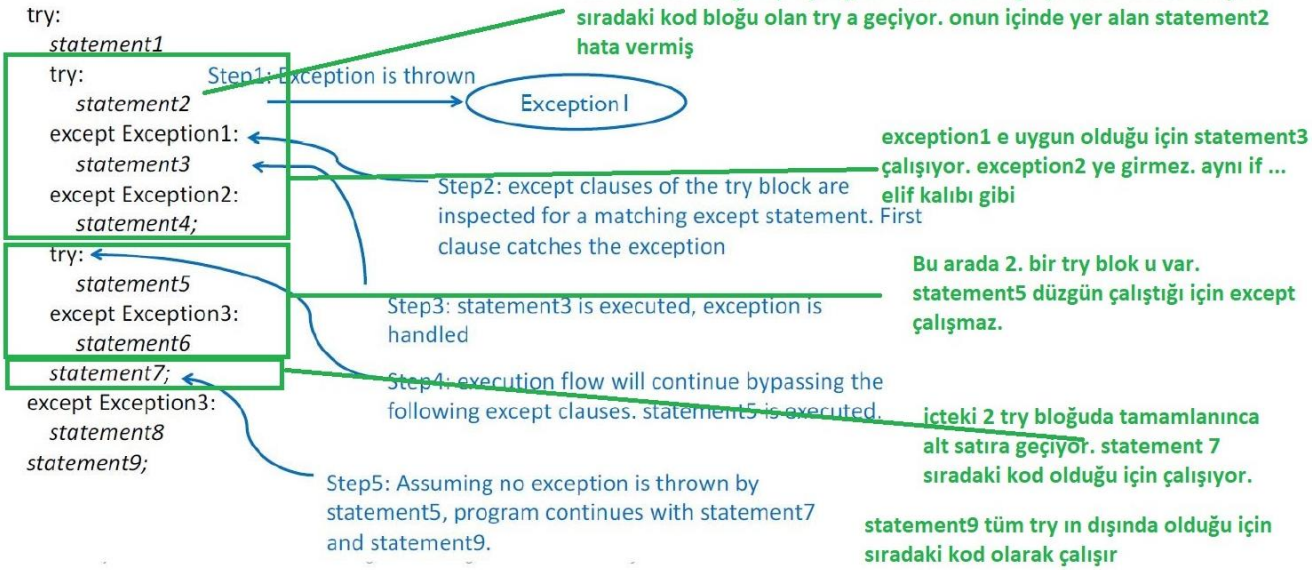


1. statementta hata verdiği için en dıştaki try a ait except Exception3 e gider.

statement8 çalışır.

statement9 girdi olarak try la aynı hizada olduğu için o kod try dışında o zaten çalışır

Scenario: statement2 throws Exception1



statement1 düzgün çalışmış, bir hata olmadığı için kod devam ediyor. sıradaki kod bloğu olan try a geçiyor. onun içinde yer alan statement2 hata vermiş

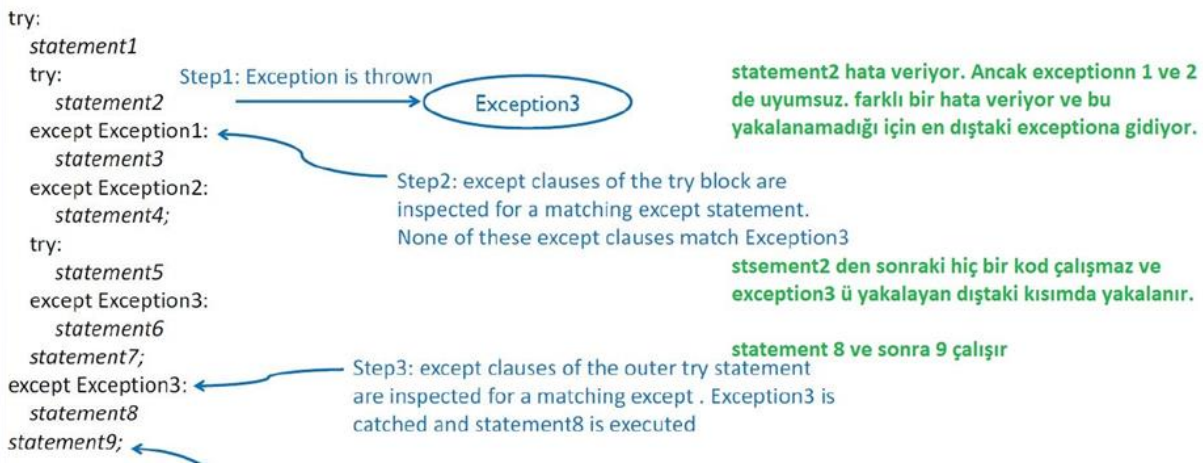
exception1 e uygun olduğu için statement3 çalışıyor. exception2 ye girmez. aynı if ... elif kalıbı gibi

Bu arada 2. bir try blok u var. statement5 düzgün çalıştığı için except çalışmaz.

içteki 2 try bloğuda tamamlanınca alt satıra geçiyor. statement 7 sıradaki kod olduğu için çalışıyor.

statement9 tüm try ın dışında olduğu için sıradaki kod olarak çalışır

Scenario: statement2 throws Exception3



statement2 hata veriyor. Ancak exceptionn 1 ve 2 de uyumsuz. farklı bir hata veriyor ve bu yakalanamadığı için en dıştaki exceptiona gidiyor.

stsement2 den sonraki hiç bir kod çalışmaz ve exception3 ü yakalayan dıştaki kısımda yakalanır.

statement 8 ve sonra 9 çalışır

MODÜLLER (MODULE)

Modül Nedir?

Modüllerin, bazı işlevleri kolaylıkla yerine getirmemizi sağlayan birtakım fonksiyonları ve nitelikleri içinde barındıran araçlar olduğunu söyleyebiliriz.

Python'ın fonksiyon sistemi, nasıl aynı dosya içinde tekrar tekrar kullanma imkanı veriyorsa, modül sistemi de bir fonksiyonu farklı dosyalar ve programlar içinde tekrar tekrar kullanma imkanı verir.

Eğer modül sistemi olmasaydı, biz bir kez yazdığımız fonksiyonunu başka bir programda da kullanmak istediğimizde, bu fonksiyonu alıp her defasında yeni programa elle kopyalamak zorunda kalırdık. Ama modül sistemi sayesinde, bir program içinde bulunan fonksiyonları başka Python programları içine 'aktarabiliyoruz'. Dolayısıyla modüller sayesinde, bir kez yazdığımız kodları pek çok farklı program içinde kullanma imkanı elde ediyoruz. Bu da bizim;

7- Daha az kod yazmamızı
8- Bir kez yazdığımız kodları tekrar tekrar kullanabilmemizi
9- Daha düzenli, daha derli toplu bir şekilde çalışabilmemizi sağlıyor.

Modüller iki farklı başlık altında incelenebilir:

1- Kendi tanımladığımız modüller
2- Hazır modüller
    a. Standart Kütüphane Modülleri
    b. Üçüncü Şahıs Modülleri

Standart Kütüphane Modülleri, doğrudan Python geliştiricileri tarafından yazılıp dile kaynaştırılmış modüllerdir. Bu yönüyle bu modüller daha önce öğrendiğimiz gömülü fonksiyonlara çok benzer. Biz bunları istediğimiz her an, herhangi bir ek yazılım kurmak zorunda kalmadan, kendi programlarımız içinde kullanabiliriz.

Modüllerin İçe Aktarılması

Python'da herhangi bir modülü kullanabilmek için öncelikle onu import etmemiz gerekir.

Bu modüle adını veren os kelimesi operating system (işletim sistemi) ifadesinin kısaltmasıdır. Bu modül, kullandığımız işletim sistemine ilişkin işlemler yapabilmemiz için bize çeşitli fonksiyonlar ve nitelikler sunar.

```
import os
if os.name != 'nt':
    print('Kusura bakmayın! Bu programı yalnızca',
          'Windows\'ta kullanabilirsiniz!')
else:
    print('Hoşgeldin Windows kullanıcısı!')
```

Hoşgeldin Windows kullanıcısı!

Farklı İçe Aktarma Yöntemleri

1- import modul_Adi as farkli_Isim

```
subprocess.call('notepad.exe')
```

Bu şekilde 'Notepad' programını Python içinden çalıştırmış olduk. Ancak gördüğünüz gibi, 'subprocess' biraz uzun bir kelime. Eğer isterseniz modülü import subprocess şeklinde kendi adıyla değil de daha kısa bir adla içe aktarmayı tercih edebilirsiniz:

```
import subprocess as sP
```

Burada şöyle bir formül uyguladığımıza dikkat edin:

```
import modül_Adi as farkli_Isim
```

2- from modul_Adi import isim1, isim2

Eğer arzu ederseniz, import os gibi bir komutla bütün o isimleri içe aktarmak yerine, yalnızca kullanacağınız isimleri içe aktarmayı tercih de edebilirsiniz. Mesela os modülünün yalnızca name niteliğini kullanacaksanız, modülü şu şekilde içe aktarabilirsiniz:

```
from os import name
```

```
os.name
NameError: name 'os' is not defined
```

Neden hata aldık, çünkü biz from os import name komutunu verdiğimizde, os modülünü değil, bu modül içindeki bir nitelik olan name'i içe aktarmış oluyoruz. Dolayısıyla os ismini kullanamıyoruz.

3- from modul_Adi import isim as farklı_Isim

Bir modül içinden belli nitelik ve fonksiyonları farklı bir adla içe aktarmak için bu yapıyı kullanırız.

```
from os import name as isim
```

os modülü içinden name adlı niteliği isim adıyla içe aktardık. Böylece name niteliğini isim adıyla kullanabiliriz.

4- from modul_Adi import *

Bu şekilde bir modül içindeki bütün fonksiyon ve nitelikleri içe aktarmış oluruz.

```
from sys import *
```

Böylece sys modülü içindeki bütün fonksiyon ve nitelikleri, başlarına modül adını eklemeye gerek olmadan kullanabiliriz.

NOT: Bir modülü yıldızlı olarak içe aktaracaksak, bu işlemi lokal etki alanları içinden gerçekleştiremeyiz. Örneğin aşağıdaki örnek yanlış bir kullanıma sahiptir;

```
def fonksiyon():
    from os import *
```