**MAT 310**

**Programlama Dilleri I – Python**

**Dr. Öğretim Üyesi Rıfat AŞLIYAN**

**Oda:**

**C Blok 125 nolu oda**

**Web Sayfası:**

[**http://www.rifatasliyan.com**](http://www.rifatasliyan.com/)

**E-Posta**

[**rasliyan@adu.edu.tr**](mailto:rasliyan@adu.edu.tr)

[**rasliyan@yahoo.com**](mailto:rasliyan@yahoo.com)

**Dersin Amacı**

Python programlama diliyle nasıl program yazılabileceğini öğretmektir.

**Konular:**

* Python ile genel bilgiler, tarihi ve özellikleri
* Python değişkenleri, operatörler
* Python karar yapıları, döngü yapıları, break ve continue
* Python yorumları

**Öğrenme Çıktıları**

1.

**Dersle İlgili Kaynaklar**

**Sınavlar ve Ödevler**

* 1 arasınav (%40), 1 Final (%60) ve 1 Bütünleme(%60) şeklinde olacak.
* Klasik sınav olacak
* Test olmayacak
* Bazı derslerin sonunda ödevler verilecek.

**Python Nedir?**

* **Python yüksek seviyeli programlama dilidir:** Yani, konuşma diline çok yakındır.
* **Yorumlayıcılıdır:** Derleyicisi yoktur. Yorumlayıcısı ile çalışma zamanında kodlar makine diline dönüştürülerek programlar çalıştırılır.
* **Etkileşimlidir:** Python, komut penceresinde kullanıcı ile yorumlayıcı arasında etkileşimli olarak kod yazmaya izin verir.
* **Nesneye dayalıdır:** Nesneye dayalı kod yazmaya izin verir. Nesneler oluşturulabilir.
* **Yeni başlayanlar için uygun bir dildir:** Hem öğrenmesi kolay bir dildir hem de her türlü program yazmaya elverişlidir.
* Python programlama dili, **Guido van Rossum** tarafından **1985-1990** arasında Hollandanın Ulusal Matematik ve Bilgisayar Araştırma Enstitüsünde geliştirildi.
* Python için iki ana versiyon vardır: **Python 2** ve **Python 3**
* Python 3 2008’de piyasaya sürüldü

**Python Özellikleri**

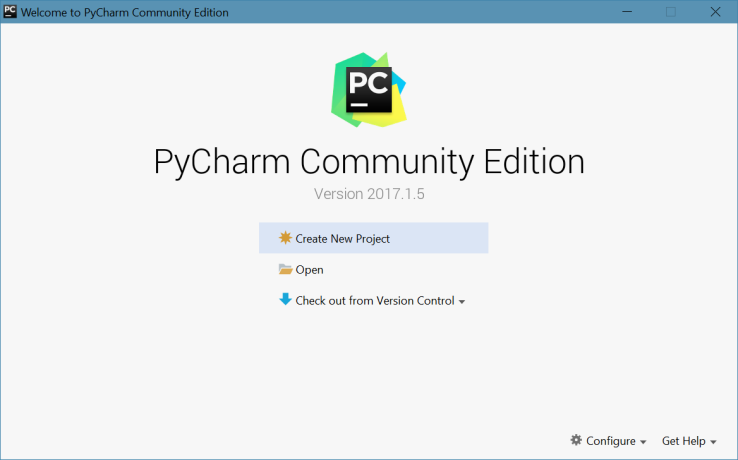
* **Öğrenmesi kolay:** Öğrenciler kolay ve hızlı bir şekilde python’u öğrenebilir. Az sayıda anahtar sözcüğü var. Yapıları basit ve kolay bir şekilde tanımlanan yazımı vardır.
* **Okuması kolay:** Başka bir programcının yazdığı python kodlarının okunarak anlaşılması kolaydır.
* **Kodların yeniden düzenlenmesi kolay:** Yazılmış python kodlarının zamanla şartlara göre değiştirmek gerekir. Python ile bu kodların değiştirilmesi kolaydır.
* **Geniş standart kütüphaneleri var:** Bir çok python kütüphaneleri farklı platformlarda (Unix, Windows ve Macintosh) kullanılabilmektedir.
* **Etkileşimli moda sahip:** Python kod bloklarının etkileşimli olarak test ve hata bulup düzeltilmesi gerçekleştirilebilir.
* **Taşınabilir:** Python, farklı donanım platformlarında da çalışabilir ve aynı kullanıcı arabirimi ile çalışabilir.
* **Genişletilebilir:** Python yorumlayıcısına düşük seviyeli modüller eklenerek programcıların daha verimli kodlar yazmaları sağlanır.
* **Veritabanları desteklenir:** Bütün gelişmiş veritabanlarına arabirimler sağlanır.
* **GUI programlamaya izin verir:** Yani grafiksel kullanıcı arabirimi uygulamalarının geliştirilmesini destekler.
* **Ölçeklenebilir:** Python, büyük programlar geliştirilmesinde daha iyi yapı ve destek sağlar.

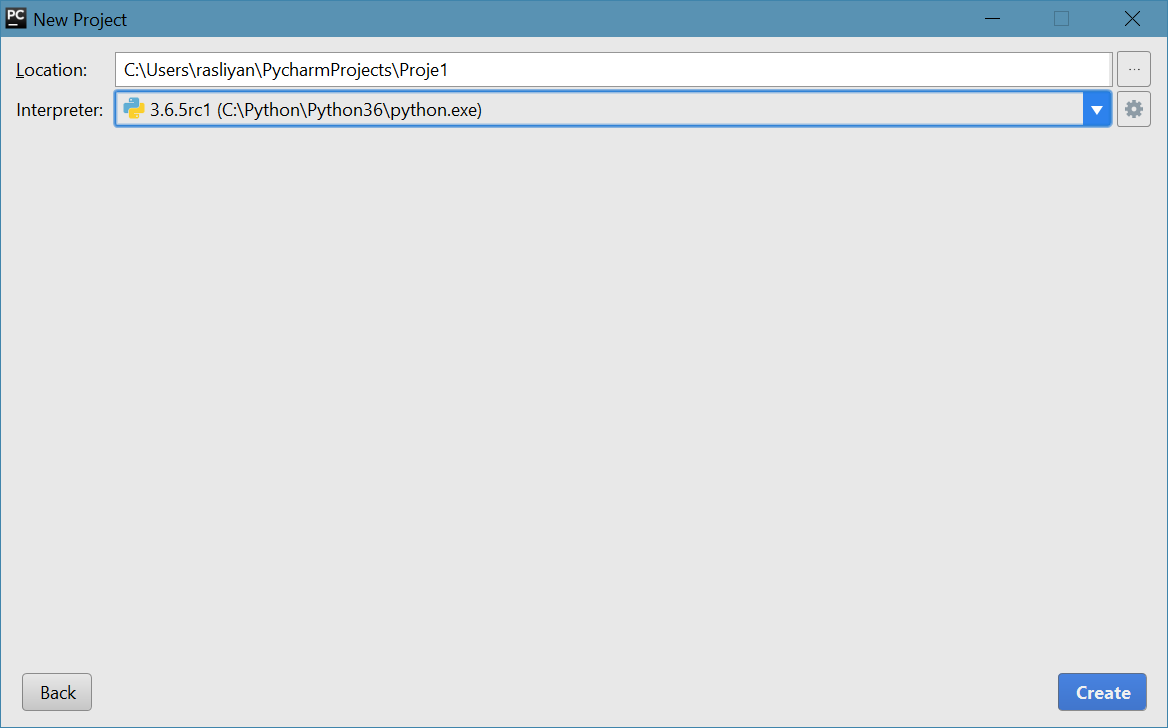
**Yukardaki özelliklere ek olarak şu özelliklere sahiptir:** Fonksiyonel, yapısal ve nesneye dayalı programlamayı destekler. Yorumlayıcı dili olmasıyla birlikte büyük ölçekli programlar geliştirilmesinde byte-kodlara derleme imkanıda sağlar. Dinamik veri tiplerini ve dinamik tip kontrolü de yapılır. Otomatik çöp toplama (Automatic Garbage Collection) özelliği de vardır. Bazı önemli programlama dilleriyle (C, C++, Java) kolay bir şekilde entegre edilebilir.

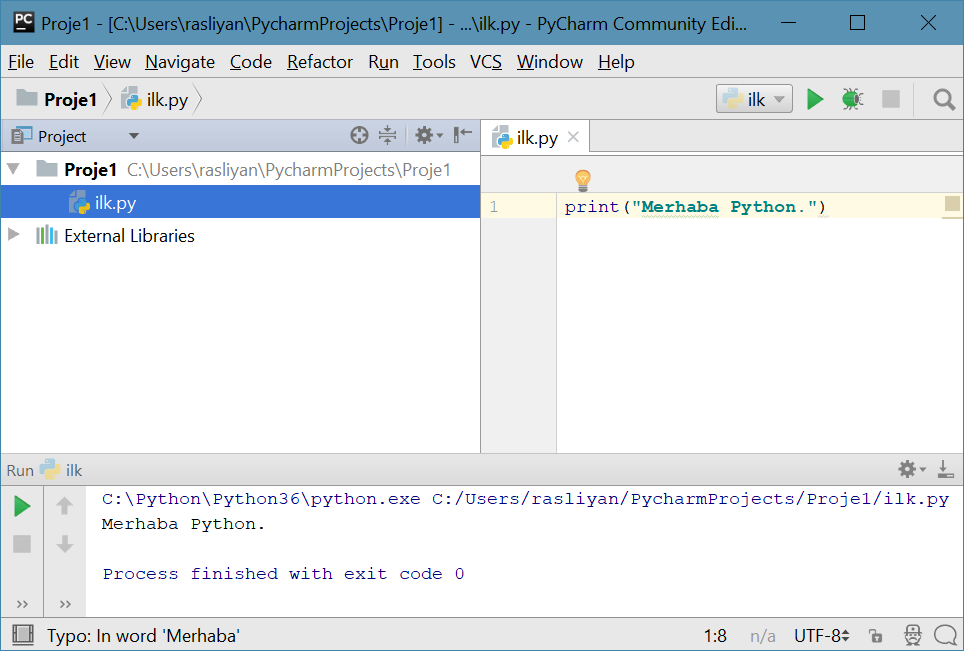
**İlk Python Uygulaması**

Python uygulamalarını **PyCharm** editör programlarıyla yazacağız. PyCharm’ı ilk açtığımızda aşağıdaki gibi bir pencere gelececektir. “Create new Project” seçilerek bir alt pencerece görüldüğü gibi Proje adını gireriz. Proje1 adını verdik. “Create” diyerek Proje1’i açarız. Proje1’e sağ tıklayıp “new” den “Python File” seçilir. Açılan pencereden de dosya ismini örneğin “ilk” yazıp “Ok” tıklanır. “ilk.py” adında dosya oluşur. Kodları burada yazarız. Python dosya uzantısı “py” dir.

Aşağıdaki resimde görüldüğü gibi **print (“Merhaba Python”)** yazalım. Bu python programını çalıştırmak için “Run” menüsünden “Run” seçeneği tıklanır ve “ilk.py“ programı seçilir. Böylece bu programın kodları çalışmış olur. Ekran çıktısında da “Merhaba Python” yazar.







Python programlama dili, Perl, C ve Java dilleriyle çok benzerlikleri vardır. Fakat, bu diller arasında da belirli farklılıklar da vardır.

**Python programlarının çalıştırılmasında iki çalıştırma modu vardır: Kullanıcı arabirimi modu (Interactive Mode) ve Yazım Modu (Script Mode)**

**Kullanıcı Arabirimi Modu:**

Kullanıcı arabirimi moduyla, kodları metin dosyasına yazmaksızın, Python komut penceresinde kodlar yazılarak çalıştırılır. Aşağıda bu moddaki kullanım gösterilmektedir. Bu kod çalıştırıldığında (Enter’a basılır) ekranda “Merhaba Python” yazacaktır.

>>> print ("Merhaba Python")

>>> Merhaba Python

**Yazım Modu:**

Yazım modunda Python kodları .py uzantılı dosya içine kodlar yazılır ve çalıştırılır. Örneğin “Test.py” dosyası içinde “print (“Merhaba Python”)” yazılıp çalıştırıldığında “Merhaba Python” yazacaktır.

**Python Tanımlayıcıları:**

Python tanımlayıcıları, değişken, sabit, fonksiyon, sınıf, modul veya diğer nesneleri tanımlamak için kullanılan isimlerdir. Tanımlayıcılar ingilizce harflerle (A-Z veya a-z) yada altçizgi ( \_ ) ile başlar. Sonrasında yine harfler, altçizgi veya rakamlar (0-9) kullanılabilir. Python diğer karakterleri tanımlayıcı olarak kabul etmez. Python, küçük-büyük harf duyarlılığı (Case Sensitive) olan bir dildir. Örneğin “Kitap” ve “kitap” iki farklı Python tanımlayıcıdır.

**Python İsimlendirme Geleneği (Python Naming Conventions):**

* Sınıf (Class) isimleri büyük harfle başlar. Diğer bütün tanımlayıcılar küçük harfle başlar.
* Bir altçizgi ile başlayan tanımlayıcı, tanımlayıcının özel bir tanımlayıcı olduğunu gösterir.
* İki altçizgi ile başlayan tanımlayıcı, tanımlayıcının güçlü özel bir tanımlayıcı olduğunu gösterir.
* Eğer tanımlayıcı iki altçizgi ile bitiyorsa, bu tanımlayıcı ise dil tanımlı özel tanımlayıcıdır.

**Anahtar Kelimeler**

Aşağıdaki liste, Python anahtar kelimeleri gösterir. Bu kelimeler Python için özel kelimelerdir. Bu kelimelerden sabit, değişken, fonksiyon veya diğer tanımlayıcı isim kullanılamaz. Aksi halde hata verir. Bütün Python anahtar kelimeleri küçük harflidir.

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| as | finally | or |
| assert | for | pass |
| break | from | print |
| class | global | raise |
| continue | if | return |
| def | import | try |
| del | in | while |
| elif | is | with |
| else | lambda | yield |
| except |  |  |

**Satır ve Girinti**

Python, sınıf, fonksiyon, karar ve döngü yapılarınadaki blokları ifade etmede küme parantezleri kullanmaz. Bunun yerine satır girintilerini kullanır. Kaç karakterlik girinti kullanılacağı önemli değil, fakat aynı blok içinde olan kodlar aynı girinti seviyesinde olmalıdır. Aşağıdaki kod doğru girinti kullanıldığı için doğru çalışacaktır.

if True:

print ("True")

else:

print ("False")

Fakat aşağıdaki kod hata verecektir. Çünkü if karar yapısının bloğu girintili yazılmamıştır.

if True:

print ("True")

else:

print ("False")

**Çok Satırlı İfadeler:**

Python’da ifadeler yeni satırla biter. Fakat, satırın altındaki satırdan ifadenin devam etmesini istiyorsak (\) ters slash karakteri kullanmamız gerekiyor. Aşağıdaki örnekte bir ifade alt alta üç satırda devam etmektedir.

total = item\_one + \

item\_two + \

item\_three

Parantez, köşeli parantez ve küme parantezi içeren ifadeler için satır devam karakteri (\) kullanmaya gerek yoktur. Aşağıdaki her iki ifade de doğru çalışır.

gunler = ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma']

gunler = ['Pazartesi', 'Salı', 'Çarşamba',

'Perşembe', 'Cuma']

Python’da Tırnak İşaretleri

Python, tek tırnak (‘), çift tırnak (“) ve üç tırnak (''' veya """) metinleri tanımlamak için kullanılabilir. Her üçünü de doğru kabul eder. Üç tırnak, çok satırlı metinler için de kullanılabilir. Aşağıda bunlara örnek verilmiştir. kelime, cumle ve paragraf metin değişkenleridir. paragraf, çok satırlı olabilen metin değişkenidir.

kelime = 'kelime'

cumle = "Bu bir cümledir."

paragraf = """Bu bir paragraftır. Bu paragraf

birçok satır ve cümleden oluşabilir."""

**Python’da Yorumlar**

Pyhon’da yorumlar (#) (kare yada diyez) karakteriyle yapılır. Diyez karakterinden sonra satır sonuna kadar yazılan her şey yorumdur ve bilgisayar tarafından dikkate alınmaz. Yani kodun açıklanmasını sağlar. Kodun okunabilirliğini artırır. Aşağıda örnekte görülmektedir.

# İlk yorum

print ("Merhaba Dünya") # Merhaba Python diye yazar – İkinci yorum

Bu kod çalıştığında, aşağıdaki gibi sonuç görünür.

Merhaba Dünya

Python, çok satırlı yorumu desteklemez. Çok satırlı yourm için her satırda mutlaka (#) karakteri kullanmak gerekir. Aşağıdaki örnekte iki satırlık yorum var.

# İlk yorum

# İkinci yorum

**Boş Satır Kullanımı**

Boş alan içeren bir satır, boş satır olarak adlandırılır ve Python onu dikkate almaz.

Kullanıcı İçin Bekleme

Aşağıdaki kod ekrana “Programdan çıkmak için bir tuşa basın!” diye yazar ve kullanıcının bir tuşa basması için bekler. Konsol uygulamalarında kodların en sonunda aşağıdaki gibi input komutu kullanarak konsol penceresinin otomatik kapanmasını önler.

input("\n\nProgramdan çıkmak için bir tuşa basın!\n")

“\n” karakteri komut penceresinde imlecin bir alt satıra geçmesini sağlar. Kaç tane “\n” varsa o kadar alt satıra geçer.

**Tek Satırda Birçok İfade Kullanımı**

Her ifadenin sonunda “;” noktalı virgül kullanarak bir den fazla ifade Python tarafından çalıştırılabilir, hata vermez. Aşağıda buna örnek verilmiştir. Aşağıdaki kodda üç tane ifade bulunmaktadır. Bunlar arasında noktalı virgül kullanıldığından hatalı bir işlem yoktur. Python tarafından doğru bir şekilde çalıştırılır.

import sys; deg = 'Merhaba Python'; sys.stdout.write(deg + '\n')

**Birçok İfade Grubu (Suit)**

Bir kod bloğunu oluşturan ifadeler grubuna **suit** adı verilir. if, while, for, def, class gibi birleşik ifadeler (Compound Statements) bir başlık satırı ve bir de suit içerirler. Başlık satırı, anahtar kelime ile başlar ve “:” üstüste iki nokta ile biter. Sonra başlık satırını, bir veya birden çok satırlı ifadelerden oluşan suitler takip eder.

if ifade1 :

suite

elif ifade2 :

suite

else :

suite

**Değişken Tipleri**

Değişken, verilerin saklandığı bellek alanına verilen isimdir. Bir değişken tanımlandığında bu değişken için bir yer ayrılır.

Değişkenin veri tipine bağlı olarak yorumlayıcı, reserve edilmiş alana ne kaydedileceğine karar verir. Bu alana, veri tipine göre tamsayı, karakter yada ondalıklı sayı kaydedilebilir.

**Değişkenlere Değer Atama**

Python’da değişkenlerin açıkça tanımlanmasına gerek yoktur. Bir değişkene değer atandığında otomatik olarak bu değişken tanımlanır. Değer atama operatörü eşittir (=) operatörüdür.

sayac = 1 # Tamsayı atama

metre = 10.0 # Ondalıklı sayı atama

isim = "Üniversite" # Metin atama

print (sayac)

print (metre)

print (isim)

Yukardaki kod çalıştırıldığında aşağıdaki çıktıyı verecektir.

1

10.0

Üniversite

**Birden Çok Atama**

Python birden fazla değişkene bir değerin atanmasına izin verir.

Örneğin aşağıdaki kod ile a, b ve c değişkenlerine 1 değeri atanmıştır.

a = b = c = 1

Örneğin aşağıdaki örnekte ise a değişkenine 1, b değişkenine 2 değeri ve c deişkenine ise “kitap” değeri atanmıştır.

a = b = c = 1, 2, “kitap”

**Standart Veri Tipleri**

Bellekte depolanan veri birçok farklı tipte olabilir. Örneğin bir kişinin yaşı sayı tipinde; adı veya adresi alfanümerik (metin) tipinde olabilir. Python 5 farklı standard veri tipinde değişkenler tanımlamasına izin vermektedir.

* Sayılar (Numbers)
* Metin (String)
* Liste (List)
* Tuple (İkili)
* Sözlük (Dictionary)

**Python Sayıları**

Sayı veri tipleri sayısal verilerin bellekte depolanmasını sağlar. Sayısal bir değer atandığında sayı tipinde bir değişken tanımlanır. Aşağıdaki örnekte deger1 ve deger2 adında iki sayı tipinde değişkenler tanımlandı.

deger1=5

deger2=7

Aşağıda del komutunun genel yazım şekli gösterilmektedir.

del deger1[,deger2[,deger3[....,degerN]]]]

Aşağıda gösterildiği gibi del komutu ile bu değişken nesnelerini bellekten tamamiyle silebiliriz. Örnekte deger1, a ve b değişkenleri bellekten silinir.

del deger1

del a, b

Python 3 farklı sayı tipini destekler.

* int (işaretli tamsayılar)
* float (ondalıklı tamsayılar)
* complex (kompleks sayılar)

Aşağıdaki tabloda int, float ve complex tipinde değerler örnek olarak verilmiştir.

|  |  |  |
| --- | --- | --- |
| **int** | **float** | **complex** |
| 10 | 0.0 | 3.14j |
| 100 | 15.20 | 45.j |
| -786 | -21.9 | 9.322e-36j |
| 080 | 32.3+e18 | .876j |
| -0490 | -90. | -.6545+0J |
| -0x260 | -32.54e100 | 3e+26J |
| 0x69 | 70.2-E12 | 4.53e-7j |

Komplex (complex) sayılar x+yj şeklinde yazılan sıralı ikililerden oluşur. x ve y değerleri reel sayılardan oluşur.

[**Python Metinleri (Strings**](https://www.tutorialspoint.com/python3/python_strings.htm)**)**

Pythonda metin değerleri tırnak içerisindeki karakter dizilerinden oluşur. Python, tek ve çift tırnağı destekler. Bir metnin alt parçaları köşeli parantez [ ] ile elde edilebilir. Üstüste iki nokta (:) ile bir indisten sonraki indise olan bütün karakterler seçilebilir. İndis 0 dan başlar. Metinlerde, + operatörü ekleme işlemi, \* operatörü tekrar işlemi olarak kullanılır.

deger= 'Merhaba Python'

print (deger) # Bütün metni yazdırır

print (deger[0]) # deger metninin ilk karakterini yazdırır

print (deger[2:5]) # 3. karakterden 5. Karaktere kadar yazdırır

print (deger [2:]) # 3. karakterden son karaktere kadar yazdırır

print (deger \* 2) # deger metnini iki defa ekleyerek yazdırır

print (deger + " deneme") # + ile iki metin eklenir ve yazdırır

Aşağıdaki sonuçlar ekrana yazdırılır:

Merhaba Python

M

rha

rhaba Python

Merhaba PythonMerhaba Python

Merhaba Python deneme

**Python Listeler (Lists)**

Listeler, köşeli parantezle kullanılır ve liste elemanları virgüllerle ayrılır. Farklı veri tiplerinden verilerden de oluşabilir.

Aşağıda listelerle ilgili örnekler verilmiştir.

liste = [ 'abcd', 786 , 2.23, 'kitap', 70.2 ]

kucukliste = [123, 'kitap']

print (liste) # Bütün listeyi yazdırır

print (liste[0]) # Listenin ilk elemanını yazdırır

print (liste[1:3]) # Listenin 2. elemanından 3. elemanına kadar yazdırır

print (liste[2:]) # Listenin 3. elemanından son elemana kadar yazdırır

print (kucukliste \* 2) # kucukliste’yi iki defa ard arda ekleyerek yazdırır

print (liste+ kucukliste) # Liste ile kucukliste eklenerek yazdırılır

Yukardaki kod çalıştığında ekrana aşağıdaki gibi yazacaktır:

['abcd', 786, 2.23, 'kitap', 70.200000000000003]

abcd

[786, 2.23]

[2.23, 'kitap', 70.200000000000003]

[123, 'john', 123, 'kitap']

['abcd', 786, 2.23, 'kitap', 70.200000000000003, 123, 'kitap']

**Python’da Demet (Tuple)**

Demet, listelere benzeyen bir başka dizi dipidir. Demet, virgüllerle ayrılan değerler kümesinden oluşur. Listelerden farklı olarak değerler parantezler () içine alınır; köşeli parantezlerle değil.

Listelerle demetler arasındaki temel fark şudur: Listelerler köşeli parantezler içinde ifade edilir; onun elemanları ve boyutu değişebilir. Fakat, demetler parantez içinde yazılır; değerleri ve boyutu değiştirilemez. Demetler sadece okuma amacıyla kullanılır.

demet = ( 'abcd', 786 , 2.23, 'kitap', 70.2 )

kucukdemet = (123, 'kitap')

print (demet) # Bütün demeti yazdırır

print (demet[0]) # Demetin ilk elementini yazdırır

print (demet[1:3]) # İkinciden 3’üncüye kadar elemanları yazdırır

print (demet[2:]) # Üçüncüden son elemana kadar yazdırır

print (kucukdemet \* 2) # Demeti iki defa yazdırır

print (demet + kucukdemet) # İki demeti ekleyerek yazdırır

Yukardaki kodlar çalıştırıldığında aşağıdaki sonuçlar ekranda görünür.

('abcd', 786, 2.23, 'kitap', 70.200000000000003)

abcd

(786, 2.23)

(2.23, 'kitap', 70.200000000000003)

(123, 'kitap', 123, 'kitap')

('abcd', 786, 2.23, 'kitap', 70.200000000000003, 123, 'kitap')

Aşağıdaki kod demetler için yanlıştır. Çünkü demetleri güncellemeye çalıştık. Demetlerde verilerin değiştirilmesine izin verilmez. Benzer durumda listeler için problem oluşturmaz yani listelerde güncelleme mümkündür.

demet = ( 'abcd', 786 , 2.23, 'kitap', 70.2 )

liste = [ 'abcd', 786 , 2.23, 'kitap', 70.2 ]

demet[2] = 1000 # Demetle hatalı yazım şekli; hata verir

liste[2] = 1000 # Listelerde doğru yazım; hata vermez

**Pythonda Sözlük (Dictionary)**

Pythonda sözlükler, komut tablosu (hash table) tipinin bir türüdür. Sözlükler, anahtar değer çiftlerinden oluşur. Sözlük anahtarı herhangi bir Python tipi olabilir, fakat genelde sayılar ve metinlerden oluşur. Bununla birlikte sözlük değerleri ise herhangi bir Python nesnesi olabilir.

Sözlükler, küme parantezi { } içinde kullanılır ve değerler köşeli parantezler [ ] ile atanabiliri ve erişilebilir.

sozluk = {}

sozluk['bir'] = "Bu ilkidir"

sozluk[2] = "Bu ikincidir"

kucuksozluk = {'isim': 'veli','kod':111, 'bolum': 'matematik'}

print (sozluk['bir']) # 'bir' anahtarı için değer yazılır

print (sozluk[2]) # 2 anahtarı için değer yazılır

print (kucuksozluk) # Bütün sözlük yazılır

print (kucuksozluk.keys()) # Bütün anahtarlar yazılır

print (kucuksozluk.values()) # Bütün değerler yazılır

Yukardaki kodlar çalıştığında ekrana aşağıdaki gibi yazdırılır.

Bu ilkidir

Bu ikincidir

{'isim': 'veli', 'kod': 111, 'bolum': 'matematik'}

dict\_keys(['isim', 'kod', 'bolum'])

dict\_values(['veli', 111, 'matematik'])

Sözlüklerdeki elemanlar sıralı değildir.

**Veri Tipi Dönüşümü**

Biz bazen veri tipi dönüşümleri için Python tip dönüşüm fonksiyonlarına ihtiyaç duyabiliriz. Tipler arası dönüşüm yapmak için tip ismiyle aynı olan fonksiyonları kullanmak gerekir.

Aşağıda tip dönüşümüyle ilgili fonksiyonlar aşağıda gösterilmektedir:

|  |  |
| --- | --- |
| No | Fonksiyonlar ve Açıklamaları |
| 1 | int(x [,taban]) # x’i tam sayıya çevirir. Eğer x metin ise taban belirtilerek dönüşüm de yapılır |
| 2 | float(x) # x’i reel sayıya dönüştürür |
| 3 | complex(Real x [, Imaginary y])# Karmaşık sayı oluşturur |
| 4 | str(x) # x nesnesini metne dönüştürür |
| 5 | repr(x) # x nesnesini ifade metnine dönüştürü |
| 6 | eval(str) # str metnini değerlendirip bir nesneye dönüştürü |
| 7 | tuple(s) # s’yi demet’e (tuple) dönüştürür |
| 8 | list(s) # s’yi listeye dönüştürür |
| 9 | set(s) # s’yi kümeye (set) dönüştürür |
| 10 | dict(d) #Bir sözlük oluşturur. d, (anahtar, değer) demet dizisi olmalıdır |
| 11 | frozenset(s) # s’yi frozen set’e dönüştürür |
| 12 | chr(x) # Tamsayıyı karaktere dönüştürür |
| 14 | ord(x) # Tek karakteri onun tamsayı değerine dönüştürür |
| 15 | hex(x) # Tamsayıyı heksadesimal (16’lı sayı) sayıya dönüştürür |
| 16 | oct(x) # Tamsayıyı oktal (8’li sayı) sayıya dönüştürür |

**Temel Operatörler**

Operatörler, değerlerin değiştirilmesini sağlayan yapılardır.

Operatör Tipleri: Aritmetik Operatörler, Karşılaştırma Operatörleri, Atama Operatörleri, Mantıksal Operatörleri, İkili Tabandaki Operatörler, Üyelik Operatörleri, Kimlik Operatörleri

**Python Aritmetik Operatörleri**

Toplama operatörü: +

Çıkarma operatörü: -

Çarpma operatörü: \*

Bölme operatörü: /

Mod alma operatörü: %

Üs alma operatörü: \*\*

Aşağıya yuvarlayarak bölme: //

**Örnekler:**

x=4+5

x=6-3

x=3\*4

x=8/2

x=6%4

x=4\*\*2

x=5.3//2

**Karşılaştırma Operatörleri**

== Eşit olup olmadığını kontrol eder; eşit ise True değilse False döndürür

!= Farklı olup olmadığını kontrol eder; farklı ise True değilse False dödürür

> Büyük ise True, değil ise False döndürür

>= Büyük veya eşit ise True, değil ise False döndürür

< Küçük ise True, değil ise False döndürür

<= Küçük veya eşit ise True, değil ise False döndürür

**Örnekler:**

print(2==2) # Ekranda True yazar

print(4==9) # Ekranda False yazar

print(2!=2) # Ekranda True yazar

print(2>2) # Ekranda False yazar

print(2>=2) # Ekranda True yazar

print(5<=7) # Ekranda True yazar

**Atama Operatörleri**

= Atama operatörü, sağındaki işlemin sonucunu yada değeri soldaki değişkene atar.

+= Toplama ve atama operatörü; eşitin sağındaki değer soldaki değişkene eklenir

-= Çıkarma ve atama operatörü; eşitin sağındaki değer soldaki değişkenden çıkarılır

\*= Çarpma ve atama operatörü; eşitin sağındaki değer soldaki değişken ile çarpılır ve bu değişkene atanır.

/= Bölme ve atama operatörü; soldaki değişken, eşitin sağındaki değer ile bölünür ve sonuç bu değişkene atanır.

%= Mod ve atama operatörü; soldaki değişkenin, eşitin sağındaki değer ile modu alınır ve bulunan kalan bu değişkene atanır.

\*\*= Üs ve atama operatörü; soldaki değişken, eşitin sağındaki değer ile üssü alınır ve sonuç bu değişkene atanır.

//= Aşağıya yuvarlayarak bölme ve atama operatörü; soldaki değişken, eşitin sağındaki değer ile bölünür, sonra aşağıya yuvarlanır ve sonuç bu değişkene atanır.

**Örnekler:**

x=3  
print(x) # Ekranda 3 yazar  
x+=1  
print(x) # Ekranda 4 yazar  
x-=2  
print(x) # Ekranda 2 yazar  
x\*=3  
print(x) # Ekranda 6 yazar  
x/=2  
print(x) # Ekranda 3.0 yazar  
x\*\*=3  
print(x) # Ekranda 27.0 yazar  
x%=10  
print(x) # Ekranda 7.0 yazar  
x//=3  
print(x) # Ekranda 2.0 yazar

**İkili Tabanda (Bitwise) Operatörler**

İkili tabanda operatörler: | Veya operatörü, & Ve operatörü, ^ XOR operatörü, ~ Bire tümler operatörü, << Sola kaydırma operatörü ve >> Sağa kaydırma operatörü

bin(x) # x tamsayısını ikili tabana dönüştürür

a=60; b=55  
print(bin(a)+' '+bin(b))# Ekranda 0b111100 0b110111 yazar  
x= a & b  
print(bin(x)) # Ekranda 0b110100 yazar  
x= a | b  
print(bin(x)) # Ekranda 0b111111 yazar  
x= a ^ b  
print(bin(x)) # Ekranda 0b1011 yazar  
x= ~a  
print(bin(x)) # Ekranda -0b111101 yazar, bu sayı aslında 0b000011 dır  
x= a << 1  
print(bin(x)) # Ekranda 0b1111000 yazar, bir bit sola kaydırır sağdan 0 gelir  
x= a >> 1  
print(bin(x)) # Ekranda 0b11110 yazar, sağa bir bit kaydırır, soldan 0 gelir

**Mantıksal Operatörler**

Mantıksal operatörler: and ve işlemi; or veya işlemi; not değil işlemi yapar

**Örnekler:**

a=True ; b=False  
print(a and b)# Ekranda False yazar  
print(a or b)# Ekranda True yazar  
print(not a)# Ekranda False yazar

**Üyelik operatörleri:**

Üyelik operatörleri: **in** içinde olduğunda True, içinde olmadığında False döndürür. Metin, liste veya demetlerde kullanılabilir; **not in** ise içinde olmadığında False döndürür.

Örnekler:

liste=[2,5,7,9]; x=7  
print(x in liste) # Ekranda True yazar, çünkü 7  
x=8  
print(x in liste) # Ekranda False yazar  
x=8  
print(x not in liste) # Ekranda True yazar

**Kimlik (Identity) Operatörleri**

Kimlik operatörleri, iki nesnenin bellekteki yerlerini karşılaştırır. Eğer iki nesnenin bellekteki yerleri aynı ise True, farklı ise False döndürür. İki tane kimlik operatörür vardır: **is** İki nesnenin bellekteki yeri aynı olduğunda True döndürür ve **is not** iki nesnenin bellekteki yerleri farklı olduğunda True döndürür.

**Örnekler:**

class Dikdortgen:  
 uzunluk=10  
 genislik=5  
  
x = Dikdortgen()  
x.uzunluk=20  
x.genislik=15  
y = Dikdortgen()  
print(y is x) # Ekranda False yazar  
print(y is not x) # Ekranda True yazar  
y=x  
print(y is x) # Ekranda True yazar  
print(y.uzunluk, y.genislik) # Ekranda 20 15 yazar

**Operatör Önceliği**

|  |  |
| --- | --- |
| Öncelik Sırası | Operatörler ve Açıklama |
| 1 | **\*\*** Üs alma operatörü |
| 2 | **~ + -** 1’e tümler, artı ve eksi işaretleri (Sayıları pozitif veya negatif yapar) |
| 3 | **\* / % //** Çarpma, bölme, mod ve aşağıya yuvarlayarak bölme |
| 4 | **+ -** Toplama, çıkarma |
| 5 | **>> <<** İkili tabanda sağa kaydırma, sola kaydırma |
| 6 | **&** İkili tabanda ve operatörü |
| 7 | **^ |** İkili tabanda xor işlemi, veya operatörleri |
| 8 | **<= < > >=** Karşılaştırma operatörleri |
| 9 | **== !=** Eşitlikve eşitsizlik öperatörleri |
| 10 | **= %= /= //= -= += \*= \*\*=** Atama operatörleri |
| 11 | **is is not** Kimlik operatörleri |
| 12 | **in not in** Üyelik operatörleri | |
| 13 | **not or and** Mantıksal operatörler | |

**Karar Yapıları**

Belirli şartlara göre belirli kodların çalıştırılmasını yada çalıştırılmamasını sağlayan yapılardır. Karar yapıları, doğru (True) yada yanlış (False) çıktı veren bir çok ifade içerebilir.

Aşağıdaki şekil çoğu programlama dilinde karar yapılarının genel yapısını göstermektedir.

Şart

Şart Sağlandığında Çalışacak Kodlar

Şart doğru olduğunda

Python dilinde sıfırdan farklı ve null’dan farklı bütün değerler True; sıfır ve null değerler False olarak kabul edilir.

Aşağıdaki tablo if karar yapılarını açıklamaktadır.

|  |  |
| --- | --- |
| if seçenekleri | İfadeler ve açıklamaları |
| 1 | **if şartlar** : if karar yapısında şartlar doğru olduğunda devamındaki kodlar çalışır. |
| 2 | **if şartlar...else**  : if’den sonraki şartlar doğru olduğunda devamındaki kodlar çalışır, yanlış olduğunda else’nin devamındaki kodlar çalışır. |
| 3 | **if şartlar...elif şartlar**  : if’den sonraki şartlar doğru olduğunda devamındaki kodlar çalışır, yanlış olduğunda elif’in şartları doğru ise devamındaki kodlar çalışır. |
| 4 | **if şartlar...elif şartlar… else**  : if’den sonraki şartlar doğru olduğunda devamındaki kodlar çalışır, yanlış olduğunda elif’in şartları doğru ise devamındaki kodlar çalışır. elif’in şartları yanlış ise else’nin devamındaki kodlar çalışır |
| 5 | **İç içe if ifadeleri :** İç içe if karar yapıları kullanılabilir. |

**Örnekler:**

# Sadece if şart ifadesi  
x=3  
if x<5:  
 print('Beşten küçüktür') # Ekranda 'Beşten küçüktür' yazar  
if x>=5:  
 print('Beşten büyük veya eşittir')# Ekranda birşey yazmaz. Çünkü şart sağlanmaz  
  
# if şartlar else yapısı  
x = 9  
if x == 8:  
 print('Sekize eşittir') # Ekranda birşey yazmaz  
else :  
 print('Sekizden farklıdır') # Ekranda 'Sekizden farklıdır' yazar.  
  
#if şartlar elif şartlar elif şartlar  
x = 10  
if x == 9:  
 print('Dokuz') # Ekranda birşey yazmaz  
elif x==10 :  
 print('On') # Ekranda 'On' yazar.  
elif x==11 :  
 print('On bir') # Ekranda birşey yazmaz  
  
#if şartlar elif şartlar else  
x = 12  
if x == 9:  
 print('Dokuz') # Ekranda birşey yazmaz  
elif x==10 :  
 print('On') # Ekranda birşey yazmaz  
else :  
 print('Farklı') # Ekranda 'Farklı' yazar  
  
# iç içe if karar yapıları  
x=3 ; y=5  
if x==3:  
 if y==5:  
 print('x:3','y:5')  
 else:  
 print('x:3', "y:5'den farklı")  
else:  
 if y == 5:  
 print("x:3'ten farklı", 'y:5')  
 else:  
 print("x:3'ten farklı", "y:5'den farklı")

**Döngü Yapıları**

Komutlar genelde yukarıdan aşağıya mantığıyla sırayla çalışır. Programdaki ilk ifade ilk olarak çalışır; sonraki ifadeler sırayla art arda çalışır. Bazen belirli kodların birkaç kez çalıştırılması gerekebilir. Bu tip durumlarda döngü yapıları kullanılır.

Pythonda while ve for döngü yapıları vardır. Bu döngülerdeki blok ifadeler belirli şartlara göre istenildiği sayı kadar tekrar tekrar çalıştırılır.

Şart Sağlandığında Çalışacak Kodlar

Şart doğru olduğundaŞart yanlış olduğunda

Şart yanlış olduğunda

Şart

Aşağıdaki tabloda Python döngü yapıları gösterilmektedir.

|  |  |
| --- | --- |
| **Döngü Seçenekleri** | **Döngü Tipleri ve Açıklamalar** |
| 1 | **While döngüsü:** While’dan sonra gelen şart True olduğunda bağlı olduğu kod bloğu tekrar tekrar çalıştırılır, ancak şart False olduğunda döngüden çıkar. |
| 2 | **For döngüsü:** Döngü değişkeninin alacağı değerler kadar döngü devam eder ve içerdiği kod blokları tekrar tekrar çalışır. |
| 3 | **İç içe döngüler:** While veya for döngüsü içinde başka bir döngü olabilir. |

**Döngü Kontrol İfadeleri**

Döngü kontrol ifadeleri, döngünün normal çalıştırılma sırasını değiştirirler.

Döngü kontrol ifadeleri **break**, **continue** ve **pass** ifadeleridir. Aşağıdaki tabloda açıklamaları verilmiştir.

|  |  |
| --- | --- |
| **S.No.** | **Control Statement & Description** |
| 1 | **break ifadesi :** break ifadesi çalıştırıldığında içinde bulunduğu döngüden çıkılır. |
| 2 | **continue ifadesi :** continue ifadesi çalıştırıldığında geri kalan döngü kodları çalıştırılmadan bir sonraki iterasyona devam eder şart sağlanırsa yeni iterasyonan başlar. Döngüden çıkmaz. |
| 3 | **pass ifadesi:** Pythonda pass ifadesi, bir ifadeye yazımsal olarak ihtiyaç duyulduğunda kullanılır. Fakat herhangi bir kod yazmak istemediğimiz durumlarda kullanırız. |

Tekrar edici (Iterator) ve Olusturucu (Generator)

Tekrar edici, programcıya bir koleksiyonun elemanlarının art arda elde edilmesi imkanını veren bir nesnedir. Bu nesne iki önemli metoda sahiptir: **iter**() ve **next**()

iter() metodu tekrar edici nesnesini oluşturur. next() metot ise bir sonraki elemanın aktif olmasını sağlar.

Bir tekrar edici nesne oluşturmak için metin, liste ve demet nesnesi olmalıdır.

**Örnek:**

import sys  
liste = [1,2,3,4,5]  
tekrar= iter(liste) # Tekrar edici nesne oluşturur  
print (next(tekrar)) # Tekrar edicideki sonraki elemanı yazdırır  
print (next(tekrar)) # Tekrar edicideki sonraki elemanı yazdırır  
for x in tekrar:  
 print (x, end=" ")  
print()  
  
# next() metodu kullanarak  
liste1 = [1,2,3,4,5]  
tekrar1=iter(liste1)  
while True:  
 try:  
 print (next(tekrar1))  
 except StopIteration:  
 sys.exit()

**Yukardaki kodun ekran çıktısı aşağıdaki gibidir:**

1

2

3 4 5

1

2

3

4

5

**Oluşturucu** (Generator) **yield** metodu kullanarak değerler dizisi oluşturan bir fonksiyondur. Bir oluşturucu fonksiyon çağrıldığında bu fonksiyon çalıştırılmadan bir oluşturucu nesne döndürülür. next() metodu ilk kez çağrıldığında yield ifadesine gelene kadar çalıştırılır. yield’den sonra gelen değişkenin değerleri next metoduyla alınır.

**Aşağıdaki örnekte fibonacci dizisinin elemanlarını yazdırma ile ilgili bir oluşturucu tanımlanmaktadır.**

**Örnek:**

import sys  
def fibonacci(n): # oluşturucu fonksiyonu  
 a, b, sayac = 0, 1, 0  
 while True:  
 if (sayac > n):  
 return  
 yield a  
 a, b = b, a + b  
 sayac += 1  
tekrar = fibonacci(8) # f tekrar edici nesne  
  
while True:  
 try:  
 print (next(tekrar), end=" ")  
 except StopIteration:  
 sys.exit()

**Ekran çıktısı:**

0 1 1 2 3 5 8 13 21

**Sayılar**

Sayısal veritipidir. Veritipi değiştirilemez veritiplerindendir.

Aşağıdaki gibi sayı tipinde değişken tanımlanır.

deger1 = 3

deger2 = 8

Değişkenleri del komutu ile bellekten silebiliriz. Bir yada birçok değişkeni virgülle ayırarak silebiliriz.

del degisken1[,degisken2[,var3[....,degiskenN]]]]

**Örnek:**

del degisken

del degisken1, degisken2

Python 3 farklı sayısal veri tipini destekler:

**int (işaretli tamsayılar):** Tamsayılar negatif veya pozitif sayı olabilir.

**float (Reel sayılar):** Bütün virgüllü sayılar, yada bilimsel gösterimli sayılar. Örneğin E yada e ile yazılan sayılar: 2.5e2=2.5x102 veya 2.5E2.

**complex (karmaşık sayılar):** a+bj şeklinde yazılan sayılar yani karmaşık sayılar. J yada j kullanılabilir.

Bir tamsayı, hexadecimal (16’lı) yada octal (8’li) sayı sistemi formunda temsil edilebilir.

>>> sayi = 0xA0F # 16’lı sayı sistemi

>>> sayi

2575

>>> sayi2 = 0o37 # 8’li sayı sistemi

>>> sayi2

31

**Matematiksel Fonksiyonlar**

**import math** # İfadesi eklenmelidir.

|  |  |
| --- | --- |
| No | **Fonksiyonlar ve Açıklamaları** |
| 1 | **math.abs(x)** : x sayısının mutlak değerini hesaplar. |
| 2 | **math.ceil(x)** : x’ten küçük olmayan en küçük tamsayıyı hesaplar. |
| 3 | **math.exp(x)** : ex hesaplanır. |
| 4 | **math.fabs(x)** : x sayısının mutlak değerini hesaplar. |
| 5 | **math.floor(x)** : x’ten büyük olmayan en büyük tamsayıyı hesaplar. |
| 6 | **math.log(x)** : x>0 için doğal logaritmasını bulur |
| 7 | **math.log10(x)** : x>0 için 10 tabanına göre logaritmayı hesaplar. |
| 8 | **math.max(x1, x2,...)** : Maksimum değeri bulur. |
| 9 | **math.min(x1, x2,...)** : Minimum değeri bulur. |
| 10 | **math.modf(x)** : |
| 11 | **math.pow(x, y)** : x\*\*y ile aynı hesaplamayı yapar. |
| 13 | **math.sqrt(x)** : x>0 için karekökü hesaplar. |

**Rastgele Sayı Üreten Fonksiyonlar**

Rastgele sayılar, oyunlar, simulasyon, test, güvenlik ve özel uygulamalar için kullanılmaktadır. Aşağıdaki tabloda rastgele sayı üreten fonksiyonlar gösterilmektedir.

import random # Bu satır eklenmelidir

|  |  |
| --- | --- |
| No | **Fonksiyonlar ve Açıklamaları** |
| 1 | **random.choice(dizi)** : Bir liste, demet yada metindeki rastgele bir elemanı seçer. |
| 2 | **random.randrange ([başlangıç,] bitiş [,adım])** : (başlangıç, bitiş, adım) daki elemnanlardan rastgele bir eleman seçilir. |
| 3 | **random.random()** : Rastgele bir reel r sayısını üretir. 0<= r <1 |
| 4 | **random.seed([x])** : Rastgele sayı üretmede kullanılan değeri başlatan tamsayıyı yerleştirir. Herhangi bir rastgele fonksiyon çağrılmadan önce bu fonksiyon çağrılmalıdır. Hiçbirşey döndürmez. |
| 5 | **random.shuffle(lst)** : Bir listenin elemanlarını rastgele olarak karıştırır. Hiçbirşey döndürmez. |
| 6 | **random.(uniform(x, y)** : Rastgele bir r reel sayısını üretir. x <= r < y |

**Trigonometrik Fonksiyonlar**

Aşağıda trigonometrik fonksiyonlar açıklanmıştır.

import math # Bu satır eklenmelidiri.

|  |  |
| --- | --- |
| No | **Fonksiyonlar ve Açıklamaları** |
| 1 | **math.acos(x)** : Radyan cinsinden x’in arc cos’ü hesaplar. |
| 2 | **math.asin(x)** : Radyan cinsinden x’in arc sin’ü hesaplar. |
| 3 | **math.atan(x)** : Radyan cinsinden x’in arc tan’ı hesaplar. |
| 4 | **math.atan2(y, x)** : Radyan cinsinden y/x’in arc tan’ı hesaplar. |
| 5 | **math.cos(x)** : Radyan cinsinden cos(x) hesaplanır. |
| 6 | **math.hypot(x, y)** : x ve y dik kenarlarına göre hipotenüsü hesaplar. |
| 7 | **math.sin(x)** : Radyan cinsinden sin(x) hesaplanır. |
| 8 | **math.tan(x)** : Radyan cinsinden tan(x) hesaplanır. |
| 9 | **math.degrees(x)** : Radyan x’i dereceye çevirir. |
| 10 | **math.radians(x)** : Derece x’i radyan’a çevirir. |

**Matematiksel Sabitler**

Pythonda iki matematiksel sabit **pi** ve **e** sayısı tanımlanmıştır

**Metin (String)**

Metin, Pythonda en önemli veri tiplerindendir. Metin veri tipleri tırnak içerisinde alınan değişken tanımlamasıyla oluşturulur. Tek tırnak yada çift tırnak kullanılabilir.

Aşağıda metin tipinde değişkenler tanımlanmıştır.

degisken1 = 'Merhaba Python'

degisken2 = "Python Programlama"

**Metin İçindeki Verilere Erişim**

Python karakter tipini desteklemez. Karakterler, uzunluğu bir olan metin gibi kabul edilir. Bir alt metn gibi düşünülür.

Bu alt metinlere erişim için köşeli parantez ve indisler kullanılır.

Aşağıda bununla ilgili örnekler bulabilirsiniz.

deg1 = 'Merhaba Python'

deg2 = "Python Programlama"

print ("deg1[0]: ", deg1[0])

print ("deg2 [1:5]: ", deg2[1:5])

Yukardaki kodlar çalıştığında aşağıdaki gibi ekran çıktısı olur.

deg1[0]: M

deg2 [1:5]: ytho

**Metin Güncelleştirme**

Metin değişkenini tekrar başka bir metin değeri atayarak metin değişkeni güncellenebilir. Bu değişkenin önceki metin değeri yok olur.

deg1 = 'Merhaba Dünya'

print ("Güncel Metin:- ", deg1)

Yukarıdaki kod çalıştığında aşağıdaki ekran çıktısı çıkar.

Güncel Metin:- Merhaba Dünya

**Metinlere Özel Karakterler Ekleme**

Aşağıdaki tablo, ters bölü işareti kullanılarak özel karakterlerin kullanımını göstermektedir. Bu karakterler ekranda görünmez.

print ("\\") # Ters bölü yazdırır  
print ("\'") # Tek tırnak yazdırır  
print ("\"")# Çift tırnak yazdırır  
print ("hello\nworld") # \n den sonra alt satıra geçer  
print (u"\N{DAGGER}") # unicode olarak Dagger simgesi yazdırır  
print ("123456\rXX\_XX") # \r karakteri return tuşu: önceki değerler yok olur  
print ("\t\* hello") # tab tuşuna basılmış gibi olur  
print (u"\u041b") # Unicode olarak belirtilen sayı ile temsil edilen karakter yazdırılır  
print (u"\U000001a9")# Unicode olarak belirtilen sayı ile temsil edilen karakter yazdırılır  
print ("\043") # \0 8’li sayı sistemi ile karakter yazdırılır  
print ("\x23") # \x 16’lı sayı sistemi ile karakter yazdırılır

Yukardaki kodlar çalıştırıldığında aşağıdaki gibi ekran çıktısı olur.

\

'

"

hello

world

†

XX\_XX

\* hello

Л

Ʃ

#

#

**Özel Metin Operatörleri**

a=’Merhaba’

b=’Python’

|  |  |  |
| --- | --- | --- |
| **Operatör** | **Açıklama** | **Örnek** |
| + | İki metni art arda ekler. | a + b Sonucu: MerhabaPython |
| \* | Yıldızdan sonra gelecek sayı kadar metni art arda tekrar ederek ekleme işlemi yapar. | a\*2 Sonucu: MerhabaMerhaba |
| [] | Köşeli parantez içinde verilen indis ile metnin karakteri elde edilir. | a[1] sonucu: e |
| [ : ] | Köşeli parantez içinde kullanılan üst üste iki nokta ile belirli aralıktaki karakterler seçilir | a[1:4] Sonucu: erh a[:3] Sonucu: Mer a[4:] Sonucu: aba |
| in | Bir karakter metin içinde ise True döner. | print('M' in a) Sonucu: True |
| not in | Bir karakter metin içinde değilse False döner. | print('M' in a) Sonucu: False |
| r/R | R yada r karakterinden hemen sonraki metin aynen nasıl görünüyorsa o şekilde yazdırılır. | print(r"\n") Sonucu: \n |

**Metin Formatlama Operatörü**

Metin formatlama operatörü % dir. C programlama dilindeki printf() fonksiyonundaki kullanımı gibidir.

print("Aydın, %s bölgesindedir ve kodu %d dur." % ('Ege', 9))

Yukardaki kod çalıştığında aşağıdaki gibi çıktı olur.

Aydın, Ege bölgesindedir ve kodu 9 dur.

|  |  |
| --- | --- |
| **Seçenek No** | **Format Sembolü ve Dönüşümü** |
| 1 | **%c :** Karakter |
| 2 | **%s :** Metin |
| 3 | **%i :** İşaretli tamsayı |
| 4 | **%d :** İşaretli tamsayı |
| 5 | **%u :** İşaretsiz tamsayı |
| 6 | **%o :** Sekizli tamsayı |
| 7 | **%x :** 16’lı sayı sistemi (Küçük harflerle) |
| 8 | **%X :** 16’lı sayı sistemi (Büyük harflerle) |
| 9 | **%e :** Bilimsel yazım şekliyle (Küçük harfle ‘e’) |
| 10 | **%E :** Bilimsel yazım şekliyle (Büyük harfle ‘E’) |
| 11 | **%f :** Reel sayılar için |
| 12 | **%g :** %f and %e’in daha kısası |
| 13 | **%G :** %f and %E’in daha kısası |

|  |  |
| --- | --- |
| **Seçim No** | **Sembol ve Özelliği** |
| 1 | **\* :** Argüman, genişlik ve tamlığı belirler. |
| 2 | **- :** Sayıların sola yanaşık olmasını sağlar. |
| 3 | **+:** + simgesinin görünmesini sağlar. |
| 4 | **<sp>:** Pozitif sayıdan önce bir boşluk bırakır. |
| 5 | **#:** Sekizli veye on altılı tabanda sıfır (‘0’) ve ‘0x’ yada ‘0X’ ekler. |
| 6 | **0:** Boşluk yerine sıfırları soldan ekler. |
| 7 | **%:** ‘%%’ bir tane ‘%’ ekrana yazdırır. |
| 8 | **(değişken):** Parantez içinde değişkenler kullanılır. |
| 9 | **m.n. :** m, en az olabilecek genişlik ve n ise ondalık kısmının rakam sayısıdır. |

**Yanyana Üç Tırnak**

Python ile çoklu satırları ekrana yazdırmak için yan yana üç tırnak kullanabiliriz.Bu şekilde metindeki tab, yeni satır karakterleri veya başka herhangi bir karakter korunarak ekrana yazılabilir. Tırnak, tek veya çift tırnak olabilir

paragraf\_metin = “””Bu, birkaç satırdan oluşan bir \n yazı metnidir ve TAB (\t) karakteri  
gibi yazılamayan karakterler kullanılabilir ve ekranda   
tab uygulanır aynı zamanda yeni satır karakteride metin "  
içinde kullanılabillir.”””  
print (paragraf\_metin)

Yukardaki kod çalıştığında aşağıdaki gibi bir çıktı verir. \n karakter yeni bir satıra geçmeyi sağlar; \t karakteri ise tab kadar boşluk bırakır. Nasıl yazılmışsa o şekilde ekranda yazar.

Bu, birkaç satırdan oluşan bir

yazı metnidir ve TAB ( ) karakteri

gibi yazılamayan karakterler kullanılabilir ve ekranda

tab uygulanır aynı zamanda yeni satır karakteride metin

içinde kullanılabillir.

print("C:\\Klasör")  
print(r"C:\\Klasör")

Yukardaki kod çalıştığında aşağıdaki çıktıyı verir. Tırnak sonrar gelen r karakteri tırnak içindeki metnin aynen ekranda çıkmasını sağlar.

C:\Klasör

C:\\Klasör

**Unicode Metni**

Python 3’te bütün metinler Unicode olarak temsil edilirler. Fakat Python 2’de ise sekiz bitlik ASCII kod olarak saklanır. Bu yüzden metni Unicode yapmak için u karakteri eklemek gerekir. Fakat Python 3’te şimdi buna gerek yoktur.

Pythonda metinleri değiştirmek için hazır metotlar vardır. Bu metotlar aşağıda litelenmiştir:

|  |  |
| --- | --- |
| **Seçim No** | **Metotlar ve Açıklamaları** |
| 1 | [**capitalize()**](https://www.tutorialspoint.com/python3/string_capitalize.htm)**:** Metnin ilk harfini büyük harf yapar.  **Örnek:**  a="merhaba" print(a.capitalize()) |
| 2 | [**center(genişlik, karakter)**](https://www.tutorialspoint.com/python3/string_center.htm)**:** Metin, genişlik olarak verilen sayı genişliğinde metni ortalar; solunda ve sağında karakter olarak verilen karakter ile doldurur.  **Örnek:**  metin="merhaba" print(metin.center(80,'-')) |
| 3 | [**count(metin, beg = 0,end = len(string))**](https://www.tutorialspoint.com/python3/string_count.htm)**:** Asıl metin içinde metin kaç defa geçer, onu hesaplar; beg: başlangıç indis; end: aranacak son indis değerdir.  **Örnek:**  metin="Bu gün hava çok sıcak. Yarın ise çok soğuk olacak" print(metin.count("çok")) # çok sözcüğü asıl metinde 2 defa geçer  print(metin.count("çok",0,30)) # çok sözcüğü asıl metnin 0 ve 30 indisleri arasında kaç kere geçtiğini bulur |
| 4 | [**decode(encoding = 'UTF-8',errors = 'strict')**](https://www.tutorialspoint.com/python3/string_decode.htm)**:** Metnin belitilen koda göre kodlanmış metni döndürür.  **Örnek:**  import base64 encoded = base64.b64encode(b'Encode metnidir.') print (encoded) data = base64.b64decode(encoded) print (data)  **Çıktı:**  b'RW5jb2RlIG1ldG5pZGlyLg=='  b'Encode metnidir.' |
| 5 | [**encode(encoding = 'UTF-8',errors = 'strict')**](https://www.tutorialspoint.com/python3/string_encode.htm)**:** Metnin kodlanmış örneğini geri döndürür.  **Örnek:**  import base64 metin1 = "Metin örneği..." metin1 = base64.b64encode(metin1.encode('utf-8',errors = 'strict')) print ("Encode örneği:" , metin1)  **Çıktı:**  Encode örneği: b'TWV0aW4gw7ZybmXEn2kuLi4=' |
| 6 | [**endswith(sonek, beg = 0, end = len(metin))**](https://www.tutorialspoint.com/python3/string_endswith.htm)**:** Eğer asıl metnin son eki belirtilen sonek ile aynı ise True değilse False döndürür.  **Örnek:**  metin='Python metin uygulamasıdır.' sonek1='dır.' sonek2='dır' print (metin.endswith(sonek1)) print (metin.endswith(sonek2)) sonek3='uygu' print(metin.endswith(sonek3,0,17))  **Çıktı:**  True  False  True |
| 7 | [**expandtabs(tabsize = 8)**](https://www.tutorialspoint.com/python3/string_expandtabs.htm)**:** Metni içindeki tab boyutu kadar boşluk ekleyerek metin döndürür.  **Örnek:**  metin= "Bu bir\tPython örneğidir." print ("Asıl metin:" + metin) print ("Default tab uygulanmış metin:" + metin.expandtabs()) print ("2 kat tab uygulanmış metin:: " + metin.expandtabs(16))  **Çıktı:**  Asıl metin:Bu bir Python örneğidir.  Default tab uygulanmış metin:Bu bir Python örneğidir.  2 kat tab uygulanmış metin:: Bu bir Python örneğidir. |
| 8 | [**find(metin, beg = 0 end = len(asılmetin))**](https://www.tutorialspoint.com/python3/string_find.htm)**:** Eğer metin, asıl metin içinde varsa bulunan yerin indisini, bulunmazsa -1 döndürür.  **Örnek:**  asilmetin = "Bu bir Python uygulamasıdır." metin = "Python"; print (asilmetin.find(metin)) print (asilmetin.find(metin, 5)) print (asilmetin.find(metin, 10))  **Çıktı:**  7  7  -1 |
| 9 | [**index(str, beg = 0, end = len(string))**](https://www.tutorialspoint.com/python3/string_index.htm)**:** find metodu ile aynıdır. Fakat eğer metin, asıl metinde yoksa Exceptio (istisna) hatasını döndürür. |
| 10 | [**isalnum()**](https://www.tutorialspoint.com/python3/string_isalnum.htm)**:** Eğer bütün karaktrler alfanumerik ve en az bir karakter var ise True döndürü, aksi taktirde False döndürür.  **Örnek:**  metin1 = "istanbul34" # print (metin1.isalnum()) metin2 = "Bu bir Python Uygulamasıdır." print (metin2.isalnum())  **Çıktı:**  True  False |
| 11 | [**isalpha()**](https://www.tutorialspoint.com/python3/string_isalpha.htm)**:** Eğer en az bir karakter varsa ve bütün karakterler alfanümerik ise True, değilse False döndürür.  **Örnek:**  metin1 = "istanbul" print (metin1.isalpha()) metin2 = "Bu bir Python Uygulamasıdır." print (metin2.isalpha())  **Çıktı:**  True  False |
| 12 | [**isdigit()**](https://www.tutorialspoint.com/python3/string_isdigit.htm)**:** Eğer metin sadece rakamlar içeriyorsa True, değilse False döndürür.  **Örnek:**  metin1 = "9999" print (metin1.isdigit()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.isdigit())  **Çıktı:**  True  False |
| 13 | [**islower()**](https://www.tutorialspoint.com/python3/string_islower.htm)**:** Eğer metin en az bir harf varsa ve bütün harfler de küçük harf ise True, değilse False döndürür.  **Örnek:**  metin1 = "a9999;-" print (metin1.islower()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.islower())  **Çıktı:**  True  False |
| 14 | [**isnumeric()**](https://www.tutorialspoint.com/python3/string_isnumeric.htm)**:** Eğer bir unicode metin sadece sayı içeriyorsa True, değilse False döndürür.  **Örnek:**  metin1 = "123" print (metin1.isnumeric()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.isnumeric())  **Çıktı:**  True  False |
| 15 | [**isspace()**](https://www.tutorialspoint.com/python3/string_isspace.htm)**:** Eğer metin sadece boş karakterler içeriyorsa True, değilse False döndürür.  **Örnek:**  metin1 = " \n\t" print (metin1.isspace()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.isspace())  **Çıktı:**  True  False |
| 16 | [**istitle()**](https://www.tutorialspoint.com/python3/string_istitle.htm)**:** Eğer metindeki her kelimenin ilk harfleri büyükharf ise True, değilse False döndürür.  **Örnek:**  metin1 = "Bu Bir Başlıktır" print (metin1.istitle()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.istitle())  **Çıktı:**  True  False |
| 17 | [**isupper()**](https://www.tutorialspoint.com/python3/string_isupper.htm)**:**Eğer metin en az bir harf varsa ve bütün harfler de büyük harf ise True, değilse False döndürür.  **Örnek:**  metin1 = "ANKARA06;-" print (metin1.isupper()) metin2 = "Bu bir Python Uygulamasıdır. 1234" print (metin2.isupper())  **Çıktı:**  True  False |
| 18 | [**join(dizi)**](https://www.tutorialspoint.com/python3/string_join.htm)**:** Metin dizisinin elemanlarını ayırıcı dizi ile bir birine bağlayarak bir metin oluşturur.  **Örnek:**  s = "-" dizi = ("İstanbul", "Ankara", "İzmir") # Metin dizisi print (s.join( dizi ))  **Çıktı:**  İstanbul-Ankara-İzmir |
| 19 | [**len(string)**](https://www.tutorialspoint.com/python3/string_len.htm)**:** Metin içindeki karakter sayısını bulur.  **Örnek:**  Metin="Bu bir Python uygulamasıdır." print(len(Metin))  **Çıktı:**  28 |
| 20 | [**ljust(genişlik[, fillchar])**](https://www.tutorialspoint.com/python3/string_ljust.htm)**:** Sağ tarafı boşluk ile yada fillchar’da belirtilen karakterle doldurarak sola yanaşık metin döndürür.  **Örnek:**  str = "Bu bir Python" print (str.ljust(30)) print (str.ljust(30, '\*'))  **Çıktı:**  Bu bir Python  Bu bir Python\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| 21 | [**lower()**](https://www.tutorialspoint.com/python3/string_lower.htm)**:** Büyük harfleri küçük harflere dönüştürür.  **Örnek:**  str = "Bu bir Python Ugulamasıdır" print (str.lower())  **Çıktı:**  bu bir python Ugulamasıdır |
| 22 | [**lstrip()**](https://www.tutorialspoint.com/python3/string_lstrip.htm) **:** Metnin başında bulunan boş karakterleri yok eder.  **Örnek:**  metin = " Bu bir Python uygulamasıdır. " print (metin.lstrip()) metin = "------Bu bir Python uygulamasıdır.-----" print (metin.lstrip('-'))  **Çıktı:**  Bu bir Python uygulamasıdır.  Bu bir Python uygulamasıdır.----- |
| 23 | [**maketrans(input,output)**](https://www.tutorialspoint.com/python3/string_maketrans.htm)**:** input metninin karakterlerinden output metninin karakterlerine birebir fonksiyon anlamındadır. input’taki karakterler output’taki karakterlere dönüştürür.  **Örnek:**  input = "aeiouı" output = "123456" donustur = str.maketrans(input, output) metin = "Bu bir Python uygulamasıdır." print (metin.translate(donustur))  **Çıktı:**  B5 b3r Pyth4n 5yg5l1m1s6d6r. |
| 24 | [**max(metin)**](https://www.tutorialspoint.com/python3/string_max.htm)**:** Metnin en büyük karakterini döndürür.  **Örnek:**  metin = "Bu bir Python uygulamasıdır." print ("Maksimum karakter: " + max(metin))  metin = "Dönüşüm işlemi çok önemlidir." print ("Maksimum karakter: " + max(metin))  **Çıktı:**  Maksimum karakter: ı  Maksimum karakter: ş |
| 25 | [**min(metin)**](https://www.tutorialspoint.com/python3/string_min.htm)**:** Metnin en küçük karakterini döndürür.  **Örnek:**  metin = "Python" print ("Minimum karakter: " + min(metin)) metin = "Dönüşüm" print ("Minimum karakter: " + min(metin))  **Çıktı:**  Minimum karakter: P  Minimum karakter: D |
| 26 | [**replace(eski, yeni [, max])**](https://www.tutorialspoint.com/python3/string_replace.htm)**:** eski metni yeni metin yerine yerleştirir. Eğer max kullanılırsa max sayısı kadar yerine koyar.  **Örnek:**  metin = "bir iki üç dört beş altı bir iki bir bir" print (metin.replace("bir", "1")) print (metin.replace("bir", "1", 2))  **Çıktı:**  1 iki üç dört beş altı 1 iki 1 1  1 iki üç dört beş altı 1 iki bir bir |
| 27 | [**rfind(str, beg = 0,end = len(string))**](https://www.tutorialspoint.com/python3/string_rfind.htm)**:** find metodu ile aynıdır. Fakat arama geriye doğru olur. |
| 28 | [**rindex( str, beg = 0, end = len(string))**](https://www.tutorialspoint.com/python3/string_rindex.htm)**:** index metodu ile aynıdır. Fakat arama geriye doğrudur. |
| 29 | [**rjust(genişlik,[, fillchar])**](https://www.tutorialspoint.com/python3/string_rjust.htm)**:** Metni sağa yanaşık olacak şekilde ve sol tarafı boşluk ile yada fillchar ile belirtilen karakterle doldurur.  **Örnek:**  metin = "Bu bir Python" print (metin.rjust(30)) print (metin.rjust(30, '\*'))  **Çıktı:**  Bu bir Python  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Bu bir Python |
| 30 | [**rstrip()**](https://www.tutorialspoint.com/python3/string_rstrip.htm)**:** Metnin sonunda bulunan boş karakterleri yok eder.  **Örnek:**  metin = " Bu bir Python uygulamasıdır. " print (metin.rstrip()) metin = "------Bu bir Python uygulamasıdır.-----" print (metin.rstrip('-'))  **Çıktı:**  Bu bir Python uygulamasıdır.  ------Bu bir Python uygulamasıdır. |
| 31 | [**split(str="", num=string.count(str))**](https://www.tutorialspoint.com/python3/string_split.htm)**:** Metni belirtilen karaktere göre parçalara böler ve liste oluşturur. Str belirtilmezse boş karakter kabul edilir. num, kaç kere bölüneceğini belirtir.  **Örnek:**  metin = "Bu bir Python uygulamasıdır." print (metin.split( )) print (metin.split('u',1)) print (metin.split('u'))  **Çıktı:**  ['Bu', 'bir', 'Python', 'uygulamasıdır.']  ['B', ' bir Python uygulamasıdır.']  ['B', ' bir Python ', 'yg', 'lamasıdır.'] |
| 32 | [**splitlines( num=string.count('\n'))**](https://www.tutorialspoint.com/python3/string_splitlines.htm)**:** Metni \n yeni satır karakterlerine göre parçalara bölerek liste oluşturur.  **Örnek:**  metin = "Bu \nbir \nPython \nuygulamasıdır." print (metin.splitlines())  **Çıktı:**  ['Bu ', 'bir ', 'Python ', 'uygulamasıdır.'] |
| 33 | [**startswith(metin, beg=0,end=len(string))**](https://www.tutorialspoint.com/python3/string_startswith.htm)**:** Asıl metin, metin ile başlıyorsa True, değilse False döndürür. Yada beg ile başlayan indisten end ile biten indise kadarki metin ile eşleşiyorsa True döndürür.  **Örnek:**  metin = "Bu bir Python uygulamasıdır."; print (metin.startswith( 'bu' )) print (metin.startswith( 'bir', 3, 6) ) print (metin.startswith( 'bir', 2, 5 ))  **Çıktı:**  False  True  False |
| 34 | [**strip([chars])**](https://www.tutorialspoint.com/python3/string_strip.htm)**:** Metin içinde lstrip() ve rstrip() metotlarının ikisini de uygular. |
| 35 | [**swapcase()**](https://www.tutorialspoint.com/python3/string_swapcase.htm)**:** Metnin içindeki büyük harfleri küçük yapar. Küçük harfleri de büyük yapar. |
| 36 | [**title()**](https://www.tutorialspoint.com/python3/string_title.htm)**:** Metnin bütün sözcükleri büyük harfle başlayacak şekilde döndürür.  **Örnek:**  metin = "Bu biR Python uygulamasıdır."; print(metin.title())  **Çıktı:**  Bu Bir Python Uygulamasıdır. |
| 37 | [**translate(input,output, deletechars="")**](https://www.tutorialspoint.com/python3/string_translate.htm)**:** maketrans ile oluşturulan dönüşüm translate ile metne uygulanır. Karakter karakter dönüştürülür ve sil ile belirtilen karakterler silinir.  **Örnek:**  input = "aeiou" output = "12345" sil = "Pu"# Silinecek karakterler donusum = str.maketrans(input, output, sil) metin = "Bu bir Python uygulamasıdır." print (metin.translate(donusum))  **Çıktı:**  B b3r yth4n ygl1m1sıdır. |
| 38 | [**upper()**](https://www.tutorialspoint.com/python3/string_upper.htm)**:** Küçük harfleri büyük harflere dönüştürür.  **Örnek:**  metin = "Bu bir Python uygulamasıdır." print ("Büyük Harfli: ",metin.upper())  **Çıktı:**  Büyük Harfli: BU BIR PYTHON UYGULAMASIDIR. |
| 39 | [**zfill (genişlik)**](https://www.tutorialspoint.com/python3/string_zfill.htm)**:** Asıl metni genişlik kadar soldan sıfır doldurarak yeni metin döndürürler.  **Örnek:**  metin = "Bu bir Python uygulamasıdır." print ("metin.zfill : ",metin.zfill(40)) print ("metin.zfill : ",metin.zfill(50))  **Çıktı:**  metin.zfill : 000000000000Bu bir Python uygulamasıdır.  metin.zfill : 0000000000000000000000Bu bir Python uygulamasıdır. |
| 40 | [**isdecimal()**](https://www.tutorialspoint.com/python3/string_isdecimal.htm)**:** Unicode metninin sayı olup olmadığına karar verir. Sayı ise True değilse False döndürür.  **Örnek:**  metin = "2018"; print(metin.isdecimal()) metin = "abc2018"; print(metin.isdecimal())  **Çktı:**  True  False |

**Listeler**

Pythondaki en temel yapı listelerdir. Dizinin her elemanına bir sayı (dizinin indisi) atanır. İndis sıfırdan başlamaktadır. 0, 1, 2, .. şeklinde devam etmektedir.

Python 6 türde diziye sahiptir. Fakat en yaygın olanı listeler ve ikililerdir(tuples). Bütün dizilerde şu işlemleri yapabiliriz: indeksleme(indexing), parçalara ayırma (slicing), ekleme(adding), çarpma (multiplying) ve üyelik için kontrol (checking).

Buna ek olarak, Python, dizinin uzunluğunu, en büyük ve en küçük elemanı bulmak için hazır fonksiyonları vardır.

Listeler, Pythonda en kullanışlı veri tiplerindendir. Veriler köşeli parantezler içinde virgüllerle ayrılır. Listelerde önemli olan nokta şudur: Listelerdeki elemanlar aynı veri tipinde olmak zorunda değildir.

Aşağıda Python liste örnekleri gösterilmektedir.

Liste1 = [‘Fizik’, ‘Matematik’, 2012, 2018];

Liste2 = [1, 2, 3, 4, 5 ];

Liste3 = ["a", "b", "c", "d"]

Liste indisleri de 0’dan başlar. Listeler, parçalara ayrılabilir vey birleştirilebilir.

**Listelerdeki Verilere Erişim**

Listelerdeki verilere erişim için, indislerdeki verileri almak için veya parçalara ayırmak için köşekli parantez kullanırız. Aşağıda bununla ilgili örnekler görülmektedir.

liste1 = ['Matematik','Fizik', 2012, 2018]  
liste2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
print ("liste1[0]: ", liste1[0])  
print ("liste2[1:5]: ", liste2[1:5])

Yukardaki kodlar çalıştırğında aşağıdaki gibi bir çıktı verecektir.

liste1[0]: Matematik

liste2[1:5]: [2, 3, 4, 5]

Listelerdeki bir veya birçok eleman atama operatörleri kullanarak güncellenebilir. Listelere append() metodu kullanarak veri eklenebilir. Append() metodu aşağıdaki bölümlerde açıklanacaktır.

liste = ['Matematik', 'Fizik', 2012, 2018]  
print ("İndis 2 deki değer: ", liste[2])  
liste[2] = 2015  
print ("İndis 2 deki yeni değer: ", liste[2])

Yukardaki kodlar çalıştırıldığında aşağıdaki çıktır oluşur.

İndis 2 deki değer: 2012

İndis 2 deki yeni değer: 2015

**Listelerden Eleman Silme**

Listelerden eleman silmek için del metodu indis belirterek kullanılabilir. remove() metodu ile listedeki nesneyi veri kullanarak silebiliriz.

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting. You can use the remove() method if you do not know exactly which items to delete. For example –

liste = ['Matematik', 'Fizik', 2012, 2018]  
print (liste)  
del liste[2]  
print ("2. verisini sildiktens sonra: ", liste)

Yukardaki kod çalıştığında aşağıdaki çıktı oluşur.

['Matematik', 'Fizik', 2012, 2018]

verisini sildiktens sonra: ['Matematik', 'Fizik', 2018]

**Listelerdeki Temel İşlemler**

Listelerde + ve \* operatörleri metinlerde ki gibi olmaktadır. + operatörü ekleme, \* operatörü ise listenin tekrarını sağlayarak yeni liste oluşturur.

|  |  |  |
| --- | --- | --- |
| **Python İfadeleri** | **Sonuçlar** | **Açıklama** |
| len([1, 2, 3, 4]) | 4 | Liste uzunluğu |
| [1, 2, 3, 4] + [5, 6, 7] | [1, 2, 3, 4, 5, 6, 7] | Listelerin eklenmesi |
| ['Merhaba'] \* 3 | ['Merhaba', 'Merhaba', 'Merhaba'] | Liste tekrarı |
| 5 in [1, 2, 3, 5] | True | Liste üyeliği |
| for x in [1, 2, 3, 4] :  print (x, end = ' ') | 1 2 3 4 | Liste elemanı yazdırma |

**İndisleme, Parçalar Ayırma ve Matris İşlemi**

Listeler dizi olduğundan, metinlerdeki gibi indisleme, parçalara ayırma ve matris işlemleri yapılabilabilir.

liste = ['Matematik', 'Fizik', 'Kimya']

|  |  |  |
| --- | --- | --- |
| **Python İfadesi** | **Sonuçlar** | **Açıklama** |
| liste[2] | 'Kimya' | İndis 0 dan başlar |
| liste[-2] | 'Fizik' | Eksi olduğunda sağdan itibaren listeler |
| liste[1:] | ['Fizik', 'Kimya'] | 1 den itibaren bütün liste |

**Listelerde Kullanılan Hazır Metotlar**

|  |  |
| --- | --- |
| **Seçim No** | **Metotlar ve Açıklamalar** |
| 1 | **len(liste):** Listenin uzunluğunu verir.  **Örnek:**  liste1 = ['Matematik', 'Fizik', 'Kimya'] print(len(liste1))  **Çıktı:**  3 |
| 2 | **max(liste):** Listedeki maksimum değeri döndürür.  **Örnek:**  liste1 = [5, 3, 2, 1] print(max(liste1))  **Çıktı:**  5 |
| 3 | **min(liste):** Listedeki minimum değeri döndürür.  **Örnek:**  liste1 = [5, 3, 2, 1] print(min(liste1))  **Çıktı:**  1 |
| 4 | **list(ikili):** Dizileri listeye dönüştürür.  **Örnek:**  tuple = ('İstanbul', 'Ankara', 'İzmir', 2018) liste1 = list(tuple) print ("Liste elemanları: ", liste1) metin="Merhaba" liste2=list(metin) print ("Liste elemanları: ", liste2)  **Çıktı:**  Liste elemanları: ['İstanbul', 'Ankara', 'İzmir', 2018]  Liste elemanları: ['M', 'e', 'r', 'h', 'a', 'b', 'a'] |
| 5 | Nesnelerin karşılaştırılması için ==, >, <, >= ve <= kullanılabilir.  **Örnek:**  liste1 = ['İstanbul', 'Ankara'] liste2 = ['İstanbul', 'Ankara'] liste3=[5,1,3] liste4=[5,4,2] print(liste1 == liste2) print(liste4 < liste3) print(liste4 > liste3)  **Çıktı:**  True  False  True |

Python, aşağıdaki metotlarıda kullanmaktadır.

|  |  |
| --- | --- |
| **Seçim No** | **Metotlar ve Açıklamalar** |
| 1 | **list.append(nesne):** Listeye nesne eklemeyi sağlar.  **Örnek:**  liste = ['İstanbul', 'Ankara'] liste.append(2018) print(liste)  **Çıktı:**  ['İstanbul', 'Ankara', 2018] |
| 2 | **list.count(nesne):** Listede nesneden kaç tane olduğunu bulur.  **Örnek:**  liste1 = [2018, 'Ankara', 2018,1,2018] print(liste1.count(2018))  **Çıktı:**  3 |
| 3 | **list.extend(seq):** Listeye dizinin elemanlarını ekler.  **Örnek:**  liste1 = ['Matematik', 'Fizik', 'Biyoloji'] liste2 = list(range(3)) # 0-2 arasında dizi oluşturur liste1.extend(liste2) print ('Yeni Liste :', liste1)  **Çıktı:**  Yeni Liste : ['Matematik', 'Fizik', 'Biyoloji', 0, 1, 2] |
| 4 | **list.index(nesne):** Listedeki nesnenin bulunduğu ilk indisi döndürür. İlk ve son arasında aranacak indislerde belirtilebilir.  **Örnek:**  liste1 = [2018, 'Ankara','Aydın',2000,'Aydın'] print(liste1.index('Aydın')) print(liste1.index('Aydın',3,5))  **Çıktı:**  2  4 |
| 5 | **list.insert(index, nesne):** Listeye belirtilen indiste nesne ekler.  **Örnek:**  liste1 = ['Kimya', 'Fizik'] liste1.insert(1, 'Matematik') print ('Son Liste: ', liste1)  **Çıktı:**  Son Liste: ['Kimya', 'Matematik', 'Fizik'] |
| 6 | **list.pop(nesne = list[-1**]): Eğer hiç indis girilmezse listenin son nesnesini siler. İndis girilirse indisteki nesne silinir.  **Örnek:**  liste1 = ['Matematik', 'Kimya', 'Fizik', 'Biyoloji'] liste1.pop() print ("Şimdiki liste: ", liste1) liste1.pop(1) print ("Şimdiki liste: ", liste1)  **Çıktı:**  Şimdiki liste: ['Matematik', 'Kimya', 'Fizik']  Şimdiki liste: ['Matematik', 'Fizik'] |
| 7 | **list.remove(nesne):** Listeden nesneyi siler.  **Örnek:**  liste1 = ['Matematik', 'Fizik', 'Kimya','Biyoloji'] liste1.remove('Kimya') print("Yeni liste: ", liste1) liste1.remove('Biyoloji') print("Yeni liste: ", liste1)  **Çıktı:**  Yeni liste: ['Matematik', 'Fizik', 'Biyoloji']  Yeni liste: ['Matematik', 'Fizik'] |
| 8 | **list.reverse():** Listeyi tersten sıralar.  **Örnek:**  liste1 = ['Matematik', 'Fizik', 'Kimya', 'Biyoloji'] liste1.reverse() print ("Yeni liste:", liste1)  **Çıktı:**  Yeni liste: ['Biyoloji', 'Kimya', 'Fizik', 'Matematik'] |
| 9 | **list.sort([func]):** Listeyi karşılaştırma fonksiyonuna göre sıralar. Herhangi bir fonksiyon kullanılmazsa küçükten büyüğe sıralar.  **Örnek:**  liste1 = ['Matematik', 'Fizik', 'Kimya', 'Biyoloji'] liste1.sort() print ("Yeni liste: ", liste1)  **Çıktı:**  Yeni liste : ['Biyoloji', 'Fizik', 'Kimy  a', 'Matematik'] |
| 10 | **sorted(nesne,key,reverse=True):** Nesneyi listeler, key’de belirtilen itemgetter ve attrgetter kullanılarak da sıralayabilir. Normalde artan sıralama yapar fakat reverse=True dendiğinde azalan sıralama yapar.  **Örnekler:**  from operator import itemgetter, attrgetter print(sorted([5, 2, 3, 1, 4])) print(sorted("Bu bir sıralama örneğidir".split(), key=str.lower)) ogrenci\_tuples = [('Ayşe', 'A', 15), ('Ahmet', 'B', 12), ('Veli', 'B', 10), ] print(sorted(ogrenci\_tuples, key=itemgetter(1), reverse=True)) print(sorted(ogrenci\_tuples, key=itemgetter(2))) print( sorted(ogrenci\_tuples, key=itemgetter(1,2)))# önce iseme, sonra dereceye göre sıralar  **Çıktı:**  [1, 2, 3, 4, 5]  ['bir', 'Bu', 'sıralama', 'örneğidir']  [('Ahmet', 'B', 12), ('Veli', 'B', 10), ('Ayşe', 'A', 15)]  [('Veli', 'B', 10), ('Ahmet', 'B', 12), ('Ayşe', 'A', 15)]  [('Ayşe', 'A', 15), ('Veli', 'B', 10), ('Ahmet', 'B', 12)] |

**İkili (Tuples)**

İkili, değiştirilemeyen Python nesneler dizisidir. İkililer, listeler gibi dizilerdir. En önemli farkları, ikililerin elemanları değiştirilemez, eleman silinemez veya eklenemez. İkililer, parantezlerle oluşturulur. Elemanları da virgüllerle ayrılır. Aşağıda örnekler verilmiştir.

tuple1 = ('Matematik', 'Kimya', 2012, 2018)

tuple2 = (1, 2, 3, 4, 5 )

tuple3 = "a", "b", "c", "d"

Boş ikili aşağıda gösterildiği gibi aç ve kapa parantez ile oluşturulur.

tuple1 = ();

Bir elemanlı ikil oluşturmak için mutlaka bir virgül eklemek gerekiyor. Aşağıda bir örneği görülmektedir.

tup1 = (50,)

Metin indislerindeki gibi, ikili indisler de 0’dan başlar. İkililer, parçalara bölünebilir, eklenebilir.

İkililerdeki Değerlere Erişim için köşeli parantez ve içinde indisler kullanılır.

tuple1 = ('Matematik', 'Kimya', 2012, 2018)

tuple2= (1, 2, 3, 4, 5, 6, 7, 8, 9 )

print ("tuple1[0]: ", tuple1[0])

print ("tuple2[1:5]: ", tuple2[1:5])

Yukardaki kod çalıştığında aşağıdaki çıktıyı verir.

tuple1[0]: Matematik

tuple2[1:5]: (2, 3, 4, 5)

**İkililerde Güncelleme**

İkililer değiştirilemez olduğu için güncelleme ve değiştirme işlemleri yapılamaz.

tuple1 = (18, 3.14)  
tuple2 = ('Aydın', 'İzmir')  
# Aşağıdaki satırdaki kod ikililer için hatalıdır  
# tuple1[0] = 100;  
# Bunun için yeni bir ikili oluştururuz  
tuple3 = tuple1 + tuple2  
print (tuple3)

Yukardaki kod çalıştığında aşağıdaki çıktı oluşur.

(18, 3.14, 'Aydın', 'İzmir')

**İkili Elemanları Silme**

İkili elemanları silme mümkün değildir. del komutu ile bütün ikiliyi silebiliriz.

tuple1 = ('Matematik', 'Kimya', 2012, 2018)

print (tuple1)

del tuple1;

print ("Silme işleminden sonrar tuple1: ")

print (tuple1)

Yukardaki kod çalıştığında aşağıdaki sonuç görünür. Del komutu ile tuple1 ikilisi silindiğinden bir hata verir.

('Matematik', 'Kimya', 2012, 2018)

Silme işleminden sonrar tuple1:

File "Dene.py", line 5, in <module>

print (tuple1)

NameError: name 'tuple1' is not defined

**İkililerde Temel İşlemler**

İkililerde metinlerde olduğu gibi + ve \* operatörleriyle sırasıyla ekleme ve tekrar işlemleri yaptırırız. Çıkan sonuç yeni bir ikili olacaktır.

|  |  |  |
| --- | --- | --- |
| **Python İfadeleri** | **Sonuçlar** | **Açıklama** |
| len((1, 2, 3, 4, 5)) | 5 | Uzunluk |
| (0, 1, 2, 3) + (4, 5, 6, 7) | (0, 1, 2, 3, 4, 5, 6, 7) | Ekleme |
| ('Aydın',) \* 3 | ('Aydın', 'Aydın', 'Aydın') | Tekrar |
| 2 in (0, 1, 2, 3) | True | Üyelik |
| for i in (0,1,2,3) : print (i, end = ' ') | 1 2 3 | Tekrar |

**İndisleme, Parçalara Ayırma ve Matrisler**

İndisleme ve parçalar ayırma işlemi metinlerde olduğu gibi aynı şekilde olur.

Aşağıda bunun için örnekler vardır. T=('Matematik', 'Kimya', 'Fizik')

|  |  |  |
| --- | --- | --- |
| **Python İfadeleri** | **Sonuçlar** | **Açıklamalar** |
| T[2] | 'Fizik' | İndis 0’dan başlar |
| T[-2] | 'Kimya' | Eksililer sağdan başlar. -1 indisi en sağdaki değerdir |
| T[1:] | ('Kimya', 'Fizik') | : ile parçalara ayırır |

**NOT:** Listeler için köşeli parantez, ikililer için ise parantez kullanılır. Nesneleri ayırmak için ise virgül kulllanılır. Eğer parantez yada köşeli parantez kullanılmazsa yani sadece nesneler virgüllerle ayrılırsa standart olarak ikili olarak tanımlanır. Parantez varmış gibi kabul edilir ve ikili oluşturulur.

**Örnek:**

tuple1 = 'Matematik', "Kimya", 123,34  
print (tuple1)

**Çıktı:**

('Matematik', 'Kimya', 123, 34)

Python İkili Fonksiyonlar aşağıda verilmiştir.

|  |  |
| --- | --- |
| **Seçim No** | **Fonksiyonlar ve Açıklamaları** |
| 1 | **cmp(ikili1, ikili2):** İkili 1 ve ikili2’yi karşılaştırır. |
| 2 | **len(ikili):** İkilinin toplam nesne sayısını verir. |
| 3 | **max(ikili):** İkilideki maksimum değeri döndürür. |
| 4 | **min(ikili):** İkilideki minimum değeri döndürür. |
| 5 | **tuple(dizi):** Diziyi ikiliye döndürür. |

**Sözlük (Dictionary)**

Her anahar üst üste iki nokta ile onun değerinden ayrılır. Nesneler virgülle ayrılır. Bütün nesnelerin hepsi küme parantezi arasında olmalıdır. Boş sözlük { } ile belirtilir. Anahtarların hepsi bir birinden farklı olmalıdır. Yani tek olmalıdır. Fakat değerler farklı olmak zorunda değildir. Değerler herhangi bir tipte de olabilir. Anahtarlar ise değiştirilemez veri tiplerninde (metinler, sayılar veya ikililer) olmalıdır.

**Sözlükte Verilere Erişim**

Sözlük değerlerine erişmek için köşeli parantez içinde anahtarlar kullanılır.

Aşağıda sözlük kullanım örnekleri vardır.

sozluk = {'isim': 'Melis', 'yas': 7, 'sinif': 'Birinci'}  
print ("Sözlük['isim']: ", sozluk['isim'])  
print ("Sözlük['yas']: ", sozluk['yas'])

Yukardaki kodların çıktısı aşağıdaki gibi olur.

Sözlük['isim']: Melis

Sözlük['yas']: 7

Sözlükte olmayan bir anahtar kullanılırsa aşağıdaki gibi bir hatayla karşılaşılır.

sozluk = {'isim': 'Melis', 'yas': 7, 'sinif': 'Birinci'}  
print ("Sözlük['agirlik']: ", sozluk['agirlik'])

Yukardaki kod çalıştığında aşağıdaki gibi çıktı oluşur.

Traceback (most recent call last):

File "Dene.py", line 3, in <module>

print ("Sözlük['agirlik']: ", sozluk['agirlik'])

KeyError: 'agirlik'

**Sözlük Güncelleme**

Sözlük güncelleme işlemi anahtar ve değer ekleyerek, değiştirerek veya silerek yapılır.

sozluk = {'isim': 'Melis', 'yas': 7, 'sinif': 'Birinci'}  
sozluk['yas']=8 # Bir değerin değiştirilmmesi  
sozluk['dogumyeri']='Aydın' # Yeni anahtar ve değer ekleniyor  
print ("Sözlük['dogumyeri']: ", sozluk['dogumyeri'])  
print ("Sözlük['yas']: ", sozluk['yas'])

Yukardaki kodlar çalıştığında aşağıdaki çıktı oluşur.

Sözlük['dogumyeri']: Aydın

Sözlük['yas']: 8

**Sözlük Elemanların Silinmesi**

Sözlüğün belirli elemanlarını da silebiliriz, bütün elemanlarını da silebiliriz. Aşağıdaki örnekler sözlüğün elemanlarının silinmesi ile iligilidir.

sozluk = {'isim': 'Ayşe', 'yas': 7, 'sinif': 'Birinci'}  
print(sozluk)  
del sozluk['isim'] # isim anahtarlı elemanı yani Ayşe'yi siler  
print(sozluk)  
sozluk.clear() # Bütün elemanları siler  
print(sozluk)

Yukardaki kodlar çalıştığında aşağıdaki çıktılar oluşur.

{'isim': 'Ayşe', 'yas': 7, 'sinif': 'Birinci'}

{'yas': 7, 'sinif': 'Birinci'}

{}

del komutu ile örnek aşağıda verilmiştir.

sozluk = {'isim': 'Ayşe', 'yas': 7, 'sinif': 'Birinci'}  
del sozluk # Sözlüğün tamamını siler  
print (sozluk)

Yukardaki kodlar çalıştığında aşağıdaki gibi bir çıktı verir. del komutu sözlüğü tamamiyle bellekted siler ve tekrar tanımlanmadan kullanılamaz. Eğer kullanırsak tanımlanmadı şeklinde bir hatayla karşılaşırız.

Traceback (most recent call last):

File "C:Dene.py", line 3, in <module>

print (sozluk)

NameError: name 'sozluk' is not defined

Sözlüğün Anahtar Özellikleri

Sözlüğün değerlerinde hiçbir sınırlama yoktur. Herhangi bir Python nesnesi, standart nesne ya da kullanıcı tanımlı nesne değer olarak alınabilir. Fakat anahtarlar için aynı şeyi söyleyemeyiz. Sözlük anahtarları için iki önemli noktayı belirtmemiz lazım.

(a) Her anahtar için bir tane değer olabilir. Bir sözlükte iki tane aynı anahtara izin verilmez. Bu şekilde sözlük oluşturulursa en sondaki anahtar değeri kullanılır.

sozluk = {'isim': 'Ayşe', 'yas': 7, 'sinif': 'Birinci','isim':'Hakan'}  
print (sozluk['isim'])

Yukardaki kod çalıştığında aşağıdaki çıktı oluşur.

Hakan

(b) İkincisi; anahtarlar bir kere tanımlandıktan sonra değiştirilemez.

Keys must be immutable. Anahtarları metin, sayı veya ikili olabilir, fakat ['anahtar'] şeklinde bir anahtar olamaz.

sozluk = {[isim]: 'Melis', 'yas': 6}

print (sozluk['isim'])

Yukardaki kodlar çalıştığında aşağıdaki çıktı oluşur.

Traceback (most recent call last):

File "C:/Dene.py", line 1, in <module>

sozluk = {[isim]: 'Melis', 'yas': 6}

NameError: name 'isim' is not defined

**Python Fonksiyonları ve Metotları**

|  |  |
| --- | --- |
| **Sıra No** | **Fonksiyonlar ve Açıklamaları** |
| 1 | **len(sozluk):** Sözlüğün uzunluğunu verir. Sözlükteki elemanların sayısını verir. |
| 2 | **str(sozluk):** Sözlüğü metne dönüştürür. |
| 3 | **type(değişken):** Değişkenin tipini döndürür. |

|  |  |
| --- | --- |
| **Sıra No** | **Metotlar ve Açıklamaları** |
| 1 | **sozluk.clear():** Sözlüğün bütü elemanlarını siler; boş sözlük olur. |
| 2 | **sozluk.copy():** Sözlüğün kopyasını döndürür. |
| 3 | **sozluk.fromkeys():** Dizi ve değer kümelerinden değere anahtarlı yeni bir sözlük oluşturur. |
| 4 | **sozluk.get(anahtar, default=None):** Anahtarın değerini döndürür. Anahtar yoksa None döndürür. |
| 5 | **sozluk.items():** Sözlüğün anahtar ve değer çiftini döndürür. |
| 6 | **sozluk.keys():** Anahtarlar listesini döndürür. |
| 7 | **sozluk.setdefault(key, default = None):** Get metodu ile aynıdır. Fakat sözlükte anahtar olmadığında default değer yerleştirilebiliyor. |
| 8 | **sozluk.update(sozluk2):** Sözlük2 sözlüğün anahtar-değer çiftlerini sözlüğe ekler. |
| 9 | **sozluk.values():** Sözlüğün değerler listesini döndürür. |

**Tarih ve Zaman**

Python programları tarih ve zamanla ilgili işlemler birkaç farklı şekilde yapabilir. Tarih formatları arasındaki dönüşüm herzaman yapılan bilgisayar işlerindendir. Python’ın time ve calendar modülleri tarih ve zaman işlemlerinde kullanılır.

Zaman aralığı, reel sayı tipinde saniye birimi olarak ölçülür. Anlık zaman saniye olarak 1 Ocak 1970 öğle 12:00 dan itibaren geçen zamandır.

Python’da time modülünün içinde time.time() fonksiyon vardır. Bu fonksiyon 1 Ocak 1970 öğle 12:00 dan itibaren şu anki zamana kadar geçen zamanı saniye olarak döndürür.

**Örnek:**

import time; # time modülü eklenmelidir.

zaman = time.time()

print ("1 Ocak 1970 öğle 12:00 dan itibaren geçen zaman:", zaman, " saniyedir.")

Yukardaki kod çalıştığında aşağıdaki çıktı oluşur.

1 Ocak 1970 öğle 12:00 dan itibaren geçen zaman: 1537190783.2403712 saniyedir.

**TimeTuple nedir?**

Aşağıda görüldüğü gibi Python’ın zaman fonksiyonlar 9 sayılı bir zaman elemanlarından oluşur.

|  |  |  |
| --- | --- | --- |
| **İndis** | **Alan** | **Değerler** |
| 0 | 4 Rakamlı Yıl | 2018 |
| 1 | Ay | 1-12 |
| 2 | Gün | 1-31 |
| 3 | Saat | 0-23 |
| 4 | Dakika | 0-59 |
| 5 | Saniye | 0-61 (60 veya 61 artık saniyedir) |
| 6 | Haftanın günleri | 0-6 (0-Pazartesi) |
| 7 | Yılın günleri | 1-366 (Julian günü) |
| 8 | Gün ışığından faydalanma | -1, 0, 1, -1: Kütüphane DST(Day Saving Time)’ye karar verecek demektir. |

import time

print (time.localtime());

Yukardaki kod çalıştığında aşağıdaki gibi çıktı verir.

time.struct\_time(tm\_year=2018, tm\_mon=9, tm\_mday=17, tm\_hour=23, tm\_min=55, tm\_sec=25, tm\_wday=0, tm\_yday=260, tm\_isdst=0)

|  |  |  |
| --- | --- | --- |
| **İndis** | **Alanlar** | **Değerler** |
| 0 | tm\_year | 2018 |
| 1 | tm\_mon | 1-12 |
| 2 | tm\_mday | 1-31 |
| 3 | tm\_hour | 0-23 |
| 4 | tm\_min | 0-59 |
| 5 | tm\_sec | 0-61 (60 veya 61: Artık saniyedir) |
| 6 | tm\_wday | 0-6 (0: Pazartesi) |
| 7 | tm\_yday | 1-366 (Julian günü) |
| 8 | tm\_isdst | -1, 0, 1, -1: Kütüphane DST’ye karar verecek demektir. |

**Şimdiki zamanı alma**

Saniye zaman değerinden zaman elemanları şekline dönüştürmek için saniye olarak alınan reel sayı değeri localtime fonksiyonuna aktarılarak 9 zaman elemanı döndürülür.

import time

yerelzaman = time.localtime(time.time())

print ("Yerel şimdiki zaman: ", yerelzaman)

Üstteki kodlar çalıştığında aşağıdaki çıktı oluşur.

Yerel şimdiki zaman: time.struct\_time(tm\_year=2018, tm\_mon=9, tm\_mday=18, tm\_hour=0, tm\_min=10, tm\_sec=3, tm\_wday=1, tm\_yday=261, tm\_isdst=0)

**Formatlanmış zaman**

Herhangibir zamanı ihtiyaca göre formalı zamana dönüştürebiliriz. Bunun için asctime() fonksiyonu kullanıyoruz.

import time

yerelzaman = time.asctime( time.localtime(time.time()) )

print ("Yerel şimdiki zaman :", yerelzaman)

Yukardaki kod çalıştığında aşağıdaki çıktı meydana gelir.

Yerel şimdiki zaman : Tue Sep 18 00:19:09 2018

**Bir ayın takvimini**

Calendar modül aylık ve yıllık takvimi elde etmek için birçok metodu var.

import calendar

takvim = calendar.month(2018, 9)

print ("2018 Eylül Takvimi:")

print (takvim)

Yukardaki kod çalıştığında aşağıdaki çıktı oluşur.

2018 Eylül Takvimi:

September 2018

Mo Tu We Th Fr Sa Su

1 2

3 4 5 6 7 8 9

10 11 12 13 14 15 16

17 18 19 20 21 22 23

24 25 26 27 28 29 30

**Zaman Modülü**

Pythonda popüler zaman (time) modülü bulunmaktadır. Bu modül zaman fonksiyonları ve dönüşüm fonksiyonları içerir.

|  |  |
| --- | --- |
| **Sıra No** | **Fonksiyon ve Açıklaması** |
| 1 | **time.altzone:** Yerel DST zaman diliminin ofseti (saniye olarak). Eğer yerel DST zamandilimi UTC’nin doğusunda ise negatif değer alır (İngiltere dahil batı avrupa). Eğer gün ışığı sıfırdan farklıysa bunu kullanırız. |
| 2 | **time.asctime([tupletime]):** Time-tuple değer alarak 'Tue Dec 11 18:07:14 2008' gibi 24 karakterlik metin döndürür. |
| 3 | **time.clock( ):** O anki işlemci zamanını saniye olarak döndürür. Farklı yaklaşımların hesaplama masraflalrını bulmak için time.clock() değeri time.time()’dan daha faydalıdır. |
| 4 | **time.ctime([saniye]):** asctime(localtime(saniye)) gibi ve argümansız asctime() gibi. |
| 5 | **time.gmtime([saniye]):** Saniye olarak verilen zamanı alır ve UTC’li time-tuple olarak döndürür. |
| 6 | **time.localtime([saniye]):** Yerel zaman olarak döndürür. |
| 7 | **time.mktime(tupletime):** Time-tuple olarak zamanı alır ve saniye olarak zamanı döndürür. |
| 8 | **time.sleep(secs):** Belirtilen saniye kadar thread’i bekletir. |
| 9 | **time.strftime(fmt[,tupletime]):** Yerel zamanda time-tuple’ı alır ve belirtilen metin formatında döndürür. |
| 10 | **time.strptime(str,fmt = '%a %b %d %H:%M:%S %Y'):** Metin fmt formatına göre ayırır ve time-tuple formatında döndürür. |
| 11 | **time.time( ):** 1970 başlangıç zamanından şimdiki zamana kadar olan saniye değerini döndürür. |
| 12 | **time.tzset():** Kütüphane fonksiyonları tarafından kullanılan zaman dönüşümlerini tekrar ayarlar. TZ çevre değişkeni bunun nasıl yapılacağını belirler. |

Time modülünde iki önemli özellik vardır.

|  |  |
| --- | --- |
| **Sıra No** | **Özellik ve Açıklama** |
| 1 | **time.timezone:** Bu özellik, UTC’den DST’siz yerel saat diliminin saniye olarak ofsetidir. America için >0, Avrupa, Asya v Africa için <=0. |
| 2 | **time.tzname:** Bu özellik DST’li ve DST’siz yerel saat dilimini metin olarak döndürür. |

**Takvim Modülü**

calendar modülü takvim ile ilgili fonksiyonları sağlar. Örneğin, bir ayın veya bir yılın metinsel takvimini yazdıran fonksiyonlar gibi.

Takvim, standart olarak haftanın ilk gününü Pazartesi (Monday) kabul eder. Pazar ise son gün olarak alınır. Bunu değiştirmek için calendar.setfirstweekday() fonksiyonu çağrılır.

Aşağıda calendar (Takvim) ile ilgili fonksiyonlar bulunmaktadır.

|  |  |
| --- | --- |
| **Sıra No** | **Fonksiyonlar ve Açıklamaları** |
| 1 | **calendar.calendar(yıl,w = 2,l = 1,c = 6):** Belirtilen yılın takvimini döndürür. C, kolon boşluklarını; w, her tarihin karakterlerindeki boşlukları; l ise her hafta için satırların sayısını verir. |
| 2 | **calendar.firstweekday( ):** Haftanın başladığı ilk günün şimdiki ayarlarmalara göre indisi verir. Standart olarak pazartesi olduğundan 0 dır. |
| 3 | **calendar.isleap(yıl):** Eğer artık yılda bulunuyorsak True döner. Aksi halde False döner. |
| 4 | **calendar.leapdays(y1,y2):** y1 ve y1 yılları arasındaki artık günlerin toplam sayısını döndürür. |
| 5 | **calendar.month(yıl,ay,w = 2,l = 1):** Belirtilen yıl ve aya göre takvimi döndürür. |
| 6 | **calendar.monthcalendar(yıl,ay):** Belirtilen yıl ve aya göre takvimi tamsayılardan oluşan listelerin listesini döndürür. Her alt liste haftayı gösterir. |
| 7 | **calendar.monthrange(yıl,ay):** İki tamsayı döndürür. İlki ayın ilk gününün haftanın kaçıncı günü olduğu; ikincisi ise ayın kaç gün olduğunu belirtiyor. |
| 8 | **calendar.prcal(year,w = 2,l = 1,c = 6):**  Like print calendar.calendar(year,w,l,c). |
| 9 | **calendar.prmonth(yıl,ay,w = 2,l = 1):** Belirtilen yıl ve aya göre takvimi döndürür. |
| 10 | **calendar.setfirstweekday(haftanın ilk günü):** Haftanın ilk gününü yerleştirir. Pazartesi:0-Pazar:6 |
| 11 | **calendar.timegm(tupletime):** time.gmtme fonksiyonunun tersidir. time.tuple formunda olan bir zamanı alıp 1970 tarihinden itibaren saniye olarak zamanı döndürür. |
| 12 | **calendar.weekday(yıl,ay,gün):** Belirtilen yıl, ay ve güne göre haftanın gününün kodunu verir. |

**Not:** Pythonda başka zaman ve takvim modülleri de vardır: datetime, pytz ve dateutil.

**Fonksiyonlar**

Fonksiyon, belirli bir işi gerçekleştirmek için planlanmış ve tekrar kullanılabilen kod bloklarıdır. Fonksiyonlar uygulamalarımızın daha modüler olmasını ve kodların tekrar kullanılabilir olmasını sağlar.

Pythonda bilindiği gibi önceden oluşturulmuş (örneğin **print** fonksiyonu) hazır fonksiyonlar vardır. Fakat kendi fonksiyonlarımızı da tanımlayabiliriz. Bu fonksiyonlara **kullanıcı-tanımlı** fonksiyonlar denir.

**Fonksiyon Tanımlama**

Belirli bir işlevi yapmak için fonksiyonlar tanımlanabilir. Fonksiyonlar **def** anahtar kelimesi ile başlar, sonra fonksiyon ismi ve parantezler kullanılır. Parantezler içinde girdi argümanları kullanılabilir. Parantezler içinde parametrelerde kullanılabilir. Fonksiyonun ilk ifadesi, bir opsiyonel ifade olabilir. Bu ifade fonksiyonun dokümantasyon (**docstring**) metnidir. Fonksiyonun başlık satırı üst üste iki nokta (:) ile devam eder. Eğer fonksiyondan bir değer döndürülecekse **return** ifadesi kullanılır.

**Fonksiyon Yazım Şekli**

def fonksiyonismi( argüman değişkenleri ):

"fonksiyon dokümantasyon metni"

fonksiyon\_suite

return [ifade]

**Örnek:**

Aşağıdaki fonksiyon bir metni girdi olarak alır ve bu metni ekranda gösterir.

def yazdir( metin ):

"Bu fonksiyon bir metni alıp ekranda gösterir."

print (metin)

return

**Fonksiyon Çağırma**

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the printme()function −

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme("This is first call to the user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

This is first call to the user defined function!

Again second call to the same function

## **Pass by Reference vs Value**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/89otk5)

#!/usr/bin/python3

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

print ("Values inside the function before change: ", mylist)

mylist[2]=50

print ("Values inside the function after change: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30]

changeme( mylist )

print ("Values outside the function: ", mylist)

Here, we are maintaining reference of the passed object and appending values in the same object. Therefore, this would produce the following result −

Values inside the function before change: [10, 20, 30]

Values inside the function after change: [10, 20, 50]

Values outside the function: [10, 20, 50]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/tJpndT)

#!/usr/bin/python3

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4] # This would assi new reference in mylist

print ("Values inside the function: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30]

changeme( mylist )

print ("Values outside the function: ", mylist)

The parameter **mylist** is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

## **Function Arguments**

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

## **Required Arguments**

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function **printme()**, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

[Live Demo](http://tpcg.io/ceXIug)

#!/usr/bin/python3

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

## **Keyword Arguments**

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the **printme()** function in the following ways −

[Live Demo](http://tpcg.io/cOAFB8)

#!/usr/bin/python3

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives a clearer picture. Note that the order of parameters does not matter.

[Live Demo](http://tpcg.io/6Vv9cq)

#!/usr/bin/python3

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age = 50, name = "miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

## **Default Arguments**

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

[Live Demo](http://tpcg.io/e6K9mu)

#!/usr/bin/python3

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age = 50, name = "miki" )

printinfo( name = "miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

## **Variable-length Arguments**

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is given below −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

[Live Demo](http://tpcg.io/JFDYk7)

#!/usr/bin/python3

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print ("Output is: ")

print (arg1)

for var in vartuple:

print (var)

return

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

## **The Anonymous Functions**

These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword. You can use the **lambda** keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression.
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

### Syntax

The syntax of **lambda** functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is an example to show how **lambda** form of function works −

[Live Demo](http://tpcg.io/GXNTkx)

#!/usr/bin/python3

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2

# Now you can call sum as a function

print ("Value of total : ", sum( 10, 20 ))

print ("Value of total : ", sum( 20, 20 ))

When the above code is executed, it produces the following result −

Value of total : 30

Value of total : 40

## **The return Statement**

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the examples given below are not returning any value. You can return a value from a function as follows −

[Live Demo](http://tpcg.io/d2UiXP)

#!/usr/bin/python3

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print ("Inside the function : ", total)

return total

# Now you can call sum function

total = sum( 10, 20 )

print ("Outside the function : ", total )

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

## **Scope of Variables**

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

## **Global vs. Local variables**

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

[Live Demo](http://tpcg.io/mR877c)

#!/usr/bin/python3

total = 0 # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print ("Inside the function local total : ", total)

return total

# Now you can call sum function

sum( 10, 20 )

print ("Outside the function global total : ", total )

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0

# **Python 3 - Modules**

Advertisements

[Previous Page](https://www.tutorialspoint.com/python3/python_functions.htm)

[Next Page](https://www.tutorialspoint.com/python3/python_files_io.htm)

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

### Example

The Python code for a module named aname normally resides in a file namedaname.py. Here is an example of a simple module, support.py −

def print\_func( par ):

print "Hello : ", par

return

## **The import Statement**

You can use any Python source file as a module by executing an import statement in some other Python source file. The **import**has the following syntax −

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script −

#!/usr/bin/python3

# Import module support

import support

# Now you can call defined function that module as follows

support.print\_func("Zara")

When the above code is executed, it produces the following result −

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

## **The from...import Statement**

Python's **from** statement lets you import specific attributes from a module into the current namespace. The **from...import** has the following syntax −

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement −

#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a + b

return result

>>> from fib import fib

>>> fib(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

## **The from...import \* Statement**

It is also possible to import all the names from a module into the current namespace by using the following import statement −

from modname import \*

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

## **Executing Modules as Scripts**

Within a module, the module’s name (as a string) is available as the value of the global variable \_\_name\_\_. The code in the module will be executed, just as if you imported it, but with the \_\_name\_\_ set to "\_\_main\_\_".

Add this code at the end of your module −

[Live Demo](http://tpcg.io/m8xId0)

#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a + b

return result

if \_\_name\_\_ == "\_\_main\_\_":

f = fib(100)

print(f)

When you run the above code, the following output will be displayed.

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

## **Locating Modules**

When you import a module, the Python interpreter searches for the module in the following sequences −

* The current directory.
* If the module is not found, Python then searches each directory in the shell variable PYTHONPATH.
* If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python3/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

## **The PYTHONPATH Variable**

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python34\lib;

And here is a typical PYTHONPATH from a UNIX system −

set PYTHONPATH = /usr/local/lib/python

## **Namespaces and Scoping**

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

* A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.
* Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
* Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
* Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
* The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable.

However, we accessed the value of the local variable *Money*before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

[Live Demo](http://tpcg.io/lHdtHD)

#!/usr/bin/python3

Money = 2000

def AddMoney():

# Uncomment the following line to fix the code:

# global Money

Money = Money + 1

print (Money)

AddMoney()

print (Money)

## **The dir( ) Function**

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

[Live Demo](http://tpcg.io/NLz0dc)

#!/usr/bin/python3

# Import built-in module math

import math

content = dir(math)

print (content)

When the above code is executed, it produces the following result −

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']

Here, the special string variable \_\_name\_\_ is the module's name, and \_\_file\_\_ is the filename from which the module was loaded.

## **The globals() and locals() Functions**

The **globals()** and **locals()** functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

* If **locals()** is called from within a function, it will return all the names that can be accessed locally from that function.
* If **globals()** is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the **keys()** function.

## **The reload() Function**

When a module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

reload(module\_name)

Here, module\_name is the name of the module you want to reload and not the string containing the module name. For example, to reload hello module, do the following −

reload(hello)

## **Packages in Python**

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in Phone directory. This file has the following line of source code −

#!/usr/bin/python3

def Pots():

print ("I'm Pots Phone")

Similar, we have other two files having different functions with the same name as above. They are −

* *Phone/Isdn.py* file having function Isdn()
* *Phone/G3.py* file having function G3()

Now, create one more file \_\_init\_\_.py in the *Phone* directory −

* Phone/\_\_init\_\_.py

To make all of your functions available when you have imported Phone, you need to put explicit import statements in \_\_init\_\_.py as follows −

from Pots import Pots

from Isdn import Isdn

from G3 import G3

After you add these lines to \_\_init\_\_.py, you have all of these classes available when you import the Phone package.

#!/usr/bin/python3

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

When the above code is executed, it produces the following result −

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single function in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

# **Python 3 - Files I/O**

Advertisements

[Previous Page](https://www.tutorialspoint.com/python3/python_modules.htm)

[Next Page](https://www.tutorialspoint.com/python3/python_exceptions.htm)

This chapter covers all the basic I/O functions available in Python 3. For more functions, please refer to the standard Python documentation.

## **Printing to the Screen**

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows −

[Live Demo](http://tpcg.io/fYN916)

#!/usr/bin/python3

print ("Python is really a great language,", "isn't it?")

This produces the following result on your standard screen −

Python is really a great language, isn't it?

## **Reading Keyboard Input**

Python 2 has two built-in functions to read data from standard input, which by default comes from the keyboard. These functions are **input()** and **raw\_input()**

In Python 3, raw\_input() function is deprecated. Moreover, input() functions read data from keyboard as string, irrespective of whether it is enclosed with quotes ('' or "" ) or not.

## **The input Function**

The **input([prompt])** function is equivalent to raw\_input, except that it assumes that the input is a valid Python expression and returns the evaluated result to you.

#!/usr/bin/python3

>>> x = input("something:")

something:10

>>> x

'10'

>>> x = input("something:")

something:'10' #entered data treated as string with or without ''

>>> x

"'10'"

## **Opening and Closing Files**

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

## **The open Function**

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a **file**object, which would be utilized to call other support methods associated with it.

### Syntax

file object = open(file\_name [, access\_mode][, buffering])

Here are parameter details −

* **file\_name** − The file\_name argument is a string value that contains the name of the file that you want to access.
* **access\_mode** − The access\_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
* **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file −

|  |  |
| --- | --- |
| **Sr.No.** | **Mode & Description** |
| 1 | **r**  Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| 2 | **rb**  Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| 3 | **r+**  Opens a file for both reading and writing. The file pointer placed at the beginning of the file. |
| 4 | **rb+**  Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| 5 | **w**  Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| 6 | **wb**  Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| 7 | **w+**  Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| 8 | **wb+**  Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| 9 | **a**  Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| 10 | **ab**  Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| 11 | **a+**  Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| 12 | **ab+**  Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

## **The file Object Attributes**

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all the attributes related to a file object −

|  |  |
| --- | --- |
| **Sr.No.** | **Attribute & Description** |
| 1 | **file.closed**  Returns true if file is closed, false otherwise. |
| 2 | **file.mode**  Returns access mode with which file was opened. |
| 3 | **file.name**  Returns name of the file. |

**Note** − softspace attribute is not supported in Python 3.x

### Example

[Live Demo](http://tpcg.io/Ux8rpv)

#!/usr/bin/python3

# Open a file

fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

print ("Closed or not : ", fo.closed)

print ("Opening mode : ", fo.mode)

fo.close()

This produces the following result −

Name of the file: foo.txt

Closed or not : False

Opening mode : wb

## **The close() Method**

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

### Syntax

fileObject.close();

### Example

[Live Demo](http://tpcg.io/xXErpK)

#!/usr/bin/python3

# Open a file

fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

# Close opened file

fo.close()

This produces the following result −

Name of the file: foo.txt

## **Reading and Writing Files**

The file object provides a set of access methods to make our lives easier. We would see how to use **read()** and **write()**methods to read and write files.

## **The write() Method**

The **write()** method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

### Syntax

fileObject.write(string);

Here, passed parameter is the content to be written into the opened file.

### Example

#!/usr/bin/python3

# Open a file

fo = open("foo.txt", "w")

fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file

fo.close()

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content −

Python is a great language.

Yeah its great!!

## **The read() Method**

The **read()** method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

### Syntax

fileObject.read([count]);

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

### Example

Let us take a file *foo.txt*, which we created above.

#!/usr/bin/python3

# Open a file

fo = open("foo.txt", "r+")

str = fo.read(10)

print ("Read String is : ", str)

# Close opened file

fo.close()

This produces the following result −

Read String is : Python is

## **File Positions**

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The **offset** argument indicates the number of bytes to be moved. The **from** argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would be taken as the reference position.

### Example

Let us take a file *foo.txt*, which we created above.

#!/usr/bin/python3

# Open a file

fo = open("foo.txt", "r+")

str = fo.read(10)

print ("Read String is : ", str)

# Check current position

position = fo.tell()

print ("Current file position : ", position)

# Reposition pointer at the beginning once again

position = fo.seek(0, 0)

str = fo.read(10)

print ("Again read String is : ", str)

# Close opened file

fo.close()

This produces the following result −

Read String is : Python is

Current file position : 10

Again read String is : Python is

## **Renaming and Deleting Files**

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module, you need to import it first and then you can call any related functions.

## **The rename() Method**

The **rename()** method takes two arguments, the current filename and the new filename.

### Syntax

os.rename(current\_file\_name, new\_file\_name)

### Example

Following is an example to rename an existing file *test1.txt* −

#!/usr/bin/python3

import os

# Rename a file from test1.txt to test2.txt

os.rename( "test1.txt", "test2.txt" )

## **The remove() Method**

You can use the **remove()** method to delete files by supplying the name of the file to be deleted as the argument.

### Syntax

os.remove(file\_name)

### Example

Following is an example to delete an existing file *test2.txt* −

#!/usr/bin/python3

import os

# Delete file test2.txt

os.remove("text2.txt")

## **Directories in Python**

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

## **The mkdir() Method**

You can use the **mkdir()** method of the **os** module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

### Syntax

os.mkdir("newdir")

### Example

Following is an example to create a directory *test* in the current directory −

#!/usr/bin/python3

import os

# Create a directory "test"

os.mkdir("test")

## **The chdir() Method**

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

### Syntax

os.chdir("newdir")

### Example

Following is an example to go into "/home/newdir" directory −

#!/usr/bin/python3

import os

# Changing a directory to "/home/newdir"

os.chdir("/home/newdir")

## **The getcwd() Method**

The **getcwd()** method displays the current working directory.

### Syntax

os.getcwd()

### Example

Following is an example to give current directory −

#!/usr/bin/python3

import os

# This would give location of the current directory

os.getcwd()

## **The rmdir() Method**

The **rmdir()** method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

### Syntax

os.rmdir('dirname')

### Example

Following is an example to remove the "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

#!/usr/bin/python3

import os

# This would remove "/tmp/test" directory.

os.rmdir( "/tmp/test" )

## **File and Directory Related Methods**

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

* [File Object Methods](https://www.tutorialspoint.com/python3/file_methods.htm)

The **file** object provides functions to manipulate files.

* [OS Object Methods](https://www.tutorialspoint.com/python3/os_file_methods.htm)

This provides methods to process files as well as directories.

# **Python 3 - Exceptions Handling**

Advertisements

[Previous Page](https://www.tutorialspoint.com/python3/python_files_io.htm)

[Next Page](https://www.tutorialspoint.com/python3/python_classes_objects.htm)

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

* **Exception Handling** − This would be covered in this tutorial. Here is a list standard Exceptions available in Python − [Standard Exceptions](https://www.tutorialspoint.com/python3/python_exceptions.htm#standard_exceptions).
* **Assertions** − This would be covered in [Assertions in Python 3](https://www.tutorialspoint.com/python3/python_exceptions.htm#assertion_python) tutorial.

## **Standard Exceptions**

Here is a list of Standard Exceptions available in Python. −

|  |  |
| --- | --- |
| **Sr.No.** | **Exception Name & Description** |
| 1 | **Exception**  Base class for all exceptions |
| 2 | **StopIteration**  Raised when the next() method of an iterator does not point to any object. |
| 3 | **SystemExit**  Raised by the sys.exit() function. |
| 4 | **StandardError**  Base class for all built-in exceptions except StopIteration and SystemExit. |
| 5 | **ArithmeticError**  Base class for all errors that occur for numeric calculation. |
| 6 | **OverflowError**  Raised when a calculation exceeds maximum limit for a numeric type. |
| 7 | **FloatingPointError**  Raised when a floating point calculation fails. |
| 8 | **ZeroDivisonError**  Raised when division or modulo by zero takes place for all numeric types. |
| 9 | **AssertionError**  Raised in case of failure of the Assert statement. |
| 10 | **AttributeError**  Raised in case of failure of attribute reference or assignment. |
| 11 | **EOFError**  Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| 12 | **ImportError**  Raised when an import statement fails. |
| 13 | **KeyboardInterrupt**  Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| 14 | **LookupError**  Base class for all lookup errors. |
| 15 | **IndexError**  Raised when an index is not found in a sequence. |
| 16 | **KeyError**  Raised when the specified key is not found in the dictionary. |
| 17 | **NameError**  Raised when an identifier is not found in the local or global namespace. |
| 18 | **UnboundLocalError**  Raised when trying to access a local variable in a function or method but no value has been assigned to it. |
| 19 | **EnvironmentError**  Base class for all exceptions that occur outside the Python environment. |
| 20 | **IOError**  Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| 21 | **OSError**  Raised for operating system-related errors. |
| 22 | **SyntaxError**  Raised when there is an error in Python syntax. |
| 23 | **IndentationError**  Raised when indentation is not specified properly. |
| 24 | **SystemError**  Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| 25 | **SystemExit**  Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| 26 | **TypeError**  Raised when an operation or function is attempted that is invalid for the specified data type. |
| 27 | **ValueError**  Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| 28 | **RuntimeError**  Raised when a generated error does not fall into any category. |
| 29 | **NotImplementedError**  Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

## **Assertions in Python**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

* The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.
* Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.
* Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

### The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The syntax for assert is −

assert Expression[, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception, using the try-except statement. If they are not handled, they will terminate the program and produce a traceback.

### Example

Here is a function that converts a given temperature from degrees Kelvin to degrees Fahrenheit. Since 0° K is as cold as it gets, the function bails out if it sees a negative temperature −

[Live Demo](http://tpcg.io/r1PgMi)

#!/usr/bin/python3

def KelvinToFahrenheit(Temperature):

assert (Temperature >= 0),"Colder than absolute zero!"

return ((Temperature-273)\*1.8)+32

print (KelvinToFahrenheit(273))

print (int(KelvinToFahrenheit(505.78)))

print (KelvinToFahrenheit(-5))

When the above code is executed, it produces the following result −

32.0

451

Traceback (most recent call last):

File "test.py", line 9, in <module>

print KelvinToFahrenheit(-5)

File "test.py", line 4, in KelvinToFahrenheit

assert (Temperature >= 0),"Colder than absolute zero!"

AssertionError: Colder than absolute zero!

## **What is Exception?**

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

## **Handling an exception**

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

### Syntax

Here is simple syntax of *try....except...else* blocks −

try:

You do your operations here

......................

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

......................

else:

If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax −

* A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
* You can also provide a generic except clause, which handles any exception.
* After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
* The else-block is a good place for code that does not need the try: block's protection.

### Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

[Live Demo](http://tpcg.io/oxCWSE)

#!/usr/bin/python3

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print ("Error: can\'t find file or read data")

else:

print ("Written content in the file successfully")

fh.close()

This produces the following result −

Written content in the file successfully

### Example

This example tries to open a file where you do not have the write permission, so it raises an exception −

[Live Demo](http://tpcg.io/ceZWjn)

#!/usr/bin/python3

try:

fh = open("testfile", "r")

fh.write("This is my test file for exception handling!!")

except IOError:

print ("Error: can\'t find file or read data")

else:

print ("Written content in the file successfully")

This produces the following result −

Error: can't find file or read data

## **The except Clause with No Exceptions**

You can also use the except statement with no exceptions defined as follows −

try:

You do your operations here

......................

except:

If there is any exception, then execute this block.

......................

else:

If there is no exception then execute this block.

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

## **The except Clause with Multiple Exceptions**

You can also use the same *except* statement to handle multiple exceptions as follows −

try:

You do your operations here

......................

except(Exception1[, Exception2[,...ExceptionN]]]):

If there is any exception from the given exception list,

then execute this block.

......................

else:

If there is no exception then execute this block.

## **The try-finally Clause**

You can use a **finally:** block along with a **try:** block. The **finally:**block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

try:

You do your operations here;

......................

Due to any exception, this may be skipped.

finally:

This would always be executed.

......................

**Note** − You can provide except clause(s), or a finally clause, but not both. You cannot use *else* clause as well along with a finally clause.

### Example

[Live Demo](http://tpcg.io/RR5c1J)

#!/usr/bin/python3

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

finally:

print ("Error: can\'t find file or read data")

fh.close()

If you do not have permission to open the file in writing mode, then this will produce the following result −

Error: can't find file or read data

Same example can be written more cleanly as follows −

[Live Demo](http://tpcg.io/s2jDA4)

#!/usr/bin/python3

try:

fh = open("testfile", "w")

try:

fh.write("This is my test file for exception handling!!")

finally:

print ("Going to close the file")

fh.close()

except IOError:

print ("Error: can\'t find file or read data")

This produces the following result −

Going to close the file

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

## **Argument of an Exception**

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

try:

You do your operations here

......................

except ExceptionType as Argument:

You can print value of Argument here...

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### Example

Following is an example for a single exception −

[Live Demo](http://tpcg.io/aP6Dfo)

#!/usr/bin/python3

# Define a function here.

def temp\_convert(var):

try:

return int(var)

except ValueError as Argument:

print ("The argument does not contain numbers\n", Argument)

# Call above function here.

temp\_convert("xyz")

This produces the following result −

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'

## **Raising an Exception**

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows −

### Syntax

raise [Exception [, args [, traceback]]]

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

### Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

def functionName( level ):

if level <1:

raise Exception(level)

# The code below to this would not be executed

# if we raise the exception

return level

**Note** − In order to catch an exception, an "except" clause must refer to the same exception thrown either as a class object or a simple string. For example, to capture the above exception, we must write the except clause as follows −

try:

Business Logic here...

except Exception as e:

Exception handling here using e.args...

else:

Rest of the code here...

The following example illustrates the use of raising an exception −

[Live Demo](http://tpcg.io/j1AB30)

#!/usr/bin/python3

def functionName( level ):

if level <1:

raise Exception(level)

# The code below to this would not be executed

# if we raise the exception

return level

try:

l = functionName(-10)

print ("level = ",l)

except Exception as e:

print ("error in level argument",e.args[0])

This will produce the following result

error in level argument -10

## **User-Defined Exceptions**

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

class Networkerror(RuntimeError):

def \_\_init\_\_(self, arg):

self.args = arg

So once you have defined the above class, you can raise the exception as follows −

try:

raise Networkerror("Bad hostname")

except Networkerror,e:

print e.args