

Python Programlama Dili

Kurs Notları

Kaan ASLAN

C ve Sistem Programcılar Derneği

Son Güncelleme Tarihi: 02/06/2022

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabılır.

1. Temel Kavramlar ve Genel Bilgiler

Bu bölümde Python Programlama Dili hakkında bazı temel kavramların açıklaması yapılacak ve genel bilgiler verilecektir.

1.1. Python Nasıl Bir Dildir?

Python ("pay(th)ın, pay(th)an " diye okunuyor) yüksek seviyeli genel amaçlı prosedürel, fonksiyonel ve nesne yönelimli programlama modellerini destekleyen çok modelli (multiparadigm) bir programlama dilidir. Dil Guido van Rossum ("guido ven rasim" biçiminde okunuyor) isimli Hollandalı programcı tarafından 1991 yılında tasarlanmıştır. (HTTP protokolünün yanı WWW'nin de aynı yıl tasarlandığını, C ve Sistem Programcılar Derneği'nin de 1993'te kurulduğunu anımsayınız.)

Temel olarak Python dili yorumlayıcılarla kullanılmaktadır. Python yorumlayıcıları pek çok işletim sistemi için port edilmiştir. Dolayısıyla biz Windows, Mac OS X ve Linux sistemlerinde Python Programlama Dilinde uygulamalar geliştirebiliriz.

Python dinamik tür sistemine (dynamic type system) sahip bir programlama dilidir. Dinamik tür sistemine sahip programlama dillerinde değişkenlerin değil nesnelerin türleri vardır. Başka bir deyişle bu dillerde bir değişkene herhangi bir türden değer atayabiliriz. Farklı bir türden değer atananan kadar artık o değişken o türden olur.

1.2. Python Dilinin Tarihsel Gelişimi

Guido van Rossum Python'ın tasarımına ve gerçekleştirilmesine 1989 yılının sonlarında başlamıştır. Bu tarihten itibaren dilin pek çok versiyonu oluşturulmuştur. Aşağıda versyonların çıkış tarihlerini görüyorsunuz:

Aralık 1989	Tasarım ve gerçeklestirime başlandı
1990	İlk versiyonlara içsel numnaralar verilmiştir
Şubat 1991	0.9
Ocak 1994	1.0
Ekim 1994	1.1
Ekim 1995	1.3
Ekim 1996	1.4
Ocak 1998	1.5
Eylül 2000	1.6
Ekim 2000	2.0
Nisan 2001	2.1
Aralık 2001	2.2

Temmuz 2003	2.3
Kasım 2004	2.4
Eylül 2006	2.5
Ekim 2008	2.6
Aralık 2008	3.0
Haziran 2009	3.1
Şubat 2011	3.2
Eylül 2012	3.3
Mart 2014	3.4
Eylül 2015	3.5
Aralık 2016	3.6
Haziran 2018	3.7
Mart 2019	3.7.3
Ekim 2019	3.8.3
Ekim 2020	3.9
Kasım-2022	3.10
Mart-2022	3.10.4

Kursun yapıldığı tarihteki Python'ın en son sürümü 3.10.4'tür. Bu sürüm Mart 2022'de piyasaya sürülmüştür. Python Programlama Dilinin geliştirilmesi 2001'den bu yana "Python Software Foundation" isimli kurum tarafından yapılmaktadır (www.python.org).

1.3. Python Dili, Gerçekleştirimleri ve Dağıtımları

Python standardizasyon komiteleri tarafından resmi olarak standardize edilmiş bir dil değildir. Dilin sentaks ve semantik özellikleri "Python Language Reference" isimli dokümanlarda tanımlanmıştır (<https://docs.python.org/3/reference/index.html>).

Python dünyasında "gerçekleştirme (implementation)" denildiğinde yorumlayıcıları ve kütüphaneleri anlamışmaktadır. Python dilinin pek çok gerçekleştirme (implementation) vardır. Yukarıda da belirtildiği gibi Python yorumlayıcı (interpreter) ile çalışan bir dildir. Biz kodumuzu derleyiciye sokarak değil yorumlayıcıya sokarak çalıştırırız. Python için pek çok yorumlayıcı yazılmıştır. Bu yorumlayıcıların bazıları bağımsız olarak sıfırdan gerçekleştirilirken bazıları ise bazı diğer gerçekleştirimler temel alınarak onlardan klonlanarak gerçekleştirilmiş durumdadır.

Python'ın en önemli ve en çok kullanılan gerçekleştirimini "Python Software Foundation" tarafından sürdürulen (ilk versiyonları Guido van Rossum tarafından yazılmıştır) olan CPython isimli yazılımdır. Bu gerçekleştirme doğrudan "python.org" sitesinden indirilebilir. Biz de kursumuzda CPython yorumlayıcısı ve kütüphanelerini kullanacağız. CPython C Programlama Dili kullanılarak gerçekleştirilmiştir. CPython arakod tabanlı bir çalışma sistemine sahiptir. İleride ele alınacağı gibi CPython'da import edilen modüller önce yorumlayıcı tarafından "Python bytecode" denilen bir ara koda dönüştürülmemekte, daha sonra da bu arakod yorumlanarak çalıştırılmaktadır. CPython "cross platform" bir yorumlayıcı sistemidir. Windows, Mac OS X ve Linux sistemlerinde kullanılan sürümleri vardır.

Jython (cay(th)ın, cay(th)an biçiminde okunuyor) isimli yorumlayıcı sistemi Java'da yazılmıştır. Jython'da yazılmış olan programlar Java sınıflarını doğrudan kullanabilirler. Örneğin Jython da GUI uygulamaları Java'nın AWT, Swing ya da SWT kütüphaneleri kullanılarak gerçekleştirilebilmektedir. Jython yorumlayıcı çıktı olarak "Java Bytecode" üretmektedir. Üretilen bu çıktı Java çalışma ortamı olan JVM (Java Virtual Machine) tarafından çalıştırılabilenmektedir. Jython ve CPython gerçekleştirimleri arasında dil bakımından küçük farklılıklar vardır.

IronPython gerçekleştirmi Jython gerçekleştiriminin .NET (genel ismiyle CLI) versiyonu gibidir. IronPython C# Programlama Dili kullanılarak yazılmıştır. Visual Studio IDE'si için de "Python Tools for Visual Studio" isminde bir eklentisi de vardır. IronPython CLI (Common Language Infrastructure) arakodu üretmektedir. Üretilen bu arakod .NET'in (ya da Mono projesinin) CLR ortamı tarafından çalıştırılmaktadır.

PyPy önemli Python gerçekleştirimlerinden biridir. PyPy üretilen ara kodu yorumlayıcı olarak değil JIT Derlemesi (Just in Time Compilation) yaparak çalıştırır. Bu yüzden PyPy standard Python gerçekleştirmi olan CPython'dan daha yüksek bir çalışma zamanı performansına sahiptir.

Kursumuzda Python yorumlayıcısı olarak klasik CPython kullanılacaktır. Ayrıca yukarıdakiler dışında Python'ın daha pek çok gerçekleştirmi de vardır. Bu gerçekleştirimlerin listesini <https://wiki.python.org/moin/PythonImplementations> sitesinden görebilirsiniz.

Python dağıtımları belli bir Python gerçekleştirmi temel alınıp onlara çeşitli araçlar eklenerek oluşturulmuş paketlerdir. Yani Python dağıtımları hem Python yorumlayıcılarını hem de uygulama geliştirmeye yardımcı olabilecek birtakım araçları barındırmaktadır. Burada en çok kullanılan bazı Python dağıtımları üzerinde duracağız.

CPython dağıtımını kendi içerisinde CPython gerçekleştirimini, çeşitli kütüphaneleri ve IDLE (Integrated Development and Learning Environment) isimli basit bir IDE'yi barındırmaktadır. Biz de kursumuzun başlangıç bölümlerinde CPython dağıtımını kullanacağız.

Continuum Analytics isimli firma tarafından (bu firmanın ismi 2017'de Anaconda olarak değiştirilmiştir açık kaynak kodlu (dolayısıyla da ücretsiz) olarak dağıtılan Anaconda CPython'dan sonra en çok kullanılan dağıtımlardan biridir. Anaconda yorumlayıcı sistem olarak CPython kullanmaktadır.

ActiveState firması (bu firma aynı zamanda Komodo IDE'sinin üreticisidir) tarafından geliştirilmiş ActivePython dağıtımını en yaygın kullanılan dağıtımlardan biridir. ActivePython dağıtımının ücretli ve "Community Edition" ismiyle ücretsiz sürümleri de vardır. ActivePython yorumlayıcı sistem olarak CPython gerçekleştirimini kullanmaktadır.

Diger Python dağıtımlarına <https://wiki.python.org/moin/PythonDistributions> sitresinden göz gezdirebilirsiniz. Kursumuzda Python dağıtımını olarak CPython ve Anaconda kullanacaktır.

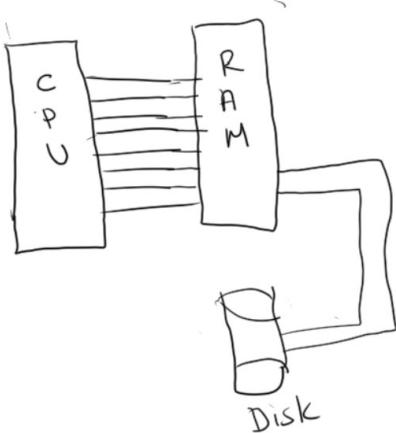
Bir gerçekleştirim ya da bir dağıtım olmayan Python IDE'leri de vardır. Örneğin IntelliJ IDEA firmasının PyCharm isimli IDE'si sıkça kullanılmaktadır.

1.4. Python Ortamının Kurulumu

Python denildiğinde ilk akla gelen şüphesiz CPython dağıtımıdır. Bu dağıtım doğrudan python.org sitesinden indirilerek kurulabilir. Benzer biçimde Anaconda dağıtımını da kendi web sitesinden indirilerek basit bir biçimde kurulabilmektedir. Kursumuzda kullanacağımız PyCharm IDE'sinin Community versiyonu de kendi web sitelerinden indirilerek kurulabilir.

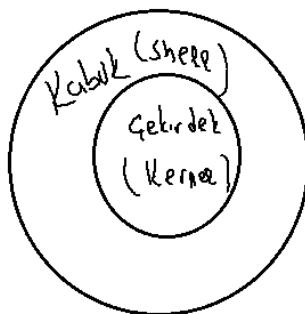
1.5. Temel Bilgisayar Mimarisi

Bir bilgisayar mimarisi çok kabaca üç bileşenden oluşur: CPU, RAM ve Disk. Bilgisayarda tüm işlemlerin yapıldığı entegre devre biçiminde üretilmiş çiplere "mikroişlemci" ya da kavramsal olarak "CPU (Central Processing Unit)" denilmektedir. CPU elektriksel olarak RAM denilen bellek ile bağlantı halindedir. Böyle CPU'nun doğrudan elektriksel olarak bağlantı halinde olduğu belleklere kavramsal olarak "birincil bellek (primary memory)" ya da "ana bellek (main memory)" denilmektedir. Ana bellekler entegre devre modülleri biçiminde üretilmektedir. Bu fiziksel modüller RAM biçiminde isimlendirilmektedir. CPU çalışırken sürekli RAM ile iletişim halindedir. Oradan bilgileri çeker, işler ve oraya geri yazar. Programlama dillerindeki değişkenler RAM'de yaratılmaktadır. Bilgisayarın elektriğini kestigimizde CPU durur. RAM'in içerisindeki bilgiler de kaybolur. Bilginin kalıcılığını sağlamak için diskler kullanılmaktadır. Diskler manyetik temelli elektromekanik biçimde olabileceği gibi tamamen yarı iletkenlerle (flash EPROM, SSD de denilmektedir) de gerçekleştirilebilmektedir. Elektromekanik olarak üretilmiş disklere kişisel bilgisayarlarda "hard disk" de denilmektedir. Bugün artık "flash EPROM" biçiminde üretilen SSD'ler hard disklerin yerini almaya başlamıştır. Ancak kavramsal olarak hard diskler, SSD'ler, CD ve DVD ROM'lar, memory sticklere "ikincil bellek (secondary memory)" denilmektedir. Disk kavramı da çoğu kez bunların hepsini içerecek biçimde kullanılmaktadır.



1.6. İşletim Sistemi (Operating System)

İşletim sistemi makinenin donanımını yöneten, makine ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. İşletim sistemi olmasa daha biz bilgisayarı açtığımızda bile bir şeyler göremeyiz. İşletim sistemleri iki katmandan oluşmaktadır. Çekirdek (kernel), makinenin donanımını yöneten kontrol kısımdır. Kabuk (shell) ise kullanıcıyla arayüz oluşturan kısımdır. (Örneğin Windows'ta masaüstü kabuk görevindedir. Biz çekirdeği bakarak göremeyiz.) Tabii işletim sistemi yazmanın asıl önemli kısmı çekirdeğin yazılımıdır. Çekirdek işletim sistemlerinin motor kısımidır.



Bugün çok kullanılan bazı işletim sistemleri şunlardır:

- Windows
- Linux
- BSD'ler
- Mac OS X
- Android
- iOS
- Windows Mobile (Windows CE)
- Solaris
- QNX
- ...

İşletim sistemlerinin bir kısmı açık kaynak kodlu bir kısmı da mülkiyete bağlıdır. Örneğin Linux, BSD açık kaynak kodlu olduğu halde Windows mülkiyete sahip bir işletim sistemidir. Mac OS X sistemlerinin çekirdeği açıktır (buna Darwin deniyor) ancak geri kalan kısmı kapalıdır.

Bazı işletim sistemleri diğer sistemlerin kodları değiştirilerek gerçekleştirilmiştir. Bazıları sıfırdan (orijinal kod tabanına sahip) yazılmışlardır. Orijinal kod tabanına sahip olan yani sıfırdan yazılmış olan işletim sistemlerinden bazıları şunlardır:

- Microsoft Windows

- Linux
- BSD'ler
- Solaris

Android Linux işletim sisteminin çekirdek kodları alınarak bazı modüllerin atılması ve mobil cihazlara yönelik bazı işlevselliklerin eklenmesiyle gerçekleştirilmiş bir sistemdir. Benzer biçimde IOS da Mac OS X kodlarından devşirilmiştir. Darwin çekirdeği Free BSD ve Mach isimli çekirdeklerin birleştirilmesiyle oluşturulmuş hibrit bir çekirdektir.

Çok kullanılan masaüstü işletim sistemlerinin mobil versiyonları da vardır. Örneğin Windows'un mobil versiyonu Windows CE (türevlerinden biri Windows Mobile)'dır. MAC OS X'in mobil versiyonu IOS'tur. Android'e Linux çekirdeğinin mobil hale getirilmiş bir versiyonu gözüyle bakılabilir.

Maalesef her mobil ortamın doğal program geliştirme ortamı farklıdır. Windows mobil aygıtlarının doğal geliştirme ortamı .NET'tir. Android'in Java'dır. Ancak Android artık Kotlin denilen yeni bir dile geçme hazırlığındadır. IOS'un doğal programlama dili eskiden Objective-C idi. Apple artık Objective-C yerine Swift denilen yeni bir dile geçmektedir. Ancak C# ile Android ve IOS'ta geliştirme de yapılmaktadır. Bunu mümkün hale getiren ortamlardan biri olan Xamarin Microsoft tarafından satın alınmıştır.

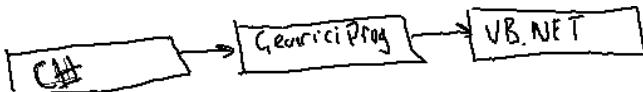
Bugün masaüstü (laptoplar da dahil olmak üzere) işletim sistemlerinde Windows %70-%80 arası bir kullanım sahiptir. Mac OS X'in kullanım oranı %10 civarındadır. Linux'un gündelik yaşamda kişisel bilgisayar olarak kullanım oranı ise %1 civarlarındadır. Ancak sunucu (server) dünyasında UNIX türevi işletim sistemlerinin payları %60'ın yukarısındadır. Yani UNIX/Linux sistemleri sunucu dünyasında en çok kullanılan sistemlerdir. 2017 yılı itibarıyla Android %64, IOS ise %32 civarında bir yaygınlığa sahiptir. Windows Mobile sistemlerinin kullanım oranı %1 civarındadır.

1.7. Gömülü Sistemler (Embedded Systems)

Asıl amacı bilgisayar olmayan fakat bilgisayar devresi içeren sistemlere genel olarak gömülü sistemler denilmektedir. Örneğin elektronik kartlar, biyomedikal aygıtlar, GPS cihazları, turnike geçiş sistemleri, müzik kutuları vs. birer gömülü sistemdir. Gömülü sistemlerde en çok kullanılan programlama dili C'dir. Ancak son yıllarda Raspberry Pi gibi, Banana Pi gibi, Orange Pi gibi güçlü ARM işlemcilerine sahip kartlar çok ucuzlamıştır ve artık gömülü sistemlerde doğrudan kullanılır hale gelmiştir. Bu kartlar tamamen bir bilgisayarın işlevselligine sahiptir. Bunlara genellikle Linux işletim sistemi ya da Android işletim sistemi yüklenir. Böylece gömülü yazılımların güçlü donanımlarda ve bir işletim sistemi altında çalışması sağlanabilmektedir. Örneğin Raspberry Pi'a biz Mono'yu yükleyerek C#'ta program yazıp onu çalıştırabiliriz.

1.8. Çevirici Programlar (Translators), Derleyiciler (Compilers) ve Yorumlayıcılar (Interpreters)

Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çeviriçi programlar (translators) denilmektedir. Çeviriçi programlarda dönüştürülmek istenen programın diline kaynak dil (source language), dönüşüm sonucunda elde edilen programın diline de hedef dil (target/destination language) denir. Örneğin:



Burada kaynak dil C#, hedef dil VB.NET'tir.

Eğer bir çeviriçi programda hedef dil aşağı seviyeli bir dil ise (saf makine dili, arakod ve sembolik makine dilleri alçak seviyeli dillerdir) böyle çeviriçi programlara derleyici (compiler) denilmektedir. Her derleyici bir çeviriçi programdır fakat her çeviriçi program bir derleyici değildir. Bir çeviriçi programa derleyici diyebilme için hedef dile bakmak gereklidir. Örneğin arakodu gerçek makine koduna dönüştüren CLR bir derleme işlemi yapmaktadır. Sembolik makine dilini saf makine diline dönüştüren program da bir derleyicidir.

Bazı programlar kaynak programı alarak hedef kod üretmeden onu o anda çalıştırırlar. Bunlara yorumlayıcı (interpreter) denilmektedir. Yorumlayıcılar birer çeviriçi program değildir. Yorumlayıcı yazmak derleyici yazmaktan daha kolaydır. Fakat programın çalışması genel olarak daha yavaş olur. Yorumlayıcılarda kaynak kodun çalıştırılması için onun başka kişilere verilmesi gereklidir. Bu da kaynak kod güvenliğini bozar.

Bazı diller yalnızca derleyicilere sahiptir (C, C++, C#, Java gibi). Bazıları yalnızca yorumlayıcılara sahiptir (PHP, Perl gibi). Bazılarının hem derleyicileri hem de yorumlayıcıları vardır (Basic, Swift, Python gibi). Genel olarak belli bir alana yönelik (domain specific) dillerde çalışma yorumlayılar yoluyla yapılmaktadır. Genel amaçlar diller daha çok derleyiciler ile derlenerek çalıştırılırlar.

1.9. Decompiler'lar ve Disassembler'lar

Alçak seviyeli dillerden yüksek seviyeli dillere dönüştürme yapan (yani derleyicilerin yaptığından tam tersini yapan) yazılımlara "decompiler" denilmektedir. Örneğin C#'ta yazılıp derlenmiş olan .exe dosyadan yeniden C# programı oluşturan bir yazılım "decompiler"dir. Saf makine dilini decompile etmek neredeyse mümkün değildir. Ancak .NET'in arakodu olan "CIL (Common Intermediate Language)" ve Java'nın ara kodu olan "Java Byte Code" kolay bir biçimde decompile edilebilmektedir. C#'ta derlenmiş bir .exe dosyası yeniden C#'a dönüştüren pek çok decompiler vardır (örneğin Salamander, Dis#, Reflector, ILSpy gibi). İşte bu tür durumlar için C# ve Java programcılar kendileri bazı önlemler almaktan zorundadırlar. Ancak C, C++ gibi doğal kod üreten derleyicilerin ürettiği kodlar geri dönüştürülememektedir.

1.10. IDE (Integrated Development Environment)

Derleyiciler komut satırından çalıştırılan programlardır. Bir programlama faaliyetinde program editör denilen bir program kullanılarak yazılır. Diske save edilir. Sonra komut satırından derleme yapılır. Bu yorucu bir faaliyettir. İşte yazılım geliştirmeyi kolaylaştıran çeşitli araçları içerisinde barındıran (integrated) özel yazılımlara IDE denilmektedir. IDE'nin editörü vardır, menüleri vardır ve çeşitli araçları vardır. IDE'lerde derleme yapılırken derlemeyi IDE yapmaz. IDE derleyiciyi çalıştırır. IDE yardımcı bir araçtır, mutlak gerekli bir araç değildir.

Microsoft'un ünlü IDE'sinin ismi "Visual Studio"dur. Apple'ın "X-Code" isimli IDE'si vardır. Bunların dışında başka şirketlerin mali olan ya da "open source" olan pek çok IDE mevcuttur. Örneğin "Eclipse" ve "Netbeans" yaygın kullanılan cross-platform "open source" IDE'lerdir. Linux altında Mono'da "Mono Develop" isimli bir IDE tercih edilmektedir. Bu IDE'nin Windows versiyonu da vardır.

VisualStudio'nun "Express Edition" ya da "Community Edition" isimli bedava bir sürümü de vardır. Bu bedava sürüm bu kurstaki gereksinimleri tamamen karşılayabilir. Visual Studio'nun bugün için son versiyonu "Visual Studio 2015"tir. Ayrıca Visual Studio'nun Mac OS X sistemleri için bir versiyonu da kullanıma girmiştir. Bu versiyon kendi içerisinde Mono ortamını kullanmaktadır. Microsoft bunu Xamarin denilen geliştirme ortamı için oluşturmuştur.

Python dünyasında da çeşitli IDE'lerden faydalılmaktadır. Bazı Python IDE'leri zaten bazı dağıtımların içerisinde onların bir parçası biçiminde bulunurlar. Bazıları ise dağıtımdan bağımsız bir biçimde yüklenerek kullanılabilmektedir.

IDLE (Integrated Development and Learning Environment) isimli minik IDE CPython dağıtımının IDE'sidir. Bu nedenle en çok kullanılan Python IDE'lerinden biridir.

Sypder isimli IDE Anaconda Python dağıtımında default olarak gelen bir Python IDE'sidir. Dolayısıyla Spyder da yaygın biçimde kullanılmaktadır. Tabii Spyder Anaconda olmadan bağımsız olarak da kurulabilmektedir.

PyCharm isimli IDE JetBrains (ünlü IntelliJ IDEA isimli Java IDE'sinin üreticisi olan firma) firmasında çalışan bir grup tarafından geliştirilmiştir. Herhangi bir dağıtıma bağlı değildir. Ancak Anaconda dağıtımının kütüphanelerini içermektedir ve Anaconda dağıtımlıyla belli bir uyumu vardır.

Eclipse'in PyDev isimli plugin'i Eclips IDE'sinin Python uygulamaları geliştirme amacıyla kullanımına olanak sağlamaktadır.

Bu arada ele alınanlar dışında daha pek çok irili ufaklı Python IDE'leri vardır. Kursumuzda IDLE, PyCharm ve Spyder IDE'leri kullanılacaktır.

1.11. Mülkiyete Sahip Yazılımlar, Özgür ve Açık Kaynak Kodlu Yazılımlar

Yazılımların çoğu bir firma tarafından ticari amaçla yazılırlar. Bunlara mülkiyete bağlı yazılım (proprietary) denilmektedir. 1980'lî yılların ortalarında Richard Stallman tarafından "Özgür Yazılım (Free Software)" hareketi başlatılmıştır. Bunu daha sonra "Open Source (Açık Kaynak Kod)" ve türevleri izlemiştir. Bunların çoğu birbirine benzer lisanslara sahiptir. Özgür yazılımin ve açık kaynak kodlu yazılımin temel prensipleri şöyledir:

- Program yazılılığında yalnızca çalıştırılabilen (executable) dosyalar değil, kaynak kodları da verilir.
- Kaynak kodları sahiplenilemez.
- Bir kişi bir kaynak kodu değiştirmiş ya da onu geliştirmiştir ise o da bunun kaynak kodlarını açmak zorundadır.
- Program istenildiği gibi dağıtılp kullanılabılır.

Linux dünyasındaki ürünlerin çoğu bu kapsamdadır. Biz bir yazılımı ya da bileşeni kullanırken onun lisansına dikkat etmeliyiz. Bugün özgür yazılım ve açık kaynak kod hareketi çok ilerlemiştir. Neredeyse popüler pek çok ürünün açık kaynak kodlu bir versiyonu vardır.

1.12. Bit ve Byte Kavramları

Bilgisayarlar ikilik sistemi kullanırlar. Bu nedenle bilgisayarların belleğinde, diskinde vs. her şey ikilik sistemde sayılar biçiminde bulunmaktadır. İkilik sistemde sayıları yazarken yalnızca 1'ler ve 0'lar kullanılır. Böylece bilgisayarın içerisinde yalnızca 1'ler ve 0'lar vardır. Her şey 1'lerden ve 0'lardan oluşmaktadır. İkilik sistemdeki her bir basamağa bit (binary digit'ten kısaltma) denilmektedir. Bu durumda en küçük bellek birimi bit'tir. Bit çok küçük olduğu için 8 bit'e 1 byte denilmiştir. Bellek birimi olarak byte kullanılır. Bilgisayar bilimlerinde Kilo 1024 katı anlamına gelir. Yani 1KB = 1024 byte'tır. Mega da kilonun 1024 katıdır. Yani 1MB=1024KB'tır. Giga Mega'nın Tera da Giga'nın 1024 katıdır.

1.13. Doğal Kodlu Çalışma, Arakodlu Çalışma ve JIT Derlemesi

Kullandığımız CPU'lar ikilik sistemdeki makine komutlarını çalıştırmaktadır. Bir kodun CPU tarafından çalıştırılabilmesi için o kodun o CPU'nun makine diline dönüştürülmeli olması gereklidir. Zaten derleyiciler de bunu yapmaktadır. Eğer bir çeviriçi program (yani derleyici) o anda çalışılmakta olan makinenin CPU'sunun işletebileceği makine kodlarını üretiyor CPU da bunları çalıştırıysa buna doğal kodlu (native code) çalışma denilmektedir. Örneğin C ve C++ programlama dillerinde doğal kodlu çalışma uygulanmaktadır. Biz bu dillerde bir programı yazıp derlediğimizde artık o derlenmiş program ilgili CPU tarafından çalıştırılabilecek doğal kodları içermektedir.

Bazı sistemlerde derleyiciler doğrudan doğal kod üretmek yerine hiçbir CPU'nun makine dili olmayan (dolayısıyla hiçbir CPU tarafından işletilemeyecek) yapay bir kod üretmektedir. Bu yapay kodlara genel olarak "ara kodlar (intermediate codes)" denilmektedir. Bu arakodlar doğrudan CPU tarafından çalıştırılamazlar. Arakodlu çalışma Java ve .NET dünyasında ve daha başka ortamlarda kullanılmaktadır. Java dünyasında Java derleyicilerinin üretikleri ara koda "Java Bytecode", .NET (CLI) dünyasında ise "CIL (Common Intermediate Language)" denilmektedir. Pekiyi bu arakodlar ne işe yaramaktadır? İşte bu arakodlar çalıştırılmak istendiğinde ilgili ortamın alt sistemleri devreye girerek önce bu arakodları o anda çalışılan CPU'nun doğal makine diline dönüştürüp sonra çalıştırmaktadır. Bu süreçte (yani arakodun doğal makine koduna dönüştürülmesi sürecine) tam zamanında derleme (just in time compilation) ya da kısaca "JIT derlemesi" denilmektedir. Java ortamında bu JIT derlemesi yapıp programı çalıştan alt sisteme "Java Sanal Makinesi (Java Virtual Machine)", .NET ortamında ise CLR (Common Language Runtime)" denilmektedir.

Süphesiz doğal kodlu çalışma arakodla çalışmadan daha hızlıdır. Pek çok benchmark testleri arasındaki hız farkının %20 civarında olduğunu göstermektedir. Pekiyi arakodlu çalışmanın avantajları nelerdir? İşte bu çalışma biçimini derlenmiş kodun platform bağımsız olmasını sağlamaktadır. Buna "binary portability" de denilmektedir. Böylece arakodlar başka bir CPU'nun ya da işletim sisteminin bulunduğu bir bilgisayara götürüldüğünde eğer orada ilgili ortam (framework) kuruluysa doğrudan çalıştırılabilmektedir.

Python dünyasında da bazı Python gerçekleştirimlerinde gizli bir arakodlu çalışma vardır. Bu nedenle bazı Python dil işlemcisinin bir derleyici yoksa yorumlayıcı mı olduğu tartışılabilir. Örneğin CPython gerçekleştiriminde yorumlayıcı Python kodunu okuyup onu o anda çalıştırırmak yerine önce bir arakod üretip o arakodu çalıştırmaktadır.



Ancak CPython bir JIT derlemesi yapmamaktadır. Çünkü CPython dönüştürüdüğü arakodu tane tane alarak o anda yorumlama sistemiyle çalışmaktadır. Oysa Python'in PyPy gerçekleştirmi tıpkı Java ve .NET dünyasına benzer biçimde bir JIT derlemesi işlemiyle kodu çalıştırır. Dolayısıyla PyPy gerçekleştirmi çalışma zamanı (run time) bakımından CPython gerçekleştirimine göre daha iyidir. Aslında CPython tüm Python kodunu değil Python modüllerini arakodlara dönüştürmektedir. Ana script kodlarını arakodlara dönüştürmemektedir. CPython gerçekleştiriminin arakodlu bir çalışma sağlamasının temel nedeni aynı programın ikinci kez çalıştırılması sırasında daha hızlı çalıştırılmasını sağlamaktır.

Burada önemli bir nokta şudur: Python genelinde bir arakod standardı yoktur. Halbuki Java ve .NET (CLI) dünyalarında onların arakodları oldukça sağlam biçimde standardize edilmiştir. Fakat belli bir gerçekleştirim (örneğin CPython) ele alındığında onun farklı platformlardaki arakodlarının aynı olduğunu söyleyebiliriz.

1.14. Dil Nedir?

Dil karmaşık bir olgudur. Tek bir cümleyle tanımını yapmak pek mümkün değildir. Fakat kısaca “iletişim için kullanılan semboller kümesidir” denebilir. Bir dilin tüm kurallarına gramer denir. Gramerin en önemli iki alt alanı sentaks (syntax) ve semantik (semantic)'tir. Bir dili oluşturan en yalın öğelere atom ya da sembol (token) denilmektedir. Örneğin doğal dillerdeki atomlar sözcüklerdir.

Bir olgunun dil olabilmesi için en azından sentaks ve semantik kurallara sahip olması gereklidir. Sentaks doğru yazma ve dizilime ilişkin kurallardır. Örneğin:

“I school to am going”

Burada İngilizce için bir sentaks hatası söz konusudur. Sözcükler doğrudur fakat dizimleri yanlışdır. Örneğin:

“Herkez çok mutluydu”

Burada da bir sentaks hatası vardır. Türkçe'de “herkez” biçiminde bir sözcük (yani sembol) yoktur. Örneğin:

if a > 10)

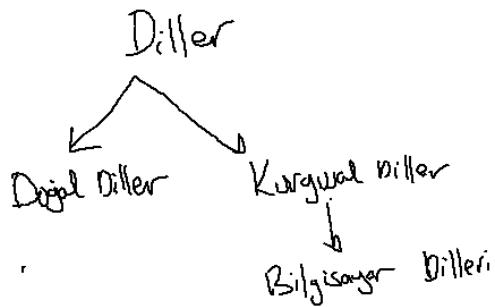
Burada da C#'ça bir sentaks hatası yapılmıştır.

Semantik doğru yazılmış ve dizilmiş olan öğelerin ne anlam ifade ettiğine ilişkin kurallardır. Yani bir şey doğru yazılmıştır fakat ne anlama gelmektedir? Örneğin:

" I am going to school"

sentaks bakımından geçerlidir. Fakat burada ne denmek istenmiştir? Bu kurallara semantik denilmektedir.

Diller doğal ve kurgusal (ya da yapay) olmak üzere ikiye ayrılabilir. Doğal dillerde sentaksın tam bir formülasyonu yoktur. Kurgusal diller insanlar tarafından formüle edilebilecek biçimde tasarlanmış dillerdir. Bilgisayar dilleri kurgusal dilleridir.



Kurgusal dillerde istisnalar ya yoktur ya da çok azdır. Sentaks ve semantik tutarlıdır. Doğal dillerde pek çok istisna vardır. Doğal dillerin zor öğrenilmesinin en önemli nedenlerinden birisi de istisnalardır.

1.14.1. Bilgisayar Dilleri ve Programlama Dilleri

Bilgisayar bilimlerinde kullanılan dillere bilgisayar dilleri (computer languages) denir. Bir bilgisayar dilinde akış varsa ona aynı zamanda programlama dili de (programming language) denilmektedir. Örneğin HTML bir bilgisayar dilidir. Fakat HTML'de bir akış yoktur. Bu nedenle HTML bir programlama dili değildir. HTML'de de sentaks ve semantik kurallar vardır. Oysa C#'ta bir akış da vardır. C# bir programlama dilidir.

1.14.1.1. Programlama Dillerinin Sınıflandırılması

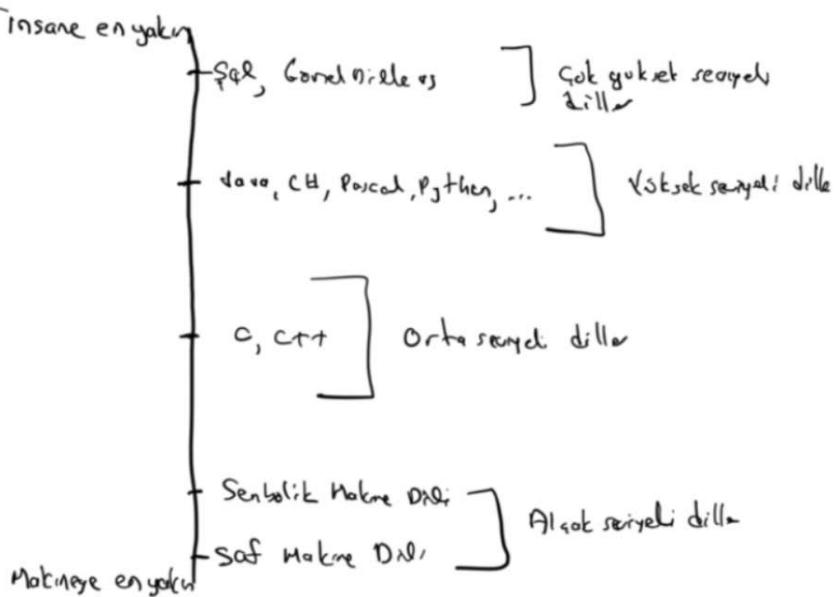
Programlama dilleri üç biçimde sınıflandırılabilir:

- 1) Seviyelerine Göre Sınıflandırma
- 2) Kullanım Alanlarına Göre Sınıflandırma
- 3) Programlama Modeline Göre Sınıflandırma

Seviyelerine Göre Sınıflandırma: Seviye (level) bir programlama dilinin insan algısına yakınlığının bir ölçüsüdür. Yüksek seviyeli diller kolay öğrenilebilen insana yakın dillerdir. Alçak seviyeli diller bilgisayara yakın dillerdir. Olabilecek en alçak seviyeli diller 1'lardan oluşan saf makine dilleridir. Bunun biraz yukarısında sembolik makine dilleri (assembly languages) bulunur. Biraz daha yukarıda orta seviyeli diller bulunmaktadır. Daha yukarıda ise yüksek seviyeli, en yukarıda da “çok yüksek seviyeli diller” vardır. Örneğin:

- Java, C#, Pascal, Basic “yüksek seviyeli” dillerdir.
- C “orta seviyeli” bir dildir.
- C++ orta ile yüksek seviye arasındadır.

Tabii aslında dillerin seviyelerini sürekli çizgi üzerinde noktalar biçiminde düşünürebiliriz. İki yüksek seviyeli dil bu çizgide aynı yerde bulunmak zorunda değildir. Seviyelerine göre dilleri sınıflandırırken genellikle bir seviye çizgisinden faydalananır:



Aslında aynı kategorideki diller arasında da bir seviye farkı vardır. Yani örneğin iki yüksek seviyeli programlama dili aslında aynı seviyede olmayabilir.

Kullanım Alanlarına Göre Sınıflandırma: Bu sınıflandırma biçimi dillerin hangi amaçla daha çok kullanıldığına yönelikir. Tipik sınıflandırma şöyle yapılabilir:

- Bilimsel ve Mühendislik Diller: C, C++, Java, C#, Fortran, Python, Pascal, Matlab gibi...
- Veritabanı Yoğun İşlemlerde Kullanılan Diller: SQL, Foxpro, Clipper, gibi...
- Web Dilleri: PHP, C#, Java, Python, Perl gibi...
- Animasyon Dilleri: Action Script gibi...
- Yapay Zeka Dilleri: Lisp, Prolog, Python, C, C++, C#, Java gibi...
- Sistem Programlama Dilleri: C, C++, Sembolik Makina Dilleri
- Genel Amaçlı Diller: C, C++, Pascal, C#, Java, Python, Basic gibi...

Programlama Modeline Göre Sınıflandırma: Program yazarken hangi modeli (paradigm) kullandığımıza yönelik sınıflandırmadır. Altprogramların birbirlerini çağırmasıyla program yazma modeline “prosedürel programlama modeli (procedural programming paradigm)” denilmektedir. Bir dilde yalnızca alt programlar oluşturabiliyorsak, sınıflar oluşturamıyorsak bu dil için “prosedürel programlama modeline uygun olarak tasarlanmış” bir dil diyebiliriz. Örneğin klasik Basic, Fortran, C, Pascal prosedürel dillerdir. Bir dilde sınıflar varsa ve program sınıflar kullanılarak yazılıyorsa böyle dillere “nesne yönelimli diller (object oriented languages)” denilmektedir. Eğer program formül yazar gibi yazılıyorsa bu modelede fonksiyonel model (functional paradigm), bu modeli destekleyen dillere de fonksiyonel diller denilmektedir. Bazı dillerde program görsel olarak fare hareketleriyle oluşturulabilmektedir. Bunlara görsel diller denir. Bazı diller birden fazla programlama modelinin kullanılmasına olanak sağlayacak biçimde tasarlanmıştır. Bunlara da çok modelli diller (multiparadigm languages) denilmektedir. Örneğin C++ gibi. C#'a son yıllarda Microsoft tarafından eklenen bazı özellikler ona belli oranda fonksiyonel programlama yeteneği de kazandırmıştır. Bu durumda C# için de belki çok modelligidir denilebilir. Yine Apple’ın yeni tasarladığı Swift dili de çok modelligidir.

Tüm bunlar ışığında Python için şunlar söylenebilir: Python yüksek seviyeli, prosedürel, nesne yönelimli, fonksiyonel, genel amaçlı, üretken, öğrenilmesi nispeten daha basit, veri analizinde ve yapay zeka alanlarında özellikle tercih edilen bir dildir. Geniş bir standart kütüphanesi vardır. Az kodlamaya önemli işler yapılmaktadır. Örneğin Java ve C#'a göre 2 ila 5 kat arasında kod uzunluğu bakımından fark vardır. Bu fark C ve C++ için 10 kata yakındır. Yorumlayıcı temelli bir dildir. Çalışma performansı derleyici tabanlı dillerden daha yavaş olma eğilimindedir. Eğitimde programlamaya giriş amacıyla ilk öğretilen dillerden biri haline gelmiştir.

1.15. Klavyedeki Karakterlerin İngilizce İsimleri

Sembol	İsim
--------	------

+	plus
-	minus, hyphen, dash
*	asterisk
/	slash
\	back slash
.	period, dot
,	comma
:	colon [ko:lın]
;	semicolon
"	double quote [dabil kvot]
'	single quote
(...)	paranthesis [piran(th)isi:s] left, right, opening, closing
[...]	(square) bracket left, right, opening, closing
{...}	brace [breys] left, right, opening, closing
=	equal sign [i:kvıl sayn]
&	ampersand
~	tilda
@	at
<...>	less than, greater than, angular bracket
^	caret [k(ea)rıt]
	pipe [payp]
_	underscore [andırsko:r]
?	question mark
#	sharp, number sign, hashtag
%	percent sign [pörsint sayn]
!	exclamation mark [eksklemeşin mark]
\$	dollar sign [dalır sayn]
...	ellipsis [elipsis]

1.16. Sentaks Gösterimleri

Programlama dillerinin sentaklarını betimlemek için pek çok meta dil oluşturulmuştur. Bunlardan en ünlüsü ve çok kullanılanı BNF (Backus-Naur Form) denilen notasyondur. Gerçekten de bugün programlama dillerinin resmi standartlarında ya da referanslarında hep BNF notasyonu ya da bunun türevleri kullanılmaktadır. Python'ın orijinal referans kitabında da bu notasyon tercih edilmiştir. BNF notasyonu ISO tarafından da standardize edilmiştir. İsmine EBNF denilmektedir. Ancak BNF notasyonun anlaşılması biraz çalışma istemektedir. Derneğimizde "Sistem Programlama ve İleri C Uygulamaları - II" kursunda ele alınmaktadır.

Kursumuzda sentas açıklamak için BNF yerine aşsal parantez köşeli parantez teknigi kullanılacaktır. Bu teknikte köşeli parantez içerisindekiler istege bağlı (optional) öğeleri aşsal parantez içerisindekiler zorunlu öğeleri belirtmektedir. Diğer tüm atomlar aynı pozisyonda bulunmak zorundadır. Örneğin C dilindeki if deyimi bu notasyonla şöyle belirtilir:

```
if (<ifade>
    <deyim>
[<deyim>
]else <deyim>
```

Ayrıca biz bu teknikte anahtar sözcüklerin altını da çizeceğiz.

2. Python Programlama Diline Giriş

Bu bölümde Python Programlama Dili hakkında temel bilgiler verilecektir.

2.1. Komut Yorumlayıcı Oramda Çalışma ve IDLE Ortamı

Python bir programlama dili olmasına karşın pek çok gerçekleştirm ve dağıtım bize Python ifadelerini bir komut satırında yazma olanağı da vermektedir. Python tarafından sunulan bu komut yorumlayıcı ortamda birtakım kod parçalarının test edilmesinde sıkılıkla kullanılmaktadır. Biz de kursumuzda sık sık bu komut yorumlayıcı ortamı kullanacağiz. Aslında Python programları bu komut yorumlayıcı ortamda yazılan deyimlerin peşi sıra hızlı bir biçimde çalıştırılması ile çalıştırılmaktadır. Örneğin python yorumlayıcısı komut satırında hiçbir komut satırı argümanı kullanılmadan çalıştırılırsa karşımıza bir komut yorumlayıcı ortam çıkar. Örneğin:

```
C:\>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Aynı komut satırı CPython dağıtımının IDLE IDE'sinde de bulunmaktadır. Kullanıcı komut satırında tekil komutlar girerek işlemleri yapabilir. IDLE'in menüleri kullanılarak Python script'leri ve programları yazılıp çalıştırılabilir.



IDLE tam ekranlı (full screen) bir editöre de benzemektedir. İmleç ile bir satır gelinip ENTER tuşuna basılırsa o satırın kopyası imlece alınır. Windows ortamında Alt + P ile önceki komutlar, Alt + N ile sonraki komutlar arasında gezinti yapılmaktadır.

Python dinamik tür sistemine (dynamic type system) sahip bir dildir. Dinamik tür sistemine sahip dillerde değişkenlerin türleri aksın hangi noktada olduğuna bağlı olarak değişebilir. Başka bir deyişle bu tür dillerde biz aynı değişkene istediğimiz türden bilgileri atayabiliriz. O anda değişkene hangi türden bir bilgi atamışsa onun türü o anda atadığımız

değerin türünden olur. Yine başka bir deyişle bu tür dillerde aslında değişkenlerin değil o değişkenlerin tuttuğu ya da referans ettiği nesnelerin türleri vardır. Bunun tersine "statik tür sistemi (static type system)" denilmektedir. Yaygın pek çok dil statik tür sistemine sahiptir. Örneğin C, C++, Java, C#, Pascal gibi. Bu dillerde bir değişken önce bildirilir. Bildirim sırasında onun türü de belirtilir. Sonra o değişken faaliyet alanı boyunca hep o türden olur. Türü bir daha değiştirilemez.

Atom (Token) Kavramı

Programlama dillerinde kendi başına anlamlı olan daha fazla parçaya ayrılamayan en yalın sentaktik elemana atom denilmektedir. Örneğin:

```
a + b * 10
```

Buradaki atomlar şunlardır:

```
a  
+  
b  
*  
10
```

Atomlar Python'da beş gruba ayrılmaktadır:

1) Anahtar Sözcükler (Keywords): Dil için özel anlamı olan değişken olarak kullanılması yasaklanmış sözcüklerdir. Örneğin if, while, for gibi atomlar birer anahtar sözcüktür.

2) Değişkenler (Identifiers): İsmini bizim (ya da başka bir programcının) belirli kurallar dahilinde istediği gibi verebildiği atomlardır. Örneğin a, foo, print gibi.

3) Sabitler (Literals): Doğrudan yazdığımız sayılara ve yazınlara sabit denilmektedir. Örneğin:

```
a + 123
```

İfadesinde a bir değişken, + bir operatör ve 123 de bir sabittir. Bir değer ya bir değişkenin içerisinde bulunur ya da doğrudan yazılmıştır. Doğrudan yazılmış değerlere sabit denir. Python'da aynı zamanda iki tırnak ya da tek tırnak içerisindeki yazılar da birer sabit biçiminde ele alınmaktadır. Örneğin 'Ankara' yazısı da bir sabittir.

4) Operatörler (Operators): Bir işleme yol açan işlem sonucunda belli bir değerin üretilmesini sağlayan atomlara operatör denir. Örneğin:

```
a + 10
```

Burada + bir operatördür. Tipik opertaörler bazı özel sembollerden oluşturulmuştur. Örneğin +, -, * gibi. Ancak anahtar sözcükler de işlev olarak operatör görevinde olabilirler. Örneğin:

```
a is b
```

Burada is hem bir anahtar sözcüktür hem de bir operatördür. İşte hem bir anahtar sözcük hem de operatör olan atomlar operatör olarak ele alınmaktadır.

5) Ayıraçlar (Delimiters): Yukarıdaki grupların dışında kalan tüm atomlar ayıraç görevindedir. Örneğin:

```
if a > 10:  
    b = 100
```

Buradaki ':' karakteri ayıraç bir atomdur. Ya da örneğin:

```
a = 10; b = 20
```

Buradaki ';' de ayıraç bir atomdur.

İfade (Expression) Kavramı

Programlama dillerinde değişkenlerin, operatörlerin ve sabitlerin her bir kombinasyonuna ifade denilmektedir. Örneğin:

```
a  
a + 1  
(a + 1) * 2  
10
```

Tek başına bir değişken ve sabit bir ifade belirtir. Ancak tek başına bir operatör ifade belirtmez.

Boşluk Karakterleri (White Space)

Programlama dillerinde boşluk duygusu oluşturmak için kullanılan karakterlere boşluk karakterleri (white space) denilmektedir. İngilizce "space" karakteri denildiğinde ASCII ve UNICODE 32 numaralı boşluk karakteri anlaşıılır. Ancak "white space" denildiğinde tüm boşluk karakterleri anlaşılmaktadır. Tipik boşluk karakterleri şunlardır:

```
SPACE  
TAB  
VTAB  
LF (ya da CR/LF)
```

Windows'ta ENTER tuşuna bastığımızda aslında iki karakter kodu dosyaya dahil edilmektedir: CR ve LF. Ancak UNIX/Linux ve MAC OS X sistemlerinde yalnızca LF dosyaya dahil edilmektedir. Windows'ta yalnızca CR karakteri "bulunulan satırın başına geçmek için", yalnızca LF karakteri "aşağı satırın bulunulan sütununa geçmek için" kullanılmaktadır. Dolayısıyla Windows'ta "aşağı satırın başına geçmek için" yalnızca CR ya da yalnızca LF karakterleri yeterli değildir. Bunların CR/LF biçiminde bir arada bulunması gereklidir. Halbuki UNIX/Linux ve MAC OS X sistemlerinde tek başına LF karakteri bu işi yapmaktadır.

TAB Karakteri

Klavizeden TAB tuşuna basıldığında elde edilen karaktere TAB karakter denilmektedir. TAB ASCII ve UNICODE tabloda 9 numada olan tek bir karakterdir. TAB karakter aslında yatay biçimde belirli bir miktar zıplamak için kullanılır. Ancak TAB karakterin ne kadar zıplaya yol açacağı konusunda bir belirleme yapılmamıştır. Bu tamamen TAB karakteri ele alan ortamın (tipik editörün) isteğine bağlıdır. Biz TAB karakter görüldüğünde ne kadar zıplanacağını editörün ayarlarından değiştirebiliriz. Programcılar en çok kullandığı TAB aralığı 4'tür. Ancak bazı ortamlar ve editörler TAB karakteri görüldüğünde bulunulan yerden itibaren n karakter kadar zıplamak yerine ilk "tab durağına (tab stop)" gidebilmektedir. Aslında bilgisayar klavyesi daktilo temel alınarak geliştirilmiştir. Tab durakları daktiolarda kullanılmaktadır.

Programlama editörlerinde genellikle isteğe bağlı biçimde TAB karakter yerine n tane SPACE karakterinin yer değiştirmesi sağlanabilmektedir. Yani programcı isterse editör seçeneklerinden TAB tuluna basıldığında dosyaya tek bir TAB karakter yerine örneğin 4 tane SPACE karakterinin basılmasını sağlayabilmektedir. Bunun bir avantajı farklı editörlerde aynı görüntünün elde edilmesidir. Dezavantaj ise kaynak programın dosyada daha fazla yer kaplamasıdır. Örneğin IDLE IDE'sinde TAB yerine her zaman programcı tarafından ayarlanan bir miktarda (default'u 4) SPACE karakteri basılmaktadır. Python programlamasında TAB karakterin belli bir miktarda SPACE karakteri ile değiştirilmesi daha çok tercih edilmektedir.

Python Kodlarının Yazım Biçimi

Python'da ifadeleri ayırmak için temel olarak LF (Windows'ta CR/LF) karakteri kullanılmaktadır. Bu durumda her satıra tek bir ifadenin yazılması gereklidir. Halbuki diğer bazı dillerde ifadeleri sonlandırmak için ';' gibi ayıraç atomlarından faydalananmaktadır. Python'da bir ifade -özel durumlar haricinde- C, C++, Java ve C#'ta olduğu gibi farklı satırlara bölünmemektedir. Örneğin:

```
a  
=  
10
```

İfadesi Python'da yazım bakımından geçerli değildir. Yine Python'da istenirse ya da gerektiğiinde ifadeleri ayırmak için ';' atomu kullanılabilmektedir. Gerçekten de Python'da basit deyimler (simple statements) aralarına ';' ayıracı konularak aynı satır üzerine yazılabilmektedir. Örneğin:

```
a = 10; b = 20; c = 30;
```

Brada son ';' atomu konulmasa da olurdu. Çünkü zaten satır sonu (yani LF karakteri) bir ayıraç olarak işlev görmektedir. Ancak Python'da her deyim tek satır üzerine yazılmaz. Yalnızca basit deyimler bir arada tek satır üzerine yazılabilir. if, for, while gibi kontrol deyimleri tek satır üzerine yazılamayacağı gibi tek satırda diğerleri ile birlikte de yazılamaz. Bu konunun ayrıntıları deyimlerin anlatıldığı bölümde ele alınacaktır.

Python'da satır sonu dışında atomlar arasında istenildiği kadar SPACE ve TAB karakter kullanılabilmektedir. Örneğin:

```
c = a + b
```

İfadesi geçerlidir. Aynı zamanda değişkenlerle anahtar sözcükler dışında atomlar istenildiği kadar bitişik de yazılabılır. Örneğin:

```
a=a+b
```

Yazımı da geçerlidir. Burada anlatılanları söyle özetleyebiliriz:

- Python'da basit deyimler ya farklı satırlar üzerinde ya da aynı satırda ';' ayırcı kullanılarak bir arada yazılabılır.
- Kontrol deyimleri (genel olarak bileşik deyimler) için farklı bir yazım kuralı vardır. Bu konu ileride ele alınacaktır.
- Basit deyimlerde atomlar arasında istenildiği kadar SPACE ve TAB karakteri bulundurulabilir ve bunlar istenildiği kadar bitişik yazılabılır.

Python'da ifadelerin ve deyimlerin genel sentaks biçimleri konusunda bazı ayrıntılar vardır. Bu ayrıntılara ilgili bölümlerde değinilecektir.

Python'da Temel Veri Türleri

Tür (type) bir nesnenin bellekte kapladığı alan, ona uygulanacak işlemler (operatörler) ve ona yerleştirilecek değerlerin biçimi hakkında bilgi veren temel bir özelliktir. Her dilde nesnelerin türleri vardır. Python'daki temel türler aşağıdaki tabloda listelenmiştir:

Tür İsmi	Özellik
int	İşaretli tamsayı türü. Python'da int türünün bir sınırı yoktur.
float	IEEE 754 Standardının 8 byte'lık gerçek sayı formatını (long real format) belirtmektedir.
bool	Bu tür True ya da False biçiminde belirtilen ikil bilgileri tutmaktadır.
str	Yazılırı tutan string türüdür. Yazılıar tek tırnakla ya da iki tırnakla belirtilebilirler.

NoneType	Python'da özel bir NoneType türü vardır. Bu tür None anahtar sözcüğü ile temsil edilmektedir. None boş değer anlamına gelir.
complex	Python'da karmaşık sayı türü de vardır. i sayısı j ile temsil edilmektedir.

- Python'da int türünün bir sınırı yoktur. Dolayısıyla int türünden bir nesnenin kaplayacağı alan da onun içeresine yerleştirilmiş olan değere bağlı olarak değişebilir. Aslında işlemciler çok büyük tamsayılar üzerinde işlem yapamazlar. Bu yüzden Python yorumlayıcıları büyük sayılar üzerinde işlemleri algoritmik olarak (yani onları parçalara ayırarak) yaparlar. Fakat programcı Python yorumlayıcısının büyük tamsayılarla nasıl işlem yaptığını bilmek zorunda değildir. Programcı bu işlemlerin yorumlayıcı tarafından etkin bir biçimde yapıldığını varsaymalıdır.

- Python'da float türü C, C++, Java ve C#'taki double türünün eşdeğерidir. float türü noktalı sayıları tutabilen bir türdür. float türü kayan noktalı bir formata sahip olduğu için yuvarlama hatalarına (rounding errors) yol açabilmektedir. float türü $[+/-1.8 * 10^{308}, +1.8 * 10^{-308}]$ aralığında sayıları belli bir mantis aralığında tutabilmektedir. Yuvarlama hatası "bir noktalı sayının tam olarak tutulamayıp ona yakın bir sayının tutulmasıyla oluşan hatalara" denilmektedir. Yuvarlama hatası noktalı sayıların kayan noktalı formatla tutulmasından kaynaklanmaktadır ve bunu yok etmenin pratik bir yolu yoktur. Yuvarlama hatası bazen noktalı sayının ilk kez bir değişkene atanması sırasında oluşurken bazen de aritmetik işlemlerin sonucunda oluşabilmektedir. Örneğin:

```
>>> 0.3 - 0.2
0.0999999999999998
>>> 0.3 - 0.2 == 1
False
```

- bool türü yalnızca True ve False değerlerini tutar. Ancak Python'da ileride ele alınacağı gibi tipki C'de olduğu gibi nümerik türlerden bool türüne otomatik dönüştürme vardır.

- Python'da str türü string türünü belirtmektedir. String'ler yazı tutan türlerdir. Yazılıar tek tırnak ya da eşdeğer olarak çift tırnak ile belirtilebilirler. Python'da ayrıca pek çok dilde var olan "char" biçiminde bir tür yoktur. Karakterler tek karakterli string'ler gibi ele alınmaktadır. Yani başka bir deyişle Python'da bir karakter aslında tek karakterli bir string'tir.

- Python'da None anahtar sözcüğü ile temsil edilen özel bir None türü vardır. None anahtar sözcüğü her değişkene atanabilir. Bu durumda o değişkenin içerisinde boş bir değerin olduğu (başka bir deyişle bir değerin olmadığı) düşünülmelidir. Biz bir değişkenin içerisinde None değerinin olup olmadığını test edebiliriz. None değeri genellikle fonksiyonlarda başarısızlığı anlatmak için kullanılmaktadır. Bir değişkene henüz hiçbir değer atanmamışsa aslında o değişken henüz yaratılmıştır. Ancak None değeri atanmışsa o değişken yaratılmıştır. Fakat içerisinde bir şey yoktur.

- Python'da karmaşık (complex) sayılar j harfiyle temsil edilmektedir. Ancak j alfabetik bir karakter olduğu için j karakterine yapışık bir sayının bulundurulması gereklidir. Örneğin:

```
>>> 1j
1j
>>> 2j + 1
(1+2j)
>>> -1j * 2
-2j
```

gibi. İçerisinde j geçen sayılar complex türündendir. Bu complex türünün real ve imag isimli örnek öznitelikleri sayının gerçek ve sanal kısımlarını bize vermektedir. Yani z complex türünden bir değişken olmak üzere bu nesnenin gerçek kısmına z.real, sanal kısmına da z.imag ifadesi ile erişilir. Örneğin:

```
>>> z = 3 + 2j
>>> type(z)
<class 'complex'>
>>> z.real
```

```
3.0
>>> z.imag
2.0
```

complex türünün real ve imag elemanları float türündendir.

Python'da yukarıda açıkladığımız temel türler aslında birer sınıf (class) belirtmektedir. Yani aslında int, float, bool, str, NoneType birer sınıfıtır. Sınıflar kursumuzda ilerideki bölümlerde ele alınacaktır.

Python'da Değişkenlerin Yaratılması

Python dinamik tür sistemine sahip olduğu için Python'da C, C++, Java, C# gibi statik tür sistemine sahip dillerdeki gibi bir bildirim işlemi yoktur. Bir değişken o değişene ilk kez değer atandığında yaratılmış olur. Yani örneğin x değişkenine henüz bir atanmadıysa x değişkeni yaratılmamıştır. x değişkenine ilk kez değer atandığında x değişkeni yaratılır. Artık x değişkeni ona atanın değer türünden olur. Sonra x değişkenine başka türden bir değer atanırsa artık x en son atanın değerin türünden olur.

Python'da henüz yaratılmamış bir değişkenin içerisindeki değerin kullanılacağı bir ifadede kullanılması error oluşturur. Yani biz henüz yaratılmamış bir değişkeni ancak ona atama yapacak biçimde kullanabiliriz. Örneğin k ve z değişkenleri henüz yaratılmamış (yani içerisinde bir değer atanmamış) olsun:

```
>>> z = k + 2
```

Bu işlemde hataya yol açan şey z'ye atama yapılması değildir. Henüz yaratılmamış olan k'nın kullanılmasıdır. Bu işlemi yaparsak aşağıdaki gibi bir hata mesajıyla karşılaşabiliriz:

```
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    z = k + 2
NameError: name 'k' is not defined
```

Python büyük harf-küçük harf duyarlılığı olan (case sensitive) bir dildir. Yani büyük harfler ve küçük harfler tamamen farklı karakterler gibi değerlendirilmektedir. Dolayısıyla örneğin count değişkeni ile Count değişkeni aynı değişkenler değildir.

Python'da değişken isimlendirme kuralları diğer pek çok dildeki gibidir:

- Değişken isimleri boşluk karakterlerini ve operatör sembollerini içermemelidir.
- Anahtar sözcüklerden değişken ismi yapılamaz.
- Değişken isimleri sayısal karakterlerle başlatılmalıdır ancak alfabetik karakterlerle başlatılıp sayısal karakterlerle devam ettirilebilir._ karakteri alfabetik bir karaktermiş gibi ele alınmaktadır.

Ayrıca Python kaynak program için default olarak UTF8 UNICODE kodlama biçimini (encoding) kabul etmektedir. Python'da değişken isimlendirmelerinde başka dillerin (örneğin Türkçe'nin) kendi karakterleri kullanılabilir. Kursumuzda biz değişken isimlendirmelerinde Türkçe sözcükleri değil İngilizce sözcükleri tercih edeceğiz.

Python'da Sabitler (Literals)

Doğrudan yazılmış olan sayılar ve yazılar sabit (literal) denilmektedir. Örneğin a + 123 gibi bir ifadede a bir değişken, 123 ise bir sabittir. Sabitlerin de türleri vardır. Sabitlerin türlerini belirlemek önemlidir. Çünkü bir sabit bir değişkene atandığında o değişkenin türü o sabitin türünden olur. Şimdi sabit türlerini görelim:

1) Nokta içermeyen tamsayı biçiminde yazılmış sabitler int türündendir. Örneğin:

```
123
0
```

birer int türden sabittir. Python'da int türünün bir sınırı olmadığına göre int türden sabitler de istenildiği kadar uzun olabilirler.

int türden sabitler Python'da 10'luk, 16'luk, 8'luk ve 2'luk sistemde belirtilebilmektedir. Default sistem 10'luk sistemdir. Bir int sabiti 16'luk sistemde yazmak için sabit 0x ya da 0X öneki başlatılır. Örneğin:

```
0xA2F
0x123
```

Benzer biçimde 8'luk sistem için 0o ya da 0O, 2'luk sistem için de 0b ya da 0B önekleri kullanılmaktadır. Örneğin:

```
0B1010
0O123
```

Python'da ayrıca tamsayıların başında 0 bulunamaz. Eğer tamsayıların başında 0 varsa bu sıfırı o, b ya da x izlemek zorundadır.

Ayrıca Python'da tamsayıların basamakları arasında istenildiği kadar _ karakteri kullanılabilir. Bu _ karakterleri aslında yorumlayıcı tarafından dikkate alınmamaktadır. Örneğin:

```
123_000
1_216_456
1_2_3_4_6
0x123_454
```

Bu özellik büyük sayıları binler hanelerinden grüplamak için genellikle kullanılmaktadır. Ancak sayı içerisinde yan yana birden fazla _ karakteri bulunamamaktadır. Örneğin:

```
>>> x = 1__000
SyntaxError: invalid token
```

2) Sayı '.' içeriyorsa (tabii bir tane '.' içerebilir) sabit float türündendir. Örneğin:

```
12.34
12.0
```

'.' karakterinin solu ya da sağı boş bırakılabilir. Bu durumda boş bırakılan yerde 0 olduğu kabul edilir. Örneğin:

```
12.
.234
```

sayıları da float türden sabit belirtir.

float sabitler pek çok programlama dilinde olduğu gibi Python'da da üstel biçimde belirtilebilmektedir. Üstel biçimde önce sayının üstel olmayan kısmı yazılır. Sonra bunu 'e' ya da 'E' karakteri izler. Bu karakter on üzeri anlamına gelmektedir. Sonra bunu da üssü belirten pozitif ya da negatif ya da sıfır değeri izler. Örnek:

```
1.23e12
1e24
1.17e-21
```

Burada sayının üstel olmayan kısmı nokta içermese bile sabit yine üstel yazımından dolayı float türündendir.

Yine float türünde de sayıların arasında istenildiği kadar _ karakterleri kullanılabilir. Örneğin:

123_23.45_56
1.23_34e23
1.23E1_2

3) bool türden sabitler iki tanedir. Bunlar True ve False anahtar sözcükleridir. True ve False anahtar sözcükleri int ve float türlerle aritmektik işlemlere sokulabilirler. Bu durumda True değeri 1 olarak False değeri 0 olarak işleme girer. Örneğin:

```
>>> True + 3  
4
```

Örneğin:

```
>>> False * 5  
0
```

4) String türünden sabitler tek tırnak ya da çift tırnak ile belirtilebilmektedir. Tek tırnakla çift tırnak arasında bu anlamda hiçbir farklılık yoktur. (Tabii yazı tek tırnakla babaşatılıp çift tırnakla sonlandırılamaz. Benzer biçimde çift tırnakla başlatılıp tek tırnakla bitirilemez.)

Python'da da pek çok dilde olduğu gibi özel basılamayan (non-printable) karakterin bazıları için "ters bölü karakter sabitleri (escape sequence)" kullanılmaktadır. Önce bir '\' karakteri sonra buna yapışık olarak özel bir karakter çifti aslında basılamayan başka bir karakteri belirtmektedir. Bunların listesi aşağıdaki gibidir.

\a	alert
\b	back space
\f	form feed
\n	new line
\r	carriage return
\t	tab
\v	vertical tab

Ters bölünün yanında yukarıdaki karakterlerin dışında başka bir karakter getirilirse oradaki ters bölü artık gerçekten ters bölü karakteri anlamına gelir. Örneğin:

```
s = "ali\selami"
```

Burada \s biçiminde bir ters bölü karakteri olmadığı için \s özel bir karakter anlamına değil '\' karakteri ve ayrıca 's' karakteri anlamına gelmektedir. Ters bölü karakterinin kendisi '\\' ile temsil edilir.

Çift tırnak içerisinde çift tırnak karakterini, tek tırnak içerisinde tek tırnak karakterini doğrudan kullanamayız. Fakat çift tırnak içerisinde tek tırnak karakterini, tek tırnak içerisinde de çift tırnak karakterini doğrudan kullanabiliyoruz. Ancak her zaman \" çift tırnak karakteri olarak, \' da tek tırnak karakteri olarak yazı içerisinde kullanılabilir. Örneğin:

```
>>> s = "\"Ankara\""  
>>> print(s)  
"Ankara"  
>>> s = '\\'Ankara\''  
>>> print(s)  
'Ankara'
```

Tabii mademki Python'da tek tırnak ile çift tırnak arasında bir farklılık yok bu durumda biz aynı string'i pratik bir biçimde şöyle de ifade edebilirdik:

```
>>> s = '"Ankara"'  
>>> print(s)  
"Ankara"
```

String'ler normal olarak tek satırda yazılmak zorundadır. Örneğin aşağıdaki yazımlar geçersizdir:

```
s = "ankara  
izmir"
```

Python'da string'ler ayrıca üç tek tırnakla ya da üç çift tırnakla da belirtilebilmektedir. Örneğin:

```
"""Ali, "Veli", Selami"""  
'''Ali'nin yeri'''  
"""Ali, Veli  
Selami"""
```

Üç tırnakla string belirtmenin tek tırnakla string belirtmekten şu farklılıklarları vardır:

- 1) Üç tırnaklı belirtmede string içerisindeki yazı birden fazla satıra yaydırılabilir.
- 2) Üç tırnak içerisinde tek tırnak ya da çift tırnak karakterleri istenildiği gibi kullanılabilir.

Üç tırnak içerisinde tırnak karakterlerinin kullanımında bir istisna durum vardır. Üç tırnağın bitiminde ilgili tırnak karakterinden bulundurulursa Python yorumlayıcılarının ayrıştırma (parsing) tekniğinden dolayı sorun oluşmaktadır. (Pek çok programlama dilinde olduğu gibi Python'da da yan yana anlamlı en uzun karakter kümelerinden atom yapılmaktadır.) Örneğin:

```
>>> '''Ali'''  
SyntaxError: EOL while scanning string literal
```

Burada string bitimindeki ilk üç tırnak bitimi belirten üç tırnak olarak ele alınmaktadır dolayısıyla sonraki tırnak tek başına kalmaktadır. Tabii aynı durum başta olsaydı bir sorun oluşmayacaktı:

```
>>> ''''Ali'''  
"Ali"
```

Tabii üç tırnağın içerisinde biz istersek ters bölümü karakterlerini de kullanabiliriz. Örneğin:

```
>>> '''Ali\''''  
"Ali'"
```

Ayrıca string'lerin başına yapışık biçimde u, U, r, R, b, B, f, F önekleri de getirilebilmektedir. u ya da U öneki UNICODE anlamına gelir. Zaten Python'daki string'ler 3'lü versiyonlarla birlikte default olarak UNICODE biçimdedir. Bu u, U önekleri Python'ın 2'li sürümlerinden kalmadır. (Dolayısıyla 3'lü sürümlerde bunların bir etkisi yoktur. Başka bir deyişle zaten normal string'lerin başında bu önekin olduğunu düşünebilirsiniz.) f önekleri "string interpolasyonu" için kullanılmaktadır. String interpolasyonu ileride ele alınacakır.

r ya da R önekleri "düzenli string (regular string)" oluşturmada kullanılır. Düzenli string de normal bir string'tir. Bunların tek farkı bunlar içerisindeki '\' karakterlerinin özel ters bölümü karakterleri anlamına gelmeyip gerçekten ters bölümü karakteri anlamına gelmesidir. Eğer ters bölümü yoğun bir yazı kullanıyorsak onu başına r ya da R öneki getirerek daha kolay yazabiliriz. Böylece çift ters bölümü kullanmaya gerek kalmaz. Örneğin:

```
>>> s = 'c:\\ali\\\\temp'  
>>> print(s)  
c:\\ali\\temp  
>>> s = r'c:\\ali\\temp'  
>>> print(s)  
c:\\ali\\temp
```

Bir stringin başına ona yapışık olarak b ya da B öneki getirilirse böyle string'lere "binary string"ler ya da "bytes nesneleri" denilmektedir. Binary string'ler içerisinde UNICODE karakterler kullanılamaz. Yalnızca 128 ASCII karakteri kullanılabilir. Binary string'ler ileride ele alınacaktır.

Pyton'da binary string'ler dışında bir string önekli ya da öneksiz nasıl belirtilmiş olursa olsun o str türünden olur. Örneğin:

```
>>> s = """Ali"""
>>> type(s)
<class 'str'>
>>> s = 'Veli'
>>> type(s)
<class 'str'>
```

5) Paython'da None anahtar sözcüğü NonType türünden sabit olarak değerlendirilir. Zaten türler konusunda da belirttiğimiz gibi NonType türünün tek bir değeri vardır. O da None değeridir. Örneğin:

```
>>> a = None
>>> type(a)
<class 'NoneType'>
```

6) Python'da bir int ya da float türden bir sayıya bitişik yazılmış j harfi complex türünden sabit belirtmektedir. Örneğin:

```
>>> z = 2j + 2
>>> type(z)
<class 'complex'>
```

Python'da Yorumlama

Python'da satır sonuna kadar yorumlama '#' ile yapılmaktadır. Bazı dillerde olduğu gibi // ile satır sonuna kadar yorumlama Python'da yoktur. Python'da birden fazla satıra yaydırılmış yorumlar da diğer pek çok dilde olduğu gibi /* ... */ ile yapılmamaktadır. Bunun yerine Python'da üç tek tırnaklı ya da üç çift tırnaklı string'ler kullanılmaktadır. Böyle string'lerin bir değişkene atanmaması genel olarak bir soruna yol açmamaktadır. Örneğin:

```
# Test program
"""
Bu satırlar
yorumlayıcı tarafından
dikkate alınmaz
"""
x = 10
print(x)
```

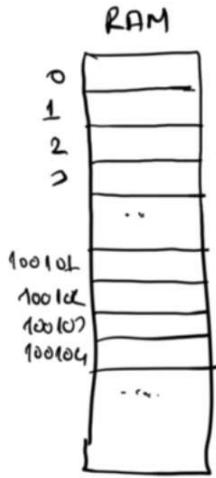
Adres Kavramı

CPU ana bellek ile elektriksel olarak bağlantılıdır. Tüm aritmetik işlemler, karşılaştırma işlemleri ve bit işlemleri CPU tarafından yapılmaktadır. Fakat değişkenlerin değerleri bellekte tutulur. Örneğin:

```
a = b + c
```

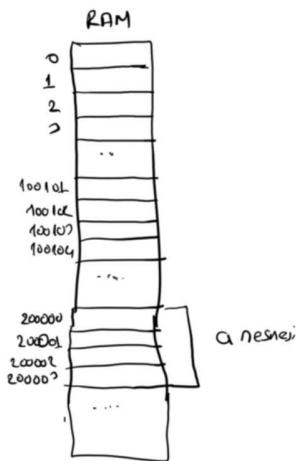
Böyle bir işlemde aslında a, b ve c ana bellektedir. CPU ana bellekteki b ve c'yi kendi içersine çeker. Elektrik devreleriyle bu toplamayı yapar. Sonucu da yeniden ana bellekteki c'ye yerleştirir.

Belleğin tepesindeki byte 0 olmak üzere belleğin her bir byte'ına artan sırada bir sayı karşılık düşürülmüştür. Bu sayıya ilgili byte'ın adresi denilmektedir. CPU bir nesneyi onun ismini bilerek değil adresini bilerek işleme sokmaktadır.



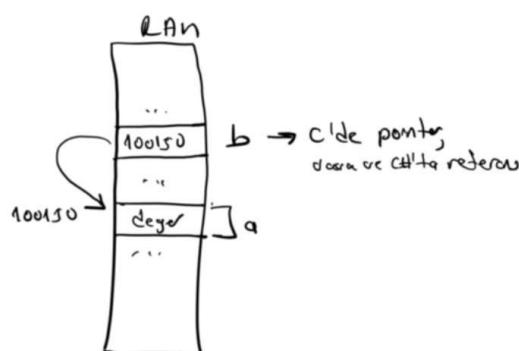
Temel yazılım terminolojisinde bellekte yer kaplayan ve kendisine erişilebilen alanlara nesne (object) denilmektedir. Değişkenler aslında programcının nesnelere verdiği isimlerdir. Programlar derlenip çalıştırılabilir hale getirildiğinde (tabii derleme sistemi için bu durum söz konusudur) değişkenler yerine makine kodlarında onların adresleri bulunmaktadır.

Mademki nesneler bellekte yer kaplamaktadır o halde onların da adresleri vardır. Örneğin:



Nesneler bir byte'tan daha büyük olabilirler. Bir byte'tan daha büyük nesnelerin adresleri tek tek onların byte'larının adresleriyle ifade edilmez. Yalnızca onların en küçük adresli byte'larının adresleriyle ifade edilir. Örneğin yukarıdaki a nesnesinin adresi 200000 biçimindedir.

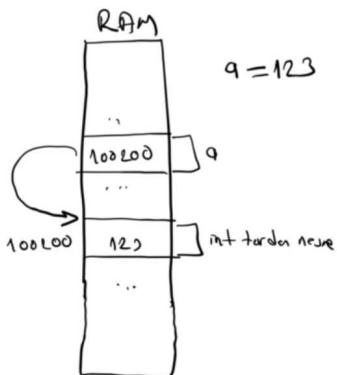
Bazen bir nesne doğrudan değer tutmaz da başka bir nesnenin adresini tutabilir. Bu durumda bu nesne diğer nesneyi göstermektedir. Böyle nesnelere C ve C++'ta gösterici (pointer), C# ve Java'da referans denilmektedir. Örneğin:



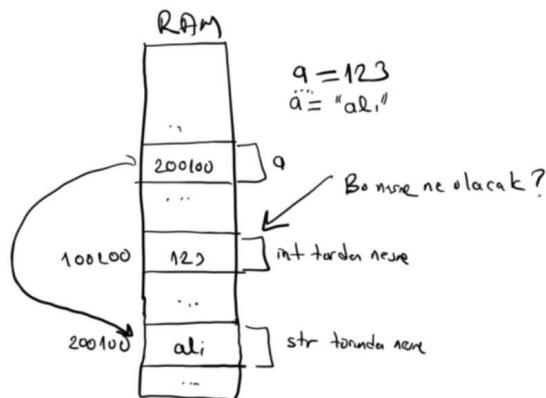
Burada b nesnesi başka bir nesnenin (a nesnesinin) adresini tutmaktadır. Yani b a'yi göstermektedir. Bu durumda b'ye erişebilen bir kişi a'ya dolaylı olarak erişebilir. İşte böyle dolaylı erişimler bazı dillerde ve uygulamalarda sıkılıkla kullanılmaktadır.

Python'da Değişkenlerin Anlamı

Dinamik tür sistemine sahip olan Python'da aslında tüm değişkenler adres tutan birer gösterici (pointer) biçimindedir. Bir değişkene bir sabit atandığında atanın sabit önce Python yorumlayıcı sistemi tarafından bir nesne olarak ana bellekte tahsis edilir. Sabit o nesnenin içerisine yerleştirilir. O değişkene de aslında o nesnenin adresi atanır. Yani Python'da tüm değişkenler aslında nesneleri gösteren birer adres tutmaktadır. Biz o değişkenleri kullandığımızda onların içerisindeki adresleri değil onların gösterdiği yerdeki değerleri kullanmış oluruz. Örneğin:



Bir değişkene başka bir sabit atandığında o başka sabit için yine yeni bir alan tahsis edilir. Artık değişken o yeni sabitin yerleştirildiği nesneyi gösteriyor durumda olur. Örneğin:



Burada a'ya daha sonra "ali" biçiminde bir string nesnesi atanmıştır. Artık a eski int nesnesini değil yeni string nesnesini gösteriyor durumda olur. Pekiyi eski int nesnesine ne olacaktır? İşte Python bir çöp toplayıcı mekanizamaya sahiptir. Kullanılmaz duruma gelen nesneler en kısa zamanda yorumlayıcı sistem tarafından yok edilmektedir. Python'ın çöp toplayıcı mekanizaması hakkında ileride bazı bilgiler verilecektir. Yani bizim eski nesneler için kaygılanması gereklidir.

Pekiyi bir değişken bir değişkene atandığında ne olur? İşte bir değişken başka bir değişkene atandığında aslında arka planda o değişkenin içindeki adres diğer değişkene atanmaktadır. Böylece iki değişken aynı nesneyi gösterir duruma gelir. Örneğin aşağıdaki gibi bir işlem yapıldığını düşünelim:

```
a = 123
b = a
```

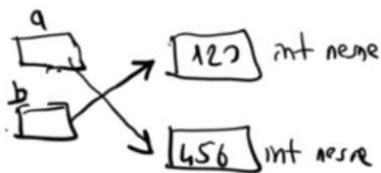
Burada a değişkeni 123'ün bulunduğu int nesneyi göstermektedir. b = a işlemi ile b de orayı gösterir hale gelir:



Şimdi artık biz a'yı da yazdırsak b'yı de yazdırsak 123 yazdırılacaktır. Pekiyi bu noktadan sonra a'ya yeni bir değer atarsak ne olur. Örneğin:

```
a = 123
b = a
a = 456
```

İşte burada eğer bu 456 değeri a'nın gösterdiği yere atansayı b de bundan etkilenirdi. Halbuki b bu işleminden etkilenmemektedir. Çünkü bu tür durumlarda bir değişkene yeni bir değer atandığında o değişkenin içeriği içeren nesneyi gösterir duruma gelmektedir. Örneğin:



Tabii burada a'nın eskiden gösterdiği ve içerisinde 123 değerinin bulunduğu nesne henüz çöp toplayıcı tarafından yok edilmeyecektir. Çünkü onu gösteren başka bir değişken daha vardır.

Python'da Atama İşlemleri

Python'da her atama işlemi bir "adres ataması" anlamına gelmektedir. Yani biz bir değişkeni diğerine atadığımızda aslında o değişkenin içindeki adresi diğerine atamış oluruz. Örneğin:

```
b = a
```

Burada a'nın içindeki adres b'ye atanmaktadır. Böylece a ve b aynı nesneyi gösterir duruma gelmektedir. Biz bir değişkene bir sabit atadığımızda sabitin yerleştirildiği nesnenin adresini değişkene atamış oluruz. Örneğin:

```
a = 100
```

Burada bir int nesne yaratılıp onun adresi a'ya atanmaktadır.

Ancak Python'da bir değişken atamanın dışında bir ifadede kullanıldığında artık o değişkenin içerisindeki adres değil o değişkenin gösterdiği yerdeki nesne kullanılıyor olur. Örneğin:

```
b = a + 1
```

Burada a'nın gösterdiği yerdeki nesneye 1 toplanacak yeni bir int nesne yaratılıp bu toplam onun içerisinde yerleştirilecek ve o yeni nesnenin adresi b'ye atanacaktır.

Göründüğü gibi bir değişkene bir ifade atandığında bu da yeni bir nesnenin yaratılması yol açmaktadır. Örneğin:

```
a = 10
b = a + 1
```

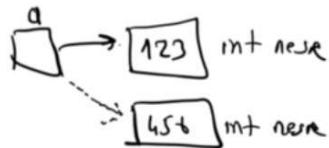
Burada a değişkeni 10 değerinin yerleştirilmiş olduğu nesnenin adresini tutar. b değişkeni de 11 değeri tutan bir nesne yaratılacak ve onun adresini tutacaktır.

Değiştirilemez (Immutable) ve Değiştirilebilir (Mutable) Tür Kavramı

Python'da türler kategorik olarak iki bölüme ayrılmaktadır: Değiştirilemez türler (immutable types) ve değiştirilebilir türler (mutable types). Bir değişkenin gösterdiği yerdeki nesnenin değeri (ya da içeriği) nesne yaratıldıktan sonra bir daha değiştirilemiyorsa bu türden nesnelere değiştirilemez nesneler bu nesnelerin türlerine de değiştirilemez türler denilmektedir. Eğer nesne yaratıldıktan sonra onun değeri değiştirilebiliyorsa bu türden nesnelere de değiştirilebilir nesneler, onların türlerine de değiştirilebilir türler denilmektedir. Şimdiye kadar görmüş olduğumuz temel Python türleri olan int, float, bool, str, NoneType değiştirilemez türlerdir. Örneğin int türünün değiştirilemez bir tür olduğunu söyle açıklayabiliriz. Aşağıdaki gibi bir kod olsun:

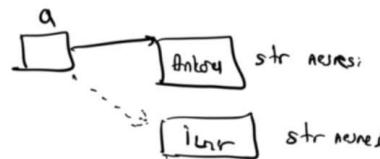
```
a = 123  
a = 456
```

Burada a'ya ikinci kez değer atandığında (üstelik aynı türden) bu değer a'nın eskiden gösterdiği yere atanmamaktadır. Başka bir deyişle 123 nesnesi bir int nesnedir. Onun değeri yaratıldıktan sonra bir daha asla değiştirilmeyecektir. 456 değeri 123 silinerek o nesneye atanmaz. 456 değeri için yeni bir int nesne yaratılır. Artık a o yeni int nesneyi gösterir duruma gelir.



Aynı durum diğer temel türler için de geçerlidir. Örneğin:

```
a = 'Ankara'  
a = 'İzmir'
```



Aynı durum değişken atamalarında da aynı biçimde söz konusudur. Örneğin:

```
a = 123  
b = a  
a = 456
```

Burada daha önce de gördüğümüz gibi a'ya yeni değer atandığında eski değer değiştirilmemekte yeni değer için yeni bir nesne yaratılmaktadır.

Fakat Python'da bütün türler değiştirilemez türler değildir. Listeler, sözlükler gibi türler ya da programcının bildirdiği sınıflar değiştirilebilir türlerdir. Bunlar ileride ele alınacaktır.

Python'da bir nesnenin tür bilgisi nesnenin içerisinde saklanmaktadır. Yani örneğin int türden bir nesnesin içerisinde hem onun değeri vardır hem de onun int türden olduğunu belirten bir tür bilgisi de vardır. Benzer biçimde str türünden bir nesnenin içerisinde hem string yazısı hem de o nesnenin str türünden olduğunu belirten tür bilgisi bulunmaktadır.

Python Standart Kütüphanesi

Python'ın da diğer bazı dillerde olduğu gibi geniş bir standart kütüphanesi vardır. Bu standart kütüphanenin içerisinde değişik konulara ilişkin hazır pek çok fonksiyon ve sınıf bulunmaktadır. Bizim bu fonksiyonları ve sınıfları kullanmak için ayrıca bir yükleme yapmamıza gerek yoktur. Bunlar Python sistemi kurulurken zaten birinci elden kurulmuş olmaktadır.

Python'da fonksiyonlar ve sınıflar modül denilen dosyalara yerleştirilmiş biçimde bulunurlar. Bir modül içerisinde fonksiyonların ve sınıfların bulunduğu bir Python dosyasıdır. Ancak bu fonksiyonlar derlenmiş biçimde değildir. Kaynak kod biçimindedir. Fakat ne olursa olsun Python'daki modülleri biz bazı dillerdeki DLL'lere bazı dillerdeki paketlere (packages) benzetebiliriz. Bir modülün kullanımına hazır hale getirilmesi için import deyimi kullanılmaktadır. Yani modüller import edildikten sonra kullanılırlar.

Python'da çok gereksinim duyulan bazı fonksiyonlar ve türler hiç import edilmeden doğrudan kullanılabilmektedir. Bunlara Python terminolojisinde "built-in fonksiyonlar ve türler" denilmektedir. Örneğin int, float, bool, str gibi temel türler bu anlamda "built-in" türlerdir. Built-in fonksiyonlardan bazıları ise da şunlardır:

```
print  
input  
len  
format  
range  
type  
max  
chr  
round  
help  
dir
```

Kursumuz ilerledikçe çeşitli built-in fonksiyonları konular içerisinde açıklayacağız.

Python'da Fonksiyon ve Metot Kavramları

Python'da hiçbir sınıfın içerisinde olmayan global altprogramlara fonksiyon (function), sınıfların içerisinde bulunan altprogramlara ise metot (method) denilmektedir. Fonksiyonlar için iki önemli kavram vardır: "Fonksiyonun tanımlanması (function definition)" ve "fonksiyonun çağrılması (function call)". Fonksiyonun tanımlanması onun bizim tarafımızdan ya da başkaları tarafından yazılması anlamına gelir. Fonksiyonun çağrılması ise onun çalıştırılması anlamına gelmektedir. Şüphesiz bir fonksiyonun çağrılabilmesi için onun daha önceden tanımlanmış olması gereklidir. Python'ın standart kütüphanesinde önceden tanımlanmış pek çok fonksiyon hazır bir biçimde bulunmaktadır. Fonksiyon tanımlama işlemi ilerde ayrı bir başlık halinde ele alınacaktır.

Python'da Fonksiyonların ve Metotların Çağrılması

Python'da global fonksiyon çağrıma işleminin genel biçimi şöyledir:

```
<fonksiyon ismi>([argüman listesi])
```

Göründüğü gibi çağrıma işleminde fonksiyon isminden sonra parantezler içerisinde argüman listesi belirtilmektedir. Örneğin:

```
>>> a = 10  
>>> print(a)  
10  
>>> a = 'ali'  
>>> print(a)  
ali
```

print fonksiyonu her türden nesnenin içerisindeki bilgiyi ekrana yazdırmak için kullanılan built-in bir Python fonksiyonudur. Örneğin:

```
>>> print('a =', a, 'b =', b)
a = 10 b = 20
```

Fonksiyonları çağrıırken argüman olarak ifadeler kullanılabilir. Örneğin:

```
>>> x = 10
>>> print(x + 2)
12
```

Örneğin help isimli fonksiyon bizden bir string nesnesi alıp onun belirttiği fonksiyonun bilgisini ekrana yazdırmaktadır:

```
>>> help('print')
Help on built-in function print in module builtins:

print(*args, **kwargs)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
        file: a file-like object (stream); defaults to the current sys.stdout.
        sep:   string inserted between values, default a space.
        end:   string appended after the last value, default a newline.
        flush: whether to forcibly flush the stream.
```

Aslında print fonksiyonu istenildiği kadar çok argüman alabilmektedir. Örneğin:

```
>>> a = 10
>>> b = 20
>>> print(a, b)
10 20
```

Bir fonksiyonu çağrıırken aralarına ',' atomu getirerek yazdığımız ifadelere argüman denilmektedir. Fonksiyon çağrırların argümanlar fonksiyonun parametre değişkenleri ile eşleştirilirler. Bu konu ileride ele alınacaktır.

Bir sınıfın metodunu (yani sınıf içerisinde bulunan bir fonksiyon) o sınıf türünden bir değişkenle ve '=' operatörü kullanılarak çağrılr. Metot çağırma işleminin genel biçimini şöyledir:

```
<sınıf türünden değişken>.<metodun ismi>([argüman listesi])
```

Örneğin s Sample isimli bir sınıf türünden olsun (yani a değişkeninin gösterdiği yerde bir Sample nesnesi olsun). Biz de Sample sınıfının foo metodunu çağırmak isteyelim:

```
s.foo(10, 20, 30)
```

10, 20, 30 metodun argümanlarıdır. Bu konunun ayrıntıları Sınıfların anlatıldığı bölümde ele alınacaktır.

id Fonksiyonu, is ve is not Operatörleri

id fonksiyonu bir nesne hakkında (yani değişkenin gösterdiği yerdeki nesne hakkında) tamsayısal tek (unique) bir teşhis değeri vermektedir. Genellikle bu teşhis değeri doğrudan nesnenin bellek adresidir. Gerçekten de pek çok Python yorumlayıcısında id fonksiyonu nesnenin bellek adresini verir. Örneğin:

```
>>> a = 10
>>> b = a
>>> id(a)
1714384640
>>> id(b)
1714384640
```

İki değişkenin id değerleri aynı ise bunlar aynı nesneyi gösteriyor durumdadır.

id fonksyonunun değişkene ilişkin bir teşhis değeri vermediğine o değişkenin gösterdiği nesneye ilişkin bir değer verdiğine dikkat ediniz. Örneğin:

```
>>> a = 10
>>> b = 10
>>> id(a)
1380308160
>>> id(b)
1380308160
>>> c = 5 + 5
>>> id(c)
1380308160
```

Burada a, b ve c'nin içerisinde aynı adreslerin olduğuna dikkat ediniz. a'ya ve b'ye 10 atandığında Python yorumlayıcısı bir optimizasyon yaparak içerisinde 10 olan üç ayrı int nesne tahsis etmek yerine tek bir int nesne tahsis edip bu değişkenlere aynı adresi atamıştır. Genel olarak programlama dillerinde kodun anlamını değiştirmemek koşuluyla derleyiciler ya da yorumlayıcılar daha etkin kod üretme hakkına sahiptir. Buna derleyicilerin ya da yorumlayıcıların "kod optimizasyonu" denilmektedir.

Python'da iki değişkenin aynı nesneyi gösterip göstermediğini anlamanın daha basit yolu is operatörünü kullanmaktır. Yani:

a is b

ifadesi

```
id(a) == id(b)
```

ifadesi ile eşdeğerdir. is operatörünün bool türeden bir değer ürettiğine dikkat ediniz. Örneğin:

```
>>> a = 10
>>> b = a
>>> c = 10
>>> a is b
True
>>> a is c
True
```

Fakat örneğin:

```
>>> a = 100
>>> b = 200
>>> a is b
False
```

is not operatörü is operatörünün ters işlemini yapar. Örneğin:

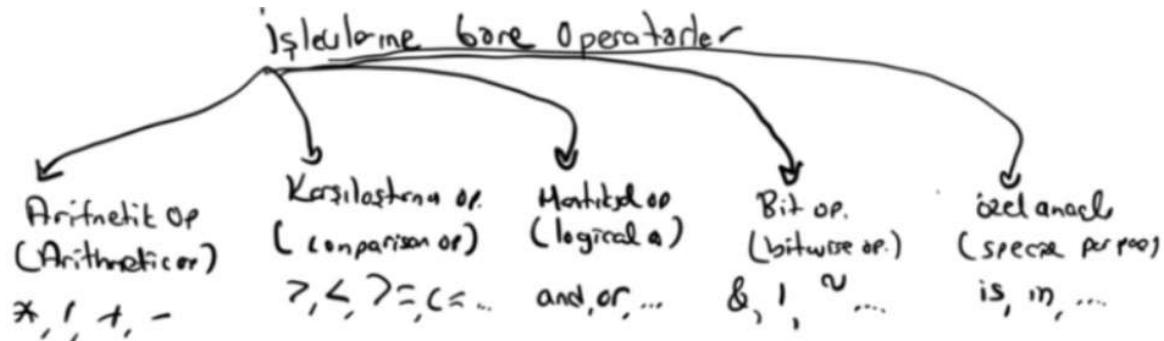
```
>>> a = 10
>>> b = a
>>> a is not b
False
```

Python'da Temel Operatörler

Bir işleme yol açan ve işlem sonucunda bir değer üretilmesini sağlayan atomlara operatör denilmektedir. Örneğin +, -, *, is, is not birer operatördür. Genellikle operatörler üç bakımdan sınıflandırılırlar:

- 1) İşlevlerine Göre Sınıflandırma
- 2) Operand Sayılarına Göre
- 3) Operatörün Konumuna göre

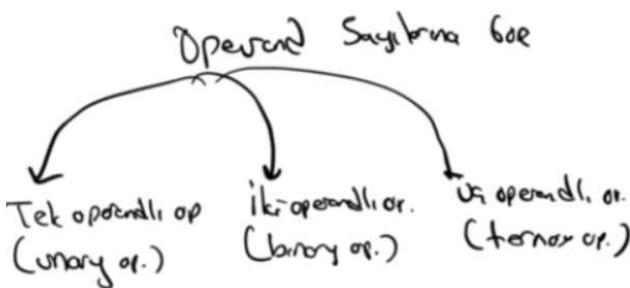
İşlevlerine göre sınıflandırma o operatörün nasıl bir işlem yaptığına yönelik sınıflandırmadır. İşlevlerine göre operatörler genellikle beş grupta değerlendirilmektedir:



Bir operatörün işleme soktuğu ifadelere operand denilmektedir. Örneğin:

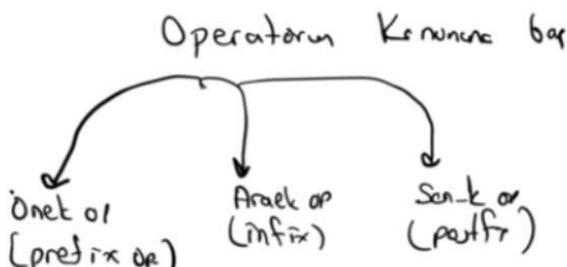
$$a + b$$

Burada + bir operatör, a ve b de bunların operandlarıdır. Operand sayılarına göre operatörler üç gruba ayrılmaktadır:



Örneğin '+' operatörü iki operandlı (binary) bir operatördür. Biz onu $a + b$ gibi iki operand vererek kullanırız. Halbuki not operatörü tek operandlı bir operatördür. Biz onu not a biçiminde tek operand vererek kullanırız.

Operatörün konumuna göre operatörler yine üç gruba ayrılmaktadır:



Bazı operatörler operand'larının arasına getirilerek, bazıları önüne getirilerek bazıları da sonuna getirilerek kullanılırlar. Örneğin:

$$a * b$$

İşlemine bakıldığından burada '*' bir aritmetik opeartördür. İki operand almaktadır ve operatör operand'larının arasına getirilmektedir. Yani '*' "iki operandlı araek (binary infix)" bir aritmetik operatördür. Örneğin:

not a

not operatörü ise "tek operand'lı önek (unary prefix)" bir mantıksal operatördür.

Bir operatörü teknik bakımdan tanımlamak için bu üç sınıflandırmada da nereye düştüğü belirtilmelidir. Örneğin: "*" operatörü iki operandlı (binary) araek (infix) aritmetik operatördür". Ya da örneğin "not operatörü tek operandlı (unary) önek (prefix) mantıksal operatördür" gibi. Tabii sonra operatöre ilişkin diğer bilgiler verilebilir. Çünkü operatörün birtakım kendine özgü durumları vardır.

Operatörler Arasındaki Öncelik İlişkisi

Bir ifadede birden fazla operatör bulunuyor olabilir. Bunlar belli bir sırada yapılırlar. Örneğin:

a = b + c * d

i1: c * d
i2: b + i1
i3: a = i2

Örneğin:

a = b + c + d

i1: b + c
i2: i1 + d
i3: a = i2

Operatörler arasındaki öncelik ilişkisi "operatörlerin öncelik tablosu" denilen bir tabloyla betimlenmektedir. Bu tablo satırlardan oluşur. Üst satırdaki operatörler alt satırdaki operatörlerden daha önceliklidir. Aynı satırdaki operatörler eşit önceliklidir. Her satırın yanında "Soldan-Sağ'a" ya da "Sağdan-Sola" ibaresi bulunur. Örneğin:

()	Soldan Sağa
* /	Soldan Sağa
+ -	Soldan Sağa
=	Sağdan Sola

Satırların yanlarında bulunan "Soldan-Sağ'a" ya da "Sağdan-Sola" ibareleri (associativity) ne anlama gelmektedir? Aynı satırdaki operatörler eşit öncelikli olduğuna göre bunlar soldaki önce ya da sağdaki önce olacak biçimde yapılırlar. Örneğin:

a = b - c + d

Burada + ile - operatörleri soldan-sağ'a eşit oldukları için önce - operatörü sonra + operatörü yapılacaktır. Tablonun bir satırındaki operatörlerin yer değiştirmesinin bir önemi yoktur. Bunlar bir grup belirtir. O gruptaki operatörler yanı ifadede kullanıldığından ifade içerisinde hangi soldaysa ya da sağdaysa o önce yapılır.

Öncelik tablosunun tepesindeki (...) operatörü fonksiyon çağrıma ve öncelik parantezini belirtmektedir. Örneğin:

result = foo() + bar()

i1: foo()
i2: bar()
result = i1 + i2

Örneğin:

a = (b + c) * d

```
i1: b + c  
i2: i1 * d  
i3: a = i2
```

Şimdi operatörleri inceleyelim.

* , / , + ve - Operatörleri

Bu operatörler iki operandlı araek aritmetik operatörlerdir. Operandları üzerinde dört işlem yaparlar. / operatörünün her iki operandı int türden olsa bile sonuç float türünden çıkmaktadır. Örneğin:

```
>>> 10/4  
2.5  
>>> 12/4  
3.0
```

Halbuki C, C++, Java ve C# gibi dillerde / operatörünün iki operandı da tamsayı türlerindense sonuç tamsayı türlerinden çıkar.

// Operatörü

Bu operatör de iki operandlı araek aritmetik operatördür. Tamsayılı bölme yapar. Yani soldan operandı sağdaki operanda böler, eğer bölüm pozitifse bölümden elde edilen değerin noktadan sonraki kısmını atar. Örneğin:

```
>>> 10 // 4  
2
```

// operatörünün iki operandı da int türdense sonuç int türden çıkar. Eğer operandlardan en az biri float türündense yine bölümden elde edilen sonuçtaki noktadan sonraki kısmı atılır. Fakat elde edilen sonuç float türünden olur. Örneğin:

```
>>> 10.5 // 4  
2.0
```

// operatörüne İngilizce "floordiv" operatörü de denilmektedir. Bu operatör eğer sonuç negatifse sayıyı kendisinden küçük ya da kendisine eşit olan ilk negatif tamsayıya yuvarlamaktadır. Örneğin:

```
>>> -20 / 4  
-5.0
```

Bu operatör de öncelik tablosunda * ve / ile aynı öncelik grubunda bulunmaktadır:

()	Soldan Sağa
*	Soldan Sağa
/	Soldan Sağa
+	Soldan Sağa
-	Soldan Saşa
=	Sağdan Sola

% Operatörü

İki operandlı aritmetik operatördür. Sol taraftaki operandın sağ taraftaki operanda bölümünden elde edilen kalan değerini üretir. Örneğin:

```
>>> 10 % 4  
2
```

Yine operand'ların her ikisi de int türdense sonuç int türden çıkar. Ancak operand'ların en az biri float türdense sonuç float türden olur. Örneğin:

```
>>> 10. % 4  
2.0  
>>> 10.2 % 4  
2.199999999999999
```

Negatif bir sayının pozitif sayıya bölümünden elde edilen kalan pozitif olmaktadır. (Halbuki C, C++, Java ve C#'ta sonuç negatif olur.) Örneğin:

```
>>> -10 % 4  
2
```

Pekiyi negatif bir değerin pozitif bir değere bölümünden elde edilen kalan neden negatifdir? Bunu şöyle açıklayabiliriz: a'yı b'ye böldüğümüzü düşünelim. Bu bölme işleminden elde edilen bölüm değeri c ve kalan değeri d olsun. Bu durumda $b * c + d$ işleminin a'ya eşit olması gerekmektedir. Dolayısıyla $-10 // 4$ işlemi bize Python'da -3 değerini verdiğine göre $-3 * 4 = -12$ dir. Bu durumda kalanın 2 olması gerekmektedir. Başka bir deyişle a % b işleminin sonucu aslında $a - a // b * b$ biçimindedir. Şimdi de pozitif bir sayının negatif bir sayıya bölümünden elde edilen kalana bakalım:

```
>>> 10 % -4  
-2
```

Burada da $10 - 10 // -4 * -4$ işleminin -2 olduğunu görüyorsunuz.

% operatörü öncelik tablosunda *, / ve // ile aynı öncelik grubunda bulunmaktadır:

()	Soldan Sağa
*	Soldan Sağa
/ //	Soldan Sağa
+	Soldan Sağa
=	Sağdan Sola

** (Üs) Operatörü

** operatörü iki operandlı arakek bir aritmetik operatördür. Sol taraftaki operandın sağ taraftaki operandla belirtilen kuvvetini alır. Örneğin:

```
>>> 2 ** 10  
1024  
>>> 2 ** 1.2  
2.2973967099940698  
>>> 2 ** -4  
0.0625
```

Yine bu operatörün de her iki operandı da int türdense sonuç int türden çıkar. Operandlardan en az biri float türdense sonuç float türden olur. ** operatörü öncelik tablosunda *, /, // ve % operatörlerinden daha yüksek öncelikte bulunmaktadır:

()	Soldan Sağa
**	Sağdan Sola
*	Soldan Sağa
/ //	Soldan Sağa
+	Soldan Sağa
=	Sağdan Sola

** operatörünün sağdan sola öncelikli olduğuna dikkat ediniz. Örneğin:

```
>>> 2 ** 4 ** 2  
65536
```

Halbuki:

```
>>> (2 ** 4) ** 2  
256
```

** operatörünün * operatöründen yüksek öncelikli olduğuna da dikkat ediniz. Örneğin:

```
>>> 2 * 3 ** 2  
18
```

İşaret + ve İşaret - Operatörleri

Bu operatörler tek operandlı önek operatörlerdir. - operatörü operandının negatif değerini + operatör operandı ise operandıyla aynı değeri üretir. Bu operatörler öncelik tablosunun üçüncü düzeyinde sağdan-sola grupta bulunurlar:

()	Soldan Sağa
**	Sağdan Sola
- +	Sağdan Sola
* / // %	Soldan Sağa
+ -	Soldan Sağa
=	Sağdan Sola

Örneğin:

```
>>> ---3  
-3
```

Burada:

```
i1: -3  
i2: -i1  
i3: -i2
```

Aşağıdakilerden hangisi çıkartma operatörü hangileri işaret - operatördür?

```
>>> 10 ----- 5  
15
```

Burada ilk - operatörü çıkartma diğer - operatörleri işaret - operatörleridir.

** operatörünün işaret + ve işaret - operatörlerinden daha yüksek öncelikli olduğuna dikkat ediniz. C, C++, Java ve C# gibi dillerden gelen kişiler bu durumu yadırgayabilmektedir. Örneğin:

```
>>> -2 ** 2  
-4
```

Fakat örneğin:

```
>>> (-2) ** 2  
4
```

Python'da işaret + ve işaret - operatörlerinin ** operatöründen düşük öncelikli olması okunabilirliği bozabilmektedir. Dolayısıyla bu öncelik yerlesimi eleştirilebilir. Ayrıca işaret + ve işaret - operatörleri ** operatörünün sağ tarafındaki operand'ta bulunuyorsa mecburen bu işaret işlemi önce yapılmaktadır. Örneğin:

```
>>> 2 ** -2  
0.25
```

Python'da ++ ve -- Operatörleri Yoktur

Python'da C, C++, Java ve C# gibi dillerdeki ++ ve – operatörleri yoktur.

Karşılaştırma Operatörleri

Python'da diğer pek çok dilde olduğu gibi 6 karşılaştırma operatörü vardır. Bunların hepsi iki operandlı arakek operatörlerdir:

> < >= <= == !=

Bu operatörler öncelik tablosunda aritmetik operatörlerden daha düşük grupta bulunurlar.

()	Soldan Saşa
**	Soldan Saşa
- +	Sağdan Sola
* / // %	Soldan Saşa
+ -	Soldan Saşa
< > <= >= == !=	Soldan Saşa
=	Sağdan Sola

Karşılaştırma operatörleri bool değer üretirler. Örneğin:

```
>>> 3 > 2  
True  
>>> 3 == 3  
True
```

Örneğin:

```
>>> 10 + 2 > 3 + 8  
True  
  
i1: 10 + 2  
i2: 3 + 8  
i3: i1 > i2
```

Python'da karşılaştırma operatörleri kombine edildiğinde matematiksel anlamla ifade ele alınmaktadır. Örneğin:

```
>>> a = 10  
>>> 5 < a < 20  
True
```

$5 < a < 20$ ifadesi şöyle ele alınmamaktadır:

```
i1: 5 < a  
i2: i1 < 20
```

Bu ifade matematikteki gibi a değerinin 5 ile 20 arasında olup olmadığını sorgulamaktadır. O halde:

$a < b < c$

gibi bir ifade aslında şu anlama gelmektedir:

$a < b \text{ and } b < c$

Örneğin:

```
>>> a = 10
```

```
>>> 10 == a > 5  
True
```

Buradaki işlemin eşdeğeri şöyledir:

```
10 == a and a > 5
```

Örneğin:

```
>>> 3 < a < 20 > 10  
True
```

İşleminin eşdeğeri de şöyledir:

```
3 < a and a < 20 and 20 > 10
```

Örneğin:

```
>>> a = 10  
>>> b = 10  
>>> c = 10  
>>> a == b == c  
True
```

Bu işlemin eşdeğeri:

```
>>> a == b and b == c  
True
```

birimindedir.

Göründüğü gibi Python'da karşılaştırma operatörleri kombine edildiğinde and etkisi yaratılmaktadır. Bunun pek çok programlama dilinde böyle olmadığına dikkat ediniz.

Mantıksal Operatörler

Python'da üç mantıksal operatör vardır: and, or ve not. and ve or operatörleri iki operandlı araek, not operatörü ise tek operandlı önek operatörlerdir. Bu operatörler önce operandlarını bool türü olarak yorumlarlar. Sonra işlemlerini yaparlar. and, or ve not işlemleri şöyledir:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

a	not a
True	False
False	True

Python'da and, or ve not operatörlerinin operandları bool türden olmak zorunda değildir. Mantıksal işlemlerde int ve float türler bool türüne dönüştürülebilir. Bu durumda sıfır dışı değerler True olarak, sıfır değeri False olarak dönüştürülmektedir.

Python'da diğer çok dilde olduğu gibi and ve or operatörlerinin kısa devre özelliği vardır. Yani bu operatörlerin önce sol tarafı yapılmış duruma göre sağ tarafları hiç yapılmayabilirler.

and operatörü şöyle çalışmaktadır: and operatörünün önce sol tarafındaki operanda bakılır. Bu operand False ise sol taraftaki operandın aynısı, True ise (yani sıfır dışı değer ya da True) sağ taraftaki operandın aynısı elde edilir. Yani and operatörünün ürettiği değer Python'da bool türden olmak zorunda değildir. and operatörünün ürettiği değer ya sol taraftaki operand türünden ya da sağ taraftaki operand türündendir. Örneğin:

```
>>> result = False and True  
>>> result  
False  
  
>>> result = True and True  
>>> result  
True
```

Örneğin:

```
>>> 100 and -200  
-200
```

Örneğin:

```
>>> 0 and 100  
0
```

Or operatörü de şöyle çalışmaktadır: Önce or operatörünün de sol tarafındaki operand yapılır. Eğer bu operand True (sıfır dışı bir değer ya da True) ise bu operandın aynısı, False (sıfır ya da False) ise sağ taraftaki operandın aynısı elde edilir. Yani Python'da or operatörünün ürettiği değer de bool türden olmak zorunda değildir. Sonuç duruma göre ya soldaki operand türünden ya da sağdaki operand türünden olur. Örneğin:

```
>>> 100 or False  
100  
>>> False or True  
True
```

not operatörü True değeri False değere, False değeri True değere dönüştürür. Örneğin:

```
>>> not True  
False  
>>> not -150  
False
```

Mantıksal and ve or operatörleri karşılaştırma operatörlerinden daha düşük önceliklidir:

()	Soldan Sağa
- +	Sağdan Sola
**	Sağdan Sola
* / // %	Soldan Sağa
+ -	Soldan Sağa
< > <= >= == !=	Soldan Sağa
not	Sağdan Sola
and	Soldan Sağa
or	Soldan Sağa
=	Sağdan Sola

Örneğin:

```
>>> 3 > 2 and 1 > 4  
False
```

not operatörünün öncelik tablosundaki yerine dikkat ediniz. Pek çok programlama dilinde bu operatör oldukça önceliklidir. Halbuki Python'da and ve or operatörlerinden yüksek öncelikli olsa da aritmetik operatörlerden düşük önceliklidir. Örneğin:

```
>>> not 10 * 3  
False
```

```
i1: 10 * 3  
i2: not i1
```

Bu ifadenin eşdeğer C karşılığı şöyledir:

```
!10 * 3
```

Sonuç C'de 0 çıkar. Örneğin:

```
>>> (not 10) * 3  
0
```

Python'da şu ana kadar görmediğimiz bazı türler bool türüne dönüştürülebilmektedir. Örneğin:

- Dolu bir liste bool türüne True olarak boş bir liste False olarak dönüştürülür.
- Dolu bir demet (tuple) bool türüne True olarak, boş bir demet False olarak dönüştürülür.
- Dolu bir sözlük (dictionary) ya da küme (set) True olarak boş bir sözlük ya da küme False olarak dönüştürülür.
- Dolu bir string bool türüne True olarak boş bir string False olarak dönüştürülür.
- None değeri False olarak bool türüne dönüştürülür.

Atama Operatörü

Atama operatörü aslında Python'da bir operatör olarak değil bir deyim biçiminde ele alınmıştır. Çünkü atama operatörünün pek çok biçimini vardır. Biz burada şimdilik onu bir operatör olarak değerlendireceğiz.

Atama operatörü iki operandlı araek özel amaçlı bir operatördür. Sağdaki ifadenin belirttiği nesnenin adresini soldaki değişkene atamakta kullanılır. Atama operatörü atanmış olan değeri üretmektedir. Atama operatörü sağdan sola önceliklidir. Örneğin:

```
>>> a = b = 100  
>>> a  
100  
>>> b  
100  
  
i1: b = 100  
i2: a = i1
```

Yukarıda da belirttiğimiz gibi Python'ın referans dokümanlarında aslında atama operatörü bir operatör olarak değil de bir deyim olarak ele alınmaktadır. Bu nedenle Python'da atama operatörü bir değer üretmez. Dolayısıyla C, C++, Java, C# gibi dillerde olduğu gibi atanan değer aynı ifadede işleme sokulamaz. Örneğin:

```
b = (a = 10) + 20;
```

gibi bir ifade Python'da geçerli değildir:

```
>>> b = (a = 10) + 20
SyntaxError: invalid syntax
```

Python'da atama işlemlerinin adres ataması anlamına geldiğini bir kez daha anımsatmak istiyoruz. Atama operatörünün birtakım ayrıntıları vardır. Bunlar ilgili konularda ele alınacaktır.

Python'a 3.8 versiyonuyla birlikte := biçiminde yeni bir atama operatörü de eklenmiştir. Bu atama operatörü gerçekten bir operatör olup C, C++, Java, C# dillerinde olduğu gibi değer de üretmektedir. Örneğin:

```
>>> a = (b := 10) + 20
>>> a
30
>>> b
10
```

Walrus operatörü if, while, for gibi deyimlerde, lambda ifadelerinde, her türlü atama işleminde kullanılabilmektedir.

Walrus operatörü bir deyimin içerisinde ya da başka bir sentakik yapının içerisinde kullanılmayıza paranteze alınmak zorundadır. Örneğin:

```
>>> a := 10
SyntaxError: invalid syntax
```

Buradadaki parantezler aslında atama operatörünün kullanılması gereken bir yerde gereksiz bir biçimde Walrus operatörünün kullanılması nedeniyle programcayı caydırırmak amacıyla gerekli göeülmüştür. Örneğin:

```
>>> (a := 10)
10
```

Gerçekten de bu tür durumlarda Walrus operatörünün kullanılmasına hiç gerek yoktur. Bu operatör atama sonucunda elde edilen değerin bir biçimde kullanıldığı durumlar için anlamlıdır. Aşağıdaki durumda da aynı gerekçeyle error oluşturmaktadır:

```
>>> a = b := 10
SyntaxError: invalid syntax
```

Burada da b := 10 ifadesi parantez içerisinde alınmalıdır:

```
>>> a = (b := 10)
>>> a
10
>>> b
10
```

Tabii bu örnekte de aslında Walrus operatörünün kullanımına hiç gerek yoktu:

```
>>> a = b = 10
>>> a
10
>>> b
10
```

İşlemli Atama Operatörleri (Augmented Assignment Operators)

Python'da bir grup +=, -=, *=, /=, %=, //=, ... biçiminde iki operandlı araek işlemeli atama operatörü vardır. op operatörü temsil etmek üzere:

a op= b

ifadesi,

a = a op b

anlamına gelir. Örneğin:

a += 1 ifadesi a = a + 1 ile eşdeğerdir. a *= 2 ifadesi a = a * 2 ile eşdeğerdir. Python'da C, C++, Java ve C#'ta bulunan ++ ve -- operatörleri yoktur. Bu nedenle ++a gibi bir ifade yerine Python'da a += 1 ifadesi kullanılmaktadır.

İşlemli atama operatörleri normal atama operatörüyle aynı öncelik grubunda bulunmaktadır. Örneğin:

()	Soldan Saşa
- +	Sağdan Sola
**	Sağdan Saşa
* / // %	Soldan Saşa
+ -	Soldan Saşa
< > <= >= == !=	Soldan Saşa
not	Sağdan Sola
and	Soldan Saşa
or	Soldan Saşa
= := += -= *= /=... .	Sağdan Sola

Örneğin:

```
>>> a = 3
>>> a *= 2 + 3
>>> a
15

i1: 2 + 3
i2: a *= i1
```

İşlemli atama operatörlerinden elde edilen değer diğer işlemlerde kullanılamamaktadır. Örneğin:

```
>>> b = a += 2
SyntaxError: invalid syntax
```

Mantıksal and ve or operatörlerinin işlemli biçimleri yoktur. Örneğin:

```
>>> a or= False
SyntaxError: invalid syntax
```

print Fonksiyonu

Python'ın komut satırında bir ifadeyi yazdığımızda komut satırı bize onun sonucunu göstermektedir. Ancak bu durum yalnızca komut satırına özgüdür. Program ve script içerisinde ifadelerin sonuçları bu biçimde ekrana basılmaz. İşte Python'da ekrana bir şeyler yazmak için built-in print fonksiyonu kullanılmaktadır. Biz henüz bu noktada fonksiyonların parametre değişkenleri hakkında bilgi edinmedik. Ancak burada biz henüz görmediğimiz bazı özellikleri kullanacağız.

print fonksiyonu değişken sayıda parametre alabilmektedir. Yani biz pek çok ifadeyi tek hamlede print fonksiyonuyla yazdırabiliriz. Örneğin:

```
>>> a = 10
>>> b = 20
>>> c = 30
```

```
>>> print(a, b, c)
10 20 30
>>> print(a * 2, b * 3, c * 4)
20 60 120
```

Tabii print fonksiyonundaki argümanlar farklı türlerden de olabilirler. Örneğin:

```
>>> print('a =', a)
a = 10
```

print default olarak argümanlar arasına bir tane SPACE karakteri basmaktadır. Ancak bu karakter sep isimli parametre belirtilerek değiştirilebilir. Örneğin:

```
>>> a = 10; b = 20; c = 30
>>> print(a, b, c, sep=',')
10,20,30
```

Örneğin:

```
>>> print(a, b, c, sep='xxx')
10xxx20xxx30
```

Örneğin:

```
>>> print(a, b, c, sep = '')
102030
```

print en son argümanı da yazdırdıktan sonra default olarak imleci aşağıdaki satırın başına geçirir. Ancak bu da end parametresiyle ayarlanabilmektedir. Örneğin:

```
>>> print(a, b, c, sep = '', end='.')
102030.
```

Örneğin:

```
>>> print(1);print(2);print(3)
1
2
3
```

Fakat örneğin:

```
>>> print(1, end=' ');print(2, end=' ');print(3)
1 2 3
```

Tür Fonksiyonları (Tür Sınıflarının __init__ Metotları) ve Tür Dönüşürmeleri

Python'da aslında int, float, str, bool gibi türler birer sınıf belirtmektedir. Sınıfların da aynı isimli başlangıç metotları (dunder init) bulunmaktadır. Bu durumda Python'da built-in int, float, str ve bool isimli fonksiyonlar aslında bu sınıfların başlangıç metotları gibi düşünülebilir. Bu fonksiyonlar aynı zamanda tür dönüştürmesi için de kullanılmaktadır. Sınıfların konusu ileride ele alınacaktır. Biz burada biraz üstün körü bir biçimde bu fonksiyonları tanıtabiliriz.

int fonksiyonu farklı türlerden değerleri argüman olarak alıp bize onların int türüne dönüştürülmüş hallerini verir. float bir değer int fonksiyonuna sokulduğunda sayının noktadan sonraki kısmı tamamen atılır. Bir string int fonksiyonuna sokulduğunda yazının belirttiği sayı int bir sayıya dönüştürülür. True ve False değerleri int fonksiyonuna sokulduğunda sırasıyla 1 ve 0 değerleri elde edilir. Örneğin:

```
>>> int(1.3)
```

```
1
>>> int(True)
1
>>> int('123')
123
```

Örneğin:

```
>>> int('ali')
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    int('ali')
ValueError: invalid literal for int() with base 10: 'ali'
>>> int('12.34')
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    int("12.34")
ValueError: invalid literal for int() with base 10: '12.34'
```

Göründüğü gibi int fonksiyonuna uygun olmayan bir argüman girildiğinde fonksiyon "exception" oluşturuyor. Exception terimi program çalışırken ortaya çıkan hatalar için kullanılmaktadır. Bir exception oluştuğunda o exception ele alınabilir (handle edilebilir). Eğer oluşan exception ele alınmazsa program çöker.

str türünden int ve float türlerine dönüştürmede genel olarak yazının başında ve sonundaki boşluk karakterleri (leading/trailing space) bir soruna yol açmaktadır. Örneğin:

```
>>> int(' 1234 ')
1234
```

str türünden int türüne dönüştürme yapılırken default taban 10'luk sistemdir. Ancak dönüştürme sırasında başka bir taban da belirtilebilir. Örneğin:

```
>>> s = '100'
>>> int(s, 16)
256
```

Burada 100 değeri 16'lık sistemde 100 olarak ele alınmıştır. Örneğin:

```
>>> int('101010', 2)
42
```

float fonksiyonu da çok benzerdir. int bir değer float türüne dönüştürüldüğünde float türünden aynı değere ilişkin yeni bir int nesne yaratılır. Örneğin:

```
>>> float(-123)
-123.0
```

bool bir değer float türüne dönüştürüldüğünde False için 0.0, True için 1.0 float değerleri elde edilir. Örneğin:

```
>>> float(True)
1.0
>>> float(False)
0.0
```

Bir string eğer içerisindeki karakterler float bir sayı belirtebiliyorsa float türüne dönüştürülebilir. Örneğin:

```
>>> s = '-12.34'
>>> float(s)
-12.34
```

Yine float olarak ele alınamayacak yazılar exception oluşturmaktadır. Örneğin:

```
>>> float('ali')
Traceback (most recent call last):
  File "<pyshell#80>", line 1, in <module>
    float('ali')
ValueError: could not convert string to float: 'ali'
```

str türü float türüne dönüştürülürken taban belirtilememektedir. Dönüşürme her zaman 10'luk tabana göre yapılmaktadır.

None ve complex türleri int ve float türüne dönüştürülmek istenirse exception olmaktadır.

str fonksiyonu ters bir işlem yapmaktadır. Yani sayısal bilgiyi string türüne (başka bir deyişle yazıya) dönüştürür. Örneğin:

```
>>> str(123)
'123'
>>> str(12.34)
'12.34'
>>> str(True)
'True'
>>> str(False)
'False'
>>> str(None)
'None'
```

Diğer türlerin bool türüne nasıl dönüştürüldüğü mantıksal operatörlerin anlatıldığı bölümde açıklanmıştır. Sıfır dışı değerler True olarak, sıfır değeri False olarak dönüştürülmüyordu. Örneğin:

```
>>> bool(12.3)
True
>>> bool(10)
True
>>> bool(0)
False
>>> bool('ali')
True
>>> bool('')
False
```

Boş string'in bool türüne False olarak dönüştürüldüğüne dikkat ediniz. None değeri de bool türüne False olarak dönüştürilmektedir. Örneğin:

```
>>> bool(None)
False
```

Aslında int, float, str ve bool fonksiyonları argümansız da kullanılabilir. Bu durumda int 0 değerine, float 0.0 değerine, str boş bir stringe ve bool de False değerine geri döner:

```
>>> int()
0
>>> float()
0.0
>>> str()
''
>>> bool()
False
```

int ve float türleri complex türüne önüstürülebilir. Bu durumda j kısmı 0 olan complex nesneler elde edilmektedir. Örneğin:

```
>>> complex(10)
(10+0j)
>>> complex(12.3)
(12.3+0j)
```

Bir string de eğer biçimini uygunsa complex türüne dönüştürülebilmektedir. Örneğin:

```
>>> complex('12+3j')
(12+3j)
>>> complex('12')
(12+0j)
```

bool türden complex türüne dönüştürme yapılabılır. Bu durumda gerçek kısmı 0 ya da 1 olan fakat sanal kısmı 0 olan complex nesneler elde edilmektedir. Örneğin:

```
>>> complex(True)
(1+0j)
>>> complex(False)
0j
```

input Fonksiyonu

Programlama dillerinin standart kütüphanelerinde genellikle ekrana (stdout dosyasına) yazdırma yapmak ve klavyeden (stdin dosyasından) okuma yapmak amacıyla farklı ve çeşitli fonksiyonlar bulundurulmaktadır. Halbuki Python bu konuda minimalist bir tasarıma sahiptir. Python'da ekrana birşeyler yazdırmak için yalnızca print fonksiyonu, klavyeden de bir şeyler okuyabilmek için yalnızca input fonksiyonu kullanılmaktadır.

input fonksiyonu parametresiyle aldığı yazıyı ekrana basar ve klavyeden bir yazı bekler. Kullanıcı yazıyı yazıp ENTER tuşuna bastığında input yazılan yazıyı bir string nesnesine yerleştirir ve o stringi bize verir. Örneğin:

```
>>> s = input('Bir isim giriniz:')
Bir isim giriniz:Ali
>>> s
'Ali'
```

input ile biz int, float, bool türden bir değer okumak istersek input fonksiyonundan elde edilen değeri ilgili türlerin fonksiyonlarına sokarak dönüştürmeliyiz. Örneğin:

```
>>> a = int(input('Bir değer giriniz:'))
Bir değer giriniz:123
>>> a
123
```

Örneğin:

```
>>> n = int(input('Bir sayı giriniz:')); print(n * n)
Bir sayı giriniz:10
100
```

Örneğin:

```
>>> a = float(input('Bir değer giriniz:'))
Bir değer giriniz:12.34
>>> a
12.34
```

Yanlış girişler ValueError exception'ına yol açabilir. Örneğin:

```
>>> a = int(input('Bir değer giriniz:'))
Bir değer giriniz:ali
Traceback (most recent call last):
  File "<pyshell#97>", line 1, in <module>
    a = int(input('Bir değer giriniz:'))
ValueError: invalid literal for int() with base 10: 'ali'
```

Aslında input fonksiyonu argüman olarak string almak zorunda değildir. Herhangi bir türden değer alabilir. input argüman olarak ne almışsa giriş istemeden önce onu ekrana bastırmaktadır. input fonksiyonu argümansız olarak da çağrılabılır. Bu durumda input ekrana bir şey yazmaz ancak imleci de aşağı satırın başına geçirerek giriş ister.

Farklı Temel Türlerin Birbirleriyle İşleme Sokulması

Python'da genel olarak farklı türler birbirleriyle işleme sokulamazlar. Örneğin biz bir string ile bir int değeri işleme sokamayız. Tabii farklı türleri işleme sokabilmek için operator metodunu denilen özel metodlar kullanılabilmektedir. Bu konu sınıfların anlatıldığı bölümde ele alınacaktır. Python'da istisna olarak aşağıdaki farklı türler bir arada işleme sokulabilmektedir:

- int ile float türü beraber işleme sokulabilir. Sonuç float türden çıkar. Örneğin:

```
>>> 10 + 20.2
30.2
```

- bool türü ile int ve float türleri ile işleme sokulabilir. int türüyle işleme sokulduğunda sonuç int türünden, float türüyle işleme sokulduğunda sonuç float türünde elde edilir. Daha önceden de belirttiğimiz gibi bool türü eğer belirttiği değer True ise 1 olarak False ise 0 olarak işleme sokulmaktadır. Örneğin:

```
>>> b = True
>>> i = 10 + b
>>> i
11
>>> f = 10.1 + b
>>> f
11.1
```

Tabii buradaki işlem karşılaştırma işlemi de olabilir. Ancak karşılaştırma operatörlerinin bool türünden değerler ürettiğini anımsayınız. Örneğin:

```
>>> 1 == True
True
>>> 0 == False
True
>>> 1.0 == True
True
>>> 10 > True
True
>>> 0.4 > True
False
```

- İki bool türü kendi aralarında işleme sokulursa işlem sonucunda int bir değer elde edilmektedir. Örneğin:

```
>>> True + False
1
```

- complex türü ile int float ve bool türü beraber işleme sokulabilir. Bu durumda sonuç complex türünden olur. Örneğin:

```
>>> 2j + 3  
(3+2j)  
>>> 2j + 3.2  
(3.2+2j)
```

Python'da Temel Veri Yapıları

Yazılım dünyasında aralarında belli bir ilişki olan bir grup nesnenin oluşturduğu topluluğa veri yapısı (data structure) denilmektedir. Örneğin diziler, yapılar, birlikler, bağlı listeler, hash tabloları birer veri yapısıdır. Pek çok dilde çok temel veri yapıları dilin senktası tarafından dilin bir parçası biçiminde bulunmaktadır. Fakat diğer veri yapıları bu temel veri yapıları kullanılarak fonksiyonlar ya da sınıf biçiminde yazılmak durumundadır. Tabii pek çok dil standart kütüphanesinde böyle fonksiyonları ya da sınıfları hazır biçimde yazılmış olarak da bulundurabilmektedir.

Python'da bazı veri yapıları (listeler, sözlükler, demetler, kümeler gibi) dilin kendi sentaksına dahil edilmiştir. Halbuki pek çok programlama dilinde bu veri yapıları ayrı birer sınıf olarak onların standart kütüphanelerinde bulundurulmaktadır. Biz de bu bölümde Python'daki temel veri yapılarını inceleyeceğiz.

Dolaşılabilir (iterable) Nesneler ve Dolaşım (Iterator) Nesneleri

Python'da bu bölümde göreceğimiz gibi görevi başka nesneleri tutmak olan çeşitli nesneler vardır. Bu tür nesnelere Java ve C# dillerinde "collection", C++'ta ise "container" denilmektedir. Python'da bir nesnenin dolaşılabilir olması demek elemanlarının tek tek türden bağımsız biçimde elde edilebilmesi demektir. Gerçekten de bu bölümde göreceğimiz listeler, demetler, kümeler, sözlükler ve string'ler dolaşılabilir türlerdir. Bir fonksiyon ya da metot dolaşılabilir bir parametre alıyorsa biz o fonksiyona ya da metoda burada göreceğimiz veri yapılarını argüman olarak geçirebiliriz. Dolaşılabilir nesneleri biz fonksiyonlara ve metodlara geçirdiğimizde o fonksiyonlar ve metodlar onların içerisindeki elemanların hepsini elde edebilmektedir. Tabii dolaşılabilir nesneler yalnızca liste, demet, sözlük ve string nesneleri değildir. Başka pek çok sınıfın dolaşılabilir olma özelliği bulunabilmektedir. Sınıflar kısmında kendi sınıflarımızın nasıl dolaşılabilir (iterable) hale getirilebileceğini göreceğiz.

Dolaşım (iterator) nesneleri aynı zamanda dolaşılabilir (iterable) nesneleridir. Dolaşım nesneleri ile dolaşılabilir nesneler arasında küçük bir fark vardır: Dolaşım nesneleri dolaşım bittikten sonra yeni bir dolaşım yapamazlar ancak dolaşılabilir nesneler dolaşım bittikten sonra yeni bir dolaşım yapabilirler. Dolaşılabilir nesneler ve dolaşım nesneleri son böülümlere doğru ayrı bir başlık haline ayrıntılı biçimde ele alınmaktadır. Ancak siz şimdilik dolaşılabilir nesne ya da dolaşım nesnesi denildiğinde bunların aynı anlamına geldiğini varsayıbilsiniz.

Listeler (Lists)

Veri yapıları dünyasında aralarında öncelik-sonralık ilişkisinin bulunduğu bir grup nesnenin oluşturduğu topluluğa "liste (list)" denilmektedir. Bu bakımından diziler, bağlı listeler birer listedir. Python'daki listeler diğer dillerdeki dizilere oldukça benzemektedir. Listeler Python'da "list" isimli sınıf ile temsil edilmektedir. Belli bir liste de aslında Python'da "list" sınıfı türünden bir nesnedir.

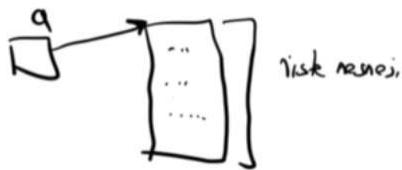
Python'da listeler çeşitli biçimlerde yaratılabilirler. En doğal yöntem köşeli parantez operatörünü kullanmaktadır. Köşeli parantezlerle liste yaratmanın genel biçimini söyleyelim:

```
[<[liste elemanları]>]
```

Liste elemanları aralarına ',' atomu yerleştirilerek oluşturulur. Örneğin:

```
>>> a = [1, 2, 3]  
>>> a  
[1, 2, 3]  
>>> type(a)  
<class 'list'>
```

Bir liste yaratılıp bir değişkene atandağında aslında tüm atamalarda olduğu gibi liste nesnesinin adresi değişkene atanmaktadır.



Örneğin:

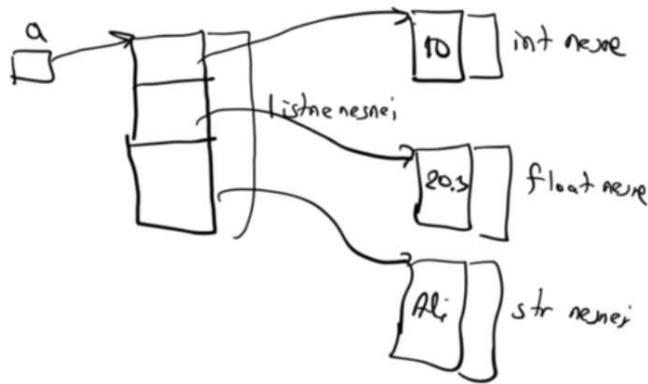
```
>>> names = ['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
>>> names
['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
>>> type(names)
<class 'list'>
```

Liste elemanları farklı türlerden (yani heterojen) olabilir. Örneğin:

```
>>> x = [10, 20.2, 'Ali', 'Selami']
>>> x
[10, 20.2, 'Ali', 'Selami']
>>> type(x)
<class 'list'>
```

Bir liste oluşturulduğunda aslında listenin elemanları o elemanlara ilişkin nesnelerin adreslerini tutmaktadır. Yani liste elemanlarında nesnelerin kendisi değil yine onların adresleri tutulmaktadır. Başka bir deyişle listeler birer adres dizisi gibi düşünülebilir. Örneğin:

$a = [10, 20.3, 'Ali']$



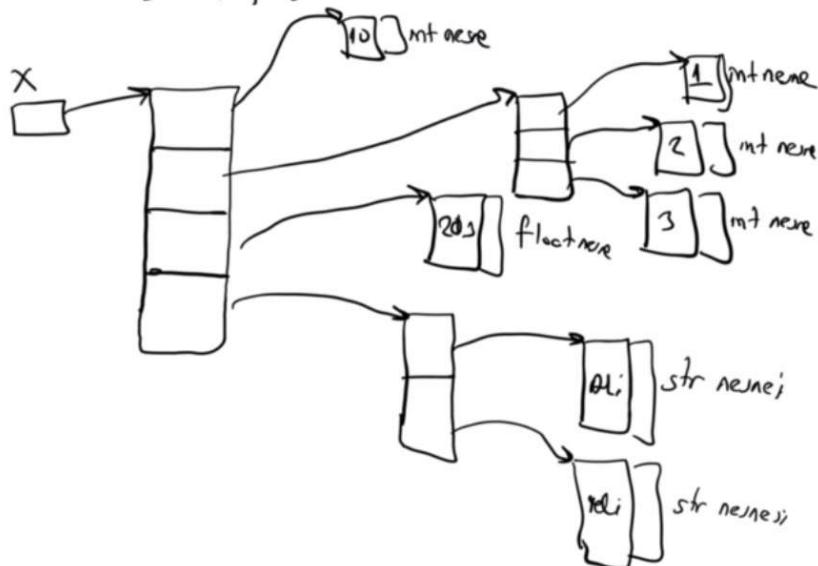
Yani nasıl bir değişken aslında bir nesnenin adresini tutuyorsa bir listenin de böyle n tane değişkenden olduğunu düşünebilirsiniz.

Bir listenin elemanı başka bir liste de olabilir. Örneğin:

```
>>> x = [10, [1, 2, 3], 20.3, ['Ali', 'Veli']]
>>> x
[10, [1, 2, 3], 20.3, ['Ali', 'Veli']]
```

Bu listenin bellekteki durumu aşağıdaki gibi temsil edilebilir:

$x = [10, [1, 2, 3], 20.3, 'Ali', 'veli']$



Bir listenin eleman sayısı built-in len fonksiyonuyla elde edilebilir. Örneğin:

```
>>> a = [10, 20, 30]
>>> len(a)
3
```

Örneğin:

```
>>> a = [1, [2, 3], 4, [5, 6]]
>>> len(a)
4
```

Listenin elemanlarına ayrı ayrı köşeli parantez operatörüyle erişilebilir. Listenin her elemanın sıfırdan itibaren bir indeks numarası vardır. Listenin ilk elemanı 0'inci indekstedir. Liste elemanlarına erişirken köşeli parantez içérisine int türden bir ifade yerleştirilir. Bu durumda o ifadenin önce değeri hesaplanır sonra listenin o indeksli elemanına erişilir. Örneğin:

```
>>> i = 0
>>> x[i + 1]
[2, 3]
```

Listenin elemanlarına erişirken indeks olarak tamsayı olmayan (int türden olmayan) bir ifade kullanamayız. Örneğin:

```
>>> a[2.3]
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    a[2.3]
TypeError: list indices must be integers or slices, not float
```

Listenin olmayan bir elemanına erişilmeye çalışılırsa exception oluşur. "Exception" programın çalışma zamanında ortaya çıkan problemli durumlar için kullanılan bir terimdir. Exception oluştuğunda bu durum ele alınmamışsa program çöker. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> a[0]
10
>>> a[3]
40
>>> a[100]
```

```
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    a[100]
IndexError: list index out of range
```

Aslında oluşan exception "ele alınarak" çökme engellenebilmektedir. Exception mekanizması ileride ayrıntılarıyla ele alınacaktır.

Elemana erişme işlemi aslında tüm "dizilim (sequence)" türleri için aynıdır. Genel biçim şöyledir:

```
<dizilim ismi><[<ifade>]>
```

Yukarıda da belirttiğimiz gibi bir listenin her elemanı aslında ayrı bir değişken gibidir. Onlar da adres tutmaktadır. Ancak onları kullandığımızda onların içerisindeki adresteki nesneyi kullanmış oluruz. Örneğin:

```
>>> a = [1, [2, 3, 4], 'Ali']
>>> b = a[0]
>>> id(a[0])
1497329120
>>> id(b)
1497329120
```

Bu örnekte a[0] ile b değişkenlerinin içerisinde aynı adresin bulunduğu görüyorsunuz.

İç içe listelerde içteki listenin elemanına erişmek için birden fazla köşeli parantez kullanmak gereklidir. Örneğin:

```
>>> a = [1, [2, 3, 4], 'Ali']
>>> a[1][2]
4
```

Örneğin:

```
>>> a = [10, [20, 30, [40, 50]], 60]
>>> a[1][2][1]
50
```

Boş bir liste söz konusu olabilir. Örneğin:

```
>>> a = []
>>> len(a)
0
```

Boş listelerin bool türüne False olarak, dolu listelerin ise True dönüştürüldüğünü anımsayınız. Örneğin:

```
>>> a = []
>>> bool(a)
False
>>> bool([10])
True
```

Listenin elemanlarına erişirken köşeli parantez içerisindeki ifadenin sayısal değeri negatif olabilir. Negatif değerler sondan itibaren başa doğru indeks belirtmektedir. Köşeli parantez içerisinde negatif değer varsa bu negatif değer ile listenin uzunluğu toplanarak gerçek indeks değeri elde edilmektedir. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> a[-1]
50
```

Burada diziminin uzunluğu 5'tir. Köşeli parantez içerisinde -1 olduğuna göre efektif indeks $5 - 1 = 4$ 'tür. Bu da listenin son elemanı anlamına gelir. O halde köşeli parantez içerisinde -1 son elemanı, -2 sondan bir önceki elemanı, -n de sondan n'inci elemanı belirtir. Örneğin:

```
>>> a = [1, [2, 3, 4], 5, [6, 7, 8]]  
>>> a[-1][-2]  
7  
>>> a[1][-1]  
4
```

Listeleri (genel olarak dizimleri) indekslemek için "dilimleme (slicing)" denilen bir yöntem de kullanılmaktadır. Dilimleme iki indeks arasındaki elemanları ayrı bir liste olarak elde eder. Dilimleme köşeli parantezler içerisinde ':' atomu ile ayrılmış "ilk indeks", "son indeks" ve "adım miktarı" belirtilerek yapılmaktadır. Biz ilk indekse "start", son indekse "stop" ve adım değerine de "step" diyeceğiz. Bu durumda dilimleme işleminin genel biçimi şöyledir:

```
[start]:[stop]:[step]
```

Buradaki köşeli parantezler dilimleme sırasında start, stop ya da step değerlerinin belirtilmeyebilecegi anlamına gelmektedir.

Dilimleme işlemine start indeks dahildir ancak stop indeks dahil değildir. Step indeks belirtilmezse step değeri 1 alınmaktadır. Dilimlemenin soldan kapalı sağdan açık bir aralık belirttiğine dikkat ediniz. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]  
>>> a[1:3]  
[20, 30]  
>>> a[0:len(a)]
```

Dilimlemede start ve stop indeksler negatif değerleresahip olabilir. Bu durumda bu negatif değerlere yine listenin uzunluğu ile toplanarak efektif indeks elde edilmektedir. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]  
>>> a[1:-2]  
[20, 30]
```

Buradaki a[1:-2] dilimlemesi a[1:len(a)-2] ile aynı anlamdadır.

Tabii dilimlemede ilk indeksin ikincisine eşit ya da ondan büyük olması anlamsızdır. Genel olarak dilimlemede eleman elde edilememesi exception'a yol açmaz. Bu durumda boş liste elde edilmektedir. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]  
>>> a[3:2]  
[]  
>>> a[1:1]  
[]
```

Ayrıca dilimlemede start ya da stop indeks listenin uzunluğundan büyük bir değerde olması bir soruna yol açmamaktadır. Listenin uzunluğundan büyük değerler liste uzunluğuna indirgenmektedir. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]  
>>> a[1:30]  
[20, 30, 40, 50]
```

Örneğin:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
>>> a[9:10]  
[10]
```

```
>>> a[9:11]
[10]
>>> a[9:12]
[10]
```

Benzer biçimde dilimlemede start indeks efektif olarak 0'dan küçükse de (yani liste uzunluğu ile toplandığı halde sıfırdan küçükse de) sorun oluşmamaktadır. Bu durumda ilk indeksin 0 olduğu varsayılar. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> a[-20:3]
[10, 20, 30]
```

Dilimlemeyle tek bir eleman çekilsse bile bu eleman bir liste biçiminde verilir. Yani dilimleme her zaman bir liste vermektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[2:3]
[3]
```

Dilimlemede start indeks belirtilmemezse sanki start indeksin 0 biçiminde girildiği, stop indeks belirtilmemezse stop indeksin de liste uzunluğu biçiminde girildiği kabul edilmektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[:5]
[1, 2, 3, 4, 5]
```

Örneğin:

```
>>> a[5:]
[6, 7, 8, 9, 10]
```

Bu sayede biz belli bir elemandan öncesini ya da sonrasıni elde edebiliriz.

Dilimlemede start indeks de stop indeks de belirtilmeyebilir. Bu durumda dilimlemeden listedeki tüm elemanlar elde edilmektedir. Örneğin:

```
>>> a[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

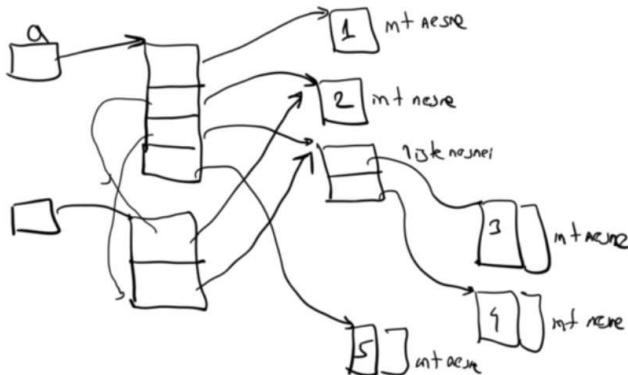
Dilimleme işleminde yaratılan yeni liste kopyalama biçiminde oluşturulmaktadır. Yani asıl listenin belli elemanları (liste elemanlarının adres tuttuğunu anımsayınız) yeni listeye eleman eleman kopyalanmaktadır. Böylece aslında kopyalanan şeyler adresler olmaktadır. Örneğin:

```
>>> a = [1, 2, [3, 4], 5]
>>> b = a[1:3]
>>> a
[1, 2, [3, 4], 5]
>>> b
[2, [3, 4]]
```

Buradaki a ve b'nin bellek görüntüsü aşağıdaki gibi olacaktır:

$a = [1, 2, [3, 4], 5]$

$b = a[1:3]$



Göründüğü gibi dilimleme yapılırken asıl listedeki elemanlar tek tek yeni oluşturulan listeye kopyalanmaktadır. Tabii liste elemanları aslında birer adres tuttuğuna göre burada kopyalanan şeyler adresler olmaktadır. Buradaki gibi kopyalama sırasında nesnelerin değil de onların adreslerinin "sığ kopyalama (shallow copy)" denilmektedir.

a isimli bir listeyi tümdeğer kopyalamanın pratik bir yolu $a[:]$ biçiminde dilimleme uygulamaktır.

```
>>> a = [1, 2, [3, 4], 5]
>>> b = a[:]
>>> id(a)
2272015060360
>>> id(b)
2272015134984
>>> id(a[0])
1497329120
>>> id(b[0])
1497329120
```

Burada görüldüğü gibi a nesnesinin adresiyle b nesnesinin adresi farklıdır. Ancak bu iki listenin elemanlarında aynı adresler vardır.

Dilimlemedeki step değeri elemanlar elde edilirken yapılacak atlamayı belirtmektedir. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[3:8:2]
[3, 5, 7]
```

Burada 3'üncü indeksten 8'inci indekse kadar (bu indeks dahil değil) ikişer aralıklı (yani birer atlanarak) dilimleme yapılmıştır. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[3:7:2]
[3, 5]
```

Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:9:4]
[0, 4, 8]
```

Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[::-2]
```

```
[0, 2, 4, 6, 8]
```

Dilimlemede adım miktarının belirtilmemesi onun 1 olarak belirtilmesi ile aynı anlaşılmadır.

Adım miktarı negatif olabilir. Bu durumda ilk operand'ın ikinci operand'tan daha büyük olması beklenir. Başka bir deyişle adım miktarı negatif ise dilimleme yönü sağdan sola (yani sondan başa doğru) yapılmaktadır. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[8:1:-1]
[8, 7, 6, 5, 4, 3, 2]
```

Burada 8'inci indeksten (8'inci indeks dahil) 1'nci indekse (1'inci indeks dahil değil) sağdan sola ilerleme yapılmaktadır.

Eğer step değeri negatif ise ve start değeri belirtilmemişse bu durumda start değerinin listenin son indeksi biçiminde belirtilmiş olduğu kabul edilir. Step değeri negatif ise ve stop değeri belirtilmemişse bu durumda da stop değerinin efektif -1 biçimde olduğu (yani $-len(a) - 1$ biçiminde olduğu) kabul edilmektedir. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:2:-1]
[9, 8, 7, 6, 5, 4, 3]
```

Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[9::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Bir listeyi ters yüz etmek için start ve stop değerini belirtmeden step değerini -1 yapmak sıkça uygulanan bir yöntemdir. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Burada `a[::-1]` ifadesi aslında `a[len(a) - 1:-len(a) - 1:-1]` ifadesi ile eşdeğerdir.

Step değerinin belirtilmemesi durumunda bu değerin 1 olarak alındığını anımsayınız. Bu durumda aşağıdaki ifadelerin hepsi aynı anlama gelmektedir:

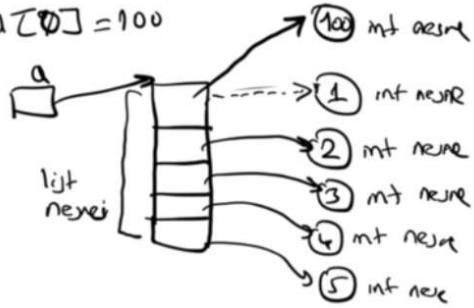
```
a[x:y]
a[x:y:]
a[x:y:1]
```

list sınıfı değiştirilebilir bir sınıfıdır. Dolayısıyla listeler de “değiştirilebilir (mutable)” nesnelerdir. Biz bir list nesnesinin elemanlarını değiştirebiliriz. (int, float, bool ve str türlerinin kategorik olarak “değiştirilemez (immutable)” türler olduğunu anımsayınız.) Listenin elemanları adres tuttuğuna göre biz bir listenin belli bir elemanını değiştirdiğimizde aslında o elemandaki adresi değiştirmış oluruz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0] = 100
>>> a
[100, 2, 3, 4, 5]
```

$a = [1, 2, 3, 4, 5]$

$a[0] = 100$



Örneğin:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> a[0] = 200
>>> a
[200, 2, 3]
>>> b
[1, 2, 3]
```

Python'da daha önce de belirttiğimiz gibi tüm atama işlemleri bir adres ataması anlamındadır. Yukarıdaki örnekte biz $a[0]$ 'a yeni bir içerisinde 200 olan int nesnenin adresini atadık. Ancak bundan $b[0]$ etkilenmemektedir.

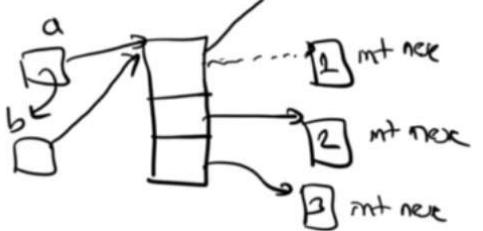
Kopyalama yerine doğrudan listeleri birbirine atasaydık iki değişken aynı listeyi gösteriyor olacaktı:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[0] = 200
>>> a
[200, 2, 3]
>>> b
[200, 2, 3]
>>> id(a)
2800657016008
>>> id(b)
2800657016008
```

Bu durumu çizimle şöyle açıklayabiliriz:

$a = [1, 2, 3]$

$b = a$

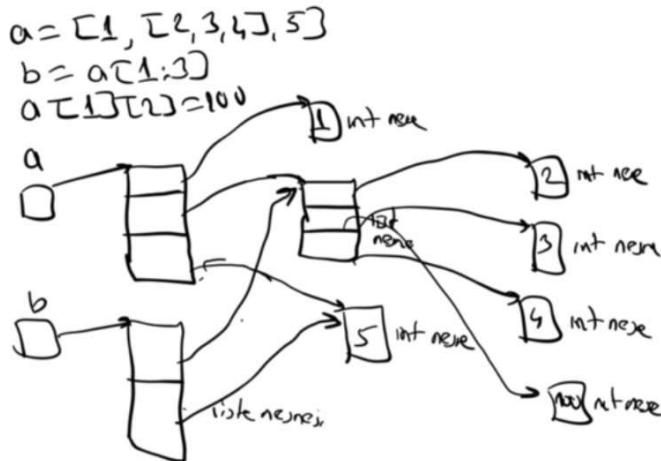


a ve b değişkenleri aynı listeyi gösteriyorsa biz listenin i 'inci elemanına $a[i]$ ya da $b[i]$ ifadesiyle erişebiliriz. İkisi arasında hiçbir farklılık yoktur.

Listelerin değiştirilebilir olması bazı durumlarda önemli olmaktadır. Örneğin:

```
>>> a = [1, [2, 3, 4], 5]
>>> b = a[1:3]
>>> a[1][2] = 100
>>> a
[1, [2, 3, 100], 5]
>>> b
[[2, 3, 100], 5]
```

Burada `b = a[1:3]` işlemi ile biz `a`'nın içerisindeki değerleri kopyalamış olduk. Bunların bir tanesi de diğer listedir. Bu durumda `a[1]` ile `b[0]` aynı alt liste nesnesini göstermektedir. Listeler değiştirilebilir olduğuna göre bu değişiklikten hem `a` hem de `b` etkilenmiştir. Şekilsel olarak durumu şöyle açıklayabiliriz:



İki liste '+' operatörüyle toplanabilir. Bu durumda yeni bir liste elde yaratılır. Bu yeni listenin elemanları iki listenin elemanlarının birleşiminden oluşur. Örneğin:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

Örneğin:

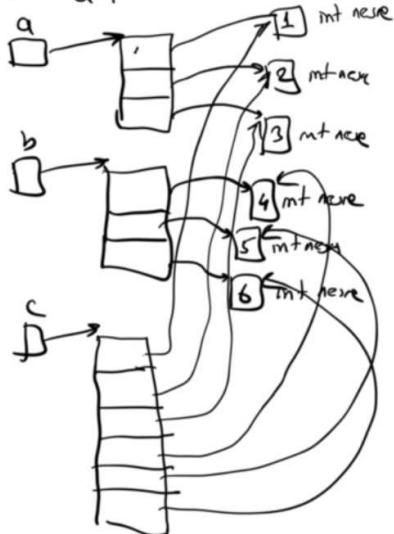
```
>>> [10, 20, 'Ali'] + ['Veli', 10, 'Selami']
[10, 20, 'Ali', 'Veli', 10, 'Selami']
```

Toplama işlemi sırasında iki listenin uzunlıklarının toplamı uzunlığında yeni bir liste oluşturulmaktadır. Bu yeni listeye bu iki listenin elemanları kopyalanmaktadır. Tabii listelerin elemanları aslında birer adres belirttiğine göre kopyalanan şeyler de aslında bu adreslerdir. Bu işlemi şeşkisel olarak söyleyebiliriz:

$a = [1, 2, 3]$

$b = [4, 5, 6]$

$c = a + b$



Örneğin:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> id(a[0])
1502765280
>>> id(c[0])
1502765280
```

Örneğin:

```
>>> a = [1, [2, 3]]
>>> b = [4, 5]
>>> c = a + b
>>> c
[1, [2, 3], 4, 5]
>>> a[1][0] = 100
>>> c
[1, [100, 3], 4, 5]
```

Python'da `+=` operatörü listelerde "sona ekleme" anlamına gelmektedir. Örneğin:

```
>>> a = [1, 2, 3]
>>> id(a)
1383848916936
>>> a += [3, 4]
>>> id(a)
1383848916936
>>> a
[1, 2, 3, 3, 4]
```

Bu anlamda Python'da listeler söz konusu olduğunda $a += b$ ile $a = a + b$ aynı anlama gelmemektedir. Listelerin sonuna eleman eklenmesi list sınıfının metotlarıyla da yapılabilir. Bu konu ileride ele alınmaktadır.

Listelerde `*` operatörü "tekrarlama (repetition)" anlamına gelmektedir. a bir liste n de int türden bir değer belirtmek üzere $a * n$ ya da $n * a$ işlemi a listesinin n defa kendisiyle eklenmesi anlamına gelmektedir. Örneğin $a * 3$ işlemi $a + a + a$ işlemi ile eşdeğerdir. Örneğin:

```

>>> a = [1, 2, 3]
>>> b = a * 3
>>> b
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> b = a + a + a
>>> b
[1, 2, 3, 1, 2, 3, 1, 2, 3]

```

* operatörünün listenin her elemanı ile ilgili çarpma işlemi yapmadığına aynı listeyi üç uça n defa eklediğine dikkat ediniz. Listede tekrarlama yapılırken tekrar sayısının int türden olması zorunludur. Bir liste yalnızca int bir değerle çarpılabilir. Tabii çarpmanın değişme özelliği burada da vardır. Yani listenin ve int türden operandın '*' operatörünün neresinde olduğunun bir önemi yoktur. Bu durumda a bir liste ve n bir int değer olmak üzere a * n ya da n * a işlemi tamamen a + a + a... biçiminde a'nın n defa toplanması ile aynı anlamdadır.

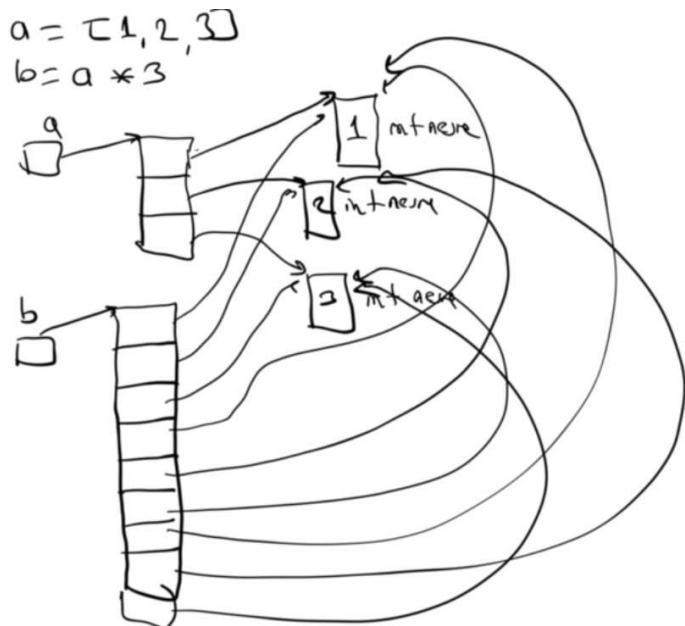
Bir listeyi negatif bir değerle çarpmaya çalışırsak boş bir liste elde ederiz. Örneğin:

```

>>> a * -1
[]

```

Listelerde tekrarlama işlemi de yine kopyalama yoluyla yapılmaktadır. Yani a bir liste ve n de bir tamsayı belirtmek üzere a * n ya da n * a işleminde önce len(a) * n uzunlığında yeni bir liste yaratılır. Sonra a'nın elemanları (yani içerisindeki adresler) n defa bu yeni listeye kopyalanır. Örneğin:



Benzer biçimde yine a bir liste belirtmek üzere a *= n işlemi listenin sonuna aynı listenin elemanlarını n - 1 defa ekleme anlamına gelir. Bu anlamda a *= n işlemi a = a * n işleminden farklıdır. Örneğin:

```

>>> a = [1, 2, 3]
>>> id(a)
1383849166088
>>> a *= 4
>>> id(a)
1383849166088
>>> a
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

```

Yani başka bir deyişle a *= n işlemi n - 1 defa a += n anlamına gelmektedir.

Şimdiye kadar biz listeleri hep [...] sentaksıyla yarattık. Daha önceden de belirttiğimiz gibi aslında listeler "list" isimli bir sınıfıyla temsil edilmektedir. list sınıfının list isimli tür fonksiyonu ile de listeler yaratılabilir. list built-in bir sınıfıdır.

Dolayısıyla list fonksiyonu da built-in bir sonksiyondur. Bu fonksiyon argümansız olarak çağrılırsa boş bir liste yaratılır. Örneğin:

```
>>> a = list()
>>> a
[]
>>> len(a)
0
```

Fonksiyona biz "dolaşılabilir (iterable)" bir nesneyi de argüman olarak verebiliriz. Bu durumda fonksiyon o dolaşılabilir nesnenin elemanlarından yeni bir list oluşturur. String'ler ve listeler dolaşılabilir nesnelerdir. Bir string dolaşıldığında onun tek tek karakterleri birer string olarak elde edilmektedir. Bir liste dolaşıldığında ise listenin elemanları sırasıyla elde edilmektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = list(a)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
```

Burada biz list fonksiyonuyla a'nın elemanlarından oluşan yeni bir liste yarattık.

Stringler de dolaşılabilir nesneler olduğuna göre biz bir string'in karakterlerinden bir liste oluşturabiliriz. Örneğin:

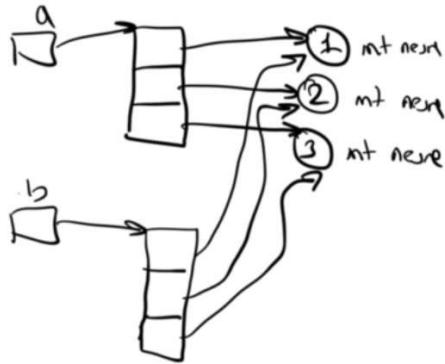
```
>>> a = list('Ankara')
>>> a
['A', 'n', 'k', 'a', 'r', 'a']
```

list fonksiyonuyla dolaşılabilir bir nesne yoluyla yeni bir liste yaratıldığında bu yeni liste de yine sağlam kopyalama yoluyla oluşturulmaktadır. Örneğin:

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> id(a)
1383849167496
>>> id(b)
1383849175752
>>> id(a[0])
140714104443936
>>> id(b[0])
140714104443936
```

Burada aslında şöyle bir işlem gerçekleşmektedir:

```
a = [1, 2, 3]
b = list(a)
```



Burada yapılan işlem işlevsel olarak aşağıdaki ile eşdeğerdir:

```
>>> a = [1, 2, 3]
>>> b = a[:]
```

Dilimleme Yoluyla Liste Elemanlarının Güncellenmesi

Dilimleme yoluyla liste elemanları güncellenebilir. Bu durumda önce dilimlenen elemanlar silinir. Sonra silinen indeksinden itibaren dilimlenen elemanlar listeye insert edilir. Yani dilimleme yoluyla işlem yaparken bu işlem hem silme hem de insert etme işlemlerini barındırmaktadır. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:4] = [100, 200, 300]
>>> a
[1, 2, 100, 200, 300, 5]
```

Burada önce a[2:4] elemanları silinmiş sonra da bunların yerine [100, 200, 300] elemanları insert edilmiştir. Silinen ve insert edilen kısmın eşit uzunlukta olmasının gerekliliğine dikkat ediniz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:4] = [100]
>>> a
[1, 2, 100, 5]
```

Burada listenin 2 ve 3 indeksli elemanları silinmiş yerine 100 değeri insert edilmiştir. Dilimlenmiş kısma boş liste atamaya çalışırsak o elemanları silmiş oluruz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[1:3] = []
>>> a
[1, 4, 5]
```

Hiç eleman silmeden de insert işlemi yapabiliriz. Bunun için dilimlemede start değerinin stop değerine eşit ya da ondan daha küçük olması gereklidir. Bu durumda bir eleman seçilemeyeceği için yalnızca insert işlemi yapılacaktır. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:2] = [100, 200]
>>> a
[1, 2, 100, 200, 3, 4, 5]
```

Insert işleminin her zaman start indeksine (yani insert edilenler o indekste olacak biçimde) yapıldığına dikkat ediniz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:1] = [100]
>>> a
[1, 2, 100, 3, 4, 5]
```

Dilimleme yoluyla atama işleminde aslında atanacak değer bir liste olmak zorunda değildir. Ancak dolaşılabilir (iterable) bir nesne olmak zorundadır. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:4] = 100
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    a[2:4] = 100
TypeError: can only assign an iterable
```

Stringler de dolaşılabilir (iterable) nesneler olduğuna göre biz dilimleme ifadesine bir string'i de atayabiliriz. Bu durumda aslında stringin karakterleri tek tek listeye yerleştirilecektir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[1:1] = 'Ali'
>>> a
[1, 'A', 'l', 'i', 2, 3, 4, 5]
```

Tabii biz 'Ali' yazısını karakter karakter değil de bir bütün olarak insert etmek istersek onu bir listenin elemanı olarak atama işlemine sokmamız gereklidir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[1:1] = ['Ali']
>>> a
[1, 'Ali', 2, 3, 4, 5]
```

Şimdi de listenin başına insert işlemi yapalım:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0:0] = [100]
>>> a[:0] = [200]
>>> a
[200, 100, 1, 2, 3, 4, 5]
```

Şimdi de sonuna yapalım:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[len(a):len(a)] = [100]
>>> a[len(a):] = [200]
>>> a
[1, 2, 3, 4, 5, 100, 200]
>>> a = [1, 2, 3, 4, 5]
>>> a[200:] = [100]
>>> a
[1, 2, 3, 4, 5, 100]
```

Liste elemanlarının dilimleme yoluyla güncellendiği durumda dilimlemede step değeri verilmişse atanacak listenin tam seçilen eleman sayısı kadar olması gerekmektedir. Örneğin:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[0:8:2] = [10, 20, 30]
Traceback (most recent call last):
  File "<pyshell#170>", line 1, in <module>
    a[0:8:2] = [10, 20, 30]
```

```
ValueError: attempt to assign sequence of size 3 to extended slice of size 4
```

Halbuki örneğin:

```
>>> a[0:8:2] = [10, 20, 30, 40]
>>> a
[10, 1, 20, 3, 30, 5, 40, 7, 8, 9]
```

Sona ekleme işlemini daha önce de ele aldığımız gibi += ya da *= operatörüyle de yapabiliriz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a += [100, 200]
>>> a
[1, 2, 3, 4, 5, 100, 200]
```

Benzer biçimde dolaşılabilir nesnedeki değerler ters sırada da listeye atanabilirler. Örneğin:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[:5:-1] = [100, 200, 300, 400]
>>> a
[1, 2, 3, 4, 5, 6, 400, 300, 200, 100]
```

Burada da step değeri belirtildiği için dolaşılabilir nesnedeki eleman sayısının seçilen eleman sayısıyla aynı olması gerekmektedir.

Listelerde in ve not in Operatörleri

in operatörü yalnızca listelerde değil diğer biçimlerdeki dizilimlerde (sequences) de kullanılan iki operandı araeğenel bir operatördür. Belli bir değerin dizilimde olup olmadığını kontrol eder. Eğer ilgili değer dizilimde varsa True değerini yoksa False değerini üretir. not in operatörü de tam ters işlem yapmaktadır. Örneğin:

```
>>> a = [3, 13, 18, 'Ali', 4.38]
>>> 13 in a
True
>>> 23 in a
False
>>> 'Ali' in a
True
>>> 'Veli' in a
False
>>> 'Veli' not in a
True
```

Örneğin:

```
>>> a = [10, [20, 30], 40]
>>> 20 in a
False
>>> [20, 30] in a
True
>>> [20] in a
False
>>> a = [10, [20, 30], 40]
>>> 10 in a
True
>>> 20 in a
False
>>> [20, 30] in a
True
>>> [20] in a
```

False

List Sınıfının Metotları

Python'da aslında bütün türler birer sınıf (class) belirtmektedir. Python'da sınıfın içerisinde olmayan alt programlara fonksiyon, sınıfın içerisinde bulunan alt programlara ise metot denilmektedir. Daha önceden de belirtildiği gibi bir sınıfın metodu (yani sınıfın içindeki fonksiyonu) o sınıf türünden değişken ve '.' operatörüyle çağrılır. Örneğin:

```
a.foo()
```

Burada biz a değişkeni hangi sınıf türündense o sınıfın foo metodunu çağrıyoruz. Halbuki:

```
foo()
```

Burada hiçbir sınıfın içerisinde olmayan global foo fonksiyonu çağrılmaktadır.

list sınıfının append metodu listenin sonuna yeni tek bir elemanı eklemek için kullanılır. Örneğin:

```
>>> a = [1, 2, 3]
>>> a.append(10)
>>> a
[1, 2, 3, 10]
```

Örneğin:

```
>>> a = list(range(1, 11))
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a.append(100)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100]
```

append metodunun eklemeyi çağrıldığı liste üzerinde yaptığına dikkat ediniz. append metodu tek bir elemanı sona eklemektedir. Bu tek eleman bir liste olsa bile eleman bir liste olarak listenin sonuna eklenir. Örneğin:

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.append([10, 20, 30])
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, [10, 20, 30]]
```

append metoduna benzeyen extend isimli bir metot da vardır. Bu metot argüman olarak dolaşılabilir (iterable) bir nesne alır ve bu dolaşılabilir nesnenin tüm elemanlarını listenin sonuna ekler. Örneğin:

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.extend([10, 20, 30])
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]
```

Örneğin:

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.extend('Ali')
>>> a
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'l', 'i']
>>> a.extend(range(100, 110))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'l', 'i', 100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
```

extend metodunun dolaşılabilir bir nesne istedigine dikkat ediniz. Örneğin:

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.extend(10)
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    a.extend(10)
TypeError: 'int' object is not iterable
```

Listelerde += operatörü extend ile aynı şeyi yapmaktadır. += operatörünün sağındaki operand tipki extend metodunda olduğu gibi dolaşılabilir bir nesne olabilmektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> id(a)
4491743296
>>> a += range(10, 20, 2)
>>> a
[1, 2, 3, 4, 5, 10, 12, 14, 16, 18]
>>> id(a)
4491743296
```

insert metodu belli bir indekse tek bir eleman ekler. Birinci parametre eklenecek elemanın bulunacağı indeksi ikinci parametre eklenecek elemanı belirtmektedir. Eklenecek eleman o indekste olacak biçimde diğerleri kaydırılır. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a.insert(3, 100)
>>> a
[1, 2, 3, 100, 4, 5]
```

insert metodunda negatif indeksler de kullanılabilmektedir. Bu negatif indeksler yine listenin uzunluğu ile toplanır. Örneğin:

```
>>> a = [10, 20, 30]
>>> a.insert(-1, 100)
>>> a
[10, 20, 100, 30]
```

insert metodunda etkin indeks sınır dışında kalıyorsa exception oluşmaz. Başa ya da sona insert işlemi uygulanır. Örneğin:

```
>>> a = [10, 20, 30]
>>> a.insert(100, 'ali')
>>> a
[10, 20, 30, 'ali']
>>> a.insert(-100, 'veli')
>>> a
['veli', 10, 20, 30, 'ali']
```

pop metodu argümansız ya da tek argümanlı biçiminde kullanılabilir. Metot argümansız kullanıldığında listenin son elemanını listeden siler. Argümanlı kullanıldığında belirtilen indeksteki elemanı silmektedir. pop metodu aynı zamanda silinen elemanı geri dönüş değeri olarak verir. Örneğin:

```
>>> a.pop()
```

```
5
>>> a
[1, 2, 3, 4]
```

pop metoduna da negatif indeks değeri girilebilir. Örneğin:

```
>>> a = [10, 20, 30]
>>> a.pop(-1)
30
>>> a
[10, 20]
>>>
```

clear metodu parametresizdir. Listedeki tüm elemanları siler. Örneğin:

```
>>> a = [10, 20, 30]
>>> a.clear()
>>> a
[]
```

clear metodu ile listenin tüm elemanları silindiğinde nesnenin id değerinin değişmediğine dikkat ediniz:

```
>>> a = [10, 20, 30, 40, 50]
>>> a
[10, 20, 30, 40, 50]
>>> id(a)
2185224789768
>>> a.clear()
>>> a
[]
>>> id(a)
2185224789768
>>> a.extend(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> id(a)
2185224789768
```

remove metodu bir elemanı arar, onu bulursa siler. Örneğin:

```
>>> a = [10, 20, 'Ali', 'Selami', 30]
>>> a.remove('Ali')
>>> a
[10, 20, 'Selami', 30]
>>> a.remove(20)
>>> a
[10, 'Selami', 30]
>>> a.remove('xxx')
```

Eleman birden fazla yerde varsa remove ilk bulduğunu silmektedir. Eğer ilgili eleman yoksa exception oluşur. Örneğin:

```
>>> a.remove('xxx')
Traceback (most recent call last):
  File "<pyshell#157>", line 1, in <module>
    a.remove('xxx')
ValueError: list.remove(x): x not in list
```

index metodu belli bir elemanı arar. Onu bulursa bize onu ilk bulduğu yerin indeksini verir. Bulamazsa exception oluşur. Örneğin:

```
>>> a = [10, 20, 30, 'Ali', 'Veli', 40, 'Selami']
```

```
>>> a.index(30)
2
```

index metodunu istersek iki ya da üç argümanla da çağrılabiliriz. Bu argümanlar bu işlemi listenin belli kısmında yapmak için kullanılmaktadır. Eğer metot iki argümanla çağrılırsa birinci argüman aranacak değeri ikinci argüman da aramanın listenin kaçinci indeksinden başlatılacağını belirtir. Örneğin:

```
>>> a = [10, 20, 30, 'Ali', 'Veli', 40, 'Selami']
>>> a.index(40, 4)
5
```

Eğer index metodu üç argümanlı olarak çağrılsa ikinci argüman aramanın başlatılacağı indeksi, üçüncü argüman da aramanın bitirileceği indeksi belirtir. Ancak bitim indeksi aramaya dahil değildir. İkinci ve üçüncü argümanları index ':' operatörünün sol tarafındaki ve sağ tarafındaki operand gibi değerlendirebilirsiniz. Örneğin:

```
>>> a = [10, 20, 30, 'Ali', 'Veli', 40, 'Selami']
>>> a.index(40, 2, 5)
Traceback (most recent call last):
  File "<pyshell#166>", line 1, in <module>
    a.index(40, 2, 5)
ValueError: 40 is not in list
```

Yine index metodunda başlangış. ve bitiş index numaraları negatif olarak da verilebilir. Örneğin aramayı bu sayede listenin son 5 elemanında yapabiliriz:

```
>>> a = [3, 7, 8, 60, 'ali', 72, 42, 'veli', 86, 32]
>>> a.index('veli', 5)
7
```

count metodu belli bir elemandan listede kaç tane olduğunu bize verir. Örneğin:

```
>>> a = [1, 2, 2, 5, 2, 9]
>>> a.count(2)
3
```

sort metodu liste içerisindeki elemanları sıraya dizer. Fonksiyonu tek bir argümanla çağrırsak liste küçükten büyüğe sıraya (ascending) dizilmektedir. Örneğin:

```
>>> a = [13, 4, 43, 21, 45, 78]
>>> a.sort()
>>> a
[4, 13, 21, 43, 45, 78]
```

Tabii türler arasında karşılaştırma mümkün değilse sort fonksiyonu exception'a yol açmaktadır. Bu konunun ayrıntıları sınıflar kısmında ele alınacaktır. Örneğin:

```
>>> a = [23, 'Veli', 54, 'Ali', 12, 7, 9]
>>> a.sort()
Traceback (most recent call last):
  File "<pyshell#175>", line 1, in <module>
    a.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

İlleride de göreceğimiz gibi string'ler karşılaştırılabilirmektedir. Dolayısıyla biz string'lerden oluşan bir listeyi sıraya dizebiliriz. Örneğin:

```
>>> names = ['selami', 'ali', 'veli', 'sacit', 'burhan']
>>> names
['selami', 'ali', 'veli', 'sacit', 'burhan']
>>> names.sort()
```

```
>>> names  
['ali', 'burhan', 'sacit', 'selami', 'veli']
```

Tabii stirng'ler belli bir dile göre değil UNICODE tabloya göre karşılaştırılmaktadır. Türkçe yazıların sıraya dizilmesi bu nedenle beklediğiniz gibi olmayabilir. Örneğin:

```
>>> names = ['şamil', 'ilayda', 'ahmet', 'çelik', 'mansur']  
>>> names  
['şamil', 'ilayda', 'ahmet', 'çelik', 'mansur']  
>>> names.sort()  
>>> names  
['ahmet', 'ilayda', 'mansur', 'çelik', 'şamil']
```

sort metodu varsayılan durumda küçükten büyüğe (ascending) sıraya dizme uygulamaktadır. Büyüktен küçüğe (descending) sıraya dizme için sort metodunun reverse isimli parametresi True girilmelidir. Python'da bir argüman girilirken o argümanın hangi parametre için girildiği de belirtilebilmektedir. Böyle girilen argümanlara "isimli argümanlar (keyword arguments)" denilmektedir. Isimli argümanlar girilirken önce parametrenin ismi sonra '=' karakteri sonra da argüman değeri yazılır. Isimli argümanlar sonraki bölümlerde ayrıntılı bir biçimde ele alınmaktadır. İşte büyuktenten küçüğe sıraya dizme işlemi için sort metodunda da reverse parametresi "reverse=True" biçiminde girilmelidir. Örneğin:

```
>>> a = [23, 54, 12, 7, 9]  
>>> a.sort(reverse=True)  
>>> a  
[54, 23, 12, 9, 7]
```

Tabii biz reverse parametresini False olarak da girebiliriz. Ancak varsayılan durum zaten böyledir. Aslında sort metodunun listenin her değerini bir fonksiyona sokup o fonksiyonun sonucuna göre sıraya dizemeye bir key parametresi vardır. Örneğin:

```
>>> names = ['Selami', 'Ali', 'Veli', 'Ayşe', 'Fatma']  
>>> names.sort(key=len)  
>>> names  
['Ali', 'Veli', 'Ayşe', 'Fatma', 'Selami']
```

Yukarıda da belirtildiği gibi stringler için default sıralama UNICODE tabloya göre yapılmaktadır. Fakat örneğin:

```
>>> names = ['ALİ', 'veli', 'Selami', 'ayşe', 'FATMA']  
>>> names.sort(key = str.lower)  
>>> names  
['ALİ', 'ayşe', 'FATMA', 'Selami', 'veli']
```

sort metodunun dışında ayrıca bir de global built-in sorted isimli fonksiyon vardır. Bu fonksiyon dolaşılabılır bir nesneyi parametre olarak alır. Fakat o nesneyi sıraya dizmez. Yani sorted fonksiyonu in-place sıraya dizme yapmamaktadır. Sıraya dizilmiş yeni bir listeyi bize vermektedir. Örneğin:

```
>>> x = [34, 23, 1, 17, 45]  
>>> y = sorted(x)  
>>> y  
[1, 17, 23, 34, 45]  
>>> x  
[34, 23, 1, 17, 45]
```

Örneğin:

```
>>> sorted('istanbul')  
['a', 'b', 'i', 'l', 'n', 's', 't', 'u']
```

Yine bu fonksiyonun reverse ve key isimli parametreleri vardır. Örneğin:

```
>>> y = sorted(x, reverse=True)
>>> y
[45, 34, 23, 17, 1]
```

reverse metodu parametresizdir. Liste içerisindeki elemanları ters yüz eder. Örneğin:

```
>>> a = [3, 6, 2, 9]
>>> a.reverse()
>>> a
[9, 2, 6, 3]
```

Ayrıca reverse metodunun yanı sıra reversed isimli global built-in bir fonksiyon da vardır. Bu fonksiyon dolaşılabilir bir nesneyi parametre olarak alır. Bize ters yüz edilmiş yeni bir dolaşım nesnesi verir. Tabii biz bu dolaşım nesnesini list fonksiyonuyla listeye dönüştürülebiliriz. Örneğin:

```
>>> x = [1, 2, 3, 4, 5]
>>> y = list(reversed(x))
>>> x
[1, 2, 3, 4, 5]
>>> y
[5, 4, 3, 2, 1]
```

Örneğin:

```
>>> list(reversed(range(10)))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Burada range fonksiyonu bize dolaşılabilir bir nesne vermektedir. Bu dolaşılabilir nesne reversed fonksiyonuna parametre olarak geçirilmiştir. reversed fonksiyonu da bize tersten dolaşım yapacak bir dolaşım nesnesi vermiştir. Nihayet list fonksiyonuyla da reversed fonksiyonunun verdiği nesne dolaşarak ondan bir liste elde edilmiştir.

Burada bir ayrıntıdan bahsetmek istiyoruz. Aslında reversed fonksiyonuyla her türlü dolaşılabilir nesne tersten dolaşılamamaktadır. Dolaşılabilir bir nesnenin tersten dolaşılabilmesi için dolaşılabilir nesneye ilişkin sınıfın reversed isimli bir metoda sahip olması gerekmektedir. Nesnelerin tersten dolaşılması ileride ayrı bir başlık halinde ayrıntılarıyla açıklanmaktadır.

copy metodu nesnenin özdeş kopyasından oluşturur. Yani bu metot bize elemanları aynı olan yeni bir liste vermektedir. Örneğin:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> id(a)
1593657691080
>>> id(b)
1593657693320
>>> id(a[0])
140724931253280
>>> id(b[0])
140724931253280
```

Örneğin:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
```

```

>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> a.append(100)
>>> a
[1, 2, 3, 100]
>>> b
[1, 2, 3]

```

copy işleminin sonucunda a ve b içeriği aynı fakat farklı iki nesneyi gösteriyor durumda olur. copy metoduyla kopya çıkarmakla dilimleme yoluyla kopya çıkarmak arasında bir fark yoktur. İki yöntem de daha önceden açıklandığı gibi sığ kopyalama (shallow copy) yapmaktadır. Örneğin:

```

>>> a = [1, 2, 3, 4, 5]
>>> b = a[:]
>>> c = a.copy()
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>> c
[1, 2, 3, 4, 5]

```

Pek çok programlama dilinde çok boyutlu diziler vardır. Örneğin iki boyutlu diziler yani matrisler pek çok dilde özel bir sentaksla desteklenmektedir. Halbuki Python'da çok boyutlu dizi diye bir kavram yoktur. Listeler zaten çok boyutlu dizi olarak da kullanılabilirler . Örneğin:

```
>>> m = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Burada aslında m listesi 4x3'lük bir matris gibidir. Onun istediğimiz elemanına yan yana iki köşeli parantezle erişebiliriz. Örneğin:

```

>>> m[3][2]
12
>>> m[1][0]
4

```

Örneğin:

```

>>> a = [[10, 20], [5, 3], [5, 8]]
>>> a.sort()
>>> a
[[5, 3], [5, 8], [10, 20]]

```

Burada listenin elemanlarının da aslında list türünden olduğuna dikkat ediniz.

Demetler (Tuples)

Python'da demetler bir grup bilgiyi tutmak için kullanılan listelere oldukça benzer veri yapılarıdır. Listeler için geçerli olan pek çok özellik demetler için de geçerlidir. Python'da elemanlarına köşeli parantez operatörüyle indeks belirtilerek erişilebilen listeler, demetler, string'ler gibi veri yapılarına "dizimsel veri yapıları (sequence container)" denilmektedir.

Bir demet tipik olarak normal parantezlerin içerisinde ',' atomlarıyla ayrılmış ifadeler girilerek oluşturulur. Örneğin:

```

>>> t = (10, 'Ali', 12.4)
>>> t
(10, 'Ali', 12.4)
>>> type(t)

```

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

Demetler de tıpkı listeler gibi farklı türden elemanlara sahip olabilirler. Örneğin bir demetin bir elemanı bir liste olabilir:

```
>>> t = (1, [2, 3, 4], 'Ali')
>>> t
(1, [2, 3, 4], 'Ali')
```

Tabii bir listenin elemanı bir demet de olabilir:

```
>>> a = [1, (2, 3, 4), 'Ali']
>>> a
[1, (2, 3, 4), 'Ali']
```

Elemanlar demetin içerisinde sıralı bir biçimde tutulduğu için onlara listelerde olduğu gibi [...] operatörüyle erişebiliriz. Örneğin:

```
>>> t = (10, 20, 30)
>>> t[0]
10
>>> t[1]
20
>>> t[2]
30
```

Yine listelerde olduğu gibi demetlerde de dilimleme yoluyla alt demetler elde edilebilmektedir. Örneğin:

```
>>> t = ('Ali', 123, 34.7)
>>> t
('Ali', 123, 34.7)
>>> t[1]
123
>>> t[1:3]
(123, 34.7)
```

Demetlerde listelerde olduğu gibi negatif indeksleme yapılmamaktadır. Örneğin:

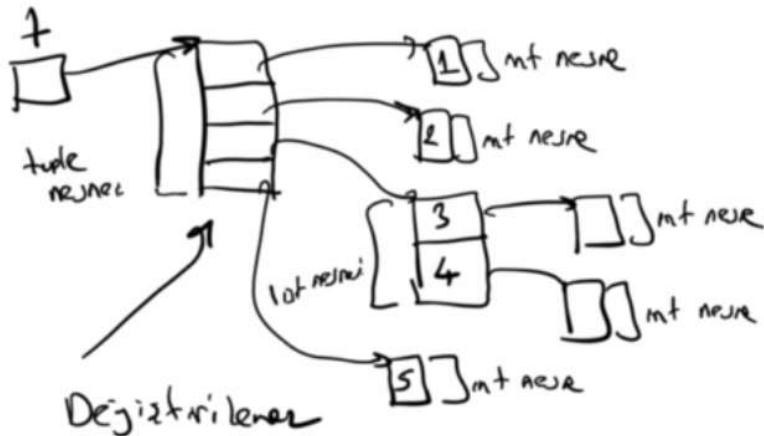
```
>>> t = (10, 20, 30, 40, 50)
>>> t[-2]
40
>>> t[1:-2]
(20, 30)
```

Demetlerin listelerden en önemli farkı demetlerin değiştirilemez (immutable) olmasıdır. Biz bir demeti yaratırken onun elemanlarını yaratım sırasında belirleriz. Daha sonra onun elemanları üzerinde değişiklikler yapamayız. Örneğin:

```
>>> t = (10, 20, 30, 40, 50)
>>> t[1] = 'Ali'
Traceback (most recent call last):
  File "<pyshell#346>", line 1, in <module>
    t[1] = 'Ali'
TypeError: 'tuple' object does not support item assignment
```

Tabii demetin elemanı bir liste ise bu durumda demetin kendisi değiştirilemezken onun elemanı olan liste üzerinde değişiklikler yapılabilir. Aşağıdaki kodu ve şekli inceleyiniz:

$t = (1, 2, [3, 4], 5)$



Burada demetin elemanları değiştirilemez. Dolayısıyla örneğin biz onun 2'inci indeksli elemanına yeni bir nesne atayamayız:

```
>>> t = (1, 2, [3, 4], 5)
>>> t
(1, 2, [3, 4], 5)
>>> t[2] = [6, 7]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    t[2] = [6, 7]
TypeError: 'tuple' object does not support item assignment
```

Ancak biz bu örnekte demetin 2'inci indeksli elemanı bir liste olduğu için o liste üzerinde değişiklikler yapabiliriz:

```
>>> t[2][0] = 100
>>> t
(1, 2, [100, 4], 5)
```

Bizim burada demetin değil listenin elemanını değiştirdiğimize dikkat ediniz. Demetin elemanları yine aynı liste nesnesini (yani aynı adresi) gösteriyor durumdadır. Örneğin:

```
>>> a = (10, [20, 30, 40], 50)
>>> a[1].append(100)
>>> a
(10, [20, 30, 40, 100], 50)
```

Biz burada aslında elemanı demete eklemedik. Demetin 1'inci indeksli elemanın gösterdiği listeye ekledik.

Demetler de dolaşılabilir (iterable) nesnelerdir. Böylece biz örneğin list fonksiyonu ile bir demeti dolaşarak onun elemanlarından bir liste elde edebiliriz:

```
>>> t = ('Ali', 123, 'Veli', 12.4)
>>> a = list(t)
>>> t
('Ali', 123, 'Veli', 12.4)
>>> a
['Ali', 123, 'Veli', 12.4]
```

Tabii bu işlem yine sağlam kopyalama yoluyla yapılmaktadır. Yani demetin elemanlarındaki adresler listenin elemanlarına kopyalanmaktadır. Örneğin:

```
>>> t = (1, 2, [3, 4], 5)
```

```
>>> a = list(t)
>>> id(t[0])
2597868339440
>>> id(a[0])
2597868339440
>>> t
(1, 2, [3, 4], 5)
>>> a
[1, 2, [3, 4], 5]
>>> a[2][0] = 100
>>> t
(1, 2, [100, 4], 5)
>>> a
[1, 2, [100, 4], 5]
```

Demetleri biz tuple sınıfının tür fonksiyonu olan tuple fonksiyonu ile de oluşturabiliriz. Tıpkı list fonksiyonunda olduğu gibi tuple fonksiyonu da bizden dolaşılabilir (iterable) bir nesne alıp onun elemanlarından bir demet yapmaktadır. Örneğin:

```
>>> t = tuple('ankara')
>>> t
('a', 'n', 'k', 'a', 'r', 'a')
```

Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> t = tuple(a)
>>> a
[1, 2, 3, 4, 5]
>>> t
(1, 2, 3, 4, 5)
```

Tabii burada da yine sig kopyalama yapılmaktadır. Bu durumda biz listenin bir elemanına yeni bir nesne atadığımızda bundan demet etkilenemeyecektir. Örneğin:

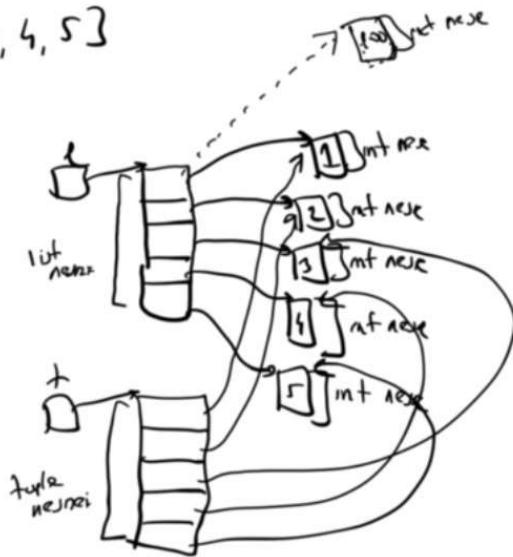
```
>>> a = [1, 2, 3, 4, 5]
>>> t = tuple(a)
>>> t[0] = 100
>>> a
[100, 2, 3, 4, 5]
>>> t
(1, 2, 3, 4, 5)
```

Bunu çizimle şöyle gösterebiliriz:

$l = [1, 2, 3, 4, 5]$

$t = tuple(l)$

$l[0] = 100$



Boş bir demet de söz konusu olabilir. Boş bir demeti parantezlerin içerisinde boş bırakarak yaratabiliriz. Örneğin:

```
>>> t = ()  
>>> t  
()
```

tuple fonksiyonunu argümansız olarak çağırarak da boş liste elde edilebilmektedir:

```
>>> t = tuple()  
>>> t  
()
```

Tek elemanlı bir demet de söz konusu olabilir. Tek elemanlı demetleri yaratırken parantezler içerisinde elemandan sonra ekstra bir ',' atomunu da bulundurmak gereklidir. Örneğin:

```
>>> t = (100, )  
>>> t  
(100,)
```

Burada ',' atomunu kullanmazsa parantezler demet parantezi olarak değil öncelik parantezi olarak ele alınmaktadır. Örneğin:

```
>>> t = (100)  
>>> t  
100  
>>> type(t)  
<class 'int'>
```

Buradaki (100) ifadesi tek elemanlı bir demet anlamına gelmemektedir. Parantez içerisinde 100 anlamına gelmektedir. İşte bu nedenle tek elemanlı demetler belirtilirken elemandan sonra ayrıca bir ',' atomunun bulundurulması gerekmektedir. Örneğin:

Built-in len fonksiyonu demetlere de uygulanabilir. Bu durumda demetin eleman sayısı elde edilir. Örneğin:

```
>>> t = (1, 2, 'Ali', 23.4, [3, 4, 5])  
>>> len(t)  
5
```

Aslında demet oluştururken birkaç durum dışında parantezler hiç kullanılmayabilir. Yani biz parantezler olmadan ifadelerin arasına yalnızca ',' atomu yerleştirerek de demetleri oluşturulabiliriz. Örneğin:

```
>>> t = 'Ali', 123, 'Veli', 34.6
>>> t
('Ali', 123, 'Veli', 34.6)
```

Örneğin:

```
>>> t = 'Ali',
>>> t
('Ali',)
```

Fakat bazı durumlarda '' atomu başka anlamlara geldiği için bu tür durumlarda demet belirtmek için parantezler mecburen kullanılmak zorundadır. Örneğin:

```
>>> print(10, 20)
10 20
```

Buradaki ',' argüman ayıracı olan '' atomudur. Yani buradaki '' 10 ve 20'yi bir demet yapmaz. Dolayısıyla burada 10 ve 20 değerleri ayrı ayrı yazdırılmaktadır. Eğer fonksiyona geçen argümanın demet olması isteniyorsa bunu sağlamak için parantezlerin mecburen kullanılması gereklidir:

```
>>> print((10, 20))
(10, 20)
```

Benzer biçimde:

```
a = [1, 2, 3, 4, 5]
```

Buradaki '' atomları da liste ayıracı olan '' atomlarıdır. Eğer listenin bir elemanı demet olacaksa onun ayrıca yine parantezlere alınması gereklidir. Örneğin:

```
a = [1, 2, (3, 4), 5]
```

Örneğin:

```
a = (10, 20), 30, 40
```

Burada ilk elemanı bir demet olan bir demet oluşturulmuştur. İlk elemanı demet yapmak için parantezlerin kullanılması gerekiğine dikkat ediniz.

Demetler değiştirilebilir nesneler olmadığı için append, insert, remove, erase gibi metodlara sahip değildir. tuple sınıfının yalnızca iki metodu vardır: index ve count. Bu metodları list sınıfından da anımsıyorsunuz. index metodу demet içerisinde bir elemanı arar, eğer onu bulursa ilk bulduğu yerin indeks numarası ile geri döner. Örneğin:

```
>>> a = (10, 20, 30, 40, 30)
>>> a.index(30)
2
```

index metodу tipki list sınıfında olduğu gibi elemanı bulamazsa exception oluşturmaktadır. Örneğin:

```
>>> a.index(100)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    a.index(100)
ValueError: tuple.index(x): x not in tuple
```

count metodу da listelerde olduğu gibi bir elemanın demet içerisinde yinelenme sayısına geri dönmektedir. Örneğin:

```
>>> a = (1, 2, 3, 1, 2, 1, 5, 7, 1)
>>> a.count(1)
4
```

Python'da bir fonksiyon birden fazla değere geri döndürüyorsa onun geri dönüş değerinin bir demet yapılması sık karşılaşılan bir durumdur.

Python'da tipki listelerde olduğu iki demet de '+' operatörü ile toplanabilirler. Bu durumda iki demetin elemanlarından oluşan yeni bir demet nesnesi yaratılır. Örneğin:

```
>>> a = (1, 'Ali', 2)
>>> b = (3, 'Veli', 4)
>>> c = a + b
>>> c
(1, 'Ali', 2, 3, 'Veli', 4)
```

Anımsanacağı gibi a ve b birer liste olmak üzere:

```
a = a + b
```

İşlemi ile

```
a += b
```

İşlemi birbirinden farklı anımlara geliyordu. İlkinde yeni bir liste nesnesi yaratılırken ikincisinde mevcut nesnenin sonuna ekleme yapılmıyordu. Oysa demetler değiştirilemez türler olduğu için yukarıdaki iki işlemin demet karşılıkları tamamen aynıdır. Örneğin:

```
>>> a = (1, 'Ali', 2)
>>> b = (3, 'Veli', 4)
>>> id(a)
1532942104896
>>> a = a + b
>>> id(a)
1532941395616
>>> a = (1, 'Ali', 2)
>>> id(a)
1532942104896
>>> a += b
>>> id(a)
1532941395616
```

Demetler değiştirilebilir nesneler olmadığı için demetlerin kopyalanmasının da bir anlamı yoktur. Bir demeti dilimleme yöntemiyle kopyalamaya çalışalım:

```
>>> t = (1, 'Ali', 2)
>>> k = t[:]
>>> t
(1, 'Ali', 2)
>>> k
(1, 'Ali', 2)
```

İki demet bu biçimde kopyalandığında Python yorumlayıcısı optimizasyon amaçlı yeni bir nesne oluşturmayabilir. Çünkü demetler değiştirilemez olduğu için böyle bir kopyalama sonucunda yeni bir demet nesnesinin yaratılmasının bir faydası yoktur. İşte kodda işlevsel bir değişiklik gözlenemedikten sonra derleyiciler ve yorumlayıcılar kodun daha hızlı çalışması ya da daha az yer kaplaması için optimizasyon yapabilirler. Zaten listelerle demetlerin en önemli farklılıklarından biri demetlerin bu biçimde bazı optimizasyonlara izin vermesidir. Örneğin:

```
>>> t = (1, 'Ali', 2)
```

```
>>> k = t[:]
>>> id(t)
4489966360
>>> id(k)
4489966360
```

Demetlerde * operatörü listelerde olduğu gibi yineleme amacıyla kullanılabilmektedir. Örneğin:

```
>>> t = 1, 2, 3
>>> t * 2
(1, 2, 3, 1, 2, 3)
```

Örneğin:

```
>>> (10, 20) * 2
(10, 20, 10, 20)
```

Örneğin:

```
>>> 2 * (10, 20)
(10, 20, 10, 20)
```

Burada parantezlerinin bulundurulması gereğine dikkat ediniz. Bu parantezler olmasayı ifade başka bir anlama gelirdi:

```
>>> 2 * 10, 20
(20, 20)
```

Tıpkı dizilerde olduğu gibi demetlerde de in operatörü belli bir elemanın demetin içerisinde olup olmadığını belirlemek için kullanılabilmektedir. Örneğin:

```
>>> t = (10, 'Ali', 12.4)
>>> 10 in t
True
>>> 'Ali' in t
True
>>> 20 in t
False
>>> 20 not in t
True
```

Demetlerle Listeler Arasındaki Farklılıklar

Demetlerin veri yapısı olarak temsil edilmesi listelerden çok daha yalın ve etkin biçimde gerçekleştirilebilmektedir. Yani demetlerle işlemler listelerle işlemlere göre hem daha hızlı hem de daha az yer kaplayan biçimde gerçekleştirilmeye eğilimindedir. Dolayısıyla eğer programcı veri yapısı üzerinde bir ekleme çıkartma yapmayacaksça listeler yerine demetleri tercih etmelidir. Tabii bazen programcı veri yapısına eleman eklemek ya da ondan eleman silmek isteyebilir. Bu durumda listelerin kullanılması zorunludur. Çünkü demetler değiştirilebilir türler değildir. Demetlerin bir avantajı da eğer bir demetin bütün elemanları hash'lenebilir (hashable) ise o demetin de hash'lenebilir (hashable) olmasıdır. Listeler hiçbir durumda hash'lenebilir değildir.

range Sınıfı

range built-in bir belli bir aralıkta tamsayı değerleri elde etmek için kullanılan built-in dolaşılabilir (iterable) bir sınıfıdır. Range nesnesi range sınıfının tür fonksiyonu olan range fonksiyonuyla elde edilir. range fonksiyonu bir, iki ya da üç argüman girilerek çağrılabilmektedir. Bu üç parametreye sırasıyla start, stop ve step parametreleri denilmektedir:

```
range(start, stop, step)
```

Fonksiyonun verdiği range nesnesi dolaşıldığında start değeri dahil olan, stop değeri dahil olmayan step artırımlarıyla int türden değerler elde edilmektedir.

Eğer fonksiyon tek argümanla çağrılırsa bu argüman stop değeri anlamına gelir. Bu durumda start = 0 ve step = 1 alınmaktadır. Örneğin:

```
>>> r = range(10)
>>> a = list(r)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Eğer fonksiyon iki argümanla çağrılırsa bu durumda ilk argüman start, ikinci argüman stop değeri olur. Bu durumda step = 1 kabul edilir. Örneğin:

```
>>> r = range(10, 20)
>>> a = list(r)
>>> a
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Nihayet fonksiyon üç argümanla da çağrılabılır. Bu durumda ilk argüman start, ikinci argüman stop ve üçüncü argüman da step anlamına gelir. Örneğin:

```
>>> r = range(10, 20, 2)
>>> t = tuple(r)
>>> t
(10, 12, 14, 16, 18)
```

range nesnesinde start değerinin dahil, stop değerinin dahil olmadığına dikkat ediniz. range fonksiyonunun start, stop ve step değerleri int türden olmak zorundadır. Yani örneğin biz range nesni ile noktalı artırımlı değerler elde edemeyiz:

```
>>> r = range(0, 10, 0.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

range fonksiyonunu biz dolaşılabilir nesnelerin gerektiği her yerde kullanabiliriz. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:4] = range(10)
>>> a
[1, 2, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5]
```

Burada a listenin 2'inci ve 3'üncü indeksli elemanları silinmiş yerine 0'den 10 kadar (10 dahil değil) değerler insert edilmiştir. Örneğin:

```
>>> a = [10, 20, 30]
>>> a.extend(range(40, 110, 10))
>>> a
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Burada listenin sonuna range nesnesinin dolaşılmaması ile ilde edilen değerler eklenmiştir. range fonksiyonu eğer argümanlar yanlış girilirse boş bir dolaşılabilir nesne verir. Örneğin:

```
>>> a = list(range(10, 10))
>>> a
[]
>>> a = list(range(20, 10, 2))
>>> a
```

[]

Step değeri 0 olamaz. Eğer step değeri 0 olarak girilirse exception oluşur:

```
>>> a = list(range(20, 10, 0))
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    a = list(range(20, 10, 0))
ValueError: range() arg 3 must not be zero
```

range fonksiyonuyla biz büyükten küçüğe doğru değerler de elde edebiliriz. Ancak bunun için step değerinin negatif girilmesi gereklidir. Örneğin:

```
>>> a = list(range(10, 0, -1))
>>> a
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

range fonksiyonunda noktalı artırımların mümkün olmaması pek çok kişi tarafından yadırılganmaktadır. Noktalı artırımların istenmemesinin temel nedeni noktalı sayılarındaki yuvarlama hatalarının kararsız sayıda eleman oluşturma potansiyelidir.

range sınıfı da tipki list ve tuple sınıflarında olduğu gibi bazı işlevlere izin vermektedir. Örneğin range nesnesi üzerinde dilimleme yapılmaktadır. Dilimleme işleminden yeni bir range nesnesi elde edilir. Örneğin:

```
>>> r = range(10, 20)
>>> k = r[2:6]
>>> a = list(k)
>>> a
[12, 13, 14, 15]
```

range nesnesindeki elemanların sayısı len fonksiyonu ile elde edilebilir. Örneğin:

```
>>> a = range(10)
>>> len(a)
10
```

range sınıfının index metodu yine arama yapmak için, count metodu ise belli bir elemanın sayısını bulmak için kullanılmaktadır. Örneğin:

```
>>> r = range(10, 20)
>>> i = r.index(15)
>>> i
5
>>> c = r.count(15)
>>> c
1
```

Veri Yapılarının Açılması (Unpacking)

Dolaşılabilir bir nesnenin elemanları açım (unpacking) denilen bir işlemle pratik bir biçimde değişkenlere atanabilmektedir. Açım işleminin genel biçimleri şöyledir:

- 1) [x, y, z, ...] = <dolaşılabilir nesne>
- 2) (x, y, z, ...) = <dolaşılabilir nesne>
- 3) x, y, z, ... = <dolaşılabilir nesne>

Eskiden Python'da yalnızca demetler açılabiliyordu. Sonra tüm dolaşılabilir nesnelerin açılması mümkün hale getirildi. Bu genel biçimlerde '=' operatörünün solundaki değişken listesi köşeli parantezler içerisinde, normal parantezler içerisinde ve parantezsiz biçimde bulunabilmektedir. Bunların arasında semantik bir farklılık yoktur.

Açım sırasında '=' operatörünün sağındaki dolaşılabilir nesne dolaşılarak Her dolaşımından elde edilen değer solundaki değişkene sırasıyla atanır. Örneğin:

```
>>> x = [1, 2, 3]
>>> [a, b, c] = x
>>> a
1
>>> b
2
>>> c
3
```

Biz burada listenin ilk elemanını a değişkenine, ikinci elemanını b değişkenine ve üçüncü elemanını da c değişkenine atamış olduk. Aynı işlemi normal parantezleri kullanarak da şöyle de yapabilirdik:

```
>>> x = [1, 2, 3]
>>> (a, b, c) = x
>>> a
1
>>> b
2
>>> c
3
```

Açım sırasında soldaki parantezler hiç kullanılmayabilir. Örneğin:

```
>>> x = [1, 2, 3]
>>> a, b, c = x
>>> a
1
>>> b
2
>>> c
3
```

Köşeli parantezlerle açım ile normal parantezlerle açım ve parantezsiz açım arasında semantik hiçbir farklılık yoktur.

Açım işleminde sağdaki dolaşılabilir nesnenin eleman sayısının soldaki değişken listesindeki değişken sayısı ile aynı olması gereklidir. Örneğin:

```
>>> x = [1, 2, 3]
>>> a, b = x
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    [a, b] = x
ValueError: too many values to unpack (expected 2)
>>> a, b, c, d = x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 4, got 3)
```

Dolaşılabilir nesnedeki elemanlar listelerden ve demetlerden de oluşabilir. Örneğin:

```
>>> a, b, c = [1, [2, 3, 4], 5]
>>> a
1
```

```
>>> b  
[2, 3, 4]  
>>> c  
5
```

İç listeler de özyinelemeli olarak açım işlemine sokulabilmektedir. Örneğin:

```
>>> a, [b, c, d], e = [1, [2, 3, 4], 5]  
>>> a  
1  
>>> b  
2  
>>> c  
3  
>>> d  
4  
>>> e  
5
```

İçteki parantezler için yine köşeli parantezler ya da normal parantezler kullanılabilir:

```
>>> a, (b, c, d), e = [1, [2, 3, 4], 5]  
>>> a  
1  
>>> b  
2  
>>> c  
3  
>>> d  
4  
>>> e  
5
```

Bir listeyi dilimlediğimizde yeni bir liste elde ettiğimize göre dilimlenmiş bir listeyi de açabiliriz. Örneğin:

```
>>> x = [1, 2, 3, 4, 5]  
>>> a, b = x[2:4]  
>>> a  
3  
>>> b  
4
```

Açım işleminde atama işleminin solunda herhangi bir dolaşılabilir nesne olabileceği dikkat ediniz. Örneğin:

```
>>> a, b, c = 'ali'  
>>> a  
'a'  
>>> b  
'l'  
>>> c  
'i'
```

Örneğin:

```
>>> a, b, c = reversed('ali')  
>>> a  
'i'  
>>> b  
'l'  
>>> c
```

'a'

Örneğin:

```
>>> a, b, c, d, e = range(10, 15)
>>> a
10
>>> b
11
>>> c
12
>>> d
13
>>> e
14
```

Python'da iki değişkeni birbiriyle yer değiştirmek açım işlemiyle kolay bir biçimde yapılabilmektedir. Örneğin:

```
>>> a = 10; b = 20
>>> b, a = a, b
>>> a
20
>>> b
10
```

Kümeler (Sets)

Kümeler matematiğin en temel konularından biridir. Farklı elemanlardan oluşan toplulukları belirtir. Python'da da kümeler matematik tanımındaki gibi "farklı elemanlardan oluşan" değerlei temsil etmektedir. Bir küme kümeye parantezleri ile aşağıdaki gibi oluşturulur:

```
{<[eleman listesi]>}
```

Küme elemanları ',' atomu ile birbirinden ayrılmaktadır. Örneğin:

```
>>> s = {1, 2, 3, 4, 5}
>>> s
{1, 2, 3, 4, 5}
>>> type(s)
<class 'set'>
```

Python'da kümeler set isimli sınıfı temsil edilmiştir.

Bir küme oluştururken aynı elemandan birden fazla kez yazmak bir hata oluşturmaz. Fazla elemanların yalnızca bir tanesi alınarak kümeye yerleştirilmektedir. Örneğin:

```
>>> s = {1, 2, 1, 1, 1, 3, 4, 3, 5, 5, 5, 4}
>>> s
{1, 2, 3, 4, 5}
```

Tabii küme elemanları da farklı türlerden olabilir. Örneğin:

```
>>> s = {1, 'Ali', (3, 4), 12.3}
>>> s
{(3, 4), 1, 'Ali', 12.3}
```

Ancak her türden nesne küme elemanı yapılamaz. Bir nesnenin küme elemanı olabilmesi için o nesnenin hash'lenebilir (hashable) olması gerekmektedir. Örneğin listeler ya da kümeler hash'lenebilir değildir. Bu nedenle bir kümeyi bir liste olamaz:

```
>>> s = {1, 'Ali', [2, 3], 12.3}
Traceback (most recent call last):
  File "<pyshell#106>", line 1, in <module>
    s = {1, 'Ali', [2, 3], 12.3}
TypeError: unhashable type: 'list'
```

Aynı nedenden dolayı bir kümenin elemanı bir kümede olamaz:

```
>>> s = {1, 2, {3, 4, 5}, 6}
Traceback (most recent call last):
  File "<pyshell#124>", line 1, in <module>
    s = {1, 2, {3, 4, 5}, 6}
TypeError: unhashable type: 'set'
```

Fakat int, float, double, str ve demetler gibi değiştirilemez türler küme elemanı olarak kullanılabilirler. Tabii bir demetin küme elemanı olabilmesi için onun her elemanının da hash'lenebilir olması gerekmektedir. Örneğin:

```
>>> s = {1, 'Ali', ([2, 3], 4), 12.3}
Traceback (most recent call last):
  File "<pyshell#107>", line 1, in <module>
    s = {1, 'Ali', ([2, 3], 4), 12.3}
TypeError: unhashable type: 'list'
```

Kümeler indekslenebilir bir dizilim (sequence) belirtmezler. Bu nedenle bir kümenin elemanlarının bizim onu oluşturduğumuz sırada gösterilmesi ya da bize geri verilmesi de garanti değildir. Ayrıca küme elemanlarına [...] operatörü ile bir indeks belirterek de erişemeyiz. Yani bir kümenin tipki matematikte olduğu gibi n'inci elemanı diye bir kavramı yoktur. Başka bir deyişle küme elemanlarında öncelik sonralık ilişkisi yoktur. Tabii bu nedenlerden dolayı kümeler dilimlenemezler de.

Kümelerde '*' operatörü kullanılamamaktadır. Çünkü zaten küme elemanları birbirinden farklı olacağı için yineleme işleminin bir anlamı yoktur. Benzer biçimde iki küme '+' operatörüyle de toplanamamaktadır. (Ancak ileride görüleceği gibi iki küme üzerinde kesişim, birleşim gibi işlemler yapılmaktadır.)

Python'da boş bir küme {} biçiminde yaratılamamaktadır. Çünkü {} işlemi Python'da boş bir sözlük (dictionary) oluşturmaktadır. Boş kümeler set fonksiyonun argümansız çağrımasıyla oluşturulabilmektedir. Örneğin:

```
>>> s = set()
>>> s
set()
>>> type(s)
<class 'set'>
```

Yine set sınıfının tür fonksiyonu olan set fonksiyonuyla biz bir küme nesnesi oluşturabiliriz. Diğer veri yapılarında olduğu gibi set fonksiyonu da argüman olarak dolaşılabilir bir nesne alır, o nesneyi dolaşarak küme elemanlarını oluşturur. Örneğin:

```
>>> a = [1, 2, 2, 3, 1, 4]
>>> s = set(a)
>>> s
{1, 2, 3, 4}
```

Örneğin:

```
>>> s = set('ankara')
>>> s
{'a', 'n', 'r', 'k'}
```

Built-in len fonksiyonu kümelere de uygulanabilmektedir. Bu durumda kümenin eleman sayısı elde edilmektedir. Örneğin:

```
>>> s = {1, 2, 'Ali', (3, 4)}  
>>> len(s)  
4
```

Kümelerde de in operatörü liste ve demetlere olduğu gibi "içinde var mı" testini yapmaktadır. Örneğin:

```
>>> s = {1, 'Ali', 2, 'Veli'}  
>>> 'Ali' in s  
True  
>>> 1 in s  
True  
>>> 10 in s  
False  
>>> 10 not in s  
True
```

Kümeler veri yapısı olarak in ve not in işlemlerinin daha hızlı yapılabilmesine olanak sağlamaktadır. Yani genel olarak bir elemanın kümenin içinde olup olmadığı kontrolü bir liste ya da demete göre daha hızlı yapılmaktadır. Tabii az sayıda elemandan oluşan listeler ve demetlerde de bu işlemler oldukça hızlı yapılabilmektedir.

Bir kümenin başka bir kümenin altkümesi olup olmadığı issubset metoduyla ya da \leq operatörü ile belirlenebilmektedir. Örneğin:

```
>>> a = {1, 2, 'Ali', 3}  
>>> b = {1, 'Ali'}  
>>> b.issubset(a)  
True  
>>> b <= a  
True
```

issubset metodunun parametresi herhangi bir dolaşılabilir nesne olabilir. Oysa \leq operatörünün operandları bu bağlamda set türünden (ya da frozenset türünden) olmak zorundadır.

Bir kümenin başka bir kümeyi kapsayıp kapsamadığı da issuperset metoduyla ya da \geq operatörüyle belirlenebilir. Örneğin:

```
>>> a = {1, 2, 'Ali', 3}  
>>> b = {1, 'Ali'}  
>>> a >= b  
True  
>>> a.issuperset(b)  
True  
>>>
```

Benzer biçimde issuperset metodunun parametresi herhangi bir dolaşılabilir nesne olabilir. Oysa \geq operatörünün operandları bu bağlamda set türünden (ya da frozenset türünden) olmak zorundadır.

Bir kümenin başka bir kümenin öz altkümesi (proper subset) olup olmadığı da $<$ operatörüyle sorgulanabilir. Bu işlem ayrıca bir metotla yapılamamaktadır. Örneğin:

```
>>> a = {1, 2, 3, 4, 5}  
>>> a < a  
False  
>>> a <= a  
True
```

Benzer biçimde bir kümenin diğer bir kümeyi öz kapsayıp kapsamadığı da `>` operatörü ile belirlenebilmektedir. Bu işlem de ayrıca bir metotla yapılamamaktadır. Örneğin:

```
>>> a = {1, 2, 3, 4, 5}
>>> a > a
False
>>> a >= a
True
```

set sınıfının `isdisjoint` isimli metodunu iki kümenin ayrık olup olmadığını belirlemek için kullanılmaktadır. Örneğin:

```
>>> s = {1, 2, 3}
>>> k = {'ali', 'veli', 'selami'}
>>> s.isdisjoint(k)
True
```

Metodun parametresi herhangi bir dolaşılabilir nesne olabilir.

İki kümenin birleşimi `union` metoduya ya da `|` operatörü ile elde edilir. `union` metoduna biz dolaşılabilir (`iterable`) bir nesneyi argüman olarak verebiliriz. Ancak `|` operatörünün operandlarına biz dolaşılabilir herhangi bir nesne veremeyiz. Bu operatörün operanları set (ya da `frozenset`) türünden olmak zorundadır. Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> b = {3, 'Velı', 2, 1}
>>> c = a.union(b)
>>> c
{1, 2, 'Ali', 3, 'Velı'}
>>> c = a | b
>>> c
{1, 2, 'Ali', 3, 'Velı'}
```

Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> b = [1, 'Velı', 2]
>>> c = a.union(b)
>>> c
{1, 2, 'Ali', 'Velı'}
>>> c = a | b
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    c = a | b
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

Tabii biz birden fazla kümenin de birleşimini benzer biçimde elde edebilirdik:

```
>>> a = {'ali', 1, 'velı'}
>>> b = {10, 20, 'ali'}
>>> c = {'sacit', 'velı', 100}
>>> d = a.union(b).union(c)
>>> d
{1, 10, 20, 100, 'velı', 'ali', 'sacit'}
```

Aynı işlem `|` operatörüyle de yapılabilirdi:

```
>>> a = {'ali', 1, 'velı'}
>>> b = {10, 20, 'ali'}
>>> c = {'sacit', 'velı', 100}
>>> d = a | b | c
>>> d
```

```
{1, 10, 20, 100, 'veli', 'ali', 'sacit'}
```

İki kümenin kesişimini elde etmek için intersection metodu ya da & operatörü kullanılmaktadır. Yine intersection metodu argüman olarak herhangi dolaşılabilir bir nesneyi alırken & operatörü set (ya da frozenset)nesneleriyle kullanılabilmektedir. Örneğin:

```
>>> a = {1, 2, 3, 'Ali'}
>>> b = {1, 10, 'Ali'}
>>> c = a.intersection(b)
>>> c
{1, 'Ali'}
>>> c = a & b
>>> c
{1, 'Ali'}
```

Örneğin:

```
>>> set('ankara') & set('kastamonu')
{'k', 'n', 'a'}
```

İki kümenin tamamen aynı elemanlardan oluşup oluşmadığı == ya da != opeatörüyle test edilebilir. Örneğin:

```
>>> s = {3, 4}
>>> k = {4, 3}
>>> id(s)
2856213233224
>>> id(k)
2856213230984
>>> s == k
True
```

Örneğin aşağıdaki program anagram testi yapmakta kullanılabilir:

```
s = input('Birinci yazıyı giriniz:')
k = input('İkinci yazıyı giriniz:')

print(set(s) == set(k))
```

İki küme farkı için difference metodu ya da - operatörü kullanılabilmektedir. Yine difference metoduna argüman olarak herhangi bir dolaşılabilir nesne girilebilmektedir. Ancak - operatörünün operand'ları set (ya da frozenset) türünden olmak zorundadır. Örneğin:

```
>>> a = {1, 'Ali', 2, 'Veli'}
>>> b = {1, 'Süleyman', 2}
>>> c = a.difference(b)
>>> c
{'Veli', 'Ali'}
>>> c = a - b
>>> c
{'Veli', 'Ali'}
```

İki kümenin ortak olmayan elemanları da (buna exor işlemi de denilmektedir) symmetric_difference ya da ^ operatörü ile elde edilmektedir. Yine symmetric_difference metoduna argüman olarak herhangi bir dolaşılabilir nesne girilebilmektedir. Ancak ^ operatörünün operandı set (ya da frozen_set) türünden olmak zorundadır. Örneğin:

```
>>> a = {1, 'Ali', 2, 'Veli'}
>>> b = {1, 'Süleyman', 2}
>>> c = a ^ b
>>> c
{'Süleyman', 'Ali', 'Veli'}
```

set sınıfı değiştirilebilir (mutable) bir sınıf olduğu için yaratılmış bir set nesnesinin elemanları da daha sonra değiştirilebilir. Örneğin biz bir kümeye add metoduyla yeni elemanlar ekleyebiliriz:

```
>>> a = {1, 'Ali', 2}
>>> a.add(100)
>>> a
{'Ali', 1, 2, 100}
>>> a.add((10, 20, 30))
>>> a
1, 2, 100, (10, 20, 30), 'Ali'}
```

Tabii biz add metoduyla yalnız tek bir eleman ekleyebiliriz. Birden fazla elemanın eklenmesi için update metodu ya da |= operatörü kullanılır. update metodu herhangi bir dolaşılabilir nesneyi argüman olarak kabul etmektedir. Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> a.add(100)
>>> a
{100, 1, 2, 'Ali'}
```

Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> a |= {1, 2, 10, 'Ali', 20}
>>> a
{1, 2, 10, 'Ali', 20}
```

Yine |= operatörü yalnızca set ya da frozenset operandı almaktadır.

Benzer biçimde intersection_update metodu ile &= operatörü, difference_update metodu ile -= operatörü, symmetric_difference_update metodu ile de ^= operatörü aynı amaçlarla kullanılabilir. intersection_update, difference_update ve symmetric_difference_update metodlarının parametreleri herhangi bir dolaşılabilir nesne olabilir. Ancak &=, -= ve ^= operatörlerinin sağ tarafındaki operandları yalnızca set ya da frozen set türünden olabilmektedir. Örneğin:

```
>>> s = {1, 2, 3, 4, 5}
>>> s.difference_update([2, 3, 4])
>>> s
{1, 5}
```

Burada da yine a &= b ile a = a & b aynı etkiye sahip olduğu halde aynı semantik sahip değildir. a = a & b işleminde a & b sonucunda yeni bir set nesnesi yaratılmaktadır. Halbuki a &= b işleminde yeni bir set nesnesi yaratılmaz. Mevcut a nesnesinin içi değiştirilir. Aynı durum a |= b, a -= b, a ^= b durumları için de geçerlidir. Ancak sınıfın union_update isimli bir metodu yoktur. Çünkü işlem zaten union_update gibi bir işlem update işlemiyle aynı anlama gelmektedir. update işleminin aynı zamanda |= operatörü ile de yapılabildiğini anımayınız.

remove metodu belli bir elemanı kümeden çıkartmak için kullanılır. Eğer o eleman yoksa exception oluşur. Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> a.remove(2)
>>> a
{1, 'Ali'}
>>> a.remove('Veli')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    a.remove('Veli')
KeyError: 'Veli'
```

Tabii remove metodu da bir aramaya yol açmaktadır ve arama işlemi de kümelerde listelere göre çok daha hızlı yapılmaktadır.

discard metodu da remove gibi elemanı kümeden çıkartmak için kullanılmaktadır. Ancak eğer eleman kümede yoksa exception oluşmaz. Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> a.discard(2)
>>> a
{'Ali', 1}
>>> a.discard('Veli')
>>> a
{'Ali', 1}
```

Kümelerde de pop metodu parametresiz olarak kullanılabilir. Bu durumda pop metodu kğmeden rastgele bir elemanı siler ve sildiği elemanı verir. Örneğin:

```
>>> s = {'ali', 1, 'veli', 2}
>>> s.pop()
1
```

clear metodu kümedeki tüm elemanları siler. Örneğin:

```
>>> a = {1, 'Ali', 2}
>>> a.clear()
>>> a
>>> set()
```

Bir kümenin özdeş yeni bir kopyasından oluşturmak için set sınıfının copy metodu kullanılır. Örneğin:

```
>>> a = {1, 2, 3}
>>> b = a.copy()
>>> b
{1, 2, 3}
```

Tabii yine bu işlem sığ kopyalama (shallow copy) biçiminde gerçekleştirilmektedir. Yani kopyalama sırasında yalnızca ana nesnenin kopyası çıkartılır.

```
>>> a = {1, 2, 3}
>>> b = a.copy()
>>> id(a)
1962657789096
>>> id(b)
1962657789992
>>> a.add(100)
>>> a
{1, 2, 3, 100}
>>> b
{1, 2, 3}
```

Değiştirilemez Kümeler

Normal kümeler set sınıfıyla temsil edilmektedir ve değiştirilebilir (mutable) nesnelerdir. Ancak bir de değiştirilemez (immutable) kümeler vardır. Bunlar frozenset sınıfıyla temsil edilmektedir. Bir değiştirilemez küme küme parantezleriyle değil doğrudan sınıfın tür fonksiyonu olan frozenset fonksiyonuyla oluşturulmaktadır. Örneğin:

```
>>> fs = frozenset([1, 'Ali', 2, 'Veli'])
>>> fs
frozenset({'Veli', 1, 2, 'Ali'})
```

frozenset fonksiyonu yine dolaşılabilir herhangi bir nesneyi parametre olarak alabilmektedir. frozenset sınıfı da set sınıfı gibi temel kümeye işlemlerine sokulabilmektedir. Fakat frozenset sınıfında elemanları değiştiren add gibi update gibi, clear gibi metodlar yoktur.

Bir frozenset nesnesi başka bir frozenset nesnesiyle ya da set nesnesiyle yukarıda ele aldığımız kesişim, bileşim gibi operatör işlemlerine sokulabilmektedir. Bu işlemlerden elde edilen ürün sol taraftaki operand hangi türdense o türden olmaktadır. Örneğin:

```
>>> fs = frozenset(['ali', 'veli', 'selami'])
>>> s = {'ayşe', 'ali', 'selami'}
>>> fs & s
frozenset({'ali', 'selami'})
>>> s & fs
{'ali', 'selami'}
>>> fs = frozenset(['ali', 'veli', 'selami'])
>>> fs | {'selami', 'ayşe'}
frozenset({'selami', 'veli', 'ayşe', 'ali'})
>>> fs - s
frozenset({'veli'})
>>> s ^ fs
{'ayşe', 'veli'}
```

Tabii bu operatörlere karşı gelen metodlar da yine frozenset sınıfında bulunmaktadır. Yani & işlemi yine intersection metodu ile, | işlemi union metodu ile, - işlemi difference metodu ile ^ işlemi symmetric_difference metodu ile yapılabilmektedir. Örneğin:

```
>>> fs.intersection(['selami', 'ayşe'])
frozenset({'selami'})
>>> fs = frozenset(['ali', 'veli', 'selami'])
>>> fs.union({'selami', 'ayşe'})
frozenset({'selami', 'veli', 'ayşe', 'ali'})
>>> fs.difference(['selami', 'ayşe'])
frozenset({'veli', 'ali'})
>>> fs.symmetric_difference(['selami', 'ayşe'])
frozenset({'ayşe', 'ali', 'veli'})
```

frozenset sınıfının yine set sınıfında olan nesne üzerinde değişiklik yapmayan metodları vardır. Örneğin isdisjoint metodu kümelerin ayrık olup olmadığını anlamakta kullanılabilir, copy metodu nesneyi kopyalamakta kullanılabilir.

frozenset sınıfının küme üzerinde değişiklik yapan difference_update gibi, intersection_update gibi metodlarının olmadığını belirtmiştik. frozenset nesneleri ile &=, |= gibi işlemler set üzerindeki işlemlerden farklıdır. frozenset üzerinde &=, |= gibi işlemlerde yeni nesne yaratılmaktadır. Yani örneğin:

```
a = frozenset([3, 4, 5, 6])
print(id(a))
a |= {1, 2, 3}
print(id(a))
```

Burada a nesnesinin adresi değişecektir. Halbuki aynı örneği set üzerinde yapsaydık a nesnesinin adresi değişmezdi.

frozenset sınıfı dolaşılabilir bir sınıfıdır. Sınıf değiştirilemez olduğu için aynı zamanda -eğer tüm elemanları hash'lenebilir ise- hash'lenebilirdir. Dolayısıyla örneğin biz bir frozenset nesnesini bir set nesnesinin içerisine yerleştirebiliriz:

```
Type "help", "copyright", "credits" or "license" for more information.
>>> s = {10, 20, {30, 40}, 50}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> s = {10, 20, frozenset([30, 40]), 50}
```

```
>>> s
{frozense({40, 30}), 10, 20, 50}
```

Sözlükler (Dictionaries)

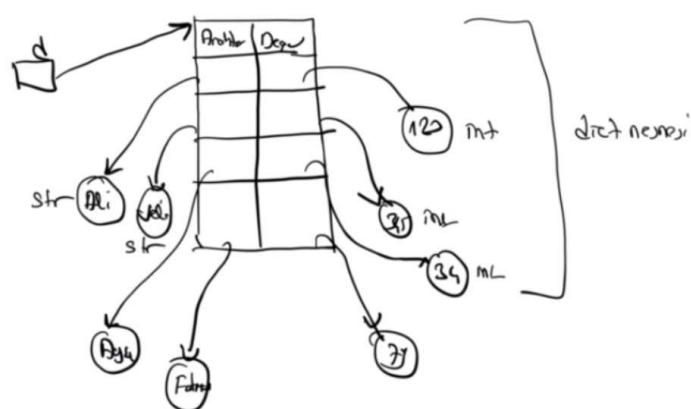
Sözlükler anahtar-değer çiftlerinden oluşan, arama amaçlı kullanılan veri yapılarıdır. Başka dillerde bunlara "look up table", "hash table" ya da "map" denilebilmektedir. Sözlük daha genel bir isim olarak Python'da tercih edilmiştir. Sözlükler anahtardan değeri elde etmek için kullanılan built-in veri yapılarıdır. dict isimli sınıfı temsil edilmişlerdir. Bir sözlük yaratmanın çeşitli yolları varsa da en çok kullanılan yöntem yine kümelerde olduğu gibi küme parantezi sentaksıdır. Bu sentaksın genel biçimini şöyledir:

```
{ <anahtar>: <değer>, <anahtar>: <değer>, <anahtar>: <değer>, ... }
```

Bu sentaksta anahardan sonra ':' atomunun getirildiğine dikkat ediniz. Örneğin:

```
>>> d = {'Ali': 123, 'Veli': 345, 'Selami': 34, 'Ayşe': 65, 'Fatma': 79}
>>> d
{'Ali': 123, 'Veli': 345, 'Selami': 34, 'Ayşe': 65, 'Fatma': 79}
>>> type(d)
<class 'dict'>
```

Burada isimler anahtar onlara karşı gelen numaralar da değer olarak kullanılmıştır. Sözlük nesneleri de aslında anahtar ve değerlerin kendilerini değil onların adreslerini tutmaktadır. Yukarıdaki sözlük nesnesini sembolik biçimde aşağıdaki gibi betimleyebiliriz:



Buradaki şekil yalnızca bir fikir oluştursun diye verilmiştir. Yoksa sözlük içerisindeki anahtar-değer çiftleri böyle ardışılık biçimde tutulmamaktadır. Sözlük anahtar-değer çiftlerini hızla arama yapmak için daha karmaşık bir veri yapısı biçiminde tutar. (Örneğin sözlükler bu tür dillerde ve kütüphanelerde tipik olarak "hash tabloları" ya da "ikili ağaçlar" biçiminde gerçekleştirilmektedir.)

Sözlüklerde anahtar-değer çiftlerindeki anahtarlar hash'lenebilir türlerden olmak zorundadır. int, float, str, bool, complex ve NonType türlerinin hash'lenebilir olduğunu daha önce belirtmiştık. (Yine anımsayacağınız gibi listeler hiçbir biçimde hash'lenebilir değildi. Ancak demetlerin elemanları hash'lenebilir ise demetler hash'lenebilir nesneler olarak kullanılabiliyordu.) Bu nedenle listeler ve kümeler sözlüklerde anahtar olarak kullanılamazlar. Sözlüklerin kendileri de hash'lenebilir değildir. Dolayısıyla sözlükler de sözlüklerde anahtar olarak kullanılamamaktadır. Fakat sözlük elemanlarındaki değerlerin hash'lenebilir olması gerekmektedir. Bu nedenle anahtara karşı gelen değerler herhangi bir türden olabilirler. Örneğin:

```
>>> cities = {'Ankara': ['Çankaya', 'Ulusal', 'Kızılay'], 'Eskişehir': ['Sivrihisar', 'Mihalıççık', 'Alpu', 'Seyitgazi'], 'Konya': ['Karatay', 'Meram', 'Çumra']}
>>> cities
{'Ankara': ['Çankaya', 'Ulusal', 'Kızılay'], 'Eskişehir': ['Sivrihisar', 'Mihalıççık', 'Alpu', 'Seyitgazi'], 'Konya': ['Karatay', 'Meram', 'Çumra']}
```

Burada sözlüğün anahtarları şehir isimlerinden değerleri ise onların ilçelerini belirten listelerden oluşmaktadır.

Sözlükler anahtarları verince değerin hızlı bir biçimde elde edilmesi için kullanılmaktadır. [...] operatörü ile biz anahtarları verirsek bu operatör bize sözlükte o anahtara karşı gelen değeri verir. Eğer anahtar sözlükte yoksa exception oluşacaktır. Örneğin:

```
>>> d = {'Ali': 123, 'Veli': 345, 'Selami': 34, 'Ayşe': 65, 'Fatma': 79}
>>> d['Ayşe']
65
>>> d['Veli']
345
>>> d['Sacit']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Sacit'
```

Sözlüklerde anahtar verildiğinde değerin nasıl elde edildiğini bilmemize gerek yoktur. Fakat bu elde etme işlemi yorumlayıcı tarafından özel algoritmalar kullanılarak çok hızlı bir biçimde gerçekleştirilmektedir.

Anahtarların ve değerlerin sözlük içerisinde aynı türden olması zorunlu değildir. Örneğin:

```
>>> d = {10: 'Ali', 'Veli': 20, 'Selami': 12.4}
>>> d[10]
'Ali'
>>> d['Selami']
12.4
```

Bir anahtara ilişkin değer dict sınıfının get metodunu ile de elde edilebilir. Ancak get metodunu anahtar sözlükte yoksa exception oluşturmaz, ikinci parametresiyle girilmiş olan değeri geri döndürür. Eğer ikinci parametre için değer girilmezse get metodunu anahtarın bulunamaması durumunda None değeri ile geri dönmektedir. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
>>> d.get(3)
'Selami'
>>> d[3]
'Selami'
>>> d.get(100)
>>> d.get(100, 'Not Found')
'Not Found'
```

Sözlüklerde bir anahtar bir değerle eşleştirilmektedir. Bir anahtar verildiğinde birden fazla değer elde edilmek isteniyorsa değerin demet, liste ya da başka bir veri yapısı biçiminde ifade edilmesi gereklidir. Örneğin:

```
>>> d = {'Ali Yılmaz': ('Eskişehir', 1970, 'Odunpazarı'), 'Mehmet Şen': ('İstanbul', 1992, 'Şişli'), 'Ayşe Er': ('Manisa', 1983, 'Beyaz Fil'), 'Sacit Yurt': ('Ankara', 1969, 'Seymenler')}
>>> birth_place, birth_date, region = d['Mehmet Şen']
>>> print(birth_place, birth_date, region)
İstanbul 1992 Şişli
```

Buradaki sözlükte kişinin ismi verildiğinde onun doğum yeri, doğum tarihi ve oturduğu semt elde edilmek istenmiştir. Elde edilen demetin açılarak kullanıldığına dikkat ediniz.

Sözlüklerde her zaman anahtardan hareketle değer bulunur. Değerden hareketle anahtar bulunamaz.

Bir sözlük nesnesi kümeye parantezi sentaksının yanı sıra dict sınıfının tür fonksiyonu olan dict fonksiyonuyla da oluşturulabilir. Bu durumda dict fonksiyonuna argüman olarak dolaşılabilir bir nesne vermek gereklidir. Ancak argüman olarak verilen dolaşılabilir nesnelerin de elemanlarının iki elemanlı dolaşılabilir nesneler olması gereklidir. Örneğin:

```
>>> l = [(1, 'Adana'), (26, 'Eskişehir'), (6, 'Ankara'), (34, 'İstanbul'), (35, 'İzmir')]
>>> d = dict(l)
```

```
>>> d  
{1: 'Adana', 26: 'Eskişehir', 6: 'Ankara', 34: 'İstanbul', 35: 'İzmir'}
```

Aslında burada iki elemanlı demet listesi yerine iki elemanlı liste listesi de kullanılabildi:

```
>>> l = [[1, 'Adana'], [26, 'Eskişehir'], [6, 'Ankara'], [34, 'İstanbul'], [35, 'İzmir']]  
>>> d = dict(l)  
>>> d  
{1: 'Adana', 26: 'Eskişehir', 6: 'Ankara', 34: 'İstanbul', 35: 'İzmir'}
```

Ya da benzer biçimde argüman olarak iki elemanlı demetlerden oluşan demetleri ya da listelerden oluşan demetleri de kullanılabildik. Örneğin:

```
>>> d = dict(['ab', 'cd', 'ef', range(2), (10, 20), [30, 40]])  
>>> d  
{'a': 'b', 'c': 'd', 'e': 'f', 0: 1, 10: 20, 30: 40}
```

dict fonksiyonunda aslında biz anahtarları ve değerleri <argüman ismi>=<değer> biçiminde de oluşturabiliriz. Örneğin:

```
>>> d = dict(alı=100, veli=200, selami=300, ayşe=400, fatma=500)  
>>> d  
{'alı': 100, 'veli': 200, 'selami': 300, 'ayşe': 400, 'fatma': 500}
```

Burada dict fonksiyonunda sanki alı, veli, selami, ayşe, fatma birer parametre ismi gibi kullanılmıştır. Bu biçimdeki kullanımda parametre isimleri her zaman string biçiminde sözlüğe anahtar yapılmaktadır. Tabii buradaki parametre isimleri değişken isimlendirme kurallarına uygun bir biçimde isimlendirilmiş olmalıdır. Bu nedenle aşağıdaki kullanımlar geçersizdir:

```
>>> d = dict(10='alı', 20='veli', 30='selami')  
>>> d = dict(10alı=100, 20veli=200, 30selami=300)
```

Sözlük içerisindeki anahtarların tek (unique) olması gereklidir. Yani sözlükte aynı anahtara sahip birden fazla sözlük elemanı olamaz. Zaten aynı anahtara sahip yeni bir sözlük elemanı eklemeye çalışırsak bu işlem güncelleme anlamına gelmektedir. Örneğin:

```
>>> d = {34: 'Ağrı', 26: 'Eskişehir', 35: 'İzmir', 34: 'İstanbul'}  
>>> d  
{34: 'İstanbul', 26: 'Eskişehir', 35: 'İzmir'}
```

Sözlükler "değiştirilebilir (mutable)" nesnelerdir. Yani biz bir sözlük nesnesi üzerinde değişiklikler yapabiliriz. Örneğin bir sözlükteki anahtara karşı gelen değeri değiştirebiliriz:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}  
>>> d[3]  
'Selami'  
>>> d[3] = 'Sacit'  
>>> d[3]  
'Sacit'
```

Bir sözlükte olmayan bir anahtara [...] operatörü ile değer atamak geçerli bir işlemdir. Bu durumda o eleman sözlüğe eklenir. Tabii olmayan bir anahtarın değerine [...] operatörü ile erişmeye çalışırsak daha önceden de belirttiğimiz gibi exception oluşacaktır. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}  
>>> d[10]  
Traceback (most recent call last):  
  File "<pyshell#79>", line 1, in <module>  
    d[10]  
KeyError: 10
```

```
>>> d[10] = 'Fehmi'  
>>> d  
{1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 10: 'Fehmi'}
```

Yukarıda da belirttiğimiz gibi bir sözlüğe birden fazla aynı anahtar girilmeye çalışılırsa bu durum anahtarın değerinin güncellenmesi anlamına gelmektedir. Bu nedenle sözlük yaratırken küme parantezi içerisinde aynı anahtarın yeniden kullanılması geçerlidir ancak anlamlı değildir. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 3: 'Sacit'}  
>>> d  
{1: 'Ali', 2: 'Veli', 3: 'Sacit', 4: 'Ayşe', 5: 'Fatma'}
```

Bu örnekte 3 anahtara iki kez atama yapılmıştır. Anahtarda son atanın değerin kaldığına dikkat ediniz.

Sözlüklerde de built-in len fonksiyonu sözlükteki anahtar-değer çiftlerinin sayısını vermektedir. Örneğin:

```
>>> a = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}  
>>> len(a)  
5
```

Sözlüklerde in ve not in operatörü "anahtarın sözlükte bulunup bulunmadığını" bakmaktadır. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 99: 'Süleyman'}  
>>> 3 in d  
True  
>>> 'Ali' in d  
False  
>>> 3 not in d  
False  
>>> 100 not in d  
True
```

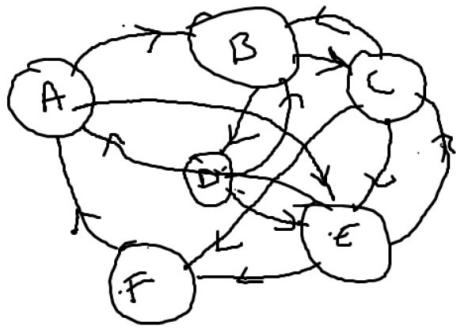
Tabii tıpkı kümelerde olduğu gibi sözlüklerde de in ve not in operatörleri bu işlemi çok hızlı bir biçimde gerçekleştirilmektedir.

Sözlük nesnelerinin kendileri dolaşılabilir nesnelerdir. Biz sözlük nesnesini dolaştığımızda sözlüğün anahtarlarını elde ederiz. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 6: 'Süleyman'}  
>>> list(d)  
[1, 2, 3, 4, 5, 6]  
>>> tuple(d)  
(1, 2, 3, 4, 5, 6)  
>>> set(d)  
{1, 2, 3, 4, 5, 6}
```

Eskiden sözlük nesnesini dolaştığımızda anahtarların hangi sırada elde edileceğinin bir garantisı yoktu. Python 3.7 ile birlikte anahtarların sözlüğe eklenme sırası nasilsa öyle elde edileceği garanti edilmiştir. Yani biz sözlüğü dolaştığımızda anahtarları insert ettiğimiz sırada elde ederiz.

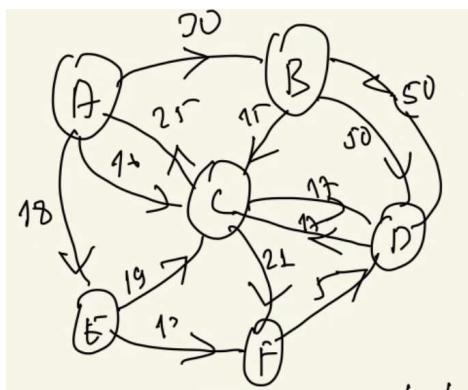
Örneğin bir graf veri yapısı sözlüklerle basit biçimde gerçekleştirilebilir:



```
graph = {'A': {'B', 'E'}, 'B': {'D', 'C'}, 'C': {'E', 'B', 'F'}, 'D': {'E', 'B'}, 'E': {'C', 'A'}, 'F': {'A'}}
```

```
result = 'F' in graph['A']
print(result)
```

Tabii grafin yollarına başka bilgiler de ilişkilendirilebilir. Örneğin graf bir haritadaki yolları temsil ediyorsa bu yolların bir uzunluğu söz konusu olabilir:



```
graph = {'A': {'B': 30, 'C': 16, 'E': 18}, 'B': {'D': 50, 'C': 15}, 'C': {'A': 25, 'D': 17, 'F': 21}, 'D': {'B': 50, 'C': 17}, 'E': {'C': 19, 'F': 13}}
```

```
l = graph['A']['C']
print(l)
```

dict Sınıfının Metotları

list sınıfında olduğu gibi dict sınıfının da clear metodu tüm elemanları silmek için, copy metodu ise elemanları kopyalamak için kullanılmaktadır. Örneğin:

```
d = {1: 'ali', 2: 'veli', 3: 'selami'}
>>> k = d.copy()
>>> k
{1: 'ali', 2: 'veli', 3: 'selami'}
>>> d
{1: 'ali', 2: 'veli', 3: 'selami'}
>>> d[1] = 'hüsün'
>>> d
{1: 'hüsün', 2: 'veli', 3: 'selami'}
>>> k
{1: 'ali', 2: 'veli', 3: 'selami'}
```

Kopyalama yine diğer veri yapılarında olduğu gibi sık kopyalama biçiminde gerçekleştirilmektedir.

dict sınıfının get isimli metodunu yukarıda ele almıştık. Anımsanacağı gibi bu metod tipki [...] operatörü gibi anahtarları verip değeri almak için kullanılmaktadır. Ancak get metodunun anahtar bulunamazsa exception fırlatmadığını anımsayınız.

dict sınıfının keys isimli metodу bize tüm anahtarları, values isimli metodу ise bize tüm değerleri dolaşılabilir bir sınıf nesnesi biçiminde vermektedir.

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
```

```
>>> d.keys()
```

```
dict_keys([1, 2, 3, 4, 5])
```

```
>>> d.values()
```

```
dict_values(['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma'])
```

Tabii biz dolaşılabilir bir nesneden hareketle list gibi, tuple gibi, set gibi nesneleri oluşturabiliriz. Örneğin:

```
>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
```

```
>>> list(d.keys())
```

```
[1, 2, 3, 4, 5]
```

```
>>> list(d.values())
```

```
['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
```

```
>>> tuple(d.keys())
```

```
(1, 2, 3, 4, 5)
```

```
>>> tuple(d.values())
```

```
('Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma')
```

```
>>> set(d.keys())
```

```
{1, 2, 3, 4, 5}
```

```
>>> set(d.values())
```

```
{'Veli', 'Fatma', 'Ali', 'Ayşe', 'Selami'}
```

dict sınıfının keys ve values isimli metodları sözlüğün anahtarlarını ve değerlerini bize bir "view" dolaşılabilir nesnesi biçiminde vermektedir. Burada "view" nesnesi ile anlatılmak istenen asıl sözlükte değişiklik yapıldığında keys ve values metodlarından elde edilen nesnelerin bu değişikliği yansıtılmasına izin vermemektedir. Örneğin:

```
>>> d = {'ali': 10, 'veli': 20, 'selami': 30, 'ayşe': 40, 'fatma': 50}
```

```
>>> k = d.keys()
```

```
>>> v = d.values()
```

```
>>> list(k)
```

```
['ali', 'veli', 'selami', 'ayşe', 'fatma']
```

```
>>> list(v)
```

```
[10, 20, 30, 40, 50]
```

```
>>> d['sacit'] = 60
```

```
>>> list(k)
```

```
['ali', 'veli', 'selami', 'ayşe', 'fatma', 'sacit']
```

```
>>> list(v)
```

```
[10, 20, 30, 40, 50, 60]
```

dict sınıfının items isimli metodу bize dolaşıldıkça anahtar değer çiftlerinden oluşan demetlerin elde edildiği dolaşılabilir bir nesne verir. Yani items metodу bize dolaşılabilir bir nesne vermektedir. Biz bu nesneyi dolaştıkça (anahtar, değer) çiftlerini demet olarak elde ederiz. Örneğin:

```
>>> d = {'ali':10, 'veli': 20, 'selami': 30}
```

```
>>> d.items()
```

```
dict_items([('ali', 10), ('veli', 20), ('selami', 30)])
```

```
>>> list(d.items())
```

```
[('ali', 10), ('veli', 20), ('selami', 30)]
```

```
>>> tuple(d.items())
```

```
((('ali', 10), ('veli', 20), ('selami', 30)))
```

Yine items metodу da dolaşılabilir nesneyi bize bir "view" olarak vermektedir. Yani biz bu nesneyi alıp sakladığımızda sözlükteki değişimler bu nesneye yansıtılmaktadır. Örneğin:

```

>>> d = {'ali': 10, 'veli': 20, 'selami': 30}
>>> i = d.items()
>>> list(i)
[('ali', 10), ('veli', 20), ('selami', 30)]
>>> d['ayşe'] = 40
>>> list(i)
[('ali', 10), ('veli', 20), ('selami', 30), ('ayşe', 40)]

```

dict sınıfının pop isimli metodu belli bir anahtarı arar. Onu bulursa siler. Yani listelerdeki remove metodu gibidir. Örneğin:

```

>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
>>> d.pop(3)
'Selami'
>>> d
{1: 'Ali', 2: 'Veli', 4: 'Ayşe', 5: 'Fatma'}

```

pop metodu bize silinen elemanın değerini geri dönüş değeri olarak vermektedir. Eğer anahtar yoksa exception oluşur. Ancak pop metodu da iki argümanlı biçimde kullanılabilmektedir. Metot iki argümanlı kullanıldığında eğer anahtar bulunamazsa ikinci argümanın değeri ile geri döner. Örneğin:

```

d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
>>> d.pop(100, 'Not found')
'Not found'

```

dict sınıfının update isimli metodu sözlüğe yeni anahtar değer çiftleri eklemekte kullanılır. Bu metot dolaşılabilir bir nesneyi parametre olarak alır. Ancak parametre olarak aldığı bu dolaşılabilir nesnenin de iki elemanlı dolaşılabilir bir nesne olması gereklidir. (Yani sanki dict sınıfının tür fonksiyonu olan dict fonksiyonunda olduğu gibi.) Örneğin:

```

>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
>>> d.update([(6, 'Aydın'), (7, 'Sibel')])
>>> d
{1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 6: 'Aydın', 7: 'Sibel'}

```

Tıpkı dict sınıfının tür fonksiyonunda (yani dict fonksiyonunda) olduğu gibi update metodu da isimli parametrelerle sözlüğe eleman eklemekte kullanılabilmektedir. Yine isimli parametreler <isim>=<değer> sentaksiyla belirtilirler. Buradaki isimler sözlüğe string türünden anahtar biçiminde eklenmektedir. Örneğin:

```

>>> d = {'ali': 10, 'veli': 20, 'selami': 30}
>>> d.update(ayşe=40, fatma=50)
>>> d
{'ali': 10, 'veli': 20, 'selami': 30, 'ayşe': 40, 'fatma': 50}

```

Ayrıca dict sınıfının bir de setdefault isimli metodu vardır. Bu metot bir anahtar verildiğinde tıpkı [...] gibi bize onun değerini verir. Ancak anahtar yoksa yeni bir anahtar yaratıp ona ikinci argümanıyla verilen değeri yerleştirmektedir. ikinci argüman hiç girilmeyebilir. Bu durumda setdefault metodu anahtarın olmaması durumunda None değerini değer olarak yerleştirmektedir. Örneğin:

```

>>> d = {1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma'}
>>> d.setdefault(3)
'Selami'
>>> d.setdefault(100, 'Siracettin')
'Siracettin'
>>> d
{1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 100: 'Siracettin'}
>>> d.setdefault(200)
>>> d
{1: 'Ali', 2: 'Veli', 3: 'Selami', 4: 'Ayşe', 5: 'Fatma', 100: 'Siracettin', 200: None}

```

Bu durumda sözlüğe bir anahtar/değer çifti eklemenin üç yolu olabilir: Köşeli parantez operatörü ile, setdefault metodu ile ya da update metodu ile. Örneğin:

```
d[key] = value  
d.update([(key, value)])  
d.setdefault(key, value)
```

dict sınıfına Python 3.9 ile birlikte '|' operatörü de eklenmiştir. Bu operatör iki sözlük nesnesinden onları birleştirerek tek bir sözlük nesnesi yaratmaktadır. Örneğin:

```
>>> a = {'ali': 10, 'veli': 20, 'selami': 30}  
>>> b = {'ayşe': 40, 'fatma': 50}  
>>> c = a | b  
>>> c  
{'ali': 10, 'veli': 20, 'selami': 30, 'ayşe': 40, 'fatma': 50}
```

|= operatörü tamamen update işlemi ile aynı işlemi yapmaktadır. Başka bir deyişle iki sözlüğü soldaki sözlük üzerinde birleştirmektedir.

Stringler

Sabitler konusunda stringlerin nasıl oluşturulduğunu görmüştük. Şimdi string kullanımı üzerinde duracağız. Daha önceden de belirttiğimiz gibi str sınıfı kategorik olarak "değiştirilemez (immutable)" bir sınıfır. str sınıfı değiştirilemez olduğu için onun metodları da asıl yazında değişiklik yapmamakta değiştirilmiş bir yazıyı bize vermektedir.

Boş bir string nesnesi yine tür fonksiyonu olan str fonksiyonuyla yaratılabilir. Örneğin:

```
>>> s = str()  
>>> len(s)  
0
```

Tabii biz boş string'i daha önce de gördüğümüz gibi boş tırnaklarla da yaratabiliz. Örneğin:

```
>>> s = ''  
>>> len(s)  
0
```

String nesnesinin karakterlerine tıpkı listelerde ve demetlerde olduğu gibi [...] operatörüyle erişilebiliriz. Benzer biçimde string'ler üzerinde tıpkı listelerde ve demetlerde olduğu gibi dilimleme yapabiliriz. Yani string'leri sanki karakterlerden oluşan bir listeler gibi düşünebilirsiniz. Negatif indeksler stirngler'de de list ve demetlerde olduğu gibi kullanılabilir. Örneğin:

```
>>> s = 'ankara'  
>>> s[2]  
'k'  
>>> s[2:4]  
'ka'  
>>> s[2:]  
'kara'  
>>> s[:4]  
'anka'  
>>> s = 'ankara'  
>>> s[-1]  
'a'  
>>> s[-3:]  
'ara'  
>>> s[1:6:2]  
'naa'
```

```
>>> s[::-1]
'arakna'
```

String'ler değiştirilemez (immutable) nesneler olduğu için biz yaratılmış bir string'in hiçbir karakterini daha sonra değiştiremeyiz. Örneğin:

```
>>> s = 'ankara'
>>> s[0] = 'A'
Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    s[0] = 'a'
TypeError: 'str' object does not support item assignment
```

Daha önceden de belirtildiği gibi string'ler dolaşılabilir nesnelerdir. Onlar dolaşıldığında onların tek tek karakterlerini elde ederiz. Bu nedenle biz bir string'i list, tuple ve set fonksiyonlarıyla listelere, demetlere ve kümelere dönüştürebiliriz. Örneğin:

```
>>> s = 'Ankara'
>>> list(s)
['A', 'n', 'k', 'a', 'r', 'a']
>>> tuple(s)
('A', 'n', 'k', 'a', 'r', 'a')
>>> set(s)
{'r', 'A', 'a', 'n', 'k'}
```

Benzer biçimde len fonksiyonu string'in karakter uzunluğunu bize verir. in ve not in operatörleri belli bir karakterin ya da string'in string içerisinde olup olmadığını sorgulamak için kullanılmaktadır. Örneğin:

```
>>> s = 'Ankara'
>>> len(s)
6
>>> 'k' in s
True
>>> 'k' not in s
False

>>> s = 'ankara, izmir, istanbul'
>>> 'ankara' in s
True
```

Python'da aralarında hiçbir operatör bulunmayana yan yana yazılmış string'ler birleştirilmektedir. Örneğin:

```
>>> s = 'ali' 'veli'
>>> s
'alivelei'
```

Tabii yan yana string'lerin tek tırnakla mı yoksa çift tırnakla mı yazıldığıının bir önemi yoktur. Örneğin:

```
>>> s = 'ali' "veli"
>>> s
'alivelei'
```

Tek tırnaklı string'lerin aynı satır üzerine yazılması gerektiğini anımsayınız. Öte yandan üç tırnaklı string'ler farklı satırlara yayılmış olarak bulundurulabilmektedir. Ancak bu durumda aşağı satıra geçmek için kullanılan LF karakteri de string'in bir parçası haline gelir. Tek tırnaklı string'lerin iki ayrı satıra yazılması için C/C++ gibi dillerde bulunan \ uzatması kullanılmaktadır. String'ler söz konusu olsun ya da olmasın Python'da bir satırın sonuna yerleştirilen \ karakteri aşağıdaki satır ile \ karakterinin yerleştirildiği satırın birleştirilmesi anlamına gelmektedir. Örneğin:

```
>>> s = 'bugün hava ' \

```

```
'çok güzel'  
>>> s  
'bugün hava çok güzel'
```

Örneğin:

```
>>> s = 'ankara'  
>>> x = len(\  
    s)  
>>> x  
6
```

str Sınıfının Metotları

Yukarıda da belirtildiği gibi str sınıfı "değiştirilemez (immutable)" bir sınıfıdır. Dolayısıyla bir string nesnesi yaratıldığında artık onun karakterlerini değiştiremeyiz. Bu nedenle str sınıfının metotları asıl yazı üzerinde değişiklik yapamazlar. Bize değiştirilmiş yeni bir yazı verirler.

str sınıfının capitalize isimli metodunu parametre almaz. Bu metod yazının ilk harfinin büyük yapıldığı yeni bir yazıyı bize verir. Örneğin:

```
>>> s = 'bugün hava çok güzel'  
>>> k = s.capitalize()  
>>> k  
'Bugün hava çok güzel'
```

Yazının tüm sözcüklerini büyük harf yapmak için title isimli metod da bulunmaktadır:

```
>>> name = 'ali serçe'  
>>> k = name.title()  
>>> k  
'Ali Serçe'
```

center metodunu yazıyı parametresiyle belirtilen uzunlukta bir alanın ortasına yerleştirmektedir. Yazının iki tarafı boşlukla doldurulur. Örneğin:

```
>>> s = 'ankara'  
>>> k = s.center(20)  
>>> k  
'           ankara           '
```

Buradaki 20 toplam alanı belirtmektedir. center metoduna iki argüman da girebiliriz. Bu durumda ikinci argüman doldurulacak karakteri belirtir. Örneğin:

```
>>> s = 'ankara'  
>>> k = s.center(20, 'x')  
>>> k  
'xxxxxxxxankaraxxxxxxx'
```

center metodunun ikinci parametresi yalnızca tek karakterden oluşan bir string olmak zorundadır. Eğer center metodunda belirtilen alan uzunluğu yazının uzunluğundan kısa ise bu durumda yazının aynısı geri döndürülür.

find metodunu yazı içerisinde bir karakteri ya da yazıyı arar. Bulursa ilk bulduğu yerin indeks değeri ile, bulamazsa -1 değeri ile geri döner. Örneğin:

```
>>> s = 'ankara'  
>>> result = s.find('k')  
>>> result  
2
```

```
>>> result = s.find('ar')
>>> result
3
```

Find metodunu iki argüman girerek de çağrılabılır. İkinci argüman aramanın hangi indeks'ten başlanarak yapılacağını belirtir. Örneğin:

```
>>> s = 'mal sahibi mülk sahibi hani bunun ilk sahibi'
>>> result = s.find('sahibi', 15)
>>> result
16
>>> s[result]
's'
>>> s[result:]
'sahibi hani bunun ilk sahibi'
```

Metot üç argümanla çağrılsa ikinci argüman aramanın başlatılacağı indeksi, üçüncü argüman aramanın bitirileceği indeksi belirtir. Aramanın bitirileceği indeks aramaya dahil değildir. Örneğin:

```
>>> s = 'ankara'
>>> result = s.find('a', 1, 4)
>>> result
3
```

index metodu da find metodu ile aynı işlemi yapmaktadır. Ancak index metodu aranan karakteri ya da yazıyı bulamazsa exception oluşturur. Örneğin:

```
>>> s = 'ankara'
>>> n = s.index('m')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    s.index('m')
ValueError: substring not found
```

Yine index metodu da iki argümanlı ve üç argümanlı olarak kullanılabilir. Örneğin:

```
>>> s = 'ankara'
>>> result = s.index('a', 2)
>>> result
3
```

rfind fonksiyonu find fonksiyonu ile aynı şeyleri yapmaktadır. Ancak rfind yazı içerisinde aranan karakter ya da yazının son bulunduğu yerin indeksiyle geri döner. Örneğin:

```
>>> s = 'ankara'
>>> n = s.rfind('a')
>>> n
5
```

rindex de rfind ile aynı işlemi yapmaktadır. Ancak rindex aranan karakter ya da yazı bulunamazsa -1 ile geri dönmez exception oluşturur. Örneğin:

```
>>> s = 'ankara'
>>> n = s.rindex('m')
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    s.rindex('m')
ValueError: substring not found
```

`rfind` ve `rindex` metotları da iki argümanlı ve üç argümanlı kullanılabilir. Metotların ikinci ve üçüncü argümanları her zaman baştan itibaren indeks belirtmektedir. Örneğin:

```
>>> s = 'ankara'  
>>> result = s.rfind('a', 2, 4)  
>>> result  
3
```

`count` metodu belli bir yazının asıl yazı içerisinde kaç defa bulunduğu bilgisini bize verir. Örneğin:

```
>>> s = 'ankara'  
>>> c = s.count('a')  
>>> c  
3
```

Örneğin:

```
>>> s = 'ankara ankara güzel ankara'  
>>> result = s.count('ankara')  
>>> result  
3
```

`count` metodu iki argümanla çağrılsa ikinci argüman aramanın nereden başlatılacağını belirtir. Örneğin:

```
>>> s = 'ankara ankara güzel ankara'  
>>> result = s.count('ankara', 6)  
>>> result  
2
```

`count` metodu üç argümanla da çağrılabılır. Bu durumda ikinci argüman aramanın başlatılacağı indeksi, üçüncü argüman ise aramanın bitirileceği indeksi belirtir. Aramanın bitirileceği indeks aramaya dahil değildir. Örneğin:

```
>>> s = 'ankara ankara güzel ankara'  
>>> result = s.count('ankara', 6, 15)  
>>> result  
1
```

`str` sınıfının bir grup başı `is` ile başlayan `isxxx` biçiminde metodu vardır. Bu metodlar yazıyı test edip bool bir değere geri dönerler. Sınıfın `isalpha`, `isdigit` ve `isalnum` metodları bir yazının tüm karakterlerinin sırasıyla alfabetik mi, nümerik mi, alfanümerik mi olduğunu anlamak için kullanılmaktadır. Örneğin:

```
>>> s = 'ankara06'  
>>> s.isalpha()  
False  
>>> s.isalnum()  
True  
>>> s.isdigit()  
False
```

`islower` ve `isupper` metodları da büyük harf ve küçük harf kontrolünü yapmaktadır. `isspace` metodu ise yazının yalnızca boşluk karakterlerinden oluşup olmadığı bilgisini bize verir. Örneğin:

```
>>> s = ' '  
>>> s.isspace()  
True
```

Sınıfın diğer `isxxx` metodlarını Python Standart Kütüphane dokümanlarından inceleyebilirsiniz.

str sınıfının join isimli metodunu string'lerden oluşan dolaşılabilir bir nesneyi (örneğin liste olabilir, demet olabilir, kümeler olabilir) parametre olarak alır ve bu dolaşılabilir nesnedeki yazıları bunların arasında join metodunun çağrılmasında kullanılan yazıyı yerleştirecek birleştirir. Örneğin:

```
>>> s = ','  
>>> k = s.join(['ali', 'veli', 'selami'])  
>>> k  
'ali,veli,selami'
```

Tabii aynı işlem şöyle de yazılabildi:

```
>>> s = ','.join(['Ali', 'Veli', 'Selami'])  
>>> s  
'Ali,Veli,Selami'
```

Örneğin:

```
>>> s = ','.join('ankara')  
>>> s  
'a,n,k,a,r,a'
```

Örneğin:

```
>>> s = '+'.join('1234')  
>>> s  
'1+2+3+4'
```

Örneğin:

```
>>> s = ''.join(['ali', 'veli', 'selami'])  
>>> s  
'aliveveliselami'
```

Tabii join metodunu tek karakterli bir string ile kullanılmak zorunda değildir. Örneğin:

```
>>> s = 'xxx'.join(['ali', 'veli', 'selami'])  
>>> s  
'alixxxvelixxxselami'
```

str sınıfının split metodunu yazımı belli bir karakteri temel alarak parçalara ayırır ve onu bize bir liste biçiminde verir. Yani adeta join metodıyla ters işlem yapmaktadır. Örneğin:

```
>>> s = 'ali,veli,selami,ayşe,fatma'  
>>> k = s.split(',')  
>>> k  
['ali', 'veli', 'selami', 'ayşe', 'fatma']
```

Örneğin:

```
>>> s = 'ali, veli, selami, ayşe, fatma'  
>>> k = s.split(',')  
>>> k  
['ali', ' veli', ' selami', ' ayşe', ' fatma']
```

Burada boşluk bir ayıracı karakteri olarak kullanılmamıştır. Tabii biz ayıracı birden fazla karakterden oluşan bir biçimde de alabiliz. Örneğin:

```
>>> s = 'ali, veli, selami, ayşe, fatma'  
>>> k = s.split(', ')
```

```
>>> k  
['ali', 'veli', 'selami', 'ayşe', 'fatma']
```

split argümansız kullanılırsa tüm boşluk karakterlerini atarak ayırma yapar. Örneğin:

```
>>> s = 'bugün    hava çok     güzel'  
>>> k = s.split(' ')  
>>> k  
['bugün', '', '', '', 'hava', 'çok', '', '', '', 'güzel']  
>>> k = s.split()  
>>> k  
['bugün', 'hava', 'çok', 'güzel']
```

Örneğin:

```
>>> date = '13/12/2008'  
>>> s = date.split('/')  
>>> s  
['13', '12', '2008']  
>>> t = tuple(date.split('/'))  
>>> t  
('13', '12', '2008')
```

Örneğin:

```
>>> s = 'ali,veli,,,selamiayşe,fatma'  
>>> k = s.split(',')  
>>> k  
['ali', 'veli', '', '', 'selamiayşe', 'fatma']
```

Bu örnekte yazı ',' karakterlerinden ayrıştırılmak istenmiştir. ',' karakterlerinin arasında hiçbir karakter olmadığı durumda boş string'lerin elde edildiğine dikkat ediniz.

split metodu iki argümanla da çağrılabılır. Bu durumda ikinci argüman yazının toplamda kaç parçaaya ayrılacağını belirtmektedir. Yazı en fazla ikinci parametresiyle belirtilen sayıdan bir fazla sayıda parçaaya ayrılır. Örneğin:

```
>>> s = 'ali, veli, selami, ayşe, fatma'  
>>> k = s.split(', ', 2)  
>>> k  
['ali', 'veli', 'selami, ayşe, fatma']
```

Burada yazının iki parçaaya ayrıldığına ve listenin son elemanın geri kalan tüm yazısından oluştuğuna dikkat ediniz.

split metodunun rsplit isminde bir biçimi de vardır. rsplit eğer tek argümanla çağrılırsa tamamen split gibi çalışır. Ancak iki argümanla çağrılırsa ayırma işlemini sağdan sola yapmaktadır. Örneğin:

```
>>> path = '/home/kaan/study/python/sample.py'  
>>> s = path.rsplit('/')  
>>> s  
[ '', 'home', 'kaan', 'study', 'python', 'sample.py']  
>>> s = path.rsplit('/', 1)  
>>> s  
[ '/home/kaan/study/python', 'sample.py']  
>>> s = path.rsplit('/', 2)  
>>> s  
[ '/home/kaan/study', 'python', 'sample.py']  
>>> s = path.split('/', 2)  
>>> s  
[ '', 'home', 'kaan/study/python/sample.py']
```

Burada split ile rsplit arasındaki farka dikkat ediniz. rsplit ayırtmayı tersten yapıyor ve listeye tersten yerleştiriyor.

split ile join metotları adeta mantıksal bakımdan ters işlemler yapmaktadır. Örneğin:

```
>>> s = 'ali, veli, selami, ayşe, fatma'  
>>> a = s.split(', ')  
>>> k = ', '.join(a)  
>>> s  
'ali, veli, selami, ayşe, fatma'  
>>> k  
'ali, veli, selami, ayşe, fatma'
```

str sınıfının partition isimli metod parametre olarak bir string alır ve bu string'i üç parçaya ayırır ve bize o üç parçayı bir demet olarak verir. Birinci parça argüman olarak verilen yazının solundaki kısımdan, ikinci parça argüman olarak verilen yazidan, üçüncü parça da argüman olarak verilen yazının sağındaki kısımdan oluşur. Örneğin:

```
>>> s = 'aliveleiselami'  
>>> k = s.partition('veli')  
>>> k  
('ali', 'veli', 'selami')
```

Örneğin:

```
>>> s = 'aliveleiselami'  
>>> left, middle, right = s.partition('veli')  
>>> print(left, middle, right)  
ali veli selami
```

Tabii yazı ana yazının birden fazla yerinde varsa partition metod ilk bulduğu yerden ayırma yapmaktadır. Örneğin:

```
>>> s = 'aliveleiselamivelisacit'  
>>> k = s.partition('veli')  
>>> k  
('ali', 'veli', 'selamivelisacit')
```

Argüman olarak verilen yazı bulunamazsa verilen demetin ilk elemanı tüm yazıyı ikinci ve üçüncü elemanları boş bir yazayı içerir. Örneğin:

```
>>> s = 'ankaraizmirbursa'  
>>> k = s.partition('imzir')  
>>> k  
('ankaraizmirbursa', '', '')
```

Sınıfın replace metod yaz içerişindeki belli bir kısmı başka bir yazıyla değiştirerek bize değiştirilmiş yeni yazıyı verir. Örneğin:

```
>>> s = 'istanbul içanadolu bölgесindedir ve istanbul ikinci büyük şehirdir'  
>>> k = s.replace('istanbul', 'ankara')  
>>> k  
'ankara içanadolu bölgесindedir ve ankara ikinci büyük şehirdir'
```

str sınıfının startswith metod yazının başının belli bir yazıyla başlayıp başlamadığını belirlemek için kullanılır. Örneğin:

```
>>> s = 'ankara'  
>>> result = s.startswith('an')  
>>> result  
True  
>>> result = s.startswith('is')  
>>> result
```

False

`endswith` metodu ise yazının sonunun belli bir yazıyla bitip bitmediğini belirlemek için kullanılmaktadır. Örneğin:

```
>>> s = 'ankara'  
>>> result = s.endswith('ra')  
>>> result  
True  
>>> result = s.endswith('is')  
>>> result  
False
```

Daha önceleri de gördüğümüz gibi iki string "+" operatörüyle toplama işlemine sokulabilir. Bu durumda iki string'in birleşmesinden oluşan yeni bir string elde edilir. Örneğin:

```
>>> s = 'ali'  
>>> k = 'veli'  
>>> z = s + k  
>>> z  
'alivelei'
```

Java, C# gibi bazı programlama dillerinde "+" operatörünün bir operandı string ise diğer operand string olmayabilmektedir. O dillerde diğer operandın string'e dönüştürülmesi o dillerin derleyicileri tarafından otomatik olarak yapılmaktadır. Ancak Python'da durum böyle değildir. "+" operatörünün bir operandı string ise diğer operandı da string olmak zorundadır. Örneğin:

```
>>> a = 10
>>> print('a = ' + str(a))
a = 10
```

str sınıfı değiştirilemez olduğu için `+=` operatörü "mevcut string'in sonuna ekle" anlamına gelmemektedir. Örneğin:

```
s = 'ali'  
s += 'veli'
```

İşlemi ile aşağıdaki tamamen eşdeğerdir:

```
s = 'ali'  
s = s + 'veli'
```

* operatörü yine çöklama (repititon) amaçlı kullanılabilir. Örneğin:

Örneğin:

```
>>> s = 'ankara'  
>>> k = 2 * s  
>>> k  
'ankaraankara'  
>>> k = 2 * s * 3  
>>> k  
'ankaraankaraankaraankaraankaraankaraankara'
```

Örneğin:

```
>>> s = 'ankara'  
>>> s *= 3  
>>> s  
'ankaraankaraankara'
```

Tabii str türü değiştirilemez olduğu için bu işlem sona ekleme anlamına gelmemektedir. Örneğin:

```
>>> s = 'ankara'  
>>> id(s)  
4470509104  
>>> s *= 3  
>>> id(s)  
4470629120  
>>> s  
'ankaraankaraankara'
```

upper ve lower metotları ise yazının tüm karakterlerini büyük harfe ya da küçük harfe dönüştürür. Örneğin:

```
>>> s = 'AnKaRa'  
>>> k = s.upper()  
>>> k  
'ANKARA'  
>>> k = s.lower()  
>>> k  
'ankara'
```

str sınıfının rstrip metodu yazının sağındaki boşluk karakterlerini, lstrip metodu solundaki boşluk karakterlerini ve strip metodu da hem sağındaki hem solundaki boşlukları atar. Tabii bu metotlar asıl nesne üzerinde değişiklik yapmazlar boşlukları atılmış yeni bir string nesnesi verirler. Örneğin:

```
>>> s = '    bugün hava çok soğuk    '  
>>> k = s.rstrip()  
>>> m = s.lstrip()  
>>> z = s.strip()  
>>> print(':' + k + ':')  
:    bugün hava çok soğuk:  
>>> print(':' + m + ':')  
:bugün hava çok soğuk :  
>>> print(':' + z + ':')  
:bugün hava çok soğuk:
```

Örneğin:

```
>>> s = '    bugün hava çok güzel    '  
>>> k = ' '.join(s.split())  
>>> k  
'bugün hava çok güzel'
```

String'ler de karşılaştırma operatörleriyle karşılaştırma işlemine sokulabilir. Örneğin:

```
if passwd == 'maviay':  
    ...
```

Örneğin:

```
>>> s = 'ankara'  
>>> k = 'izmir'  
>>> s == k  
False  
>>> s != k
```

```
True  
>>> s > k  
False  
>>> s < k  
True
```

Büyükük küçüklük karşılaştırması UNICODE tablo dikkate alınarak sözlükteki sıraya göre yapılmaktadır. UNICODE tabloda büyük harflerin küçük harflerden daha önce geldiğini anımsayınız. Örneğin:

```
>>> s = 'Ankara'  
>>> k = 'ankara'  
>>> s > k  
False  
>>> s < k  
True
```

Örneğin:

```
>>> s = 'ali'  
>>> k = 'aliye'  
>>> k > s  
True
```

str sınıfının format isimli metodu yazıyı formatlamak için kullanılmaktadır. Normal formatlamada asıl yazının içerisindeki {n} biçiminde bulunan kalıplar format metodunki parametrelerle pozisyonel olarak eşleştirilir. Pozisyon indeksi 0'dan başlamaktadır. Örneğin:

```
>>> a = 10  
>>> b = 20  
>>> s = 'a = {0}, b = {1}'.format(a, b)  
>>> s  
'a = 10, b = 20'
```

Burada {0} ve {1} birer yer tutucudur. format metodundaki argümanlar sırasıyla bu yer tutucularla eşleştirilir ve yeni bir yazı elde edilir.

a = 10
b = 20
a = {0}, b = {1}.format(a, b)

Tabii yazı içerisinde yer tutucular birden fazla yerde bulunabilirler. Örneğin:

```
>>> s = 'a = {0}, b = {1}, a + b = {0} + {1}'.format(a, b)  
>>> s  
'a = 10, b = 20, a + b = 10 + 20'
```

Eğer yazı içerisinde bu format karakterleri yalnızca bir kez kullanılacaksa küme parantezlerinin içi boş da bırakılabilir. Bu durumda sıralı eşleştirme yapılır. Örneğin:

```
>>> s = 'a = {}, b = {}'.format(a, b)  
>>> s  
'a = 10, b = 20'
```

İçin boş küme parantezi ile dolu küme parantezi birlikte kullanılamaz. Örneğin:

```
>>> 'a = {}, b = {}'.format(a, b)  
Traceback (most recent call last):
```

```
File "<pyshell#191>", line 1, in <module>
  'a = {}, b = {}'.format(a, b)
ValueError: cannot switch from automatic field numbering to manual field specification
```

Python'a 3.6 ile birlikte "string interpolasyonu" özelliği de eklenmiştir. String interpolasyonu string'lerin başlarına onlarla yapışık bir biçimde f ya da F harfi getirilerek oluşturulur. Böyle string'lerde kümeye parantezlerinin içerisinde herhangi bir ifade yazılabilmektedir. Bu ifadelerin değerleri Python yorumlayıcıları tarafından o anda hesaplanarak kümeye parantezlerinin yerine yerleştirilir. Örneğin:

```
>>> a = 10; b = 20
>>> s = f'a = {a}, b = {b}'
>>> s
'a = 10, b = 20'
```

Örneğin:

```
'a = 10, b = 20'
>>> a = 10; b = 20
>>> s = f'a square = {a * a}, b square = {b * b}'
>>> s
'a square = 100, b square = 400'
```

del Deyimi

Python'da değişkenler ve bazı veri yapılarının elemanları del deyimi ile silinebilmektedir. del deyiminin genel biçimi şöyledir:

del <değişken listesi>

del deyimi ile biz global ya da yerel değişkenleri silebiliriz. (Yerel değişkenler konusu sonraki bölümlerde ele alınmaktadır.) Örneğin:

```
>>> x = 10
>>> del x
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

Gördüğünüz gibi bir değişken del deyimi ile silindiğinde o değişken hiç yaratılmamış gibi bir durum oluşmaktadır. Tek bir del deyimi ile birden fazla değişkeni silebiliriz. Örneğin:

```
>>> a = 10
>>> b = 20
>>> del a, b
```

del deyimi ile nesneler değil değişkenler silinmektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = a
>>> del a
```

Burada a ve b değişkenleri aynı list nesnesini gösteriyor durumdadır. del a işlemi ile a değişkeninin gösterdiği list nesnesi değil, a değişkeni silinmiştir. Kursumuzun ilerleyen bölümümüzlerinde bir nesnenin hiçbir değişken tarafından gösterilmediği durumda Python'ın çöp toplayıcı ("garbage collector") mekanizmasının nesneyi sildiğini göreceksiniz.

del deyimi ile değiştirilebilir (mutable) bazı veri yapılarının elemanları da silinebilmektedir. Örneğin listelerin elemanlarını del deyimi ile silebiliriz:

```
>>> a = [1, 2, 3, 4, 5]
>>> del a[3]
>>> a
[1, 2, 3, 5]
```

Birden fazla liste elemanı del ile silinirken silme işlemi soldan sağa doğru sırayla yapılmaktadır. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> del a[0], a[2]
>>> a
[2, 3, 5]
```

Burada önce listenin 0'inci indeksli elemanı silinmiştir. Ondan sonra 2'nci indeksli elemanı silinmiştir. Silme işleminin tek hamlede birlikte yapılmadığını dikkat ediniz. Listelerde eleman silme işleminin pop ve remove metotları ile de yapılabildiğini anımsayınız.

Liste elemanlarının silinmesinde dilimleme sentaksı da kullanılabilirmektedir. Örneğin:

```
>>> a = [1, 2, 3, 4, 5]
>>> del a[2:4]
>>> a
[1, 2, 5]
```

Aynı işlemin dilimlenen kısma boş bir dolaşılabilir nesnenin atanmasıyla da yapılabildiğini görmüştük:

```
>>> a = [1, 2, 3, 4, 5]
>>> [2:4] = []
>>> a
[1, 2, 5]
```

Benzer biçimde del deyimi ile sözlük elemanlarını da anahtara bağlı olarak silebiliriz. Örneğin:

```
>>> a = {'ali': 1, 'veli': 2, 'selami': 3, 'ayşe': 4, 'fatma': 5}
>>> d = {'ali': 1, 'veli': 2, 'selami': 3, 'ayşe': 4, 'fatma': 5}
>>> del d['ali'], d['fatma']
>>> d
{'veli': 2, 'selami': 3, 'ayşe': 4}
```

Sözlük elemanlarının anahtara dayılı olarak dict sınıfının pop metodu ile de silinebildiğini anımsayınız.

tuple, str gibi değiştirilemez (immutable) türlerin elemanları del deyimi ile silinmez. Örneğin:

```
>>> s = 'ali'
>>> del s[0]
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    del s[0]
TypeError: 'str' object doesn't support item deletion
```

Örneğin:

```
>>> t = 10, 20
>>> del t[0]
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    del t[0]
```

```
TypeError: 'tuple' object doesn't support item deletion
```

set türü değiştirilebilir olmasına karşın biz kümelerin elemanlarını del deyimi ile silemeyez. set sınıfında elemanlar arasında bir sıralama ilişkisinin olmadığını dolayısıyla da set nesnesinin elemanlarına [...] operatörüyle erişilemediğini anımsayıınız. Örneğin:

```
>>> a = {1, 2, 3, 4, 5}
>>> del a[1]
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    del a[1]
TypeError: 'set' object doesn't support item deletion
```

Farklı Türlerin Birbirleriyle Karşılaştırılması

Python'da farklı türler her zaman birbirleriyle == ve != operatörü kullanılarak karşılaştırma işlemeye sokulabilir. int, float ve bool türlerinin dışında farklı türlerin == operatörüyle karşılaştırılmaları her zaman False, != operatörüyle karşılaştırılmasları ise her zaman True değerini vermektedir. Örneğin:

```
>>> 'ali' == 45
False
>>> 10 == [1, 2, 3, 4]
False
>>> 10 != [1, 2, 3, 4]
True
```

Örneğin:

```
>>> a = (1, 2, 3)
>>> b = [1, 2, 3]
>>> a == b
False
>>> a != b
True
```

Anımsanacağı gibi int, float ve bool türleri birbirinden farklı olmasına karşın karşılaştırma işlemeye sokulabilmektedir. Bu durumda karşılaştırmanın sayısal düzeyde yapıldığını anımsayıınız.

int, float ve bool dışındaki farklı türlerin >, <, >= ve <= operatörleriyle karşılaştırılması exception'a yol açmaktadır. Örneğin:

```
>>> a > b
Traceback (most recent call last):
  File "<pyshell#199>", line 1, in <module>
    a > b
TypeError: '>' not supported between instances of 'tuple' and 'list'
```

Örneğin:

```
>>> 10 > [1, 2, 3]
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    10 > [1, 2, 3]
TypeError: '>' not supported between instances of 'int' and 'list'
```

Aynı türlerin karşılaştırılması o sınıfların operatör metotları tarafından onlara özel bir biçimde yapılmaktadır. Operatör metotları kursumuzun sonlarına doğru ele alınmaktadır. Örneğin biz iki string'i karşılaştırdığımızda aslında bu string'lerin adreslerini değil, bu stirng'lerin içerisinde aynı yazılar olup olmadığını karşılaştırmış oluruz:

```
>>> s = 'ankara'  
>>> k = s[2:]  
>>> m = 'kara'  
>>> k  
'kara'  
>>> m  
'kara'  
>>> id(k)  
2343926447024  
>>> id(m)  
2343926446448  
>>> k == m  
True
```

Aynı Türden Veri Yapılarının Birbirleriyle Karşılaştırılması

Aynı türden iki listenin ya da demetin karşılaştırılması string karşılaştırmasında olduğu gibi leksikografik biçimde yapılmaktadır. Yani elemanlar aynı olduğu sürece ilerlenir. İlk farklı elemanın büyükük küçükük durumuna bakılır. Örneğin:

```
>>> a = [1, 2, 3]  
>>> b = [1, 2, 3]  
>>> a == b  
True  
>>> a != b  
False
```

Örneğin:

```
>>> a = [1, 2, 3]  
>>> b = [1, 2, 4]  
>>> a == b  
False  
>>> a > b  
False  
>>> b > a  
True
```

Örneğin:

```
>>> a = (1, 2, 3)  
>>> b = (1, 2, 3, 4)  
>>> a > b  
False  
>>> b > a  
True
```

Karşılaştırma sırasında liste ya da demet uzunlıklarının eşit olması gerekmektedir. Örneğin:

```
>>> [10, 20, 30, 40] > [10, 20, 30]  
True
```

>, <, >=, <= operatörleriyle karşılaştırma yapılırken listenin karşılıklı elemanları karşılaştırılamaz türlendense exception oluşacaktır. Örneğin:

```
>>> [10, 20, 'ali'] > [10, 20, 30]  
Traceback (most recent call last):  
  File "<pyshell#269>", line 1, in <module>  
    [10, 20, 'ali'] > [10, 20, 30]  
TypeError: '>' not supported between instances of 'str' and 'int'
```

Burada 'ali' ile 30 '>' ile karşılaştırılamadığı için exception oluşmuştur.

Tabii listenin ya da demetin karşılıklı elemanları farklı türlerden olabilir. Bu durumda o elemanın == ile karşılaştırılması False, != ile karşılaştırılması True değerini verecektir. Elemanların farklı türlerden olması durumunda == ve != operatörünün dışındaki operatörlerle karşılaştırma exception'a yol açacaktır. Örneğin:

```
>>> a = (1, 2, 3)
>>> b = (1, 2, 3, 4)
>>> a > b
False
>>> b > a
True
```

Örneğin:

```
>>> a = (1, 2, 3)
>>> b = (1, 2, 'Ali')
>>> a == b
False
>>> a != b
True
>>> a > b
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    a > b
TypeError: '>' not supported between instances of 'int' and 'str'
```

Örneğin:

```
>>> a = [1, 'Ali', 2]
>>> b = [1, 'Aliye', 2]
>>> a < b
True
```

Tabii int, float ve bool türleri kendi aralarında karşılaştırma işlemine sokulabilmektedir. Yani listenin ya da demetin karşılıklı elemanları bu türlerdense sorun olusmaz. Örneğin:

```
>>> a = [1, 2, 3]
>>> b = [1, 2.0, 3.0]
>>> a == b
True
```

Örneğin:

```
>>> a = [1, 2, 3]
>>> b = [1, 2.0, 4.0]
>>> a > b
False
>>> a < b
True
```

İki küme == ve != operatörleriyle karşılaştırılabilir. Bu durumda iki kümenin tamamen aynı elemanlara sahip olup olmadığına bakılmaktadır. Örneğin:

```
>>> a = {1, 2, 2, 1, 3, 3, 4, 2}
>>> b = {4, 3, 1, 2}
>>> a == b
True
```

İki kümenin == ve != operatörleri dışındaki diğer karşılaştırma operatörleriyle karşılaştırılması alt küme işlemi yapmaktadır. Bu konu kümelerin anlatıldığı bölümde ele alınmıştır.

İki sözlük == ve != operatörleriyle karşılaştırıldığında anahtar değer çiftleri karşılaştırılmaktadır. Yani adeta anahtar değer çiftleri sanka birer demet gibi düşünülerek karşılaştırma yapılır. Örneğin:

```
>>> a = {1: 'Ali', 2: 'Veli'}
>>> b = {1: 'Ali', 2: 'Selami'}
>>> a == b
False
```

Sözlüklerde diğer karşılaştırma operatörlerinin anlamı yoktur. Bu operatörler exception oluşmasına yol açar.

Python Programlarının Çalıştırılması

Biz şimdiye kadar hep komut satırında çalıştık. Komut satırında komutlar tek tek verilmektedir. İşte komutlar bir araya getirilip bir dosyaya yazılırsa Python yorumlayıcısı tarafından işletilebilir. Biz bir Python program dosyasını komut satırında "python" ya da "python3" isimli Python yorumlayıcılarıyla çalıştırabiliriz. Windows'ta eğer biz Python'ın 3'lü versiyonlarını kurmuşsa "python.exe" programı Python yorumlayıcısının 3'lü versiyonunu çalıştmaktadır. Ayrıca bir "python3.exe" programı yoktur. Ancak Linux sistemlerinde genel olarak iki ayrı python programı bulunabilmektedir. "python" isimli program bu sistemlerde 2'li versiyonları, "python3" isimli program da 3'lü versiyonları çalıştmak için kullanılmaktadır.

Normalde "python" ya da "python3" programı karşılıklı etkileşimli (interactive) komut satırına geçiş yapar. Yani biz aslında Python'da komut satırında çalışmak için IDLE gibi bir IDE'ye sahip olmak zorunda değiliz. Örneğin:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC>python
Python 3.6.4 (v3.6.4:d48ebeb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 10
>>> x
10
>>>
```

Aynı şeyi Linux sistemlerinde yapalım:

```
csd@csd-virtual-machine ~ $ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
csd@csd-virtual-machine ~ $ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

İşte bir Python programını komut satırından çalıştmak için Python programının yazılı olduğu kaynak dosyası "python" ya da "python3" programlarına komut satırı argümanı olarak veririz. Bu durumda Python yorumlayıcısı dosyanın içerisindeki programı komut satırına düşmeden doğrudan çalıştıracaktır. Örneğin:

```
python sample.py
```

Tabii bu çalışma için bizim öncelikle kodumuzu sample.py isimli bir dosyada saklamamız gerekmektedir. Python program dosyalarının uzantısı ".py" biçimindedir. Biz Python programımızı herhangi bir editörle yazıp save edebiliriz. Python kaynak dosyası genel olarak UNICODE UTF-8 kodlamasıyla kodlanmış olmalıdır. (Standart ASCII kodlamısının zaten UTF8 uyumu olduğunu anımsayınız.). Şimdi biz Windows'ta "Notepad.exe" isimli editörü kullanarak basit Python programı yazıp onu komut satırında çalıştırıralım:

sample.py

```
a = 10  
b = 20  
c = a + b  
print(c)
```

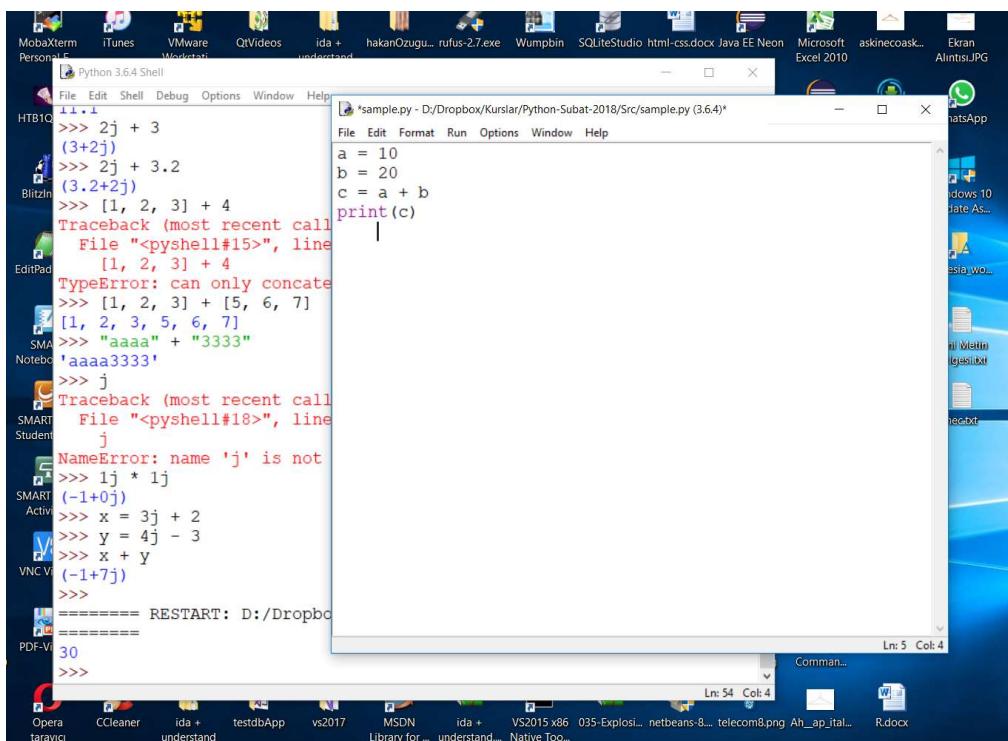
```
D:\Dropbox\Kurslar\Python-Subat-2018\Src>python sample.py  
30
```

Benzer biçimde Linux sistemlerinde de çalıştırmayı şöyle yapabiliriz:

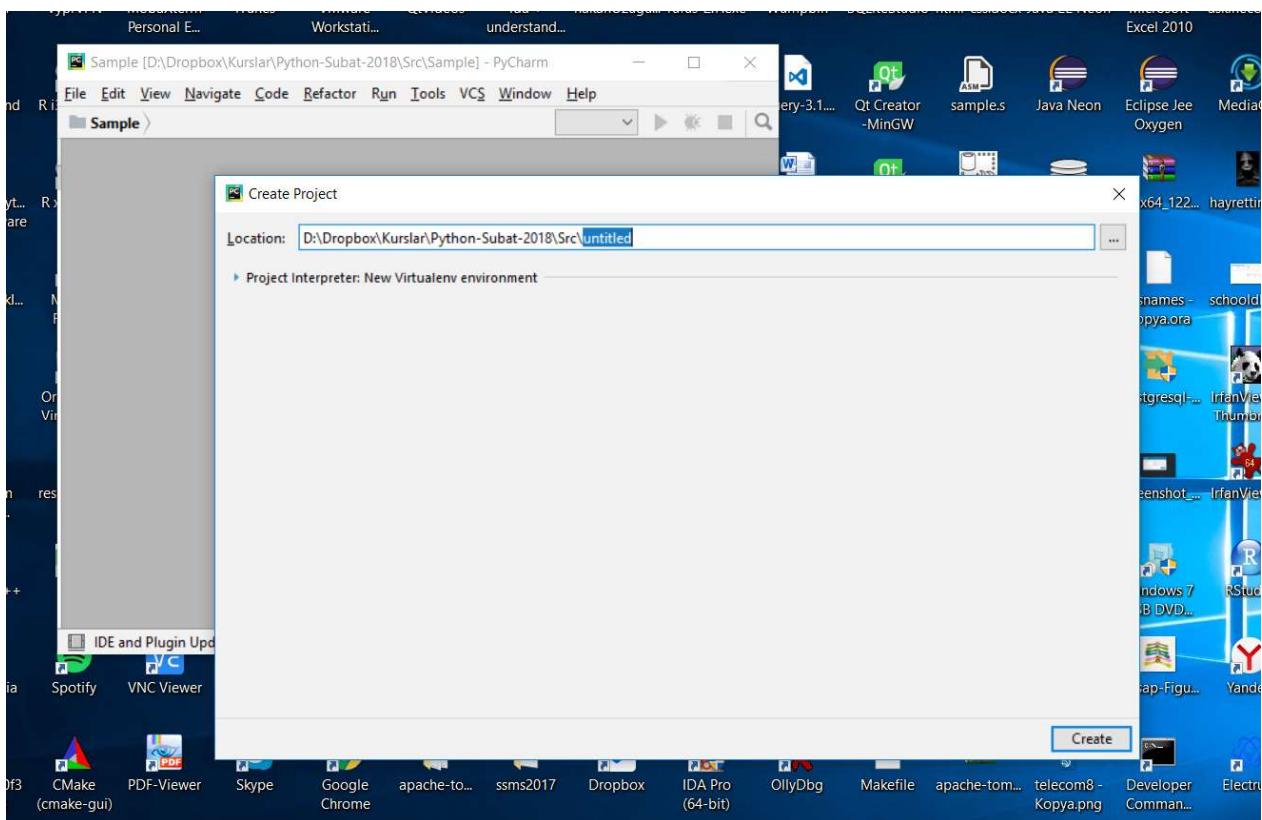
```
csd@csd-virtual-machine ~ $ python3 sample.py  
30
```

Tabii IDE'ler program yazma ve çalışma işlemlerinde bize çok kolaylıklar sağlamaktadır. Bu nedenle pratikte Python programları bu IDE'ler kullanılarak yazılıp çalıştırılırlar.

IDLE IDE'sinde çalışma daha kolaydır. Tek yapılacak şey ".py" uzantılı bir program dosyası oluşturmak programı onun içerisinde yazmak ve menüden Run/Run Module (kısa yol tuşu F5) seçeneğini seçmektir. Örneğin:

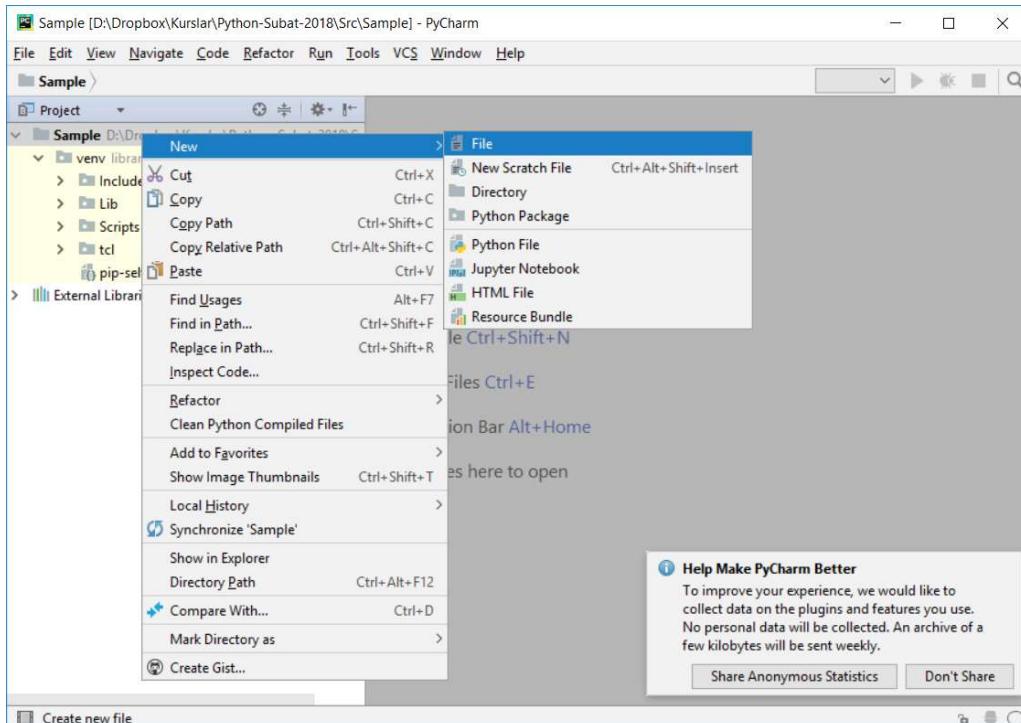


PyCharm IDE'si IDLE IDE'sine göre çok daha yeteneklidir. PyCharm IDE'sinde çalışmak için önce bir proje oluşturmak gereklidir. Proje oluşturma işlemi File/New Project menüsü seçilerek yapılır. PyCharm'da her proje ayrı bir dizinde bulundurulmaktadır. Proje oluşturulurken IDE bizden proje bilgilerinin saklanacağı klasörü belirlememizi ister:

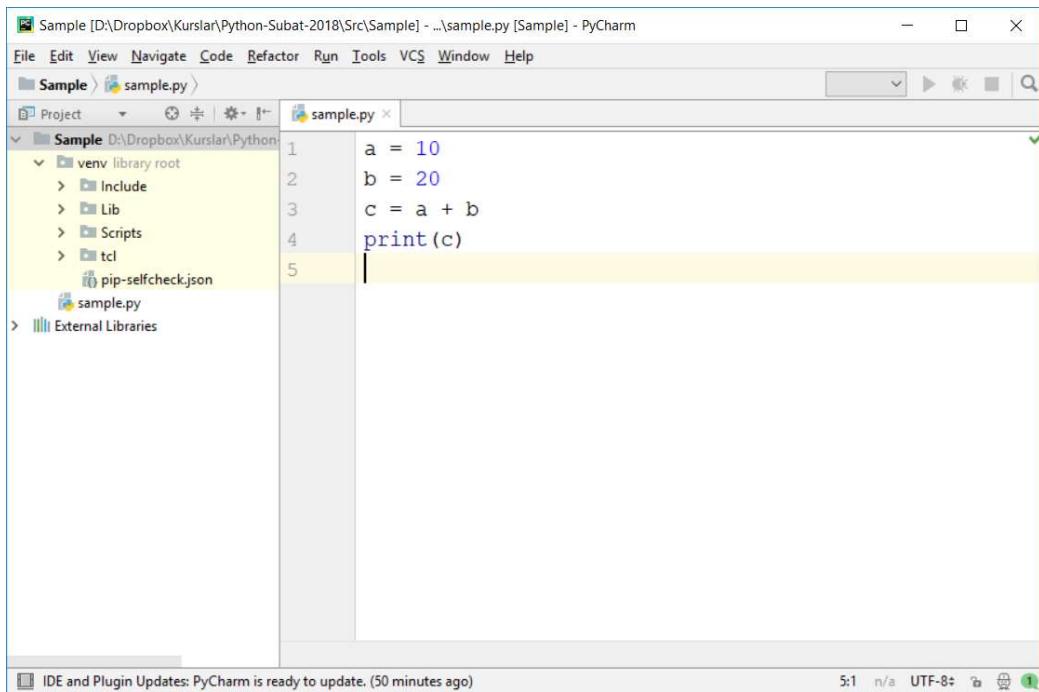


Burada girilecek dizinin var olması gerekmektedir. Zaten PyCharm burada girdiğimiz isimli dizini kendisi yaratacaktır.

Proje yaratıldıktan sonra artık projeye bir Python kaynak dosyası eklemek gereklidir. Bunun için fare ile projenin üzerine gelinip bağlam menüsünden New/File seçilir.

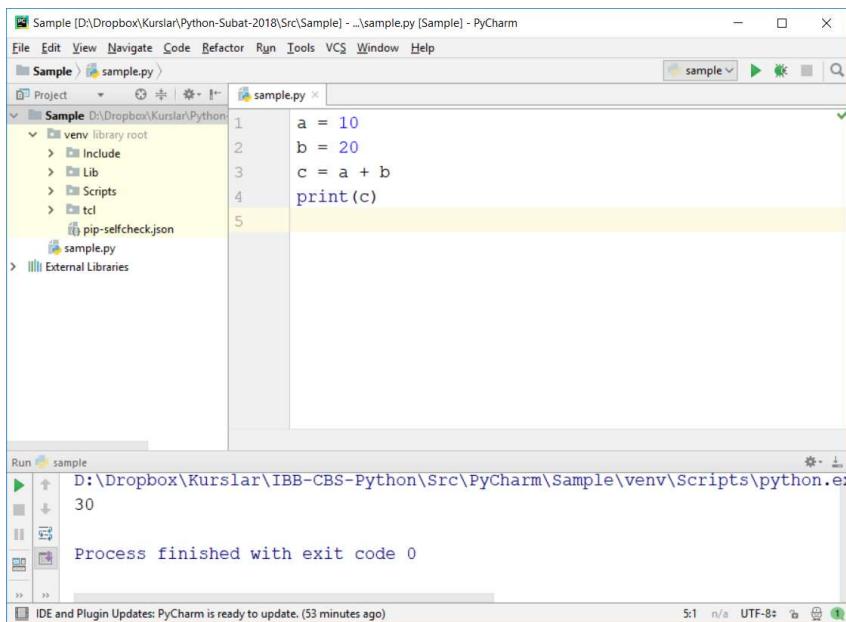


Dosyaya isim verilip dosya proje eklenir. Sonra dosyanın içerisinde Python programı yazılır:



```
a = 10
b = 20
c = a + b
print(c)
```

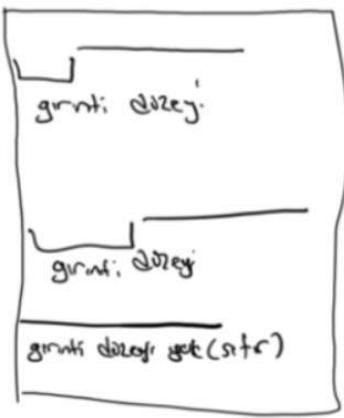
Artık program Run/Run menüsü (Alt + Shift + F10) seçilerek çalıştırılır:



```
D:\Dropbox\Kurslar\IBB-CBS-Python\Src\PyCharm\Sample\venv\Scripts\python.exe sample.py
Process finished with exit code 0
```

Satırların Girinti (Indent) Düzeyleri

Bir satırın başından ilk boşluk olmayan karaktere kadarki uzaklığı girinti düzeyi (indent level) denilmektedir. Python'da girinti düzeyini oluşturmak için SPACE ve TAB karakterleri kullanılmaktadır.



Python'da girinti oluşturmak için SPACE ve TAB karakterlerinin bir arada kullanılması tavsiye edilmemektedir. Programcı girinti oluşturmak için ya hep SPACE ya da hep TAB kullanmalıdır. Ancak SPACE ve TAB'in bir arada kullanılması da error oluşturmaz. Bir satırın girinti düzeyi şöyle hesaplanmaktadır: Önce TAB karakterlerin 8'in katlarına tamamlanması için onlar 1 ile 8 arasında SPACE karakterine dönüştürülür. Sonra bu SPACE karakterlerinin toplam sayısı girinti düzeyini verir. Yani başka bir deyişle satırın toplam girinti düzeyi SPACE karakterlerinin sayısıyla hesaplanmaktadır. Fakat bundan önce TAB'lar SPACE karakterlerine dönüştürülmektedir.

Örneğin:

SPACE TAB SPACE SPACE <ilk boşluksuz karakter>

Buradaki ilk TAB'ı 8'e tamamlamak için 7 SPACE gereklidir. Bu TAB 7 SPACE'e dönüştürülür. Sonra bunu izleyen 2 SPACE daha olduğuna göre bu satırın girinti düzeyi = $1 + 7 + 1 + 1 = 10$ 'dur. Örneğin:

TAB SPACE TAB SPACE <ilk boşluksuz karakter>

İlk TAB için 8 SPACE, sonraki TAB için 8'in katına tamamlamak amacıyla 7 SPACE dönüştürmesi yapılır. O halde satırın toplam girinti düzeyi: $8 + 1 + 7 + 1 = 17$ 'dir. Örneğin:

TAB SPACE SPACE SPACE SPACE TAB <ilk boşluksuz karakter>

Bu satırın da girinti düzeyi = $8 + 1 + 1 + 1 + 1 + 4 = 16$ 'dır.

TAB'lar SPACE'lerle yer değiştirdikten sonra girinti düzeylerinin hep 8'in katlarında olduğuna dikkat ediniz.

Tabii programcı hep SPACE ya da hep TAB kullanırsa hesaplama sorunu olmaz. Örneğin:

SPACE SPACE SPACE SPACE <ilk alfabetik karakter>

Bu satırın girinti düzeyi 4'tür. Örneğin:

TAB TAB TAB <ilk alfabetik karakter>

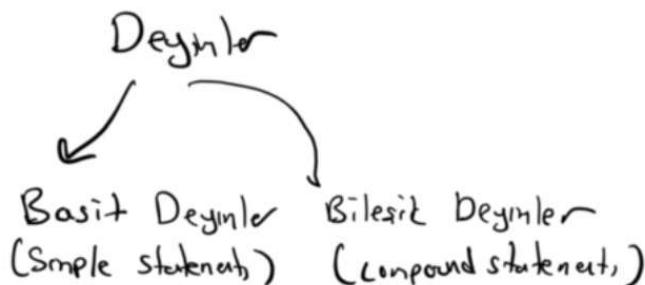
Bu satırın girinti düzeyi de 24'tür. Programcı girinti düzeyleri için hep TAB ya da hep SPACE kullanırsa iki satırın başlangıcı aynı hizaya geldiğinde onların girinti düzeyleri de aynı olmuş olur.

Bazı programlama editörleri ve IDE'ler TAB tuşuna basıldığında TAB karakter yerine N tane SPACE karakteri basmaktadır. Örneğin IDLE , PyCharm ve Spyder IDE'leri default durumda biz TAB tuşuna bastığımızda onun yerine 4 tane SPACE yerleştirmektedir. Bu durumda örneğin biz PyCharm IDE'sinde çalışırken TAB tuşuna bastığımızda aslında bu IDE TAB yerine dosyaya 4 SPACE karakteri yerleştireceğinden dolayı Python yorumlayıcısı zaten dosyada TAB karakter görmeyecektir. Bu durumda yalnızca SPACE karakterlerinden oluşan girintilerin girinti düzeyleri SPACE karakterlerinin sayısı kadar olacaktır. Python yorumlayıcılarının editörler tarafından save edilmiş dosyaları okuduğuna dikkat ediniz.

Pekiyi Python'da satırların girinti düzeylerinin önemi nedir? İşte Python offside dillerdendir. Bu tür dillerde bloklama girintilerle yapılmaktadır. Python'da da kodun sentaks bakımından geçerli olabilmesi için bazı satırların girinti düzeylerinin aynı olması bazı satırların ise önceki satırlardan daha yüksek bir girinti düzeyine sahip olması gerekmektedir. Python programları her zaman sıfır girinti düzeyiyle başlatılmak zorundadır. (Başka bir deyişle Python programları en soldaki sütuna dayalı bir biçimde başlatılmalıdır.)

Python'da Deyimler

Deyim (statements) bir dildeki çalışma birimleridir. Yani bir program aslında deyimlerin çalıştırılmasıyla çalıştırılır. Bir Python programını deyimler topluluğu olarak düşünebiliriz. Python'da deyimler iki gruba ayrılmaktadır:



Python'da tek bir satıra yazılabilen deyimlere basit deyim denilmektedir. Deyimlerin bir grubu böyle tek satıra yazılabilen basit deyimlerden, bir grubu da bileşik deyimlerden oluşmaktadır.

Python'da birden fazla basit deyim istenirse aralarına ';' atomu konularak aynı satıra da yazılabilir. Son basit deyimdeki ';' artık isteğe bağlıdır. Örneğin:

```
a = 10  
b = 20  
c = 30
```

Bu deyimler aşağıdaki gibi tek satıra da yazılabilir:

isteğe bağlı
a = 10; b = 20; c = 30;

Ancak bileşik deyimler ilerde de görüleceği üzere tek satır üzerine yazılamazlar. Basit deyimlerin en çok kullanılanı ifadesel deyimlerdir. Bunlar bizim bildiğimiz ifadelerdir. İfadeler de program içerisinde kullanıldığından birer deyim statüsündedir. Diğer basit deyimler çeşitli konuların içerisinde ele alınacaktır.

İfadesel Deyimler (Expression Statements)

Anımsanacağı gibi operatörlerin, sabitlerin ve değişkenlerin her bir kombinasyonuna ifade (expression) deniliyordu. İşte ifadeler aynı zamanda bir deyim de belirtmektedir. İfadesel deyimler kategori olarak basit deyim durumundadır. Örneğin:

```
a = b  
x = y + 10  
z = foo(10) * 2
```

Bu satırların her biri birer ifadesel deyimdir.

Bileşik Deyimler

Bileşik deyimler kendi içerisinde başka deyimleri de içerebilen deyimlerdir. Bunların çoğu program akışını yönetmek için kullanılmaktadır. Bunlara pek çok programlama dilinde "kontrol deyimleri" de denilmektedir. Bileşik deyimlerin içindeki deyimler aynı girinti füzeyine sahip olacak biçimde "girintili olarak" yazılmak zorundadır. Her iç bileşik deyime geldiğinde girinti düzeyi artırılır. Aynı girinti düzeyine sahip olan deyimler aynı bileşik deyimin içerisinde kabul edilmektedir.

Bileşik Deyimlerin Genel Yazım Biçimi

Bileşik deyimlerin genel biçimleri birbirlerine benzemektedir. Bileşik deyimler bir anahtar sözcük ile başlatılır (if, for, while gibi) sonra bunların bir başlık kısmı olur. Ondan sonra da bu başlık kısmını ':' atomu izler. Sonra bu deyimlerin içerisindeki deyimler girintilenerek (yani indent'lenerek) yazılırlar. Bileşik deyimlerin içerisindeki deyimler aynı girinti düzeyine sahip olmalıdır. Yukarıda da belirtildiği gibi Python yorumlayıcısı aynı girinti düzeyine sahip olan deyimlerin aynı bileşik deyimin içerisinde olduğunu düşünmektedir.

Bileşik deyimlerin yazımı "suite" kavramı ile daha kolay açıklanabilir. "suite" bileşik deyimin ':' atomundan sonraki kısmına ilişkin bir kavramdır. Bir suit aynı satır üzerine yazılmış olan birden fazla basit deyimi ya da aynı girinti düzeyine sahip farklı satırlarda yazılmış birden fazla deyimi belirtmektedir. Örneğin:

```
ifade1; ifade2; ifade3
```

Bu bir suite belirtmektedir. Örneğin:

```
ifade1  
ifade2  
ifade3
```

Bu da bir suite belirtmektedir. suit kavramının ':' ile ya aynı satırda yazılan basit deyimleri ya da ':' ile aynı satırda olmayan aynı girinti düzeyine sahip deyimleri belirttiğine dikkat ediniz. İşte bileşik deyimler genellikle böyle suit'ler içermektedir. Örneğin while deyimi (ileride görülecek) ':' atomundan sonra bir suite almaktadır. Genel biçimini şöyledir:

```
while <ifade>: <suite>
```

Bu durumda aşağıdaki while deyimi geçerlidir:

```
while i < 10: ifade1; ifade2; ifade2
```

Aşağıdaki while deyimi de geçerlidir:

```
while i < 10:  
    ifade1  
    ifade2  
    ifade3
```

Ancak aşağıdaki while deyimi geçerli değildir:

```
while i < 10: ifade1  
    ifade2  
    ifade3
```

Çünkü burada hem aynı satırda hem de aşağıdaki satırda deyim yazılmıştır. Oysa suit ya aynı satırda yazılan birden fazla basit deyimi ya da aynı girinti düzeyine sahip birden fazla satırda yazılan deyimleri belirtmektedir.

suit içerisinde alt satırda yazılanların girinti düzeyleri ana bileşik deyimin girinti düzeyinden fazla olmak zorundadır. Örneğin aşağıdaki yazım hatalıdır:

```
while i < 10:  
    ifade1  
    ifade2  
    ifade3
```

Bileşik deyimlerin içerisinde en az bir deyim bulunmak zorundadır. Burada eğer bu deyimler suit deyimleri ise farklı satırlarda yazıldığı için ana while deyiminin girinti düzeyinden fazla olmalıdır. Örneğin:

```
while i < 10:  
    ifade1  
    ifade2  
ifade3
```

Bu yazım geçerlidir. Bileşik deyimlerin içerisinde en az bir deyim olmak zorundadır. Burada ifade1 ve ifade2 while deyiminin içerisindeştir ancak ifade3 while deyiminin dışındadır. Örneğin:

```
while i < 10:  
    ifade1  
        ifade2
```

Bu while deyimi sentaks bakımından geçersizdir.

Bir kez daha belirtmek gerekirse "suit" oluşturmak için "ya aynı satır üzerine tüm basit deyimler aralarına ';' atomu yerleştirilerek yazılır ya da farklı satırlarda aynı girinti düzeyine sahip deyimler" yazılır. Aşağıdaki while deyimi geçerlidir:

```
while i < 10:  
    ifade1; ifade2  
    ifade3
```

Burada aslında deyimler farklı satırlara yazılmışlardır. Ancak bir satırda birden fazla basit deyim yazılabilir.

if Deyimi

if deyimi bir koşulun doğru ya da yanlışmasına göre birtakım deyimleri çalıştıran temel bir kontrol deyimidir. Genel biçimini söyledir:

```
if <koşul ifadesi>: <suit>  
[else: <suit>]
```

if anahtar sözcüğü ile else anahtar sözcüğünün aynı girinti düzeyine sahip olması gereklidir. Ancak if deyiminin doğruysa ve yanlışsa kısmındaki suite'lerin aynı girinti düzeyine sahip olması gerekmez.

if deyimi şöyle çalışır: Önce if anahtar sözcüğünün yanındaki ifadenin değeri hesaplanır. Bu değer bool türüne dönüştürülür (Sıfır dışı değerler True olarak sıfır değeri False olarak) sonra bu ifade True ise if deyiminin doğruysa kısmındaki suite çalıştırılır, False ise yanlışsa kısmındaki suite çalıştırılır. Örneğin:

```
a = int(input("Bir değer giriniz:"))  
if a % 2 == 0: print("Çift")  
else: print("Tek")  
print('Son')
```

Burada son rint deyiminin if dışında olduğuna dikkat ediniz.

Örneğin:

```
import math
```

```

a = float(input('a:'))
b = float(input('b:'))
c = float(input('c:'))

delta = b * b - 4 * a * c
if delta < 0:
    print('Kök yok')
else:
    x1 = (-b + math.sqrt(delta)) / (2 * a)
    x2 = (-b - math.sqrt(delta)) / (2 * a)
    print('x1 = {}, x2 = {}'.format(x1, x2))

```

Burada ikinci derece bir denklemin kökleri bulunmuştur.

suit içerisindeki deyimlerin if deyiminin doğruysa ya da yanlışsa kısmında olup olmadığına girinti düzeylerine bakılarak karar verilmektedir. Örneğin:

```

if a > 0:
    ifade1
    ifade2
else:
    ifade3
    ifade4
ifade5

```

Burada ifade5 if deyiminin tamamen dışındadır.



Şimdi bazı if sentakslarının geçerliliği üzerinde duralım:

```
if a > 0: ifade1; else: ifade2
```

Bu sentaks geçersizdir. Çünkü else anahtar sözcüğü aynı satırda bulunamaz. else anahtar sözcüğü if anahtar sözcüğü ile farklı satırlarda ancak aynı girinti düzeyinde bulunmak zorundadır. Örneğin:

```
if a > 0: ifade1
else: ifade2
```

Bu if sentaksı geçerlidir. Örneğin:

```
if a > 0: ifade1
    else: ifade2
```

Bu if sentaksı geçersizdir. if ile else aynı girinti düzeyine sahip olmak zorundadır. Örneğin:

```
if a > 0: ifade1
    ifade2
    ifade3
else: ifade4
```

Bu if sentaksı geçersizdir. suite yazımında hata vardır. Örneğin:

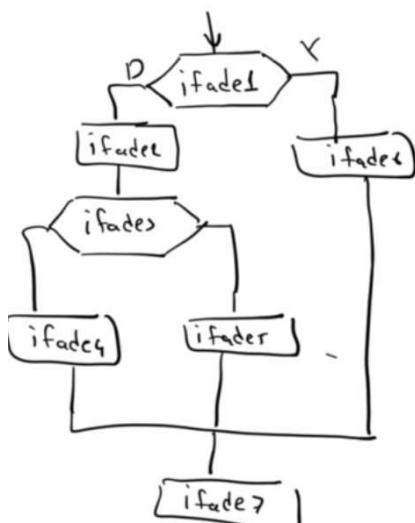
```
if a > 0:  
    ifade1  
    ifade2  
        ifade3  
else: ifade4
```

Bu if sentaksı geçersizdir. if deyiminin doğruysa ve yanlışsa kısımlarındaki deyimlerin aynı girinti düzeyine sahip olması gereklidir. Örneğin:

```
if a > 0:  
    ifade1  
    ifade2  
    ifade3  
else: ifade4
```

Bu if deyimi de geçersizdir. Suite yanlış yazılmıştır.

İç içe if deyimleri de söz konusu olabilir. If deyiminin kendisi de dışarıdan bakıldığındaysa tek bir deyimdir. Örneğin aşağıdaki akış diyagramının if cümlesini yazalım:



```
if ifade1:  
    ifade2  
    if ifade3:  
        ifade4  
    else:  
        ifade5  
else:  
    ifade6  
ifade7
```

Şimdi üç sayının en büyüğünü ekrana yazdırın bir Python programı yazalım:

```
a = int(input("a:"))  
b = int(input("b:"))  
c = int(input("c:"))  
  
if a > b:  
    if a > c:  
        print(a)  
    else:
```

```

        print(c)
else:
    if b > c:
        print(b)
    else:
        print(c)

```

if deyiminin yanlışsa kısmı hiç olmayabilir. Örneğin:

```

a = int(input('a:'))
if a > 0:
    print('pozitif')
print('son')

```

Buradaki if deyiminin yanlışsa kısmı yoktur. Yani son print deyimi if dışındadır.

Koşullardan biri doğru olduğunda diğerlerinin doğru olma olasılığı yoksa bu tür koşullara ayrık koşullar denilmektedir. Örneğin:

```

a == 3
a == 5

```

koşulları ayrıktır. Örneğin:

```

a > 0
a < 0
a == 0

```

koşullarıda ayrıktır. Fakat örneğin:

```

a > 0
a > 10

```

koşulları ayrık değildir. Ayrık koşulları ayrı if deyimleriyle el ele almak iyi teknik değildir. Örneğin:

```

if a > 0:
    ifade1
    ifade2

if a < 0:
    ifade3
    ifade4

```

Ayrık koşulları else-if biçiminde ele almak iyi tekniktir. Örneğin bir sayının pozitif negatif ya da sıfır olduğunu anlamaya çalışalım. Buradaki koşullar ayrıktır:

```

a = int(input('a:'))
if a > 0:
    print('pozitif')
else:
    if a < 0:
        print('negatif')
    else:
        print('sıfır')

```

Python'da girintileme zorunluluğu olduğu için else if merdivenleri uzadığında yazım biçimini bozulabilmektedir. Bu nedenle else-if merdivenlerinin yazımını kolaylaştırmak için elif anahtar sözcüğü düşünülmüştür. elif aslında else if anlamına gelmektedir. Örneğin:

```
if a > 0:  
    ifade1  
    ifade2  
elif a < 0:  
    ifade3  
    ifade4
```

Bu işlemin eşdeğeri şöyledir:

```
if a > 0:  
    ifade1  
    ifade2  
else:  
    if a < 0:  
        ifade3  
        ifade4
```

Örneğin:

```
a = int(input("Değer giriniz:"))  
if a > 0:  
    print("Pozitif")  
else:  
    if a < 0:  
        print("Negatif")  
    else:  
        print("Sıfır")
```

Bu işlem daha pratik olarak Python'da elif ile şöyle yapılabilir:

```
a = int(input("Değer giriniz:"))  
if a > 0:  
    print("Pozitif")  
elif a < 0:  
    print("Negatif")  
else:  
    print("Sıfır")
```

Yukarıda da belirtildiği gibi elif else-if merdivenlerinin çok olduğu durumda aşırı girinti oluşmasını engellemektedir.
Örneğin:

```
a = int(input("Bir sayı giriniz:"))  
if a == 1:  
    print("Bir")  
else:  
    if a == 2:  
        print("İki")  
    else:  
        if a == 3:  
            print("Üç")  
        else:  
            if a == 4:  
                print("Dört")  
            else:  
                print("Diğer")
```

Bunun da elif karşılığı şöyledir:

```
a = int(input('Bir sayı giriniz:'))  
if a == 1:  
    print('Bir')
```

```

elif a == 2:
    print('İki')
elif a == 3:
    print('Üç')
elif a == 4:
    print('Dört')
else:
    print('Diğer')

```

Göründüğü gibi if deyimi elif kısmı ile devam ettirilip en sonunda else kısmı kullanılabilmektedir.

Anımsanacağı gibi Python'da diğer türlerden bool türüne dönüştürme tanımlıdır. Boş bir liste, boş bir küme, boş bir demet, boş bir sözlük ve boş bir string bool türüne False olarak dolu bir liste, dolu bir küme, dolu bir demet, dolu bir sözlük ve dolu bir string de True olarak dönüştürülmemektedir. Ayrıca sıfır dışı int ve float türden değerler de True olacak biçimde, sıfır değeri False olacağın biçimde dönüştürülmemektedir. None özel değeri ise False olarak bool türüne dönüştürülür. Örneğin aşağıdaki if cümleleri geçerlidir:

```

a = int(input("Bir değer giriniz:"))
if a:
    print('Doğru')
else:
    print('Yanlış')

b = []
if b:
    print('Doğru')
else:
    print('Yanlış')

s = input('Bir yazı giriniz:')
if s:
    print('Doğru')
else:
    print('Yanlış')

```

Döngü Deyimleri (Loop Statements)

Bir grup deyimin yinelemeli olarak çalıştırılmasını sağlayan kontrol deyimlerine döngü (loop) denilmektedir. Program aslında çalışma zamanının çoğunu döngülerde geçirir. Python'da iki çeşit döngü vardır: while döngüleri ve for döngüleri. Python'daki while döngülerinde kontrol başta yapılır. Ayrıca Python'da kontrolün sonda yapıldığı while döngüleri (do-while döngüleri) yoktur.

while Döngüleri

while döngüleri bir koşul sağlandığı sürece yinelenmeye yol açar. Genel biçim şöyledir:

```
while <koşul>: <suite >
[else: <suit>]
```

while döngüsü şöyleden çalışır: Her yinelemede while anahtar sözcüğünün yanındaki ifadenin değeri hesaplanır. Bu değer bool türüne dönüştürülür. Eğer ifadenin değeri True ise suit içerisindeki deyimler sırasıyla çalıştırılır, False ise döngü sonlandırılır. Döngü sonlandığında eğer döngüden break deyimi ile çıkışmamışsa döngünün else kısmındaki suit de çalıştırılır.

Örneğin:

```
i = 0
while i < 10:
    print(i)
```

```
i += 1
```

Örneğin:

```
i = 0
while i < 10:
    print(i, end = ' ')
    i += 1
print()
```

while deyiminin else kısmı döngü break ile sonlandırılmamışsa döngü bittikten sonra çalıştırılır. Örneğin:

```
i = 0
while i < 10:
    print(i)
    i += 1
else:
    print('son')
```

Burada döngü sonlandıktan sonra else kısmı çalıştırılacak dolayısıyla ekrana 'son' yazısı da basılacaktır.

Biz bir listenin tüm elemanlarını while döngüsü ile tek tek ekrana yazdırabiliriz. Örneğin:

```
a = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
i = 0

while i < len(a):
    print(a[i], end=' ')
    i += 1
```

Tabii aslında print fonksiyonu bir listeyi de tek hamlede ekrana yazdırabilmektedir. Şimdi while döngüsü ile 1'den 100'e kadar sayıların toplamını hesaplayalım:

Örneğin:

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

Örneğin:

```
a = [4, 3, 12, 56, 33, 78, 21, 78, 9, 45]
i = 0
while i < len(a):
    if a[i] % 2 == 0:
        print(a[i], end=' ')
    i += 1
```

Diger türlerden bol türüne dönüştürme tanımlı olduğu için aşağıdaki while deyimi geçerlidir:

```
i = 10

while i:
    print(i, end=' ')
    i -= 1
else:
```

```
print('Çıkış değeri:', i)
```

Bu işlemin eşdeğeri şöyledir:

```
i = 10

while i > 0:
    print(i, end=" ")
    i -= 1
else:
    print('Çıkış değeri:', i)
```

Örneğin:

```
s = 'ankara'

while s:
    print(s)
    s = s[1:]
```

Örneğin:

```
s = {1, 10, 'ali', 20, 'veli', 'selami'}

while s:
    x = s.pop()
    print(x, end = ' ')
```

while döngülerinin else kısımları gereksiz gibi gelebilir. Yani örneğin:

```
while koşul:
    ifade1
else:
    ifade2
```

ile:

```
while koşul:
    ifade1
ifade2
```

arasında bir farklılık yoktur. Fakat aslında else kısmı break deyimi ile anlam kazanmaktadır. break deyimi ileride ele alınmaktadır.

while döngüsü ile sonsuz döngü (infinite loop) oluşturma tipik olarak şöyle yapılabilir:

```
while True:
    ....
```

Sınıf Çalışması: Bir liste oluşturunuz. Bu listenin elemanlarını while döngüsü kullanarak aynı satırda aralarında boşluk karakteri olacak biçimde ters sırada yan yana yazdırınız.

Çözüm:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i = len(a) - 1
while i >= 0:
    print(a[i], end=' ')
    i -= 1
```

Aynı işlem şöyle de yapılabilirdi:

```
i = 1
while i <= len(a):
    print(a[-i], end=' ')
    i += 1
```

Sınıf Çalışması: İçerisinde int değerlerin bulunduğu bir listenin elemanlarının ortalamasını yazdırın programı yazınız. (Anımsanacağı gibi sum fonksiyonu zaten dolaşılabilir bir nesneyi parametre olarak alıp bunun toplamına geri dönmektedir. sum fonksiyonundan elde edilen değeri listenin eleman sayısına bölgerek ortalama değeri bulabiliyoruz. Ancak bu soruda bu yöntem kullanmayınız.)

```
a = [3, 4, 3, 5, 10]
total = 0
i = 0
while i < len(a):
    total += a[i]
    i += 1
avg = total / len(a)
print('Ortalama =', avg)
```

Aslında statistics modülündeki global mean fonksiyonu da bize dolaşılabilir nesnenin ortalama değerini vermektedir. Yani yukarıdaki programı aslında şöyle de yazabilirdik:

```
import statistics

a = [3, 4, 3, 5, 10]
total = 0
i = 0
print('Toplam =', sum(a), ", Ortalama =", statistics.mean(a))
```

Bir modül içerisindeki global fonksiyonların modül ismi ve '.' operatörü ile kullanıldığına dikkat ediniz.

Sınıf Çalışması: Klavyeden bir yazı okuyunuz hiç dilimleme yapmadan yazıyı tersten yazdırınız.

Çözüm:

```
s = input('Bir yazı giriniz:')
i = len(s) - 1
while i >= 0:
    print(s[i], end='')
    i -= 1
```

Sınıf Çalışması: Bir listeyi elemanları yer değiştirerek ters yüz ediniz

Çözüm:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i = 0
lena = len(a)
while i < lena // 2:
    a[i], a[lena - 1 - i] = a[lena - 1 - i], a[i]
    i += 1
print(a)
```

Aynı işlemi negatif indeksleme yoluyla daha kolay yapabiliyoruz:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```

i = 0
while i < len(a) // 2:
    a[i], a[-i - 1] = a[-i - 1], a[i]
    i += 1
print(a)

```

Python'a 3.8'den sonra eklenen Walrus operatörü bir değer ürettiği için while döngülerinde döngünün koşul kısımlarında kullanılabilir. Örneğin:

```

while (s := input('Bir yazı giriniz:')) != 'quit':
    print(s[::-1])

```

Burada klavyeden okunan yazı (yani str nesnesinin adresi) önce s değişkenine atanmış daha sonra "quit" yazısıyla karşılaşmıştır. Bu döngüden quit yazılıncaya çıkışacaktır. Walrus operatörünün olmadığı durumda yukarıdaki kod parçasını biraz daha karmaşık biçimde oluşturmak zorunda kalırız. Örneğin:

```

while True:
    s = input('Bir yazı giriniz:')
    if s == 'quit':
        break
    print(s[::-1])

```

Okuma işlemini iki kere yaparak da benzer kodu oluşturabilirdik:

```

s = input('Bir yazı giriniz:')
while s != 'quit':
    print(s[::-1])
    s = input('Bir yazı giriniz:')

```

for Döngüleri

Python'da artırımlı for döngüleri yoktur. Python'daki for döngüleri diğer dillerdeki foreach döngüleri gibi çalışmaktadır. Python'da for döngüleri yalnızca dolaşılabilir (itrerable) nesnelerle kullanılabilir. Bu durumda her yinelenmede dolaşılabilir nesnenin sıradaki elemanı dönü değişkenine atanır sonra döngüyü oluşturan deyimler çalıştırılır. Dolaşma işlemi bittiğinde döngü de biter.

for döngülerinin genel biçimini söylemek:

```

for <değişken> in <dizilim>: <suit>
[else: <suit>]

```

Daha açık bir anlatımla Python'daki for döngüleri şöyle çalışmaktadır: in anahtar sözcüğünün sağındaki dolaşılabilir nesnenin sıradaki elemanı döngü değişkenine atanır, sonra döngü deyimleri çalıştırılır. Bu böyle devam eder. Dolaşılabilir nesnenin elemanları bitince döngü de sonlanmış olur. Tipki while döngülerinde olduğu gibi for döngülerinin de else kısmı vardır. Bu else kısmı döngüden break deyimi ile çıkışmamışsa döngü bitince çalıştırılmaktadır.

Örneğin:

```

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for i in a:
    print(i, end=' ')

```

Buradaki döngü değişkeni ayrı bir değişken statüsündedir. Bu değişkeni değiştirdiğimizde biz dizilimin elemanını değiştirmiş olmayız. Örneğin:

Örneğin:

```
s = 'ankara'  
for c in s: print(c)
```

Tabii for döngüsüyle dolaştığımız nesnenin elemanlarının aynı türden olması zorunlu değildir. Python dinamik tür sistemine sahip olduğuna göre her dolaşımda döngü değişkeninin türü de nesnenin o elemanın türü ile aynı olacaktır. Örneğin:

```
s = {'Ali', 'Veli', 10, 12.4, 'Selami'}  
  
for x in s:  
    print(x, '=>', type(x))
```

Anımsanacağı gibi sözlükler de dolaşılabilir (iterable) nesnelerdi. Bir sözlük dolaşıldığında anahtarlar elde diliyordu. O halde örneğin:

```
d = {'Ali': 123, 'Veli': 478, 'Selami': 75, 'Ayşe': 642, 'Fatma': 39}  
  
for key in d:  
    print(key, '=>', d[key])
```

Tabii sözlüklerde elimizde anahtar varsa her zaman köşeli parantezlerle değeri elde edebiliriz.

Sınıf Çalışması: Bir sözlüğü for döngüsüyle dolaşarak anahtarları elde ediniz. Yeni bir sözlük yaratarak dolaştığınız sözlüğün değerlerini anahtar, anahtarlarını değer haline getiriniz. Sonra da her iki sözlüğü yazdırınız.

Çözüm:

```
d = {'Ali': 123, 'Veli': 478, 'Selami': 75, 'Ayşe': 642, 'Fatma': 39}  
k = dict()  
  
for key in d:  
    k[d[key]] = key  
  
print(d)  
print(k)
```

Listelerden ya da demetlerden oluşan listeler ya da demetler for döngüsü ile dönülürken aynı zamanda açım (unwrap) işlemi de yapılabilir. Örneğin:

```
l = [('ali', 123), ('veli', 65), ('selami', 340), ('ayşe', 71)]  
for name, no in l:  
    print(name, no)
```

Tabii eğer yukarıdaki örnekte demetin elemanları üç tane olsayı bizim de for döngüsünde üç değişken kullanmamız gerekiirdi. Örneğin:

```
l = [('ali', 123, 1982), ('veli', 65, 1970), ('selami', 340, 1990), ('ayşe', 71, 1969)]  
for name, no, year in l:  
    print(name, no, year)
```

Tabii her zaman biz tek değişkenle içteki veri yapılarını bütün olarak da elde edebiliriz. Yukarıdaki örnekte demetleri bir bütün olarak aşağıdaki gibi de elde edebilirdik:

```
l = [('ali', 123, 1982), ('veli', 65, 1970), ('selami', 340, 1990), ('ayşe', 71, 1969)]  
for t in l:  
    print(t)
```

Şimdi de Bir demet listesini dolaşarak elemanların yerleri değiştirilmiş yeni bir demet listesi elde edelim:

```

a = [('ali', 10), ('veli', 20), ('selami', 30), ('ayşe', 40), ('fatma', 50)]
b = []

for name, no in a:
    b.append((no, name))

print(a)
print(b)

```

Bir sözlüğü de benzer biçimde açıp yaparak for döngüsü ile dolaşabiliriz. Örneğin:

```

d = {'ali': 123, 'veli': 765, 'selami': 745, 'ayşe': 271, 'fatma': 754}

for key, value in d.items():
    print(key, value)

```

Pekiyi biz Python'daki for döngülerini diğer dillerdeki for döngüleri gibi nasıl kullanabiliriz? Yani for döngüleriyle artırımlı ya da azaltımlı değerleri nasıl elde ederiz? Örneğin 0'dan 100'e kadar dönen for döngüsü oluşturmak istesek? İşte Python'da bunun için range fonksiyonundan faydalanailmaktadır. range fonksiyonun bize belli bir aralıkta tamsayılardan oluşan dolaşılabilir bir nesne verdiği anımsayınız. Örneğin:

```

for i in range(1, 10):
    print(i, end=' ')
print()

```

Örneğin:

```

for i in range(10):
    print(i, end=' ')
print()

for i in range(10, 20):
    print(i, end=' ')
print()

for i in range(1, 10, 2):
    print(i, end=' ')
print()

for i in range(10, 0, -2):
    print(i, end=' ')
print()

```

Çıktı da şöyle olacaktır:

```

0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
1 3 5 7 9
10 8 6 4 2

```

Şimdi bir listedeki tek ve çift sayıları bularak onları başka bir listeye yerleştirelim:

```

a = [23, 56, 44, 12, 13, 97, 45, 39, 80, 54]
odd = []
even = []

for i in a:
    if i % 2 == 0:
        even.append(i)

```

```

else:
    odd.append(i)

print(odd)
print(even)

```

ord isimli built-in global fonksiyon bir karakterin sayısal değerini bize verir. chr ise bunun tam tersini yapmaktadır. Yani bir değeri bizden alarak onun karakter karşılığını string olarak bize verir. Örneğin:

```

for i in range(ord('A'), ord('Z') + 1):
    print(chr(i), end=' ')

```

for döngüleriyle bir dizimin elemanlarını nasıl değiştirebiliriz? İlk akla gelen yöntem dizimi dolaşarak elemanların değerlerini değiştirmektir. Örneğin bir listenin elemanlarının değerlerini onların karesiyle değiştirelim:

```

a = [2, 5, 1, 9, 10]
for i in range(len(a)):
    a[i] *= a[i]
print(a)

```

Şimdi de bir listedeki en büyük elemanı bulmaya çalışalım. (Aslında standart kütüphanede dolaşılabilir bir nesnenin en büyük ve en küçük elemanlarını bulmak için kullanılan max ve min fonksiyonları bulunmaktadır.) Bir yöntem şöyle olabilir:

```

a = [3, 5, 7, 9, 21, 10, 62, 34, 9, 5, 8]

maxval = a[0]
for i in range(1, len(a)):
    if a[i] > maxval:
        maxval = a[i]

print(maxval)

```

Diger bir yöntem şöyle olabilir

```

a = [3, 5, 7, 9, 21, 10, 62, 34, 9, 5, 8]

maxval = a[0]
for i in a[1:]:
    if i > maxval:
        maxval = i
print(maxval)

```

for döngülerinin de bir else kısmı olabilmektedir. Eğer döngüden break ile çıkışlmamışsa for deyimi bittiğinde bir kez bu else kismı çalıştırılır. Örneğin:

```

for i in range(1, 11):
    print(i, end = " ")
else:
    print("for ends...")

```

Örneğin:

```

for i in range(0, 10):
    print(i, end = ' ')
else:
    print()
print('ends...')

```

Sınıf Çalışması: Klavyeden int bir değer okuyunuz. 1'den bu değere kadar olan tek sayıların toplamını ekrana yazdırınız.

Çözüm:

```
n = int(input('Bir sayı giriniz:'))  
  
total = 0  
for i in range(1, n + 1, 2):  
    total += i  
print(total)
```

Sınıf Çalışması: Klavyeden bir sayı isteyiniz. Sayının asal çarpanlarını aralarına boşluk bırakarak yazdırınız. Örneğin:

Bir sayı giriniz: 20

2 2 5

Çözüm:

```
n = int(input('Bir sayı giriniz:'))  
div = 2  
while n != 1:  
    if n % div == 0:  
        n /= div  
        print(div, end=' ')  
    else:  
        div += 1
```

Sınıf Çalışması: Klavyeden (stdin dosyasından) bir sayı okuyunuz. Sayının basamaklarını aralarına boşluk karakteri bırakarak yüksek anlamlı digitten düşük anlamlı digite doğru yazdırınız. Örneğin:

Bir sayı giriniz: 12345

1 2 3 4 5

Çözüm:

```
n = int(input('Bir sayı giriniz:'))  
s = str(n)  
for c in s:  
    print(c, end=' ')
```

Düzen bir çözüm şöyle olabilir:

```
n = int(input('Bir sayı giriniz:'))  
  
digits = []  
while n > 0:  
    r = n % 10  
    digits.append(r)  
    n /= 10  
  
for digit in reversed(digits):  
    print(digit, end=' ')
```

Sınıf Çalışması: Bir listede belli bir değerden kaç tane olduğunu yazdırın programı yazınız. (Anımsanacağı gibi zaten list sınıfının count metodu bu işi yapmaktadır).

Çözüm:

```

a = [1, 2, 2, 3, 4, 5, 2, 1, 2, 3, 1, 2]
val = int(input('Aranacak değeri giriniz:'))

count = 0
for i in a:
    if i == val:
        count += 1
print(count)

```

Sınıf Çalışması: Klavyeden tek bir string olarak int değerleri boşluklarla ayrılmış biçimde giriniz. Bu int değerleri bir liste içerisine int olarak yerleştiriniz.

Çözüm:

```

s = input('Liste değerlerini aralarına boşluk karakterleri koyarak giriniz:')
a = []
for s in s.split():
    a.append(int(s))
print(a)

```

break Deyimi

break basit bir deyimdir. Döngüden erken çıkmak için kullanılır. Genel biçim şöyledir:

```
break
```

break deyimi yalnızca döngüler içerisinde kullanılabilir. Programın akışı break deyimini gördüğünde döngü sonlandırılır, akış döngüden sonraki ilk deyim ile devam eder. Örneğin:

```

while True:
    a = int(input("Değer giriniz:"))
    if a == 0:
        break
    print(a * a)
print("program bitti")

```

Programın akışı break anahtar sözcüğünü gördüğünde döngüden sonraki ilk deyim ile devam eder. Dögüyü sonsuz döngü biçiminde organize ederek içерiden gerektiğinde break ile çıkmak bazen tasarımlı kolaylaştırımaktadır.

Python'da while ve for döngülerinin else kısımları eğer döngüden break deyimi ile çıkışmışsa çalıştırılmaz. Örneğin:

```

for i in range(5):
    val = int(input('Bir değer giriniz:'))
    if val == 0:
        break
    print(val * val)
else:
    print('Döngü sonlandı')
print('Program sonlandı')

```

Pekiyi neden break ile çıktıduğında döngülerin else kısımları çalıştırılmamaktadır? Bazen döngünün kendi kendine sonlanması ile break ile sonlandırılması arasında programcı açısından farklılık oluşabilemektedir. İşte bu farklılıktan programcı lehine faydalılmak istenmiştir. Örneğin bir liste içerisinde bir değeri sıralı olarak aradığımızı düşünelim (gerci list sınıfının böyle bir metodu vardır). Eğer liste bittiği halde biz aradığımız değeri bulamamışsak başarısız olduğumuz sonucunu çıkarabiliriz. Tabii aradığımız değeri bulursak break ile döngüden çıkarız:

```

a = [23, 45, 34, 67, 89, 34, 56, 11, 23, 45]
val = int(input('Aranacak sayıyı giriniz:'))
for i in a:

```

```

if i == val:
    print('bulundu')
    break
else:
    print('bulunamadı')

```

Burada eğer döngüden normal biçimde çıkışmışsa bu durum bizim aradığımız değeri bulamadığımız anlamına gelmektedir. Tabii döngünün else kısmını kullanmadan da yukarıdaki kodu şöyle düzenleyebilirdik:

```

a = [23, 45, 34, 67, 89, 34, 56, 11, 23, 45]
val = int(input('Aranacak sayıyı giriniz:'))
for i in a:
    if i == val:
        print('bulundu')
        break

if i != val:
    print('bulunamadı')

```

Örneğin:

```

for i in range(3):
    s = input('Enter password:')
    if s == 'maviay':
        print('Ok')
        break
    else:
        print('invalid password')
else:
    print('Your account blocked!')

```

Sınıf Çalışması: Klavyeden bir int sayı isteyiniz. Sayının asal olup olmadığını yazdırınız.

Çözüm:

```

n = int(input('Bir sayı giriniz:'))

for i in range(2, n):
    if n % i == 0:
        print('Asal değil')
        break
    else:
        if n < 2:
            print('Geçersiz sayı!')
        else:
            print('Asal')

```

Tabii asal sayı testlerinin daha etkin algoritmik yöntemleri de vardır.

continue Deyimi

continue deyimi o anki yinelemeyi sonlandırarak yeni bir yinelemeye geçiş sağlar. Bu deyim de yalnızca döngülerin içerisinde kullanılabilir. Genel biçimini şöyledir:

```
continue
```

Programın akışı continue deyimini gördüğünde o yineleme bitirilir yeni yinelemeye geçilir. Örneğin:

```
a = [13, 20, 31, 40, 47, 60, 70, 80, 91, 100]
```

```

for i in a:
    if i % 2 == 0:
        continue
    print(i, end=' ')
print()

```

Örneğin:

```

while True:
    cmd = input('CSD>').strip()
    if cmd == '':
        continue
    if cmd == 'dir':
        print('dir command')
    elif cmd == 'del':
        print('del command')
    elif cmd == 'copy':
        print('copy command')
    elif cmd == 'exit':
        break
    else:
        print('bad command!')

```

Bu döngü yalnızca listedeki tek sayıları yazdıracaktır. Tabii biz listedeki tek sayıları if deyiminin koşul kısmını değiştirerek de yazdırabilirdik. continue deyimi döngüler içerisindeki geniş if bloklarını elimine etmek için tercih edilmektedir. Böylece kod çok daha sade gözükmemektedir. Bir döngüyü tamamen kapsayan bir if deyimi algılamayı zorlaştırılmamaktadır. Örneğin:

```

for i in a:
    if i % 2 == 0:
        ifade1
        ifade2
        ifade2

```

Burada liste içerisindeki yalnızca çift değerler üzerinde işlem yapılmıştır. Bunu continue deyimi ile şöyle de ifade edebilirdik:

```

for i in a:
    if i % 2 != 0:
        continue
    ifade1
    ifade2
    ifade2

```

match Deyimi

Pek çok programlama dilinde bir ifadenin çeşitli sayısal değerine göre değişik işlemleri yapabilmek için switch deyimleri vardır. Ancak switch deyimlerine benzer işlemi yapan match deyimi Python diline Ekim 2021 yılında 3.10 versiyonu ile eklenmiştir. Eskiden Python'da switch deyimi olmadığı için bu gibi işlemler ağırlıklı olarak if deyimleriyle yapılmıyordu. Örneğin:

```

val = int(input('Bir değer giriniz:'))

if val == 1:
    print('bir')
elif val == 2:
    print('iki')
elif val == 3:
    print('üç')

```

```

elif val == 4:
    print('üç')
elif val == 5:
    print('beş')
else:
    print('hiçbiri')

```

Python'da sözlükler de switch benzeri bir yapı oluşturmak amacıyla programcılar tarafından kullanılıyordu. Python'da sözlükler de switch benzeri bir kullanım olanağı sunmaktadır. Her ne kadar henüz fonksiyonlar konusunu ele almamış olsak da aşağıdaki örnek sözlüklerle nasıl switch benzeri bir yapı oluşturulacağı hakkında size bir fikir verebilecektir:

```

def foo():
    print("bir")

def bar():
    print("iki")

def tar():
    print("üç")

def car():
    print("dört")

def zar():
    print("beş")

d = {1: foo, 2: bar, 3: tar, 4: car, 5: tar}

n = int(input("Bir değer giriniz:"))

if n >= 1 and n <= 5:
    d[n]()
else:
    print("hiçbiri")

```

Göründüğü gibi Python'da sözlüklerin değerleri fonksiyonlar da olabilmektedir.

Python'da switch benzeri bir deyimin olmaması uzun süredir Python programcıları tarafından yadırGANIYORDU. Nihayet tasarım ekibi 3.10 ile match isimli switch benzeri bir deyimi dile ekledi. Bu deyim PEP-634, PEP-635 ve PEP 636 ile aytınlı biçimde açıklanmıştır.

match deyimi C/C++, Java ve C# gibi dillerdeki switch deyimlerine göre daha genel ve daha ayrıntılıdır. match deyiminin yalın genel biçimi şöyledir:

```

match <ifade>:
    case <kalıp>:
        # ...
    case <kalıp>:
        # ...
    case <kalıp>
        #...
    #...

```

match deyiminin de case bölümlerinden oluştuğunu görüyorsunuz.

match deyimi şöyle çalışmaktadır: Önce match anahtar sözcüğünün yanındaki maifade (biz buna "match ifadesi" diyeceğiz, Python resmi dokümanları buna "subject expression" demektedir) hesaplanır. Sonra bu ifade case bölümlerindeki kalıplarla sırasıyla karşılaştırılır ve bu karşılaştırmanın başarılı olduğu ilk kalıba ilişkin case bölümündeki deyimler çalıştırılır. Eğer match ifadesi case bölümlerindeki kalıpların hiçbiri ile uyuşmazsa mahtc deyiminin çalışması sonlanır.

match ifadesi ile case kalıplarının uyuşmasına ilişkin bazı ayrıntılar vardır. Burada bu ayrıntılar üzerinde duracağz. Önce basit bir örnek vermek istiyoruz:

```
a = int(input('bir değer giriniz:'))  
  
match a:  
    case 1:  
        print('bir')  
    case 2:  
        print('iki')  
    case 3:  
        print('üç')  
    case 4:  
        print('dört')  
    case 5:  
        print('beş')  
    case _:  
        print('hiçbiri')
```

Burada case bölümündeki kalıplar birer sabit olarak kullanılmıştır. Bu kalıplara "sabit kalıpları (literal patterns)" denilmektedir. Python'da case bölümündeki sabit kalıpları diğer bazı dillerdeki gibi sabit ifadesi biçiminde olamaz. Tek bir sabit biçiminde olmak zorundadır. Ancak string'ler de sabit olarak kalıplarda kullanılabilirler. Son case bölümündeki '_' kalıbı her zaman uygunluk sağlayan özel bir kalıptır. Bu kalıbı "joker kalıbı (wildcard pattern)" denilmektedir. Şimdi benzer bir örnek daha verelim:

```
s = input('Bir renk giriniz: ')  
  
match s:  
    case 'kırmızı':  
        print('red')  
    case 'mavi':  
        print('blue')  
    case 'yeşil':  
        print('üç')  
    case 4:  
        print('green')  
    case _:  
        print('another color')
```

match deyiminde C/C++, Java gibi dillerindeki switch deyimindeki "aşağıya düşme (fall through)" kuralı yoktur. match deyiminde her zaman tek bir case bölümü çalıştırılmaktadır.

case bölgelerinde birden fazla bulundurulabilir. Bu durumda kalıpların '|' atomuyla birbirinden ayrılmaktadır. Bu biçimdeki kalıplara "veya kalıpları" denilmektedir. Veya kalıpları farklı seçenekler için aynı işlemlerin yapılması amacıyla kullanılmaktadır. Örneğin:

```
while True:  
    cmd = input('CSD>').strip()  
    match cmd:  
        case 'delete' | 'remove' | 'erase':  
            print('silme işlemi')  
        case 'copy':  
            print('kopyalama işlemi')  
        case 'rename':  
            print('isim değiştirme işlemi')  
        case 'exit' | 'quit':  
            break  
        case _:  
            print('geçeriz komut!')
```

Burada bir komut satırı eşliğinde kullanıcı komutlar girmekte ve girilen komutlar işlenmektedir. Silme işlemi için "delete", "erase" ve "remove" komutlarının hepsi aynı işlemi yapmaktadır. Benzer biçimde "exit" ve "quit" komutları da işlevsel olarak eşdeğerdir.

Veya kalıbında biz uyuşan kalının bir değişkene yerleştirilmesini de isteyebiliriz. Bunun için kalının sonuna as cümleceğinin yerleştirilmesi gerekmektedir. Örneğin:

```
while True:
    cmd = input('CSD>').strip()
    match cmd:
        case 'delete' | 'remove' | 'erase': as x:
            print(f'silme işlemi "{x}" komutu ile正在执行')
        case 'copy':
            print('копирование')
        case 'rename':
            print('имя изменение')
        case 'exit' | 'quit':
            break
        case _:
            print('无效命令!')
```

Burada "delete", "remove" ya da "erase" komutlarının hangisi girilmişse x değişkenine o komutun yazısı atanacaktır. (Tabii Python'da tüm değişkenlerin aslında adres tuttuğunu, tüm atamaların da aslında adres ataması olduğunu anımsayınız.) Bu örnekte cmd zaten girilen komutu belirttiğine göre bu komutun içeriğinin aynı zamanda x değişkenine atanması size anlamsız gelebilir. Ancak izleyen paragraflarda alt kalıplar için as cümleceği gerekmektedir. as cümleciğinde bir noktaya daha dikkatinizi çekmek istiyoruz. as cümleceği ile belirtilen değişkene atama işlemi ancak ilgili case bölümündeki kalıp uyuşumu başarılı ise yapılmaktadır. Örneğin yukarıdaki kodda "delete", "remove" ya da "erase" komutları hiç girilmediyse x değişkeni de aslında hiç yaratılmayacaktır. Ayrıca buradaki "x" değişkeninin -eğer yaratılmışsa- match deyiminden sonra da kullanılabileceğini belirtmek istiyoruz.

Biz case bölgümlerinde liste, demet gibi bir dizimin (sequence) uyuşumuyla ilgili kalıp da oluşturulabiliriz. Bu tür kalıplara "dizim kalıpları (sequence patterns)" denilmektedir. Bunun için case bölümünün aşağıdaki biçimlerden biri gibi oluşturulması gereklidir:

```
case [elem1, elem2, ..., elemn]:
case (elem1, elem2, ..., elemn):
case elem1, elem2, ... elemn:
```

Burada elemanların köşeli parantezler içerisinde, normal parantezler içerisinde ya da parantezsiz bir biçimde belirtilmeler arasında fark yoktur. Örneğin:

```
a = ['ali', 'veli']

match a:
    case 'ali', 'veli':
        print('kalıp bulundu')
    #...
```

Burada a listesinin elemanları sırasıyla "ali" ve "veli" ise kalıp uyuşumu sağlanacaktır. Burada case bölümündeki elemanlar köşeli parantezler içerisinde ya da normal parantezler içerisinde de yazılabildi. Bu tür kalıplarda match ifadesinin liste, demet, range gibi bir dizim türünden (sequence type) olması zorunludur.

Dizim kalıplarında biz dizimin uyuşan parçalarının değişkenlere atanmasını sağlayabiliyoruz. Bunun için kalıpta sabitler yerine değişken isimleri bulundurulur. Örneğin:

```
while True:
    cmd = input('CSD>').split()
```

```

match cmd:
    case ('delete', path) | ('remove', path) | ('erase', path):
        print(f'{path} dosyası siliniyor')
    case 'copy', source_path, dest_path:
        print(f'{source_path} dosyası {dest_path} olarak kopyalanyor')
    case 'rename', oldname, newname:
        print(f'{oldname} dosyanının ismi {newname} olarak degistiriliyor')
    case 'exit' | 'quit':
        break
    case _:
        print('geceriz komut!')

```

Dizilim kalıplarında yıldızlı isim de kullanılabilir. Bu durumda yıldızlı isim sıfır, bir ya da birden fazla elemanla uyuşum sağlar. Yıldızlı isim bu elemanları bir liste biçiminde kabul eder. Örneğin:

```

while True:
    cmd = input('CSD>').split()
    match cmd:
        case 'delete', path, *others:
            print(f'delete {path} {others}')
        case _:
            print('geceriz komut!')

```

Burada others değişkeni uyuşma durumunda her zaman bir liste belirtir Örneğin:

```

CSD>delete a.txt b.txt c.txt d.txt
delete a.txt ['b.txt', 'c.txt', 'd.txt']
CSD>delete a.txt
delete a.txt []
CSD>delete
geceriz komut!

```

pass Deyimi

C, C++, Java ve C# gibi dillerde yalnızca ';' boş deyim anlamına gelmektedir. Python'da girintili bir yazım biçimi olduğu için boş deyimler pass isimli anahtar sözcükle temsil edilmişlerdir. Örneğin biz bir fonksiyonun, for ya da while döngülerinin içerisinde hiçbir şey yapmak istemeyebiliriz. Bu durumda oraya bir deyim olarak pass yerleştirilmelidir. Örneğin:

```

for k in range(10):
    for i in range(100000000):
        pass
    print(k)

```

yalnızca döngülerde değil suit gereken her yerde (örneğin fonksiyon ve sınıf tanımlamalarında) pass deyimini kullanabiliriz.

Aslında pass deyimi boş bir suite anlamı dışında herhangi bir yerde de boş deyim olarak kullanılabilmektedir. Tabii böylesi kullanımların çoğu zaman anlamı yoktur. Örneğin:

```

i = 0
while i < 10:
    print(i)
    i += 1
    pass

```

Burada pass deyiminin bulundurulması error oluşturmaz ancak anlamsızdır.

Koşul Operatörü

Python'da da tıpkı C, C++, Java ve C#'ta olduğu gibi bir koşul operatörü vardır. Koşul operatörü if deyimi gibi çalışır fakat bir değer üretir. Genel biçim şöyledir:

```
<ifade1> if <bool türden ifade> else <ifade2>
```

Koşul operatörü bir operatör olduğu için bir değer üretir. Önce if ile else anahtar sözcüklerinin arasındaki ifadenin değeri hesaplanır. Bu ifade Bool türüne dönüştürülür. Eğer bu ifade True ise if anahtar sözcüğünün solundaki ifade, False ise else anahtar sözcüğünün sağındaki ifade çalıştırılır. Koşul operatöründen elde edilen değer bu ifadelerden hangi çalıştırılmış aonun sonucudur. Bu operatör yerine pek çok dilde aynı işlemi yapan ? : operatörü vardır. Örneğin:

```
n = int(input('Bir değer giriniz:'))  
  
result = 100 if n % 2 == 0 else 200      # result = n % 2 == 0 ? 100 : 200  
print(result)
```

Koşul operatörü bazı durumlarda if deyimi yerine kullanıldığından daha sade görüntü verebilmektedir. Örneğin:

```
a = int(input('a:'))  
b = int(input('b:'))  
  
max = a if a > b else b  
print(max)
```

Koşul operatörü düşük öncelikli bir operatördür. Dolayısıyla bunun sonucu üzerinde işlem yapabilmek için paranteze alınması gereklidir. Örneğin:

```
a = int(input('a:'))  
b = int(input('b:'))  
  
c = (a if a > b else b) + 100  
print(c)
```

İç içe koşul operatörü de kullanılabilir.

```
a = int(input('a:'))  
b = int(input('b:'))  
c = int(input('c:'))  
  
max = (a if a > c else c) if a > b else (b if b > c else c)  
print(max)
```

Koşul operatörü sağdan sola önceliğe sahiptir. Aslında yukarıdaki örneklerde prantezler hiç kullanılmamıştır:

```
a = int(input('a:'))  
b = int(input('b:'))  
c = int(input('c:'))  
  
max = a if a > c else c if a > b else b if b > c else c  
print(max)
```

Fonksiyon çağrılarında argüman olarak koşul operatörünün bulundurulmasına sık rastlanmaktadır. Örneğin:

```
val = int(input('Bir sayı giriniz:'))  
print('çift' if val % 2 == 0 else 'tek')
```

Bu programın eşdeğeri if deyimi kullanılarak şöyle yazılabılır:

```

val = int(input('Bir sayı giriniz:'))
if val % 2 == 0:
    print('çift')
else:
    print('tek')

```

Göründüğü gibi koşul operatörü daha kısa bir yazım sağlayabilmektedir. Örneğin:

```

a = [12, 3, 4, 5, 9, 8, 21]

for i in a:
    print('{} ---> {}'.format(i, 'çift' if i % 2 == 0 else 'tek'))

```

Aynı şeyi string interpolasyonuyla da şöyle yapabilirdik:

```

a = [12, 3, 4, 5, 9, 8, 21]

for i in a:
    print(f'{i} ---> {"çift" if i % 2 == 0 else "tek"}')

```

Her ne kadar return deyimini henüz görmemişsek de koşul operatörü return deyimlerinde de güzel bir biçimde kullanılabilmektedir. Örneğin:

```
return 100 if a % 2 == 0 else 200
```

Burada a çift ise fonksiyondan 100 ile, tek ise 200 ile geri dönülmüştür. Kodun if deyimi eşdeğeri şöyledir:

```

if a % 2 == 0:
    return 100
else:
    return 200

```

Fonksiyonların Tanımlanması

Şimdiye kadar biz yalnızca var olan (yani başkaları tarafından yazılmış olan) fonksiyonları çağrırdık. Şimdi biz de fonksiyon yazacağız. Bir fonksiyonun yazımına Python referans dokümanlarında "fonksiyonun tanımlanması (function definition)" denilmektedir. Fonksiyon tanımlama işleminin genel biçimi şöyledir:

```
def <fonksiyon ismi>([parametre listesi]): <suit>
```

Örneğin:

```
def foo():
    print('I am foo')
```

Burada fonksiyonun ismi foo olarak verilmiştir. Fonksiyonun herhangi bir parametresi yoktur. Örneğin:

```
def bar(a, b, c):
    print(a, b, c)
```

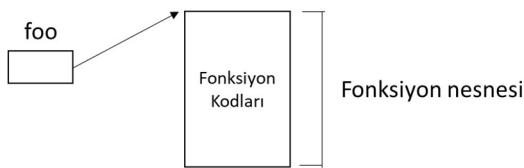
Burada da bar fonksiyonun ismini, a, b ve c de parametrelerini belirtmektedir. Fonksiyon tanımlamasında bir "suit" gerektiğine dikkat ediniz. Yani birden fazla basit deyim hemen aynı satırda yazılabilir.

```
def foo(): print('bir'); print('iki'); print('üç')
```

Python'da fonksiyon tanımlamaları da birer deyim statüsündedir. Python yorumlayıcısı bir fonksiyonun tanımlandığını gördüğünde önce bir fonksiyon nesnesi yaratır. Yarattığı fonksiyon nesnesinin içerisinde fonksiyonun kodlarını yerleştirir. Fonksiyon nesnesinin adresini de fonksiyon ismine ilişkin değişkene atar. Yani Python'da fonksiyon isimleri de aslında birer değişkendir. Örneğin:

```
def foo():
    pass
```

Burada önce bir fonksiyon nesnesi yaratılacak, fonksiyon nesneinin içerisinde fonksiyonun kodları yerleştirilecek, fonkiyon nesnesinin adresi de foo değişkenine atanacaktır. Bu durumu şekilsel olarak aşağıdaki gibi gösterebiliriz:



Bir fonksiyon çağrıldığında yorumlayıcı fonksiyon isminin gösterdiği yerdeki fonksiyon nesnesinin içerisinde bulunan kodları çalıştırmaktadır. Örneğin:

```
foo()
```

Burada foo değişkeninin içerisindeki adreste bulunan fonksiyon nesnesinin içerisindeki kodlar çalıştırılacaktır.

Fonksiyonlar tür olarak function isimli bir sınıfla temsil edilmektedir. Örneğin:

```
def foo():
    print('foo first');

print(type(foo))      # <class 'function'>
```

O halde Python'da aynı fonksiyonun ikinci kez tanımlanması aynı değişkene ikinci kez değer atama anlamına gelir. Örneğin:

```
def foo():
    print('foo')

def foo():
    print('diğer foo')
```

Aslında bu işlemin aşağıdaki gibi bir işlemden hiçbir farkı yoktur:

```
x = 10
x = 20
```

Örneğin:

```
def foo():
    print('foo first');

foo()

def foo():
    print('foo second')

foo()
```

Mademki fonksiyon isimleri aslında fonksiyon kodlarının bulunduğu function türünden nesnelerin adreslerini tutmaktadır o halde biz bu adresleri başka değişkenlere atayıp o değişkenlerle de fonksiyonları çağırabiliriz. Python'da tüm atama işlemlerinin adres ataması anlamına geldiğini anımsayınız. Fonksiyonların diğer değişkenler gibi atama işlemine sokulması durumuna programlama dillerinde esprili bir biçimde "fonksiyonların birinci sınıf vatandaş (first class citizen)" olması denilmektedir. Python'da fonkiyonlar birinci sınıf vatandaştır. Örneğin:

```

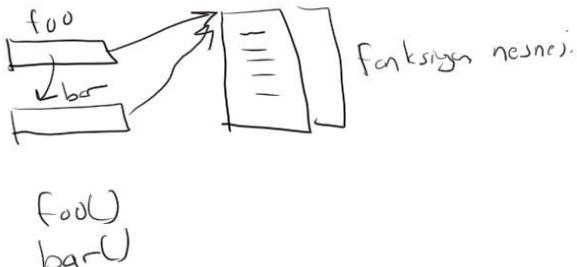
def foo():
    print('foo');

bar = foo

foo()
bar()

```

Buradaki atama işlemini şekilsel olarak şöyle açıklayabiliriz:



Göründüğü gibi foo ve bar değişkenlerinin içerisinde aynı fonksiyon nesnesinin adresleri vardır. Dolayısıyla çağrıının foo() biçiminde yapılması ile bar() biçiminde yapılması arasında bir farklılık yoktur. Daha önceden de belirttiğimiz gibi aslında fonksiyon çağrımda kullandığımız parantezler bir operatördür. Bu parantezler "bu adresteki fonksiyon nesnesinin içerisindeki kodları çalıştır" anlamına gelmektedir.

Fonksiyonlar birinci sınıf vatandaş olduğuna göre fonksiyonlar da listelerin, sözlüklerin elemanları yapılabilirler. Örneğin:

```

def foo():
    print('foo')

def bar():
    print('bar')

def tar():
    print('tar')

a = [foo, bar, tar]

for f in a:
    f()

```

Ya da örneğin:

```

def foo():
    print('foo')

def bar():
    print('bar')

def tar():
    print('tar')

d = {1: foo, 2: bar, 3: tar}

d[1]()

```

```
d[2]()
d[3]()
```

Fonksiyonların Parametre Değişkenleri

Fonksiyonlar parametrelere sahip olabilirler. Python dinamik tür sistemine sahip olduğu için parametreler bildirilirken tür belirtilmmez. Parametrelerin yalnızca isimleri parametre parantezinin içeresine aralarına ',' atomu yerleştirilerek yazılmaktadır. Örneğin:

```
def foo(a, b, c):
    print(a, b, c)
```

Tıpkı normal değişkenlerde olduğu gibi parametre değişkenleri için de bir tür bildiriminin yapılmadığına dikkat ediniz. Parametrelerin türleri fonksiyonun hangi türden argümanlarla çağrıldığına bağlı olarak programın çalışma zamanı sırasında değişimlebilir. Örneğin:

```
def foo(x):
    print(type(x))

foo(10)
foo(20.2)
foo('Ali')
```

Kodun çıktısı şöyledir:

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Normal olarak parametreli bir fonksiyon parametre sayısı kadar argümanla çağrılır. Argümanlar birer ifade olabilir. Fonksiyon çağrılığında argümanların değerleri hesaplanır. Bunlar (yani bu nesnelerin adresleri) karşılıklı olarak fonksiyon parametrelerine atanır ve akış fonksiyona aktarılır. Fonksiyonun parametre değişkenleri yalnızca fonksiyon içerisinde (yani suite içerisinde) kullanılabilirmektedir. Örneğin:

```
def foo(x,y):
    print(x+y)

foo(3+2, 5-3)
```

Örneğin:

```
def print_add(x, y):
    print(x + y)

print_add(10, 20)
print_add(10, 20.5)
print_add(12.4, 23.5)
```

Şöyle bir çıktı elde edilmiştir:

30.5

35.9

Örneğin:

```
def disp_banner(s):
    print('-' * len(s))
    print(s)
    print('-' * len(s))

disp_banner('Ankara')
```

Şöyledir bir çıktı elde edilmiştir:

```
-----  
Ankara  
-----
```

Pekiyi biz buradaki disp_banner fonksiyonuna yanlış türden değer geçersek ne olur? Örneğin:

```
disp_banner(10)
```

İşte Python dinamik bir tür sistemine sahip olduğu için bu tür durumlarda hata derleme aşamasında tespit edilemeyecektir. Program çalışırken ya yanlış çalışacak ya da exception oluşacaktır. Bu durum dinamik tür sistemine sahip programlama dillerinin olumsuz bir özelliğidir. Örneğin:

```
disp_banner(10)
```

Çağrısı yapılmış olsun. Şöyledir bir exception oluşacaktır:

```
File "D:/Dropbox/Kurslar/Python/Src/Sample/sample.py", line 6, in <module>
    disp_banner(10)
File "D:/Dropbox/Kurslar/Python/Src/Sample/sample.py", line 2, in disp_banner
    print('-' * len(s))
TypeError: object of type 'int' has no len()
```

Bu exception int bir nesnenin len fonksiyonuna sokulmasından dolayı oluşmuştur. Burada exception'ı oluşturan len fonksiyonudur. Exception'lar programcı tarafından da oluşturulabilmektedir. Bu konu ileride ele alınacaktır.

Şimdi de örnek olarak ikinci derece bri denklemin köklerini ekrana yazdırın bir fonksiyon tanımlayalım:

```
import math

def disp_roots(a, b, c):
    delta = b ** 2 - 4 * a * c
    if delta >= 0:
        x1 = (-b + math.sqrt(delta)) / (2 * a)
        x2 = (-b - math.sqrt(delta)) / (2 * a)
        print(f'x1 = {x1}, x2 = {x2}')
    else:
        print('kök yok')

disp_roots(1, 0, -4)
```

Kodun çalıştırılması sonucunda aşağıdaki gibi bir çıktı elde edilecektir:

x1 = 2.0, x2 = -2.0

Fonksiyonların Geri Dönüş Değerleri

Bir fonksiyon onu çağırılan fonksiyona bir değer iletebilmektedir. İletilen bu değere fonksiyonun "geri dönüş değeri (return value)" denilmektedir. Fonksiyonun geri dönüş değeri fonksiyon içerisinde return deyimi ile oluşturulmaktadır. return basit bir deyimdir. return deyiminin genel biçimini söylemek:

```
return [ifade]
```

Programın akışı return deyimini gördüğünde fonksiyon sonlanır ve geri dönüş değeri oluşturularak akış onu çağırılan fonksiyona geri döner. Fonksiyonun geri dönüş değerinin olması çağrıyan onu kullanmasını zorunlu hale getirmez. Yani geri dönüş değeri fonksiyonu çağrıran tarafından kullanılabilir. Örneğin:

```
def square(a):
    return a * a

x = square(5)
print(x)
```

Python'da fonksiyonların geri dönüş değerlerine ilişkin türlerin de fonksiyon tanımlamasında belirtilmemiş olmasına dikkat ediniz. Programın akışı return deyimini gördüğünde return deyiminin yanındaki ifadenin türü zaten fonksiyonun geri dönüş değerinin türü olmaktadır. Peki programın akışı hiç return eyimini görmeden fonksiyon sonlanırsa ne olur? İşte Python'da programın akışı return deyimini görmeden fonksiyon sonlanırsa fonksiyondan geri dönüş değeri olarak None değeri elde edilmektedir. return deyiminde return anahtar sözcüğünün yanına bir ifade de yazılmayabilir. Bu durumda da geri dönüş değeri olarak None değeri elde edilmektedir. Yani Python'da fonksiyonun her zaman bir geri dönüş değeri vardır. Eğer biz onu belirlememişsek o değer None biçimindedir. Örneğin:

```
def foo():
    print('foo')

result = foo()
print(result)
```

Kod çalıştırıldığında ekranda None yazısını göreceğiz.

Fonksiyonların tek bir geri dönüş değeri vardır. Eğer bir fonksiyonun bize birden fazla değer vermesini istiyorsak geri dönüş değerini bir demet ya da liste gibi bileşik bir tür olarak düzenlemeliyiz.

Örneğin içerisinde sayılar olan dolaşılabilir bir nesneyi parametre olarak alıp onun toplam değerine geri dönen bir fonksiyon yazalım:

```
def gettotal(l):
    total = 0
    for i in l:
        total += i
    return total

result = gettotal([1, 2, 3, 4, 5])
print(result)
```

Burada biz bu fonksiyonu yalnızca bir listele içinde sayı olan ve dolaşılabilir olan geriye çağrılabilecektir. Örneğin:

```
result = gettotal(range(10))
print(result)
```

Şimdi ikinci derece denklemin köklerini bize bir demet biçiminde veren fonksiyonu yazalım:

```
import math

def getroots(a, b, c):
```

```

delta = b * b - 4 * a * c
if delta < 0:
    return None
x1 = (-b - math.sqrt(delta)) / (2 * a)
x2 = (-b + math.sqrt(delta)) / (2 * a)

return x1, x2

def main():
    a = float(input('a:'))
    b = float(input('b:'))
    c = float(input('c:'))

    result = getroots(a, b, c)
    if result == None:
        print('Kök yok')
    else:
        x1, x2 = result
        print('x1 = {}, x2 = {}'.format(x1, x2))

main()

```

Daha önceden de belirttiğimiz gibi Python'da diğer bazı dillerde olduğu gibi programın başlangıç noktası olan (entry point) main gibi bir fonksiyon yoktur. Python programları dosyanın başından itibaren satır satır çalışmaktadır. Fakat istersek biz yukarıdaki gibi programımız sanki main benzeri bir fonksiyondan başlayarak çalışıyoymuş etkisini oluşturabiliriz. Örneğin:

```

def main():
    ...
main()

```

Sınıf Çalışması: Parametre olarak dolaşılabilir bir nesne alan ve geri dönüş değerleri olarak iki listeyi demet biçiminde veren get_odd_even isimli fonksiyonu yazınız. Fonksiyon parametresiyle aldığı dolaşılabilir nesnedeki tek sayıları ve çift sayıları iki ayrı listede toplayıp (aritmetik anlamda değil) ilk elemanı tek sayılarından oluşan, ikinci elemanı çift sayılarından oluşan bir demeti geri döndürmektedir.

Cevap:

```

def get_odd_even(iterable):
    odd = []
    even = []
    for x in iterable:
        if x % 2 == 0:
            even.append(x)
        else:
            odd.append(x)

    return odd, even

odd, even = get_odd_even([1, 2, 3, 4, 5])
print(odd, even)

```

Sınıf Çalışması: Dolaşılabilir bir nesneyi parametre olarak alan ve bu nesnedeki değerlerin ortalaması ve standart sapmasını bir demet biçiminde geri döndüren get_mean_stddev isimli fonksiyonu yazınız. Standart sapma için aşağıdaki formül kullanılacaktır:

$$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$$

Cevap:

```
import math

def get_mean_stddev(iterable):
    total = 0
    count = 0
    for x in iterable:
        total += x
        count += 1
    mean = total / count

    total = 0
    for x in iterable:
        total += (x - mean) ** 2
    stddev = math.sqrt(total / count)

    return mean, stddev

mean, stddev = get_mean_stddev([1, 2, 5, 6, 8])
print(mean, stddev)
```

Parametre Değişkenlerinin Default Değer Alması Durumu

Python'da fonksiyonların parametre değişkenleri default değer alabilmektedir. Biz default değer almış olan parametreler için o fonksiyonu çağırırken argüman girmeyebiliriz. Eğer default değer almış olan parametreler için argüman girmediysek sanki onun için o default değerler argüman olarak girilmiş gibi bir etki oluşur. Örneğin:

```
def foo(a, b = 100, c = 200):
    print('a =', a, 'b =', b, 'c =', c)

foo(1)          # foo(1, 100, 200)
foo(1, 2)       # foo(1, 2, 200)
foo(1, 2, 3)    # foo(1, 2, 3)
```

Fonksiyon tanımlamasında default değer alan parametreler almayanların sağında bulunmak zorundadır. Yani parametre listesinde önce default değer almayan parametreler, sonra default değer alan parametreler bulunmak zorundadır. Başka bir deyişle bir parametre değişkeni default değer alırsa onun sağındakilerin de default değer almış olması gereklidir. (Bu konuda biraz daha ayrıntılı ele alınmaktadır.) Örneğin:

```
def foo(a = 10, b = 20, c):
    print('a = {}, b = {}, c = {}'.format(a, b, c))
```

Bu tanımlama geçersizdir. Çünkü default değer alan parametreler default değer almayanların sağında bulunmak zorundadır. Ya da başka bir deyişle bir parametre değişkeni default değer alırsa onun sağındaki parametre değişkenlerinin hepsinin default değer almış olması gereklidir. Halbuki yukarıdaki tanımlamada bu durum söz konusu değildir.

Default argüman sayesinde bazı çok kullanılan değerler için giriş yapma zahmeti ortadan kaldırılmaktadır. Örneğin:

```
def disp_banner(s, ch = '-'):
    print(ch * len(s))
    print(s)
    print(ch * len(s))

disp_banner('Ali')
disp_banner('Ali', '**')
```

Süphesiz parametre değişkenlerine verilen default değerlerin çok kullanılan değerler olması beklenir. Aksi takdirde kodun anlamlandırılması zorlaşır. Örneğin:

```
def add(a = 10, b = 20): return a + b
```

Burada böyle bir fonksiyonun default değer alması anlamsızdır ve iyi bir teknik değildir.

Örneğin range fonksiyonunu biz tek argümanla çağırırsak bu argüman stop değeri anlamına gelmektedir. Bu durumda start değeri 0 olur. Eğer biz bu fonksiyonu iki argümanla çağırırsak bu durumda ilk argüman start ikinci argüman stop değeri olur. Eğer biz bu fonksiyonu üç argümanla çağırırsak ilki start, ikincisi stop, üçüncü step değeri olmaktadır. Şimdi biz bu fonksiyonun benzerini yazmaya çalışalım:

```
def disp_range(start, stop=None, step=1):
    if stop == None:
        stop = start
        start = 0

    for i in range(start, stop, step):
        print(i, end=' ')
```



```
disp_range(10)
disp_range(2, 8)
disp_range(2, 8, 3)
```

Sınıf Çalışması: İki parametresi olan disp_chars isimli fonksiyonu aşağıdaki şekli çıkartacak biçimde yazınız ve çağrılarınız:

```
def disp_chars(n, ch = '*')
```

Bu fonksiyon aşağıdaki kalıbı ekrana basmalıdır:



Çözüm:

```
def disp_chars(n, ch = '*'):
    for i in range(1, n + 1): print(ch * i)
```

```
disp_chars(10)
disp_chars(10, '-')
```

Python'ın standart kütüphanesindeki pek çok fonksiyon default değer almış parametrelere sahiptir. Örneğin aslında int türüne dönüştürme yapan int fonksiyonunun ikinci bir default değer almış parametresi vardır. Bu parametre dönüştürmenin kaçlık sisteme göre yapılacağını belirtmektedir. Örneğin:

```
s = '123'
i = int(s)
print(i)
```

Burada dönüştürme default olarak 10'luk sisteme göre yapılmaktadır. Biz istersek istediğimiz tabana göre dönüştürme yaptmak için fonksiyonun base parametresi için argüman girebiliriz. Örneğin:

```
s = '123'  
i = int(s, 16)  
print(i)
```

Fonksiyon Çağrılarda İsimli Argümanların Kullanılması

Bir fonksiyon çağrılarında argümandan parametrenin ismi "isim = ifade" biçiminde belirtilebilir. Bu biçim adeta "ben bu argümanı falanca parametre için girdim" anlamına gelmektedir. Bu sayede biz argümanları karışık sırada da girebiliriz. Örneğin:

```
def disp_banner(text, ch = '-'):  
    print(ch * len(text))  
    print(text)  
    print(ch * len(text))  
  
disp_banner(ch='*', text='Ankara')
```

Burada artık argümanlar isimlendirilmiştir, pozisyonuna değil isme bakılmaktadır. Örneğin:

```
def foo(a, b, c):  
    print('a = {}, b = {}, c = {}'.format(a, b, c))  
  
foo(10, 20, 30)  
foo(c=100, b=200, a=100)  
foo(c=100, a=200, b=300)
```

Python'ın orijinal referans kitabında isim almamış argümanlara "pozisyonel argümanlar (positional arguments)", isim almış argümanlara ise "isimli argümanlar (keyword arguments)" denilmektedir. Kural olarak çağrıma sırasında önce pozisyonel argümanlar sonra isimli argümanlar belirtilmek zorundadır. Başka bir deyişle bir isimli argümanın sağındaki tüm argümanların da isimli olması gereklidir. Örneğin foo fonksiyonu şöyleden tanımlanmış olsun:

```
def foo(a, b, c):  
    pass
```

Aşağıdaki çağrımları geçerli değildir:

```
foo(10, b=20, 30)      # error!  
foo(a=100, 200, 300)   # error!
```

Ancak aşağıdaki çağrımları geçerlidir:

```
foo(10, 20, c=30)      # geçerli  
foo(10, c=30, b=20)    # geçerli
```

Örneğin:

```
def foo(a, b, c):  
    print('a = {}, b = {}, c = {}'.format(a, b, c))  
  
foo(10, 20, 30)          # geçerli  
foo(10, c=30, b=20)      # geçerli  
foo(c=30, a=10, b=20)    # geçerli
```

İsimli argümanlar özellikle çok sayıda default değer alan parametrelere sahip fonksiyonlarda yalnızca bazı parametrelere için değer girilmek istendiğinde tercih edilmektedir. Bu tür durumlarda isimli argümanlar yazım kolaylığı sağlamaktadır. Örneğin:

```

def foo(a, b, c = 10, d = 20, e = 30, f = 40):
    print("a = {}, b = {}, c = {}, d = {}, e = {}, f = {}".format(a, b, c, d, e, f))

foo(1, 2, 3, f=100)  # Burada boşuna d ve e için argüman girişi yapmadık

```

Bir fonksiyon çağrılarında fonksiyonun (* ve ** dışındaki) her parametresi bir ve yalnızca bir kez değer almak zorundadır. Başka bir deyişle çağrımda fonksiyonun (* ve ** parametreleri dışındaki) tüm parametreleri değer almak zorundadır ve bir parametre birden fazla kez değer alamaz. Örneğin:

```

def foo(a, b, c = 10, d = 20, e = 30, f = 40):
    print("a = {}, b = {}, c = {}, d = {}, e = {}, f = {}".format(a, b, c, d, e, f))

```

Fonksiyonu söz konusu olsun:

```

foo(10)                      # geçersiz!
foo(10, 20)                   # geçerli!
foo(100, c=200, b=300)        # geçerli
foo(100, c=200, b=300, d=400)  # geçerli
foo(100, 200, b=300)          # geçersiz!

```

Yalnızca İsimli ve Yalnızca Pozisyonel Olarak Kullanılabilen Argümanlar

Fonksiyon parametre bildiriminde yalnızca '*' karakterinden oluşan bir parametrenin sağındaki tüm parametreler için girilecek argümanlar isimli olmak zorundadır. Örneğin:

```

def foo(a, b, *, c, d):
    print(a, b, c, d)

foo(10, 20, c=10, d=20)      # geçerli
foo(10, 20, d=20, c=20)      # geçerli

```

Burada foo fonksiyonu çağrırlarken '*' parametresinin yanında bulunan c ve d parametreler için argüman belirtilmek zorundadır. Örneğin aşağıdaki çağrımlar error ile sonuçlanacaktır:

```

foo(10, 20, 30, 40)          # error!
foo(10, 20, 30, d=40)        # error!

```

Python'ın 3.8 sürümüyle birlikte yalnızca pozisyonel olarak kullanılabilen argümanlar da dile eklenmiştir. Eğer fonksiyon tanımlamasında bir parametre yalnızca '/' karakteri içeriyorsa çağrıma sırasında onun solundaki parametreleri isimli olarak kullanamayız. Örneğin:

```

def foo(a, b, /, c, d):
    print(a, b, c, d)

foo(10, 20, c=30, d=40)      # geçerli

```

Fakat örneğin:

```

foo(a=10, b=20, c=30, d=40)  # error!

```

Yalnızca pozisyonel olarak kullanılabilen argümanların Python'a şu andaki son versiyon olan 3.8 ile eklendiğini bir kez daha belirtmek istiyoruz. Bu özellik daha önceki Python yorumlayıcılarında kullanılamamaktadır.

Yalnızca isimli ve yalnızca pozisyonel argüman kullanımı birlikte de belirtilebilir. Örneğin:

```

def foo(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

```

Burada a ve b parametreleri için argümanlar isimsiz biçimde, e ve f parametreleri için argümanlar isimli biçimde girilmek zorundadır. Ancak c ve d parametreleri için argümanlar isimli ya da isimsiz biçimde girilebilir. Tabii eğer * ile / bir arada kullanılacaksa '/nün *'in solunda olması gereklidir. (Tersi durumun anlamsız olduğuna dikkat ediniz.)

Fonksiyonların Tek Yıldızlı ve Çift Yıldızlı Parametreleri

Fonksiyonlarda bir parametre değişkeninin önüne '*' getirilirse bunun özel bir anlamı vardır. '*' parametresi birden fazla argüman ile eşleşir. '*' parametresiyle eşleşen argümanlar bir demet (tuple) olarak fonksiyona aktarılmaktadır. '*' parametresi sayesinde fonksiyonlara değişken sayıda argüman aktarmak mümkün olmaktadır. Örneğin:

```
def foo(*a):
    print(a)

foo()                  #
foo(10)                # (10,)
foo(10, 20)             # (10, 20)
foo(10, 20, 30)         # (10, 20, 30)
```

Tabii '*'li parametreye karşı gelen argümanlar farklı türlerden de olabilirler. Örneğin:

```
foo('Ali', 10)
foo('Veli', 'Selami', 10, 10.2)
foo('Ayşe', [10, 20, 30], (40, 50, 60))
```

Bir fonksiyonda yalnızca bir tane tek * parametresi olabilir. Ancak bir fonksiyonda hem normal parametreler hem de tek * parametresi birlikte bulunabilir. Örneğin:

```
def foo(a, *b):
    print('a = {}, b = {}'.format(a, b))

foo(10, 20, 30)
```

Burada 10 a parametresine 20 ve 30 da bir demet olarak b parametresine aktarılacaktır.

Fonksiyonun hem normal hem de '*'li parametresi varsa önce normal parametrenin gelmesi zorunlu değildir. Tabii böyle bir durumda bizim '*'li olmayan parametreler için mecburen argümanları isimli girmemiz gereklidir. Örneğin:

```
def foo(*a, b):
    print('a = {}, b = {}'.format(a, b))

foo(10, b = 100)          # a = (10,), b = 100
foo(10, 20, b=100)        # (10, 20), b = 100
foo(10, 20, b=100)        # (10, 20), b = 100
foo(10, 20, 30)           # geçerli değil! b değer almamış
```

Burada biz mecburen b'yi argüman listesinde isimli olarak kullanmak zorunda kaldık. Aksi takdirde b değer almamış olurdu. Bu da çalışma zamanı sırasında exception'a yol açardı. Çünkü çağrımda bütün parametrelerin bir ve yalnızca bir kez değer almış olması zorunluluğu vardır. Tabii aslında * parametresinin normal parametrelerden sonra bulundurulması daha sık karşılaşılan durumdur.

Aslında print fonksiyonu da -birden fazla argümanın değerini yazdırabiligine göre- '*'li parametreye sahiptir. Gerçekten de print fonksiyonun parametrik yapısı Python dokümanlarında şöyle belirtilmiştir:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Örneğin biz de print fonksiyonuna benzer bir myprint fonksiyonu yazabilirdik. Fakat bu fonksiyonda biz yine orijinal print fonksiyonunu kullanacağımız:

```

def myprint(*Objects, sep = ' ', end = '\n'):
    i = 0
    while i < len(Objects):
        if i != 0:
            print(sep, end='')
        print(Objects[i], end='')
        i += 1
    print(end = end)

myprint(10, 20)
myprint(30, 40)
myprint(10, 20, 30, sep=',')

```

Aynı fonksiyonu for döngüsü kullanarak şöyle de yazabilirdik:

```

def myprint(*Objects, sep = ' ', end = '\n'):
    if len(Objects) == 0:
        print(end = end)
        return

    for obj in Objects[:-1]:
        print(obj, end=sep)
    print(Objects[-1], end=end)

myprint(10, 20)
myprint(30, 40)
myprint(10, 20, 30, sep = ',')

```

Gerçekten de orijinal print fonksiyonunun değişken sayıda argüman alan birinci parametresinden sonra sep ve end parametreleri de vardır. sep parametresi birden fazla değerin yazdırılacağı durumda bunların arasının hangi karakterle ayrılacağını belirtir. end parametresi ise en sonunda basılacak karakteri belirtmektedir.

Fonksiyonların ayrıca bir de ** parametresi olabilmektedir. Eğer bu parametre bulundurulacaksa parametre listesinin sonunda olmak zorundadır. ** parametresi, olmayan bir parametrenin argüman olarak isimli bir biçimde kullanılmasına olanak sağlar. Herhangi bir parametreye karşılık gelmeyen isimli biçimde kullanılmış tüm argümanlar yorumlayıcı tarafından bir sözlük nesnesi oluşturularak onun içerisinde yerleştirilir. Bu sözlük nesnesi de ** ile belirtilen parametreye geçirilir. Tabii fonksiyonu çağırın kişi eğer olmayan bir parametre ismini kullanmamışsa bu sözlük parametresi boş bir sözlük biçiminde olacaktır. Örneğin:

```

def foo(a, b, **c):
    print("a = {}, b = {}, c = {}".format(a, b, c))

```

Fonksiyonu şöyle çağrırmış olalım:

```
foo(b = 10, a=20, xx=30, yy=40)
```

Çağrıda fonksiyonun xx ve yy parametresi olmadığı için bu iki parametre anahtar 'xx', değer 30 ve anahtar 'yy', değer 40 olacak biçimde bir sözlük oluşturularak fonksiyona aktarılmıştır. Programın çıktısı şöyle olur:

```
a = 20, b = 10, c = {'xx': 30, 'yy': 40}
```

Tabii çağrımda sırasında olmayan parametrelere ilişkin isimli argümanlar herhangi bir sırada belirtilebilmektedir. Örneğin:

```
foo(xx=30, b=10, a=20, yy=40)
```

Programcı genellikle **'lı parametredeki argümanları ve değerleri elde edip işleme sokmak ister. Bir sözlüğün nasıl dolaşıldığını önceki bölümlerde görmüştük:

```

def foo(a, b, **c):
    print('a = {}, b = {}, c = {}'.format(a, b, c))
    for key, val in c.items():
        print('{} ---> {}'.format(key, val))

foo(10, 20, ali=100, veli=200, selami=300)

```

** parametresine karşı gelen sözlüğün anahtarlarının string türünden değerlerinin de isimli argümanda belirtilen türden olduğuna dikkat ediniz. Biz anahtarı string olmayan bir **'lı sözlük oluşturamayız. Yani örneğin aşağıdaki gibi bir çağrı geçerli değildir:

```
foo(10, 20, 30='Ali') # geçerli değil!
```

Yukarıda da belirttiğimiz gibi eğer biz olmayan bir parametre ismini isimli biçimde argümanda kullanmamışsa bu sözlük parametresine boş bir sözlük nesnesi geçirilir:

```
foo(b=10, a=20)
```

Çıktı olacaktır:

```
a = 20, b = 10, c = {}
```

Şimdi hem '*'lı hem de '**'lı parametreye sahip biraz daha karmaşık fonksiyon örneği verelim:

```

def foo(a, b, *c, d = 100, **e):
    print('a = {}, b = {}, c = {}, d = {}, e = {}'.format(a, b, c, d, e))

foo(10, 20, 30, 40, xx = 50, yy = 60)

```

Çıktı şöyle olacaktır:

```
a = 10, b = 20, c = (30, 40), d = 100, e = {'xx': 50, 'yy': 60}
```

Örneğin:

```
foo(10, 20)
```

Bunun da çıktısı şöyle olacaktır:

```
a = 10, b = 20, c = (), d = 100, e = {}
```

Biz '**'lı parametredeki sözlüğün içerisinde bazı parametre isimlerinin olup olmadığını kontrol edip duruma göre uygun işlemler yapabiliriz:

```

def printmsg(s, **d):
    count = d.get('count')
    if count == None:
        count = 1

    print(s * count)

printmsg('ok', count=4)

```

* ve '**'lı parametreler için argüman girme zorunluluğunun olmadığına dikkat ediniz.

Pekiyi fonksiyonun '**'lı parametreleri neden programcılar tarafından kullanılmaktadır? Bazı fonksiyonların çok sayıda parametresi olabilmektedir. Örneğin foo isimli bir fonksiyonun a ve b'nin dışında 50 tane daha parametresinin

olduğunu düşünelim. Bu 50 parametreyi fonksiyon tanımlamasında tek tek belirtmek, sonra da bu parametrelerin değerlerini fonksiyon içerisinde ele almak oldukça zordur. Zaten bir fonksiyonun 10'dan fazla parametreye sahip olması iyi bir teknik kabul edilmemektedir. İşte fonksiyonların **'lı parametreleri çok zayıda parametreye sahip fonksiyonların daha makul bir biçimde tanımlanıp çağrılmasını sağlayabilmektedir. Örneğin:

```
def foo(a, b, **kwargs):
    pass
```

Burada fonksiyonun a ve b dışındaki 50 parametresini biz tanımlamada belirtmek zorunda kalmadık. Tabii fonksiyonun içerisinde biz bu parametreleri çok hızlı bir biçimde alıp işleme sokabiliriz. Bir fonksiyonun **'lı bir parametreye sahip olması bizim o fonksiyonu istediğimiz gibi uydurma parametre isimleriyle çağırabileceğimiz anlamına gelmemektedir. ** parametresine sahip fonksiyonlarda genellikle fonksiyonu yazanlar bu ** için hangi isimli parametrelerin geçerli olduğunu ve bunların ne anlam ifade ettiklerini dokümantasyonda belirtirler ve fonksiyonun başında girilen isimli argümanların geçerliliğini kontrol ederler.

Örneğin:

```
def foo(a, b, **kwargs):
    legal_args = ['width', 'height', 'color', 'spec']
    for key in kwargs:
        if key not in legal_args:
            print('{} is invalid arguments'.format(key))
            return

    width = kwargs.get('width', 1)
    height = kwargs.get('width', 1)
    color = kwargs.get('color', 'red')
    spec = kwargs.get('spec', 'default')

    print(f'a = {a}, b = {b}, width = {width}, height = {height}, color = {color}, spec =
{spec}')

foo(10, 20)
foo(10, 20, color='blue', spec='x12')
foo(10, 20, height=10, color='red')
```

Burada foo fonksiyonun aslında width, height, color ve spec parametreleri vardır. Bu parametreler ayrı ayrı değil **kwargs parametresiyle bir sözlük biçiminde alınmaktadır. Fonksiyon içerisinde girilen parametre isimlerinin geçerliliği kontrol edilmiştir. Eğer ilgili parametre çağrıda hiç kullanılmamışsa onun için default değerlerin alındığına dikkat ediniz.

Fonksiyonun parametrelerini organize ederken hangi parametrelerin hangi sırada yazılması gerekiğine yönelik kuralların bazı ayrıntıları vardır. Bu ayrıntılar şöyledir:

- 1) Eğer bir parametre değişkeni default değer almışsa '*'lı parametreye kadar (tabii '*'lı parametre varsa) onun sağındaki parametrelerin hepsinin default değer olması gereklidir. Ancak '*'lı parametreden sonra default değer alan ve default değer almayan parametreler karışık sırada bulunabilirler.
- 2) Fonksiyonda en fazla bir tane '*'lı ve en fazla bir tane '**'lı parametre bulunabilir. (Tabii '*'lı ve '**'lı parametre bulunmak zorunda değildir.)
- 3) '**'lı parametre eğer bulundurulmuşsa parametre listesinin en sonunda olmak zorundadır.

Örneğin:

```
def foo(a, b = 10, c, *d): # geçersiz!
    pass
```

Burada b parametresi default değer aldığı için '*'li parametreye kadar diğer parametrelerin hepsinin default değer alması gerekiirdi. Örneğin aşağıdaki fonksiyon tanımlaması geçerlidir:

```
def foo(a, b, *c, d = 100, e):
    print("a = {}, b = {}, c = {}, d = {}, e = {}".format(a, b, c, d, e))
```

Fonksiyonu şöyle çağrırmış olalım:

```
foo(10, 20, 30, 40, 50, e=60)
```

Şöyle bir sonuç elde deriz:

```
a = 10, b = 20, c = (30, 40, 50), d = 100, e = 60
```

Örneğin:

```
def foo(a, b = 10, c = 20, *d, e = 30, f, **g):           # geçerli
    pass
```

Burada tüm kurallara uyulmuştur. Örneğin:

```
def foo(a, b = 10, c = 20, *d, e = 30, **f, g):          # geçersiz!
    pass
```

'**'li parametrenin varsa sonda olması gereklidir. Tabii bu kuralların nedenleri vardır. Ancak burada bu neden üzerinde durulmayacaktır.

Fonksiyon Çağrılarında Tek Yıldızlı ve Çift Yıldızlı Argümanlar

Bir fonksiyon çağrırlarında argümanlarda da '*' ve '**' kullanılabilmektedir. Örneğin:

```
foo(x, y, *z, **k)
```

gibi. Argümanlardaki '*' ve '**' ters bir anlam ifade etmektedir. Bunları açıklayalım.

Argümanda '*' varsa bunun yanında dolaşılabilir (iterable) bir nesnenin (demet, liste, sözlük, küme vs.) olması gereklidir. Bu durumda parametredekilerin tam tersine bu dolaşılabilir nesnenin elemanları sıradaki parametre değişkenlerine tek tek dağıtılr. Başka bir deyişle argümandaki '*' ile belirtilen dolaşılabilir nesnenin elemanları sanki argümanlarda ayrı ayrı yazılmış gibi bir durum söz konusu olur. Örneğin:

```
def foo(a, b, c, d, e):
    print("a = {}, b = {}, c = {}, d = {}, e = {}".format(a, b, c, d, e))

t = (30, 40)
foo(10, 20, *t, 50)      # a = 10, b = 20, c = 30, d = 40, e = 50
```

Burada eğer t'nin solunda '*' olmasaydı bu durumda t yalnızca c parametre değişkeni ile eşleştirilirdi. Fakat '*' olduğu için artık t'nin içeriği sıradaki parametre değişkenlerine (yani c ve d'ye) dağıtılmaktadır. Başka bir deyişle:

```
t = (30, 40)
foo(10, 20, *t, 50)      # a = 10, b = 20, c = 30, d = 40, e = 50
```

çağıırısı ile aşağıdaki çağrı eşdeğerdir:

```
t = (30, 40)
foo(10, 20, 30, 40, 50)  # a = 10, b = 20, c = 30, d = 40, e = 50
```

Yukarıdaki örnekte '*'in sağında bir demet vardır. Ancak başka bir dolaşılabilir (iterable) nesne de olabilirdi. Örneğin:

```
l = [30, 40]
foo(10, 20, *l, 50)      # a = 10, b = 20, c = 30, d = 40, e = 50
```

Tabii bu dolaşılabilir (iterable) nesneler doğrudan da yazılabildi:

```
foo(10, 20, *(30, 40), 50)
foo(10, 20, *[30, 40], 50)
```

Fonksiyonların tek yıldızlı argümanları tek yıldızlı parametre tarafından da alınabilir. Örneğin:

```
def foo(a, b, *c):
    print('a = {}, b = {}, c = {}'.format(a, b, c))

l = [10, 20, 30, 40, 50]
foo(10, 20, *l)
```

Burada listenin elemanları tek yıldızlı parametre tarafından alınacaktır. Programın ekran çıktısı şöyle olacaktır:

```
a = 10, b = 20, c = (10, 20, 30, 40, 50)
```

Örneğin:

```
def foo(a, b, c, *d):
    print('a = {}, b = {}, c = {}, d = {}'.format(a, b, c, d))

foo(100, *range(10))
```

Buradaki çağrıının eşdeğeri şöyledir:

```
foo(100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Örneğin:

```
def foo(a, b, c, *d):
    print('a = {}, b = {}, c = {}, d = {}'.format(a, b, c, d))

foo(100, *'ankara')
```

Buradaki çağrıının eşdeğeri de şöyledir:

```
foo(100, 'a', 'n', 'k', 'a', 'r', 'a')
```

Örneğin:

```
def foo(a, b, c, d, e):
    print(a, b, c, d, e)
```

```
def bar(*a):
    foo(*a)
```

```
l = [10, 20, 30, 40, 50]
bar(*l)
```

Burada bar fonksiyonundaki a parametresi bir demettir. foo fonksiyonu da bu demetin elemanlarıyla çağrılmıştır.

Aşağıdaki örneğe dikkat ediniz:

```
def foo(*a):
    print(a)
    print(*a)

foo(10, 20, 30)
```

Benzer biçimde iki çağrı arasındaki farka dikkat ediniz:

```
print('ankara')
print('*'+'ankara')
```

Burada 10, 20, 30 argümanları a'ya bir demet olarak aktarılacaktır. Bu durumda foo'daki ilk print çağrısı bir demeti yazdırırken, ikinci çağrı demetin elemanlarını tek tek yazdıracaktır. Programın çıktısı şöyle olacaktır:

```
(10, 20, 30)
10 20 30
```

Örneğin:

```
def printrev(*a):
    print(*reversed(a))

printrev(10, 20, 30, 40, 50)
```

Burada printrev fonksiyonu girilen argümanları ters sırada ekrana yazdırmaktadır.

Örneğin:

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for b in a:
    print(*b)
```

*'lı ifadeler argümanlar dışında demetlerde ve listelerde de kullanılabilir. Örneğin:

```
a = [1, 2, 3]
b = [10, *a, 20, 30]

print(b)
```

Burada aslında b listesinin oluşturulma ifadesi aşağıdaki ile eşdeğerdir:

```
b = [10, 1, 2, 3, 20, 30]
```

Benzer biçimde '*'lı ifadeler demetlerde de kullanılabilir. Ancak sözlüklerde kullanılamazlar. Örneğin:

```
a = [1, 2, 3]
b = (10, *a, 20, 30)

print(b)
```

Argümandaki ** 'ın sağına bir sözlük nesnesi getirilebilir. Bu sözlüğün anahtarları parametre değişkenlerinin isimlerini (yani anahtarlar str türünden olmak zorundadır), değerleri de bunlara aktarılacak değerleri belirtmektedir. Örneğin:

```
def foo(a, b, c, d, e):
    print("a = {}, b = {}, c = {}, d = {}, e = {}".format(a, b, c, d, e))
```

```
d = {'c': 30, 'd': 40, 'e': 50}
foo(10, 20, **d) # a = 10, b = 20, c = 30, d = 40, e = 50
```

Burada d argümanındaki sözlük elemanları açılarak sırasıyla c, d ve e parametreleri için isimli argümanlar değerler haline getirilir. Yani yukarıdaki çağrıın eşdeğeri şöyledir:

```
foo(10, 20, c=30, d=40, e=50)
```

Olmayan parametre isimleri de bu yolla argüman olarak verilebilir. Tabii bu argümanları almak için **'lı bir parametre gerekecektir. Örneğin:

```
def foo(a, b, c, d, e, **f):
    print("a = {}, b = {}, c = {}, d = {}, e = {}, f = {}".format(a, b, c, d, e, f))

d = {'c': 30, 'd': 40, 'e': 50, 'x': 100, 'y': 200}
foo(10, 20, **d) # a = 10, b = 20, c = 30, d = 40, e = 50, x = 100, y = 200
```

Örneğin:

```
def revprint(*a, **b):
    print(*reversed(a), **b)

revprint(10, 20, 30, 40, 50, sep=',')
```

Buradaki sep isimli argümanı ** parametreli b tarafından alınacaktır. b parametresi bir sözlük belirttiğine göre print fonksiyonu **b ile çağrıldığında aslında sep isimli argümanıyla çağrılmış olacaktır.

Python'da fonksiyon çağrıısında isimli olmayan ve **'lı olmayan argümanlara isimsiz argümanlar denilmektedir. Aynı zamanda isimsiz ve *'lı argümanlar da bir arada "pozisyonel argümanlar (positional argument)" olarak isimlendirilmektedir. Örneğin:

```
foo(10, 20, c=30, d=40, *t, **d)
```

Burada 10 ve 20 isimsiz argümanlar, c ve d isimli (keyword) argümanlar, t *'lı ve d de **'lı argümandır. Bu örnekte 10, 20 ve *t için açılan argümanlara aynı zamanda pozisyonel argüman da denilmektedir.

**'lı ifadeler sözlük oluştururken de kullanılabilmektedir. Örneğin:

```
d = {'ali': 10, 'veli': 20, 'selami': 30}
k = {**d, 'ayşe': 40, 'fatma': 50}

print(k)
```

Argüman oluştururken sıralama için izlenecek kurallar şunlardır:

1) Çağrıda isimli argümanlardan sonra isimsiz argümanlar getirilemez. Yani isimli argümanlar isimsiz argümanların sağında olmak zorundadır. Ancak isimli argümanlardan sonra *'lı, **'lı ve isimli argümanlar karışık sırada bulundurulabilirler (**'lı argümanlar isimli argüman olarak ele alınmaktadır).

2) *'lı ve **'lı argümanlar birden fazla kez çağrıda bulunabilirler. Ancak *'lı argümanların **'lı argümanların solunda bulunması gereklidir.

Python'ın referans kitaplarında belirtilen bu kurallara maalesef CPython gerçekleştirimi tarafından tam olarak uygulamamaktadır.

Argümanlarla parametrelerin eşleştirilmesi de şöyle yapılmaktadır: Çağrım sırasında önce yalnızca pozisyonel argümanlara bakılır. (Pozisyonel argümanlar isimsiz argümanlardır, *'lı argümanlar da pozisyonel kabul edilmektedir.)

Eğer çağrımda hangi sırada olursa olsun n tane pozisyonel argüman varsa bunlar ilk n tane parametreyle eşleştirilir. Sonra isimli argümanlar ele alınarak bunlar da uygun parametrelerle eşleştirilir. (İsimli argümanlar da **'lı biçimde bulunuyor olabilirler.) Bu işlemler sonucunda eğer parametrelerle eşleştirilemeyen isimli argümanlar varsa bunlar da fonksiyonun **'lı parametresiyle eşleştirilmektedir. Çağrım sonucunda *'lı ve **'lı olmayan ve default değer almayan her parametrenin bir ve yalnızca bir argümanla eşleştirilmiş olması gerekmektedir.

Örneğin aşağıdaki gibi bir foo fonksiyonu olsun:

```
def foo(a, b, c, d, e, f):
    print('a = {}, b = {}, c = {}, d = {}, e = {}, f = {}'.format(a, b, c, d, e, f))
```

Aşağıdaki çağrı geçerlidir:

```
foo(10, 20, e=30, *(40, 50), f=60)
```

Bu çağrıda 10, 20, 40, 50 pozisyonel argümanlardır. Bu nedenle önce bu pozisyonel argümanlar ilk n tane parametreyle eşleştirilir. Sonra isimli argüman olan e ve f'nin eşleştirilmesi yapılacaktır. Dolayısıyla programın çıktısı şöyle olacaktır:

```
a = 10, b = 20, c = 40, d = 50, e = 30, f = 60
```

Aşağıdaki çağrı geçerlidir:

```
foo(10, e=50, *(30, 40), **{'d':100, 'f': 200})
```

Çıktısı şöyle olacaktır:

```
a = 10, b = 30, c = 40, d = 100, e = 50, f = 200
```

Aşağıdaki çağrı da geçerlidir:

```
foo(10, 20, *[40], **{'d':100}, **{'e':200, 'f': 300})
```

Burada da bütün kurallara uyulmuştur. Ancak aşağıdaki çağrı geçersizdir:

```
foo(10, c=20, *(30, 40, 50, 60, 70))
```

Burada c birden fazla kez değer almıştır.

Fonksiyon Parametrelerinde Tür Kontrolü ve isinstance Fonksiyonu

Python dinamik tür sistemine sahip bir dil olduğu için biz fonksiyonları ve metotları herhangi türden argümanlarla çağrıabilirmiz. Fakat fonksiyonlara ve metotlara yanlış türden argüman geçmek exception oluşmasına yol açabilmektedir. Örneğin:

```
def disp_banner(s, ch = '-'):
    print(ch * len(s))
    print(s)
    print(ch * len(s))

disp_banner('ankara')
disp_banner(120)
```

Burada bizim normal olarak dispbanner fonksiyonunu bir string argümanıyla çağrırmamız gereklidir. Eğer biz bu fonksiyona yanlış argüman geçersek (örneğin int bir değer) programın çalışma zamanı sırasında exception oluşacaktır. İşte bazen fonksiyonlar ve metotlar çeşitli türden parametreleri kabul edebilmektedir. Tabii bunun için programcının fonksiyonun içerisinde tür kontrolü yapması gereklidir. Tür kontrolü için en çok kullanılan fonksiyon

`isinstance` isimli built-in fonksiyondur. Bu fonksiyon bizden iki parametre ister. Fonksiyonun birinci parametresi türü kontrol edilecek ifadeyi, ikinci parametresi ise kontrol edilecek türü belirtir. `isinstance` fonksiyonu bool bir değere geri dönmektedir. Örneğin:

```
def disp_banner(s, ch = '-'):
    if isinstance(s, int):
        s = str(s)
    print(ch * len(s))
    print(s)
    print(ch * len(s))

disp_banner('ankara')
disp_banner(120)
```

Burada `dispbanner` artık `int` türünden argümanı da kabul etmektedir. Özette bir Python fonksiyonu ya da metodу değişik türlerle çağrılabılır. O fonksiyonu ya da metodу yazan kişi fonksiyon ya da metodun içerisinde `isinstance` fonksiyonu ile tür kontrolü yapmış olabilir. Genellikle programcılar fonksiyonun ya da metodun doğru türden argümanla çağrılp çağrılmadığını test etmezler. Zaten yanlış argümanla çağrılmış olan fonksiyonlar ve metodlar bir biçimde exception oluşturacaktır. Fonksiyonu ya da metodу doğru argümanla çağrırmak onu çağrıranın sorumluluğundadır. Şüphesiz fonksiyonları ya da metodları yazan kişiler onların parametrelerinin hangi türleri kabul ettiğini dokümante etmeli dirler.

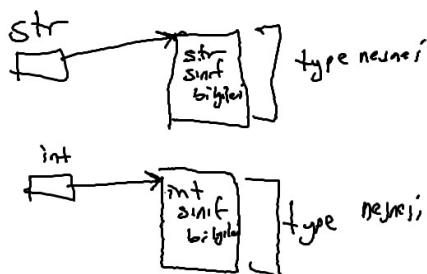
`isinstance` fonksiyonunun ikinci parametresi türlerden oluşan bir demet olarak (dolaşılabilir nesne değil demet) girilebilir. Bu durumda ilgili nesnenin bu türlerden birine ilişkin olup olmadığı kontrolü yapılmış olur. Örneğin:

```
>>> a = 10
>>> isinstance(a, (int, float))
True
>>> a = 10.5
>>> isinstance(a, (int, float))
True
>>> a = 'ankara'
>>> isinstance(a, (int, float))
False
```

Peki yeter kontrolü için built-in `type` fonksiyonu kullanılamaz mı? Evet kullanılabilir. Ancak `isinstance` fonksiyonu ile `type` fonksiyonun etkileri farklıdır. `isinstance` türetme durumunu da dikkate almaktadır. Bu nedenle birincil olarak tercih edilir. Bir ifade `type` fonksiyonuna sokulduğunda biz o ifadenin türünü elde ederiz. Örneğin:

```
>>> type(s)
<class 'str'>
```

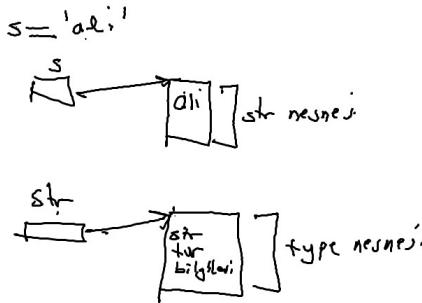
Burada `s` `str` isimli bir sınıf türündendir. `type` fonksiyonu bize ilgili ifadenin hangi türden olduğunu belirten bir sınıf referansı verir. Yani `type` fonksiyonunun geri döndürdüğü değer aslında `type` isimli bir sınıf nesnesinin adresidir. `str` gibi, `float` gibi türlerin tür isimleri de aslında bu türlerin tür bilgilerinin bulunduğu `type` türünden birer nesneyi gösteren değişkenlerdir. Başka bir deyişle `str`, `int`, `float` ve `bool` sıradan birer değişkendir. Ancak bu değişkenlerin gösterdiği yerde `type` isimli sınıf türünden nesneler bulunmaktadır. O türün bilgileri de o nesnenin içerisindeindedir. Örneğin:



Tabii Python'da int, float, str ve bool gibi tür nesnelerinin tek bir kopyası vardır. Yani aslında x ve y int türden ise type(x) ile type(y) aynı nesnenin adresini vermektedir.

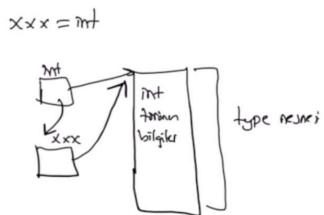
Örneğin:

```
>>> type(str)
<class 'type'>
```



Mademki aslında int, float, str gibi tür isimleri bu tür bilgilerinin tutulduğu type türünden nesnelerin adreslerini tutan birer değişkendir o halde onları da atama işleminde kullanabiliriz. Örneğin:

```
>>> xxx = int
>>> id(int)
140706832851424
>>> id(xxx)
140706832851424
>>> int is xxx
True
```



Tür isimleri aslında normal birer değişken olduğuna göre o değişkenlere başka değerler de atayabiliriz. Örneğin:

```
int = 100
```

Tabii böyle bir işlem yapmanın hiçbir pratik faydası yoktur. Biz burada yalnızca tür isimlerinin sıradan birer değişken olduğunu vurgulamak istiyoruz.

Python'da tüm türler için toplamda tek bir tane type türünden nesne yaratılmaktadır. O halde biz a ve b int türden olmak üzere type(a) ve type(b) çağrılarından aynı nesnenin adresini elde ederiz. Örneğin:

```
>>> a = 10
>>> b = 20
>>> x = type(a)
>>> y = type(b)
>>> id(x)
140705823990240
>>> id(y)
140705823990240
>>> id(int)
140705823990240
```

Bu durumda biz tür kontrolünü type fonksiyonuyla şöyle de yapabiliriz:

```
>>> s = 'ali'  
>>> type(s) == str  
True
```

Burada type fonksiyonu bize s'nin türüne ilişkin bir type nesne referansı verir. str de zaten str türüne ilişkin bir type nesne referansıdır. Bu iki referan birbirine eşitse bu da s'nin str türünden olduğu anlamına gelir.

Fakat tür kontrolünde yukarıda da belirttiğimiz gibi isinstance fonksiyonu tercih edilmelidir.

Aşağıdaki örnekte get_random isimli fonksiyon a ile b arasında (a dahil, b değil) size tane rastgele tamayı üretip onu bir liste olarak vermektedir. Ancak size değeri iki elemanlı bir demet olarak girilebilmektedir. Fonksiyon içerisinde bu durum kontrol edilmiş eğer size parametresi bir demet olarak girilmişse rastgele sayılarından oluşan matrisel bir listeye geri dönülmüştür:

```
import random  
  
def get_random(a, b, size):  
    result = []  
    if isinstance(size, int):  
        for i in range(size):  
            result.append(random.randint(a, b - 1))  
    elif isinstance(size, tuple):  
        if len(size) != 2:  
            raise TypeError('tuple must have 2 elements!')  
        for i in range(size[0]):  
            col = []  
            for k in range(size[1]):  
                col.append(random.randint(a, b - 1))  
            result.append(col)  
    else:  
        raise TypeError('invalid type!')  
  
    return result  
  
x = get_random(0, 10, 5)  
print(x)  
  
y = get_random(0, 10, (5, 4))  
print(y)
```

Programın çıktısı şöyledir:

```
[8, 9, 9, 8, 6]  
[[7, 9, 9, 7], [3, 5, 5, 1], [6, 5, 2, 8], [2, 0, 7, 5], [0, 5, 6, 6]]  
get_random fonksiyonu içerisinde raise deyimini kullandık. Bu raise deyimi "exception" oluşturmak için kullanılmaktadır. Exception konusu ilerleyen bölümlerde ele alınmaktadır.
```

Aşağıdaki örnekte add isimli fonksiyon argümanlarıyla aldığı değerlerin toplamına geri dönmektedir. Buradaki add fonksiyonu '*'lı bir parametre almıştır. Ancak bu parametre için girilen argümanlar liste ya da demet de olabilmektedir. Fonksiyon eğer argüman liste ya da demet ise bu liste ya da demetin elemanlarını da toplama işlemine sokmaktadır.

```
def add(*a):  
    total = 0  
    for x in a:  
        if isinstance(x, (list, tuple)):  
            total += sum(x)
```

```

else:
    total += x
return total

x = [20, 30]
result = add(10, x, 40)
print(result)

```

Bazen programcı fonksiyonun dolaşılabilir herhangi bir nesneyle çağrılmış çağrılmadığını da belirlemek isteyebilir. Bu durumda dolaşılabilir nesnelerin ortak özelliklerinden hareketle kontrolün yapılması gereklidir. Bu konu ileride ele alınacaktır.

Modüller ve Modüllerin Import Edilmesi

Python'da .py uzantılı Python kaynak dosyaları aynı zamanda bir modül (module) belirtmektedir. Yani her yazdığımız script dosyası aslında bir modüldür. Bir modül dosyasının içerisinde program kodları da olabilir, sınıflar, fonksiyonlar ve değişkenler de olabilir. Örneğin biz a.py isimli bir Python script dosyasını çalıştıracak olsun. İşte bunun için bizim a.py içerisinde b.py isimli bir Python dosyasındaki fonksiyonları kullanacak olsun. İşte bunun için bizim a.py içerisinde b.py modülünü import etmemiz gereklidir.



Import etme sırasında bazı Python yorumlayıcıları bazı hazırlık işlemlerini de yapmaktadır. Örneğin CPython ve diğer bazı yorumlayıcılar import işlemi sırasında kaynak dosyadaki kodları daha hızlı ele almak için onalrı bir çeşit ara koda dönüştürmektedir.

Import Python'da bir deyim statüsündedir. Genel biçimini şöyledir:

```
import <dosya ismi> [as <isim>] [, <dosya ismi> [as <isim>], <dosya ismi> [as <isim>] , ...]
```

Örneğin:

```
import a
import b, c as d
import a as b, b as c
```

Import işlemi sırasında dosyanın uzantısının (yani .py biçiminde) yazılmadığına dikkat ediniz.

Import edilmiş bir Python modülündeki isimler doğrudan değil modül ismi ve '.' operatörü ile birlikte kullanılır. Örneğin b.py isimli modüldeki foo fonksiyonu biz şöyle çağırırız:

```
b.foo()
```

Örneğin sample.py isimli modülden biz test.py isimli modüldeki global x değişkenini ve foo fonksiyonunu kullanmak isteyelim:

```
#sample.py

import test

def main():
    print(test.x)
    test.foo()

main()
```

```
# test.py

def foo():
    print('I am foo')

x = 10
```

Bir modül import edildiğinde o modülün içerisindeki tüm deyimler çalıştırılır. Yani o模块ün içerisinde fonksiyonlar tanımlamalarından başka deyimler varsa onlar da işletilecektir. Fonksiyonların da aslında Python'da deyim statüsünde olduğunu anımsayınız. Örneğin:

```
# test.py

print('I am test module')

def foo():
    print('I am foo')

x = 10
```

Burada test.py modülü import edildiğinde ekranında 'I am test module' yazısı da görülecektir. Python'da bir modülü import etmek aslında bir Python programını çalıştırmanın diğer bir yoludur.

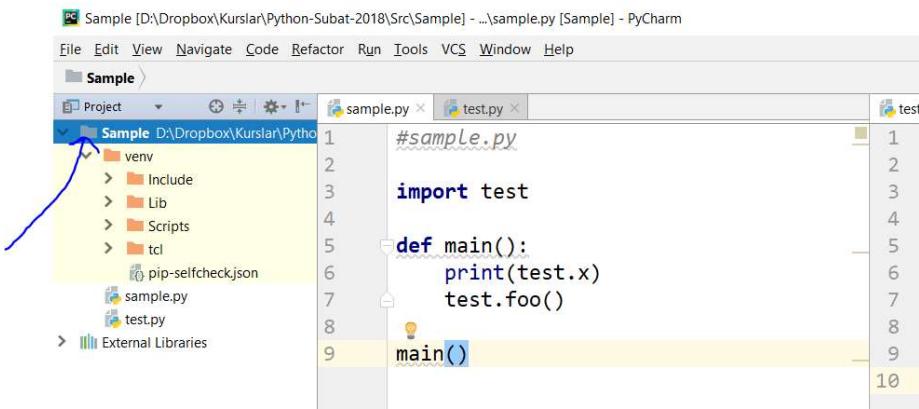
Python'in CPython gerçekleştirmesi ve diğer bazı gerçekleştirmeler bir modül import edildiğinde o modülün içerisindeki Python kodlarını daha kolay işleme sokabilmek için ara kodlara dönüştürmektedir. Böylece ilgili modül binary bir kod olarak daha hızlı yorumlanır. Üstelik bunun belli bir kalıcılığı da vardır. Yani programı tekrar çalıştırduğumda artık bu arakod dönüşümü yapılmayacaktır.

CPython yorumlayıcı sisteminin ürettiği arakodlar .pyc uzantılı bir dosya içeresine yerleştirilmektedir. Python'ın 2.X sürümleri modülün bulunduğu dizinde aynı isimli bir .pyc dosyası oluşturuyordu (örneğin a.py için a.pyc gibi). Fakat Python'ın 3.X sürümleri artık modülün bulunduğu dizinde __pycache__ isimli bir dizin açıp .pyc dosyasını bunun içeresine yerleştirmektedir. Artık başka zaman biz aynı modülü import etsek bile bu arakod dosyası yeniden oluşturulmayacaktır. Peki ya modülde değişiklik yapmışsa? İşte Python sistemi bu .pyc uzantılı arakod dosyasının tarih ve zamanı ile kaynak modül dosyasının tarih ve zamanını karşılaştırır. Eğer kaynak modül dosyasının tarih ve zamanı daha ileri ise bu .pyc dosyasını günceller.

Peki ya import deyimi ilgili modül dosyasını hangi dizinlerde aramaktadır? Biz modül dosyamızı herhangi bir dizine yerleştirsek olur mu? İşte bizim import ettiğimiz dosyaların özel bazı dizinlerde bulunması gereklidir. Bu dizinlere Python'un "path dizinleri" denilmektedir. Python'ın import için sırasıyla sys modülündeki path isimli değişkene bakmaktadır. path değişkeni bir string listesi biçimindedir. Bu listenin her bir elemanı bir dizini belirtir. İşte import edilen dosyalar bu dizinlerde sırasıyla aranmaktadır. Örneğin:

```
>>> import sys
>>> sys.path
['', 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\Lib\\\\idlelib',
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\python36.zip',
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\DLLs',
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\lib',
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32',
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\lib\\\\site-packages']
```

Listenin başındaki boş string (yani '' elemanı) çalışma dizini (current working directory)" anlamına gelmektedir. Yani yukarıdaki örnekte o anda bulunulan çalışma dizinine öncelikle bakılacaktır. Tabii path değişkeninin başında boş string bulunacağıının bir garantisini yoktur. Çalışma dizini eğer yorumlayıcıyı komut satırından çalıştırılmışsa (yani komut satırında python ya da python3 diyerek) o anda bulunduğuımız dizindir. PyCharm IDE'sinde çalışma dizini ana proje dizini olarak ayarlanmaktadır. Örneğin:



IDLE IDE'sinin çalışma dizini onun seçtiği bir yerdedir. Örneğin bu yer kursun yürütüldüğü makinede şöyledir:

```
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36'
```

Biz o andaki çalışma dizininin neresi olduğunu os modülündeki getcwd fonksiyonuyla elde edebiliriz. Örneğin:

```
>>> import os
>>> os.getcwd()
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32'
```

Biz çalışma dizinini yine os modülündeki chdir fonksiyonuyla değiştirebiliriz. Örneğin:

```
>>> import os
>>> os.getcwd()
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32'
>>> os.chdir("D:\\Dropbox\\Kurslar\\IBB-CBS-Python\\Src")
>>> os.getcwd()
'D:\\Dropbox\\Kurslar\\IBB-CBS-Python\\Src'
```

Python yorumlayıcısı çalıştırıldığında sys.path değişkeninde belirli dizinler vardır. Pekiyi bu dizinleri oraya kim yerleştirmektedir? Tabii ki çalıştığımız Python yorumlayıcısı kendine uygun biçimde bazı dizinleri bu sys.path dizinine eklemektedir. Her Python yorumlayıcısının eklediği dizinler diğerlerinden farklı olabilmektedir.

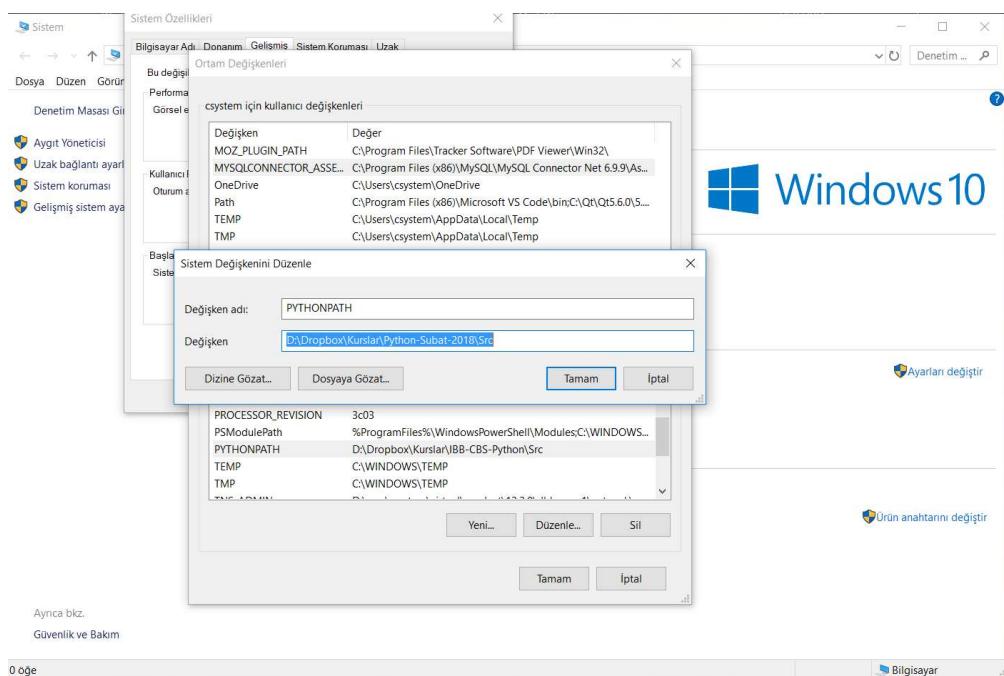
Pekiyi programcı kendi modüllerini nereye yerleştirmelidir? İşte bunun için çalışma dizini iyi bir yer olabilir. Ya da bu amaçla Python'a ait olmayan sys.path içerisindeki bir dizin de kullanılabilir. Ya da biz sys.path değişkeninin belirttiği listeye eklemeler de yapabiliriz. Böylece kendi modüllerimizi kendi dizinlerimize yerleştirebiliriz. Örneğin:

```
>>> sys.path.append('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src')
>>> sys.path
['', 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36\\\\Lib\\\\idlelib',
 'D:\\Dropbox\\Kurslar\\IBB-CBS-Python\\Src',
 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36\\\\python36.zip',
 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36\\\\DLLs',
 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36\\\\lib',
 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36',
 'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36\\\\lib\\\\site-packages',
 'D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src']
```

Burada artık import işlemlerinde 'D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src' dizinine de bakılacaktır.

Pekiyi bu ekleme ne zamana kadar geçerli olacaktır? sys.path değişkeni sys modülündedir. Bu modül her Python çalışmasında yeniden import edilir. Yani kalıcı değildir. Başka bir deyişle bu path listesi biz ilgili komut satırında ya da IDE'de çalışmayı sürdürdüğümüz sürece kalıcı olmakta, çalışmayı bitirdiğimizde kaybolmaktadır. Yani her import işleminde bu değişken eski değerleriyle karşıma gelir. Bu değişkenin kalıcılığını sağlamak için işletim sisteminin çevre değişkenlerini kullanmak gereklidir. PYTHONPATH isimli çevre değişkeni Python çalıştırıldığında yorumlayıcı sistem

tarafından işin başında okunur ve sys.path değişkeninin önüne alınır. Windows'ta gelişmiş ayarlardan bu çevre değişkenini set edebiliriz. Örneğin:



Bir modül import edildiğinde aslında import ismine ilişkin yeni bir değişken yaratılmaktadır. Bu değişken "module" isimli bir sınıf türündendir. Başka bir deyişle bir modülü import ettiğimizde Python yorumlayıcısı "module" türü den bir nesne yaratır. Modül bilgilerini o nesnenin içerisinde yerleştirir. Modül ismine ilişkin değişkene de o nesnenin adresini atar. Örneğin:

```
>>> import sys  
>>> sys  
<module 'sys' (built-in)>  
>>> type(sys)  
<class 'module'>
```



Bir modül üst üste birden fazla kez import edilirse ne olur? Öncelikle bu işlemin programcının hatası olduğunu düşünebilirsiniz. Ancak normal bir çalışmada bu durumla karşılaşılabilir. (Örneğin biz a.py ve b.py modüllerini import etmiş olalım. Bu modüller de kendi içlerinde sys modülünü import etmiş olsun. Toplamda sys modülü birden fazla kez import edilmiş olmaktadır.) İşte bir modül birden fazla kez import edildiğinde ilki dışındaki import işlemlerinde bir şey yapılmamaktadır. Örneğin biz a.py modülünü ilk kez import etmiş olalım. Bu modülün içerisindeki deyimler çalıştırılıp Python yorumlayıcısının parse ağacına eklenecektir. Biz artık bu a.py dosyasında değişiklik yapmış olsak bile ikinci kez aynı dosyayı import ettiğimizde bir şey yapılmayacaktır. (Tabii Python yorumlayıcısını yeniden çalıştırırsak her şey sil baştan yapılır.) Pekiyyi biz gerçekten import yapılanların ikinci kez yeniden import yapılmasını istersek bunu nasıl sağlarız? İşte bunun için bir deyim yoktur, fonksiyon vardır. imp isimli modülün içerisindeki reload isimli fonksiyon yeniden import işlemini yapmaktadır. reload fonksiyonu modül değişkenini parametre olarak alır. Ancak imp modülü Python'ın yeni sürümlerinde artık "deprecated" ilan edilmiştir. Bunun yerine importlib isimli modülün kullanılması tavsiye edilmektedir. importlib modülünün içerisindeki reload metodu aynı işlemi yapmaktadır. Örneğin:

Örneğin:

```
#sample.py

import importlib
import mample

importlib.reload(mample)

# mample.py

print('I am test module')

def foo():
    print('I am foo')

x = 10
```

Bir modül import edilirken import ismi as ile değiştirilebilir. Örneğin:

```
>>> import importlib as il
>>> il.reload(sys)
```

Burada artık bir module sınıfını gösteren değişkeni il biçiminde belirledik. Dolayısıyla o modüldeki isimleri kullanırken artık bu isimle niteliklendirmek zorundayız. Örneğin:

```
>>> import numpy as np, sympy as sp
```

Burada numpy ve sympy isimli iki modül import edilmiştir. Ancak bunlar sırasıyla np ve sp ismiyle kullanılacaktır. as ile isimlendirme genellikle ismi kısaltmak için ya da daha okunabilir hale getirmek için tercih edilmektedir.

Anımsanacağı gibi bir modül import edildiğinde onun içerisindeki değişkenler artık modülün ismi belirtilerek '.' operatörüyle kullanılıyordu. Bu durum bazen biraz sıkıcı olabilmektedir. İşte bunun için from import deyimi kullanılmaktadır. from import deyiminin genel biçimleri şöyledir:

```
from <modül ismi> import [<değişken> [as <isim>], [<değişken> [as isim], ...] []]
from <modül ismi> import *
```

Bu deyimle biz belli bir modüldeki belli değişkenleri hiç modül ismi belirtmeden kullanabiliriz. Örneğin:

```
#sample.py

from test import foo, tar

foo()
tar()

# test.py

print('I am test module')

def foo():
    print('I am foo')

def bar():
    print('I am bar')

def tar():
    print('I am tar')
```

```
x = 10
```

Programın çıktısı şöyle olacaktır:

```
I am test module  
I am foo  
I am tar
```

Burada biz test modülündeki foo ve tar metodlarını hiç test ismi ile niteliklendirmeden doğrudan kullanabildik. Ancak bar metodu bu biçimde import edilmediği için onu dışarıdan kullanamayız.

Biz from import deyimi ile modüldeki bazı değişkenleri niteliklendirmeden kullanmak istediğimizde modüldeki tüm kodlar yine çalıştırılmaktadır. (Tabii from import deyimleri aynı modül için birden fazla kez kullanıldığında modüldeki deyimler yalnızca bu işlem ilk kez yapıldığında çalıştırılacaktır.) Ancak from import deyiminde bir modül değişkeni oluşturulmaz. Yani biz yukarıdaki örnekte test modülü içerisindeki bar fonksiyonunu test.bar() biçiminde çağrıramayız. Eğer bunu yapmak istiyorsak modülü ayrıca import etmemeliyiz. Bu durumda:

```
from x import y
```

İşlemi ile:

```
import x  
y = x.y  
del x
```

İşlemleri işlevsel olarak eşdeğerdir.

Bir modüldeki tüm değişkenlerin aktarımı değişken ismi yerine '*' karakteri getirilerek sağlanabilir. Örneğin:

```
# sample.py  
  
from test import *  
  
foo()  
bar()  
tar()  
print(x)  
  
# test.py  
  
print('I am test module')  
  
def foo():  
    print('I am foo')  
  
def bar():  
    print('I am bar')  
  
def tar():  
    print('I am tar')  
  
x = 10
```

From import deyiminde as ile değişkenlere yeni isimler de verilebilmektedir. Örneğin:

```
# sample.py  
  
from test import foo as f, bar as b, tar as t
```

```
f()  
b()  
t()
```

CPython gerçekleştiriminde bir modül içerisinde yalnızca belirli fonksiyonların `from import` işlemi ile `import` edilmesi tüm modülün arakoda dönüştürülmesini engellemektedir. CPython'da bu durumda yalnızca kullanılan fonksiyonlar modülden alınmaktadır.

`from import` işlemi sırasında isim çakışmaları oluşabilir. `from import` deyimi uygulandığında eğer isim çakışması oluşursa artık yeni `import` edilen isimler kullanılır. Bu işlemin aynı değişkene yeni bir değer atama işleminden bir farkı yoktur. Örneğin:

```
# sample.py  
  
def bar():  
    print('sample bar')  
  
bar()  
  
from test import *  
  
foo()  
bar()  
  
def foo():  
    print('sample foo')  
  
foo()  
  
# test.py  
  
def foo():  
    print('test foo')  
  
def bar():  
    print('test bar')
```

Ekran çıktısı şöyle olacaktır:

```
sample bar  
test foo  
test bar  
sample foo
```

Bir Modülü Import Etmekle Çalıştırmak Arasındaki Fark

Bir Python modülü komut satırından ya da IDE'den çalıştırılabilceği gibi `import` işlemeyle de çalıştırılabilmektedir. İşte bizim bazen bir modülün nasıl çalıştırıldığını bilmemiz gerekebilir. Çünkü biz doğrudan yorumlayıcı ile çalıştırımda bir şey `import` ile çalıştırımda başka bir şey yapmak isteyebiliriz. Bir modülün doğrudan yorumlayıcı ile mi çalıştırıldığı yoksa `import` deyimiyle mi çalıştırıldığı `built-in __name__` isimli değişkene bakılarak belirlenir. Eğer bu değişken modülün ismini veriyorsa bu durumda ilgili modül `import` ile çalıştırılmıştır. Yok eğer bu değişken "`__main__`" ismini veriyorsa bu durumda modül doğrudan yorumlayıcı tarafından çalıştırılmıştır. `__name__` değişkeni her zaman `str` türündendir. Örneğin `test.py` dosyasının içeriği şöyle olsun:

```
# test.py  
  
print(__name__)
```

Şimdi biz bu modülü python yorumlayıcısı ile çalışıralım. Sonuç şöyle olacaktır:

```
__main__
```

Şimdi de biz bu modülü sample.py tarafından import ederek çalışıralım:

```
#sample.py
```

```
import test
```

Ekranda şunları göreceğiz:

```
test
```

Bu durumda biz aşağıdaki gibi bir test işlemi ile modülün nasıl çalıştırıldığını anlayabiliriz:

```
# test.py
```

```
if __name__ == '__main__':
    print('yorumlayıcı yoluyla çalıştırılmış')
else:
    print('import ile çalıştırılmış')
```

Programcılar bazen bir modülde hem bazı işlemler yapıp ekrana bir şeyler yazdırıyor olabilirler hem de o modülün dışarıdan import edilmesini sağlayabilirler. Tabii dışarıdan import edilme işleminde artık normal çalışmada yapılan işlemlerin yapılmaması uygun olur. Örneğin:

```
# test.py

def foo():
    print('foo')

def bar():
    print('bar')

def tar():
    print('tar')

def main():
    print('program olarak çalışma başladı')
    foo()
    bar()
    tar()

if __name__ == '__main__':
    main()
```

Burada test.py hem bağımsız çalışan bir program gibidir hem de import edilerek kullanılabilcek bir kütüphane gibidir. Tabii eğer bu modül import edilerek kullanılıyorsa artık program gibi çalıştırılmada yapılacak işlemler yapılmayacaktır.

Python'da Rastgele Sayıların Elde Edilmesi

Python'da rasgele sayı üretmek için random isimli modül kullanılmaktadır. Modüldeki randint isimli fonksiyon belli bir aralıkta rastgele bir tamsayı geri döndürür. Fonksiyonun parametrik yapısı şöyledir:

```
random.randint(a, b)
```

Fonksiyon iki parametreye sahiptir. [a, b] aralığında rastgele bir tamsayı geri döndürür. Örneğin:

```
>>> random.randint(10, 20)
10
```

Örneğin:

```
for i in range(10):
    val = random.randint(0, 99)
    print(val, end = ' ')
```

Rassal sayaç rastgele bir tohum degerden başlamaktadır. Dolayısıyla her çalışmada rastgele bir dizilim elde edilir.

Sınıf Çalışması: n kere yazı tura atarak yazı ve tura oranını belirlemeye çalışınız.

Çözüm:

```
import random

def coin(n):
    head = tail = 0
    for i in range(n):
        x = random.randint(0, 1)
        if x == 0:
            tail += 1
        else:
            head += 1

    return head / n, tail / n
```

```
head, tail = coin(1000000)
print(f'tura: {head}, yazı: {tail}')
```

Sınıf Çalışması: getrand isimli bir fonksiyon yazınız. Bu fonksiyon a, b ve count isimli üç parametreye sahip olsun. İlk iki parametre rastgele üretilecek olan tamsayılar için kapalı aralık belirtmektedir. Üçüncü parametre üretilecek rassal sayıların sayısını belirtir. Fonksiyon tamsayılardan oluşan bir liste ile geri dönmelidir. Fonksiyonu randint fonksiyonunu kullanarak yazınız ve çağırınız.

Çözüm:

```
import random

def getrand(a, b, count):
    l = []
    for i in range(count):
        val = random.randint(a, b)
        l.append(val)
    return l

a = getrand(0, 99, 5)
print(a)
```

random modülünün randrange isimli fonksiyonu bize belli bir aralıkta rastgele sayı verir. Aralık range fonksiyonundaki gibidir. Parametrik yapı şöyledir:

```
random.randrange(start, [stop , step])
```

Örneğin:

```
>>> random.randrange(100)
53
>>> random.randrange(10, 20)
13
```

Modülün choice isimli fonksiyonu bizden indeksli dolaşılabilir (sequence container) bir nesne alır. O nesne içerisindeki rastgele bir değerle geri döner. Listeler, string'ler demetler "indeksli dolaşılabilir" nesnelerdir. Ancak kümeler ve sözlükler indeksli dolaşılabilecek nesneler değildir. Örneğin:

```
import random

names = ['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
for i in range(3):
    val = random.choice(names)
    print(val)
```

Örneğin:

```
>>> random.choice('istanbul')
'a'
```

Örneğin:

```
>>> for i in range(10): print(random.choice('istanbul'), end=' ')
a s a i i a i n l a
```

random modülüne Python 3.6 versiyonuyla birlikte choices isimli bir fonksiyon da eklenmiştir. Bu fonksiyon dört parametreye sahiptir.

```
random.choices(population, weights = None, *, cum_weights = None, k = 1)
```

Fonksiyonun birinci parametresi indeksli dolaşılabilir bir nesne almaktadır. Fonksiyonun diğer parametrelerinin default değer aldığı görüyorsunuz. Buradaki weights ve cum_weights dolaşılabilir nesnedeki elemanların seçilme olasılıklarını belirtmektedir. Son parametre olan k elde edilecek listenin kaç eleman içereceğini belirtmektedir. Örneğin biz 5 elemanlı bir listeden rastgele 3 elemanı şöyle elde edebiliriz:

```
import random

a = ['ali', 'veli', 'selami', 'ayşe', 'fatma']

val = random.choices(a, k=3)
print(val)
```

Burada fonksiyonun geri döndürdüğü listenin iadelî seçim ile elde edildiğine dikkat ediniz.

Modülün shuffle isimli fonksiyonu bizden indeksli dolaşılabilir (sequence container) bir nesne alır onu karıştırır. Örneğin:

```
import random

names = ['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']
random.shuffle(names)
print(names)
```

Fonksiyonun nesnenin içini değiştirdiğine dikkat ediniz.

Pekiyi bu karıştırma işlemi nasıl yapılmaktadır? İşte karıştırma (shuffling) işlemi için değişik algoritmalar kullanılabilmektedir. Bunlardan en basit ve tercih edileni Knuth, Fisher, Yates algoritmasıdır. Bu algoritmada dizimin elemanları sırasıyla dizilimin rastgele bir elemanıyla yer değiştirilir. Bu işlem Python'da şöyle yapılabilir:

```
import random

def myshuffle(sequence):
    for i in range(len(sequence)):
        k = random.randrange(len(sequence))
        sequence[i], sequence[k] = sequence[k], sequence[i]

a = list(range(10))
print(a)
myshuffle(a)
print(a)
```

Sınıf Çalışması: randint fonksiyonunu kullanarak Sayısal Lotoda bir kolon veren ([1-49] arasında 6 sayı) get_column fonksiyonunu yazınız. Fonksiyon parametre almamaktadır. Geri dönüş değeri ise 6 elemanlı bir listedir. Elde edilen 6 sayı içerisinde aynı elemanlardan olmaması gerekmektedir.

Çözüm:

```
import random

def get_column():
    result = []
    for i in range(6):
        while True:
            val = random.randint(1, 49)
            if val not in result:
                break
        result.append(val)
    return result

col = get_column()
for i in col:
    print(i, end = ' ')
```

Soru set kullanılarak daha Python'ca (Pythonic) da yapılabilirdi:

```
import random

def get_column():
    result = set()
    while len(result) != 6:
        val = random.randint(1, 49)
        result.add(val)

    return list(result)

col = get_column()
for i in col:
    print(i, end = ' ')
```

Aşağıdaki çözüm pek etkin olmamakla birlikte alıştırma amacıyla uygulanabilir:

```
import random

def get_column():
    result = []
    numbers = list(range(1, 50))
```

```

for i in range(6):
    x = random.choice(numbers)
    result.append(x)
    numbers.remove(x)

return result

column = get_column()
print(column)

```

Aslında random modülüne ileri versyonlarda sample isimli bir fonksiyon da eklenmiştir. Bu sample fonksiyonu bir anakütle içerisinde rastgele örneklem elde etmek için düşünülmüştür. Yani choices fonksiyonunun iadesiz versiyonu gibi çalışmaktadır. sample fonksiyonun parametrik yapısı şöyledir:

```
random.sample(population, k)
```

Fonksiyon dolaşılabilir bir nesneyi ve örneklem sayısını parametre olarak almaktadır ve k elemanlı bir listeye geri dönmektedir. Örneğin:

```

>>> import random
>>> names = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
>>> random.sample(names, 3)
['ayşe', 'selami', 'veli']

```

Bu durumda yukarıdaki sorunun çözümü aslında sample fonksiyonuyla şöyle yapılabilir:

```

import random

b = random.sample(range(1, 50), 6)
print(b)

```

Sınıf Çalışması: 52 kartı temsil eden string'lerden oluşan bir listeyi veren create_deck isimli fonksiyonu yazınız. Bu fonksiyonun parametresi yoktur. Geri dönüş değeri Sinek-2', 'Maça-2', Karo-2', Kupa-2', ..., 'Sinek-As', 'Maça-As', 'Karo-As', 'Kupa-As' biçiminde yazılırdan oluşan bir listedir. Tabii fonksiyonda listeyi tüm kartları el ile yazarak oluşturmayınız.

Çözüm:

```

import random

def create_deck():
    types = ('Sinek', 'Maça', 'Karo', 'Kupa')
    numbers = ('2', '3', '4', '5', '6', '7', '8', '9', '10', 'Vale', 'Kız', 'Papaz', 'As')

    deck = []
    for number in numbers:
        for type in types:
            deck.append(type + '-' + number)

    return deck

deck = create_deck()
print(deck)
random.shuffle(deck)
print(deck)

```

Sınıf Çalışması: Yukarıda create_deck fonksiyonuyla yaratılan desteyi dağıtan distribute_deck isimli fonksiyonu yazınız. Bu fonksiyon 52 elemanlı kart listesini parametre olarak alacak, geri dönüş değeri olarak 4 elemanlı bir demet verecektir. Demetin her elemanı 13 karttan oluşan bir liste olmalıdır.

Çözüm:

```
import random

def create_deck():
    types = ('Sinek', 'Maça', 'Karo', 'Kupa')
    numbers = ('2', '3', '4', '5', '6', '7', '8', '9', '10', 'Vale', 'Kız', 'Papaz', 'As')

    deck = []
    for number in numbers:
        for type in types:
            deck.append(type + '-' + number)

    return deck

def distribute_deck(deck):
    cards = ([], [], [], [])

    for i in range(len(deck)):
        cards[i % 4].append(deck[i])

    return cards

deck = create_deck()
random.shuffle(deck)
print(deck)

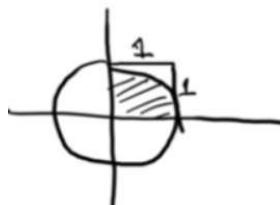
north, east, south, west = distribute_deck(deck)
print(north)
print(east)
print(south)
print(west)
```

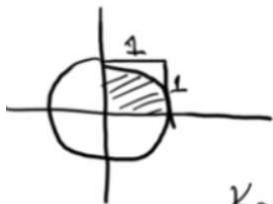
random isimli fonksiyon [0, 1] aralığında rastgele bir float sayı üretmektedir. Örneğin:

```
>>> random.random()
0.565082075313238
>>> random.random()
0.2841654499081152
```

Sınıf Çalışması: Pi sayısını rassal sayı üreterek aşağıdaki gibi hesaplayan programı yazınız:

Birim çemberin $1/4$ 'ünü ele alalım:





Karenin alanı = 1

Daire dilininin alanı = $\frac{\pi}{4}$

Toplam n roşiyede nette elde edelim

$$\frac{n}{K(\text{dilin içeriindeki})} = \frac{1}{\frac{\pi}{4}}$$

$$\frac{n\pi}{4} = k$$

$$\pi = \frac{4k}{n}$$

Çözüm:

```
import random
import math

def getpi(n):
    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if math.sqrt(x ** 2 + y ** 2) < 1:
            count += 1

    return 4 * count / n

while True:
    n = int(input('Bir sayı giriniz:'))
    if n == 0:
        break
    pi = getpi(n)
    print(pi)
```

enumerate Fonksiyonu

enumerate isimli built-in fonksiyon bizden bir indeksli dolaşılabilir nesne alır ve bize bir dolaşım nesnesi verir. Fonksiyonun bize verdiği dolaşım nesnesi dolaşıldığında iki elemanlı demetler elde elde edilmektedir. Bu demetlerin birinci elemanları indeks değerlerinden ikinci elemanları ise bizim verdiği dolaşılabilir nesnesdeki o indekste bulunan değerlerden oluşmaktadır. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> e = enumerate(a)
>>> type(e)
<class 'enumerate'>
>>> list(e)
[(0, 10), (1, 20), (2, 30), (3, 40), (4, 50)]
```

Örneğin:

```
>>> d = {'ali': 10, 'veli': 20, 'selami': 30, 'ayşe': 40, 'fatma': 50}
>>> list(enumerate(d))
[(0, 'ali'), (1, 'veli'), (2, 'selami'), (3, 'ayşe'), (4, 'fatma')]
```

Örneğin:

```
>>> s = {'ali': 123, 12.5, 'veli', True}
>>> list(enumerate(s))
[(0, True), (1, 'ali'), (2, 12.5), (3, 'veli'), (4, 123)]
```

Örneğin:

```
a = [12, 34, 2, 78, 4, 30, -9, 44, 88, 41]
```

```
for index, val in enumerate(a):
    print(index, val)
```

Örneğin int bir listenin tek olan elemanlarına 1 toplamak isteyelim:

```
a = [10, 13, 8, 22, 64, 73]
for index, val in enumerate(a):
    if val % 2 == 1:
        a[index] += 1
print(a)
```

Sınıf Çalışması: Bir listenin en büyük elemanın listenin kaçinci indeksinde olduğunu bulan programı yazınız.

Çözüm: Klasik çözüm şöyledir:

```
a = [23, 56, 12, 34, 8, 21, 16, 43]
index = 0
for i in range(1, len(a)):
    if a[i] > a[index]:
        index = i
print(f'En büyük eleman: {a[index]}, indeksi: {index}')
```

Aynı işlemi enumerate fonksiyonunu kullanarak da şöyle yapabilirdik:

```
a = [12, 34, 2, 78, 4, 30, -9, 44, 88, 41, 20]
mi = 0
for i, x in enumerate(a):
    if x > a[mi]:
        mi = i
print(f'En büyük eleman: {a[mi]}, indeksi: {mi}')
```

enumerate fonksiyonu iki parametrelidir. Fonksiyonun bu ikinci parametresi index değerinin kaçtan başlatılacağını belirtir. Örneğin:

```
a = [12, 34, 2, 78, 4, 30, -9, 44, 88, 41]
for index, val in enumerate(a, 10):
    print(index, val)
```

Burada index değeri 10'dan başlatılmıştır.

Değişkenlerin Faaliyet Alanları (Scope)

Bir değişkenin kullanılıldığı program aralığına faaliyet alanı (scope) denilmektedir. Python'ın da kendine özgü bir faaliyet alanı kuralı vardır. Maddeler halinde bu kuralları açıklayalım:

1) Bir değişken sınıfların ve fonksiyonların dışında oluşturulmuşsa bunlara global değişken denilmektedir. Global değişkenler yalnızca isimleriyle her yerden kullanılabilirler. Örneğin:

```
x = 10

def foo():
    print(x)

def bar():
    print(x)

foo()
x = 20
bar()
print(x)
```

Burada x global bir değişkendir. Bu nedenle her fonksiyonun içerisinde ya da dışında herhangi bir yerde kullanılabilir.

2) Eğer bir değişkene bir fonksiyonun ya da metodun içerisinde değer atanmışsa bu değişkene faaliyet alanı bakımdan yerel değişken denir. Yerel değişkenler yalnızca yaratılmış oldukları fonksiyon ya da metodun içerisinde kullanılabilirler. Örneğin:

```
x = 10

def foo():
    y = 20
    print(y)

print(x)
print(y)      # error!
```

Farklı fonksiyonlardaki aynı isimli değişkenler aslında farklı değişkenlerdir. Örneğin aşağıdaki foo ve bar fonksiyonlarındaki x değişkeni farklı x'lerdir. foo ve bar kendi içerisinde kendi x değişkenlerini kullanıyor durumdadırlar.

```
def foo():
    x = 10
    print(x)

def bar():
    x = 20
    print(x)

foo()
bar()
```

Bir fonksiyon ya da metot içerisinde global bir değişkenle aynı isimli bir değişkene atama yapıldığında bu atama global değişkene yapılan atama anlamına gelmemektedir. Aynı isimli yerel bir değişken yaratılıp atama ona yapılmaktadır. Örneğin:

```
x = 100
def foo():
    x = 200
    print(x)      # yerel 200 çıkar
```

```
foo()  
print(x)          # global 100 çıkar
```

Burada foo içerisindeki x artık global x değildir. foo'da yeni yaratılmış aynı isimli yerel bir x'tir. Artık foo'da bu x isminini kullandığımızda yerel olan x anlaşıılır.

Ayrıca Python'da bir fonksiyon ya da metot içerisinde önce bir global değişkeni kullanıp daha sonra bu değişkene atama yapamayız (yani global değişken fonksiyon içerisinde kullanıldıktan sonra artık aynı isimli bir yerel değişken yaratılamamaktadır). Örneğin:

. Örneğin:

```
x = 100  
  
def foo():  
    print(x)      # error  
    x = 200  
    print(x)  
  
foo()  
print(x)
```

Tabii fonksiyon ya da metot içerisinde biz gerçekten global olan değişkene atama yapmak isteyebiliriz. Bu durumda global bildirimi yapılmak zorundadır. Örneğin:

```
x = 10  
  
def foo():  
    global x  
    x = 20  
    print(x)    # 20  
  
foo()  
print(x)      # 20
```

global bildiriminin (Python referans kitaplarına göre bu da bir deyimdir) genel biçimi şöyledir:

```
global <değişken listesi>
```

Değişken listesi ',' ile ayrılmış değişken isimlerinden oluşturulabilir. Örneğin:

```
global x  
global y, z, k
```

Fonksiyonda global bildiriminden önce bu bildirimde belirtilen isimlerle aynı isimlere sahip değişkenler kullanılamaz. Örneğin:

```
x = 10  
  
def foo():  
    print(x)  
    global x  # error  
    x = 20  
  
foo()
```

Global deyimi Python'ın referans kitaplarında her ne kadar bir deyim biçiminde ele alınıyorsa da programın çalışma zamanı sırasında işletilmemektedir. Bu deyim Python yorumlayıcısı için bir direktif görevini yerine getirir. Yani Python yorumlayıcısı global direktifini gördüğünde artık bu direktiften sonraki kodda ilgili değişkenlerin global

olduğunu kabul eder. Dolayısıyla global deyiminde belirtilen isimlerin global düzeyde o noktada bir tanımlamalarının yapılmış olması gerekmez. Örneğin aşağıdaki program geçerlidir.

```
def foo():
    global x
    x = 10
    print(x)

foo()
print(x)
```

Ancak global deyiminden sonra bu deyimdeki global değişkenler henüz yaratılmadan kullanılırsa bu durum çalışma zamanı sırasında exception'a yol açmaktadır. Örneğin:

```
def foo():
    global x
    print(x)          # exception!

foo()
print(x)
```

3) Python'da C, C++, C# ve Java gibi dillerdeki yerel blok faaliyet alanı yoktur. Fonksiyon içerisindeki değişkenler yaratıldıkten sonra o fonksiyonun her yerinde kullanılabilirler. for döngülerindeki değişkenler de benzer biçimde döngü dışında kullanılabilirlerdir. Örneğin:

```
def foo():
    x = [1, 2, 3, 4, 5]
    for i in x:
        print(i)
    print(i)          # geçerli

foo()
```

Buradaki döngü değişkeni olan i fonksiyonun yerel bir değişkenidir. Dolayısıyla yaratıldıktan sonra her yerde kullanılabilir. Aşağıda C, C++ C# ve Java dillerinde oluşturulmuş bir while döngüsü görüyorsunuz:

```
while (i < 10) {
    int x;
    //...
}
```

Bu dillerde bloklar ayrı bir faaliyet alanı belirtmektedir. Dolayısıyla while bloğu içerisindeki x değişkeni yalnızca bu while bloğluğunun içerisinde kullanılabilir. Halbuki Python'da deyimlerin girintili blokları ayrı bir faaliyet alanı belirtmemektedir. Örneğin:

```
def foo():
    i = 0
    while i < 10:
        x = 10
        i += 1
    print(x)          # geçerli

foo()
```

Peki bir yerel değişken fonksiyon içerisinde bir koşul altında yaratılmışsa ne olacaktır? İşte bu tür durumlarda eğer programın akışı henüz yaratılmamış bir değişkenin kullanıldığı noktaya geldiğinde çalışma zamanı sırasında exception oluşmaktadır. Örneğin:

```

def foo(x):
    if x > 0:
        y = 10
    print(y)

foo(10)      # geçerli, sorun yok
foo(-10)     # exception oluşur!

```

Tabii bu durum yerel değişkenlere özgü değildir. Aslında her türlü değişkenin kullanımında o değişken akışın o noktasında yaratılmamışsa exception oluşmaktadır. Örneğin:

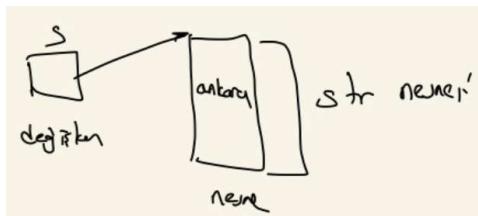
```

print('begins... ')
x = 10
print(x)      # sorun yok!
print(y)      # exception oluşur!

```

Python'da Değişkenlerin Ömürleri

Python'da değişken ile nesne arasında farklılık vardır. Anımsanacağı gibi Python'da tüm değişkenler aslında nesnelerin adreslerini tutmaktadır. Örneğin:



Değişkenlerin bellekte yer kapladığı zaman aralığına ömür denilmektedir. Değişkenler sürekli bellekte yer kaplamazlar. Programın çalışmasının belli bir aşamasında yaratılıp belli bir aşamasında yok edilirler. Python'da bir global değişken akış o değişkene ilk kez değer atandığı noktada yaratılır. Program sonlanana kadar bellekte kalır. Yerel değişkenler ise fonksiyon ya da metot çağrılmış onlara ilk kez değer atandığında yaratılırlar akış fonksiyondan ya da metottan çıktığında otomatik olarak yok edilirler. Yani yerel değişkenler sürekli bellekte kalmazlar. Fonksiyon ya da bittiğinde onlar bellekten boşaltılırlar. Parametre değişkenleri de Python'da fonksiyon ya da metot çağrıldığında yaratılırlar. Fonksiyon ya da metot sonlandığında yok edilirler.

Python'da nesneler onları hiçbir değişken göstermediği durumda "çöp toplayıcı (garbage collector)" denilen mekanizma tarafından otomatik yok edilmektedir. Çöp toplayıcı mekanizma ileride ele alınmaktadır.

map fonksiyonu

map fonksiyonu bir fonksiyonu ve dolaşılabilir bir nesneyi parametre olarak alır. Dolaşılabilir nesnedeki her elemanı bu fonksiyona parametre olarak geçirir, onun geri dönüş değerlerinden bir dolaşım nesnesi oluşturarak o nesneye geri döner. Parametrik yapısı şöyledir:

```
map(function, iterable, *args)
```

Örneğin:

```

a = [1, 2, 3, 4, 5]
def foo(n):
    return n * n

b = map(foo, a)
print(list(b))

```

Örneğin:

```
import math

a = [1, 2, 3, 4, 5]
b = list(map(math.sqrt, a))
print(b)
```

Aşağıdaki örnekte bir string listesindeki en uzun yazının uzunluğu ekrana yazdırılmaktadır:

```
names = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
print(max(map(len, names)))
```

Sınıf Çalışması: İki boyutlu bir listenin en uzun olan elemanın uzunluğunu map fonksiyonu kullanarak elde ediniz.

Çözüm:

```
a = [
    [1, 2, 3, 4, 5],
    [5, 6],
    [7, 8, 9],
    [10]
]

print(max(map(len, a)))
```

Aslında map fonksiyonu birden fazla dolaşılabilir nesneyi parametre olarak alabilmektedir. Bu durumda çağrılmakacak fonkiyonun parametre sayısının da map fonksiyonuna girilen dolaşılabilir nesnelerin sayısı kadar olması gereklidir. Dolaşım işlemi ilk dolaşılabilir nesne bittiğinde sonlandırılmaktadır. Örneğin:

```
a = [1, 2, 3, 4, 5]
b = [10, 20, 30]
c = [5, 15, 15, 20, 25, 30]

def foo(x, y, z):
    return x + y + z

for x in map(foo, a, b, c):
    print(x, end=' ')
print()
```

Kod çalıştırıldığında şu çıktı elde edilecektir:

```
16 37 48
```

Burada en kısa listenin üç eleman uzunlığında olduğuna dikkat ediniz.

Python'a içemler eklendikten sonra map fonksiyonunun kullanımı içemler lehine azalmıştır. İçemler konusu ilerideki bölgümlerde ele alınmaktadır.

İç İçe Fonksiyon Tanımlamaları

Pyton'da bir fonksiyon içerisinde başka bir fonksiyon tanımlanabilmektedir. Bu durumda iç fonksiyon yalnızca tanımlandığı fonksiyon içerisinde çağrılabılır. Örneğin:

```
def foo():
    print('foo begins')
    def bar():
        print('bar begins')
        print('bar ends')
```

```
bar()
print('foo ends')

foo()
```

Bir fonksiyon içerisinde global bir fonksiyonla aynı isimli bir fonksiyon tanımlayabiliriz. Fonksiyon isimleri birer değişken belirttiğine göre ve fonksiyon tanımlamaları da bu değişkenlere atama yapma anlamına geldiğine göre bu durumun bir sakıncası yoktur. Örneğin:

```
def bar():
    print('Global bar')

def foo():
    def bar():
        print('Nested bar')
    bar()

foo()
bar()
```

Ancak bir dış fonksiyon iç fonksiyonla aynı isimli bir başka global fonksiyonu daha önce çağrıp daha sonra tanımlayamaz.

```
def bar():
    print('Global bar')

def foo():
    bar()          #error
    def bar():
        print('Nested bar')
    bar()

foo()
```

Daha önceden de belirttiğimiz gibi aslında bir fonksiyon tanımlaması o fonksiyonun ismi olan değişkene bir fonksiyon nesnesinin adresini atamak anlamına gelmektedir. Bu bağlamda yukarıdaki örneğin aşağıdakinden bir farkı yoktur:

```
bar = 10
def foo():
    print(bar)      # error!
    bar = 20
    print(bar)

foo()
```

Bir fonksiyon önce global fonksiyonu çağrıp daha sonra bu global fonksiyonla aynı isimli bir fonksiyonu tanımlayacaksa global bildiriminin yapılması gereklidir:

```
def bar():
    print('Global bar')

def foo():
    global bar
    bar()
    def bar():
        print('Nested bar')
    bar()

foo()
bar()
```

Burada artık foo içerisinde bar değişkeni global olan bar değişkenidir. dolayısıyla biz foo içerisinde bar fonksiyonunu tanımladığımızda aslında global bar değişkenine atama yapmış gibi oluruz. Programın çıktısı şöyle olacaktır:

```
Global bar
Nested bar
Nested bar
```

İç içe fonksiyon tanımlamalarında içteki fonksiyon dışındaki fonksiyonun yerel değişkenlerini ve tabii global değişkeleri kullanabilmektedir. Örneğin:

```
def foo():
    val = 10
    print('foo begins')
    def bar():
        print('bar begins')
        print('val = {}'.format(val))
        print('bar ends')
    bar()
    print('foo ends')

foo()
```

İç bir fonksiyon içerisinde dış fonksiyondaki yerel değişkenlerle ya da global değişkenlerle aynı isimli yeni yerel değişkenler oluşturulabilir. Örneğin:

```
def foo():
    x = 10
    def bar():
        x = 20      # yeni bir x
        print(x)
    bar()
    print(x)

foo()
```

Burada ekran çıktısı şöyle olacaktır:

```
20
10
```

Yine iç fonksiyonda önce dış fonksiyonun yerel değişkenleri kullanılıp sonra aynı isimli yeni yerel değişkenler yaratılmazlar. Örneğin:

```
def foo():
    x = 10
    def bar():
        print(x)      # error
        x = 20
        print(x)
    bar()
    print(x)

foo()
```

Böylesi bir durumda eğer iç fonksiyon dış fonksiyondaki değişkeni kullanmak isterse (global bildirimine benzer biçimde) nonlocal bildirimi uygulanmalıdır. nonlocal bildiriminin genel biçimi şöyledir:

```
nonlocal <değişken listesi>
```

Örneğin:

```

def foo():
    x = 10
    def bar():
        nonlocal x
        print(x)      # foo'daki x
        x = 20        # foo'daki x
        print(x)      # foo'daki x
    bar()
    print(x)

foo()

```

nonlocal bildiriminde global bir değişken belirtilemez. Ancak üst fonksiyondaki bir değişken belirtilebilir. Örneğin:

```

x = 10
def foo():
    def bar():
        nonlocal x      #error!
        print(x)
        x = 20
        print(x)
    bar()

foo()

```

nonlocal bildirimi yukarıda doğru arama yapıp bildirimdeki isimle eşleşen ilk değişkeni bulmaktadır. Örneğin:

```

def foo():
    x = 10
    def bar():
        def tar():
            nonlocal x      # foo'daki x
            x = 20
        tar()
    bar()
    print(x)          # 20

foo()

```

Örneğin:

```

def foo():
    x = 10
    def bar():
        nonlocal x      # foo'daki x
        x = 20
        def tar():
            nonlocal x      # bar'daki x
            x = 30
        tar()
    bar()
    print(x)          # 30

foo()

```

nonlocal bildirimindeki değişken üst fonksiyondaki global bildirilmiş değişken olamaz. Örneğin:

```

x = 10
def foo():
    global x

```

```

x = 20
def bar():
    nonlocal x      # error, x üst fonksiyonda global
    x = 30
bar()

foo()
print(x)

```

Eğer üst ve alt fonksiyonların her ikisi de aynı global değişkeni kullanacaksa iç fonksiyonda da yine global bildiriminin yapılması gereklidir. Örneğin:

```

x = 10
def foo():
    global x
    x = 20
    def bar():
        global x
        x = 30
    bar()

foo()
print(x)

```

Pekiyi bir fonksiyonu dışarda tanımlamak yerine başka bir fonksiyonun içerisinde tanımlamanın anlamı ne olabilir? İşte eğer bir fonksiyon genel olmaktan ziyade yalnızca başka bir fonksiyonun yazılmasında bir araç olarak kullanılacaksa o fonksiyonu asıl fonksiyonun içerisinde tanımlamak daha iyi bir soyutlamaya olanak sağlar. Çünkü bir fonksiyon dışında tanımlandığında -herkes onu kullanabileceğine göre- genel bir fonksiyon olduğu izlenimi yaratılmaktadır. Örneğin:

Örneğin:

```

def write_primes(val):
    def is_prime(val):
        if val % 2 == 0:
            return val == 2
        for i in range(3, val, 2):
            if val % i == 0:
                return False
        return True

    for i in range(2, val + 1):
        if is_prime(i):
            print(i, end=' ')

```

write_primes(100)

Burada isPrime fonksiyonu her yerden kullanılabilen bir fonksiyon değildir. Yalnızca writePrimes fonksiyonundan kullanılabilisin diye iç fonksiyon olarak yazılmıştır.

Programlama dillerinde genel olarak yerel değişkenler ilgili fonksiyondan çıktılığında yok edilmektedir. Yani yerel değişkenler içinde tanımlandığı fonksiyon çalıştığı sürece yaşarlar. Halbuki Python'da iç fonksiyon dış fonksiyonun yerel değişkenlerini kullanıyorsa bu iç fonksiyon tarafından kullanılan yerel değişkenler eğer bu iç fonksiyon dışarıya verilmişse (yani çöp toplayıcı tarafından yok edilmemişse) yaşamaya devam etmektedir. Örneğin:

```

def foo():
    x = 10
    def bar():
        nonlocal x
        print(x)

```

```
x += 1
return bar

f = foo()
f()
f()
f()
```

Burada foo fonksiyonu bar fonksiyonuyla (bar fonksiyonun adresiyle) geri dönmektedir. Ancak bar fonksiyonu foo fonksiyonunun yerel x değişkenini de kullanmaktadır. İşte bu nedenden dolayı foo fonksiyonun çalışması bittiği halde x yerel değişkeni yaşamaya devam eder. Şimdi aşağıdaki örneği inceleyelim:

```
def foo(a):
    x = a
    def bar():
        print(x)
    return bar

f = foo(10)
f()
foo(20)
f()
```

Burada foo her çağrıldığında yeni bir x değişkeni yaratılmaktadır. Halbuki bar fonksiyonun kullandığı x ilgili çağrıstaki x değişkenidir. Böylece bu örnekte 10 ve 20 değerlerinin değil 10 ve 10 değerlerinin ekranaya yazıldığını görürüz.

Python'da İsim Arama

İsim araması (name lookup) bir değişkenin hangi noktada hangi faaliyet alanlarında aranacağını belirtmektedir. Python dinamik tür sistemine sahip olduğu için ve çalışma yorumlayıcılarla yapıldığı için isim araması da dinamik bir biçimde yapılmaktadır. Kullanılan bir değişken isminin yaratılmış olup olmadığına diğer dillerde olduğu gibi işin başında derleme aşamasında bakılmaz. Arama program akışı o değişkenin kullanıldığı noktaya geldiğinde yapılmaktadır. Böylece değişkenin kullanıldığı noktası, önce bulunan fonksiyon içerisinde sonra kapsayan fonksiyonlar içerisinde sonra da global alanda arama yapılmaktadır. Comprehension'lar ve lambda ifadeleri kendi faaliyet alanlarına sahiptir. Bu konu ileride ele alınacaktır. Örneğin:

```
def foo():
    print(a)

a = 10
foo()      # geçerli
```

Burada her ne kadar foo fonksiyonu tanımlanırken henüz a değişkeni yoksa da foo fonksiyonu çağrılp akış foo içerisindeki a'nın kullanıldığı yere geldiğinde artık a değişkeni isim aramasında bulunacaktır. Ancak çağrıma işlemi aşağıdaki gibi a'nın yaratılmasından önce yapısayıdı error oluşurdu:

```
def foo():
    print(a)

foo()      # error
a = 10
```

Örneğin:

```
def foo():
    bar()

def bar():
    pass
```

```
foo()      # geçerli
```

Burada biz foo çağrılığında bar zaten tanımlanmış olmaktadır. Yani çağrım geçerlidir.

Python'da Permütasyon ve Kombinasyon İşlemleri

Python'ın ileri versiyonlarında math modülünün içerisine permütasyon ve kombinasyon sayılarını elde etmek için perm ve comb fonksiyonları eklenmiştir. Bu fonksiyonlar iki parametre alırlar. perm fonksiyonu n elemanlı kümenin k'lı permütasyonlarının sayısına, comb fonksiyonu da n elemanlı kümenin k'lı kombinasyonlarının sayısına geri dönmektedir. Örneğin:

```
import math

result = math.perm(6, 2)
print(result)

result = math.comb(6, 2)
print(result)
```

Python'da permütasyon işlemleri için standart itertools modülündeki permutations fonksiyonu kullanılmaktadır. Bu fonksiyon bizden dolaşılabilir bir nesne alır ve bize bir dolaşım nesnesi verir. permutations fonksiyonu iki parametrelidir. Fonksiyonun birinci parametresi permütasyonu oluşturacak dolaşılabilir nesneyi alır. İkinci parametresi birinci parametredeki nesnenin kaçlı permütasyonlarının alınacağını belirtmektedir. Fonksiyonun bize geri döndürdüğü dolaşım nesnesi dolaşıldığından tek tek permütasyonlar birer demet biçiminde elde edilmektedir. Örneğin:

```
>>> import itertools
>>> result = itertools.permutations(['ali', 'veli', 'selami'])
>>> list(result)
[('ali', 'veli', 'selami'), ('ali', 'selami', 'veli'), ('veli', 'ali', 'selami'), ('veli', 'selami', 'ali'), ('selami', 'ali', 'veli'), ('selami', 'veli', 'ali')]
```

permutations fonksiyonu iki parametreli de kullanılabilir. Bu durumda ikinci parametre birinci parametredeki dolaşılabilir nesnenin kaçlı permütasyonlarının elde edileceğini belirtir. Örneğin:

```
>>> import itertools
>>> result = itertools.permutations(['ali', 'veli', 'selami'], 2)
>>> list(result)
[('ali', 'veli'), ('ali', 'selami'), ('veli', 'ali'), ('veli', 'selami'), ('selami', 'ali'), ('selami', 'veli')]
```

Aynı modüldeki combinations isimli fonksiyon da aynı biçimde kombinasyonları bize verir. Bu fonksiyonun da iki parametresi vardır. Örneğin:

```
>>> result = itertools.combinations(['ali', 'veli', 'selami'], 2)
>>> list(result)
[('ali', 'veli'), ('ali', 'selami'), ('veli', 'selami')]
```

globals ve locals Fonksiyonları

globals bir built-in fonksiyondur. Bu fonksiyon alandaki değişkenlerin isimlerini ve değerlerini bize bir sözlük nesni olarak vermektedir. Örneğin:

```
x = 10
y = 20

def foo():
```

```
print('foo')

g = globals()

val = g['x']
print(val)

val = g['y']
print(val)

val = g['foo']
print(val)
```

globals fonksiyonuyla elde ettiğimiz sözlüğün anahtarları string olarak değişken isimlerinden, değerleri de o değişkenlerin gerçek değerlerinden oluşmaktadır. globals fonksiyonuyla elde edilen sözlük nesnesinde sizin hiç yaratmadığınız değişken görürseniz şaşırmayınız. Çünkü Python yorumlayıcıları tarafından bazı global değişkenler programın başında yaratılmaktadır.

globals fonksiyonuyla elde ettiğimiz global değişkenleri tutan sözlüğe eklemeler yaptığımda aslında biz yeni global değişkenler yaratmış oluruz. Örneğin:

```
x = 10
y = 20
def foo():
    pass

g = globals()

g['z'] = 100
print(z)
```

Burada x, y ve foo değişkenleri program içerisinde deyimlerle yaratılmıştır. Ancak global z değişkeni sözlüğe ekleme yapma yoluyla yaratılmıştır.

locals builtiln fonksiyonu ise bize çağrılmış noktasına göre yaratılmış olan yerel değişkenleri sözlük olarak vermektedir. Örneğin:

```
def foo():
    y = 20

    def bar():
        z = 30
        d = locals()
        print(d)

    d = locals()
    print(d)

bar()

foo()
```

Kodun çıktısı şöyle olacaktır:

```
{'y': 20, 'bar': <function foo.<locals>.bar at 0x000002A032194798>}
{'z': 30}
```

Ancak biz locals fonksiyonuyla elde ettiğimiz sözlüğe eleman eklediğimizde o fonksiyonun yerel değişken listesine eleman eklemiş olmayız. Örneğin:

```

def foo():
    a = 10
    d = locals()
    d[ 'b' ] = 100

    print(b)

foo()

```

Burada biz b değişkenini foo fonksiyonunun yerel değişken listesine ekleyemedik. Bu nedenle print çağrılarında error oluşacaktır.

İçlemler (Comprehensions)

İçlemler fonksiyonel dillerde çok karşılaşılan kavamlardandır. İçlemler Python'da bir listeyi, kümeyi ya da sözlüğü belki bir özelliğe göre oluşturan sentaks biçimleridir. Bu yönyle içlemler dörde ayrılmaktadır:

- 1) Liste içlemleri (list comprehension)
- 2) Küme içlemleri (set comprehension)
- 3) Sözlük içlemleri (dictionary comprehension)
- 4) Üretici İfadeler

Biz ilk üç içemi bu bölümde inceleyeceğiz. Üretici ifadeler üretici fonksiyonların (generators) işlendiği bölümde ele alınacaktır.

Liste İçlemleri (List Comprehensions)

Liste içlemleri listeler nasıl köşeli parantezlerle oluşturuluyorsa yine köşeli parantezler içerisinde oluşturulmaktadır. Genel biçimleri şöyledir:

```
[<ifade> <for döngüsü> [if koşulu]]
```

Örneğin:

```

>>> a = [1, 2, 3, 4, 5]
>>> b = [x * x for x in a]
>>> b
[1, 4, 9, 16, 25]
>>>

```

List içlemleri şöyle çalışmaktadır: for döngüsü işletilir. Her yinelemede döngü değişkenine çekilen değer soldaki ifadeye sokulur. Bu ifadeden elde edilen değerler bir listede biriktirilir ve o liste ürün olarak verilir. Örneğin yukarıdaki örnekte döngüden sırasıyla x olarak 1, 2, 3, 4, 5 değerleri elde edilecektir. Bu değerler x * x işlemine sokulduğunda 1, 4, 9, 16, 25 değerleri elde edilir. İşte bu değerler de bir liste biçiminde verilmiştir. Aslında yapılan işlem aşağıdaki eşdeğerdir:

```

>>> a = [1, 2, 3, 4, 5]
>>> b = []
>>> for x in a:
    b.append(x * x)

```

Pekiyi işlem ile eşdeğerkarlığı arasında ne fark vardır? İşte içlemler kodun daha kısa ve özlü ifade edilmesini sağlamaktadır. Bu nedenle daha okunabilirdir. Aynı zamanda da daha hızlı olma eğilimindedir. Çünkü içlemler bir bütün olarak arka planda C'de yazılmış olan kodlarla işletilmektedir. Halbuki tek tek yapılan işlemler yorumlayıcılarda daha fazla zaman kaybına yol açmaktadır. (Bir kaynağa içlemlerin göre hız farkı iki kat civarındadır.)

Örneğin:

```
>>> a = [3, 6, 2, 10, 34, 23, 19]
>>> b = [x % 2 == 0 for x in a]
>>> b
[False, True, True, True, True, False]
```

Örneğin:

```
x = list(range(-10, 11, 1))
y = [i * i for i in x]

print(x)
print(y)

import matplotlib.pyplot as plt

plt.plot(x, y)
```

Örneğin:

```
names = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
a = [len(name) for name in names]
print(a)
```

Örneğin:

```
names = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
a = [name[::-1] for name in names]
print(a)
```

Liste içlemelerinde for döngüsünden sonra bir if deyimi de getirilebilir. Tabii burada if deyiminin yalnızca koşul kısmı bulundurulur. Böylece belli koşulu sağlayan değerler soldaki ifadeye sokularak listeler oluşturulabilmektedir.

Örneğin:

```
a = [3, 6, 8, 34, 23, 90, 37, 42, 43]
b = [x for x in a if x % 2 == 0]
print(b)
```

Burada listedeki çift sayılar elde edilmiştir. Sonuç aşağıdaki gibidir:

```
[6, 2, 10, 34]
```

Örneğin bir listede içerisinde 'a' karakteri geçen isimleri bir liste olarak elde etmek isteyelim. Şöyledir bir liste içlemi kullanabailiriz:

```
names = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
a = [name for name in names if 'a' in name]
print(a)
```

Sınıf Çalışması: names isimli bir string listesine isimler yerleştiriniz. İsmi 'a' ya da 'A' ile başlayanları yeni bir liste olarak içem yoluyla elde ediniz.

Çözüm:

```
names = ['Ali', 'veli', 'ayşe', 'Selami', 'fatma', 'Arzu']
anames = [name for name in names if name[0] == 'a' or name[0] == 'A']
print(anames)
```

Koşul kısmı aşağıdaki gibi daha yalınlaştırılabilir:

```
names = ['Ali', 'veli', 'ayşe', 'Selami', 'fatma', 'Arzu']
anames = [name for name in names if name[0].lower() == 'a']
print(anames)
```

Sınıf Çalışması: cities isimli bir listede küçük harfli biçimde bazı şehir isimleri bulunmaktadır. Bunlardan içem yoluya büyük harfli bir liste elde ediniz.

Çözüm:

```
cities = ['ankara', 'izmir', 'eskişehir', 'muğla', 'kastamonu']
upperCities = [city.upper() for city in cities]
print(upperCities)
```

Sınıf Çalışması: Yukarıdaki cities listesinde içerisinde 'a' harfi geçen şehirlerin büyük harf olarak ilk üç harflerinden bir liste oluşturunuz.

Çözüm:

```
cities = ['ankara', 'adana', 'izmir', 'bolu', 'çanakkale', 'amasya']
xcities = [city[:3].upper() for city in cities if city.lower().find('a') != -1]
print(xcities)
```

Aynı işlem in operatörüyle de yapılabilir. Örneğin:

```
cities = ['ankara', 'adana', 'izmir', 'bolu', 'çanakkale', 'amasya']
xcities = [city[:3].upper() for city in cities if 'a' in city.lower()]
print(xcities)
```

Sınıf Çalışması: Yazıldan oluşan bir liste oluşturunuz. Bu listedeki palindrom olan elemanları içem yoluya bir liste biçiminde elde ediniz.

Çözüm:

```
texts = ['anastas mum satsana', 'salih', 'kabak', 'ali', 'ey edip adanada pide ye']
palindromes = [text for text in texts if text == text[::-1]]
print(palindromes)
```

Sınıf Çalışması: int değerlerden oluşan bir liste oluşturunuz. Palindrom sayıları içem yoluya liste biçiminde elde ediniz.

Çözüm:

```
numbers = [12, 1221, 13431, 12345, 197262]
a = [number for number in numbers if str(number) == str(number)[::-1]]
print(a)
```

Sınıf Çalışması: (şehir, plaka kodu) biçimindeki demetlerden oluşan listeden içem yoluya "şehir-plaka kodu" biçiminde bir string listesi elde ediniz.

Çözüm:

```
s = '{}-{}'.format(name, no) for name, no in cities ]
print(s)
```

Tabii şöyle de yapılabılır:

```
cities = [('ankara', 6), ('izmir', 35), ('eskişehir', 26), ('muğla', 48), ('kastamonu', 37)]
s = [name + '-' + str(no) for name, no in cities]
print(s)
```

Şöyleden yapılabildi:

```
cities = [('ankara', 6), ('izmir', 35), ('eskişehir', 26), ('muğla', 48), ('kastamonu', 37)]
s = ['{}-{}'.format(*city) for city in cities]
```

Örneğin:

```
def isprime(val):
    if val % 2 == 0:
        return val == 2
    for i in range(3, val, 2):
        if val % i == 0:
            return False
    return True

result = [i for i in range(2, 1000) if isprime(i)]
print(result)
```

Sınıf Çalışması: Bir listenin çift indisli elemanlarından yeni bir listeyi işlem yoluyla elde ediniz.

Çözüm:

Soru aşağıdaki gibi çözülebilir:

```
l = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

result = [l[i] for i in range(0, len(l), 2)]
print(result)
```

Ancak enumerate fonksiyonu daha sık bir çözüm sunmaktadır:

```
result = [val for index, val in enumerate(l) if index % 2 == 0]
print(result)
```

İçlemi oluştururken iç içe for döngüleri de kullanılabilir. Örneğin şehirlerde geçen karakterlerden bir liste oluşturmak isteyelim:

```
cities = ['ankara', 'izmir', 'eskişehir', 'muğla', 'kastamonu']
charList = [ch for city in cities for ch in city]
s = set(charList)
print(s)
```

Örneğin bir matristeki 20'den büyük ya da 20'ye eşit olan elemanlardan bir listeyi şöyle oluşturabiliriz:

```
a = [[1, 20, 3], [21, 87, 8, 10], [5, 9, 6, 44]]

result = [y for x in a for y in x if y >= 20]
print(result)
```

Sınıf Çalışması: 'izmir' ve 'ankara' sözcüklerinden ilki 'izmir'den ikincisi 'ankara'dan olmak üzere iki karakterli yazılarından oluşan bir listeyi işlem yoluyla elde ediniz. Örneğin: ia, in, ik,... biçiminde.

```
s = [a + b for a in 'izmir' for b in 'ankara']
print(s)
```

İçlemler daha karmaşık bir biçimde oluşturulabilirler. Örneğin içlemlerin sol tarafındaki ifadelerde başka bir işlem kullanılabilir. Bu durumda işlem aşağıdaki gibi bir görünüşe sahip olacaktır:

```
[[ifade for y in x if koşul] for x in a if koşul]
```

Burada sağdaki içlemenin ifadesinin soldaki içlemenin oluşturulmasına dikkat ediniz. Örneğin:

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
b = [[y for y in x if y % 2 == 0] for x in a]  
print(b)
```

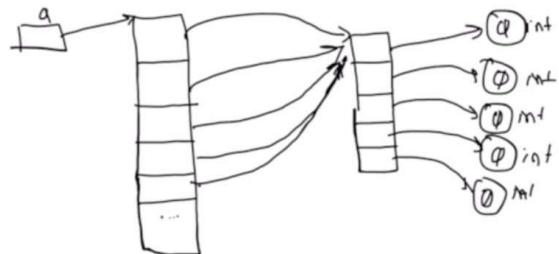
Burada liste içlemenin solundaki ifadede yeni bir işlem kullanılmıştır. Kodun çıktısı şöyle olacaktır:

```
[[2], [4, 6], [8]]
```

Özellikle çok boyutlu listelerin elde edilmesi için buna benzer içlemlerden faydalılmaktadır. Çok boyutlu dizilerin * operatörüyle (repetition) oluşturulması sırasında bir noktaya dikkat çekmek istiyoruz. Örneğin amacımız elemanları 5 tane 0'dan oluşan 10 tane listeyi bir liste biçiminde elde etmek olsun. Bu durumda ilk akla yöntem bu listeyi * operatörüyle oluşturmakdır:

```
a = [[0, 0, 0, 0, 0]] * 10  
print(a)
```

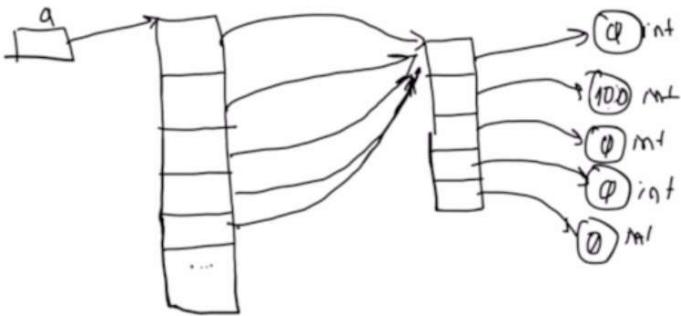
Ancak burada önemli bir problem vardır. * operatörüyle liste çoklandığında aslında liste elemanları aynı liste nesnesinin adresini tutmaktadır.



Burada aslında 5 elemanlı 0'lardan oluşan tek bir liste vardır. Dolayısıyla bu elemanlı listenin elemanları herhangi bir yolla değiştirildiğinde bu değişiklik ana listenin her elemanında ortaya çıkacaktır. Örneğin:

```
a = [[0, 0, 0, 0, 0]] * 10  
print(a)  
a[0][1] = 100  
print(a)
```

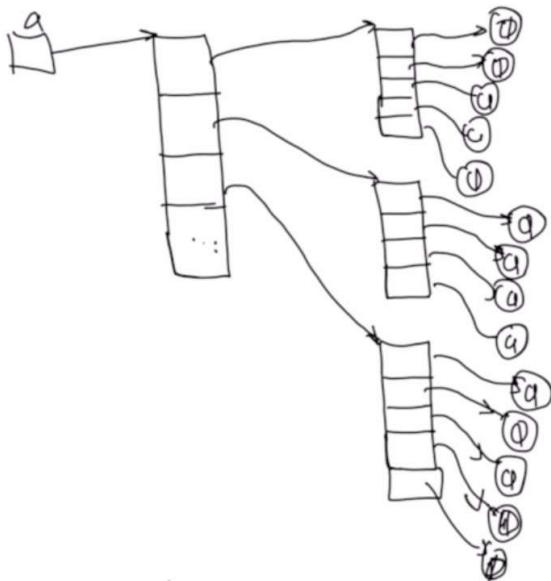
Biz burada dış listenin 0'inci indeksli listesinin 1'inci indeksli elemanını değiştirdiğimizde dış listenin tüm elemanları aynı listeyi gördüğü için aslında onların hepsini değiştirmiş gibi olduk:



İşte eğer iç listelerin bağımsız listeler olması isteniyorsa ya bu işlemin for döngüsüyle açıkça yapılması ya da liste işlemlerinin kullanılması gerekir. Örneğin:

```
a = [[0, 0, 0, 0, 0] for i in range(10)]
print(a)
a[0][1] = 100
print(a)
```

Burada artık iç listelerin tamamen birbirinden farklı olduğuna dikkat ediniz:



Tabii aynı işlem for döngüsüyle de yapılabilirdi:

```
a = []
for i in range(10):
    a.append([0, 0, 0, 0, 0])

print(a)
a[0][1] = 100
print(a)
```

İşlemlerin `in` ifadesi içerisinde de başka işlemler bulunabilir. Bu durumda işlem aşağıdaki gibi bir görünüşe sahip olacaktır:

```
[ifade1 for i in [ifade2 for k in a if koşul] if koşul]
```

Küme İşlemleri (Set Comprehensions)

Bir küme elde edecek biçimde oluşturulan işlemlere küme işlemleri denilmektedir. Sintaks bakımından liste işlemlerinden tek farkı küme işlemlerinin küme parantezlerinin içerisinde yazılmasıdır. Örneğin:

```
>>> s = {ch for ch in 'ankara'}
>>> s
{'n', 'r', 'k', 'a'}
```

Küme içlemelerinde elde edilen kümenin eleman sıralamasının bir kuralı yoktur. Zaten anımsayacağınız gibi kümeler elemanları bir sırayla tutan veri yapıları (sequence container) değildir.

Sınıf Çalışması: 5 tane int türden elemandan oluşan bir liste oluşturunuz. Sonra iki eleman toplamlarından oluşan bir kümeyi küme içlemiyle elde ediniz. Yani listenin her elemanını (kendisi dahil olmak üzere) başka bir elemanıyla toplayarak yeni bir küme elde edilmelidir.

```
a = [1, 3, 5, 7, 9]
b = {x + y for x in a for y in a}
print(b)
```

Süphesiz aslında biz bir liste içlemi ile elde edilen listeyi daha sonra bir kümeye de dönüştürebiliriz. Ancak doğrudan küme içlemi oluşturmak bu tür durumlarda hem daha az kod maliyeti oluşturmaktadır hem de daha hızlıdır.

Sözlük İçlemleri (Dictionary Comprehensions)

Sözlük içlemleri yine küme içlemlerinde olduğu gibi küme parantezleri içerisinde oluşturulur. Ancak for döngüsünün solundaki ifade anahtar:değer biçiminde düzenlenmektedir. Örneğin:

```
d = {i: str(i) for i in [1, 2, 3, 4, 5] }
print(d)
```

Bu işlemim eşdegeri şöyledir:

```
d = {}
for i in [1, 2, 3, 4]:
    d[i] = str(i)

print(d)
```

Örneğin:

```
keys = [1, 3, 5, 7, 9]
values = ['Ali', 'Veli', 'Selami', 'Ayşe', 'Fatma']

d = {keys[i] : values[i] for i in range(len(keys))}

print(d)
```

Örneğin:

```
cities = [('ankara', 6), ('izmir', 35), ('eskişehir', 26), ('muğla', 48), ('kastamonu', 37)]
d = {name: no for name, no in cities}
print(d)
```

Tabii aslında dict tür fonksiyonu ile biz zaten böyle bir listeden sözlük elde edebiliyoruz:

```
d = dict(cities)
print(d)
```

Örneğin:

```
l = [('ali', 123, 1982), ('veli', 65, 1970), ('selami', 340, 1990), ('ayşe', 71, 1969)]
d = {name: (no, year) for name, no, year in l}
print(d)
```

Örneğin biz sözlük içlemeleri yoluyla bir sözlüğün anahtarlarını değer, değerlerini de anahtar yapabiliriz:

```
d1 = {'ali': 123, 'veli': 432, 'selami': 892, 'ayşe': 45, 'fatma': 574}
d2 = {d1[key]: key for key in d1}
print(d2)
```

Aynı işlem şöyle de yapılabilirdi:

```
d1 = {'ali': 123, 'veli': 432, 'selami': 892, 'ayşe': 45, 'fatma': 574}
d2 = {value: key for key, value in d1.items()}
print(d2)
```

Python'da Demet İçlemi Yoktur

Biz içlemeler başlığı altında "liste içlemeleri", "küme içlemeleri" ve "sözlük içlemeleri" biçiminde üç işlem türünü ele aldık. Ancak Python'da "demet içlemi" biçiminde bir işlem türü yoktur. Aslında demet içlemi sentaksına sahip ismine üretici ifadeler (generator expressions)" denilen ayrı bir sentaks bulunmaktadır. Yani bir işlem sentaksını demet parantezi içerisinde kullanlığımızda demet içlemi oluşturmayız. Üretici bir nesne oluşturmuş oluruz. Örneğin:

```
x = (i * i for i in range(10))

print(type(x))
for i in x:
    print(i, end=' ')
```

Üretici ifadeleri ileride "üreticiler (generators)" başlığı içerisinde ele alınmaktadır.

zip Fonksiyonu

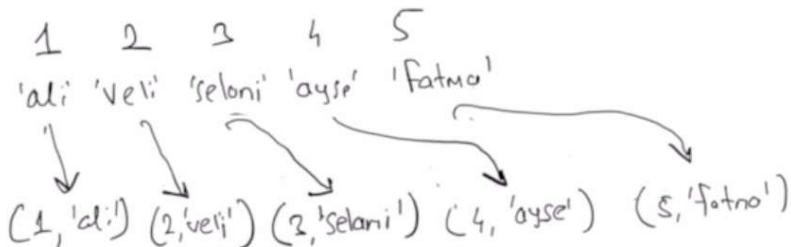
zip built-in bir Python fonksiyonudur. Bu fonksiyon bir grup dolaşılabilir nesneyi alarak bu dolaşılabilir nesnelerin karşılıklı elemanlarından oluşan demetler elde etmekte kullanılmaktadır. Örneğin:

```
a = [1, 2, 3, 4, 5]
b = ['ali', 'veli', 'selami', 'ayşe', 'fatma']

result = zip(a, b)
for t in result:
    print(t)
```

zip fonksiyonundan elde edilen ürün bir dolaşım nesnesidir. Bu nesne dolaşıldığında demetler elde edilmektedir. Elde edilen bu demetlerin elemanları zip fonksiyonuna argüman olarak geçirilen dolaşılabilir nesnedeki değerlerdir. Yukarıdaki programın çıktısı şöyle olacaktır:

```
(1, 'ali')
(2, 'veli')
(3, 'selami')
(4, 'ayşe')
(5, 'fatma')
```



Tabii biz for döngüsünde aynı zamanda açım (unpack) işlemi de yapabiliriz:

```
a = [1, 2, 3, 4, 5]
b = ['ali', 'veli', 'selami', 'ayşe', 'fatma']

for x, y in zip(a, b):
    print(x, y)
```

Örneğin:

```
for x, y in zip('ankara', 'edirne'):
    print(x, y)
```

Örneğin:

```
for x, y, z in zip(range(0, 5), range(10, 15), range(20, 25)):
    print(x, y, z)
```

zip fonksiyonunda argüman olarak girilen dolaşılabilir nesneler aynı uzunlukta olmak zorunda değildir. Bu dolaşılabilir nesnelerden herhangi birinde sona gelindiğinde dolaşım da sonlandırılır. Örneğin:

```
a = [1, 2, 3]
b = [10, 20, 30, 40, 50]

for t in zip(a, b):
    print(t)
```

Buradan şöyle bir çıktı elde edilecektir:

```
(1, 10)
(2, 20)
(3, 30)
```

Aslında zip fonksiyonunun birinci parametresi '*'lidir. Dolayısıyla biz bu fonksiyona bir tane hatta sıfır tane argüman girebiliriz. Örneğin:

```
a = [1, 2, 3]

for t in zip(a):
    print(t)
```

Kodun çıktısı şöyle olacaktır:

```
(1,)
(2,)
(3,)
```

Tabii zip fonksiyonuna girdiğimiz argümanların sayısı ikiden fazla da olabilir. Örneğin:

```
a = [1, 2, 3, 4, 5]
b = ['ali', 'veli', 'selami', 'ayşe', 'fatma']
c = [10, 20, 30, 40, 50]

for t in zip(a, b, c):
    print(t)
```

Kodun çıktısı şöyle olacaktır:

```
(1, 'ali', 10)
(2, 'veli', 20)
```

```
(3, 'selami', 30)
(4, 'ayşe', 40)
(5, 'fatma', 50)
```

Pekiyi zip işleminin tersi nasıl yapılabilir? zip işleminin tersini yapan bir unzip fonksiyonu yoktur. Bu ters işlem zip fonksiyonun kendisi tarafından yapılmakatdır. Aşağıdaki kodu inceleyiniz:

```
result = zip((10, 'ali'), (20, 'veli'), (30, 'selami'))
for t in result:
    print(t)
```

Burada result nesnesi dolaşıldığında iki elemanlı bir demet elde edilir. Demetler de şu biçimde olacaktır:

```
(10, 20, 30)
('ali', 'veli', 'selami')
```

Anımsanacağı gibi argümanda * belirleyicisi "dolaşılabilir nesneyi dolaşıp bu değerlerden argüman oluşturma" anlamına geliyordu. O halde yukarıdaki kodu şöyle de yazabiliriz:

```
a = [(10, 'ali'), (20, 'veli'), (30, 'selami')]

result = zip(*a)
for t in result:
    print(t)
```

Aslında yukarıdaki işlem şöyle de yapılabilirdi:

```
a = [(10, 'ali'), (20, 'veli'), (30, 'selami')]

x, y = zip(*a)
print(x, y)
```

Örneğin:

```
points = [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)]
x, y = zip(*points)

import matplotlib.pyplot as plt

plt.plot(x, y)
plt.show()
```

İstersek buradaki demetleri liste içersine de alabiliriz:

```
l = list(zip(*a))
```

Aşağıdaki gibi bir işlem de uygulanabilir:

```
l = [list(t) for t in zip(*a)]
```

Listeyi benzer biçimde de açabiliriz:

```
numbers, names = [list(t) for t in zip(*a)]
```

bytes ve bytearray Türleri

Python'da daha önceden gördüğümüz temel türlere ek olarak bytes ve bytearray isimli iki tür de vardır. Bu türler de birer sınıf belirtmektedir. bytes değiştirilemez (immutable), bytearray ise değiştirilebilir (mutable) bir sınıfır. bytes

türü aslında bir byte dizisini belirtmek için düşünülmüştür. Bu belirtme yine tek tırnak ya da iki tırnak ile ancak bunların başına yapışık biçimde "b" harfi getirilerek yapılmaktadır. Örneğin:

```
>>> b = b'ali'  
>>> type(b)  
<class 'bytes'>
```

Burada aslında b isimli bytes türünden nesne 3 tane byte değerini tutmaktadır. byte'lerin sayısal değerlerinin [0, 255] arasında olduğunu anımsayınız. O halde buradaki b nesnesi aslında [0, 255] arasındaki sayılarından 3 tane tutmaktadır. Ancak biz bir bytes nesnesini sayılarla değil yazışal biçimde oluştururuz. İşte ASCII tablosunun ilk 128 karakterine karşı gelen sayılar aslında bu byte dizisinin elemanlarını oluşturmaktadır. Örneğin b'ali' nesnesinde 'a'nın ASCII tablosundaki sıra numarası 97, 'l'nin 108 ve 'i'nin de 105'tir. Yani aslında yukarıdaki b nesnesinde tutulan byte değerleri sırasıyla 97, 108 ve 105'tir.

bytes ve bytearray türleri bir yazı gibi düşünülmelidir. Bu sınıflar binary verileri tutmak için oluşturulmuştur. Ancak bu binary veriler bir byte'lık ASCII karakterler verilerek oluşturulmaktadır. bytes türünden bir nesne oluştururken "b" önekinden sonra tırnaklar içerisindeki karakterler ASCII tablosunun ilk 128 karakterinden seçilmek zorundadır. Eğer byte değerleri [128-255] arasıdaysa onlar hex olarak \xhh biçiminde ya da octal olarak \0ooo biçiminde kodlanmalıdır. Örneğin:

```
>>> b = b'ali\x12\xff'  
>>> b  
b'ali\x12\xff'
```

Bu türler dizimsel (sequence) veri yapılarının bütün özelliklerine sahiptir. Yani örneğin bir liste ile yapabildiğimiz dilimleme gibi tüm temel işlemleri bu türlerle de yapabiliriz. Örneğin:

```
>>> b[1:3]  
b'li'  
>>> b[3]  
18  
>>> type(b[3])  
<class 'int'>
```

Göründüğü gibi dilimleme yapıldığında yine bir bytes nesnesi elde edilmektedir. Ancak [...] operatörüyle bytes nesnesinin belli bir indeksteki elemanı int türden verilmektedir.

bytes türü de dolaşılabilir olduğu için biz onu for döngüleriyle dolaşabiliriz. Örneğin:

```
b = b'anakara'  
for i in b:  
    print(i, type(i))
```

for döngüsü ile bir byte dizisi dolaşılmak istendiğinde int biçimde değerler elde edilmektedir. Mademki bytes türü dolaşılabilir bir türdür. O halde diğer veri yapılarını bu türle oluşturabiliriz. Örneğin:

```
>>> b = b'anakara'  
>>> l = list(b)  
>>> l  
[97, 110, 107, 97, 114, 97]
```

bytes türünün başlangıç metodu olan bytes fonksiyonuyla da biz istersek bir bytes nesnesi oluşturabiliriz. bytes fonksiyonuna dolaşılabilir bir nesne verildiğinde onların elemanlarından bytes nesnesi yapılmaktadır. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]  
>>> b = bytes(a)  
>>> b  
b'\n\x14\x1e(2'
```

Bir bytes nesnesi içerisindeki değerler eğer bir ters bölü karakterine karşılık geliyorsa görüntülenirken o ters karakteri görüntülenmektedir. Eğer değer görüntülenemeyen bir karakter belirtiyorsa (bazı ASCII karakterlerinin görüntü karşılıkları yoktur) bu durumda o karakterin \xhh biçiminde görüntülendiğine dikkat ediniz. Nihayet eğer bytes nesnesi oluşturan değer ASCII tablosunun [128-255]'lik kısmında kalmırsa bu durumda yine o karakter \xhh biçiminde görüntülenmektedir.

bytes türü değiştirilebilir (mutable) olmadığı için onun herhangi bir elemanına atama yapamayız. Örneğin:

```
>>> b[2] = 3
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    b[2] = 3
TypeError: 'bytes' object does not support item assignment
```

bytearray türü tamamen bytes gibidir. Ancak elemanlar üzerinde değişiklik yapılabilir. Ancak Python'da bytearray yaratmanın özel bir biçim yoktur. bytearray nesnesi tipik olarak sınıfın başlangıç metodu olan bytearray fonksiyonuyla yaratılmaktadır. Örneğin:

```
>>> b = bytearray()
>>> b
bytearray(b'')
>>> type(b)
<class 'bytearray'>
```

Tabii biz bir bytearray nesnesini başka bir dolaşılabilir nesneden de oluşturabiliriz. Örneğin:

```
>>> a = b'ali'
>>> b = bytearray(a)
>>> b
bytearray(b'ali')
```

Örneğin:

```
>>> b = bytearray([1, 2, 3])
>>> b
bytearray(b'\x01\x02\x03')
```

bytes nesnesini str nesnesine dönüştürmek için str sınıfının tür fonksiyonu olan str fonksiyonunu kullanabiliriz. Eğer çağrı sırasında encoding belirtilirse bu durum bytes nesnesinin içerisindeki byte'ların o encoding'e göre karakter belirttiği anlamına gelmektedir. Örneğin:

```
>>> b = b'anakara\x0Aizmir'
>>> s = str(b, encoding='ASCII')
>>> s
'anakara\nizmir'
>>> len(s)
12
```

Eğer str fonksiyonunda encoding belirtilmemezse bytes nesnesi sanki bir yazılmış gibi str nesnesine dönüştürilmektedir. Örneğin:

```
>>> b = b'anakara\x0Aizmir'
>>> s = str(b)
>>> s
"b'anakara\\nizmir'"
>>> len(s)
16
```

Burada dönüştürülmüş s string'inin "b'ankara\nizmir" biçiminde bir yazı oluşturduğuna dikkat ediniz. bytes nesnesini string nesnesine dönüştürmenin diğer bir yolu da bytes sınıfının decode isimli örnek metodunu kullanmaktır. Örneğin:

```
>>> k = b.decode('UTF-8')
>>> k
'ağrı'
```

Yine decode metodundaki encoding, bytes nesnesi içerisindeki byte'ların karakter kodlamasını belirtmektedir. Yoksa string nesneleri her zaman yazıları UNICODE olarak tutarlar. decode metodu argümansız da çağrılabılır. Bu durumda default olarak 'UTF-8' encoding kullanılır:

```
>>> k = b.decode()
>>> k
'ağrı'
```

Bir str nesnesinin bytes nesnesine dönüştürülmesi ise bytes sınıfının başlangıç metodu olan bytes fonksiyonuyla yapılmaktadır. Ancak bu dönüştürme sırasında encoding belirtilmek zorundadır. Örneğin:

```
>>> s = 'ağrı'
>>> b = bytes(s, encoding='UTF-8')
>>> b
b'a\xc4\x9fr\xc4\xb1'
```

Tabii eğer string içerisindeki karakterler hedef kodlama biçimini ifade edilemiyorsa exception oluşacaktır. Örneğin:

```
>>> s = 'ağrı'
>>> b = bytes(s, 'ASCII')
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    b = bytes(s, 'ASCII')
UnicodeEncodeError: 'ascii' codec can't encode character '\u011f' in position 1: ordinal not in range(128)
```

Ayrıca str sınıfının encode isimli örnek metoduyla da encoding belirterek bir string'i bytes nesnesine dönüştürebiliriz. Örneğin:

```
>>> s = 'ağrı'
>>> b = s.encode('UTF-8')
>>> b
b'a\xc4\x9fr\xc4\xb1'
```

encode metodunu argümansız çağrılabılır. Bu durumda default encoding 'UTF-8' alınacaktır:

```
>>> s = 'ağrı'
>>> b = s.encode()
>>> b
b'a\xc4\x9fr\xc4\xb1'
```

Python standart kütüphanesinde genel olarak string parametreli Python fonksiyonlarına ve metodlarına argüman olarak biz bytes nesnesi de geçirebiliriz. Bu durumda o fonksiyonlar bu bytes nesnesini encode metodunu kullanarak string'e dönüştürmektedir. encode metodundaki default encoding'in UTF-8 olduğunu anımsayınız. Örneğin:

```
f = open(b'ab\x61', 'w')
```

Burada yaratılacak dosyanın ismi "aba" olacaktır.

Python'da kullanılan bazı encoding isimleri şunlardır:

```
ascii  
cp1254  
latin_1  
iso8859_9  
utf_16  
utf_16_be  
utf_16_le  
utf_7  
utf_8
```

Sınıflar (Classes)

Sınıflar Nesne Yönelimli Programlama Tekniği (NYPT)'nin en önemli yapı taşıdır. Sınıflar belli bir konuda belli işlemleri gerçekleştirmek için oluşturulan, içerisinde değişkenleri ve fonksiyon barındıran veri yapılarıdır. Bir sınıf veri elemanlarından ve fonksiyonlardan oluşur. Sınıfın veri elemanlarına Python'da öznitelik (attribute), fonksiyonlarına da metot denilmektedir. Python'da fonksiyon kavramı hiçbir sınıfın içerisinde bulunmayan global ya da yerel fonksiyonlar için kullanılmaktadır.

Bir sınıf bildiriminin genel biçimi şöyledir:

```
class <isim>:  
    <deyimler>
```

Sınıf bildirimi de Python'da aslında bir deyimdir. Yani yorumlayıcı sınıf bildirimini gördüğünde sınıf içerisindeki kodları da çalıştırır. Sınıfın metotları yine def anahtar sözcüğü kullanılarak girinti olarak sınıfın içerisinde bildirilir. Örneğin:

```
class Sample:  
    print('Sample class')  
  
    def foo(self):  
        print('foo')  
  
    def bar(self):  
        print('bar')
```

Burada Sample bir sınıf, foo ve bar da onların metotlarıdır. Yukarıda da ifade edildiği gibi Python'da sınıflar aynı zamanda bir deyim statüsündedir. Bir sınıfın içerisinde birtakım kodlar varsa bu kodlar da çalıştırılır. Örneğin:

```
class Sample:  
    x = 10  
  
    def foo(self):  
        print("foo")  
  
    def bar(self):  
        print("bar")  
  
    print(x)
```

Burada yorumlayıcı Sample bildirimini gördüğünde onun içerisindeki deyimleri de çalıştırır. Böylece x = 10 deyimi ve print deyimi de yapılacaktır. Tabii anımsanacağı gibi aslında fonksiyon ve metot bildirimleri de Python'da birer deyim statüsündedir. Ancak onlar çağrılmamıştır. Fakat çağrılmınca çalışacak durumdadır.

Python'da geleneksel olarak (fakat zorunlu değil) programcının kendi sınıflarının isimleri büyük harfle başlatılarak harflendirilmektedir. Bu harflendirme biçimine "Pascal Notasyonu (Pascal Casting)" denilmektedir. Fakat Python'ın standart kütüphanesindeki sınıflar genel olarak küçük harflerle isimlendirilmiştir.

Sınıflar Türünden Nesnelerin Oluşturulması

Biz bir sınıf bildirimi yaptığımızda bir sınıf oluşturmuş oluruz. Sınıflar aynı zamanda birer tür de belirtmektedir. Yani Python'da nasıl int diye float diye bir tür varsa Sample sınıfı bildirildiğinde de artık Sample isimli bir tür de oluşturulmuş olur. Zaten aslında int, float, bool ve str türleri de birer sınıfır.

Python'da bir sınıf bildirildiğinde bildirilen sınıfın ismi de bir değişkendir. Onun da bir türü vardır. Sınıf ismi belirten değişkenlerin türleri type isimli bir sınıf türündendir.

Örneğin:

```
>>> class Sample:  
    pass  
  
>>> type(Sample)  
<class 'type'>
```

Örneğin:

```
>>> a = 100  
>>> type(a)  
<class 'int'>  
>>> type(int)  
<class 'type'>
```

Burada a'nın türü int'tir. Fakat int isminin türü type'dır.

Aslında int türünün kavramsal olarak Sample türünden bir farkı yoktur. int built-in bir sınıfır. Oysa Sample sınıfını biz oluşturmuş durumdayız. Bir sınıf bildirildiğinde onun için bir nesne yaratılır. O nesne type türündendir. O nesnenin içerisinde oluşturulan sınıfın kodları ve öznitelikleri vardır. Başka bir deyişle type nesneleri aslında sınıfların şablonlarını tutmaktadır.

```
class Sample:  
    pass
```



Örneğin hem Sample hem de int isimleri aslında birer değişken belirtmektedir. Bu değişkenler type nesnesi türündendir.

Örneğin:

```
a = 10
```

Burada a int türündendir. a'nın gösterdiği yerde int türden bir nesne vardır. Fakat int değişkeninin kendisi type türündendir.

Bir sınıf türü oluşturulduktan sonra o sınıf türünden nesneler de yaratılabilir. Nasıl int bir tür belirtiyorsa ve int türden nesneler söz konusu olabiliyorsa benzer biçimde Sample da bir tür belirtir ve Sample sınıfı türünden de nesneler söz konusu olabilir.

Bir sınıf türünden nesne yaratma işleminin genel biçimini söyledir:

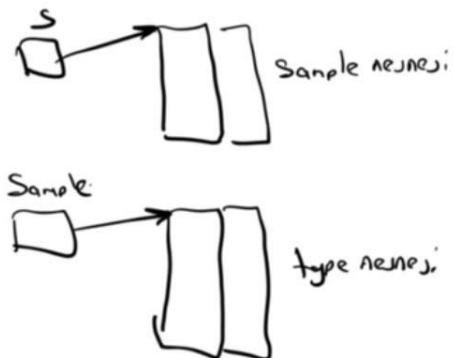
```
<sınıf ismi>([argüman listesi])
```

Bu yaratım işlemiyle nesne oluşturulur ve onun adresi elde edilir. Bu adres bir değişkene atanırsa o değişken artık o sınıf türünden olur. Örneğin:

```
s = Sample()
```

Burada s değişkeni Sample türündendir.

```
s = Sample()
```



Sınıf türünden nesneler Java, C# gibi dillerde new isimli operatörle yaratılmaktadır. Python'da böyle bir operatör olmadığına dikkat ediniz. Örneğin yukarıdaki kodun Java ya da C# karşılığı şöyle ifade edilebilir:

```
Sample s = new Sample();
```

Örneğin:

```
>>> class Sample:  
    pass  
  
>>> s = Sample()  
>>> type(s)  
<class '__main__.Sample'>  
>>> type(Sample)  
<class 'type'>
```

Biz burada önce Sample isimli bir sınıf oluşturduk. Sonra da Sample sınıfı türünden bir nesne oluşturarak o nesneyi s değişkenin göstermesini sağladık.

Aslında nesne oluşturma işlemi int, float, bool ve str türlerinde de benzer biçimde yapılmaktedir. Ancak built-in temel türlerde sabitleri kullandığımız anda nesne zaten yorumlayıcı tarafından otomatik olarak oluşturulmaktadır. Örneğin:

```
a = 123
```

Bu işlemin şundan bir farkı yoktur:

```
a = int(123)
```

Biz bu türlerden nesneleri argüman girmeden de yaratabiliriz. Bu durumda bu nesnelerin içerdikleri değerler 0 olur. Örneğin:

```
>>> a = int()  
>>> a  
0
```

Aslında nesne yaratma işlemini şöyle de düşünebiliriz. Burada sınıfın ismi T olsun. Eğer T türünden bir değişken fonksiyon çağrıma operatörüyle kullanılırsa bu o türden değişkenin yaratılacağı anlamına gelmektedir. Örneğin sınıfın ismi T olsun:

```
t = T(...)
```

Burada T türünden bir nesne yarattık. t değişkeni bu nesneyi göstermektedir.

Sınıfın Metotlarının Çağrılması ve self Parametresinin Anlamı

Sınıfın bir metodunun en azından bir parametresi olması gereklidir. Metotların ilk parametreleri ilgili sınıf türünden bir nesneyi temsil eder. Bu ilk parametre için genellikle self ismi verilmektedir. Aslında bu ilk parametre isminin self olması zorunlu değildir. Ancak böyle bir gelenek vardır. (Eğer bu parametreye farklı bir isim verilirse Python yorumlayıcı uyarısı vermektedir.)

Bir sınıfın metodu o sınıf türünden bir değişkenle ve '.' operatörü kullanılarak çağrılır. Genel biçim şöyledir.

```
<sınıf türünden değişken>.<fonksiyon ismi>([argüman listesi])
```

Örneğin:

```
class Sample:  
    def foo(self):  
        print("foo")  
  
    def bar(self):  
        print("bar")  
  
a = Sample()  
a.foo()  
a.bar()
```

Bir fonksiyon hiçbir sınıf değişkeni kullanılmadan çağrıldığında yorumlayıcı o fonksiyonun global bir fonksiyon olduğunu düşünür. Fonksiyon bir sınıf değişkeni ile çağrıldığında ise derleyici o fonksiyonun o sınıfın bir metodunu olduğunu düşünmektedir. Örneğin iki çağrı arasındaki farka bakınız:

```
foo()          # global Foo çağrılmıyor  
s.foo()        # sınıfın foo isimli metodу çağrılmıyor
```

Yukarıdaki örnekteki Sample sınıfının foo metodunun çağrılmıştırında hiçbir argüman girilmediğine dikkat ediniz. Halbuki bu foo metodunun bir parametresi vardır. Oysa biz adeta bu parametre yokmuş gibi davrandık. İşte aslında biz bu self parametresi için gizlice argüman girmiştir durumdayız. Şöyle ki: Biz bir sınıf türünden değişkenle sınıfın metodunu çağrıdığımızda bu metod çağrılmaktır. Kullandığımız değişken aslında bir argüman gibi metoda gizlice aktarılmaktadır. Başka bir deyişle aslında metodun çağrılmamasında kullanılan değişken metodun ilk parametresi için girilen argüman olmaktadır.

```
s.foo()
```

çalışımının eşdeğeri:

```
Sample.foo(s)
```

birimindedir. Python'da sınıf ismi ile sınıf metodlarının çağrılmış biçimini de geçerli olarak kullanılabilmektedir. Ancak metodların nesnelerle çağrılmış biçimini daha okunabilir ve anlaşılabilir olduğundan tercih edilmektedir. Tabii bu durum her sınıf için geçerlidir. Örneğin:

```
s = 'this is a test'  
k = s.upper()  
print(k)
```

Biz bu çağrıyı şöyle de yapabildik:

```
s = 'this is a test'  
k = str.upper(s)  
print(k)
```

Aslında C++, Java ve C#'ta da bir sınıfın bir fonksiyonu (sınıfın fonksiyonlarına C++'ta "üye fonksiyon", Java ve C#'ta metot denilmektedir) aşağıdaki gibi çağrılmaktadır:

```
a.foo()
```

Yine o dillerde derleyiciler bu fonksiyonun çağrılmamasında kullanılan değişkeni birinci parametre olarak fonksiyona gizlice geçirirmektedir. Ne var ki o dillerde programcı bu birinci parametreyi hiç belirtmez. Fakat fonksiyon içerisinde this anahtar sözcüğü ile onu kullanma hakkına sahiptir.

```
class Sample:  
    def foo(self):  
        print("Sample.foo")  
  
class Mample:  
    def foo(self):  
        print("Mample.foo")  
  
s = Sample()  
m = Mample()  
  
s.foo()  
m.foo()  
  
Sample.foo(s)  
Mample.foo(m)
```

Bu örnekte iki farklı sınıfta aynı isimli metot bulunduğuunu görüyorsunuz. Bu metodlar farklı sınıflarda olduğu için birbirlerine karışmazlar.

Python'da Değişkenlerin Bildirim (Yaratım) Yerleri

Sınıflar konusu da dahil olmak üzere Python'da bir değişken dört yerde oluşturulabilmektedir:

1) Global alanda. Fonksiyonların dışında bir değişkene ilk kez atama yaptığımız zaman onu oluşturmuş oluruz. Böyle değişkenlere "global değişkenler" denilmektedir. Örneğin:

```
x = 10  
  
def foo():  
    print(x)  
  
foo()          # 10  
print(x)       # 10
```

Burada x tüm fonksiyonların dışında yaratıldığı için global bir değişkendir. Global değişkenler her yerden kullanılabilirler.

2) Fonksiyonların ya da metodların içerisinde. Bir fonksiyonun ya da metodun içerisinde ilk kez bir değişkene değer atandığında yeni bir değişken oluşturulur. Böyle değişkenlere "yerel değişkenler" denilmektedir. Örneğin:

```
x = 10

def foo():
    y = 20
    print(y)          # 20

class Sample:
    def bar(self):
        z = 30         # 30
        print(z)

foo()
s = Sample()
s.bar()
print(x)          # 10
```

Burada foo fonksiyonunun içerisindeki y ve bar fonksiyonunun içerisindeki z yerel değişkenlerdir. Yerel değişkenler yalnızca yaratıldıkları fonksiyonlarda ve metodlarda kullanılabilirler. Anımsanacağı gibi bir fonksiyon ya da metotta global bir değişken kullanıldıktan artık ona değer atanamaz. Ancak o değişkenin global olduğu belirtilebilir. Örneğin:

```
x = 10

def foo():
    print(x)      # error
    x = 20
    print(x)
```

Fakat ancak:

```
x = 10

def foo():
    x = 20      # geçerli, yeni bir yerel x
    print(x)
```

Tabii fonksiyon ya da metot içerisinde global olan değişkeni kullanıp değiştirmek istersek global bildirimini yapmamız gereklidir. Örneğin:

```
x = 10

def foo():
    global x
    x = 20      # global olan x
    print(x)

foo()
print(x)      # 20
```

3) Bir sınıfın içerisinde. Sınıfın içerisinde fakat metodların dışında bir değişkene atama yapıldığında o sınıfa özgü yeni bir değişken yaratılmış olur. Bu değişken global ya da yerel değildir. Sınıfa özgüdür. Bunlara "sınıf öznitelikleri (class attributes)" denir. Sınıf özniteliklerine sınıfın içinden doğrudan, sınıfın dışından ve sınıfın metodlarından sınıf ismi kullanılarak erişilir. Örneğin:

```

x = 10                      # x bir global değişken

class Sample:
    y = 20                  # y bir sınıf özniteligi
    print(x)                # 10
    print(y)                # 20

    def foo(self):
        z = 30                # z bir yerel değişken
        print(z)              # 30
        print(Sample.y) # 20

    print(x)                # 10
    print(Sample.y)          # 20
s = Sample()
s.foo()

```

Yukarıdaki örnekte sınıf bildirimi içerisinde y değişkenine değer atanmıştır. Artık bu y değişkeni global değil sınıfın faaliyet alanında ayrı bir değişkendir. Nasıl global ve yerel değişkenler ayrı faaliyet alanlarında bulunan değişkenlerse sınıfın içerisinde (ama metodların içerisinde değil) yaratılmış olan değişkenler de böyle sınıfa özgü farklı değişkenlerdir. Sınıf öznitelikleri sınıfın içerisinde (ama metodların içerisinde değil) doğrudan isimleriyle, sınıfın dışında ya da metodların içehrinde sınıf ismi ve nokta operatörüyle kullanılmaktadır.

Global bir değişkenle aynı isimli bir sınıf özniteligi ve yerel değişken oluşturulabilir. Bunlar farklı değişkenlerdir, birbirlerine karışmazlar. Örneğin:

```

x = 100
class Sample:
    x = 200
    def foo(self):
        x = 300
        print(x)      # 300
    print(x)          # 200
print(x)          # 100

s = Sample()
s.foo()

```

Sınıf içerisinde bir global değişken kullanılıp sonra aynı isimli bir sınıf özniteligi oluşturulabilir. Bu durum geçerlidir. (Halbuki metodlarda bu durum geçerli değildi). Örneğin:

```

x = 10                      # x bir global değişken

class Sample:
    print(x)                # geçerli global x
    x = 20                  # yeni bir sınıf değişkeni olan x yaratılıyor

print(x)                      # 10
print(Sample.x)              # 20
print(x)                      # 10

```

4) Değişkenler sınıf nesnesinin bir örnek özniteligi (instance attribute) olarak da yaratılabilirler. Böyle değişkenler sınıfın değil o sınıf türünden nesnenin bir parçasını oluşturmaktadır. Nesnenin özniteliklerinin yaratılması izleyen başlıkta ele alınmaktadır. Örneğin:

```

class Point:
    def set(self, x, y):
        self.x = x
        self.y = y

```

```

def disp(self):
    print(self.x, self.y)

pt = Point()
pt.set(3, 5)
pt.disp()

```

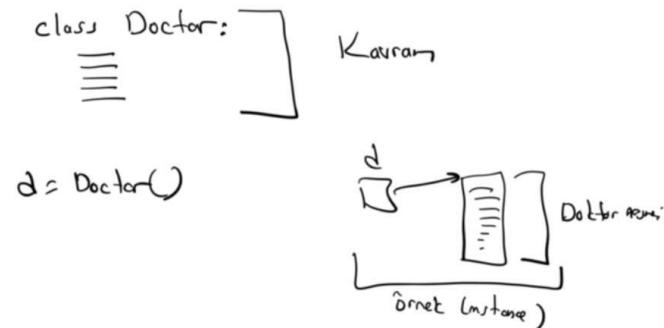
Burada set içerisinde self.x ve self.y ifadelerine atama yapıldığında oluşturulan değişkenler örnek öznitelikleridir.

Sınıf Nedir?

Sınıflar NYPT'nin en önemli yapı taşlarıdır. Bu teknik sınıflarla uygulanmaktadır. Bir sınıf birtakım özniteliklerden (attributes) ve bu öznitelikler üzerinde işlemler yapan metodlardan oluşan bir veri yapısıdır. Sınıflar belli bir konuda bir iş yapmak için tasarlanırlar ve kullanılırlar. Örneğin filse sınıfı dosya işlemleri, string sınıfı yazı işlemleri, socket sınıfı network haberleşme işlemlerini yapma iddiasındadır.

Kavramlar (concepts) doğadı somut varlıkların ortak özelliklerinden hareketle bir sınıf belirtmek amacıyla oluşturulmuş sözcükler ve imgelerdir. Kavramlar gerçek dünyada fizikal bir yer kaplamazlar. Bizim zihnimizde bir temsil belirtirler. Aslında dış dünyaya ağaç kavramı yoktur. Yaprakları olan çeşitli bitkiler vardır. Biz bu bitkilere ortak özelliklerinden dolayı ağaç diyoruz. Ağaç bir kavramdır. Belli özellikleri olan bitkiler zihnimizde temsil eder. Kavramların çeşitli özellikleri (öznitelikleri) vardır. Örneğin bir ağaçın türü, yaprakları, ömrü, nerede yetişebildiği onun öznitelikleridir. Gerçek hayatı "doktor" diye bir nesne yoktur. Doktor bizim kafamızdaki bir kavramdır. Belli eğitimi almış sağlık işleriyle uğraşan somut insanlara biz doktor diyoruz. Kafamızdaki doktor kavramı gerçek dünyada yer kaplamaz. Doktor kavramının öznitelikleri "ismi", "uzmanlık alanı" gibi bilgilerden oluşur.

Sınıflar NYPT'de birer kavram belirtirler. Örneğin Doktor sınıfı doktor kavramını, okul sınıfı okul kavramını belirtir. Ancak bunlar somut bir doktor ya da somut bir okul belirtmezler. Bir kavrama uygun olan somut nesnelere NYPT'de "örnek (instance)" denilmektedir. Örnek belli bir kavramı temsil etme yeteneğindeki somut nesnelerdir. İşte NYPT'de bir sınıf türünden nesneler (değişkenlerin gösterdiği yerdeki nesneler) o sınıf türünden örneklerdir. Örneğin:



Bir proje nesne yönelimli olarak modellenirken proje içerisindeki tüm kavramlar birer sınıfta temsil edilir. Sonra sınıflar türünden nesneler oluşturulur. Bu nesnelerin öznitelikleri (attribute'leri) set edilerek gerçek somut nesneler elde edilir. Örneğin bir hastane otomasyonunda tüm kavramlar (hastane, doktor, hemşire, hastalık vs.) birer sınıfı temsil edilmektedir. Sonra bu sınıflar türünden gerçek nesneler (örnekler) yaratılarak kodlama yapılmaktadır. Örneğin Hastane otomasyonunda Doktor bir kavramdır ve bir sınıf ile temsil edilir. O Doktor kavramından nesneler yaratıldıkça biz gerçek doktorları elde ederiz. Örneğin hastanede 20 doktor çalışıysa bizim 20 tane Doktor nesnesi (örneği) yaratmamız gereklidir.

Sınıf Nesnelerinin Örnek Özniteliklerinin Oluşturulması

Bir sınıf nesnesine özgü olan onun bir parçasını oluşturan nesnelere (elemanlara) "örnek öznitelikleri (instance attribute)" denilmektedir. Sınıf içerisinde bildirilen değişkenlere ise Python terminolojisinde "sınıf öznitelikleri" denildiğini anımsayınız. Sınıf içerisinde yaratılan bu değişkenler belli bir sınıf nesnesinin değil o sınıfın elemanlarıdır. Halbuki örnek öznitelikleri o sınıf türünden nesnenin parçalarını oluşturmaktadır. Öznitelikleri diğer programlama dillerindeki veri elemanlarına (data member) benzetebilirsiniz. Bu bağlamda örneğin örnek öznitelikleri C++, Java ve C#'taki static olmayan veri elemanlarına, sınıf öznitelikleri de static veri elemanlarına işlevsel olarak benzemektedir.

Pekiyi Python'da bir sınıf türünden nesnenin belli bir özniteligi nasıl oluşturulmaktadır? Python'da diğer dillerdeki gibi işin başında örnek öznitelikleri (veri elemanları) bildirilmezler. Bir sınıf türünden değişken ile aşağıdaki gibi nokta operatörü kullanılarak ilk kez atama yapıldığında oorneğe ilişkin örnek özniteligi oluşturulmuş olur. Bu işlemin genel bicimi şöyledir:

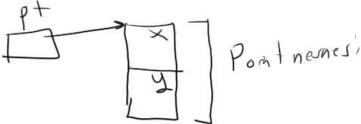
```
<sınıf türünden değişken>.<örnek özniteliginin ismi> = değer
```

Örneğin:

```
class Point:  
    pass  
  
pt = Point()  
pt.x = 10      # pt nesnesinin x isimli bir özniteligini oluşturduk  
pt.y = 20      # pt nesnesinin y isimli bir özniteligini oluşturduk  
  
print("x = {}, y = {}".format(pt.x, pt.y))
```

Burada önce Point sınıfı türünden bir pt nesnesi yaratılmıştır. Bu kullanılarak pt.x ve pt.y ifadelerine atama yapıldığında bu nesnenin x ve y öznitelikleri oluşturulmuş olur. Örnek öznitelikleri yaratılan sınıf nesnesinin parçalarını oluşturmaktadır. Bunu çizimle şu biçimde gösterebiliriz:

pt = Point()
pt.x = 10
pt.y = 20

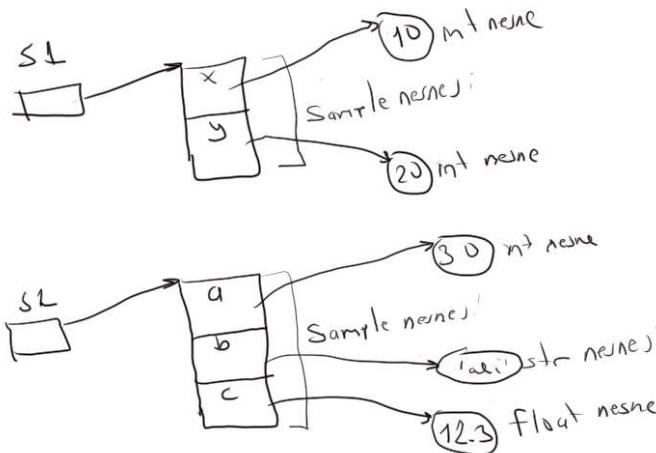


Python'da (diğer dillerin çoğu gibi) bir sınıf türünden bir nesnenin örnek öznitelikleri aynı sınıf türünden başka bir nesnenin örnek özniteliklerinden farklı olabilir. Bu durum kavramsal bakımdan tuhaftır oluşturmaktadır. Çünkü biz bir sınıfın bir kavram belirttiğini dolayısıyla kavramların da aynı özniteliklere sahip olduğunu bilmekteyiz.

Örneğin:

```
class Sample:  
    pass  
  
s1 = Sample()  
s1.x = 10  
s1.y = 20  
  
print('x = {}, y = {}'.format(s1.x, s1.y))  
  
s2 = Sample()  
s2.a = 30  
s2.b = 'ali'  
s2.c = 12.3  
  
print('a = {}, b = {}, c = {}'.format(s2.a, s2.b, s2.c))
```

Burada görüldüğü gibi s1'in öznitelikleri x ve y biçiminde, s2'nin öznitelikleri a ve b biçimindedir. Bunu şekilsel olarak söyle gösterebiliriz:



Tabii bir kavramın özniteliklerinin o kavramın her birörneğinde aynı olması beklenir. Bu nedenle böylesi bir durum pek tercih edilmez. Yani normalde aynı sınıfın bütün örneklerinin örnek özniteliklerinin aynı olması beklenir. Pekiyi bunu nasıl sağlayabiliriz? Biz nesnesin örnek özniteliklerini bir metodun içerisinde yaratırsak o metod her çağrıdığımızda o öznitelikler aynı isimle oluşturulmuş olacaktır. Örneğin:

```
class Point:
    def set(self, x, y):
        self.x = x
        self.y = y

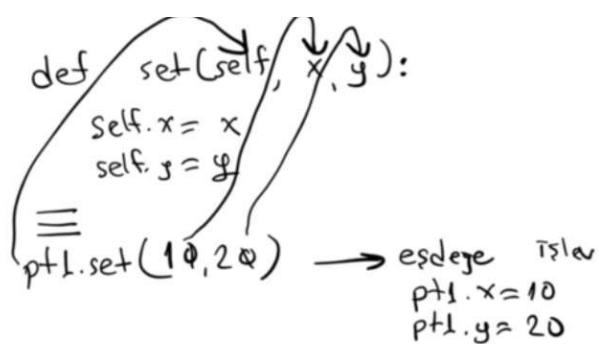
    def disp(self):
        print(self.x, self.y)

pt1 = Point()
pt2 = Point()

pt1.set(10, 20)
pt2.set(30, 40)

pt1.disp()
pt2.disp()
```

Burada biz set metodu içerisinde self.x ve self.y ifadelerine atama yaptığımızda x ve y isimli iki örnek öznitelik oluşturmuş oluruz. Bunu şekilsel olarak şöyle gösterebiliriz:



Benzer biçimde disp çağrımasında da aktarım şöyle olmaktadır:

```

def disp(self):
    print('({},{})'.format(self.x, self.y))

pt1, disp() → esdeger etti:
    print('({},{})'.format(pt1.x, pt1.y))

```

Sınıflar Python'da değiştirilebilir (mutable) türlerdir. Dolayısıyla biz bir sınıf yazdığımızda o sınıf nesnesinin parçaları olan örnek özniteliklerini değiştirebilmekteyiz. Python'da sınıfın metodları nesnesin örnek özniteliklerini doğrudan değil yine o sınıf türünden değişkenle (self ile) kullanabilir. (Halbuki C++, Java, C# gibi dillerde sınıfların metodları sınıfların veri elemanlarını doğrudan kullanabilmektedir.) Örneğin:

```

class Date:
    def set(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def disp(self):
        print('{}/{}/{}'.format(self.day, self.month, self.year))

date = Date()
date.set(10, 12, 1990)
date.disp()

```

Örneğin:

```

class Complex:
    def set(self, real, imag):
        self.real = real
        self.imag = imag

    def disp(self):
        print("{}+{}i".format(self.real, self.imag))

z1 = Complex()
z2 = Complex()

z1.set(2, 3)
z2.set(4, 5)

z1.disp()
z2.disp()

```

Pekiyi biz önce disp metodunu sonra set medunu çağrırsaydık ne olurdu? İşte sınıfın öznitelikleri set metodunda yaratıldığı için önce disp metodunu çağrırdığımızda henüz bu öznitelikler yaratılmamış olacağından çalışma zamanı sırasında exception oluşurdu.

Dunder (Double Underscore) Terimi

Python'da başında ve sonunda iki alt tire bulunan özel sınıf metodları vardır. Bu metodların ismi __xxx__ biçimindedir. Söylerken kolaylık olsun diye bu metodların isimleri "dunder" terimiyle ifade edilir. Bu durumda örneğin __init__ metoduna "dunder init" metodu denilmektedir.

Sınıfların __new__ ve __init__ Metotları

Python'da bir nesne yaratıldığında arka planda `__new__` isimli static bir metot çağrılmaktadır. `__new__` metodu nesnenin yaratımından sorumludur. Bu yaratım aslında tepedeki Object sınıfının `__new__` metodu tarafından yapılmaktadır. Programcılar gerekli gördüklerinde bu `__new__` metodunu override ederler. Override ettikleri `__new__` metodunda da taban object sınıfının `__new__` metodunu çağrırlar. Bu `__new__` metoduna Python terminolojisinde "constructor" denilmektedir.

Python'da ne zaman bir nesne yaratılsa yorumlayıcı `__new__` metodunu çağrıdıktan sonra eğer bu metodun geri dönüş değeri ilgili sınıf türünden ya da o sınıfın bir türemiş sınıfı türündense `__init__` metodunu da çağrırmaktadır. Programcılar nadiren `__new__` metodunu yazmaya gereksinim duyarlar. Default durumda tahsisat Object sınıfının `__new__` metodu tarafından yapılmaktadır.

Sınıfların `__init__` metodları nesnenin özniteliklerini yaratmak için ve bazı ilk işlemleri yapmak için kullanılmaktadır. `__init__` metoduna da Python terminolojisinde "initializer" denilmektedir. Python'daki "initializer" metodu aslında diğer dillerdeki "constructor" metoduna benzemektedir. Biz bir sınıf için `__init__` metodunu yazmak zorunda değiliz. Bu durumda yine Object sınıfının `__init__` metodu çağrılmaktadır. Dolayısıyla `__init__` metodunun sınıfta olmayı bir soruna yol açmamaktadır.

Örneğin:

```
class Sample:  
    def __init__(self):  
        self.a = 10  
        self.b = 20  
  
    def disp(self):  
        print(self.a, self.b)  
  
x = Sample()  
x.disp()
```

Burada `__init__` metodu içerisinde biz `self.a` ve `self.b` atamaları yaparak iki öznitelik oluşturmuş olduk. Artık diğer metodlar bunları kullanabilecektir. `__init__` metodu nesne yaratılırken otomatik biçimde çağrıldığı için artık diğer metodların olmayan bir özniteliği kullanma durumları da oluşmayacaktır.

Örneğin:

```
class Date:  
    def __init__(self):  
        self.day = 1  
        self.month = 1  
        self.year = 1900  
  
    def disp(self):  
        print('{}/{}/{}'.format(self.day, self.month, self.year))  
  
date = Date()  
date.disp()      # Date.disp(date)
```

`__init__` metodunun tek parametresi olmak zorunda değildir. Bu metot dışarıdan parametre alarak öznitelikleri o parametrelerle set edebilir. Bu durumda nesneyi oluştururken sınıf isminden sonra parantezlerin içerisinde bu parametrelere karşı gelen argümanlar da girilmelidir. Örneğin:

```
class Sample:  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
    def disp(self):  
        print(self.a, self.b)
```

```
x = Sample(10, 20)
x.disp()
```

Burada nesne yaratılırken girilmiş olan 10 değeri a'ya 20 değeri de b'ye aktarılmaktadır.

`__init__` metodundaki `self` parametresi yeni yaratılan nesneyi temsil etmektedir. Bu parametre aslında `__new__` metodu tarafından `__init__` metoduna geçirilmektedir. Yani nesneyi gerçekle yaratınan `__new__` metodu yaratılan nesnenin referansını bu `__init__` metoduna geçirmektedir. Örneğin:

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def disp(self):
        print('{}/{}/{}'.format(self.day, self.month, self.year))

date = Date(10, 12, 2009)
date.disp()      # Date.disp(date)
```

Örneğin:

```
class Student:
    def __init__(self, name, no):
        self.name = name
        self.no = no

    def disp(self):
        print('İsim = {}, No = {}'.format(self.name, self.no))

s = Student('Ali Serçe', 123)
s.disp()
```

Anımsanacağı gibi Python'da fonksiyonların ya da metodların overload edilmesi söz konusu değildir. Dolayısıyla sınıfın tek bir `__init__` metodu vardır. Tabii `__init__` metodunu default argüman alabilir. Örneğin:

```
class Complex:
    def __init__(self, real = 0, imag = 0):
        self.real = real
        self.imag = imag

    def disp(self):
        print('{}+{}i'.format(self.real, self.imag))

z1 = Complex()
z1.disp()

z2 = Complex(10)
z2.disp()

z3 = Complex(10, 20)
z3.disp()
```

Sınıf Kullanımına Bir Örnek: Python Standart Kütüphanelerindeki date ve timedelta Sınıfları

Python'ın standart kütüphanelerinde `datetime` isimli modüldeki `date` sınıfı tarih işlemlerini yapmak için bulundurulmuştur. Sınıfın `__init__` metodu nesnenin tutacağı gün, ay ve yıl bilgisini bizden alır. Bizden aldığı bu bilgileri sınıfın `day`, `month` ve `year` isimli örnek özniteliklerine yerleştirir. Örneğin:

```

import datetime

d = datetime.date(2009, 12, 5)
print('{}/{}/{}'.format(d.day, d.month, d.year))

```

date sınıfının weekday isimli metodу parametresizdir. Bu metot ilgili tarihin hangi gün olduğuna ilişkin int bir değer geri döndürür. Bu değer Pazartesi = 0, Salı = 1, ... biçimindedir. Örneğin:

```

import datetime

d = datetime.date(1920, 4, 23)
print('{}/{}/{}'.format(d.day, d.month, d.year))
days = ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi']

wd = d.weekday()
print(days[wd])

```

Sınıf Çalışması: Klavyeden dd/mm/yyyy formatında bir tarih bilgisi alıp o tarihin hangi gün olduğunu yazdırınız.

Çözüm:

```

import datetime as dt

s = input('Tarih giriniz: ')
day, month, year = [int(k) for k in s.split('/')]

d = dt.date(year, month, day)
day_text = ['Pazartesi', 'Salı', 'Çarşamba', 'Perşembe', 'Cuma', 'Cumartesi', 'Pazar']
print(day_text[d.weekday()])

```

Sınıfın ctime isimli metodу bize ilgili tarihe ilişkin bilgiyi yazı olarak verir. Örneğin:

```

import datetime

d = datetime.date(1999, 8, 17)
s = d.ctime()
print(s)

```

İki date nesnesi karşılaştırma operatörleriyle karşılaştırma işlemeye sokulabilmektedir. Çünkü date sınıfının bu işlemleri yapan operatör metodları vardır. Operatör metodları konusu ileride ele alınmaktadır. Örneğin:

```

import datetime

d1 = datetime.date(1999, 8, 17)
d2 = datetime.date(1999, 8, 18)

if d1 > d2:
    print('d1 > d2')
elif d1 < d2:
    print('d1 > d2')
else:
    print('d1 == d2')

```

date sınıfının today isimli sınıf metodу (class method) bilgisayarın saatine bakarak o anki tarih bilgisini bize date nesnesi olarak verir. Sınıf metodları sınıf ismi ile çağrıılır. Bu konu ileride ele alınmaktadır. Örneğin:

```

import datetime

```

```
d = datetime.date.today()  
print(d)
```

Tabii from import deyimini kullanarak date ismini doğrudan da belirtebiliriz:

```
from datetime import date  
  
d = date(2009, 12, 21)  
print(d)  
  
print(date.today())
```

datetime modülündeki timedelta isimli sınıf bir zaman aralığını tutmak için bulundurulmuştur. Sınıfın `__init__` metodu bizden tutulacak zaman aralığını alır. Metodun parametrik yapısı şöyledir:

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0,  
weeks=0)
```

Örneğin:

```
import datetime  
  
td = datetime.timedelta(hours=3, minutes=27, seconds=17)  
print(td)
```

Sınıfın days, seconds, microseconds örnek öznitelikleri tutulan zaman aralığının bileşenlerini bize vermektedir. Burada seconds zaman aralığının toplam saniye değerini bize vermektedir. Aynı bilgi sınıfın total_seconds metoduyla da elde edilebilir. Örneğin:

```
import datetime  
  
td = datetime.timedelta(hours=3, minutes=27, seconds=17)  
print(td.seconds)
```

Yine date sınıfında olduğu gibi timedelta sınıfında da karşılaştırma işlemleri yapılabilmektedir.

İki date nesnesini çıkarma işlemine soktuğumuzda ürün olarak timedelta nesnesi elde ederiz. Bu iki tarih arasındaki zaman farkını bize vermektedir. Örneğin:

```
import datetime  
  
d1 = datetime.date(2021, 7, 10)  
d2 = datetime.date(2021, 4, 18)  
  
td = d1 - d2  
print(td)
```

Bir date nesnesi ile bir timedelta nesnesi de toplama ve çıkartma işlemine sokulabilir. Bu durumda elde edilen ürün date türünden olacaktır. Örneğin bu sayede biz bir tarihten belli bir gün sonraki ya da belli bir gün önceki tarihi bulabiliyoruz:

```
import datetime  
  
d = datetime.date(2021, 7, 10)  
td = datetime.timedelta(days=50)  
  
result = d + td  
print(result)
```

Sınıf Kullanımına Bir Örnek Daha: Python'ın Standart Kütüphanesindeki `datetime` Sınıfı

`datetime` modülündeki `datetime` isimli sınıf hem tarih hem de zaman bilgisini tutan ve bunlar üzerinde işlem yapan bir sınıfıdır. Bir `datetime` nesnesi sırasıyla yıl, ay, gün, saat, dakika, saniye ve mikrosaniye belirtilerek yaratılmaktadır. Yıl, gün ve ay dışındaki parametreler default sıfır değerini almaktadır. Örneğin:

```
from datetime import datetime

dt = datetime(2019, 7, 7, 18, 58, 32)
print(dt)
```

Nesnenin `year`, `month`, `day`, `hour`, `minute`, `second` ve `microsecond` isimli örnek öznitelikleri programcının set ettiği bu değerleri bize vermektedir. Örneğin:

```
from datetime import datetime

dt = datetime(2019, 7, 7, 18, 58, 32)
print(dt.hour, dt.minute, dt.second)
```

Bu sınıfın `now` isimli metodunu bize çağrıldığı andaki tarih ve zaman bilgisini `datetime` nesnesi olarak vermektedir. Örneğin:

```
from datetime import datetime

dt = datetime.now()
print(dt)
```

Yine bu sınıfın da `weekday` isimli metodu vardır. Bu metod bize ilgili tarih zamanın haftanın hangi gününe karşılık geldiğini vermektedir. Benzer biçimde iki `datetime` nesnesi yine karşılaştırma operatörleriyle karşılaştırma işlemeye sokulabilmektedir.

Sınıflar Değiştirilebilir (Mutable) Türlerdir

Sınıflar değiştirilebilir (mutable) türlerdir. Yani sınıfın parçalarını oluşturan örnek öznitelikleri (`instance attributes`) üzerinde değişiklikler yapabiliriz. Böylece bu örnek öznitelikleri başka nesnelerin adreslerini tutar hale getirilebilir. Örneğin:

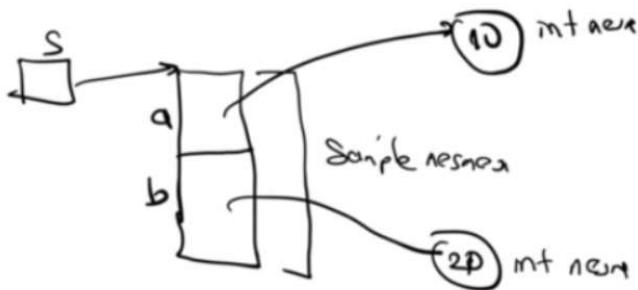
```
class Sample:
    def __init__(self):
        self.a = 10
        self.b = 20

s = Sample()
print(s.a, s.b)      # 10, 20

s.a = 100
s.b = 200
print(s.a, s.b)      # 100, 200
```

Bu örnekte biz bir `Sample` nesnesi yarattık. Bu `s` nesnesinin örnek özniteliklerinde (yani `s`'in parçalarında) değişiklikler yaptık. `s` bileşik bir nesnedir. Parçalardan (yani özniteliklerden) oluşmaktadır. `s`'nin değiştirilebilir (mutable) bir nesne olması `s`'nin parçaları üzerinde değişiklik yapılabileceği anlamına gelmektedir. Öte yandan bilindiği gibi değişkenler aslında her zaman adres tutarlar. Dolayısıyla da sınıfların öznitelikleri olan elemanlar da aslında adres tutmaktadır. Yukarıdaki örnekte `s` nesnesinin bellek organizasyonu aslında aşağıdaki gibidir:

`s = Sample()`

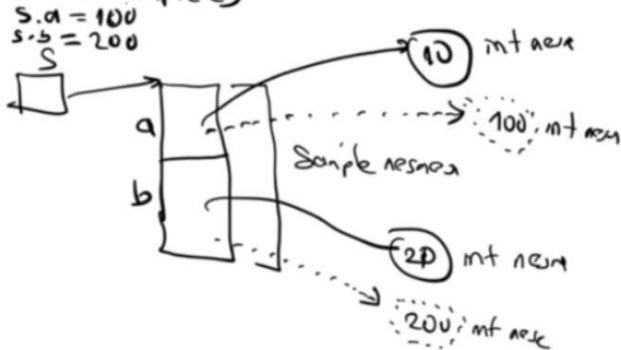


Bir kez daha ifade edersek, sınıfların örnek öznitelikleri aslında onların parçalarını oluşturan birer değişkendir. Bu değişkenler de birer adres tutmaktadır. Dolayısıyla yukarıdaki örnekte `s`'nin `a` ve `b` parçaları aslında adres tutan birer değişkendir. Biz `s` değiştirilebilir bir nesne olduğu için bu parçalar üzerinde değişiklikler yapabiliriz. Aslında arka planda bu parçaları oluşturan değişkenlerin içerisindeki adresleri değiştirmiştir oluruz. Örneğin:

`s = Sample()`

`s.a = 100`

`s.b = 200`

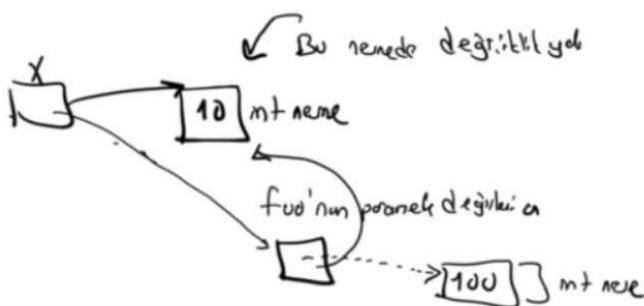


Değiştirilemez türleri fonksiyonlara parametre yoluyla geçirdiğimizde fonksiyonlar bu parametre değişkeninde değişiklik yaptığından bundan asıl nesne etkilenmemektedir. Örneğin:

```
def foo(a):
    print(a)          # 10
    a = 100

x = 10
foo(x)
print(x)          # 10
```

Bu durumu şekilsel olarak şöyle gösterebiliriz:



Burada `x`'in gösterdiği yerdeki nesnenin değişmediğine dikkat ediniz. Ancak listeler ve sözlükler ve bizim kendi sınıflarımız türünden nesneler değiştirilebilir nesnelerdir. Şimdi aynı işlemi kendi oluşturduğumuz sınıf nesnesi ile yapalım:

```

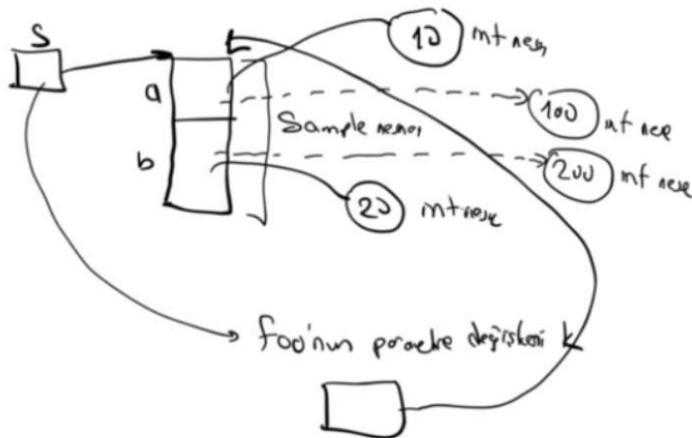
class Sample:
    def __init__(self, a, b):
        self.a = a
        self.b = b

def foo(k):
    print(k.a, k.b)      # 10 20
    k.a = 100
    k.b = 200

s = Sample(10, 20)
print(s.a, s.b)          # 10 20
foo(s)
print(s.a, s.b)          # 100 200

```

Bunu da şekilsel olarak şöyle gösterebiliriz:



Bu örnekte foo fonksiyonu içerisinde s'nin a ve b parçaları üzerinde yapılan değişikliğin fonksiyondan çıktılığında kaldığını dikkat ediniz.

Sınıfın Öznitelikleri ve Nesnenin Örnek Öznitelikleri

Sınıf içerisinde oluşturulan değişkenlere "sınıfın öznitelikleri (class attributes)" denilmektedir. Sınıfın öznitelikleri o sınıf türünden bir nesnenin elemanı değildir. Bir sınıfın içerisinde fakat metodların dışında oluşturulan değişkenler o sınıfın elemanıdır. Biz sınıf tanımlaması içerisinde oluşturulan değişkenleri (yani sınıfın özniteliklerini) hem metodlardan hem de dışarıdan sınıf ismi ve '.' operatörü ile kullanabiliriz. Ancak sınıf tanımlamasının içerisinde onlara doğrudan da erişebiliriz. Örneğin:

```

class Student:
    x = 10           # sınıf özniteliği, sınıfın bir elemanı, nesnenin değil
    print(x)         # sınıf özniteliği olan x kullanılıyor

    def __init__(self, name, no):
        self.name = name
        self.no = no

    def disp(self):
        print("İsim = {}, No = {}".format(self.name, self.no))
        print("x = {}".format(Student.x))     # x değil, Student.x

s = Student("Ali Serçe", 123)
s.disp()
print("x = {}".format(Student.x))  # x değil, Student.x

```

Burada sınıf bildiriminin içerisindeki x sınıfın bir özniteliğidir. Sınıf türünden bir nesnenin özniteliği değildir.

Dolayısıyla bu x değişkeninden toplamda bir tane vardır. Oysa self ile oluşturduğumuz değişkenler nesnenin içerisinde onun bir parçası olarak ayrıca bulunmaktadır. Yukarıdaki kodda __init__ içerisinde oluşturduğumuz name ve no değişkenleri (öznitelikleri) sınıfı ait değildir. Her yaratılan Sample nesnesinde ayrı bir name ve no elemanı olacaktır. Örneğin:

```
ali = Student("Ali Serçe", 123)
veli = Student("Veli Akkuş", 234)
```

Burada hem ali nesnesinin hem de veli nesnesinin ayrı name ve no elemanları (öznitelikleri) vardır. Halbuki sınıfın bir tek x elemanı vardır. Bu x elemanı yaratılmış olan nesnenin parçası değil, sınıf kavramının bir parçasıdır. Bu nedenle ona sınıf ismiyle erişilmektedir.

O halde bir sınıf aslında aynı elemanlardan oluşan nesneleri temsil eden bir kavramdır. Sınıfın metodları bu elemanlar üzerinde birtakım işlemler yapmaktadır.

Örneğin:

```
class Complex:
    def __init__(self, real = 0, imag = 0):
        self.real = real
        self.imag = imag

    def disp(self):
        print('{0}{1}i'.format(self.real, self.imag))

    def add(self, z):
        result = Complex()
        result.real = self.real + z.real
        result.imag = self.imag + z.imag

        return result

z1 = Complex(10, 20)
z2 = Complex(3, 4)
z3 = z1.add(z2)

z1.disp()
z2.disp()
z3.disp()
```

Burada Complex sınıfı bir karmaşık sayıyı temsil etmektedir. Her Complex nesnesinin real ve imag isimli iki parçası (yani örnek özniteliği) vardır. Sınıfın disp metodu bu karmaşık sayıyı yazdırma, add metodu ise iki karmaşık sayıyı toplayarak bize yeni bir karmaşık sayı vermektedir. add metodunun nasıl çağrıldığına dikkat ediniz:

```
z1.add(z2)
```

Metot içerisindeki self buradaki z1'i, metot içerisindeki z ise buradaki z2'yi temsil etmektedir. Sonuç olarak bu iki karmaşık sayı toplanmış ve bu metodun geri dönüş değeri olarak yeni bir karmaşık sayı elde edilmiştir.

Sınıfın Özniteliklerine Erişimin Diğer Bir Yolu

Bir sınıfın içerisinde yaratılan sınıf değişkenlerine sınıfın öznitelikleri (class attribute) deniyordu. Sınıfın özniteliklerine sınıfın dışında sınıf ismi ve nokta operatörüyle erişilebiliyordu. Örneğin:

```
class Sample:
    x = 10

print(Sample.x)      # 10
```

```
Sample.x = 20
print(Sample.x)      # 20
```

Ancak alternatif olarak (tipki C++ ve Java'da olduğu gibi) sınıf özniteliklerine ayrıca o sınıf türünden bir nesneyle de (instance) erişilebilmektedir. Bu sentaks yanlış anlaşılmalara kapı araladığı için pek tercih edilmez. Örneğin:

```
class Sample:
    x = 10

s = Sample()
print(s.x)          # 10
print(Sample.x)     # 10
```

Yani başka bir deyişle sınıf özniteliklerine sınıf ismi yerine o sınıf türünden bir değişkenle de erişilebilmektedir. Aynı isimli bir sınıf özniteliği ile bir örnek özniteliğinin bulunduğu durumda bu sınıf türünden bir değişkenle bu isim kullanıldığında örnek özniteliği anlaşılır. Ne de olsa sınıf değişkenine zaten sınıf ismiyle de erişilebilmektedir. Örneğin:

```
class Sample:
    x = 10
    def __init__(self, x):
        self.x = x

s = Sample(20)
print(s.x)          # 20
print(Sample.x)     # 10
```

Başa bir deyişle s bir sınıf türünden değişken olmak üzere s.x ifadesine atama yaptığımızda sınıfın aynı isimli bir sınıf özniteliği olsa bile biz her zaman örnek özniteliğine atama yapmış oluruz. Örneğin:

```
class Sample:
    x = 10

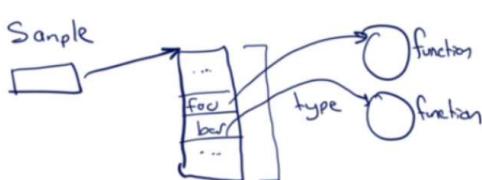
s = Sample()
print(s.x)          # buradaki x sınıf özniteliği olan x
s.x = 20            # artık biz örnek özniteliği olan x'e erişiyoruz
print(s.x)          # buradaki x artık örnek özniteliği olan x
```

Göründüğü gibi aynı isimli bir örnek özniteliği yaratılma kadar biz o sınıf türünden değişkenle sınıf özniteliğini kullanabilmekteyiz. Burada bir kez daha yinelersek bu durum bir karışıklık yarattığı için sınıf özniteliklerine o sınıf türünden nesneyle değil sınıf ismiyle erişmek tercih edilmelidir.

Sınıf İsimlerinin Anlamı

Sınıflar Python'da birer deyim statüsündendir. Python yorumlayıcısı bir sınıf tanımlamasını gördüğünde type isimli sınıf türünden bir nesne yaratır. Sınıfın elemanlarını bu sınıf nesnesinin içerisinde yerleştirir. Yani aslında Python'da sınıf isimleri o sınıfnın elemanlarının bulunduğu type isimli bir nesnenin adresini tutmaktadır. Örneğin:

```
class Sample:
    def foo(self):
        pass
    def bar(self):
        pass
```



Örneğin:

```
class Sample:
    def foo(self):
```

```

print('foo')

print(type(Sample))

```

Buradan şöyle çıktı elde edeceğiz:

```
<class 'type'>
```

Bir sınıfın öznitelikleri aslında type türünden nesnenin örnek öz nitelikleridir. Dolayısıyla biz bir sınıfa daha sonra da eleman ekleyebiliriz. Örneğin:

```

class Sample:
    def foo(self):
        print('foo')

def b(self):
    print('bar')

Sample.bar = b

s = Sample()
s.foo()
s.bar()

```

Bir sınıfın isminin type türünden bir nesneyi gösterdiğine fakat bir sınıf türünden sınıf türünden bir değişkenin o sınıf türünden bir nesneyi gösterdiğine dikkat ediniz. Örneğin:

```

class Sample:
    def foo(self):
        print('foo')

s = Sample()
print(type(Sample))
print(type(s))

```

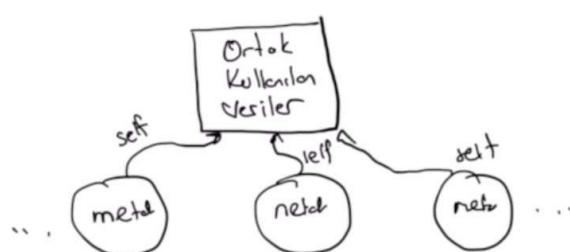
Bu programın çıktısı şöyle olacaktır:

```
<class 'type'>
<class '__main__.Sample'>
```

O Halde Bir Sınıf Gerçekte Nedir?

Bir sınıf belli bir konuda birtakım işlemleri yapan bir veri yapısı olarak düşünülebilir. Örneğin file sınıfı dosya işlemleri, str sınıfı yazı işlemleri, complex sınıfı karmaşık sayı işlemleri yapma iddiasındadır. Sınıflarda işi yapan elemanlar metotlardır. Bu metotlar ortak birtakım değişkenleri kullanırlar. Bu değişkenler nesnenin örnek öznitelikleridir. Her sınıf nesnesi yaratıldığında bu örnek özniteliklerden yeni birer kopya yaratılmış olur. Böylece her nesne aynı türden yeni bir örneği (instance) temsil eder.

Sınıfın metotları ortak veriler üzerinde işlem yapan fonksiyonlardır. Bu ortak veriler sınıfın örnek öznitelikleridir. O halde bir sınıf bir grup veri (data) ve onlar üzerinde işlem yapan fonksiyonlardan oluşan bir veri yapısıdır.



Metotların kullandığı ortak veriler metotların self parametresi ile onlara aktarılan nesnedeki elemanlardır.

Sınıflar Arasındaki İlişkiler

Daha önceden belirtildiği gibi bir proje nesne yönelimli olarak modelleneceğe önce proje içerisindeki kavramlar sınıflarla temsil edilir. Daha sonra o sınıflar türünden nesneler yaratılarak gerçek varlıklar elde edilir. Sınıflar arasında da birtakım ilişkiler söz konusu olabilmektedir. Örneğin hastane otomasyonunda Doktor sınıfı ile Hastane sınıfı, Doktor sınıfı ile Hasta sınıfı arasında ilişkiler vardır.

Sınıflar arasında dört ilişki biçimini tanımlanabilmektedir. Bunlar İçerme İlişkisi (Composition), Birleşme İlişkisi (Aggregation), Kalıtım İlişkisi (Inheritance) ve Çağrışım İlişkisi (Association) dir. Şimdi bu ilişki biçimlerini tek tek inceleyelim:

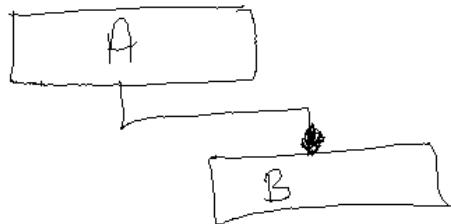
1) İçerme İlişkisi (Composition): Bir sınıf türünden nesne başka bir sınıf türünden nesnenin bir parçasını oluşturuyorsa bu iki sınıf arasında içerme ilişkisi vardır. Örneğin Araba ile Motor sınıfları arasında, İnsan ile Karaciğer sınıfları arasında içerme ilişkisi vardır. İçerme ilişkisi için iki koşulun sağlanması gereklidir:

- 1) İçerilen nesne tek bir nesne tarafından içерilmelidir.
- 2) İçeren nesneye içeren nesnenin ömürleri yaklaşık aynı olmalıdır.

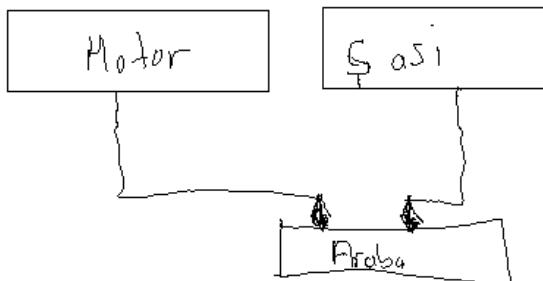
Tabii bu ölçütler tipik durumlar için düşünülmeliidir. Aksi takdirde biz doğadaki pek çok olguya tam olarak modelleyemeyiz. Örneğin insan öldüğünde karaciğeri başka bir insana takılabilmektedir. Fakat bu durum gözardı edilebilir.

Örneğin Oda ile Duvar sınıfları arasında içerme ilişkisi yoktur. Çünkü her ne kadar bunların ömürleri aynı ise de o duvar aynı zamanda yandaki odanın da duvarıdır. Satranç tahtası ile tahtanın kareleri arasında içerme ilişkisi vardır. Fakat satranç taşları ile kareler arasındaki ilişki içerme ilişkisi değildir. Saat ile akrep, yelkovan arasında içerme ilişkisi vardır. Fakat bilgisayar ile fare arasındaki ilişki içerme ilişkisi değildir. Benzer biçimde örneğin bir diyalog penceresi ile onun üzerindeki düğmeler arasında içerme ilişkisi vardır.

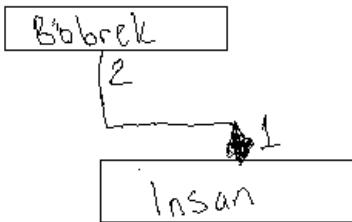
UML Sınıf diyagramlarında içerme ilişkisi içeren sınıf tarafından içi dolu bir baklavacıkla gösterilmektedir. Örneğin:



Burada B sınıfı A sınıfını içermektedir. Örneğin:

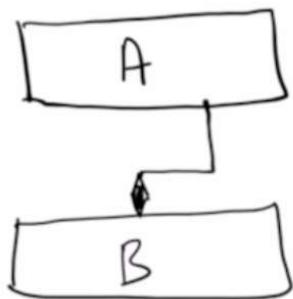


İçerme ilişkisi bire-bir olabileceği gibi bire-n de olabilir. Örneğin:



İçerme ilişkisine İngilizce "has a" ilişkisi de denilmektedir.

İçerme ilişkisi Python'da tipik olarak içeren sınıfın örnek özniteliğinde içeren sınıf türünden bir nesnenin tutulması biçiminde gerçekleştirilmektedir. Örneğin B sınıfı ile A sınıfı arasında bir içerme ilişkisi olsun. A sınıfı türünden bir nesnenin B sınıfı türünden bir nesnenin parçasını oluşturduğunu düşünelim.



```
class A:  
    pass  
  
class B:  
    def __init__(self):  
        self.a = A()  
    pass  
  
b = B()
```

Burada içerme ilişkisinin iki özelliğinin sağlandığına dikkat ediniz. Yaratılmış olan B nesnesinin ömrü ile B sınıfının `__init__` metodunda yaratılan A nesnesinin ömrü yaklaşık aynıdır. Aynı zamanda `__init__` metodunda yaratılan bu A nesnesi yalnızca B nesnesi tarafından kullanılmaktadır.

Örneğin:

```
class Square:  
    pass  
  
class Board:  
    def __init__(self):  
        self.squares = [[Square()] * 8 for i in range(8)]  
  
board = Board()
```

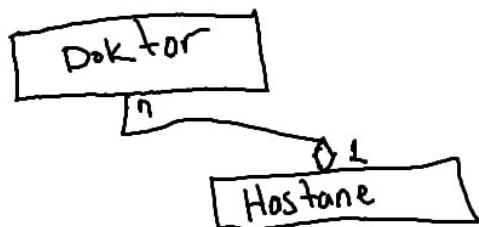
Burada Board (satranç tahtası) nesnesi yaratıldığından içerme ilişkisiyle 64 tane kareye sahip olmaktadır.

2) Birleşme ilişkisi (Aggregation): Birleşme ilişkisinde bir sınıf nesnesi başka türden bir sınıf nesnesini bünyesine katarak kullanmaktadır. Fakat kullanan nesneyle kullanılan nesnenin ömürleri aynı olmak zorunda değildir. Kullanan nesne başka nesneler tarafından da kullanılıyor olabilir. Örneğin, Araba sınıfıyla Tekerlek sınıfı arasında, Bilgisayar sınıfı ile Fare sınıfı arasında, Oda sınıfıyla Duvar sınıfı arasında, Ağaç sınıfıyla Yaprak sınıfı arasında, Hastane sınıfıyla Doktor sınıfı arasında böyle bir ilişki vardır. İçerme ilişkisine pek çok olgu birleşme ilişkisine uymaktadır.

Birleşme ilişkisi UML sınıf diyagramlarında kullanan sınıf tarafında içi boş bir baklavacık (diamond) ile gösterilir. Örneğin:

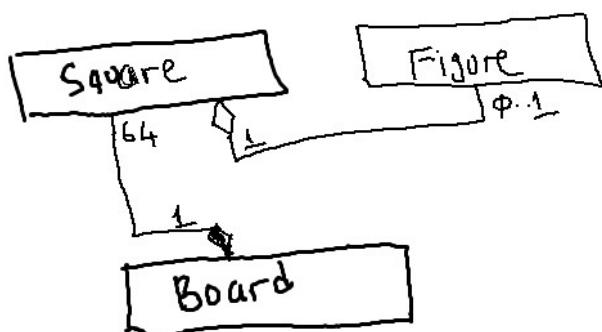


Örneğin:



Birleşme ilişkisine İngilizce "Holds a" ilişkisi de denilmektedir.

Örneğin bir satranç tahtasını modellemeye çalışalım. Tahta Board sınıfıyla tahtanın üzerindeki kareler de Square sınıfı ile temsil edilsin. Board sınıfı ile Square sınıfı arasında içerme ilişkisi vardır. Tahta üzerindeki taşlar da Figure sınıfıyla temsil ediliyor olsun. Bu durumda Square sınıfı ile Figure sınıfı arasında da birleşme ilişkisi söz konusudur. UML sınıf diyagramı şöyle oluşturulabilir:



Yine birleşme ilişkisi de bire bir olabileceği gibi bire çok da olabilir.

Birleşme ilişkisi Python'da sınıfın örnek özniteliğine nesne yaratılırken ya da yaratıldıktan sonra bir nesnenin atanması yoluyla gerçekleştirilmektedir. Bu atama işlemi bri metot yoluyla da yapılmaktadır. Örneğin:

```
class A:
    pass

class B:
    def __init__(self, a = None):
        self.a = a
        pass

    def seta(self, a):
        self.a = a

a = A()
b1 = B()
```

```
b2 = B()
```

```
b1.seta(a)  
b2.seta(a)
```

Burada bir A nesnesi iki farklı B nesnesi tarafından kullanılmaktadır. Örneğin:

```
class Doctor:  
    def __init__(self, name, specialty):  
        self.name = name  
        self.specialty = specialty  
  
    def disp(self):  
        print(f'{self.name}, {self.specialty}')  
  
class Hospital:  
    def __init__(self):  
        self.doctors = []  
  
    def add_doctor(self, doctor):  
        self.doctors.append(doctor)  
  
    def disp(self):  
        for doctor in self.doctors:  
            doctor.disp()  
  
hospital = Hospital()  
  
doctor1 = Doctor('Ali Serçe', 'İç Hastalıkları')  
doctor2 = Doctor('Medeni Demir', 'Psikiyatri')  
  
hospital.add_doctor(doctor1)  
hospital.add_doctor(doctor2)  
  
hospital.disp()
```

Bu örnekte bir hastane nesnesi bir liste biçiminde çalışan doktorları tutmaktadır. Hospital sınıfı ile Doctor sınıfı arasında birleşme ilişkisi vardır. Hospital nesnesi yaratıldıktan sonra doktorların eklenebileceğine (yani hastane ile doktorların ömürlerinin farklı olduğuna) dikkat ediniz. Biz yukarıdaki örnekte bir doktoru başka bir hastaneye de ekleyebildik.

Şimdi de satranç tahtasını oluşturalım. Yukarıda da belirttiğimiz gibi tahtada 64 kare vardır. Tahta ile kareler arasında içерme ilişkisi, karelerle taşlar arasında da birleşme ilişkisi bulunmaktadır. O halde tahtayı şöyle oluşturabiliriz:

```
class Figure:  
    def __init__(self, ftype, color):  
        self.ftype = ftype  
        self.color = color  
  
class Square:  
    def __init__(self, color):  
        self.color = color  
        self.figure = None  
  
    def set_figure(self, figure):  
        self.figure = figure  
  
    def get_figure(self):  
        return self.figure  
  
class Board:  
    def __init__(self):
```

```

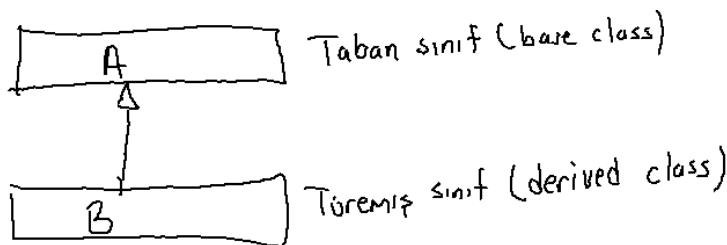
    self.squares = [[Square('Beyaz' if (i + k) % 2 == 0 else 'Siyah') for i in range(8)]
for k in range(8)]
    self.squares[0][0].figure = Figure('Kale', 'Siyah')
    self.squares[0][1].figure = Figure('At', 'Siyah')
    # ...
board = Board()

```

Karelerin renkleri vardır ve onlar üzerinde taşlar olabilmektedir. Taşların ise türleri (şah, vezir, kale, fil, at, piyon) ve renkleri vardır. Tahtanın sol üst köşesi (satrançtaki a8 karesi) squares matrisinde [0][0] karesine karşılık gelmektedir.

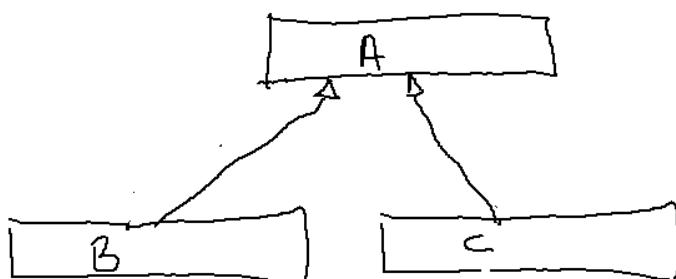
3) Kalıtım (Türetme) İlişkisi (Inheritance): Türetme mevcut bir sınıfa ona dokunmadan ekleme yapmak anlamına gelmektedir. Elimizde bir A sınıfı bulunuyor olsun. Biz buna birtakım elemanlar eklemek isteyelim. Fakat A'nın kaynak kodu elimizde olmayabilir ya da onu bozmak istemeyebiliriz. Bu durumda A sınıfından bir B sınıfı türetiriz. Eklemeleri B'ye yaparız. Böylece B sınıfı hem A sınıfı gibi kullanılır hem de fazlalıklara sahip olur.

Türetme işleminde işlevini genişletmek istediğimiz asıl sınıfa taban sınıf (base class) denilmektedir. Ondan türettiğimiz yani eklemeleri yaptığımız sınıfa da türemiş sınıf (derived class) denir. UML sınıf diyagramlarında türetme ilişkisi türemiş sınıftan taban sınıfa çekilen içi boş bir okla belirtilmektedir. Örneğin:



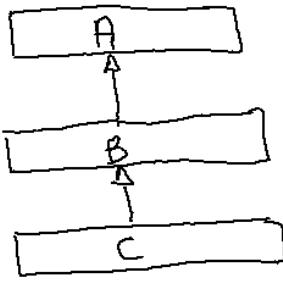
Burada B sınıfı hem A sınıfı gibi davranışları hem de fazlalıkları vardır. Türetme ilişkisine İngilizce "is a" ilişkisi denilmektedir. (B bir çeşit A'dır fakat fazlalıkları da vardır.)

Bir sınıf birden fazla sınıfın taban sınıfı durumunda olabilir. Örneğin:



Burada B ile C arasında bir ilişki yoktur. B de C de A'dan türetilmiştir. Yani B sınıfı türünden bir nesne hem B gibi hem de A gibi kullanılabilir. C sınıfı türünden bir nesne de hem C gibi hem de A gibi kullanılabilir. Ancak B ile C arasında böyle bir ilişki yoktur.

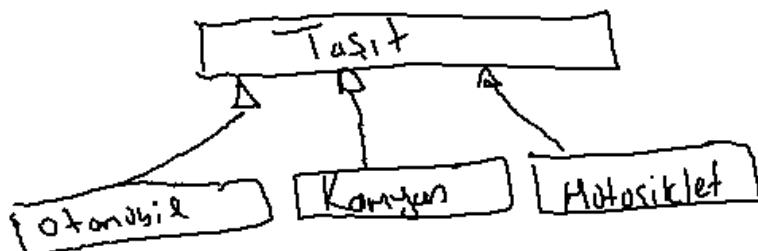
Türemiş bir sınıftan yeniden türetme yapılabilir. Örneğin:



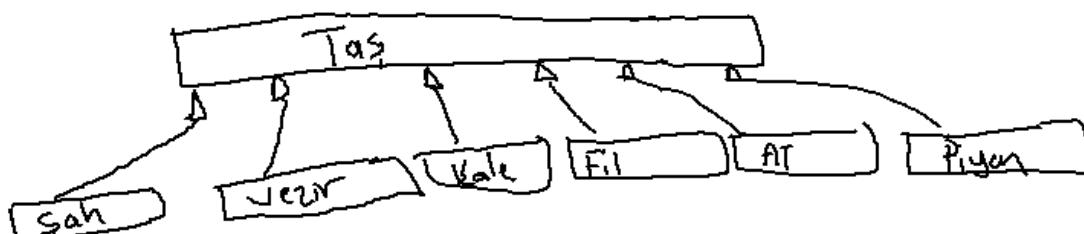
Burada C sınıfı hem B gibi hem de A gibi kullanılabilir. Ancak fazlaları da vardır.

Türetmeye çeşitli örnekler verebiliriz:

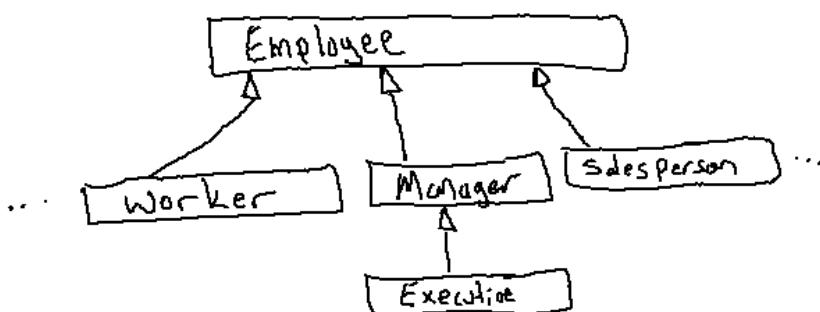
- Örneğin tüm taşıtların ortak birtakım özellikleri Taşıt sınıfında toplanabilir (plakası, trafiğe çıkış tarihi, motor gücü vs.). Bundan Otomobil, Kamyon, Motosiklet gibi sınıflar türetilmişdir. Otomobil de bir taştır (is a ilişkisi), kamyon da, motosiklet de birer taştır. Fakat kamyonda olan özellikler otomobilde olmamıştır:



- Satranç taşlarının ortak özelliklerini Taş sınıfında toplayıp ondan Şahı Vezir, Kale, Fil, At ve Piyon sınıflarını türetilmişdir. Örneğin:

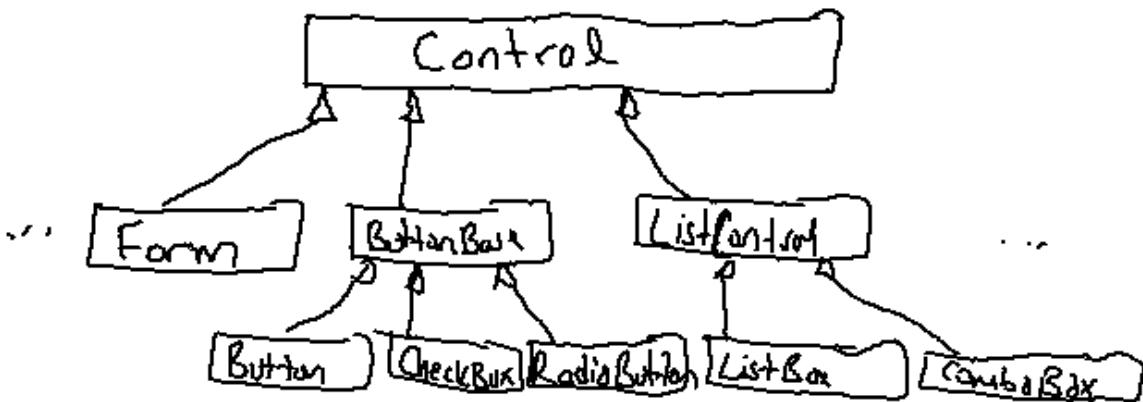


- Bir iş yerinde çalışanları sınıflarla temsil edebiliriz. Tüm çalışanların ortak özellikleri Employee isimli bir sınıfı toplanabilir. İşçiler bu sınıfından türetilmiş Worker sınıfıyla, yöneticiler Manager sınıfıyla temsil edilebilir. Üst düzey yöneticiler de bir çeşit yöneticidir. Bunlar da Executive isimli bir sınıfı temsil edilebilirler. Bu durumda Executive sınıfının da Manager sınıfından türetilmesi uygun olur:



Türetmeye neden gereksinim duyulmaktadır? Programlamadaki temel prensiplerden biri kod ve veri tekrarını engellemektir. Örneğin bir kod parçası aynı programda ikinci kez kopyala-yapıştır yapılmamalıdır. Bunu engellemek için en normal yöntem o kod parçasını bir metoda yerleştirmek ve o metodu çağrılmaktır. Yani programımız içerisinde aynı

kod parçalarının farklı yerde bulunması durumundan şüphelenmeliyiz ve bunu gidermeye çalışmalıyız. Böylece hem programımız daha az yer kaplar hale gelir, hem daha algılanabilir olur hem de hataların analiz edilmesi ve düzeltilmesi daha kolaylaşır. (Örneğin o kod parçasında bir hata olsa ve onu düzeltmeye çalışsa pek çok yerdeki kopyalarını düzeltmek yerine tek bir kopyasını düzeltmek daha sorunsuzdur.) İşte NYPT'de iki sınıfın içerisinde ortak elemanlar varsa (veri elemanları, property'ler ve metodlar) bunlar ortak bir taban sınıfı toplanmalı ve ondan türetme yapılarak bu iki sınıf oluşturulmalıdır. Örneğin GUI uygulamalarında ekranda bağımsız olarak kontrol edilebilen dikdörtgensel alanlara pencere (window) denilmektedir. Düğmeler, edit alanları, listeleme kutuları, ana pencereler hep birer pencerelerdir. .NET Form kütüphanesinde tüm pencerelerin ortak özellikleri taban bir Control sınıfında toplanmıştır. Diğer sınıflar bundan türetilmiştir:



Button, CheckBox ve RadioButton pencerelerinin de birtakım ortak özellikleri vardır. Bu özellikler de ButtonBase sınıfında toplanmıştır. Benzer biçimde ListBox ve ComboBox sınıflarının ortak elemanları da ListControl sınıfında toplanmış durumdadır.

Bir türetme şemasında yukarıda çıkıldıkça genelleşme, aşağıya inildikçe özelleşme oluşur.

Bir sınıfın birden fazla taban sınıfı olması durumu ilginç ve özel bir durumdur. Buna çoklu türetme (multiple inheritance) denilmektedir. C# ve Java'da çoklu türetme özelliği yoktur. Dolayısıyla bu dillerde bir sınıfın tek bir taban sınıfı olabilir. Fakat C++'ta çoklu türetme vardır. Örneğin hava taşıtları bir sınıfta, deniz taşıtları başka bir sınıfta temsil edilebilir. Aircraft sınıfı bunlardan çoklu türetilen bir sınıf:



Türetme ilişkisi programlama dillerinde ayrı ve özel bir sentaksla gerçekleştirilmektedir. Biz Python'da türetme işlemlerinin nasıl yapıldığını sonraki bölümde ayrı bir başlık halinde inceleyeceğiz.

4) Çağrışim İlişkisi (Association): Bu ilişki biçiminde bir sınıf bir sınıfı bünyesine katarak değil, yüzeysel biçimde, bir ya da birkaç metodunda kullanıyor durumdadır. Örneğin Taksi ile Müşteri arasında ciddi bir ilişki yoktur. Taksi müsteriyi alır ve bir yere bırakır. Halbuki Taksi ile şoförü arasında önemli bir ilişki vardır. İşte Taksi ile Müşteri ilişkisi çağrışim ilişkisi iken, Taksi ile Şoför arasındaki ilişki birleşme (aggregation) ilişkisidir. Benzer biçimde Hastane sınıfı ReklamŞirketi sınıfını yalnızca reklam yaparken kullanmaktadır. Bunların arasında da çağrışim ilişkisi vardır. Çağrışim ilişkisi UML sınıf diyagramlarında kullandığı sınıfın kullanıldığı sınıfın eklenmesiyle gösterilir. Örneğin:



Python'da çağrılmış ilişkisini oluştururken kullanılan nesneyi kullanan nesnenin örnek özniteliklerinde saklamayız. Nesne kullanan sınıfın birkaç metodunda genellikle parametre yoluyla kullanılır. Örneğin:

```
class AdvertisingCompany:
    pass

class Hospital:
    def advertise(self, adcompany):
        pass
    pass
```

Python'da Türemme İşlemleri

Python'da türemme işleminin genel biçimi şöyledir:

```
class <türemiş sınıf ismi>(<taban sınıf ismi>):
    <sınıf elemanları>
```

Örneğin:

```
class A:
    def foo(self):
        print('foo')

    def bar(self):
        print('bar')

class B(A):
    def tar(self):
        print('tar')

b = B()
b.foo()
b.bar()
b.tar()
```

Türemiş sınıfından bir değişkenle biz hem türemiş sınıfın hem de taban sınıfın elemanlarına erişebiliriz. Yani biz dışarıdan türemiş sınıfından bir değişkenle ya da türemiş sınıfın içerisindeki self parametresi ile nasıl kendi sınıfımızın elemanlarına erişiyorsak aynı biçimde taban sınıfın elemanlarına da erişebiliriz. Tabii bu elemanların daha önce oluşturulmuş olması gereklidir. Python'da türemiş sınıfından bir nesne henüz yaratıldığında henüz bu nesnenin hiçbir örnek özniteliği yoktur. Yaratılan nesnenin örnek öznitelikleri tipik olarak taban sınıf metodlarıyla (örneğin taban sınıfın dunder init metodıyla) ve türemiş sınıf metodlarıyla (örneğin türemiş sınıfın dunder init metodu) oluşturulur. Örneğin:

```
class A:
    def setA(self, a):
        self.a = a

    def dispA(self):
        print(self.a)
```

```

class B(A):
    def setB(self, b):
        self.b = b

    def dispB(self):
        print(self.a, self.b)

b = B()
b.setA(10)
b.setB(20)

b.dispA()
b.dispB()

```

Bu örnekte setA metodu B nesnenin a örnek öznitelğini, setB metodu da b örnek öznitelliğini oluşturmaktadır. Bu örnekte türemiş sınıfın dispB metodunda taban sınıfın a elemanı da kullanılmıştır. Tabii biz bu SetA metodunu çağrımadan dispA ya da dispB metodunu çağırılmış olsaydık henüz a örnek özniteligi yatailmamış olacağı için exception oluşurdu. sorun çıkacaktır. Örneğin:

```

b = B()
b.setB(20)
b.dispA()      # error

```

Biz genellikle sınıfların örnek özniteliklerini `__init__` metodlarında ilk kez oluştururuz. Taban sınıfın `__init__` metodunda taban sınıfın öznitelikleri, türemiş sınıfın `__init__` metodunda da türemiş sınıfın öznitelikleri oluşturulmaktadır. O halde biz türemiş sınıf türünden bir nesne yarattığımızda türemiş sınıfın `__init__` metodu çağrılmak için bizim de türemiş sınıfın `__init__` metodunda taban sınıfın `__init__` metodunu çağrırmamız uygun olur. İşte bu çağrıma şöyle yapılmaktadır:

```
<taban sınıf ismi>.__init__(self, ...)
```

Pek çok dilde türemiş sınıfın başlangıç metodu (constructor) taban sınıfın başlangıç metodunu otomatik olarak çağrımaktadır. Python'da böyle bir otomatik çağrıma yoktur. Örneğin:

```

class A:
    def __init__(self, a):
        self.a = a

    def dispA(self):
        print(self.a)

class B(A):
    def __init__(self, a, b):
        A.__init__(self, a)
        self.b = b

    def dispB(self):
        print(self.b)

b = B(10, 20)
b.dispA()
b.dispB()

```

Burada türemiş sınıfın `__init__` metodu taban sınıfın `__init__` metodunu çağrımıştır. Bu çağrıma biçiminin şöyle yapıldığına dikkat ediniz:

```
A.__init__(self, a)
```

Eğer biz çağrımayı böyle değil normal yöntemle yapsaydık bu durumda türemiş sınıfın kendi `__init__` metodu yine kendini çağrılmış olurdu. Bu çağrıma biçiminin de metodlar için geçerli olduğunu anımsayınız.

Taban sınıfın `__init__` metodunu çağrımanın diğer bir yolu da super fonksiyonunu kullanmaktadır. super fonksiyonu Python'ın versiyonlarında biraz değişikliklere uğratılmıştır. Bu fonksiyon tipik olarak iki parametre almaktadır. Birinci parametre taban sınıfı bulunacak türemiş sınıfın ismidir. İkinci parametre de türemiş sınıf türünden nesneyi temsil eden bir değişkendir. Bu ikinci parametre tipik olarak `self` biçiminde geçirilir. Örneğin B sınıfının `__init__` metodunda A sınıfının `__init__` metodunun diğer bir çağrılmış biçimde şöyledir:

```
super(B, self).__init__(...)
```

Tabii artık `__init__` metoduna ayrıca `self` parametresinin geçilmediğine dikkat ediniz. Örneğin:

```
class A:  
    def __init__(self, a):  
        self.a = a  
  
    def dispA(self):  
        print(self.a)  
  
class B(A):  
    def __init__(self, a, b):  
        super(B, self).__init__(a)      # A.__init__(self, a)  
        self.b = b  
  
    def dispB(self):  
        print(self.b)  
  
b = B(10, 20)  
b.dispA()  
b.dispB()
```

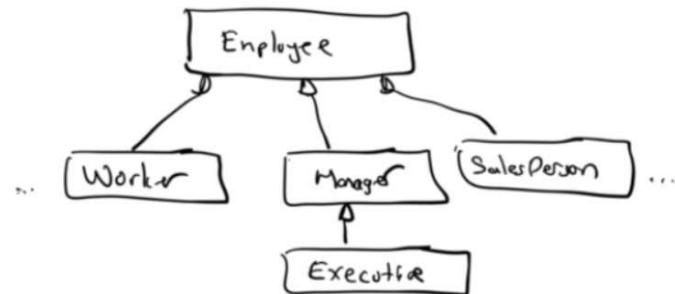
Türemiş sınıf `__init__` metodu içerisinde taban sınıfın `__init__` metodunun iki farklı biçimde çağrılmmasını şuna benzetebiliriz: A sınıfı türünden bir `a` değişken ile A sınıfının `foo` metodunun çağrılmamasının iki yolu vardır:

```
a.foo(...)  
A.foo(a, ...)
```

İşte türemiş sınıfta taban sınıfın `__init__` metodunun çağrıması da aynı biçimdedir.

```
super(B, self).__init__(...)  
A.__init__(self, ...)
```

Şimdi türetmeye daha somut bir örnek verelim. Bir personel takip programında işletmede çalışan kişilerin sınıflarla temsil edildiğini düşünelim. Her çalışanın ortak olan birtakım bilgileri taban sınıfta toplanabilir. Sonra bu taban sınıftan türetmeler yapılarak her çalışan grubunun kendi bilgileri ve işlevleri oluşturulabilir.



```

class Employee:
    def __init__(self, name, address):
        self.name = name
        self.address = address

    def disp_employee(self):
        print('Adı Soyadı: {}, Adres: {}'.format(self.name, self.address))

class Worker(Employee):
    def __init__(self, name, address, shift):
        super(Worker, self).__init__(name, address) # Employee.__init__(self, name, address)
        self.shift = shift

    def disp_worker(self):
        print('Adı Soyadı: {}, Adres: {}, Vardiya: {}'.format(self.name, self.address,
self.shift))

e = Worker('Ali Serçe', 'Maslak', 'Sabah')
e.disp_worker()

```

Göründüğü gibi burada Worker sınıfı Employee sınıfından türetilmiştir. Türetmeye İngilizce "is a" ilişkisi (is a relationship) de denilmektedir. Yani bir sınıf mantıksal bakımdan başka bir sınıfı kapsıyorsa fakat ondan fazlalarını da varsa bu iki sınıf ayrı ayrı değil türetme yapılarak oluşturulmalıdır. Bu örnekte işçi de bir çeşit çalışandır. O halde Worker sınıfı Employee sınıfından türetilmiştir. Bir Worker nesnesi yaratıldığında onun __init__ metodunda Employee'nin __init__ metodu çağrılmış, böylece onun name ve address isimli öznitelikleri oluşturulmuştur. Türemiş sınıfın taban sınıfın elemanlarına sanki kendi elemanları olmuş gibi doğrudan erişebildiğine dikkat ediniz. Yukarıdaki örneğin daha geniş bir halini şöyle verebiliriz:

```

class Employee:
    def __init__(self, name, address):
        self.name = name
        self.address = address

    def disp_employee(self):
        print('Adı Soyadı: {}, Adres: {}'.format(self.name, self.address))

class Worker(Employee):
    def __init__(self, name, address, shift):
        super(Worker, self).__init__(name, address) # Employee.__init__(self, name, address)
        self.shift = shift

    def disp_worker(self):
        print('Adı Soyadı: {}, Adres: {}, Vardiya: {}'.format(self.name, self.address,
self.shift))

class Manager(Employee):
    def __init__(self, name, address, department):
        super(Manager, self).__init__(name, address) # Employee.__init__(self, name,
address)
        self.department = department

    def disp_manager(self):
        print('Adı Soyadı: {}, Adres: {}, Departman: {}'.format(self.name, self.address,
self.department))

class Executive(Manager):
    def __init__(self, name, address, department, region):
        super(Executive, self).__init__(name, address, department) # Manager.__init__(self,
name, address, department)
        self.region = region

```

```

def disp_executive(self):
    print('Adı Soyadı: {}, Adres: {}, Departman: {}, Bölge: {}'.
          format(self.name, self.address, self.department, self.region))

```

```

e = Executive('Salih Bulut', 'Kazlıçeşme', 'Üretim', "Ortadoğu")
e.disp_executive()

```

Bu örnekte sınıfın örnek özniteliklerini yazdırın disp metodlarına aynı isimleri de verebilirdik. Ayrıca türemiş sınıfın disp metodları taban sınıfın disp metodlarını çağıracak biçimde de kod düzenlenebilirdi. Tabii taban ve türemiş sınıflarda aynı isimli metodların bulunduğu durumlarda taban sınıfındaki aynı isimli metodun sınıf ismiyle ya da super fonksiyonuyla çağrılmaması gerektiğine dikkat ediniz:

```

class Employee:
    def __init__(self, name, address):
        self.name = name
        self.address = address

    def disp(self):
        print('Adı Soyadı: {}'.format(self.name))
        print('Adres: {}'.format(self.address))

class Worker(Employee):
    def __init__(self, name, address, shift):
        super(Worker, self).__init__(name, address) # Employee.__init__(self, name, address)
        self.shift = shift

    def disp(self):
        super(Worker, self).disp() # Employee.disp(self)
        print('Vardiya: {}'.format(self.shift))

class Manager(Employee):
    def __init__(self, name, address, department):
        super(Manager, self).__init__(name, address) # Employee.__init__(self, name,
address)
        self.department = department

    def disp(self):
        super(Manager, self).disp() # Employee.disp(self)
        print('Departman: {}'.format(self.department))

class Executive(Manager):
    def __init__(self, name, address, department, region):
        super(Executive, self).__init__(name, address, department) # Manager.__init__(self,
name, address, department)
        self.region = region

    def disp(self):
        super(Executive, self).disp() # Manager.disp(self)
        print('Bölge: {}'.format(self.region))

e = Executive('Salih Bulut', 'Kazlıçeşme', 'Üretim', "Ortadoğu")
e.disp()

```

object Sınıfı

Python'da da tıpkı Java ve C#'ta olduğu gibi her sınıf doğrudan ya da dolaylı olarak tepedeki bir object isimli sınıfından türetilmiş durumdadır. Biz bir sınıfı hiçbir sınıfın türetmesek bile Python yorumlayıcısı onun object sınıfından türetildiğini varsayımaktadır. Örneğin:

```

class Sample:
    pass

```

bildirimi ile,

```
class Sample(object):
    pass
```

bildirimi eşdeğerdir. Yani her zaman türetme şemasının tepesinde object sınıfı vardır. Ancak biz kurs notlarımızda türetme şemalarını çizerken tepedeki object sınıfını belirtmeyeceğiz.

Pekiye object sınıfının elemanları nelerdir? Aslında object sınıfının herkesin faydalanaceği örnek öznitelikleri yoktur. Ancak bazı işlevsellikleri arka planda sağlayan önemli metodları vardır. Bunlardan bazıları bazı konularda ele alınacaktır.

Sınıfların __mro__ Öznitelikleri

Bir sınıfın türetme hiyerarşisi sınıfın __mro__ isimli özniteliği ile elde edilebilir. Örneğin:

```
class A:
    pass

class B(A):
    pass

class C(B):
    pass

print(C.__mro__)
```

Bu programdan şöyle bir çıktı elde ederiz:

```
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Sınıfların __mro__ öznitelikleri her biri type nesnelerinden oluşan bir demet vermektedir. __mro__ özniteliğinin nesneye (instance) ilişkin olmadığına sınıfa ilişkin olduğuna dikkat ediniz. Biz de bu özniteliği sınıf ismiyle kullandık.

Örneğin:

```
class A:
    pass

class B:
    pass

class C(A, B):
    pass

print(C.__mro__)
```

Elde edilen çıktı şöyledir:

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

Sınıfların __mro__ öznitelikleri aynı zamanda standart inspect modülündeki getmro fonksiyonuyla da elde edilebilir. getmro fonksiyonu parametre olarak bizden sınıf ismini (yani sınıf bilgilerinin bulunduğu type nesne referansını) alır ve mro bilgilerinden oluşan bir demet verir. Örneğin:

```
import inspect

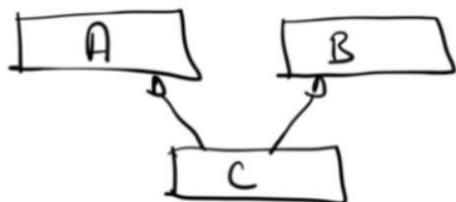
print(inspect.getmro(C))
```

Tabii getmro fonksiyonu aslında sınıfın `__mro__` özniteliğine geri dönmektedir. Şöyleder yazılmıştır:

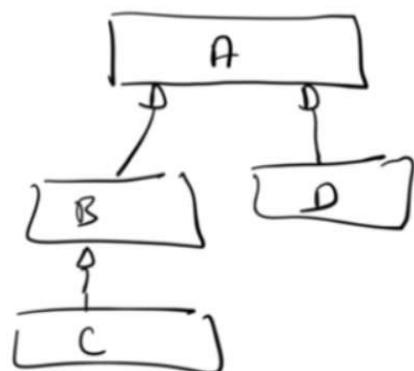
```
def getmro(cls):
    return cls.__mro__
```

Python'da Çoklu Türetme (Multiple Inheritance)

Bir sınıfın birden fazla taban sınıfına sahip olması durumuna çoklu türetme (multiple inheritance) denilmektedir. Java ve C# gibi bazı dillerde çoklu türemeye izin verilmemiştir. Ancak C++ gibi, Object Pascal gibi dillerde çoklu türetme özelliği vardır. Python'da da çoklu türetme kullanılabilmektedir.



Burada C sınıfının iki taban sınıfı vardır: A ve B. Bu durum çoklu türetme demektir. Halbuki aşağıdaki örnekte her sınıfın tek bir taban sınıfı vardır:



Çoklu türemeye doğada fazlaca rastlanmamaktadır. Python'da çoklu türetme için parantez içerisindeki sınıf listesine birden fazla sınıfın ismi yazılır. Örneğin:

```
class A:
    pass

class B:
    pass

class C(A, B):
    pass
```

Eğer bir nesne iki farklı nesnenin özelliklerini barındırıysa ve ayrıca kendine özgü özellikleri de varsa bu durum çoklu türemeyi çağrıştırmaktadır. Örneğin hava taşıtları bir sınıfta deniz taşıtları başka bir sınıfta temsil edilmiş olsun. Hem havada hem denizde giden taşıtlar bu iki sınıftan çoklu türelerek oluşturulabilirler. Çoklu türetmede isim araması için bazı kurallar vardır. Bunlar ilerde ele alınacaktır. Örneğin:

```
class A:
    def foo(self):
        print('foo')
class B:
    def bar(self):
        print('bar')
```

```

class C(A, B):
    def tar(self):
        print('tar')

c = C()
c.foo()
c.bar()
c.tar()

```

Taban ve Türemiş Sınıflarda Aynı İsimli Elemanların Bulunması Durumu

Hem taban hem de türemiş sınıfta aynı isimli örnek öznitelikleri ve metodlarının bulunduğu bir durumda biz türemiş sınıfından bir değişkenle bu öznitelikleri ya da metotları kullanırsak ne olacaktır? Önce aynı isimli örnek özniteliklerin bulunması durumuna bakalım.

Python'da diğer pek çok dilde olduğu gibi taban sınıfın veri elemanlarıyla türemiş sınıfın veri elemanları farklı bir blok olarak tutulmaktadır. Aslında Python'da türemiş sınıf nesnesinin bir taban sınıf kısmı yoktur. Türemiş sınıf nesnesi öznitelik anlamında taban sınıf nesnesini kapsıyor durumda da değildir. Bu yüzden taban sınıfta biz bir örnek özniteligi yaratlığımızda bu yaratımı yaptığımız metot hangi türden değişkenle çağrılmışsa zaten o örnek özniteligi de o değişkenin bir parçası olur. Örneğin taban sınıf metodu türemiş sınıfından bir değişkenle çağrılmışsa taban sınıfındaki öznitelikler türemiş sınıf nesnesinin öznitelikleri olacaktır. Örneğin:

```

class A:
    def __init__(self):
        print('A __init__ called: {}'.format(type(self)))
        self.x = 10

    def dispA(self):
        print('A.Disp: {}'.format(self.x))

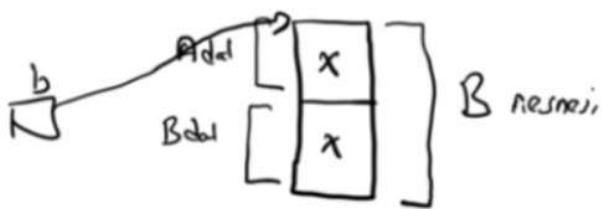
class B(A):
    def __init__(self):
        print('B __init__ called: {}'.format(type(self)))
        super(B, self).__init__()
        self.x = 20

    def dispB(self):
        print('B.Disp: {}'.format(self.x))

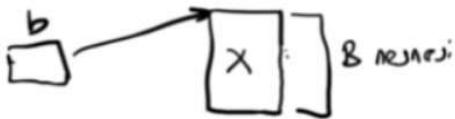
b = B()
b.dispB()
b.dispA()

```

Benzer bir durum C++, Java ve C#'ta olsaydı bu dillerde taban sınıfın veri elemanları (Python terminolojisiyle örnek öznitelikleri) ile türemiş sınıfın veri elemanları aynı isimli olsalar bile farklı elemanlar olurlardı. Bu dillerdeki durumu şekilsel olarak şöyle gösterebiliriz:



Oysa Python'da taban ve türemiş sınıfın x isimli ayrı örnek öznitelikleri (veri elemanları) yoktur. Bir tane x vardır. O da B'nin x örnek özniteligidir. Bunu şekilsel olarak şöyle gösterebiliriz:



Burada görüldüğü gibi türemede aslında bir data içermesi söz konusu değildir.

Pekiyi taban sınıf ve türemiş sınıflarda aynı isimli metodlar bulunabilir mi? Bulunursa ne olur? İşte taban sınıftaki bir metodun türemiş sınıfta yeniden yazılması durumuna "taban sınıftaki metodun türemiş sınıfta override edilmesi" denilmektedir. Override etme durumunda eğer metod taban sınıfından değişkenle çağrılmışsa taban sınıftaki metod, türemiş sınıfından değişkenle çağrılmışsa türemiş sınıftaki metod çağrılmaktadır. Örneğin:

```
class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
        print('B.foo')

a = A()
a.foo()      # A.foo çağrıılır

b = B()
b.foo()      # B.foo çağrıılır
```

Burada hem taban sınıfta bir foo metodu hem de türemiş sınıfta bir foo metodu vardır. Normal olarak türemiş sınıf türünden değişkenlerle biz taban sınıfın metodlarını da çağrılabılır. Ancak burada taban sınıftaki metodla türemiş sınıftaki metod aynı isimlidir. İşte bu durumda türemiş sınıf türünden değişkenle bu aynı isimli foo metodunu çağrıdığımızda türemiş sınıfın metodу çağrılacaktır. Fakat yine de biz istersek türemiş sınıf türünden bir değişkenle taban sınıftaki foo metodunu çağrılabılır. Bunu yapmanın iki yolu vardır. Birincisi çağrıma işlemini sınıf ismi belirterek yapmak, ikincisi super fonksiyonunu kullanmak. Örneğin:

```
class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
        print('B.foo')

b = B()
A.foo(b)          # A.foo çağrıılır
super(B, b).foo() # A.foo çağrıılır
```

super fonksiyonu izleyen bölümlerde ayrı bir başlık halinde ele alınacaktır.

Overload ve Override Kavramları

Nesne Yönelimli Programlama teknlığında (NYPT) birbirine fonetik olarak benzeyen iki önemli kavram vardır: "overload" ve "override". Bu iki kavram da fonksiyonlar ve metodlar için kullanılmaktadır. "Overload" terimi "aynı isimli farklı parametrik yapılara ilişkin fonksiyonların ya da metodların aynı faaliyet alanı içerisinde bulunması" durumuna denilmektedir. Yani örneğin C++, Java ve C# gibi dillerde aynı sınıf içerisinde aynı isimli birden fazla metod (fonksiyon) bulunabilmektedir. Ancak bunların parametrik yapılarının farklı olması gerekmektedir. Python'da bu anlamda bir overload kavramı yoktur. Yani aynı isimli birden fazla fonksiyon ya da metod ne olursa olsun beraber bulunamazlar. (Tabii farklı faaliyet alanlarında, örneğin farklı sınıflarda bulunabilirler.) Python'da zaten fonksiyonlar ve metodlar diğer değişkenler gibi birer referans belirtmektedir. Yani fonksiyon ya da metod isimleri aslında bu fonksiyon ya da metodları içeren nesnelerin adreslerini tutan değişkenlerdir. Örneğin:

```

def foo():
    print('foo')

print(foo)           # <function foo at 0x0000018153C82EA0>
print(type(foo))    # <class 'function'>

```

Bu durumu şekilsel olarak da şöyle gösterebiliriz:



Dolayısıyla biz aynı isimli yeniden bir fonksiyon tanımladığımızda Python'da aslında bu değişken farklı bir nesneyi gösteriyor duruma getirilmiş olur.

```

def foo():
    print('foo')

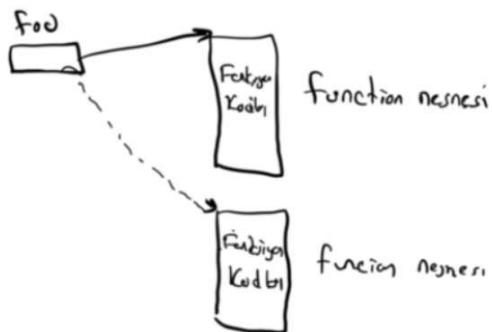
foo()      # foo

def foo():
    print('other foo')

foo()      # other foo

```

Bunu da şekilsel olarak şöyle gösterebiliriz:



Yani Python'da aynı isimli foo fonksiyonunu programın değişik yerlerinde tanımlamanın bir sakıncası yoktur. Aslında fonksiyon isimleri "function" isimli türden nesneleri gösteren birer referans (pointer) belirtmektedir. Yukarıdakş durumun aşağıdaki durumdan hiçbir farkı yoktur:

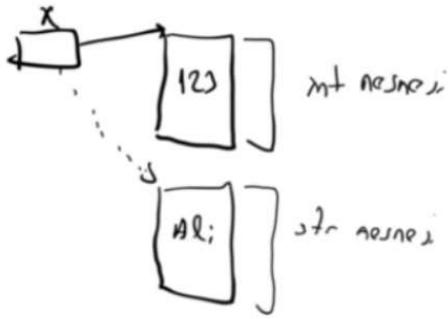
```

x = 123
print(x)      # 123

x = 'Ali'
print(x)      # Ali

```

Şekilsel olarak gösterirsek:



Mademki Python'da fonksiyon isimleri aslında fonksiyonları gösteren birer değişkendir. O halde bu isimler de birbirlerine atanabilirler. Örneğin:

```
def foo():
    print('foo')

bar = foo

foo()
bar()
```

Bu durumu şekilsel olarak da şöyle gösterebiliriz:



Python'da fonksiyonlar da "birinci sınıfı vatandaş (first class citizen)" durumundadır. Yani onlar da fonksiyonlara parametre yoluyla aktarılabilir. Örneğin:

```
def foreach(l, f):
    for x in l:
        f(x)

def foo(x):
    print('foo: {}'.format(x))

foreach([1, 2, 3, 4, 5], foo)
foreach([1, 2, 3, 4, 5], print)
```

Fonksiyonların parametre yoluyla fonksiyonlara normal birer değişken gibi aktarılabilidine dikkat ediniz.

Python'da sınıfın metodlarını bir çeşit sınıf özniteliği olarak düşünmeliyiz. Yani bir sınıfın metodu aslında sınıfın içerisinde tanımlanmış bir değişken gibidir. Bir sınıf içerisinde de Python'da aynı isimli metodlar bulunursa aslında bunlar overload edilmiş olmamaktadır. Yine son aynı isimli metod geçerli metod olarak kullanılacaktır. Çünkü sınıflar Python'da birer deyim statüsündedir. Örneğin:

```
class Sample:
    def foo(self):
        print('parametresiz foo')

    def foo(self, a):
        print('parametreli foo')
```

```
s = Sample()
s.foo(10)    # geçerli
s.foo()      # error!
```

Python'da biz istersek sınıfı daha sonra metod da ekleyebiliriz. Örneğin:

```
class Sample:
    def foo(self):
        print('foo')

def tar(self):
    print('bar')

Sample.bar = tar

s = Sample()
s.foo()
s.bar()

Sample.x = 100
print(Sample.x)
```

Burada Sample.bar ifadesine bir fonksiyon nesnesinin adresi atanmıştır. Bu atama Sample sınıfı için bir sınıf özniteligi oluşturmaktadır. Örneğimizde benzer biçimde Sample.x ifadesine atama yapılarak sınıfın x isimli bir özniteligi de oluşturulmuştur.

"Override" terimi taban sınıftaki bir metod ile aynı isimli türemiş sınıfta bir metodun bulunması durumunu anlatmaktadır. "Override" mekanizması özellikle çokbiçimlilik (polymorphism) için kullanılır. Çokbiçimlilik sayesinde her sınıf belli bir işlevi kendine göre diğerlerinden az çok farklı biçimde yerine getirebilmektedir. Override kavramı Python'da var olan bir kavramdır. Çünkü Python'da da farklı sınıflar içerisinde aynı isimli metodlar bulunabilmektedir.

Sınıflarda İsim Aramı ve MRO (Method Resolution Order) Kavramı

Aslında Python'da bir değişkenle bir metod çağrıldığında metod o değişkenin türüne ilişkin MRO sırası denilen sınıflarda sırasıyla aranmaktadır. Yani MRO çağrılan metodların sırasıyla hangi sınıflarda aranacağını belirten bir kavramdır. Sınıfların (örneklerin değil) `__mro__` isimli öznitelikleri bize bir demet olarak bu sırayı verir. Örneğin:

```
>>> class A: pass
>>> class B(A): pass
>>> class C(B): pass
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

`__mro__` değişkeninin bir demet belirttiğine ve bu demetin elemanlarının type nesneleri olduğuna dikkat ediniz. Örneğin yukarıdaki MRO sırasına göre biz C sınıfı türünden bir değişkenle bir metodу çağrıdığımızda bu metod sırasıyla önce C sınıfında, bulunamazsa B sınıfında, bulunamazsa A sınıfında, orada da bulunamazsa object sınıfında aranacaktır. Bu durumda yukarıdaki sıraya göre örneğin hem B'de hem de A'da aynı isimli bir metod varsa B'deki bulunacaktır.

```
class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
```

```

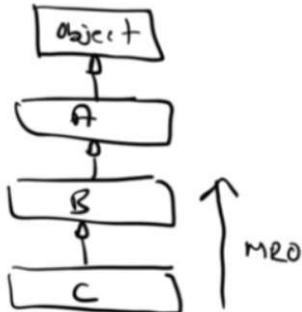
print('B.foo')

class C(B):
    pass

c = C()
c.foo()      # B.foo

```

MRO sırasının aşağıdan yukarıya doğru olduğunu görüyorsunuz:

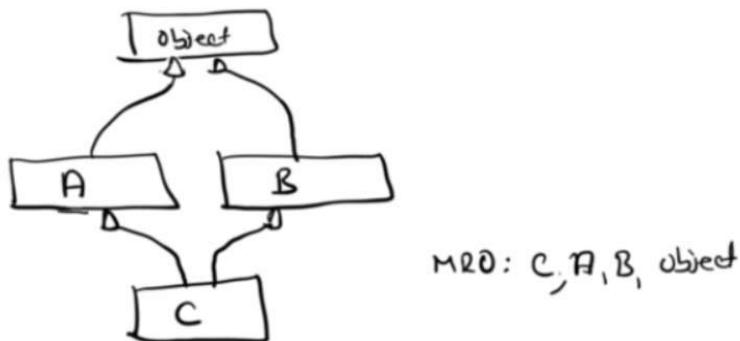


Pekiyi çoklu türetmede nasıl bir MRO sırası vardır Örneğin.

```

>>> class A: pass
>>> class B: pass
>>> class C(A, B): pass
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)

```



Örneğin:

```

class A:
    pass

class B(A):
    pass

class C:
    pass

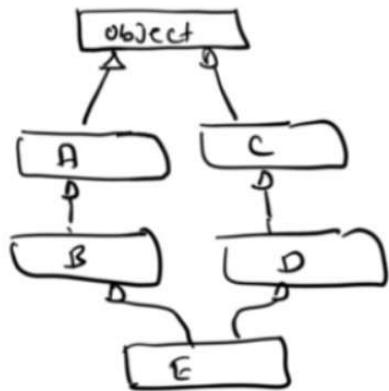
class D(C):
    pass

class E(B, D):
    pass

```

pass

```
print(E.__mro__) # (<class '__main__.E'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.D'>, <class '__main__.C'>, <class 'object'>)
```



E için MRO sırasının E, B, A, D, C, object biçiminde olduğuna dikkat ediniz. Burada iki koldan ortak olanların (object) listenin sonunda olduğuna dikkat ediniz. Önce sol koldan ortak olana kadar yukarıya çıkmıştır. Sonra sağ koldan yukarıya çıkmıştır. En sonunda ortak olan sınıfı bakılmaktadır. Tabii çok daha karışık çoklu türemeler söz konusu olabilir. Genel kuraldan ziyade bu tür durumlarda MRO sırasına gözle bakılması tavsiye edilir.

super Fonksiyonu

super built-in bir fonksiyondur. Aslında bu fonksiyonun ismi uygun verilmemiştir. İşlevine bakıldığındá bu fonksiyonun isminin "super" yerine "next" gibi bir şey olması daha uygundur (next isimli built-in başka bir fonksiyon daha vardır.) Çünkü super sözcüğü NYPT aynı zamanda taban sınıf çağrımasını da yapmaktadır.

super fonksiyonun argümansız ve argümanlı kullanım biçimleri vardır. Argümanlı kullanımda fonksiyon iki argümanla kullanılır. Birinci argüman MRO sırasında aramanın başlama yerini belirten bir sınıf ismidir. İkinci argüman da nesnenin çağrılmamasında kullanılacak değişkeni belirtir. Bu durumda arama MRO sırasına göre birinci argümandan belirtilen sınıfından sonraki sınıfın başlatılır. Aslında argümansız kullanım bu kullanımın özel bir biçimidir. (Tabii aslında iki ayrı super fonksiyonu yoktur. Zaten bunun Python'da mümkün olamayacağını daha önce belirtmiştim. Aslında super fonksiyonunun iki parametresi default değer almaktadır.)

İki argümanlı kullanımının genel biçimini şöyledir:

```
super(<sınıf ismi>, <değişken>).<fonksiyon ismi>(...)
```

Burada arama birince argümanla belirtilen sınıfından sonraki sınıfın başlatılır. İkinci argüman ise bu fonksiyonun çağrılmamasında kullanılacak değişkeni belirtmektedir. Yani örneğin:

```
super(D, d).foo()
```

çalışmasının eşdeğeri şöyledir:

```
d.foo()
```

Ancak buradaki foo MRO sırasına göre D'den sonraki sınıfın foo metodudur. Örneğin:

```
class A:  
    def foo(self):  
        print('A.foo')  
  
class B(A):  
    def foo(self):
```

```

print('B.foo')

class C(A):
    def foo(self):
        print('C.foo')

class D(B, C):
    def foo(self):
        print('D.foo')
        super(C, self).foo()      # A.foo çağrılacak

print(D.__mro__)
d = D()
d.foo()

```

Burada MRO sırası D, B, C, A, object biçimindedir. D sınıfının foo metodunu içerisindeki çağrıya dikkat ediniz:

```
super(C, self).foo()      # A.foo çağrılacak
```

Programın çıktısı şöyle olacaktır:

```

(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
D.foo
A.foo

```

super(C, self).foo ifadesi MRO sırasında C'den sonra gelen sınıfın foo metodunun çağrılmamasına yol açacaktır. İki çağrı arasındaki ilişkiye bakınız:

```
super().foo()
super(X, a).foo()
```

super fonksiyonun argümansız çağrıyı yalnızca metodların içerisinde kullanılır. Oysa argümanlı çağrı hem metodlarda hem de dışında kullanılabilirmektedir. Argümansız çağrı asıl olarak argümanlı çağrıının özel bir durumudur. Başka bir deyişle:

```
super().foo()
```

çağrısının eşdeğeri aslında şöyledir:

```
super(<çağrısının içinde yapıldığı sınıf>, çağrıının yapıldığı değişken).foo()
```

Örneğin D sınıfının foo metodunu şöyle çağrılmış olsun:

```
d = D()
d.foo()
```

Bu D sınıfının foo metodunu içerisinde şunu çağrıyı yapmış olalım:

```
super().bar()
```

Bunun eşdeğeri şöyledir:

```
super(D, self).bar()
```

Argümansız kullanımda bu fonksiyon hangi sınıf içerisinde çağrılmışsa aramayı MRO sırasında o sınıfından sonraki sınıfından itibaren başlatır.

```

class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
        print('B.foo')
        super().foo()

class C(B):
    def foo(self):
        print('C.foo')
        super().foo()

c = C()
c.foo()

```

Burada MRO sırası C, B, A, object biçimindedir. Her super çağrıını aramayı MRO sırasında sonraki sınıfından itibaren devam ettirir. Örneğin C içerisindeki super çağrıını aramayı B'den, B içerisindeki super çağrıını aramayı A'dan devam ettirecektir. Bu nedenle burada ekrana sırasıyla şunlar çıkacaktır:

```

C.foo
B.foo
A.foo

```

Örneğin aşağıdaki gibi bir çoklu türetme olsun:

```

class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
        print('B.foo')
        super().foo()

class C(A):
    def foo(self):
        print('C.foo')
        super().foo()

class D(B, C):
    def foo(self):
        print('D.foo')
        super().foo()

print(D.__mro__)
d = D()
d.foo()

```

Burada D'nin MRO sırası D, B, C, A, object biçimindedir. Her super çağrıını bu sırada kalınan yerden devamı sağlamaktadır. Bu durumda programın çıktısı da şöyle olacaktır:

```

(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
D.foo
B.foo
C.foo
A.foo

```

super fonksiyonu hangi sınıfı çağrılmışsa arama MRO sırasına göre o sınıfın sonraki sınıfın başlatılmaktadır. Örneğin:

```
class A:
    def foo(self):
        print('A.foo')

class B(A):
    def foo(self):
        print('B.foo')
        super().foo()

class C(A):
    def foo(self):
        print('C.foo')
        super().foo()

class D(B, C):
    pass

print(D.__mro__)
d = D()
d.foo()
```

Burada artık ilk bulunan foo MRO sırasına göre B'deki foo metodudur. Buradaki super çağrımı aramayı MRO sırasında B'den itibaren B'deki super çağrıını da C'den itibaren, C'deki super çağrıını da A'dan itibaren başlatacaktır. Programın çıktısı şöyle olacaktır:

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
B.foo
C.foo
A.foo
```

Pekiyi bu biçimde argümansız kullanımda super fonksiyonunun geri dönüş değeri hangi türdendir? Aslında teknik olarak bu fonksiyon bu çağrıyı yapma yeteneğine sahip olan "super" isimli bir sınıf türündendir. Bu sınıf MRO sırasına göre kalınan yerden itibaren ilgili fonksiyonu arama yeteneğine sahiptir.

Türetme Durumlarında Türemiş Sınıfın `__init__` Metodunun Taban Sınıfın `__init__` Metodunu Çağırması İle İlgili Ayrıntılar

Daha önce türemiş sınıfın `__init__` metodunun taban sınıfın `__init__` metodunu çağırması gerektiğini söylemiştık. Bu çağrımayı doğrudan sınıf ismi belirterek ya da super fonksiyonıyla yapabiliriz. Örneğin B sınıfı A sınıfından türetilmiş olsun. B sınıfının `__init__` metodu içerisinde A sınıfının `__init__` metodunu şu yollarla çağrılabılır:

```
A.__init__(self, ...)
super().__init__(...)
super(A, self).__init__(...)
```

İşte bu tür durumlarda birinci örnekteki gibi taban sınıf isminin kullanılarak taban sınıf `__init__` metodunun çağrılması iyi bir teknik değildir. Çünkü bu çağrımda MRO sırasına bakılmamaktadır. Bu çağrıının hem bakımı zordur hem de çoklu türetmede taban sınıfın `__init__` metodunun iki kere çağrılmazı gibi sorunlara yol açma potansiyeli vardır. Örneğin:

```
class A:
    def __init__(self):
        print('A.__init__')

class B(A):
    pass
```

```

def __init__(self):
    A.__init__(self)
    print('B.__init__')

class C(A):
    def __init__(self):
        A.__init__(self)
        print('C.__init__')

class D(B, C):
    def __init__(self):
        B.__init__(self)
        C.__init__(self)
        print('D.__init__')

print(D.__mro__)
d = D()

```

Programın çıktısı şöyle olacaktır:

```

(<class '__main__.D', <class '__main__.B', <class '__main__.C', <class '__main__.A', <class 'object'>)
A.__init__
B.__init__
A.__init__
C.__init__
D.__init__

```

Burada A sınıfının `__init__` metodunun birden fazla kez çağrıldığına dikkat ediniz. Bu durum önemli sorunlara yol açabilmektedir. Halbuki çağrıma super fonksiyonu ile yapılsaydı böyle bir durum oluşmayacaktı.

```

class A:
    def __init__(self):
        print('A.__init__')

class B(A):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A):
    def __init__(self):
        super().__init__()
        print('C.__init__')

class D(B, C):
    def __init__(self):
        super().__init__()
        print('D.__init__')

print(D.__mro__)
d = D()

```

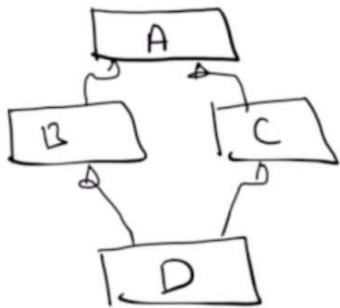
Programın çıktısı şöyle olacaktır:

```

(<class '__main__.D', <class '__main__.B', <class '__main__.C', <class '__main__.A', <class 'object'>)
A.__init__
C.__init__
B.__init__
D.__init__

```

Türemiş sınıfın `__init__` metodunda taban sınıfın `__init__` metodunun çağrılmaması işleminde eğer taban sınıfları biz yazmamışsa ve bunlar içerisinde super çağrısıyla MRO zinciri devam ettirilmemişse taban sınıf `__init__` metodlarının uygun bir biçimde çağrılmaması mümkün olmayabilir. Örneğin aşağıdaki türetme şeması söz konusu olsun:

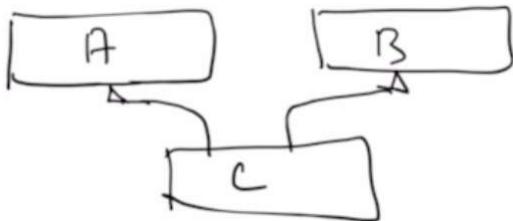


Burada A, B ve C bizim tarafımızdan yazılmamış olsun. Biz D sınıfını B ve C'den çoklu türetmek isteyelim:

```
class D(B, C):
    def __init__(self):
        super(D, self).__init__()
        # diğer işlemler
```

Biz burada D'inin `__init__` metodu içerisinde super çağrısıyla MRO sırasına göre bir yukarıdaki sınıfın `__init__` metodunu çağrırmış olalım. Pekiyi B ve C sınıflarını yazan kişiler kendi taban sınıflarının (yani A'nın) `__init__` metodunu super ile değil de taban sınıf isıyla çağrırmışlarsa sorun çıkacaktır. Eğer bu sınıfların hepsini biz yazmış olsaydık hepsinde super çağrıyı yaparak durumu ayarlayabilirdik. Bu tür durumlarda çoklu türetme yaparken dikkat etmelisiniz.

Eğer sınıfın bir taban sınıfı yoksa (yani sınıf object sınıfından türetilmişse) programcılar bu sınıfların `__init__` metodlarında object sınıfının `__init__` metodlarını çağrırmazlar. (Tabii çağrımlarında bir sakınca olmaz.) Dolayısıyla bu sınıflar bizim tarafımızdan yazılmamışlarsa onlarda super çağrısı yapılmadığı için yine sorun oluşabilecektir. Örneğin:



Burada A ve B sınıflarının başkaları tarafından yazıldığını düşünelim. Dolayısıyla bu sınıfları yazanlar böyle bir çoklu türetmenin yapılacağını bilmediklerinden taban sınıfın (yani object) sınıfının `__init__` metodunu super çağrısıyla çağrırmamış olacaklardır. Bu durumda bizim C sınıfının `__init__` metodunda super çağrısıyla taban sınıfların `__init__` metodlarının çağrılmamasını sağlamamız soruna yol açacaktır:

```
class A:
    def __init__(self):
        print('A.__init__')

class B:
    def __init__(self):
        print('B.__init__')

class C(A, B):
    def __init__(self):
        super(C, self).__init__()
```

```

print('C.__init__')
c = C()

```

Burada B sınıfının `__init__` metodu hiç çağrılmayacaktır. İşte bu tür durumlarda eğer A ve B sınıfları bizim tarafımızdan yazılmamışlarsa C sınıfında artık çağrıyı super ile değil taban sınıf ismi vererek yapmamız daha uygun olacaktır:

```

class A:
    def __init__(self):
        print('A.__init__')

class B:
    def __init__(self):
        print('B.__init__')

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')

c = C()

```

Sonuç olarak başkaları tarafından yazılmış sınıflardan çoklu türetme yaptığı durumlarda bu duruma dikkat etmek gereklidir.

Çoklu türetme az karşılaşılan bir durum olsa da burada bir sorundan daha bahsedeceğiz. Çoklu türetme sırasında türemiş sınıfın taban sınıfına `__init__` metodlarında veri aktarılması biraz zahmetlidir. MRO sırası her zaman bir sınıfın doğrudan taban sınıfı ile ilişkili olmadığı için parametrelerin diğer sınıflara aktarılması sorun olabilmektedir. İşte bunun için fonksiyonların `**`li parametrelerinden ve argümanlarından faydalanyılır. Burada biz yalnızca bir örnek verip geçeceğiz:

```

class A:
    def __init__(self, a):
        self.a = a
        print('A.__init__')

class B(A):
    def __init__(self, b, **args):
        super().__init__(**args)
        self.b = b
        print('B.__init__')

class C(A):
    def __init__(self, c, **args):
        super().__init__(**args)
        self.c = c
        print('C.__init__')

class D(B, C):
    def __init__(self, d, **args):
        super().__init__(**args)
        self.d = d
        print('D.__init__')

print(D.__mro__)
d = D(10, a = 20, b = 30, c = 40)           # 20 30 40 10
print(d.a, d.b, d.c, d.d)

```

Bazen taban sınıfın `__init__` metodu bulunduğu halde türmeş sınıfın bulunmayabilir. Aslında `__init__` metodu da normal bir metot gibi `__new__` metodu (constructor) çağrılmaktadır. Dolayısıyla MRO sırasına göre arama yapıldığında taban sınıfın `__init__` metodu çağrılır. Tabii taban sınıfı da `__init__` metodu olmayabilir. Bu durumda object sınıfındaki çağrılr. Biz bunu fark etmemektedir. Örneğin:

```
class A:  
    def __init__(self):  
        print('A.__init__')  
  
class B(A):  
    pass  
  
print(B.__mro__)          # (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)  
b = B()                  # A.__init__ çağrılr
```

Python'da Kapsülleme (Encapsulation) ve Örnek Özelliklerinin Dışarıdan Gizlenmesi (Data Hiding)

Bilindiği gibi NYPT'de bir olgunun bir sınıf ile temsil edilip o sınıfın dışarıyı ilgilendirmeyen elemanlarının dışarıdan gizlenmesine "kapsülleme (encapsulation)" ve veri elemanlarının dışarıdan gizlenmesine de "veri elemanlarının gizlenmesi (data hiding)" denilmektedir. Pek çok dilde bunu sağlamak için sınıfların "private" biçiminde dışarıya gizlenen bölmeleri vardır. Pek çok Python'da bu kavramların kullanılması nasıldır?

Python'da sınıfların diğer pek çok nesne yönelimli programlama dillerinde olduğu gibi "private", "protected" ve "public" gibi bölgeleri yoktur. Yani Python'da zaten default olarak sınıfın tüm elemanları "public" gibidir. Herhangi bir elemanı dışarıdan gizlemenin etkin bir yolu da yoktur.

Her ne kadar Python'da elemanları dışarıdan gizlemenin kesin bir yolu yoksa da bu konuda birkaç küçük tavsiye bulunmaktadır. Eğer bir özniteligin ya da metodun ismi _ (single underscore) ile başlatılarak isimlendirilmişse Python dünyasında bu öznitelikler "private" eleman olarak değerlendirilmektedir. Gerçi `_xxx` biçimindeki özniteliklere dışarıdan yine isteyenler erişebiliyorsa da kodu anlamlandıran kişiler bunları private eleman gibi ele almaktadır. Başka bir deyişle biz bir elemanın dışarıdan erişilmesini mantık olarak istemiyorsak onu başına `_` karakteri getirerek isimlendirmeliyiz. Böyle isimleri gören programcılar bu değişkenleri dışarıdan kullanmak konusunda isteksiz davranış olacaktır. Fakat bu konuda yorumlayıcının bir zorlaması söz konusu olmayacağıdır. (Yani programcı bunu dışarıdan kullanmak isterse yine kullanabilir.)

Ayrıca bir de Python'da sınıf elemanlarının `__` (double underscore) ile başlatılarak isimlendirmesi de vardır. Böyle değişkenlere dışarıdan ilgili sınıf türünden değişken ve `.` operatörü ile erişilemez. Ancak bu erişim yasağı da aslında bu değişkenin dışarıdan kullanımını tamamen ortadan kaldırılmamaktadır. Şöyled ki: `__xxx` biçiminde isimlendirilmiş sınıf elemanları dışarıdan yine `<sınıf ismi>__<eleman ismi>` biçiminde kullanılabilmektedir. Daha açık bir örnek vermek gerekirse `X` sınıfının `__x` isimli bir elemanı dışarıdan `__x` ismiyle kullanılamaz. Ancak `_X__x` ismiyle kullanılabilir. İşte `__` (double underscore) ile isimlendirme değişkenin dışarıdan kullanılmasını kısmen (fakat tamamen değil) engellemektedir. Örneğin:

```
class Sample:  
    def __init__(self, a):  
        self._a = a  
  
    def disp(self):  
        print(self._a)  
  
    def __foo(self):  
        print('Sample.__foo')  
  
s = Sample(10)  
s.disp()          # kötü teknik, _ ile başlatılan değişkenler dışarıdan kullanılmamalıdır  
s._a = 20         #  
s._Sample__foo() #  
s.__foo()         # exception!
```

Burada sınıfın `_a` isimli örnek özniteligi programcı tarafından "private" etkisi yaratmak için kullanılmıştır. Dolayısıyla sınıfı kullanan kişilerin bu öznitelige dışarıdan erişmeye çalışması kötü bir tekniktir. Sınıfın `__foo` isimli metodu ise dışarıdan `__foo` ismiyle kullanılamaz. Yorumlayıcı bu ismi `_Sample__foo` biçiminde değiştirmiştir. Dolayısıyla biz `__foo` metodunu dışarıdan şöyle çağrıramayız:

```
s.__foo()
```

Şöyleden çağrılabılır (tabii aslında hiç çağrırmaya çalışmamalıyız):

```
s._Sample__foo()
```

Tabii istesek şöyle de çağrılabılır:

```
Sample._Sample__foo(s)
```

Peki `__xxx` ile `_xxx` isimlendirmeleri arasında niyet bakımından farklılık var mıdır? İşte genel olarak `__xxx` isimlendirmesinin daha yüksek bir "buna dokunma" niyeti taşıdığını söyleyebiliriz. `_xxx` isimlendirmesi bu anlamda `__xxx` isimlendirmesine göre daha düşük bir niyet taşımaktadır.

Python'da `__xxx` isimli değişkenler `_xxx` isimli değişkenler gibi ele alınmamaktadır. Başında ve sonunda `_` olan sınıf elemanları özel elemanlardır. Başka bir deyişle bu metodlar yorumlayıcı ve dil için özel anlam ifade etmektedir. Örneğin `__init__` metodu, `__new__` metodu ve operatör metodları böyledir.

Python'da Çokbiçimlilik (Polymorphism)

Çokbiçimlilik biyolojiden aktarılmış bir terimdir. Biyolojide çokbiçimlilik canlıların çeşitli doku ve organlarının onların yaşam koşullarına göre temel işlevleri aynı kalmak üzere farklılaşmasına denilmektedir. Örneğin kulak pek çok canlıda vardır ve temel işlevi "duymak"tır. Ancak her canlı bu duyma eylemini biraz farklı gerçekleştirmektedir. Yazılımda çokbiçimlilik türden bağımsız kod parçalarının oluşturulması için kullanılan bir tekniktir. Başka bir deyişle türüne bilmeden nesne üzerinde işlem yapmayı sağlar. Çokbiçimlilik sınıflarla ilgili bir kavramdır. Çokbiçimliliğin uygulanması için farklı sınıfların bulunuyor olması gereklidir.

Çokbiçimlilik farklı sınıfların belli bir eylemi temel işlev aynı kalmak üzere kendilerine göre farklı biçimlerde yapması durumudur. Böylece farklı sınıflar programcı tarafından bu eylem temelinde sanki aynı sınıflar gibi ele alınmaktadır. Örneğin `play_music` isimli fonksiyon bizden bir sınıf nesnesini alıp ondan hareketle bir müzik çaldığını düşünelim. Bu müzik çaldırma işlemi ilgili sınıfın `play` isimli metoduyla yapılıyor olsun. İşte bu `play` metodu çokbiçimli bir metottur. Yani pek çok sınıfta `play` vardır. Bu `play` metodu bu sınıflarda bir müzik parçasını çalmakta kullanılır. Ancak her sınıfın `play` metodu kendine göre (örneğin farklı formatlara göre) bu çalışma eylemini gerçekleştirir. Örneğin:

```
def play_music(p):
    p.play()
```

Bizin bu `play_music` fonksiyonuna geçireceğimiz sınıf nesnesinin `play` isimli bir metodu olmalıdır. O halde biz bu fonksiyona farklı sınıfları parametre olarak geçirebiliriz. Yeter ki onların `play` metodları olsun. Böylece bu fonksiyon kendisine geçirilen sınıfın detaylarını bilmez. Kendisine parametre yoluyla geçirilen nesnenin bir müzik çaldırma yeteneğine sahip olduğunu bilmesi yeterlidir.

```
class MP3:
    def __init__(self, path):
        self.path = path

    def play(self):
        print("MP3 çalıyor")
```

```
class WAV:
```

```

def __init__(self, path):
    self.path = path

def play(self):
    print("Wav çalıyor")

class WMA:
    def __init__(self, path):
        self.path = path

    def play(self):
        print("WMA çalıyor")

class M4A:
    def __init__(self, path):
        self.path = path

    def play(self):
        print("M4A çalıyor")

def playMusic(p):
    p.play()

mp3 = MP3('c:\\temp\\x.mp3')
wav = WAV('c:\\temp\\y.wav')
wma = WMA('c:\\temp\\z.wma')
m4a = M4A('c:\\temp\\k.m4a')

playMusic(mp3)
playMusic(wav)
playMusic(wma)
playMusic(m4a)

```

Aslında çokbiçimlilik gerçek hayatı da karşımıza çokça çıkmaktadır. Örneğin bizim için otomobil nasıl olursa olsun temel işlevi onun gitmesidir. Tüm otomobiller gider ancak kendine göre farklılıklar olabilir. Tüm koltuklara oturabiliriz. Ancak bu koltuklar birbirlerinden farklı olabilir. Köpekler birbirinden farklı cinslere sahip olsalar da hepsi havlamaktadır. Ancak bu havlamanın biçimi köpektен köpeğe değişimdir.

Çokbiçimlilik sayesinde biz türden bağımsız işlemler yapan kodlar yazabilirmiz. Örneğin topla oynanan bir oyun söz konusu olsun. Oyunada farklı toplar bulunuyor olabilir. Normal top, patlak top, zıplayan top gibi. Biz bir topu hareket ettirirken onun nasıl bir top olduğunu bilmek zorunda kalmadan kodumuzu yazabilirmiz.

```

def play(top):
    top.git()
    ≡
    top.git()
    ≡
    top.git()
    ≡
    nt = NormalTop()
    play(nt)
    ≡
    pt = PatlakTop()
    play(pt)
    ≡
    zt = ZoplayanTop()
    play(zt)

```

Burada play fonksiyonu "git" özelliğine sahip olan her türlü top nesnesi ile çağrılabılır. Başka bir deyişle play metodu farklı toplarla oyunun oynanmasını sağlamaktadır.

Örneğin bir işletmede çalışan kişilerin maaşları değişik formüllere göre hesaplanıyor olabilir. O halde maaş hesaplama çokböbümlü bir eylemdir. Yani her çalışanın maaş hesaplaması vardır fakat birbirlerinden az çok farklı yapılmaktadır. Aşağıdaki örnekte üç tür çalışanın maaşı farklı biçimde hesaplanmaktadır. Biz de çalışanların hepsini bir liste içerisinde tutarsak onların hangi türden çalışan olduğunu bilmeden maaş işlemlerini yapabiliriz:

```
class Employee:  
    def __init__(self, name):  
        self.name = name  
  
class Worker(Employee):  
    def __init__(self, name, weekHours):  
        super().__init__(name)  
        self.weekHours = weekHours  
  
    def calc_salary(self):  
        return self.weekHours * 30  
  
class Manager(Employee):  
    def __init__(self, name, prim):  
        super().__init__(name)  
        self.prim = prim  
  
    def calc_salary(self):  
        return 7000 + 7000 * self.prim  
  
class SalesPerson(Employee):  
    def __init__(self, name, prim):  
        super().__init__(name)  
        self.prim = prim  
  
    def calc_salary(self):  
        return 3000 + 3000 * self.prim  
  
employees = [Worker('Ali', 40), Manager('Veli', 0.20), SalesPerson('Selami', 0.10)]  
  
for emp in employees:  
    print('Adı: {}, Maaş: {}'.format(emp.name, emp.calc_salary()))  
  
salary = 0  
for emp in employees:  
    salary += emp.calc_salary()  
  
print('Toplam Maaş: {}'.format(salary))
```

Bir tetris programında birtakım şekiller düşmektedir. Bu şekiller farklı sınıflarla temsil edilebilir ve çokböbümlilik sayesinde kodun büyük kısmı sekilden bağımsız biçimde (yani her şekilde çalışacak biçimde) yazılabilir. Böyle bir tema şöyle oluşturulabilir:

```
class Shape:  
    pass  
  
class SquareShape(Shape):  
    def move_down(self):  
        print('SquareShape moves down')  
  
class BarShape(Shape):  
    def move_down(self):  
        print('BarShape moves down')  
  
class TShape(Shape):  
    def move_down(self):  
        print('TShape moves down')
```

```

class LShape(Shape):
    def move_down(self):
        print('LShape moves down')

class ZShape(Shape):
    def move_down(self):
        print('ZShape moves down')

import random
import time

class Tetris:
    def get_random_shape(self):
        return random.choice([ZShape, TShape, SquareShape, BarShape])()

    def run(self):
        while True:
            shape = self.get_random_shape()
            for i in range(20):
                shape.move_down()
                time.sleep(0.5)

tetris = Tetris()
tetris.run()

```

Her türlü uygulamada çokbiçimlilikle karşılaşmak mümkündür. Örneğin bir satranç tahtası programında değişik taşlar vardır. Tüm taşların ortak özellikleri Figure isimli bir sınıfı toplanabilir. Taşların her biri türlerine göre Figure sınıfından türetilmiş olan King, Queen, Rook, Bishop, Knight, Pawn gibi sınıflarla temsil edilebilir. Satrançta her taş değişik bir biçimde gitmektedir. İşte taşların gidişleri move isimli çokbiçimli metotla sağlanabilir. Bu durumda değişik taşlar aynı isimli move metodlarına sahip oldukları halde aslında farklı bir biçimde giderler. Yani burada taşın gitmesi çokbiçimli bir eylemdir. Her taş gider fakat kendine göre değişik bir biçimde gider. Çokbiçimlilik farklı nesnelerin farklılığını dikkate almadan sanki aynı nenseymiş gibi ele alınmasını sağlamaktadır. Bu da kodun organizasyonunu ve test edilmesini kolaylaştırmaktadır.

Sınıfların `__str__` ve `__repr__` Metotları

Çokbiçimliliğe tipik bir örnek print fonksiyonudur. Biz built-in print fonksiyonu ile her şeyi print edebiliriz. Fakat print ettiğimiz şeyle onların türlerine göre değişim bilmektedir. Peki nasıl?

```

class Sample:
    def __init__(self, val):
        self.val = val

    def __str__(self):
        return 'Sample object, value = {}'.format(self.val)

s = Sample(100)
print(s)

```

Ne zaman bir sınıf nesnesini str fonksiyonu ile str türüne dönüştürecek olsak str fonksiyonu ilgili sınıf nesnesi ile `__str__` metodunu çağrılmaktadır. Örneğin:

```

result = str(x)

ile

result = x.__str__()

```

aynı anlamdadır. O halde aslında str fonksiyonu şöyle yazılmıştır:

```
def str(s):
    return s.__str__()
```

print fonksiyonu da parametreleriyle aldığı nesneler üzerinde `__str__` metodlarını çağırarak elde ettiği yazıyı ekrana yazdırmaktadır. Örneğin:

```
print(a)
```

ile

```
print(str(a))
```

ya da,

```
print(a.__str__())
```

tamamen işlevsel olarak eşdeğerdir.

Biz kendi sınıflarımız için `__str__` metodunu kendi istediğimiz gibi yazarsak bu sınıflarımız türünden nesneleri print fonksiyonuyla istediğimiz formatta ekrana yazdırabiliriz. Örneğin:

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return '{}+{}i'.format(self.real, self.imag)

z = Complex(3, 2)
print(z)
```

Örneğin:

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def __str__(self):
        return '{}/{}/{}'.format(self.day, self.month, self.year)

d = Date(10, 12, 2008)
print(d)
```

Pekiyi biz sınıfımız için `__str__` metodunu yazmazsak ne olur? İşte anımsanacağı gibi bir sınıfta belli bir metot yoksa isim arama işlemi MRO sırasına göre taban sınıflarda yapılmaktadır. Her sınıf da doğrudan ya da dolaylı olarak object sınıfından türetilmiş olduğuna göre biz sınıfımız için `__str__` metodunu yazmazsak en kötü olasılıkla object sınıfının `__str__` metodu çağrılmacaktır. object sınıfının `__str__` metodu da nesnenin bellek adresini yazı olarak bize vermektedir. Örneğin:

```
<__main__.Date object at 0x0000025002588198>
```

Sınıfların `__str__` metodlarına çok benzer olan bir de `__repr__` metodları vardır. Bu metodlar da yine tipik olarak nesneyi temsil eden bir yazıyla geri dönerler. Built-in global `repr` isimli fonksiyon parametresiyle aldığı nesne üzerinde `__repr__` metodunu çağrılmaktadır. Yani global `repr` fonksiyonu şöyle yazılmıştır:

```
def repr(o):
    return o.__repr__()
```

Sınıfların `__repr__` metotları Python'ın komut yorumlayıcıları tarafından (IDLE gibi, IPython gibi, python gibi) bir değişkenin ismi yazılıp ENTER tuşuna basıldığında da çağrılmaktadır. Yani örneğin biz IDLE'da x değişkenin içerisindeki değeri görmek için x yazıp ENTER tuşuna bastığımızda aslında IDLE x değişkeni ile `__repr__` metodunu çağrıp onun sonucunu yazdırmaktadır. O halde Python'ın komut yorumlayıcılarında bir değişkeni print ile yazdırmak ile onun ismini yazıp ENTER tuşuna basarak yazdırma arasında farklılık olabilmektedir. Örneğin:

```
>>> class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def __str__(self):
        return '{}/{}/{}'.format(self.day, self.month, self.year)

    def __repr__(self):
        return 'Date object: {}/{}/{}/{}'.format(self.day, self.month, self.year)

>>> d = Date(10, 12, 2009)
>>> d
Date object: 10/12/2009
>>> print(d)
10/12/2009
```

Örneğin:

```
>>> s = 'ali\nveli'
>>> s
'ali\nveli'
>>> print(s)
ali
veli
```

Pekiyi `__str__` ile `__repr__` arasındaki fark nedir? Neden biri varken diğerine gereksinim duyulabilmektedir? İşte bu iki metod arasındaki temel fark `__str__` metodunun kullanıcılarla yönelik teknik olmayan bir yazı geri döndürmesi `__repr__` metodunun ise programcıya yönelik daha teknik bir yazı geri döndürmesidir. Tabii böyle bir zorunluluk yoktur. Ancak tasarımcının niyeti böyledir. Programcılar bazen `__str__` ile `__repr__` metodlarının her ikisinde de aynı yazıyı geri döndürebilmektedir. Örneğin:

```
>>> a = 123
>>> print(a.__str__())
123
>>> print(a.__repr__())
123
```

Tabii yine programcı sınıfı için `__repr__` metodunu yazmayabilir. Bu durumda en kötü olasılıkla yine object sınıfının `__repr__` metodu çalıştırılacaktır. Ayrıca şöyle bir ayrıntı da vardır: str fonksiyonu ilgili sınıfta `__str__` metodу varsa onun geri döndürdüğü yazıyı verir. Fakat ilgili sınıfta `__str__` yok fakat `__repr__` metodу varsa `__repr__` metodunun geri döndürdüğü yazıyı vermektedir. Örneğin:

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def __repr__(self):
```

```

    return 'Date object: {}/{}/{}/{}'.format(self.day, self.month, self.year)

d = Date(10, 12, 2008)
print(d)                      # Date object: 10/12/2008
print(str(d))                  # Date object: 10/12/2008
print(repr(d))                 # Date object: 10/12/2008

```

Tabii tersi durum geçerli değildir. Yani sınıfı `__str__` varsa fakat `__repr__` yoksa bu durumda `repr` fonksiyonu ya da komut yorumlayıcılar `__str__` metodunu çağrırmazlar. Artık `__repr__` metodu taban sınıflarda aranacaktır.

`__str__` ve `__repr__` metodlarını bizim birer yazı ile (yani string ile) geri döndürmemiz gereklidir. Aksi takdirde exception oluşacaktır.

Nesnelerin Referans Sayaçları

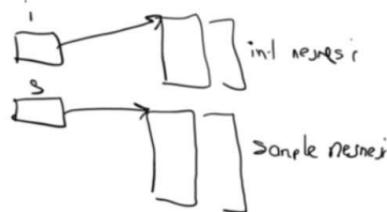
Python'da bir sınıf türünden yaratılmış olan alanlara nesne denilmektedir. Nesnelerin adresleri değişkenlerde tutulur. Yani Python'da tüm değişkenler adres tutmaktadır. Örneğin:

```
i = 10
s = Sample()
```

Burada `i` ve `s` birer değişkendir. Ancak bu değişkenlerin gösterdiği yerde `int` türünden ve `Sample` türünden nesneler vardır:

`i = 10`

`s = Sample()`



Python'da bir nesneyi birden fazla değişken gösterebilir. Örneğin:

```
s = Sample()
k = s
m = k
```

Burada yaratılmış olan `Sample` nesnesini üç farklı değişken göstermektedir. Python yorumlayıcısı her nesneyi kaç değişkenin gösterdiğini anbean tutmaktadır. Buna nesnenin referans sayacı denilmektedir. Örneğin:

`s = Sample()`

→ Bu noktada nesnenin referans sayacı 1'dır

`k = s`

→ Bu noktada nesnenin referans sayacı 2'dır

`m = k`

→ Bu noktada nesnenin referans sayacı 3'tür.

Nesnenin referans sayacı program çalışırken sürekli değişebilir. Örneğin aşağıdaki kodda içerisinde 10 değerinin bulunduğu int nesnesinin referans sayacı sürekli olarak değişmiş ve en sonunda 0 olmuştur:

```

def foo(x):
    print(x)          # RS: 4

a = 10
# RS: 1
b = a
# RS: 2
c = b
# RS: 3
foo(c)
# RS: 3

c = 'ali'
# RS: 2
b = 'veli'
# RS: 1
a = 'selami'
# RS: 0

```

Python'a değişkenler daha önceden de açıklandığı gibi otomatik bir biçimde yok edilmektedir. Ancak nesneler onları gösteren hiçbir değişken kalmadığında (yani referans sayıları 0 olduğunda) çöp toplayıcı mekanizma tarafından yok edilmektedir.

Bir nesnenin referans sayacının değeri sys modülündeki getrefcount isimli fonksiyon tarafından elde edilebilir. Ancak getrefcount fonksiyonu parametre değişkenine atama yüzünden hep sayacı 1 fazla göstermektedir. Örneğin:

```

import sys

class Sample:
    def __del__(self):
        print('__del__ called')

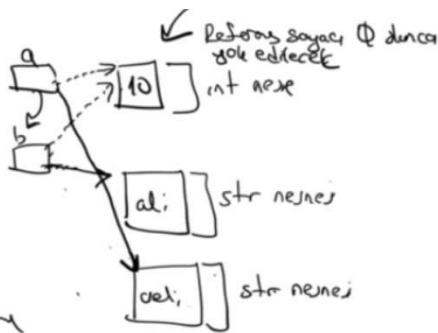
a = Sample()
print(sys.getrefcount(a))
b = a
print(sys.getrefcount(a))
c = b
print(sys.getrefcount(a))
c = None
print(sys.getrefcount(a))
b = None
print(sys.getrefcount(a))
a = None

```

Pyton'ın Çöp Toplayıcı Mekanizması (Garbage Collectors) ve Sınıfların __del__ Metotları

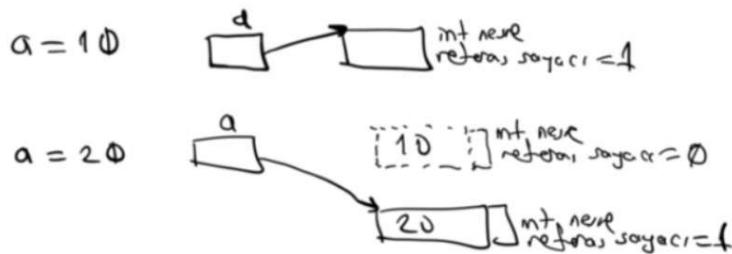
Çöp toplayıcı mekanizmadan daha önce bahsetmiştik. Python'da C++'taki kadar olmasa da Java ve .NET'ten daha deterministik bir çöp toplama mekanizması kullanılmaktadır. Python'ın çöp toplama mekanizması referans sayacı temelinde çalışmaktadır. Python'da yaratılan her nesnenin kaç referans tarafından gösterildiği nesnenin içerisindeki bir referans sayacı tarafından tutulur. Yukarıda da görüldüğü gibi bu referans sayacı nesneyi başka bir referans daha gösterdiği zaman artırılmakta, artık bir referansın o nesneyi göstermediği durumda da eksiltilmektedir. Bu referans sayacı sıfıra düşüğünde nesne çöp durumuna gelir ve çöp toplayıcı mekanizma tarafından hemen bellekten yok edilir. Örneğin:

$a = 1 \emptyset$
 $\rightarrow RS: 1$
 $b = a$
 $\rightarrow RS: 2$
 $b = 'ali'$
 $\rightarrow RS: 1$
 $a = b \& i$
 $\rightarrow RS = \emptyset$ ve artık nesne
bu noktada yok edilir

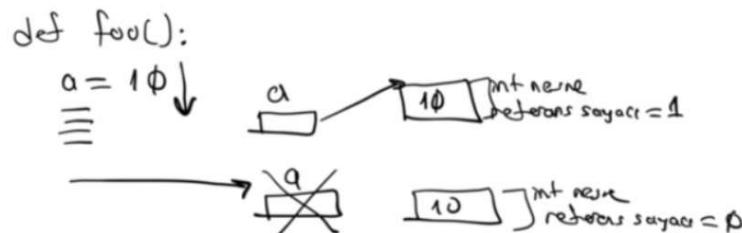


Pekiyi nesnenin referans sayacı nasıl eksilmektedir? Tipik senaryolar şunlar olabilir:

- Değişkene başka bir değer atanmıştır. Dolayısıyla artık değişken eski nesneyi değil yeni nesneyi gösterir halde gelmiştir. Bu durumda eski nesnenin referans sayacı bir eksiltılır. Örneğin:



- Değişken bir yerel ya da parametre değişkendir. Fonksiyondan çıktılığında bu yerel ya da parametre değişkeni yok edilir. Dolayısıyla artık o eski nesneyi göstermez hale gelir. Örneğin:



- Sınıfın bir özniteligi bir nesneyi göstermektedir. Sınıf nesnesi yok edildiğinde artık o değişken de o nesneyi göstermez hale gelir. Örneğin:

```

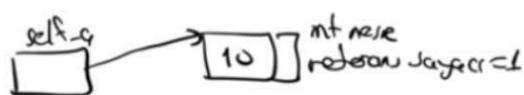
class Sample:
    def __init__(self, a)
        self.a = a;
    def foo();
    =

```

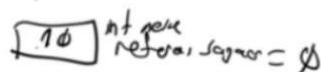
s = Sample(10)



`__init__` metodunu sonlandırdı



s yok edildiğinde



Python'da bir nesnenin referans sayacı sıfıra düşer düşmez referans sayacını düşüren yorumlayıcı hemen nesneyi yok etmektedir. Bu anlamda yok etme süreci gecikmeli bir biçimde yapılmamaktadır.

İşte bir nesnenin referans sayacı sıfıra düşer düşmez çöp toplayıcı nesneyi yok etmeden önce sınıfın `__del__` isimli bir metodunu çağırır. Örneğin:

```

class Sample:
    def __del__(self):
        print('Nesne yok edilecek!')

s = Sample()
print('Referans sayacı 1')
k = s
print('Referans sayacı 2')

k = None
print('Referans sayacı 1')
s = None
print('Referans sayacı 0')

```

Türetme durumlarında taban sınıfın `__del__` metodunun çağrılması `__init__` metodunda olduğu gibi türemiş sınıfın `__del__` metodu tarafından yapılmalıdır. Yani taban sınıfın `__del__` metodu otomatik çağrılmamaktadır. Tabii türemiş sınıf taban sınıfın elemanlarını kullanabildiğine göre taban sınıfın `__del__` metodunun türemiş sınıfın `__del__` metodunun sonunda yapılması uygun olur. Örneğin:

```

class A:
    def __init__(self):
        print('A.__init__ called')

    def __del__(self):
        print('A.__del__ called')

class B(A):
    def __init__(self):
        super().__init__()
        print('B.__init__ called')

```

```

def __del__(self):
    print('B.__del__ called')
    super().__del__()

```

```

b = B()
b = None

```

Coklu türetmede yine taban sınıfının `__del__` metodu `super` fonksiyonuyla çağrılmalıdır. Örneğin:

```

class A:
    def __init__(self):
        super().__init__()
        print('A.__init__ called')

    def __del__(self):
        print('A.__del__ called')
        super().__del__()

class B:
    def __init__(self):
        super().__init__()
        print('B.__init__ called')

    def __del__(self):
        print('B.__del__ called')

class C(A):
    def __init__(self):
        super().__init__()
        print('C.__init__ called')

    def __del__(self):
        print('C.__del__ called')
        super().__del__()

class D(B):
    def __init__(self):
        super().__init__()
        print('D.__init__ called')

    def __del__(self):
        print('D.__del__ called')
        super().__del__()

class E(C, D):
    def __init__(self):
        super().__init__()
        print('E.__init__ called')

    def __del__(self):
        print('E.__del__ called')
        super().__del__()

e = E()
e = None
print(E.__mro__)

```

Programın Ekran çıktısı şöyle olacaktır:

```

B.__init__ called
D.__init__ called
A.__init__ called

```

```
C.__init__ called
E.__init__ called
E.__del__ called
C.__del__ called
A.__del__ called
D.__del__ called
B.__del__ called
(<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.D'>, <class '__main__.B'>, <class 'object'>)
```

Türemiş sınıfların `__del__` metodlarında taban sınıfın `__del__` metodunu çağrıırken programcının taban sınıfın `__del__` metodunun bulunuyor olduğuna emin olması gereklidir. Çünkü `object` sınıfında bir `__del__` metodu yoktur. Dolayısıyla taban sınıfın `__del__` metodunun olmadığı durumda `object` sınıfında da `__del__` metodu olmadığına göre exception oluşur. Özellikle başkaları tarafından yazılmış sınıflardan türetme yapılrken taban sınıfın `__del__` metodunun bulunup bulunmadığına dikkat edilmelidir.

Aynı fonksiyonun ya da metodun içerisindeki değişkenlerin gösterdiği yerdeki nesnelerin referans sayacı, fonksiyon ya da metottan çıktıığında fonksiyon ya da metod içerisindeki bu değişkenlerin yaratım sırasına göre düşürülür. Benzer biçimde global değişkenler için de referans sayaçlarının düşürülmesi yaratım sırasına göre yapılmaktadır. (C++'da başlangıç fonksiyonları ile bitiş fonksiyonları ters sırada çağrılmaktadır. Bu durumdaki durum C++'taki duruma benzememektedir.) Örneğin:

```
class Sample:
    def __init__(self, x):
        self.x = x
        print('Sample.__init__ called: {}'.format(x))

    def __del__(self):
        print('Sample.__del__ called: {}'.format(self.x))

a = Sample(10)
b = Sample(20)

def foo():
    x = Sample(30)
    y = Sample(40)

print('first')
foo()
print('second')
```

Burada ekranda sırasıyla şunları göreceğiz:

```
A.__init__ called: 10
A.__init__ called: 20
first
A.__init__ called: 30
A.__init__ called: 40
A.__del__ called: 30
A.__del__ called: 40
second
A.__del__ called: 10
A.__del__ called: 20
```

Python'da `__del__` metodları `__init__` metodları tarafından yapılan birtakım işlemleri geri alınması için kullanılabilmektedir. Örneğin `__init__` metodunda bir dosya açılmış olabilir. Bu dosya `__del__` metodunda bu dosya kapatılabilir. Ancak ne olursa olsun bu dilde `__del__` metoduna çok az durumda gerçek bir biçimde gereksinim duyulmaktadır.

Operatör Metotları (Operator Overloading)

Operatör metotları pek çok nesne yönelimli programlama dilinde bulunmaktadır. Örneğin C++'ta, C#'ta, Object Pascal'da, Swift'te operatör metotları vardır. Ancak Java'da operatör metotları yoktur.

Operatör metotları sayesinde mevcut Python operatörleri programcının kendi sınıflarıyla da çalışabilir hale getirilebilmektedir. Operatör metotları `__xxx__` biçiminde özel isimlerle oluşturulmaktadır. Hangi operatör için hangi ismin kullanılacağı aşağıdaki tabloda verilmiştir:

Operatör	İsim
+	<code>__add__</code> , <code>__radd__</code> , <code>__pos__</code>
*	<code>__mul__</code> , <code>__rmul__</code>
-	<code>__sub__</code> , <code>__neg__</code>
%	<code>__mod__</code> , <code>__rmod__</code>
**	<code>__pow__</code>
/	<code>__truediv__</code> , <code>__rtruediv__</code>
//	<code>__floordiv__</code> , <code>__rfloordiv__</code>
<code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>//=</code> , <code>%=</code>	<code>__iadd__</code> , <code>__isub__</code> , <code>__imul__</code> , <code>__truediv__</code> , <code>__floordiv__</code> , <code>__imod__</code>
<	<code>__lt__</code>
<code><=</code>	<code>__le__</code>
<code>==</code>	<code>__eq__</code>
<code>!=</code>	<code>__ne__</code>
>	<code>__gt__</code>
<code>>=</code>	<code>__ge__</code>
and	<code>__and__</code> , <code>__rand__</code>
not	<code>__not__</code>
or	<code>__or__</code> , <code>__ror__</code>
[index]	<code>__getitem__</code> , <code>__setitem__</code>
in	<code>__contains__</code>
len	<code>__len__</code>
str	<code>__str__</code>
int	<code>__int__</code>
float	<code>__float__</code>
bool	<code>__bool__</code>
complex	<code>__complex__</code>

Aslında operatör metotları yalnızca okunabilirliği artırmaktadır. Bazı ifadelerin daha doğal gözükmescini sağlar. a değişkeni X isimli bir sınıf türünden olmak üzere,

a + b

işleminin eşdeğeri:

a.`__add__`(b)

ya da:

X.`__add__`(a, b)

birimindedir. Yani biz a + b işlemini yaptığımızda arka planda aslında bu işlem `__add__` isimli metod tarafından yapılmaktadır. Yorumlayıcı bir operatörle karşılaşlığında önce operandların türlerine bakar. Soldaki operand built-in bir sınıf türünden değilse soldaki operandın ilişkin olduğu sınıfın ilgili operatör metodunu çağırır. O operatör metodundan elde edilen değer de o metodun geri dönüş değeri olacaktır. Örneğin:

```

class Number:
    def __init__(self, number):
        self.number = number

    def disp(self):
        print(self.number)

    def __add__(self, x):
        return Number(self.number + x.number)

    def __sub__(self, x):
        return Number(self.number - x.number)

    def __mul__(self, x):
        return Number(self.number * x.number)

    def __truediv__(self, x):
        return Number(self.number / x.number)

    def __str__(self):
        return str(self.number)

x = Number(10)
y = Number(20)
z = Number(2)

k = x + y * z
print(k)

```

Burada aslında,

`z = x + y`

işlemi ile,

`z = x.__add__(y)`

ya da:

`Number.__add__(x, y)`

işlemi tamamen eşdeğerdir.

Tabii Python'da farklı parametrik yapılara ilişkin aynı isimli metodlar (method overloading) yazılmadığı için farklı türlerle aynı işlemi yapan operatör metodlarının tek bir metot biçiminde yazılması gerekmektedir. Örneğin biz yukarıdaki operatör metodıyla bir Number nesnesiyle bir int nesneyi toplayamayız. Bunun için metod içerisinde tür kontrolü yapıp uygun bir stratejinin belirlenmesi gereklidir. Örneğin:

```

class Number:
    def __init__(self, number):
        self.number = number

    def __add__(self, x):
        if isinstance(x, (int, float)):
            return Number(self.number + x)
        return Number(self.number + x.number)

    def __sub__(self, x):
        if type(x) is int or type(x) is float:
            return Number(self.number - x)
        return Number(self.number - x.number)

```

```

def __mul__(self, x):
    if isinstance(x, (int, float)):
        return Number(self.number * x)
    return Number(self.number * x.number)

def __truediv__(self, x):
    if isinstance(x, (int, float)):
        return Number(self.number / x)
    return Number(self.number / x.number)

def __str__(self):
    return str(self.number)

x = Number(10)
y = Number(20)

z = x + y
print(z)

z = x + 10
print(z)

```

Karşılaştırma işlemleri için karşılaştırma operatör metodlarının yazılması gereklidir. Bu metodların geri dönüş değerlerinin bool türden olması anlamlıdır. Örneğin:

```

class Number:
    def __init__(self, number):
        self.number = number

    def disp(self):
        print(self.number)

    def __eq__(self, x):
        return self.number == x.number

    def __ne__(self, x):
        return self.number != x.number

    def __gt__(self, x):
        return self.number > x.number

    def __lt__(self, x):
        return self.number < x.number

    def __ge__(self, x):
        return self.number >= x.number

    def __le__(self, x):
        return self.number <= x.number

n = Number(10)
k = Number(10)

if n == k:
    print('Evet')
else:
    print('Hayır')

```

Daha önceden de belirtildiği gibi `__str__` isimli metod ilgili türden sınıf nesnesinin str türüne dönüştürülmesi gerektiği zaman kullanılmaktadır. Örneğin `print` bu çokbiçimli str metodunu çağrıp sonucu yazdırmaktadır.

Şimdi de rasyonel sayı işlemi yapan bir sınıf yazalım:

```
import math

class Rational:
    def __init__(self, a=0, b=1):
        if not isinstance(a, int) or not isinstance(b, int):
            raise TypeError('numerator and denominator must be int!')
        if b == 0:
            raise ValueError('denominator must not be 0!')
        self.a = a
        self.b = b

    self._simplify()

    def __repr__(self):
        if self.b == 1:
            return str(self.a)
        if self.a == 0:
            return '0'

        return f'{self.a}/{self.b}'

    def _simplify(self):
        gcd = math.gcd(self.a, self.b)

        self.a //= gcd
        self.b //= gcd

    def __add__(self, r):
        if isinstance(r, int):
            a = r * self.b + self.a
            b = self.b
        else:
            a = self.a * r.b + self.b * r.a
            b = self.b * r.b

        return Rational(a, b)

    __radd__ = __add__

    def __sub__(self, r):
        if isinstance(r, int):
            a = r * self.b - self.a
            b = self.b
        else:
            a = self.a * r.b - self.b * r.a
            b = self.b * r.b

        return Rational(a, b)

    __rsub__ = __sub__

    def __mul__(self, r):
        if isinstance(r, int):
            a = self.a * r
            b = self.b
        else:
            a = self.a * r.a
            b = self.b * r.b

        return Rational(a, b)
```

```

__rmul__ = __mul__

def __truediv__(self, r):
    if isinstance(r, int):
        a = self.a
        b = self.b * r
    else:
        a = self.a * r.b
        b = self.b * r.a

    return Rational(a, b)

__rtruediv__ = __truediv__

def __lt__(self, r):
    if isinstance(r, int):
        return self.a / self.b < r

    return self.a / self.b < r.a / r.b

def __le__(self, r):
    if isinstance(r, int):
        return self.a / self.b <= r

    return self.a / self.b <= r.a / r.b

def __gt__(self, r):
    if isinstance(r, int):
        return self.a / self.b > r

    return self.a / self.b > r.a / r.b

def __ge__(self, r):
    if isinstance(r, int):
        return self.a / self.b >= r

    return self.a / self.b >= r.a / r.b

def __eq__(self, r):
    if isinstance(r, int):
        return self.a / self.b == r

    return self.a / self.b == r.a / r.b

def __ne__(self, r):
    if isinstance(r, int):
        return self.a / self.b != r

    return self.a / self.b >= r.a != r.b

def __float__(self):
    return self.a / self.b

def __int__(self):
    return self.a // self.b

def __bool__(self):
    return self.a != 0

x = Rational(3, 2)
y = Rational(1, 2)

```

```

result = x + y * x
print(result)

result = x / 2
print(result)

result = float(x)
print(result)

result = int(x)
print(result)

z = Rational()
if z:
    print('Doğru')
else:
    print('Yanlış')

if x > y:
    print('x > y')
elif x < y:
    print('x < y')
elif x == y:
    print('x == y')

```

Yukarıdaki Rational sınıfında `__float__`, `__int__` ve `__bool__` tür dönüştürme operatör metotları da yazılmıştır. Bu tür dönüştürme operatör metotları biraz ileride ele alınmaktadır.

Aslında yukarıda yazdığımız Rational sınıfının benzer işlevlerine sahip olan Python'ın standart kütüphanesinde `fractions` modülü içerisinde `Fraction` isimli bir sınıf da bulunmaktadır. Örneğin:

```

import fractions

x = fractions.Fraction(2, 3)
y = fractions.Fraction(1, 2)
z = fractions.Fraction(1, 4)

result = x + y * z
print(result)

```

Anımsanacağı gibi `x` bir sınıf türünden değişken olmak izere `x <operator> y` işleminin eşdeğeri `x.__<operator>__(y)` biçimindeydi. Bazı operatörlerin değişme özelliğinin olduğunu da anımsayınız. İşte eğer operatör ifadesinde sol taraftaki değişken temel türlere ilişkin sağ taraftaki değişken bizim tarafımızdan yazılmış bir sınıfa ilişkinse bu işlem yapılamamaktadır. Örneğin:

```

class Sample:
    def __init__(self, a):
        self.a = a

    def __add__(self, s):
        if isinstance(s, Sample):
            return Sample(self.a + s.a)
        if isinstance(s, int):
            return Sample(self.a + s)

    def __str__(self):
        return str(self.a)

```

Böyle bir sınıf tanımlamasında biz bir `Sample` nesnesi ile bir `int` nesneyi toplayabiliriz:

```
a = 10
s = Sample(10)
result = s + a
print(result)
```

Buradaki toplama işleminin eşdeğeri `s.__add__(10)` biçimindedir. İşte toplama işleminde matematiksel değişme özelliği olsa da biz bunun tersi olan $10 + s$ işlemini yapamayız:

```
a = 10
result = a + s
print(result)
```

Çünkü bu işlemin eşdeğeri `a.__add__(s)` biçimindedir. `int` sınıfında bu işi yapabilecek bir operatör metodu yoktur. İşte buradaki değişme özelliğini sağlayabilmek için belli bir zamandan sonra Python'a bazı `__xxx__` operatör metodlarının `__rxxx__` biçiminde versiyonları eklenmiştir. Örneğin `__add__` metodunun ters versiyonu `__radd__`, `__mul__` operatör metodunun ters versiyonu ise `__rmul__` biçimindedir. Python yorumlayıcıları eğer sol taraftaki operand'a ilişkin sınıfta `__xxx__` operatör metodu yoksa bu durumda sağ taraftaki operand'a ilişkin sınıfta `__rxxx__` metodunu aramaktadır. Böylece örneğin:

```
a = A()
b = B()

a + b
```

işleminde eğer `A` sınıfında `__add__` metodu yoksa bu durumda `B` sınıfında `__radd__` metodu aranmaktadır. Eğer `B` sınıfında `__radd__` metodu bulunursa bu işlem `b.__radd__(a)` biçiminde gerçekleştirilmektedir. Örneğin:

```
class Sample:
    def __init__(self, a):
        self.a = a

    def __add__(self, s):
        if isinstance(s, Sample):
            return Sample(self.a + s.s)
        if isinstance(s, int):
            return Sample(self.a + s)

    def __radd__(self, a):
        return self.a + a

    def __str__(self):
        return str(self.a)
```

```
a = 10
s = Sample(10)

result = s + a # s.__add__(a)
print(result)

result = a + s # s.__radd__(a)
print(result)
```

Python'da fonksiyon çağrıma işlemi için fonksiyon çağrıma operatörü de bir operatör metodu olarak yazılabilir. Fonksiyon çağrıma operatörüne ilişkin operatör metodu `__call__` ismiyle yazılmalıdır. Bu operatör metodu sınıf türünden değişken ile fonksiyon çağrıma operatörü kullanıldığında devreye girer. Böylece sınıf nesnelerinin bir fonksiyon gibi davranışları sağlanmış olur. C++'ta böyle sınıflara "functor" denilmektedir. Örneğin:

```
class Sample:
    def __call__(self, x, y):
```

```

print('call: {}, {}'.format(x, y))
return x + y

s = Sample()
result = s(10, 20)
print(result)

```

Örneğin:

```

class Sample:
    def __init__(self, a):
        self.a = a

    def __call__(self, *args, **kwargs):
        print('a = {}'.format(self.a))
        print('args = {}'.format(args))
        print('kwargs = {}'.format(kwargs))

s = Sample(100)
s(10, 20, 30, xx=40, yy=50)      # eşdeğeri -> s.__call__(10, 20, 30, xx=40, yy=50)

```

Burada `__call__` metodunun `self` parametresi dışında iki parametre daha aldığına dikkat ediniz. Aslında bu iki parametre zorunlu değildir. Bu iki parametre sayesinde biz artık sınıf nesnesini her türlü argümanla çağrıma işlemeye sokabiliriz.

Fonksiyon yerine bir sınıf nesnesinin kullanılırsa durumsal bilgi oluşturulabilir. Yani ilgili nesne (...) operatörü ile çağrıldığında `__call__` metodu daha önce depolanmış olan nesnenin örnek öznitelikindeki bilgileri kullanabilir.

Örneğin:

```

class Sample:
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        return x ** self.n

s = Sample(3)
result = map(s, range(10))

for x in result:
    print(x)

```

Burada artık `s(val)` gibi bir çağrıının `val` değerinin herhangi bir kuvvetini alması sağlanmıştır. Şüphesiz burada `__call__` metodu iki parametrelî biçimde de tasarlanabilirdi. Ancak pek çok durumsal bilgi söz konus olduğunda ve bunlar programın çalışması sırasında değiştiğinde normal fonksiyon yerine çağrılabilen (callable) bir sınıf nesnesinin kullanılması daha uygun olmaktadır.

Python'da (...) operatörüyle kullanılabilen nesnelere "çağırlabilen (callable)" nesneler denilmektedir. Fonksiyonlar, metodlar ve `__call__` metodu olan sınıflar türünden nesneler tipik çağrılabilen nesnelerdir.

Python'da bir sınıf türünden referansın [...] operatörleriyle kullanılabilmesi için ilgili sınıfın `__getitem__` ve `__setitem__` isimli özel metodlara sahip olması gereklidir. Örneğin `a` bir sınıf türünden değişken belirtmek üzere:

```
b = a[index]
```

işleminin eşdeğeri:

```
b = a.__getitem__(index)
```

birimindedir. Bener biçimde:

```
a[index] = b
```

işleminin eşdeğeri de:

```
a.__setitem__(index, b)
```

birimindedir. Yani biz sınıflarımıza bu metodları yazarak [...] operatör desteği verebiliriz. Örneğin:

```
class MyList:  
    def __init__(self, array):  
        self.array = array  
  
    def __getitem__(self, key):  
        return self.array[key]  
  
    def __setitem__(self, key, value):  
        self.array[key] = value  
  
    def __len__(self):  
        return len(self.array)  
  
    def __str__(self):  
        return str(self.array)  
  
ml = MyList([10, 20, 30, 40])  
  
for i in range(len(ml)):  
    ml[i] = ml[i] * 2  
  
print(ml)
```

Örneğin:

```
class Sample:  
    def __init__(self):  
        self.dict = {}  
  
    def __setitem__(self, key, value):  
        self.dict[key] = value  
  
    def __getitem__(self, key):  
        return self.dict[key]  
  
  
s = Sample()  
s['ali'] = 300  
s['veli'] = 200  
s[100] = 'kazım'  
  
print(s['ali'])  
print(s['veli'])  
print(s[100])
```

Köşeli parantez içerisindeki index ifadesi '' atomuyla ayrıstırılmış birden fazla ifadeden oluşabilir. Bu durumda bu index ifadeleri __getitem__ ve __setitem__ metodlarına demet biçiminde aktarılmaktadır. Python'da x indekslenebilen bir nesne olmak üzere x[a, b, c, d, e] gibi bir ifade ile x[(a, b, c, d, e)] ile tamamen eşdeğeremdir. Böylece programcı çok boyutlu dizi izlenimini verecek sınıflar yazabilirdir. Örneğin:

```

class Matrix:
    def __init__(self, nrows, ncols):
        self.matrix = [[0] * ncols for i in range(nrows)]

    def __getitem__(self, index):
        return self.matrix[index[0]][index[1]]

    def __setitem__(self, index, val):
        self.matrix[index[0]][index[1]] = val

    def __str__(self):
        s = ''
        for i in range(len(self.matrix)):
            for k in range(len(self.matrix[0])):
                if k != 0:
                    s += ' '
                s += str(self.matrix[i][k])
            s += '\n'

        return s

m = Matrix(5, 5)

for i in range(5):
    for k in range(5):
        m[i, k] = i + k

for i in range(5):
    for k in range(5):
        print(m[i, k], end=' ')
    print()

print()
print(m)

```

Pekiyi sınıfı dilimleme desteği nasıl verilmektedir? Yukarıdaki `__getitem__` ve `__setitem__` metotları dilimleme desteğine sahip değildir. Bilindiği gibi Python'da dilimleme yalnızca köşeli parantezler içerisinde yapılmaktadır. İşte Python yorumlayıcısı bir dilimlemeyle karşılaşlığında bu dilimleme ifadesi için slice isimli bir sınıf nesnesi yaratıp aslında o nesneyi `__getitem__` ya da `__setitem__` metotlarına geçirmektedir. slice sınıfının aşağıdaki gibi bir başlangıç metodu vardır:

```
slice(start, stop[, step])
```

Buradaki slice fonksiyonu iki argümanla ya da üç argümanla çağrılabılır. Örneğin:

```
>>> s = slice(3, 7)
>>> type(s)
<class 'slice'>
>>> print(s)
slice(3, 7, None)
```

Bu durumda örneğin:

```
b = a[3:5]
```

ifadesi ile:

```
b = a[slice(3, 5)]
```

ifadesi ve:

```
b = a.__getitem__(slice(3, 5))
```

ifadesi tamamen eşdeğerdir. Benzer biçimde:

```
a[3:5] = b
```

ifadesi ile:

```
a.__setitem__(slice(3, 5), b)
```

tamamen eşdeğerdir. Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> b = a[3:5]
>>> b
[40, 50]
>>> b = a.__getitem__(slice(3, 5))
>>> b
[40, 50]
```

Örneğin:

```
>>> a = [10, 20, 30, 40, 50]
>>> a[3:5] = [100, 200]
>>> a
[10, 20, 30, 100, 200]
>>> a = [10, 20, 30, 40, 50]
>>> a.__setitem__(slice(3, 5), [100, 200])
>>> a
[10, 20, 30, 100, 200]
```

Pekiyi slice nesnesi içerisindeki start, stop ve step değerleri nasıl geri alınmaktadır? İşte slice sınıfının start, stop ve step isimli örnek öznitelikleri bize bu bilgileri vermektedir. Örneğin:

```
>>> s = slice(1, 10, 2)
>>> s.start
1
>>> s.stop
10
>>> s.step
2
```

Eğer slice nesnesi oluşturulurken stop ya da step değeri verilmemişse bu öznitelikler bize None değeri vermektedir.

Örneğin:

```
>>> s = slice(10)
>>> print(s.start)
None
>>> print(s.stop)
10
>>> print(s.step)
None
```

slice sınıfının indices isimli metodunu bizden bir argüman alır. start, verdiğimiz argüman ve step değerini içeren üçlü bir demet bize verir. Örneğin:

```
>>> s = slice(1, 10, 2)
>>> s.indices(5)
```

```
(1, 5, 2)
>>> s.indices(10)
(1, 10, 2)
```

Eğer slice nesnesinde biz start ya da stop değerini vermezsek artık demette onlar None değerini almazlar. Örneğin:

```
>>> s = slice(10)
>>> s.indices(10)
(0, 10, 1)
```

Şimdi de kendi sınıfımıza dilimleme desteği verelim. Ancak bunu yaparken dilimleme kullanmayacağız:

```
class MyArray:
    def __init__(self, a):
        self.a = a

    def __getitem__(self, index):
        if isinstance(index, slice):
            return [self.a[i] for i in range(*index.indices(len(self.a)))] # return
self.a[index]
        return self.a[index]

    def __setitem__(self, index, val):
        if isinstance(index, slice):
            k = 0
            for i in range(*index.indices(len(self.a))):
                self.a[i] = val[k]
                k += 1
        else:
            self.a[index] = val

    def __len__(self):
        return len(self.a)

    def __str__(self):
        return str(self.a)

ml = MyArray([10, 20, 30, 40, 50])
result = ml[3:5]
print(result)

result = ml[3]
print(result)

ml[3:5] = [100, 200]
print(ml)

ml[3] = 300
print(ml)
```

Programın çıktısı şyle olacaktır:

```
[40, 50]
40
[10, 20, 30, 100, 200]
[10, 20, 30, 300, 200]
```

Burada dilimle yapıldığında asıl dizi içerisindeki elemanların liste içlemiyle elde edildiğine dikkat ediniz:

```
return [self.a[i] for i in range(*index.indices(len(self.a)))]
```

Burada indices metodu ile start, stop ve step değerleri bir demet olarak alınmış ve * argümanıyla yeniden range fonksiyonuna verilmiştir.

Bir sınıf türünden değişkeni int türüne, float türüne, bool türüne ve complex türüne dönüştürebilmek için sırasıyla sınıfın __int__, __float__, __bool__ ve __complex__ metodları kullanılmaktadır. Başka bir deyişle int, float, bool ve complex fonksiyonlarına biz kendi sınıf nesnemizi verirsek bu fonksiyonlar aslında sınıfımızın __int__, __float__, __bool__ ve __complex__ metodlarını çağırmaktadır. Örneğin:

```
class Number:  
    def __init__(self, val):  
        self.val = val  
  
    def __int__(self):  
        return int(self.val)  
  
    def __float__(self):  
        return float(self.val)  
  
    def __bool__(self):  
        return bool(self.val)  
  
    def __complex__(self):  
        return complex(self.val)  
  
n = Number(10)  
  
val = int(n)  
print(val)  
  
val = float(n)  
print(val)  
  
val = bool(n)  
print(val)  
  
val = complex(n)  
print(val)
```

Anımsanacağı gibi bileşik atama operatörleri sol taraftaki değişkeni sağ taraftaki değişkenle işleme sokup sonucu yine sol taraftaki değişkene atamaktadır. Bizim +=, -=, *=, /=, //=, %= gibi operatörleri kullanmamız için bu operatörlere ilişkin operatör metodu yazmamıza çoğu kez gerek yoktur. Örneğin a ve b aynı sınıf türünden olmak üzere:

```
a += b
```

İşlemi için eğer += operatörü için ayrıca metod yazılmamışsa bu işlem aşağıdaki işlem eşdeğерdir:

```
a = a.__add__(b)
```

Ancak biz bileşik atama operatörleri için özel başka birtakım işlemlerin yapılmasını isteyebiliriz. Bu durumda bileşik atama operatörleri için de operatör metodu yazmamız gereklidir. Bileşik atama operatörlerine ilişkin operatörler __ixxx__ ismiyle yazılırlar. Şimdi a ve b'nin ilişkin olduğu sınıfta __iadd__ metodunun yazılmış olduğunu füşünelim. Bu durumda artık a += b işlemi a = a.__iadd__(b) biçiminde ele alınacaktır. Örneğin:

```
class Number:  
    def __init__(self, val):  
        self.val = val  
  
    def __add__(self, x):  
        return Number(self.val + x.val)
```

```

def __iadd__(self, x):
    return Number(self.val + x.val + 1)

def __repr__(self):
    return str(self.val)

a = Number(10)
b = Number(20)

a = a + b
print(a)

a = Number(10)
b = Number(20)

a += b
print(a)

```

Burada artık `__iadd__` operatör metodu yazıldığı için `a += b` işleminde bu operatör metodu çağrılmacaktır. Bu operatör metodunun iki değerin toplamından 1 fazlasını oluşturduğuna dikkat ediniz. Şüphesiz pek çok durumda bileşik atama operatörlerinin yazılmasına gerek yoktur. Bu gereklilik seyrek bir biçimde ortaya çıkmaktadır. Örneğin list sınıfında `+=` operatörünün ekleme yaptığıni anımsayınız:

```

>>> a = [1, 2, 3]
>>> b = [10, 20]
>>> id(a)
2356196347336
>>> a = a + b
>>> id(a)
2356201387528
>>> a
[1, 2, 3, 10, 20]
>>> a = [1, 2, 3]
>>> b = [10, 20]
>>> id(a)
2356201387784
>>> a += b
>>> id(a)
2356201387784
>>> a
[1, 2, 3, 10, 20]

```

Sınıfların static Metotları ve Sınıf Metotları

Bazı fonksiyonlar bir sınıfın konusuyla ilgili işlemler yaparlar ancak herhangi bir sınıf örnek özniteliğini (instance attribute) kullanmazlar. Yani bunların yalnızca mantıksal bakımından sınıf ile ilişkileri vardır. Örneğin Date isimli bir sınıf tarih işlemleri yapıyor olsun. Bu sınıfın day, month ve year isimli örnek özniteliklerinin olduğunu düşünelim:

```

class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
    ...

```

Şimdi biz belli bir yılın artık yıl olup olmadığını tespit eden bir fonksiyon yazmak isteyelim. Bu fonksiyonun aslında tarih işlemleriyle yalnızca mantıksal bir ilgisi vardır. Böyle bir fonksiyon global düzeyde yazılabilir. Ancak bu mantıksal ilgiden dolayı bu fonksiyonun Date sınıfı içerisinde bir metot biçiminde oluşturulması tercih edilir. İşte böylece bir sınıfa dahil olan fakat o sınıfın hiçbir örnek özniteliğini kullanmayan bir metotla karşılaşmış oluruz. Böyle metotlar diğer pek çok nesne yönelimli programlama dilinde "static metot" olarak isimlendirilmektedir. İşte Python'da da bu

anlamda static metotlar vardır. static metotların bir self parametresi olmaz. Bunlar yalnızca mantıksal bakımdan sınıfla ilişkilidirler. static metotlar @staticmethod isimli bir dekoratörle oluşturulmaktadır. Örneğin:

```
class Date:  
    def __init__(self, day, month, year):  
        self.day = day  
        self.month = month  
        self.year = year  
  
    @staticmethod  
    def is_leap_year(year):  
        return year % 400 == 0 or year % 4 == 0 and year % 100 != 0  
  
result = Date.is_leap_year(2000)  
print('Artık' if result else 'Artık değil')
```

statik metotların self parametresi içermediğine ve bunların sınıf ismiyle çağrıldığına dikkat ediniz. Her ne kadar static metotlar aslında sınıf ismiyle çağrılıyor olsalar da tıpkı C++ ve Java'da olduğu gibi static metotlar aynı zamanda sınıf türünden bir nesneyle de çağrılabilmektedir. Örneğin:

```
class Date:  
    def __init__(self, day, month, year):  
        self.day = day  
        self.month = month  
        self.year = year  
  
    def disp(self):  
        print('{}/{}/{}'.format(self.day, self.month, self.year))  
  
    @staticmethod  
    def isLeapYear(year):  
        return year % 400 == 0 or year % 4 == 0 and year % 100 != 0  
  
date = Date(10, 12, 2007)  
  
date.disp()          # örnek metodun çağrılmaması için asıl biçim  
Date.disp(date)     # örnek metodun çağrılmaması için alternatif biçim  
  
result = Date.isLeapYear(2000)      # static metodun çağrılmaması için asıl biçim  
print('Artık' if result else 'Artık değil')  
  
result = date.isLeapYear(2000)      # static metodun çağrılmaması için alternatif biçim  
print('Artık' if result else 'Artık değil')
```

Statik bir metodu bir nesneyle çağrıdığımızda sanki o metot o nesnenin özniteliklerini kullanacakmış gibi yanlış bir algı oluşturmaktadır. Oysa yorumlayıcı aslında metodun o nesnesin türüne ilişkin sınıf ismiyle çağrıldığını varsayımaktadır. Başka bir deyişle s bir sınıf türünden nesne ve foo da static bir metot olmak üzere:

s.foo()

ifadesi ile,

type(s).foo()

ifadesi eşdeğerdir.

Parametreli static metotlar söz konusu olabilir. Ancak bu parametrelerin ilki ismi self olsa bile self anlamına gelmez. Örneğin:

```

class Sample:
    def foo(self):
        print('foo')

    @staticmethod
    def bar(self):
        print('bar')

s = Sample()
s.foo()      # Sample.foo(s)
s.bar(10)    # geçerli, 10 burada self'e atanacak
s.bar()      # exception, çünkü self sıradan bir parametre

```

Tabii bu örnekte bar metodundaki paramerenin self biçiminde isimlendirilmesi de kötü bir tekniktir. Çünkü metot static olduğu için artık self parametresine bu anlamda aktarım yapılmayacaktır.

Statik metodların yanı sıra Python'da aynı zamanda bir de sınıf metodları (class methods) kavramı vardır. Sınıf metodları mantıksal bakımdan static metodlara çok benzerdir. Sınıf metodları @classmethod dekoratörüyle oluşturulmaktadır. Sınıf metodları static metodlardan farklı olarak zorunlu bir ilk parametreye sahiptir. Bu ilk parametreye geleneksel olarak cls ismi verilmektedir. Örneğin:

```

class Sample:
    def __init__(self):      #örnek metodu
        self.a = 10

    @staticmethod           # statik metot
    def bar():
        print('bar')

    @classmethod            # sınıf metodu
    def tar(cls):
        print('tar')

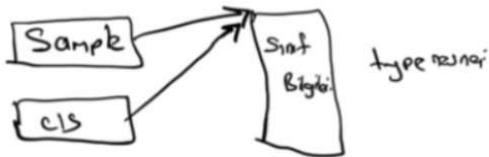
```

Sınıf metodlarının cls parametresi o sınıf metodu hangi sınıfa ilişkinse o sınıfın type nesnesini almaktadır. Anımsanacağı gibi sınıf isimleri de Python'da aslında birer değişkendir. Biz bir sınıf ismini kullandığımızda aslında o sınıfın bilgilerini içeren type isimli bir sınıf nesnesi üzerinde işlem yapmış oluruz. Örneğin:

```

class Sample:
    @classmethod
    def foo(cls):
        # ...

```



biz foo isimli bir sınıf metodunu aşağıdaki gibi çağrırmış olalım:

```
Sample.foo()
```

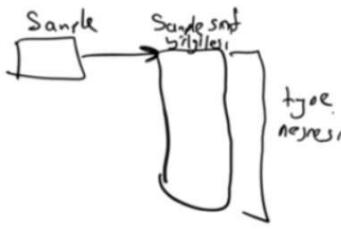
Aslında Python yorumlayıcısı foo metoduna Sample değerini parametre olarak geçirmektedir.

```

class Sample:
    @classmethod
    def foo(cls):
        pass

```

Sample.foo()



Pekiyi biz sınıf metodlarına geçirilen ve sınıfın type nesnesini belirten cls parametresine neden gereksinim duyabiliyoruz? İşte bazı uygulamalarda (bazılarıyla ileride karşılaşabiliriz) bu parametreye gereksinim duyulabilemektedir. Tabii bu parametre kullanılmayacak olduktan sonra statik metod ile sınıf metodları arasında bir fark olusmaz.

Sınıf metodları da statik metodlar gibi normal olarak sınıf ismiyle çağrırlar. Ancak bunların da aynı sınıf türünden bir değişkenle çağrılması yine mümkün değildir. Tabii bu durumda yine sınıf metodlarına bu değişken değil, nesnenin ilişkin olduğu sınıfın type nesnesi cls parametresi olarak aktarılır. Örneğin:

```

class Sample:
    def __init__(self):      #örnek metod
        self.a = 10

    def foo(self):          # örnek metod
        print('foo')

    @staticmethod
    def bar():              # statik metod
        print('bar')

    @classmethod            # sınıf metodu
    def tar(cls):           # Sınıf tar metodu, Sample cls pa
        print('tar')

s = Sample()
s.foo()                  # örnek foo metodu çağrılr

Sample.bar()              # statik bar metodu
Sample.tar()              # Sınıf tar metodu, Sample cls pa

s.bar()                  # statik bar metodu
s.tar()                  # sınıf tar metodu, Sample cls parametresi olarak geçirilir

```

Pekiyi sınıf içerisindeki bir metoda biz hiç parametre yazmasak fakat metodun başında da @staticmethod dekoratörünü bulundurmasak bu metod zaten statik olmaz mı? Örneğin:

```

class Sample:
    def foo():            # bu nasıl metod?
        print('foo')

```

İşte bu bildirim Python'ın 3'ten önceki versiyonlarında geçersiz kabul ediliyordu. Ancak Python'ın 3'lü versiyonlarıyla bu bildirim de static etkisi yaratmaktadır. Başka bir deyişle eğer metodun hiçbir parametresi yoksa bu metod Python'ın 3'lü versiyonlarıyla birlikte statik kabul edilmektedir. Ancak bu biçimdeki statik metodlar sınıf türünden referanslarla çağrılamamaktadır. Örneğin:

```

class Sample:
    def foo():            # bu nasıl metod?
        print('foo')

Sample.foo()             # geçerli
s = Sample()

```

```
s.foo()          # error!
```

Tabii bizim static metodları parametresiz olsalar bile @staticmethod ile dekore etmemiz iyi bir tekniktir.

Ancak sınıf içerisindeki bir metodun en az bir parametresi varsa ve bu metod @staticmethod ya da @classmethod biçiminde dekore edilmediyse ilk parametre her zaman self olarak değerlendirilmektedir. (Tabii ilk parametresinin "self" biçiminde isimlendirilmesi zorunlu değildir.) Örneğin:

```
class Sample:  
    def foo(a):  
        print('foo')  
  
Sample.foo()      # error!  
s = Sample()  
s.foo()          # geçerli
```

Python'da Dekoratörler

Python'da nesne yönelimli programla tekniğindeki dekoratör kalıbı özel bir sentaksla dilin içerisinde entegre edilmiştir. Dekoratörler "fonksiyon ve metod dekoratörleri" ve "sınıf dekoratörleri" olmak üzere ikiye ayrılmaktadır. Dekoratörleri belirtmek için "@" karakteri kullanılır. "@" karakterine bitişik olan isme dekoratör denilmektedir. Örneğin:

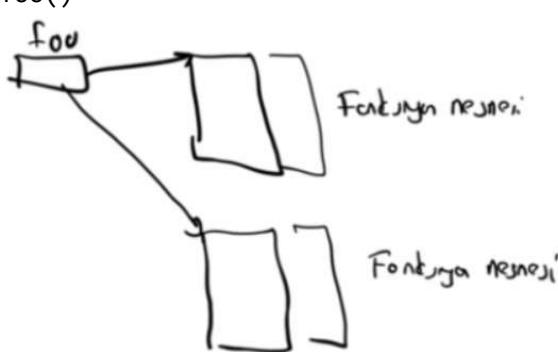
```
@foo  
@classmethod
```

birer dekoratördür.

Fonksiyon ve Metot Dekoratörleri

Daha önceden de belirtildiği gibi Python'da fonksiyonlar, metodlar ve sınıflar aslında sıradan birer değişken ismidir. Bunlar fonksiyon nesnelerinin, metod nesnelerinin ve type nesnelerinin adreslerini tutan birer referansdır. Bu nedenle Python'da aynı fonksiyon ya da metodu ikinci kez bildirmek adeta bu değişkene başka bir değer atamak gibidir ve tamamen normal bir durumdur. Örneğin:

```
def foo():  
    print('foo')  
  
foo()  
  
def foo():  
    print('other foo')  
  
foo()
```



Python'da fonksiyon dekoratörü oluşturmak için "@" simbolünün yanına bitişik olarak bir fonksiyon ismi getirilmelidir. Örneğin:

```
@foo  
def bar():  
    print('bar')
```

Burada @foo dekoratöründe foo bir fonksiyondur. bar fonksiyonu da bu dekoratörle dekore edilmiştir. Pekiyi bu dekorasyon neye yol açmaktadır? Aslında fonksiyonların bu biçimde dekora edilmesinin basit bir eşdeğeri vardır. Şöyle ki:

```
@foo  
def bar():  
    print('bar')
```

Bildirimini gören Python yorumlayıcısı bu bildirimi aşağıdakiyle eşdeğer kabul etmektedir:

```
def bar():  
    print('bar')  
  
bar = foo(bar)
```

Pekiyi burada ne olmaktadır? Bu işlemi iki aşama olarak açıklayabiliriz:

- 1) Burada bar metodu foo metoduna argüman yapılarak foo metodu çağrılmaktadır. Bu çağrıma yalnızca bildirimin bulunduğu yerde bir kez yapılır.
- 2) Bu çağrıının geri dönüş değeri yeniden bar'a atanmaktadır. Böylece artık biz bar fonksiyonunu çağrıdığımızda aslında foo fonksiyonun geri verdiği fonksiyonu çağrımiş oluruz. Genellikle (ama her zaman değil) dekoratör fonksiyonları bir şeyle yaparak parametresiyle aldığı fonksiyonun aynısına geri dönmektedir. Böylece örneğimizde aslında biz bar metodunu çağrıdığımızda yine bar metodu çalışacaktır. Fakat ondan önce foo'daki işlemler de yalnızca bir kez yapılmış olacaktır. Örneğin:

```
def foo(f):  
    print('foo')  
    return f  
  
@foo  
def bar():  
    print('bar')  
  
# eşdeğeri -> bar = foo(bar)  
  
bar()
```

Örneğin:

```
class Sample:  
    @staticmethod  
    def foo():  
        print('static foo method')
```

Bu işlemin eşdeğeri şöyledir:

```
class Sample:  
    def foo():  
        print('static foo method')  
  
foo = staticmethod(foo)  
  
Sample.foo()
```

Pekiyi dekoratörler neden kullanılırlar? İşte bazen bir metot çağrılığında araya başka bir metodu sokmak gerekebilir. Araya sokulan bu metot (yani dekoratör metodu) bazı ekstra işlemler yapıp bize aynı ya da farklı bir metodu verebilir.

Eğer asıl fonksiyonu her çağrılığımızda yeniden başka bir işlemin yapılmasını istiyorsak dekoratör fonksiyonunda biz başka bir fonksiyona geri dönmeliyiz. O fonksiyon da asıl fonksiyonu çağırmalıdır. Bu durum tipik olarak iç fonksiyonlarla sağlanabilmektedir. Örneğin:

```
def foo(f):
    def g():
        print('araya girilen kod')
        f()
    return g
```

```
@foo
def bar():
    print('bar')

bar()
bar()
```

Bu biçimin önceki biçimden farkına dikkat ediniz:

```
def foo(f):
    print('araya girilen kod')
    return f

@foo
def bar():
    print('bar')

bar()
bar()
```

Özetle durumu şöyle açıklayabiliriz: Biz dekore edilmiş fonksiyonu çağrılığımızda aslında dekoratör fonksiyonunu çağrılmış oluruz. O da dolaylı olarak birşeyler yaptıktan sonra dekore edilmiş fonksiyonu çağrımaktadır. Örneğin bir fonksiyon çağrılığında aynı zamanda bir log tutulmasını isteyelim. Bunu basit bir biçimde bir dekoratörle yapabiliriz:

```
import datetime as dt

def mylog(f):
    file = open('logfile.txt', 'w')
    def log():
        file.write('function called at ' + str(dt.datetime.now()) + '\n')
        f()
    return log

@mylog
def foo():
    print('foo')

foo()
foo()
foo()
```

Dekoratörler yukarıdaki örneklerde olduğu gibi eğer orijinal fonksiyonu yeniden çağrıracaklarsa orijinal fonksiyonun parametreye sahip olması durumunda çağrılmıştır. Örneğin:

```
def foo(f):
    print('foo')

    def g(*args):
        print('g')
        f(*args)

    return g

@foo
def bar(a, b, c):
    print(f'bar: {a}, {b}, {c}')

bar(10, 20, 30)
bar(50, 60, 70)
```

Tabii eğer orijinal fonksiyon **'lı parametre de alıyorsa bu durumda orijinal fonksiyonun çağrılması için araya giren fonksiyonun da **'lı parametre alması gereklidir. Örneğin:

```
def foo(f):
    print('foo')

    def g(*args, **kwargs):
        print('g')
        f(*args, **kwargs)

    return g

@foo
def bar(a, b, c, **kwargs):
    print(f'bar: {a}, {b}, {c}, {kwargs}')

bar(10, 20, 30, xx=100, yy=200)
bar(50, 60, 70, zz=300, kk=400)
```

Bu durumda dekoratörün geri döndüreceği fonksiyonun parametrik yapısı genel olarak aşağıdaki gibi olabilir:

```
def g(*args, **kwargs):
    pass
```

Bazen dekoratör fonksiyonu aslında bir fonksiyon değil bir sınıf da olabilmektedir. Böylece dekoratör fonksiyonun çağrılması aslında bir sınıf nesnesinin oluşturulmasına yol açmaktadır. Örneğin:

```
class foo:
    def __init__(self, f):
        print('ilk işlemler')
        self.f = f

    def __call__(self):
        print('araya giren işlemler')
        self.f()

@foo
def bar():
    print('bar')

bar()      # aslında bar.__call__() çağrısı yapılmıyor
```

Burada aslında:

```
@foo
def bar():
    print('bar')
```

işleminin eşdeğeri:

```
def bar():
    print('bar')

bar = foo(bar)
```

birimindedir. Dolayısıyla aslında burada foo sınıfı tründen bir nesne yaratılmıştır. Bu nesnenin yaratımı sırasında `__init__` metodunun çağrılacagina dikkat ediniz. Yaratılan bu nesneden sonra artık bar bir metot değil foo sınıfı türünden bir nesne (yani referans) haline gelmiştir. Yani artık biz `bar()` çağrısını yaptığımızda aslında foo sınıfının `__call__` metodu çağrılacaktır. Burada yapılanların iç içe fonksiyon sisteminde yapılanlarla benzerliğine dikkat ediniz. Tabii sınıflar bize daha esnek çalışma olanağı sunmaktadır.

Yine dekore edilen fonksiyonun parametreli olması durumunda çağrıının yapılabilmesi için genel olarak `__call__` metodunun `*args` ve `**kwargs` parametrelerine sahip olması gereklidir. Örneğin:

```
class foo:
    def __init__(self, f):
        self.f = f

    def __call__(self, *args, **kwargs):
        print('araya giren kod')
        self.f(*args, **kwargs)

@foo
def bar(a, b, c):
    print(f'foo: {a}, {b}, {c}')
# foo = bar(foo)

bar(1, 2, 3)
bar(4, 5, 6)
```

Dekoratörler yalnızca global fonksiyonlara değil bir sınıfın metodlarına da uygulanabilmektedir. Örneğin:

```
def bar(f):
    def g(*args, **kwargs):
        print('araya giren kod')
        f(*args, **kwargs)

    return g

class Sample:
    def __init__(self, a):
        self.a = a

    @bar
    def foo(self):
        print(self.a)
# foo = bar(foo)

s = Sample(10)
s.foo()      # Sample.foo(s, 10)
```

Metot dekoratörleri sınıflarla gerçekleştirilememektedir. Aşağıdaki gibi bir gerçekleştirim istenen işlemleri yapamayacaktır:

```

class bar:
    def __init__(self, f):
        self.f = f

    def __call__(self, *args, **kwargs):
        print('araya giren kod')
        self.f(*args, **kwargs)

class Sample:
    def __init__(self, a):
        self.a = a

    @bar
    def foo(self):
        print(self.a)
        # foo = bar(foo)

s = Sample(10)
s.foo()      # Sample.foo(s, 10)

```

Burada s.foo() çağrısında artık foo bir metot değil fonksiyon durumundadır. Dolayısıyla bu foo fonksiyonuna self parametresi olarak s aktarılmaz.

Sınıf Dekoratörleri

Fonksiyonlar ve metodlar gibi sınıflar da aynı mantık çerçevesinde dekore edilebilmektedir. Örneğin:

```

def foo(c):
    print('foo')
    return c

@foo
class Sample:
    def disp(self):
        print('disp')

# Sample = foo(Sample)

s = Sample()
s.disp()

```

Burada aslında Sample ismi yine aynı sınıf bilgilerini göstermektedir. Ancak bir foo çağrıması da yapılmıştır. Biz istersek her sınıf nesnesi yaratılacağı zaman bir kodun çalışmasını da sağlayabiliyoruz. Bu işlem yine iç bir fonksiyon kullanılarak ya da bir sınıf kullanılarak yapılabilir. Örneğin:

```

def foo(c):
    def bar():
        print('bar')
        return c()
    return bar

@foo
class Sample:
    def disp(self):
        print('disp')

# Sample = foo(Sample)

s = Sample()
s.disp()

```

```
k = Sample()  
k.disp()
```

Burada `Sample()` ifadesi aslında bar fonksiyonun çağrılmasına yol açmaktadır. bar fonksiyonu da araya bir kod girdikten sonra `c()` ifadesi ile aslında `Sample` türünden bir nesnenin yaratılmasına yol açmaktadır. Biz burada bar fonksiyonunu * parametreli olarak da düzenleyebilirdik. Örneğin:

```
def foo(c):  
    def g(*args, **kwargs):  
        print('g')  
        return c(*args, **kwargs)  
    return g  
  
@foo  
class Sample:  
    def __init__(self, a, b):  
        print('Sample instance created')  
        self.a = a  
        self.b = b  
  
# Sample = foo(Sample)  
  
s = Sample(10, 20)  
  
print(s.a, s.b)
```

Tabii aynı işlevsellikler callable nesne olarak sınıfın kullanılması yoluyla da yapılabildi:

```
class Mample:  
    def __init__(self, cls):  
        self.cls = cls  
  
    def __call__(self, *args, **kwargs):  
        print('Her Sample nesnesi yaratıldığında araya giren kod')  
        return self.cls(*args, **kwargs)  
  
def foo(cls):  
    return Mample(cls)  
  
@foo  
class Sample:  
    def __init__(self, a):  
        self.a = a  
  
    def bar(self):  
        print('bar')  
  
s = Sample(10)  
s.bar()
```

Burada aslında `Sample` ismi `Mample` sınıfı türünden bir nesneyi belirtmektedir. Dolayısıyla `Sample(10)` gibi bir ifade aslında `Mample` sınıfının `__call__` metodunun çağrılmasına yol açmaktadır. Örneğimizde `Mample` sınıfının `__call__` metodu da `Sample` sınıfı türünden bir nesne geri döndürmüştür. Böylece programı aslında `Sample(10)` gibi bir ifade soncunda yine `Sample` sınıfı türünden bir nesne elde etmiştir. Ancak bu nesnenin yaratımı sırasında bir araya girme kodu çalıştırılabilir olmuştur.

Parametreli Dekoratörler

Dekoratörler parametreli de olabilmektedir. Örneğin log işlemi yapan bir dekoratör log bilgilerine yazacağı dosyanın yol ifadesini parametre olarak alabilmektedir:

```
@log('test.txt')
def foo():
    print('foo')
```

```
foo()
foo()
```

Burada @log dekoratörü parantezler içerisinde parametre de almıştır. Aşağıdaki gibi bir parametreli dekoratör kullanılmış olsun:

```
@bar('ali', 'veli', 'selami')
def foo():
    pass
```

Bunun yorumlayıcı için eşdeğeri şöyledir:

```
def foo():
    pass

foo = bar('ali', 'veli', 'selami')(foo)
```

Burada dekoratör fonksiyonun verilen argümanlarla çağrıldığına, onun geri dönüş değeri olarak elde edilen fonksiyona da dekora edilen fonksiyonun verildiğine dikkat ediniz. Yani bu durumda bir kademe daha çağrıma uygulanmıştır. O halde bizim bu örnekteki bar metodunda bir fonksiyon geri döndürmemiz gereklidir. Geri döndürülen bu fonksiyonun bir parametresi olmalıdır. Geri döndürülen fonksiyon da başka bir fonksiyon geri döndürmelidir. Örneğin:

```
def bar(a, b, c):
    print(f'bar çağrıldı: {a}, {b}, {c}')
    def tar(f):
        print(f'tar çağrıldı: {a}, {b}, {c}')
        def zar():
            print(f'Araya girecek kod: {a}, {b}, {c}')
            f()
        return zar
    return tar

@foo('ali', 'veli', 'selami')
def foo():
    print('foo çağrıldı')

# foo = bar('ali', 'veli', 'selami')(foo)

foo()
foo()
foo()
```

Kod çalıştırıldığında aşağıdaki çıktı elde edilecektir:

```
bar çağrıldı: ali, veli, selami
tar çağrıldı: ali, veli, selami
Araya girecek kod: ali, veli, selami
foo çağrıldı
Araya girecek kod: ali, veli, selami
foo çağrıldı
Araya girecek kod: ali, veli, selami
```

foo çağrıldı

Bu örnekte aslında önce bar fonksiyonu çağrılmıştır. bar fonksiyonun geri döndürdüğü fonksiyona foo fonksiyonu argüman olarak geçirilmiş ve oradan elde edilen fonksiyon yeniden foo değişkenine atanmıştır. Dolayısıyla yukarıdaki örnekte programcı foo'yı çağrılığında aslında zar fonksiyonunu çağrılmış olmaktadır. Tabii bu zar fonksiyonu üst fonksiyonların yerel değişkenlerine ve parametre değişkenlerine erişebildiğine göre dekortaör parametrelerini kullanabilecek durumdadır. Burada yapılan işlemleri adım adım şöyle açıklayabiliriz:

- 1) bar fonksiyonu dekoratör parametreleriyle çağrılır. bar fonksiyonu bize geri dönüş değeri olarak tar fonksiyonunu verir.
- 2) tar fonksiyonuna foo fonksiyonu argüman olarak geçirilip geri dönüş değeri foo'ya atanmıştır. Yani tar fonksiyonunun geri dönüş değeri olan zar aslında foo çağrılığında çağrılacak fonksiyon olmaktadır.
- 3) Artık foo çağrılığında zar fonksiyonu çağrılacaktır. bar, tar ve zar fonksiyonları da dekoratör parametrelerine erişip onları kullanabilecektir.

Parametreli dekoratörler de sınıflarla oluşturulabilmektedir. Ancak burada böyle bir örnek vermeyeceğiz.

Örneğin dekore edildiği fonksiyon çağrıldıkça çağrıılma zamanlarını belirtilen dosyaya yazdırın log isimli dekoratör şöyle yazılabılır:

```
import datetime
import time

def log(path):
    file = open(path, 'w')
    def bar(f):
        def tar(*args, **kwargs):
            dt = datetime.datetime.now()
            print(dt)
            file.write(str(dt) + '\n')
            f(*args, **kwargs)
        return tar
    return bar

@log('log.txt')
def foo():
    print('foo çağrıldı')

foo()
time.sleep(3)
foo()
time.sleep(2)
foo()
```

Burada open fonksiyonuyla dosya yaratılmış ve write fonksiyonuyla dosyaya çağrıılma tarih ve zamanı yazdırılmıştır. Dosya işlemleri sonraki bölümlerde ele alınmaktadır.

Python'da Exception Mekanizması

Yazılımda programın çalışma zamanı sırasında ortaya çıkan problemler durumlara "exception" denilmektedir. Exception programın çalışma zamanına ilişkin bir kavramdır. Exception'lar çeşitli nedenlerle oluşabilmektedir. Örneğin en çok karşılaşılan exception nedenlerinden biri fonksiyonlara yanlış argüman geçirilmesidir. Örneğin:

```
>>> a = int('123')
>>> a
123
>>> a = int('aaa')
```

```

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a = int('aaa')
ValueError: invalid literal for int() with base 10: 'aaa'

```

int fonksiyonu yazısı int türden değere dönüştürür. Biz int fonksiyonuna sayısal karakterlerden oluşan bir yazı vermediğimiz için yukarıdaki örnekte exception oluşmuştur. Genel olarak oluşan exception'ların bir tür ismi vardır. Burada oluşan exception'ın türü ValueError biçimindedir. Bir fonksiyonu çağrıdığımızda argüman olarak girdiğimiz değerin türü beklenen türden değilse genel olarak bu tür durumlarda da TypeError isimli exception oluşmaktadır. Örneğin:

```

>>> a = [1, 2, 3]
>>> a.insert(2.5, 100)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a.insert(2.5, 100)
TypeError: integer argument expected, got float

```

Burada biz insert fonksiyonuna int türden bir tamsayı değer geçirecekken float bir değer geçirdik. insert metodu da TypeError isimli exception'ı oluşturdu. Örneğin bir listenin ya da demetin olmayan bir elemanına erişmeye çalıştığımızda IndexError isimli bir exception oluşmaktadır:

```

>>> a = [1, 2, 3]
>>> a[100]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a[100]
IndexError: list index out of range

```

Örneğin henüz yaratılmamış bir değişkeni kullandığımızda NameError isimli exception oluşmaktadır:

```

>>> print(z)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    print(z)
NameError: name 'z' is not defined

```

Bir exception yorumlayıcı sistemin kendisi tarafından oluşturulabileceği gibi çalıştırılan kod tarafından da oluşturulabilmektedir. Örneğin olmayan bir değişkenin kullanıldığı durumda yorumlayıcı sistem NameError exception'ını oluşturmaktadır. Halbuki kütüphanedeki bir fonksiyon belli durumlarda kendisi de kodun içerisinde exception oluşturabilmektedir.

Bir exception olduğunda program çöker. Ancak biz oluşan exception'ı ele alarak (handle ederek) programımızın çökmesini engelleyebiliriz. Exception mekanizması için try, except, finally ve raise anahtar sözcükleri (deyimleri) kullanılmaktadır. try bileşik bir deyimdir ve tek başına bulunmaz. try deyimini bir ya da birden fazla except bloğu ya da finally bloğu izlemelidir. Exception ele alma işleminin genel biçimi şöyledir:

```

try:
    <suit>
except <tür> [as <değişken ismi>]:
    <suit>
[except <tür> [as <değişken ismi>]:
    <suit>
....]
[ finally:
    <suit>
]

```

Programın akışı try deyimine girdiğinde artık exception kontrolü başlatılmış olur. Program akış bakımından try bloğunun içerisindeki bir exception oluştuğunda akış bir goto işlemi gibi oluşan exception'la aynı türden olan except deyimine aktarılır. Akış bir daha geri dönmez. Diğer except deyimleri atlanarak akış try-except deyimlerinin sonundan devam eder. Örneğin:

```
try:  
    s = input('Bir değer giriniz: ')  
    x = int(s)  
    print(x)  
except ValueError:  
    print('Girilen değer geçersiz!')  
print('Program sonlanıyor...')
```

Burada klavyeden girilen yazı sayısal karakterler içermiyorsa ValueError isimli exception oluşacaktır. Bu exception except deyimi ile yakalanmıştır. Artık program çökmeyecektir.

Bir fonksiyon içerisinde başka fonksiyonlar çağrılmış olabilir. İç bir fonksiyonda exception oluştuğunda akış bir goto işlemi gibi en son girilen try deyiminin uygun except bölümüne aktarılır. Akış bir daha geri dönmez. Örneğin:

```
def foo():  
    print('foo begins...')  
    try:  
        bar()  
    except ValueError:  
        print('yanlış değer')  
    print('foo ends...')  
  
def bar():  
    print('bar begins...')  
    tar()  
    print('bar ends...')  
  
def tar():  
    print('tar begins...')  
    val = int(input('Bir değer giriniz:'))  
    print('tar ends...')  
  
foo()
```

Burada girilen değer yanlışsa ekranda şu yazılar görülecektir:

```
foo begins...  
bar begins...  
tar begins...  
Bir değer giriniz:asdadas  
yanlış değer  
foo ends...
```

Daha önceden de belirttiğimiz gibi oluşan bir exception ele alınmazsa program çökmektedir. Exception'ın ele alınamamasının iki nedeni olabilir. Birincisi programcı bir try bloğu içerisinde değildir dolayısıyla exception oluşsa bile bunu ele alacak bir durum bulunmamaktadır. İkincisi programcı o noktada bir try bloğunun içerisinde olmalıdır ancak uygun bir except bölümü bulundurmamıştır. Örneğin aşağıda foo fonksiyonunda oluşan Type Error exception'ı uygun bir except bölümü olmadığı için ele alınamamaktadır:

```
def foo():  
    return 10 + 'ali'  
  
try:  
    print('enters the try statement...')  
foo()
```

```

except TypeError:
    print('ValueError occurs...')
print('continues...')


```

Burada exception yakalanamadığı için programımız yine çökecektir. Programın çıktısı şöyle olacaktır:

```

D:\Dropbox\Kurslar\IBB-CBS-Python\Src\PyCharm\Sample\venv\Scripts\python.exe
D:/Dropbox/Kurslar/Python-Subat-2018/Src/Sample/sample.py
Traceback (most recent call last):
  File "D:/Dropbox/Kurslar/Python-Subat-2018/Src/Sample/sample.py", line 5, in <module>
    foo()
  File "D:/Dropbox/Kurslar/Python-Subat-2018/Src/Sample/sample.py", line 2, in foo
    return 10 + 'ali'
TypeError: unsupported operand type(s) for +: 'int' and 'str'


```

Çökmeyi engellemek için bizim TypeError türünden bir except bölümü de bulundurmamız gerekiyor:

```

def foo():
    return 10 + 'ali'
try:
    foo()
except ValueError:
    print('ValueError occurred...')
except TypeError:
    print('TypeError occurred...')
print('continues...')


```

except bölümünün genel biçimi şöyledir:

```

except <tür> [as <değişken ismi>]:
    <deyimler>


```

Bu durumda try-except organizasyonu şöyledir:

```

try:
    <deyimler>
except T1:
    <deyimler>
except T2:
    <deyimler>
except T3:
    <deyimler>
...

```

except anahtar sözcüğünün yanında bir tür ismi getirilir. Örneğin ValueError, TypeError, IndexError gibi. Programın akışı try bloğuna girdiğinde artık exception kontrolündedir. Akış try bloğunun içindeyken ne zaman exception oluşursa o exception'ın türüne göre uygun except bloğuna akış goto işlemi gibi aktarılmaktadır. Akış bir daha geri dönmez. Hata oluşan exception'ın türü neyse o except bloğu tarafından yakalanır, sonra o except bloğundaki deyimler çalıştırılır, diğer except blokları atlanır ve programın akışı except bloklarının sonundan devam eder. Örneğin:

```

def foo():
    print('foo begins...')
    bar()
    print('foo ends...')

def bar():
    print('bar begins...')
    int('xxx')      # ValueError
    print('bar ends...')


```

```

try:
    foo()
except ValueError:
    print("ValueError exception'i oluştu")
except TypeError:
    print("TypeError exception'i oluştu")
except IndexError:
    print("IndexErrorexception'i oluştu")
print('ends...')

```

Eğer akış try bloğuna girdikten sonra hiç exception oluşmazsa akış try bloğundan çıkar, tüm except blokları atlanır ve akış except bloklarının sonundan devam eder. Yani except blokları "exception oluşursa devreye girmektedir".

Bir exception oluştuğunda exception ele alınıp mümkünse problemlü durumun düzeltilmesi için çaba sarf edilebilir. Örneğin:

```

while True:
    try:
        val = int(input('Bir değer giriniz:'))
        break
    except ValueError:
        print('Geçersiz bir değer girdiniz!')
print(val * val)

```

Bir programın oluşan exception'ın ele alınmamasından dolayı çökmesi iyi bir durum değildir. Program içerisindeki exception'lar mümkün olduğunda ele alınmalı, düzeltilebiliyorsa düzeltilmeli, düzeltilemiyorsa program eski yapılan birtakım işlemler geri alınarak düzgün biçimde sonlandırılmalıdır.

except anahtar sözcüğünün yanına birden fazla sınıf ismi yazılabilir. Bu durumda bu sınıf isimlerinin parantezler içine alınması gereklidir. Böylece bu exception'ların hangisi oluşursa oluşsun akış bu except bloğuna aktarılacaktır. Yani except anahtar sözcüğünün yanında parantezler içerisinde birden fazla sınıf ismi varsa bu sınıf isimlerine ilişkin exception'lardan herhangi biri oluştuğunda akış bu except bloğuna aktarılmaktadır. Örneğin:

```

def foo():
    print('foo begins...')
    bar()
    print('foo ends...')

def bar():
    print('bar begins...')
    int('xxx')      # ValueError
    print('bar ends...')

try:
    foo()
except (ValueError, TypeError):
    print("ValueError ya da TypeError exception'i oluştu")
except IndexError:
    print("IndexErrorexception'i oluştu")
print('ends...')

```

Özel bir except bloğu da parametresiz except bloğudur. Bu except bloğu yalnızca except anahtar sözcüğünden oluşur ve tüm exception'ları türü ne olursa olsun yakalar. Örneğin:

```

def foo():
    return 10 + 'ali'
try:
    foo()
except:

```

```
print('An exception occurred...')  
print('Continues...')
```

Tabii biz tüm exception'ları tek bir except bloğu ile yakalarsak hangi exception nedeniyle olayın gerçekleştiğini anlayamayız.

Parametresiz except bloğu parametreli except bloklarıyla birlikte bulundurulabilir. Ancak bu durumda parametresiz except bloğunun tüm parametreli except bloklarının sonunda bulundurulması zorunludur. Böyle bir durumda eğer oluşan exception, parametreli bir except bloğu tarafından yakalanabiliyorsa akış o except bloğuna aktarılır. Eğer oluşan exception hiçbir except bloğu tarafından yakalanamıyorsa parametresiz except bloğu tarafından yakalanır. Örneğin:

```
try:  
    val = int(input('Bir değer giriniz:'))  
    print(val * val)  
except TypeError:  
    print('Girilen değerin türü uygun değil!')  
except:  
    print('Diğer bir nedenden dolayı exception oluştu')
```

Burada ValueError isimli exception parametresiz except bloğu tarafından yakalanabilecektir.

Aslında try deyiminin bir else bölümü de vardır. try deyiminin else bölümü eğer try bloğuna girildikten sonra hiç exception oluşmamışsa çalıştırılır. Örneğin:

```
try:  
    val = int(input('Bir değer giriniz:'))  
    print(val * val)  
except TypeError:  
    print('Girilen değerin türü uygun değil!')  
except:  
    print('Diğer bir nedenden dolayı exception oluştu')  
else:  
    print('else bölümü yalnızca exception olmadığı zaman çalıştırılır...')  
print('Bu deyim her zaman çalıştırılacaktır')
```

Pekiyi else bölümü yerine else bölümünde yapılacak şeyleri try bloğunun sonuna yerleştirsek aynı etki oluşmaz mı? Bu durumda else bölümüne gerek var mıdır? Örneğin:

```
try:  
    val = int(input('Bir değer giriniz:'))  
    print(val * val)  
    print('else bölümü yalnızca exception olmadığı zaman çalıştırılır...')  
except TypeError:  
    print('Girilen değerin türü uygun değil!')  
except:  
    print('Diğer bir nedenden dolayı exception oluştu')  
print('Bu deyim her zaman çalıştırılacaktır')
```

İşte bu durum ile else bölümü arasında şöyle bir farklılık vardır: Exception olmadığı durumda try bloğunun sonuna yerleştirilecek kodların kendisi de exception'a yol açabilir. Bu durumda exception aslında bu nedenden dolayı oluşmuş olabilir. Halbuki else bölümünde oluşan exception başka bir (daha dışındaki) try deyimi tarafından ele alınacaktır. Tabii try deyiminin else bölümüne gerçekten seyrek gereksinim duyulmaktadır.

Bir exception aslında raise isimli basit bir deyim tarafından oluşturulmaktadır. Başka bir deyişle aslında şimdije kadar gördüğümüz exception'ları oluşturan kaynak da raise deyimi ile bunları oluşturmaktadır. raise deyiminin genel biçimini söyleyelim:

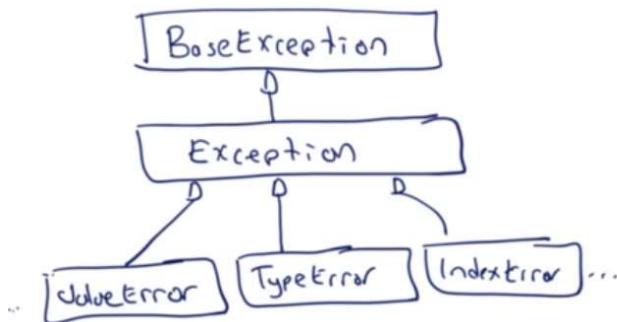
```
raise [exception sınıfının ismi ya da exception nesnesinin referansı]
```

Göründüğü gibi raise anahtar sözcüğünün yanına bir sınıf ismi ya da o sınıf türünden bir değişken ismi getirilebilmektedir. Örneğin:

```
def banner(text):
    if not isinstance(text, str):
        raise TypeError
    print('-' * len(text))
    print(text)
    print('-' * len(text))

banner('ankara')
banner(100)
```

Python'da raise işlemi herhangi bir türle yapılamamaktadır. raise deyiminin yanında BaseException türünün ya da bu sınıftan türetilmiş bir sınıf türünün ya da nesne referansının bulunması gereklidir. Bu zamana kadar gördüğümüz tüm exception'lar aslında BaseException sınıfından türetilmiş Exception isimli sınıftan türetilmiştir.



Örneğin:

```
def foo(a):
    print('foo başladı...')
    if not isinstance(a, int):
        raise TypeError
    if a < 0:
        raise ValueError
    print('foo ok')

try :
    foo('ali')
except ValueError:
    print('Sayı istenildiği gibi değil!')
except TypeError:
    print('Sayı int türden değil!')

print('son...')
```

Burada raise anahtar sözcüğünün yanına bir Exception sınıf ismi getirilmiştir. Aslında biz buraya ilgili exception sınıfı türünden bir exception nesnesi de getirebilirdik. Python'da raise anahtar sözcüğünün yanına bir sınıf ismi getirildiğinde Python yorumlayıcısı aslında bu sınıf türünden parametresiz `__init__` metoduyla bir nesne yaratıp o nesneyle raise işlemi yapmaktadır. Yani aslında aşağıdaki iki deyim tamamen eşdeğer etkiye yol açmaktadır:

```
raise TypeError
raise TypeError()
```

Programın akışı üst üste birden fazla kez try bloğuna girmiş olabilir. Bu durumda bir exception oluştuğunda akış en son girilen try bloğundan itibaren yukarıya doğru try bloklarının except blokları incelenerek ilk uygun except bloğuna aktarılır. Örneğin:

```

def bar(a):
    print('bar begins...')

    try:
        if not isinstance(a, int):
            raise TypeError
        if a < 0:
            raise ValueError
    except ValueError:
        print('İç try deyiminin ValueError except bloğu')

    print('bar ends...')

def foo(a):
    print('foo begins...')
    bar(a)
    print('foo ends...')

try :
    foo(-1)
except TypeError:
    print('Dış try deyiminin TypeError except bloğu')
except ValueError:
    print('Dış try deyiminin ValueError except bloğu')

print('program sonlanıyor...')


```

Burada bar içerisinde ValueError oluşursa bu bar içerisindeki except bloğu tarafından yakalanır. Dolayısıyla artık exception ele alınmış olduğu için akış normal biçimde devam edecektir. Ancak TypeError exception'ı oluşursa iç try bloğunun except blokları bunu yakalayamadığı için dış try bloğunun except bloğu yakalayacaktır. Tabii bir exception yukarıda doğru hiç yakalanamazsa program çökecektir.

Aslında bir exception oluştuğunda oluşan hataya ilişkin birtakım bilgilerin de exception'in yakalandığı noktada elde edilmiş olması arzulanan bir durumdur. Bunun için programcı bir sınıf nesnesi oluşturur ve o sınıf nesnesini kullanarak raise ile exception'ı fırlatır. Exception yakalandığında bu nesnenin içinden bilgiler alınabilir. Bunun için except kısmında tür ifadesinin yanına as anahtar sözcüğü ile değişken isminin belirtilmesi gereklidir. Örneğin:

```

def bar(a):
    print('bar begins...')

    if not isinstance(a, int):
        raise TypeError('Değer int türden değil!')
    if a < 0:
        raise ValueError('Değer negatif!')
    print('bar ends...')

def foo(a):
    print('foo begins...')
    bar(a)
    print('foo ends...')

try:
    foo(10.5)
except TypeError as e:
    print('Exception:', e.args[0])
except ValueError as e:
    print('Exception:', e.args[0])

print('ends..')

```

Bir exception raise ile fırlatılırken aslında raise deyiminde hiç exception nesnesi belirtilmeyebilir. Bu işleme "exception'ın yeniden fırlatılması (reraise)" denilmektedir. Argümansız raise işlemi yalnızca except bloklarının içerisinde yapılabilir. Bu durumda Python yorumlayıcısı yakalan exception'ın aynısının yeniden fırlatıldığını varsayımaktadır. Örneğin:

```
def bar(a):
    print('bar başladı')
    if a < 0:
        raise ValueError('Arguman negatif olamaz')
    print('bar bitti')

def foo(a):
    print('foo başladı')
    try:
        bar(a)
    except Exception as e:
        print(f"Exception foo'da ele alındı ve yeniden fırlatılıyor: {e}")
        raise
    print('foo bitti')

try:
    foo(-10)
except Exception as e:
    print(f'Exception dışarıda ele alındı: {e}')
```

Burada bar içerisinde ValueError exception'ı oluşmuştur. Bu exception foo fonksiyonunda ele alınmış ama yeniden fırlatılmıştır. Yeniden fırlatma (reraise) yakalanan nesnenin aynısıyla yapmaktadır.

Exception mekanizamasında bir de finally bloğu vardır. finally bloğu except bloklarının sonuna yerleştirilmek zorundadır. Exception oluşsa da oluşmasa da programın akışı try bloğundan çıkışken her zaman finally bloğu çalıştırılır. Örneğin:

```
def bar(a):
    print('bar begins...')

    if not isinstance(a, int):
        raise TypeError('Değer int türden değil!')
    if a < 0:
        raise ValueError('Değer negatif!')
    print('bar ends...')

def foo(a):
    print('foo begins...')
    bar(a)
    print('foo ends...')

try :
    foo(-10)
except TypeError as e:
    print('Exception:', e.args[0])
except ValueError as e:
    print('Exception:', e.args[0])
finally:
    print('finally!...')

print('ends...')
```

Peki yukarıdaki örnekte hiç finally bloğu olmasaydı da yine bu blokların sonundaki deyimler exception oluşsa da oluşmasa da çalıştırılmayacak mıydı? Yanıt hayır. Örneğin try bloğu içerisinde bir döngü olabilir. try bloğundan break ya da continue ile çıkışılabilir. Ya da try bloğu bir fonksiyonun içerisinde olabilir. Fonksiyon return ile sonlandırılabilir. İşte tüm bu durumlarda da finally bloğu çalıştırılmaktadır. Örneğin:

```

def foo():
    print('foo begins...')
    try:
        while True:
            a = int(input('Bir değer giriniz:'))
            if a == 0:
                return
            print(a * a)
    except:
        print('Exception oluştu!..')
    finally:
        print('finally!..')

foo()

```

Burada try bloğunun içerisinde fonksiyon return ile sonlandırılmıştır. Yine de finally bloğu çalıştırılır.

Ayrıca iç try bloğunda bir exception oluştuğunda akış dış try bloğuna atlamadan önce iç try bloğundaki finally bloğu da çalıştırılmaktadır. Örneğin:

```

def bar(a):
    print('bar begins...')

    try:
        if not isinstance(a, int):
            raise TypeError
        if a < 0:
            raise ValueError
    except ValueError:
        print('Sayı istenildiği gibi değil')
    finally:
        print('İç try bloğunun finally bölümü')

    print('bar ends...')

def foo(a):
    print('foo begins...')
    bar(a)
    print('foo ends...')

try :
    foo(10.5)
except TypeError:
    print('Sayı int türden değil')

print('ends..')

```

Pekiyi finally bloğunun anlamı nedir? İşte bazı durumlarda try bloğunda bir kaynak tahsis edilmiş olabilir. Akış try bloğundan nasıl çıkarsa çıkışın bu tahsisatın geri alınması gerekebilir. Bunun için en sağlam yol geri alma işleminin finally bloğunda yapılmasıdır. Örneğin:

```

try:
    <kaynak tahsis et>
finally:
    <kaynağı bırak>

```

Python'da programcı kendi exception sınıflarını da yazabilir. Ancak exception sınıflarının BaseException isimli bir sınıfından ya da BaseException sınıfından türetilmiş bir sınıf türünden olması gereklidir. Standart kütüphanede BaseException sınıfından türetilmiş Exception isimli bir sınıf da vardır. Programcı da bu sınıflardan birinden türetme yaparak kendi exception sınıflarını oluşturabilir. Örneğin:

```

class MyError(Exception):
    pass

try:
    a = int(input('Bir değer giriniz:'))
    if a < 0:
        raise MyError()
    print(a * a)
except ValueError:
    print('Girilen değer geçersiz!')
except MyError:
    print('Sayı negatif!')
print('ends...')

```

Burada MyError sınıfı Exception sınıfından türetilmiştir.

Tabii programcının kendi exception sınıflarını yazmak yerine zaten var olan standart kütüphanedeki exception sınıflarını kullanması tercih edilir. Örneğin bir değer hatasında ValueError, bir tür hatasında TypeError sınıfları kullanılabilir.

with Deyimi

Python'da otomatik kaynak boşaltımı için with isimli bir deyim de bulundurulmuştur. with deyiminin işleyişi "bağlam yönetim protokolü (context management protocol)" denilen bir protokole göre yapılmaktadır. with deyiminin genel biçimini söyledir:

```

with <ifade> [as <değişken ismi>]:
    <suite>

```

Örneğin:

```

with open('test.txt') as f:
    ...

```

Python'daki with deyiminin çok benzerleri başka dillerde de vardır. Örneğin C#'taki using deyimi işlevsel olarak Python'daki with deyimi gibidir. with deyiminin anlamlı bir biçimde kullanılabilmesi için with anahtar sözcüğünün yanındaki ifadenin "bağlam yönetim protokolünü" destekleyen bir sınıf türünden olması gereklidir. Örneğin open fonksiyonunun geri dönüş değeri bu protokolü destekleyen bir sınıf türündendir.

Bir sınıfın bağlam yönetim protokolüne uygun olması için o sınıfın __enter__ ve __exit__ isimli iki örnek metodunun bulunuyor olması gereklidir. __enter__ örnek metodu self parametresinin dışında bir parametre almaz. Ancak __exit__ metodu self parametresinin dışında üç parametre daha almaktadır. Geleneksel olarak bu parametrelerin isimleri exc_type, exc_value ve traceback biçimindedir. Örneğin:

```

class Sample:
    def __init__(self):
        print('__init__ called')

    def __enter__(self):
        print('__enter__ called')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print('__exit__ called')
        return False

print('begins...')
with Sample() as s:

```

```
print('suite')
print('ends...')
```

Bu programın çıktısı aşağıdaki gibi olacaktır:

```
begins...
__init__ called
__enter__ called
suite
__exit__ called
ends...
```

with deyiminin ana bloğu (yani "suite") çalıştırılmadan önce bir kez with deyiminde elde edilen sınıf nesnesi ile o sınıfın __enter__ metodu çağrılmaktadır. Burada en önemli nokta with bloğundan nasıl çıkışlırsa çıkışlsın __exit__ metodunun çağrılabileceğidır. with bloğunda bir exception oluşup akış başka bir noktaya atla bile mutlaka __exit__ metodu yine çalıştırılmaktadır.

with deyimini biraz daha ayrıntılı incelemek için önce onun try-except-finally eşdeğerini yazmaya çalışalım. Örneğin:

```
with EXPR as VAR:
    BLOCK
```

Bu with deyiminin Python dokümanlarında belirtilen eşdeğeri şöyledir:

```
mgr = (EXPR)
exit = type(mgr).__exit__ # Not calling it yet
value = type(mgr).__enter__(mgr)
exc = True
try:
    try:
        VAR = value # Only if "as VAR" is present
        BLOCK
    except:
        # The exceptional case is handled here
        exc = False
        if not exit(mgr, *sys.exc_info()):
            raise
        # The exception is swallowed if exit() returns true
finally:
    # The normal and non-Local-goto cases are handled here
    if exc:
        exit(mgr, None, None, None)
```

Bu eşdeğerlilikten hareketle with deyiminin çalışması hakkında şunları söyleyebiliriz:

- 1) with deyimindeki ifade exception kontrolünün dışındadır. Yani bu ifadede bir exception oluşursa herhangi bir sınıfın __enter__ ya da __exit__ metotları çağrılmayacaktır.
- 2) with deyimine girişte bir kez ifadeden elde edilen nesne kullanılarak ilgili sınıfın __enter__ metodu çağrılmaktadır. Bu metottan elde edilen geri dönüş değeri as ile belirtilen değişkene atanır. Programcı en azından __enter__ metodunda self ile geri dönmelidir. Tabii bazen başka bir nesneye de geri dönmek gerekebilir. Ancak bu ileri bir uygulamadır.
- 3) with deyimindeki suit içerisinde (blok içerisinde) bir exception oluşursa with deyiminde belirtilen ifade ile o sınıfın __exit__ metodu çağrılmaktadır. Sonra bu __exit__ metodunun geri dönüş değeri kontrol edilmekte, eğer bu geri dönüş değeri False ise exception yeniden fırlatılmakta, True ise exception yeniden fırlatılmamaktadır. (None değerlerin False olarak ele alındığını dikkat ediniz.) Yani __exit__ metodunun geri dönüş değeri True ise akış exception olmuşmamış gibi except bloklarının ve finally bloğunun sonundan devam edecektir.

4) Eğer suite içerisinde exception oluşmazsa yine with deyimindeki ifade ile ilgili sınıfın `__exit__` metodu çağrılmaktadır. Ancak bu durumda `__exit__` metodunun geri dönüş değeri herhangi bir biçimde kontrol edilmemektedir.

Bu anlatımlar şöyle de özetlenebilir: with deyimine girişte ifadenin ilişkin olduğu sınıfın `__enter__` metodu, çıkışta da `__exit__` metodu çalıştırılır. Ancak exception oluşmuşsa `__exit__` metodunun geri dönüş değeri True ise exception yeniden fırlatılır, False ise exception oluşmamış gibi işlemlere devam edilir. Suit içerisinde hiç exception oluşmasa da yine `__exit__` metodu çağrılmaktadır.

Pekiyi bu `__enter__` ve `__exit__` metodlarının anlamı nedir ve bu metodlar neden çağrılmaktadır? İşte bir exception oluştuğunda tahsis edilmiş kaynakların pratik bir biçimde geri bırakılması için bu `__exit__` metodu düşünülmüştür. Yani bağlam yönetim protokolünü uygulayan sınıfların `__exit__` metodlarında bu sınıfı yazanlar gerekli boşaltım işlemlerini yapmaktadır. Örneğin:

```
with open('x.txt') as f:  
    s = f.read(10)  
    print(s)
```

Burada open bize bir sınıf nesnesi geri vermektedir. O nesnenin adresi de f isimli değişkende saklanmıştır. with bloğundan çıktıığında open fonksiyonunun geri döndürüdüğü sınıf nesnesi ile sınıfın `__exit__` metodu çağrılmacaktır. İşte `__exit__` metodu da dosyayı kapatacaktır. Python'da pek çok sınıf bu biçimde "kaynak yönetim protokolünü" desteklemektedir. Örneğin:

```
class FileWrapper:  
    def __init__(self, path):  
        self.f = open(path, 'w')  
  
    def write(self, s):  
        self.f.write(s)  
  
    def close(self):  
        self.f.close()  
  
    def __enter__(self):  
        return self  
  
    def __exit__(self, exc_type, exc_value, traceback):  
        self.f.close()  
        return False  
  
try:  
    with FileWrapper('test.txt') as f:  
        f.write('this is a test')  
        ....  
except:  
    print('file io error!')  
  
# Bu noktada dosya kapatılmış olacak
```

Pekiyi `__enter__` metodunun anlamı ne olabilir? İşte bazen with bloğuna girmeden önce `__enter__` metodu içerisinde bazı kayıt işlemleri yapılabilir. Çıkışta da bu alınan kayıtlara göre `__exit__` metodundan True ile dönülebilir. Tabii `__enter__` metodunda bir şey yapılmayacaksız yukarıdaki örnekte de olduğu gibi bu metottan self değeri ile geri dönülmelidir.

`__exit__` metodunun üç parametresi olduğuna dikkat ediniz. Birinci parametre oluşan exception'ın türünü belirtmektedir. Bu parametre type sınıfı türündendir. İkinci parametre oluşan exception'a ilişkin raise edilen nesneyi temsil eder. Üçüncü parametre de akışın bu noktaya hangi yolu izleyerek geldiğini belirtmektedir. Bu parametrelere nadiren gereksinim duyulabilmektedir.

Python'da Dosya İşlemleri

İçerisinde bilgilerin bulunduğu ikincil belleklerde tanımlanmış bölgelere "dosya (file)" denilmektedir. Dosyaların isimleri ve özellikleri vardır. Dosya işlemleri en aşağı düzeyde işletim sistemi tarafından ele alınarak yapılmaktadır. Programlama dillerinde dosyalar üzerinde işlem yapan fonksiyonlar ya da sınıflar aslında eninde sonunda işletim sisteminin sistem fonksiyonlarından faydalananmaktadır. Python'da dosya işlemleri file isimli bir sınıf kullanılarak gerçekleştirilmektedir.

Modern işletim sistemlerinin çoğu ikincil bellekleri dizinler (directories) biçiminde organize etmektedir. Dizinler içerisinde dosyalar olabileceği gibi başka dizinler de olabilmektedir. Bir dosyanın yerini belirten yazışal ifadeye "yol ifadesi (path)" denilmektedir. Yol ifadesi dosyanın hangi dizinin içerisindeki hangi dizinde olduğunu gösteren bir yazıdır. Yol ifadelerindeki dizin geçişleri Windows sistemlerinde '\' karakteri ile UNIX/Linux sistemlerinde '/' karakteri ile belirtilmektedir. Hiçbir dizinin içerisinde bulunmayan en dıştaki dizine "kök dizin (root directory)" denilmektedir. Windows sistemlerinde ayrıca bir de "sürücü (drive)" kavramı vardır. Her sürücünün ayrı bir kökü bulunmaktadır. Ancak UNIX/Linux sistemlerinde sürücü kavramı yoktur. Bu nedenle örneğin bir Linux sistemlerinde bir flash belleği bilgisayara taktığımızda onun kök dizini bir dizin altında görülmektedir. Bu işleme UNIX/Linux dünyasında "mount" etme denilmektedir.

Yol ifadeleri "göreli (relative)" ve "mutlak (absolute)" olmak üzere ikiye ayrılmaktadır. Yol ifadelerinin ilk karakterleri '\' ya da '/' ise buna mutlak yol ifadesi denir. Mutlak yol ifadeleri kök dizinden itibaren yer belirtmektedir.

Örneğin:

```
"/home/kaan/study/test.txt"
```

Burada "text.txt" dosyası için UNIX/Linux sistemlerinde mutlak yol ifadesi belirtilmiştir. Eğer yol ifadelerinin ilk karakterleri '\' ya da '/' değilse böyle yol ifadelerine göreli yol ifadeleri denilmektedir. Göreli yol ifadeleri proseslerin (yani çalışmakta olan programların) çalışma dizinlerinden (current working directory) itibaren yer belirtmektedir. Python yorumlayıcılarının çalışma dizinleri os modülündeki chdir fonksiyonuyla değiştirilip getcwd fonksiyonuyla elde edilebilmektedir. Örneğin IDLE'da olalım:

```
>>> import os  
>>> os.getcwd()  
'C:\\\\Users\\\\csystem\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36'  
>>> os.chdir('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src')  
>>> os.getcwd()  
'D:\\\\Dropbox\\\\Kurslar\\\\Python-Subat-2018\\\\Src'
```

Şimdi "a\\b\\c.txt" yol ifadesine bakalım. Bu yol ifadesi görelidir. İşte bu yol ifadesi prosesin çalışma dizininden yanı Python yorumlayıcının çalışma dizininden itibaren yer belirtecektir. Örneğin IDLE'in çalışma dizini "D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src" biçimindeyse "a\\b\\c.txt" yol ifadesi artık "D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src\\a\\b\\c.txt" anlamına gelmektedir. Örneğin yine "test.txt" biçimindeki yol ifadesi görelidir. Buradaki "test.txt" dosyası prosesin çalışma dizininde aranacaktır. PyCharm IDE'sinde bir program başlatıldığında o programın çalışma dizini o program dosyasının içinde bulunduğu projenin dizini olarak ayarlanmaktadır.

Windows'ta mutlak yol ifadelerine sürücü de eklenebilir. Böyle yol ifadelerine "tam yol ifadeleri (full path)" denilmektedir. Örneğin:

```
C:\\Windows\\Notepad.exe"
```

Pekiyi Windows'ta hiç sürücü belirtmeyip yol ifadesine '\' ile başlarsak bu mutlak yol ifadesi hangi sürücüye ilişkin kökü belirtmektedir? İşte Windows bu durumda prosesin çalışma dizini hangi sürücüye ilişkinse kökün de o sürücüye ilişkin olduğunu varsaymaktadır. Yani örneğin IDLE yorumlayıcısının çalışma dizini "D:\Dropbox\Kurslar\Python-Kasım-2010\Src" biçiminde olsun. Biz burada "\test.txt" biçiminde bir yol fadesi veriresek bu aslında "D:\test.txt" anlamına gelmektedir.

Windows sistemlerinde dosya ve dizin isimlerinin büyük harf küçük harf duyarlılığı yoktur. Halbuki UNIX/Linux sistemlerinde vardır. Yani örneğin Windows'ta "test.txt" dosya ismi ile "Test.TXT" dosya ismi aynı dosyayı belirtir. Ancak UNIX/Linux sistemlerinde bunlar farklı dosyaları belirtirler. Windows dosya yaratılırken dosya ismi büyük harf küçük harf bakımından nasıl verilmişse öyle saklamaktadır. Ancak işleme sokarken aralarında fark gözetmemektedir.

Bir dosya ile işlem yapmak için önce dosya açılmalıdır. İşlem bitince de dosya normal olarak kapatılır. Dosya açımı sırasında işletim sistemi ilgili dosya üzerinde bazı ilk işlemleri yapmaktadır. Python'da bir dosyayı açmak için built-in open isimli fonksiyon bulundurulmuştur. open fonksiyonunun parametrik yapısı şöyledir:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

Göründüğü gibi ilk parametrenin dışında tüm parametreler default değer almıştır. Örneğin:

```
f = open('test.txt')
print('Dosya açılamadı')
print('Dosya açıldı')
```

Burada open ile dosya açılamazsa exception fırlatılmaktadır. Bu kod parçası eğer PyCharm IDE'sinde yazılmışsa buradaki "test.txt" dosyası göreli yol ifadesi belirttiği için o andaki proje dizininde aranacaktır.

open fonksiyonun ikinci parametresi dosyanın açış modunu belirten bir stringtir. Dosya açış modları açılan dosya üzerinde hangi işlemlerin yapılabileceğini belirtir. Bu açış modları C Programlama Dilinin standard fopen fonksiyonunun açış modlarıyla aynıdır.

Mod	Anlamı
'r'	Olan dosya açılır ve yalnızca dosyadan okuma yapılabilir. Dosya yoksa exception oluşur.
'r+'	Olan dosya açılır ve dosyadan hem okuma hem de yazma yapılabilir. Dosya yoksa exception oluşur.
'w'	Dosya varsa sıfırlanır ve açılır, dosya yoksa yaratılır ve açılır. Yalnızca yazma yapılabilir.
'w+'	Dosya varsa sıfırlanır ve açılır, dosya yoksa yaratılır ve açılır. Hem okuma hem de yazma yapılabilir.
'a'	Dosya varsa olan dosya açılır, dosya yoksa yaratır ve açılır. Her yazma sona ekleme anlamına gelir. Okuma yapılamaz.
'a+'	Dosya varsa olan dosya açılır, dosya yoksa yaratır ve açılır. Her yazma sona ekleme anlamına gelir. Dosyanın herhangi bir yerinden okuma da yapılabilir.

Built-in open fonksiyonu dosyayı başarılı bir biçimde açarsa dosya işlemlerinde kullanılabilcek bir dosya nesnesine geri dönmektedir. Artık dosya ile ilgili işlemler bu sınıfın çeşitli metodlarıyla yapılır.

Dosyaların Kapatılması

Dosyayı kapatmak için dosya nesnesinin (yani open fonksiyonundan elde edilen sınıf nesnesinin) close metodunu çağrılır. Dosya kapatıldıktan sonra artık dosya üzerinde işlem yapamayız. Ancak yeniden open fonksiyonunu çağırarak dosyayı açabiliriz. Örneğin:

```
f = open('test.txt', 'r+')
print('Dosya açılamadı')
# .....
```

```
f.close()
```

open fonksiyonun geri döndürdüğü dosya nesnesinin ilişkin olduğu sınıf bağlam yönetimi protokolünü (context management protocol) desteklemektedir. Bu durumda biz open fonksiyonunu with deyi̇miyle kullanabiliriz. Dosya nesnesinin ilişkin olduğu sınıfın `__exit__` metodu dosyayı kapatmaktadır. O halde biz open fonksiyonunu with deyi̇mi ile kullandığımızda dosya da otomatik olarak kapatılmış olaacaktır. Örneğin:

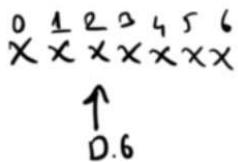
```
with open('test.txt', 'r') as f:  
    print('dosya açıldı')  
print('dosya açılamadı')
```

Peki dosyayı kapatmazsa ne olur? Aslında arka planda işletim sistemlerinin hepsi proses bitti̇nde onların açtığı dosyaları kapatmaktadır. Fakat açık dosyalar işletim sistemi genelinde bir kaynak kullanımına yol açmaktadır. Ayrıca işletim sistemlerinde her prosesin açabilecegi maksimum dosya sayısı çeşitli değerlerle sınırlanmıştır. Bu nedenlerle dosya ile işimiz bitince onu kapatmamız en doğru yol olur. Ayrıca Python'da dosya nesnesi çöp durumuna geldi̇n Python'ın çöp toplayıcısı dosya nesnesine ilişkin sınıfın `__del__` fonksiyonunu çağrılmaktadır. Bu fonksiyon da dosyayı kapatır. Yani dosya nesnesi artık kullanılmadığında (örneğin nesnenin faaliyet alanı bitti̇nde) dosya zaten kapatılmaktadır. Fakat ne olursa olsun en iyi teknik işlemler bitince dosyayı programcının kendisinin `close` metodu ile ya da with deyi̇mi sayesinde kapatmasıdır. Dosya kapatılmışsa dosyanın yeniden kapatılması exception'a yol açmamaktadır.

Python'da dosya işlemlerindeki başarısızlıklarda `IOError` isimli exception fırlatılmaktadır. Bu exception sınıfının `__str__` metodu oluşan hatayı bize yazı olarak vermektedir.

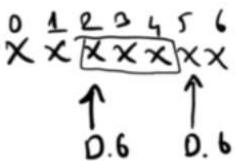
Dosya Göstericisi (File Pointer) Kavramı

Dosya göstericisi dosyadan okuma ve dosyaya yazma işlemlerinin nereden yapılacağını belirten bir kavramdır. Dosyalar byte'lardan oluşmuştur. Her byte'in ilk byte sıfır olmak üzere bir offset numarası vardır. İşte dosya göstericisi bir offset belirtir. Yazma ve okuma işlemleri o offsetten itibaren yapılmaktadır. Örneğin:

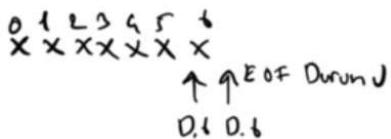


Burada dosya göstericisi 2 numaralı offset'i göstermektedir. Buradan 1 byte okuyacak olsak 2 numaralı offsetteki byte'i okuruz. Bu durumda dosyaya üç byte yazacak olsak bundan 2, 3 ve 4 numaralı offsetteki byte'lar etkilenenecektir. Dosyaya yazma işlemi sırasında insert uygulanmaz. Yani var olan byte'lar ezilerek onların üzerine yazma yapılır.

Dosyadan okuma ve dosyaya yazma işlemlerinde dosya göstericisinin konumu okunan ya da yazılan byte miktarı kadar otomatik ilerletilmektedir. Örneğin yukarıdaki şekilde dosyadan üç byte okuyacak olsak bu üç byte 2, 3 ve 4'üncü offset'teki byte'lar olur. Bu işlemden sonra dosya göstericisi 3 byte ilerletilir.



Dosya göstericisinin dosyanın son byte'ından sonraki byte'i (yani olmayan byte'i) göstermesi durumuna EOF (End of File) durumu denilmektedir. EOF durumundan okuma yapılamaz ancak yazma yapılabilir. Dosya göstericisi EOF durumundayken dosyaya yazma yapılrsa yazılanlar dosyaya eklenmiş olur. Dosyaya ekleme yapmanın tek yolu dosya göstericisini EOF durumuna çekip dosyaya yazmaktr.



open fonksiyonuyla dosya ilk açıldığında dosya göstericisi 0'inci offset'tedir.

Dosyaya Yazma ve Dosyadan Okuma İşlemleri

Dosyaya yazma işlemi open fonksiyonundan elde edilen dosya nesnesinin write metoduyla yapılmaktadır. write metodu argüman olarak bir string ister ve bu string'i dosyaya yazı olarak yazar. Örneğin:

```
f = open('test.txt', 'w')
f.write('This is a test.')
f.write('Yes this is a test')
f.close()
```

Örneğin:

```
with open('test.txt', 'w') as f:
    for i in range(10):
        f.write('Number: {}\\n'.format(i))
```

write metodunun yazma işleminden sonra '\\n' karakterini dosyaya yazmadığına dikkat ediniz. Örneğin:

```
with open('test.txt', 'w') as f:
    while True:
        s = input('Bir yazı giriniz:')
        if s == 'çık':
            break
        f.write(s + '\\n')
```

Dosyadan okuma yapmak için read metodu kullanılmaktadır. read metodu kaç karakter okunacağı bilgisini bizden alır. Dosyadan bu kadar karakteri okuyarak bize bunu bir string biçiminde verir. Örneğin:

```
with open('test.txt', 'r') as f:
    s = f.read(10)
    print(s)
```

read metodu ile biz dosyadan çok karakter okumak istemiş olalı. Ancak dosyada EOF'a kadar az sayıda karakter bulunuyor olsun. Bu durumda ne olur? İşte read metodu okuyabildiği kadar karakteri okur. Bu karakterlerden oluşan string'e geri döner. Tabii artık dosya göstericisi de EOF durumuna gelir. EOF durumundan read metodu ile okuma yapılmak istendiğinde read hiçbir karakter okuyamaz. Boş string ile geri döner. Örneğin test.txt dosyası içerisinde 9 karakter olsun:

```
with open('test.txt', 'r') as f:
    while True:
        s = f.read(5)
        if s == '':
            break
        print(s, end='')
```

read metodunun yanı sıra dosya nesnesinin bir de readline metodu vardır. Bu metot satır sonuna kadar (satır sonundaki '\\n' karakteri de dahil olmak üzere) ya da dosyanın sonuna kadar bir satırı okur ve buna ilişkin bir stringe geri döner. Yine bu metot da EOF nedeniyle okuma yapamadığında boş string'e geri dönmektedir. Örneğin:

```
with open('sample.py', 'r') as f:
    while True:
        s = f.readline()
        if s == '':
            break
```

```
    break
print(s, end='')
```

Tabii aynı işlemi := (walrus) operatörle de yapabiliyoruz:

```
with open('sample.py', 'r') as f:
    while (s := f.readline()) != '':
        print(s, end='')
```

Dosya nesnesine ilişkin sınıf "dolaşılabilir (iterable)" biçimdedir. Dolayısıyla biz bu sınıfı for döngüsüyle dolaşabiliyoruz. Dosya nesneleri bu biçimde dolaşıldığında her yinelemeye sonraki satır elde edilir. Yani dolaşma string biçiminde satır temelinde yapılmaktadır. Örneğin:

```
with open('sample.py', 'r') as f:
    for line in f:
        print(line, end='')
```

Örneğin:

```
with open('sample.py', 'r') as f:
    l = list(f)
    print(l)
```

Sınıf Çalışması: Aşağıdaki içeriğe sahip "test.csv" isimli bir dosya vardır:

```
Ali,123
Veli,456
Selami,678
Fatma,789
```

Dosyanın satırları isim,numara çiftlerinden oluşmaktadır. Bu dosyayı okuyunuz ve içeriğini isimlerin anahtar olduğu numaraların değer olduğu bir sözlüğe yerleştiriniz. Sonra sözlüğü yazdırınız.

Çözüm:

```
d = dict()
with open('test.txt', 'r') as f:
    for line in f:
        if line.strip() == '':
            continue
        l = line.split(',')
        d[l[0]] = l[1].strip()
print(d)
```

Sınıf Çalışması: Aşağıdaki içeriğe sahip "test.csv" isimli bir dosya vardır:

```
ali,123,eskişehir
veli,865,adana
sami,917,muğla
öykü,98,sivas
hasan,902,konya
```

Bu dosyanın satırlarını ',' karakterlerinden ayırarak liste listesi biçimine dönüştürünüz.

Çözüm:

```
with open('test.csv') as f:
    a = []
    for line in f:
```

```
a.append(line[:-1].split(','))

print(a)
```

Aynı işlemi aslında liste içemleriyle de yapabilirdik:

```
with open('test.csv') as f:
    a = [line[:-1].split(',') for line in f]

print(a)
```

Şimdi bir .csv dosyasını okuyan genel bir fonksiyon yazalım:

```
def read_csv(path, converters=None, header=False):
    with open(path) as f:
        if header:
            h = f.readline()[:-1].split(',')
        l = []
        for line in f:
            a = line[:-1].split(',')
            if a[0] == '':
                continue
            if converters:
                for key, value in converters.items():
                    a[key] = value(a[key])
            l.append(a)

    if header:
        return l, h

    return l
```

```
l, h = read_csv('people.csv', None, True)

print(l)
print(h)
```

read_csv fonksiyonunun birinci parametresi .csv dosyasının yol ifadesini almaktadır. İkinci parametre dönüştürme yapılacak sütunları ve türleri belirten bir sözlük nesnesini almaktadır. Üçüncü parametre de dosyanın bir başlık kısmına sahip olup olmadığını belirtmektedir. Örnek bir kullanım şöyle olabilir:

```
l, h = read_csv('people.csv', {1: int, 3: float}, True)
```

people.csv dosyası da aşağıdaki içeriğe sahiptir:

```
isim,no,doğum yeri,oran
ali,123,eskişehir,4.2
mehmet,256,eskişehir,5.8
sacit,452,urfa,7.2
```

Dosya Göstericisinin Konumlandırılması

Dosyanın herhangi bir yerinden okuma ya da yazma yapmak için öncelikle dosya göstericisi uygun offsete konumlandırılmalıdır. Bunun için seek metodu kullanılır. Metodun birinci parametresi konumlandırma offset'ini ikinci parametresi konumlandırma orijinini belirtir. İkinci parametre default değer almıştır. Eğer girilmezse konumlandırma baştan itibaren yapılır. İkinci parametre için üç senek söz konusudur:

Konumlandırma	Anlamı
---------------	--------

0	Konumlandırma dosyanın başından itibaren yapılır. Bu durumda konumlandırma offset'i 0 ya da pozitif olmak zorundadır.
1	Konumlandırma o andaki dosya göstericisinin konumuna göre yapılır. Bu durumda birinci parametre negatif, pozitif ya da sıfır olabilir. Negatif geri, pozitif ileri anlamına gelmektedir.
2	Konumlandırma EOF'tan itibaren yapılır. Bu durumda birinci parametre negatif ya da sıfır girilir.

Bu durumda dosyaya ekleme yapmak için tipik olarak f.seek(0, 2) çağrıması yapılır. Örneğin test.txt dosyasının içeriği şöyle olsun:

```
0123456789
```

```
with open('test.txt', 'r+') as f:
    f.seek(0, 2)
    s = f.write('xxx')
```

Bu işlemden sonra dosya şu biçimde gelecektir:

```
0123456789xxx
```

seek metodu başarısız konumlandırmalarda exception oluşturmaktadır. Eğer konumlandırma başarılıysa metot dosyanın başından itibaren konumlandırılmış olan offset değerine geri dönmemektedir. Dosya göstericisinin baştan itibaren konumu tell metoduyla da elde edilebilir. Örneğin biz dosya göstericisini önce EOF'a konumlandırdıktan sonra tell değerini elde edersek dosyanın uzunluğunu elde etmiş oluruz.

Text ve Binary Dosyalar

Dosyaların uzantıları aslında insanlar için düşünülmüştür. Dosyaların içerisinde ne olursa olsun dosyalar birer byte yiğini olarak düşünülebilirler. İçerisinde yalnızca yazılar bulunan ve bu niyetle oluşturulmuş olan dosyalara halk arasında "text dosyalar" denilmektedir. İçerisinde formatlı yazı bulunan ancak bu formatlama bilgisinin de yazışal olarak kodlandığı dosyalara "rich text dosyalar" denilmektedir. İçerisinde bağlantı bilgisinin (hyper link) dosyalara ise "hyper text" dosyalar denir. Örneğin HTML dosyaları "hyper text" dosyalardır. Bunların dışında içerisinde saf yazı bulunmayan dosyalara ise "binary dosyalar" denilmektedir. Örneğin ".exe" uzantılı çalıştırılabilir dosyalar, ".jpg" uzantılı resim dosyaları binary dosyalardır.

Dosyalar da içerikleri ne olursa olsun "text modda" ya da "binary modda" açılabilirler. Tabii text dosyaların text modda açılması binary dosyaların binary modda açılması normal olan durumdur. open fonksiyonunda default açış modu text moddur. Binary modda açım için açış moduna "b" harfi eklenir. Örneğin:

```
with open('test.txt', 'r+b') as f:
    pass
```

Dosya binary modda açıldığında read ve write metodları artık str türüyle değil bytes türüyle çalışmaktadır. Yani biz read ile okuma yaptığımızda read bize bir string değil bytes nesnesi verir. write ile yazma yaparken de benzer biçimde write parametre olarak bytes nesnesi ister. Örneğin:

```
with open('test.txt', 'r+b') as f:
    b = f.read(10)
    print(b)
```

Şimdi de binary açılmış dosyaya yazma işlemi yapalım:

```
with open('binary.dat', 'wb') as f:  
    f.write(b'\x10\x12\x23')
```

Önceki konularda da gördüğümüz gibi biz yazıları bytes nesnesine dönüştürebiliriz ve buradan elde ettiğimiz değerleri de dosyaya yazabiliriz. Örneğin:

```
s = 'ağrı dağı çok yüksek'  
with open('test.dat', 'wb') as f:  
    b = bytes(s, encoding='utf-8')  
    f.write(b)
```

Şimdi bir dosyadan belli büyülükte byte'ları okuyup diğer bir dosyaya yazarak dosya kopyalaması yapan bir fonksiyon yazalım. Burada dosyanın text mi binary mi olduğunu bilemeyebiliriz. Bu tür durumlarda dosyaların binary modda açılması gereklidir. Örneğin:

```
def copy_file(source_path, dest_path):  
    with open(source_path, 'rb') as fs:  
        with open(dest_path, 'wb') as fd:  
            while True:  
                b = fs.read(4096)  
                if not b:  
                    break  
                fd.write(b)
```



```
copy_file('sample.py', 'test.txt')
```

Text Dosyalarda Karakter Kodlaması (Character Encoding)

Anımsanacağı gibi içerisinde yalnızca yazışal bilgilerin olduğu dosyalara text dosyalar denilmektedir. Bilindiği gibi bilgisayar belleğinde ya da dosyaların içerisinde aslında her şey sayısal düzeyde ikilik sistemde tutulmaktadır. Yani aslında yazı diye bir şey yoktur. Bir yazı (ya da string) bir sayı dizisidir. Örneğin bir dosyanın içerisindeki "ankara" yazısı aslında birtakım sayıların yan yana getirilmesiyle kodlanmaktadır. İşte hangi sayıların hangi karakterleri temsil ettiğine yönelik çeşitli tablolar oluşturulmuştur. Bu tabloların ilki ASCII tablosudur. Karakter tabloları bir karakterin kaç byte yer kapladığını göre üç kısma ayrılmaktadır:

- 1) Tek byte'luk karakter tabloları
- 2) İki byte'luk karakter tabloları
- 3) Değişken uzunluğa sahip karakter tabloları

Tek byte'luk karakter tablolarında her karakter tek byte ile temsil edilmektedir. Örneğin ASCII tablosu, EBCDIC tablosu böyle tek byte'luk karakter tablolarıdır. Tek byte'luk karakter tabloları ile en fazla 256 farklı karakter temsil edilebilir. Bu da bazı doğal diller için sorun oluşturmaktadır. İki byte'luk karakter tablolarının en yaygın olanı UNICODE tablodur. UNICODE tabloda tüm ulusların karakterleri ve değişik pek çok karakterler aynı tabloya yerleştirilmiştir. Bazı karakter tablolarında ise her karakter eşit yer kaplamaz. Bazı karakterler 1 byte ile bazıları 2 byte ile bazıları daha çok byte ile kodlanmışlardır. Örneğin UTF-8 denilen kodlama aslında UNICODE tablonun sıkıştırılmış bir biçimidir. UTF-8 bu nedenden dolayı çok kullanılmaktadır.

Aslında ASCII tablosunun orijinali bir byte değil 7 bitti. Dolayısıyla orijinal ASCII tablosu 128 karakter içeriyordu. Ancak sonraları tablo 8 bite tamamlanmıştır. Yani tabloya 128 karakter daha eklenmiştir. Ancak bu eklemeler farklı uluslar tarafından farklı biçimde yapılmıştır. İşte ASCII tablosunun bu farklı varyasyonlarına "code page" denilmektedir. ASCII Latin-1 code page'i tipik Avrupa ülkelerinin kullandığı code page'tir. Bu code page'te Türkçe karakterler yoktur. Türkçe karakterlerin bulunduğu farklı code page'ler tanımlanmıştır. Örneğin Windows 1254, ISO 8859-9, OEM Türkçe code page'leri Türkçe karakterleri barındırmaktadır.

Built-in open fonksiyonunda text modda bir dosya açıldığında default encoding sistemden sisteme değişebilmektedir. Default encoding'in ne olduğu aşağıdaki çağrı ile elde edilebilir:

```
import locale  
  
print(locale.getpreferredencoding(False))
```

Örneğin kursun yapıldığı Windows makinede default encoding cp1254 (yani Microsoft'un 1 byte'lık Türkçe code page'i) biçimindedir.

Diğer encoding'lere ilişkin okuma ya da yazma yapılacaksa open fonksiyonunun encoding parametresi uygun biçimde girilmelidir. Örneğin:

```
with open('test.txt', 'w', encoding='utf-8') as f:  
    f.write('ağrı dağı')
```

Tabii biz bir dosyayı gerçekte onun kodlandığı karakter tablosundan farklı bir kodlama ile açıp okumaya çalışırsak okuduğumuz yazı anlamsızlaşır. Yukarıdaki örnekte "ağrı dağı" yazısı UTF-8 ile kodlanmıştır. Biz onu ASCII biçiminde okursak yanlış okumuş oluruz. O halde bizim bir text dosyayı okumadan önce onun karakter formatını bilmemiz gerekmektedir.

Python'daki str sınıfı zaten UNICODE temelli çalıştığı için her yazıyı ifade edebilmektedir. Ancak bu yazıyı write fonksiyonu ile dosyaya yazarken open fonksiyonundaki encoding parametresine göre yazım yapılmaktadır. Tabii eğer write fonksiyonuna verdığımız yazındaki herhangi bir karakter hedeflediğimiz encoding'e uygun değilse write bir exception oluşturmaktadır.

Daha önce byte ve str sınıfları yoluyla encoding dönüştürmesi yapmıştık. Aslında biz benzer işlemleri codecs isimli standart modülle de yapabiliriz. codecs modülünün encode fonksiyonu ile bir yazıyı istediğimiz bir karakter tablosuna göre kodlayabiliriz, decode fonksiyonu ile de bunu geri çözebiliriz. Örneğin:

```
>>> import codecs  
>>> b = codecs.encode('ağrı dağı', encoding='utf-8')  
>>> type(b)  
<class 'bytes'>  
>>> b  
b'a\xc4\x9fr\xc4\xb1 da\xc4\x9f\xc4\xb1'
```

encode fonksiyonu bizden kodlanacak yazıyı str sınıf olarak ister ve geri dönüş değeri olarak da bize bytes cinsinden kodlanmış olan yazıyı verir. Şimdi de biz yazıyı yeniden decode fonksiyonu ile str türüne dönüştürelim:

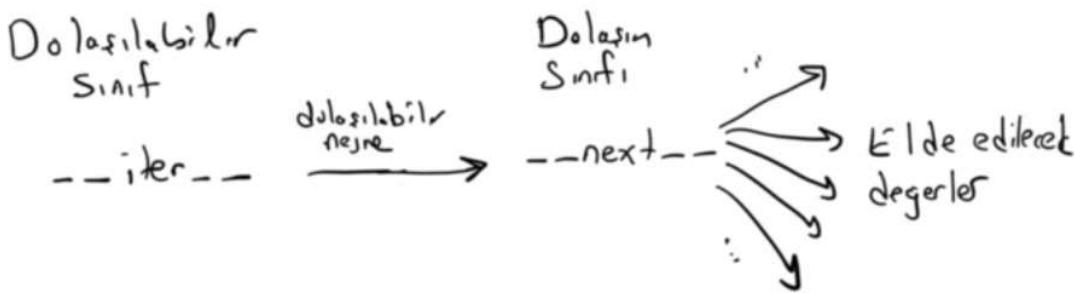
```
>>> s = codecs.decode(b, encoding='utf-8')  
>>> s  
'ağrı dağı'  
>>> type(s)  
<class 'str'>
```

Dolaşılabilir (Iterable) Sınıfların Oluşturulması

Anımsanacağı gibi Python'da bir sınıf nesnesinin for döngüsüyle dolaşılabilmesi ya da birtakım fonksiyonlara parametre yoluyla geçirilebilmesi için onun dolaşılabilir (iterable) olması gerekiyordu. Örneğin list, set, dict, tuple, str, range sınıfları dolaşılabilir sınıflardır. İşte biz de istersek kendi sınıfımızı dolaşılabilir yapabiliriz. Tabii dolaşılabilir sınıfların bir biçimde bir "collection" sınıf gibi davranışması beklenir. Yani bu sınıflar dolaşıldıkça bize birtakım değerler vermelidir.

Bir sınıf nesnesinin dolaşılabilir olması için o sınıfta `__iter__` isimli bir metodun bulunması gereklidir. `__iter__` metodu self parametresinden başka bir parametre almaz. Bu metot bize bir "dolaşım (iterator)" nesnesi verir. Bu dolaşım nesnesinin ilişkin olduğu sınıfındaki `__next__` metodu her çağrılsa bize yeni bir değer verecek biçimde yazılmıştır. `__next__` metodu da parametresizdir. Ancak geri dönüş değeri bizim dolaşım sırasında her eleman istendiğinde elde edeceğimiz değerdir. `__next__` metodunda eğer artık bir değer geri döndürilmeyeceksse (yani dolaşımın sonuna

gelinmişse) bu durumda `__next__` metodunu StopIteration isimli exception'ı fırlatmalıdır. Nesneyi dolaşan taraf da bu exception oluşana kadar `__next__` metodunu çağırır.



Göründüğü gibi bir dolaşım yapabilmek için aslında iki nesneye gereksinimiz vardır: Dolaşılabilir bir nene ve dolaşım nesnesine. Aşağıdaki örneği imceleyiniz:

```
class SampleIterable:  
    def __init__(self, *args):  
        self.args = args  
    def __iter__(self):  
        return SampleIterator(self.args)  
  
class SampleIterator:  
    def __init__(self, args):  
        self.args = args  
        self.index = 0  
  
    def __next__(self):  
        self.index += 1  
        if self.index > len(self.args):  
            raise StopIteration  
        return self.args[self.index - 1]  
  
s = SampleIterable(10, 20, 30)  
  
for x in s:  
    print(x, end=' ')  
print()
```

Bırada SampleIterable sınıfı dolaşılabilir bir sınıfır. Bu sınıfın `__iter__` metodu SampleIterator sınıfı türünden bir dolaşım nesnesi vermektedir. Asıl dolaşım SampleIterator metodunun `__next__` metodlarının çağrılmalarıyla yapılmaktadır. Yukarıdaki kodun çalıştırılması sonucunda şu çıktı elde edilecektir:

10 20 30

Aslında dolaşılabilir sınıfta dolaşım sınıfı aynı sınıf da olabilir. Bu durumda söz konusu sınıf hem `__iter__` metodunu hem de `__next__` metodunu içerecektir. Tabii bu duurmda `__iter__` metodunun nesnenin kendisiyle geri dönmesi gereklidir.

```
class MySimpleRange:  
    def __init__(self, stop):  
        self.stop = stop  
        self.count = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        self.count += 1
```

```

if self.count > self.stop:
    raise StopIteration
return self.count - 1

r = MySimpleRange(10)
a = list(r)
print(a)

for i in MySimpleRange(10):
    print(i, end=' ')
print()

```

Kodun çalıştırılması sonucunda aşağıdaki çıktı elde edilecektir:

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

Burada dolaşılabilir (iterable) sınıfıyla dolaşım (iterator) sınıfı aynı sınıfıdır. Dolayısıyla `__iter__` metodu da `self` değerine yani aynı nesneye geri dönmüştür. Ancak sonraki paragraflarda da ifade edeceğimiz üzere böyle bir tasarımda küçük bir kusur da vardır. Dolaşılabilir sınıfla dolaşım sınıfının ayrı ayrı yazılması daha uygundur.

Python referans kitabına göre dolaşım sınıfı yalnızca `__next__` metoduna değil aynı zamanda `__iter__` metoduna da sahip olmalıdır. Yani dolaşım nesnesi aynı zamanda dolaşılabilir bir nesne gibi davranmalıdır. Böylece biz dolaşım nesnesinin kendisini de istersek dolaşılabilir bir nesne gibi kullanabiliriz. Bu durumda "dolaşılabilir (iterable)" bir sınıfın yalnızca `__iter__` metodu olması yeterlidir. Ancak "dolaşım (iterator)" sınıfının hem `__iter__` hem de `__next__` metodunun bulunması gereklidir. Tabii dolaşılabilir sınıf ile dolaşım sınıfı aynı ise (bunun kusurlu bir taarım olduğunu belirtmiştim) zaten bu koşullar sağlanmış olmaktadır. Başka bir deyişle her dolaşım sınıfı aynı zamanda bir dolaşılabilir sınıfı fakat her dolaşılabilir sınıf bir dolaşım sınıfı olmak zorunda değildir.

Her ne kadar dolaşım nesneleri dolaşılabilir nesneler gibi davranabiliyorsa da dolaşılabilir nesnelerle dolaşım nesneleri arasında şöyle bir farklılık da vardır: Dolaşılabilir nesneler dolaşılırken dolaşım baştan başlamalıdır ancak dolaşım nesneleri dolaşılırken dolaşım hep kalınan yerden devam etmelidir. Bunun ne anlamına geldiğini şöyle bir örnekle açıklayabiliriz:

```

r = range(10)

for x in r:
    print(x, end=' ')

print()

for x in r:
    print(x, end=' ')

```

Burada `r` dolaşılabilir bir nesnedir. Dolayısıyla biz onu birden fazla kez dolaştığımızda dolaşım baştan başlar. Yukarıdaki kodun çalıştırılmasıyla aşağıdaki çıktı elde edilecektir:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Halbuki dolaşım nesnesi birden fazla kez dolaşıldığında dolaşım kalınan yerden devam eder. Örneğin:

```

r = range(10)

iterator = r.__iter__()

for x in iterator:
    print(x, end=' ')

```

```
print()  
  
for x in iterator:  
    print(x, end=' ')
```

Burada ikinci for döngüsü dolaşım nesnesi dolaşımı bitirdiği için herhangi bir yineleme yapmayacaktır. Bu durumda kodun çalıştırılmasıyla aşağıdaki gibi bir çıktı elde edilecektir:

```
0 1 2 3 4 5 6 7 8 9
```

Dolaşılabilir nesnelerle dolaşım nesneleri arasındaki bu farklardan dolayı dolaşılabilir sınıflarla dolaşım sınıflarının ayrı ayrı yazılması gereklidir. Örneğin yukarıda verdığımız MySimpleRange dolaşılabilir sınıfının doğru yazımı aşağıdakine benzer olmalıdır:

```
class MySimpleRange:  
    def __init__(self, stop):  
        self.stop = stop  
  
    def __iter__(self):  
        return MySimpleRangeIterator(self.stop)  
  
class MySimpleRangeIterator:  
    def __init__(self, stop):  
        self.stop = stop  
        self.count = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        self.count += 1  
        if self.count > self.stop:  
            raise StopIteration  
        return self.count - 1
```

Aşağıdaki örneğe dikkat ediniz:

```
r = MySimpleRange(10)  
  
for i in r:  
    print(i, end=' ')  
print()  
for i in r:  
    print(i, end=' ')
```

Buradan şu şöyle bir çıktı elde edilecektir:

```
0 1 2 3 4 5 6 7 8 9  
0 1 2 3 4 5 6 7 8 9
```

Şimdi aşağıdaki kodu çalıştıralım:

```
r = MySimpleRange(10)  
iterator = r.__iter__()  
for i in iterator:  
    print(i, end=' ')  
print()  
for i in iterator:  
    print(i, end=' ')
```

Buradan şu çıktı elde edilecektir:

```
0 1 2 3 4 5 6 7 8 9
```

İkinci for döngüsünün hiç dönmediğine dikkat ediniz.

Global düzeyde built-in iter isimli bir fonksiyon da vardır. Bu fonksiyon aslında parametre yoluyla aldığı nesne ile `__iter__` metodunu çağrılmaktadır (yani str fonksiyonunda olduğu gibi). Başka bir deyişle:

```
x.__iter__()
```

çalışması ile,

```
iter(x)
```

çalışması eşdeğerdir. Çünkü iter fonksiyonu şöyle yazılmıştır:

```
def iter(iterable):
    return iterable.__iter__()
```

Benzer biçimde global built-in bir next metodu da vardır. Bu metot da aldığı nesneyle `__next__` metodunu çağrılmaktadır. Yani global next metodu şöyle yazılmıştır:

```
def next(iterator):
    return iterator.__next__()
```

Bu durumda:

```
x.__next__()
```

işlemi ile,

```
next(x)
```

işlemi eşdeğerdir.

Pekiyi `__next__` metodu nasıl yazılmalıdır? İşte next her defasında dizimin bir sonraki elemanıyla geri dönecek biçimde yazılmalıdır. Dizilik bitince de next metodu StopIteration denilen bir sınıfından nesneyle raise edilmelidir. Gerçekten de önceki örneğimizdeki `__next__` şöyle yazılmıştır:

```
def __next__(self):
    if self.count == self.stop:
        raise StopIteration
    self.count += 1
    return self.count - 1
```

Burada başlangıçta `self.count` değeri 0'dır. Dolayısıyla `__next__` ilk kez çağrıldığında dolaşım 0 değerini verecektir. Sonraki çağrımlarda `self.count` hep 1 artırıldığı için sıradaki elemanlar verilir. En sonunda da stop değerine gelindiğinde artık StopIteration isimli exception fırlatılmıştır.

Pekiyi Python'da for döngüsünün eşdeğerini nasıl yazabilirim? Yani:

```
for i in s:
    <suit>
```

işlemının eşdeğeri nasıldır? Aslında Python yorumlayıcısı bu for döngüsünü aşağıdaki gibi ele almaktadır:

```

iterator = s.__iter__()           # iter(s)

while True:
    try:
        i = iterator.__next__()  # i = next(iteration)
    except StopIteration:
        break
    <suit>

```

Sınıf Çalışması: SqrtIterable isimli bir dolaşılabilir sınıf ve SqrtIterator isimli bir dolaşım sınıfını belirtildiği gibi yazınız. SqrtIterable sınıfı __init__ metodunda int bir limit parametresi alacaktır. SqrtIterator sınıfı da her defasında 1'den başlayarak bu limit değerine kadar (bu limit değeri dahil) sayıların bize kareköklerini float biçimde verecektir. Örneğin:

```

s = SqrtIterable(25)
for f in s:
    print(f)

```

Çözüm:

```

import math

class SqrtIterable:
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        return SqrtIterator(self.limit)

class SqrtIterator:
    def __init__(self, limit):
        self.limit = limit
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i == self.limit:
            raise StopIteration()
        self.i += 1

        return math.sqrt(self.i)

s = SqrtIterable(25)

for f in s:
    print(f)

```

Anımsanacağı gibi list, set, dict gibi collection sınıfların __init__ metotları bizden dolaşılabilir bir nesne de alabiliyordu. Bu durumda biz kendi dolaşılabilir nesnemizi onalara verip ilgili türden nesneler elde edebiliriz. Örneğin:

```

l = list(SqrtIterable(100))

for f in l:
    print(f)
print()

```

Sınıf Çalışması: random modülünü ve onun içerisindeki randint metodunu kullanarak RandIterable isimli bir dolaşılabilir sınıf yazınız. Bu sınıf __init__ metodunda a, b ve c isminde sırasıyla üç değer alacaktır. Sınıf her yinelemede [a, b] aralığında rastgele bir tamsayı verecektir. Bu işlem de c kere yapılacaktır. Yani örneğin:

```
for i in RandomIterable(1, 100, 10):
    print(i)
```

Bu işlemle [1, 100] aralığında rastgele 10 tane sayı ekrana yazdırılmaktadır.

Çözüm:

```
import random

class RandIterable:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __iter__(self):
        return RandIterator(self)

class RandIterator:
    def __init__(self, ri):
        self.ri = ri
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count == self.ri.c:
            raise StopIteration
        self.count += 1

        return random.randint(self.ri.a, self.ri.b)

a = list(RandIterable(10, 20, 5))
print(a)

for x in RandIterable(0, 100, 10):
    print(x, end=' ')
print()

try:
    ri = RandIterable(0, 100, 10)          # iterator = ri.__iter__()
    iterator = iter(ri)
    while True:
        x = next(iterator)             # iterator.__next__()
        print(x, end=' ')
    print()
except:
    pass
```

Bugüne kadar incelediğimiz çeşitli fonksiyonlar aslında bize bir liste değil dolaşılabilir bir nesne vermektedir. Örneğin built-in range fonksiyonu aslında bize range isimli bir sınıf türünden dolaşılabilir bir nesne verir. Zaten range aslında bir fonksiyon değil bir sınıfıtır. Bu sınıfın verdiği dolaşım nesnesinin __next__ metodu ile biz dolaşımı sağlarız. Örneğin:

```

r = range(10)
for i in r:
    print(i, end=' ')
print()

r = range(10)
iter = r.__iter__()
try:
    while True:
        i = iter.__next__()
        print(i, end=' ')
    print()
except StopIteration:
    pass
print()

```

Pekiyi range bize neden bir liste değil de dolaşılabilir bir nesne vermektedir? Çünkü bu durumda büyük bir aralık belirtildiğinde büyük bir listenin yaratılması gerekirdi. Halbuki range bunu hiç liste yaratmadan bize çok daha etkin biçimde verebilmektedir. Pekiyi şimdi range sınıfını myrange ismiyle biz yazalım:

```

class myrange:
    def __init__(self, start, stop = None, step = 1):
        if stop == None:
            self._start = 0
            self._stop = start
        else:
            self._start = start
            self._stop = stop
        self._step = step

    def __iter__(self):
        return myrange_iterator(self)

class myrange_iterator:
    def __init__(self, mr):
        self._mr = mr
        self._i = mr._start

    def __iter__(self):
        return self

    def __next__(self):
        if self._i >= self._mr._stop:
            raise StopIteration
        self._i += self._mr._step

        return self._i - self._mr._step

    for i in myrange(10):
        print(i, end=' ')
    print()

    for i in myrange(1, 10, 2):
        print(i, end=' ')
    print()

```

Tabii dolaşım sınıfının `__next__` metodunu bize demet, liste ya da başka türden bir nesne de verebilir. Örneğin:

```

import math

class SqrtIterable:
    def __init__(self, limit):

```

```

        self._limit = limit

    def __iter__(self):
        return SqrtIterator(self._limit)

class SqrtIterator:
    def __init__(self, limit):
        self._limit = limit
        self._i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._i == self._limit:
            raise StopIteration()
        self._i += 1

        return self._i, math.sqrt(self._i)

si = SqrtIterable(25)

for t in si:
    print(t, end=' ')
print()

for i, x in si:
    print(f'{i} --> {x}')

```

Burada artık dolaşım sınıfının `__next__` metodu tekil bir değer değil bir demet vermiştir. Demetin ilk elemanı karekökü alınacak int türden sayıyı, ikinci elemanı karekök değeri olan float sayıyı vermektedir. dict sınıfının `__init__` metodunun iki elemanlı dolaşılabilir nesneler veren dolaşılabilir bir nesne alabildiğini anımsayınız. Örneğin:

```
d = dict(SqrtIterable(10))
print(d)
```

Built-in `enumerate` fonksiyonu da aslında bir fonksiyon değil dolaşılabilir bir sınıf biçiminde gerçekleştirilmişdir. `enumerate` sınıfının gerçekleştirimi aşağıdaki gibi yapılabilir:

```

class myenumerate:
    def __init__(self, iterable):
        self._iterator = iter(iterable)

    def __iter__(self):
        return myenumerate_iterator(self._iterator)

class myenumerate_iterator:
    def __init__(self, iterator):
        self._iterator = iterator
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        self._index += 1
        return self._index - 1, self._iterator.__next__()

a = [10, 20, 30, 40, 50]

```

```
for index, val in myenumerate(a):
    print(index, '=>', val)
```

Anımsanacağı gibi Python'da tersten dolaşan ayrı bir dolaşım nesnesi yoktur. Yani aslında dolaşımın ters ya da düz olması biçiminde bir kavram da yoktur. Ancak önceki konularda gördüğümüz gibi global bir reversed fonksiyonu vardır. Biz bu fonksiyon yoluyla bazı dolaşılabilir nesneleri tersten dolaşılabilir hale getirebilmekteyiz. Örneğin:

```
for i in reversed(range(10)):
    print(i, end = ' ')
print()
```

Tabii her dolaşılabilir sınıf nesnesi reversed fonksiyonuna sokulamaz. Örneğin bizim yukarıda oluşturduğumuz örnekteki sınıfları biz reversed fonksiyonuna sokamayız. reversed fonksiyonu aslında ilgili sınıfın `__reversed__` isimli metodunu çağırmaktadır. Bu metod da tersten dolaşan bir dolaşım nesnesi, vermektedir. Yani başka bir deyişle biz kendi sınıfımızın reversed fonksiyonuyla kullanılmasını istiyorsak sınıfımızda tersten dolaşım yapacak `__reversed__` metodunu yazmalıyız. Aslında reversed fonksiyonu `__reversed__` metodunu çağırarak onun geri dönüş değerine geri dönmektedir. Yani reversed fonksiyonu aslında şöyle yazılmıştır:

```
def reversed(iterable):
    return iterable.__reversed__()
```

Şimdi reversed fonksiyonuna sokabileceğimiz dolaşılabilir bir sınıf yazalım:

```
class SampleIterable:
    def __init__(self, *args):
        self._args = args

    def __iter__(self):
        return SampleIterator(self._args)

    def __reversed__(self):
        return SampleReverseIterator(self._args)
```

```
class SampleIterator:
    def __init__(self, args):
        self._args = args
        self._i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._i == len(self._args):
            raise StopIteration
        self._i += 1

        return self._args[self._i - 1]
```

```
class SampleReverseIterator:
    def __init__(self, args):
        self._args = args
        self._i = len(self._args) - 1

    def __iter__(self):
        return self

    def __next__(self):
        if self._i < 0:
            raise StopIteration
        self._i -= 1
```

```

        return self._args[self._i + 1]

for i in SampleIterable(1, 2, 3, 4, 5):
    print(i, end=' ')
print()

for i in reversed(SampleIterable(1, 2, 3, 4, 5)):
    print(i, end=' ')
print()

```

Şimdi de myrange sınıfını tersten dolaşılabilir (reversible) yapalım:

```

class myrange:
    def __init__(self, start, stop = None, step = 1):
        if stop == None:
            self._start = 0
            self._stop = start
        else:
            self._start = start
            self._stop = stop
        self._step = step

    def __iter__(self):
        return myrange_iterator(self)

    def __reversed__(self):
        return myrange_reverse_iterator(self)

class myrange_iterator:
    def __init__(self, mr):
        self._mr = mr
        self._i = mr._start

    def __iter__(self):
        return self

    def __next__(self):
        if self._i >= self._mr._stop:
            raise StopIteration
        self._i += self._mr._step

        return self._i - self._mr._step

class myrange_reverse_iterator:
    def __init__(self, mr):
        self._mr = mr
        self._i = mr._stop

    def __iter__(self):
        return self

    def __next__(self):
        if self._i <= self._mr._start:
            raise StopIteration
        self._i -= self._mr._step
        return self._i

for i in myrange(10):
    print(i, end=' ')
print()

```

```
for i in reversed(range(10)):  
    print(i, end=' ')  
print()
```

Python'ın built-in list sınıfı, tuple sınıfı ve str sınıfı "tersten dolaşılabilir (reversible)" sınıflarıdır. Halbuki set ve dict sınıfları tersten dolaşılabilir sınıflar değildir.

Dolaşılabilir Nesnelerin Anlamı

Dolaşılabilir nesnelere neden gereksinim duyulmaktadır? Bu sorunun yanıtını bulabilmek için alternatif seçeneklere bakabiliriz. Dolaşılabilir nesneler bize bir grup bilgiyi tek hamlede değil parça parça vermektektir. Eğer dolaşılabilir nesneler olmasaydı bu bilgiler bize tek hamlede bir bellek alanı harcanarak verilirdi. Örneğin bir dosyadan her yinelemede bir satırın verildiğini düşünelim. Eğer bunu yapan bir dolaşılabilir nesnemiz olmasaydı bütün satırlar bize tek hamlede bir liste biçiminde verilirdi. Bu da büyük bir liste nesnesinin oluşturulmasına yol açardı. Halbuki bazı uygulamalarda biz dosyayı satır satır işleme sokmak isteyebiliriz. Bu durumda tüm satırların gereksiz bir biçimde tek hamlede depolanıp bize verilmesi yer israfına yol açar. Dolaşılabilir nesneler tüm bilgiyi tek hamlede değil istenildiğinde bize vermektedir. Tüm bilginin tek hamlede bize verilmesi yalnızca yer israfına yol açmaz aynı zamanda zaman kaybına da yol açılmaktedir. Örneğin bir dizin ağacındaki tüm dosyaların bir hamlede bize verilmesi önemli bir beklemeye yol açabilir. Bunun yerine dizin ağacındaki dosyaların teker teker gerekiğinde bize verilmesi bu bekleme zamanını tüm sürece yazardılmaktadır.

Dolaşılabilir nesneler aynı zamanda çokbüçimli bir arayüz de sunmaktadır. Örneğin list, tuple, set gibi sınıflar bizden dolaşılabilir nesneleri alarak onlardan liste, demet, küme gibi nesneler oluştururlar. Böylece çok sayıda bilgiyi aktarmak için çokbüçimli ortak bir arayüz kullanılabilmektedir.

Üretici Fonksiyonlar (Generators)

Üretici fonksiyon (generator) kavramı pek az dilde vardır. C, C++, Java, C# gibi statik tür sistemine sahip dillerde yoktur. Üretici fonksiyonlar normal birer fonksiyon gibi tanımlanırlar. Fonksiyonun içerisinde en az bir yield deyimi kullanılmışsa bu fonksiyon üretici bir fonksiyon olur. yield deyiminin genel biçimini söyleyelim:

```
yield <ifade>
```

Üretici bir fonksiyon aynı zamanda bir dolaşım nesnesi oluşturmaktadır. Üretici fonksiyon çağrılığında geri dönüş değeri olarak bir nesne elde edilir. Bu nesne "generator" isimli bir sınıf türünden bir dolaşım (iterator) nesnesidir. generator sınıfının __next__ metodu da ilerlemeyi sağlamaktadır. (Animsanacağı gibi her dolaşım nesnesi aynı zamanda __iter__ metodunu da içermek zorundadır yani her dolaşım nesnesi aynı zamanda dolaşılabilir bir nesnedir.) Üretici fonksiyonun verdiği nesne dolaşılırken her __next__ çağrılarında akış fonksiyonda ilk yield deyimi görülene kadar ilerler ve yield deyiminde geçici süre durdurulur. Böylece her __next__ çağrısı sonraki yield deyimine kadar çalışmayı sağlamaktadır. __next__ metodu yield anahtar sözcüğünün yanındaki ifadenin değerine geri döner. Üretici fonksiyon bittiğinde (bu bitiş akışın fonksiyonu bitirmesiyle de olabilir, return deyimiyle de olabilir) bu durum StopIteration exception'ının oluşmasına yol açmaktadır. Örneğin:

```
def foo():  
    print('one')  
    yield 1  
    print('two')  
    yield 2  
    print('three')  
    yield 3  
  
iterator = foo()  
  
val = iterator.__next__()  
print(val)  
  
val = iterator.__next__()
```

```
print(val)

val = iteatorr.__next__()
print(val)
```

Mademki üretici fonksiyonlar çağrıldığında bize bir dolaşım nesnesi veriyorlar, o halde biz bu nesneyi for deyiminde ve başka yerlerde kullanabiliriz. (Dolaşım nesnelerinin aynı zamanda dolaşılabılır nesneler olduğunu anımsayınız.) Örneğin:

```
def foo():
    print('one')
    yield 1
    print('two')
    yield 2
    print('three')
    yield 3

for i in foo():
    print(i)

l = list(foo())
print(l)
```

Burada foo fonksiyonu çağrıldığında bir dolaşım nesnesi elde edilmektedir. Bu dolaşım nesnesi de her adımda (yani her next işleminde) fonksiyonu kaldığı yerden yield deyimine kadar çalışıp yield deyiminde duraklatmaktadır. yield deyiminde belirtilen değer o yinelemeden elde edilen değerdir. yield deyiminde yield anahtar sözcüğünün yanına bir ifade yazılmayabilir. Bu durumda yield işleminden None değeri elde edilmektedir.

Örneğin:

```
def bar(n):
    for i in range(n):
        print('New iteration')
        yield i
    print('end')

print(type(bar))      # <class 'function'>

g = bar(3)
print(type(g))        # <class 'generator'>

for i in g:
    print(i)
```

Programın çıktısı şöyle olacaktır:

```
<class 'function'>
<class 'generator'>
New iteration
0
New iteration
1
New iteration
2
end
```

Üretici fonksiyonun çalışmasını for döngüsü temelinde özet olarak ele alalım: Bir üretici fonksiyon for döngüsüyle dolaşılarıken bu döngünün her yinelenmesinde yield deyiminde belirtilen değer döngü değişkenine atanmaktadır. for döngüsünün her yinelenmesinde fonksiyon kalınan yerden sonraki yield deyimine kadar çalışmaya devam ettilir.

`yield` deyimindeki değer döngü değişkenine atanarak o noktada geçici süre durdurulur ve döngüsünün içerisindeki deyimler çalıştırılır. En sonunda fonksiyon bittiğinde döngü de biter.

`yield` ile `return` deyimlerinin birbirlerine karıştırılmaması gereklidir. `return` fonksiyonu sonlandırmaktadır. Üretici fonksiyonlarda `return` kullanıldığında (ya da fonksiyon `return` deyimini görmeden sonlandığında) tüm fonksiyon sonlanır ve bu durum `StopIteration` exception'ının oluşmasına yol açar. `yield` ise çalışmayı geçici süre durdurmaktadır. `yield` deyiminde durdurulan fonksiyon akışı yeni bir `__next__` çağrıyla kalınan yerden devam ettirilmektedir. Şimdi de üretici fonksiyonun çalışmasını adım adım yeniden açıklayalım:

1. Üretici fonksiyonun kendisi normal bir fonksiyon gibidir. Bunun türüne baktığımızda "class <function>" olduğunu görürüz.

2. Üretici fonksiyonu çağrıdığımızda "generator" isimli bir sınıfından bir nesne elde ederiz. Bu generator sınıfı bir dolaşım (iterator) sınıfıdır.

3. Üretici fonksiyonun çağrısından elde ettiğimiz generator sınıfından sınıf nesnesinin `__next__` metodu fonksiyonu kaldığı yerden devam ettirir ve ilk `yield` deyimi gördüğünde durdurur. Dolaşımdan elde edilen değer bu `yield` deyiminde belirtilen değerdir.

4. Dolaşım nesnesinde `__next__` uygulandığında üretici fonksiyon sonlanırsa bu durum `StopIteration` isimli exception'ının oluşmasına yol açmaktadır.

Örneğin:

```
def geteven(iterable):
    for x in iterable:
        if x % 2 == 0:
            yield x

l = [2, 5, 7, 8, 4]
for i in geteven(l):
    print(i, end=' ')
print()
```

Burada biz foo üretici fonksiyonuna dolaşılabilir bir nesne geçirdik. foo fonksiyonu da o nesnedeki çift sayıları bize `yield` deyimleriyle teker teker verdi.

Üretici fonksiyonlarla yapılan işlemlerin benzerleri dolaşılabilir sınıflarla da yapılabilmektedir. Ancak üretici fonksiyonlar akışı durdurup devam ettirdikleri halde dolaşılabilir sınıflar her defasında `__next__` metodunun yeniden çalıştırılması ile işlemini yapmaktadır. Örneğin ilk n tane sayıyı bize veren bir üretici fonksiyon yazalım:

```
def get_primes(count):
    def isprime(val):
        if val % 2 == 0:
            return val == 2
        for i in range(3, int(val ** 0.5) + 1, 2):
            if val % i == 0:
                return False
        return True

    i = 2
    while count > 0:
        if isprime(i):
            count -= 1
            yield i
        i += 1
```

Aynı işlemi dolaşılabilir sınıf oluşturarak da yapabiliyoruz:

```

class GetPrimes:
    def __init__(self, count):
        self.count = count

    def __iter__(self):
        self.i = 2
        return self

    def __next__(self):
        while self.count > 0:
            self.i += 1
            if self.isprime(self.i - 1):
                self.count -= 1
                return self.i - 1

        raise StopIteration

    @staticmethod
    def isprime(val):
        if val % 2 == 0:
            return val == 2
        for i in range(3, int(val ** 0.5) + 1, 2):
            if val % i == 0:
                return False
        return True

```

Dolaşılabilir sınıfın yazımı üretici fonksiyon yazımından biraz daha zahmetlidir. Çünkü dolaşılabilir sınıflarda akış üretici fonksiyonlarda olduğu gibi durdurulmamaktadır. Her yinelemede `__next__` metodu sonlanan kadar çalışma baştan başlatılmaktadır. Bu nedenle dolaşılabilir sınıflarda durum bilgisinin nesnenin örnek özniteliklerinde saklanması gerekmektedir. Tabii sınıfların da üretici fonksiyonlara göre avantajları vardır. Sınıflar durum bilgilerini daha iyi bir biçimde tutabilirler. Türetme işlemlerine açıktırlar. Nesne yönelimli bir kullanım olanlığı sağlarlar.

Üretici fonksiyonlar yalnızca `yield` işlemleriyle değer elde etmek amacıyla kullanılmak zorunda değildir. Bunlar gerçekten bir değer elde etmenin ötesinde akışın durdurulup çalıştırılması gereği durumlarda da kullanılabilmekte dirler. Örneğin:

```

def foo():
    print('one')
    yield
    print('two')
    yield
    print('three')

try:
    g = foo()
    iterator = iter(g)
    next(iterator)
    next(iterator)
    next(iterator)
except:
    pass

```

Python'ın standart kütüphanesindeki pek çok fonksiyon aslında üretici fonksiyon olarak yazılmıştır. Başka bir deyişle bu fonksiyonlar baştan nesneleri bir listeye yerleştirip bize vermezler. Her yinelemede o anda bulup bize verirler. Yani talep edildiğinde bize verirler. Örneğin os modülünün içerisindeki `walk` isimli fonksiyon dizin ağaçını dolaşmaktadır. Ancak bu dolaşım adım adım üretici fonksiyon mekanizmasıyla yapılır. Yani `walk` her çağrıda sıradaki yeni bir dizine geçilir ve o dizin üzerinde işlem yapılır. `walk` fonksiyonu bizden dolaşılacak dizinin kökünü ister. Her dolaşımda bize bir demet verir. Bu demetin elemanları sırasıyla kök dizinin yol ifadesine ilişkin str nesnesi, dizin içerisindeki dizinlerin yol ifadelerine ilişkin bir str dizisi ve dizin içerisindeki dosyaların yol ifadelerine ilişkin bir

str dizisidir. Başka bir deyişle walk her yield işleminde bize (str, [str] ve [str]) biçiminde biz demet vermektedir. Örneğin biz de for döngüsünde açık yaparak dolaşım yapabiliriz:

```
import os

for root, dirs, files in os.walk('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src'):
    print(root)
```

Çıktıyı bir dosyaya da şöyle aktarabiliriz:

```
import os

try:
    f = open('list.txt', 'w')
    for root, dirs, files in os.walk('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src'):
        f.write(root + '\n')
    f.close()
except:
    print('file cannot open!')
```

Python'da belli bir versiyondan sonra yield deyimi değer yerlestirmek amacıyla da kullanılmıştır. Yani yield deyiminden de bir değer elde edilip üretici fonksiyond bu değer kullanılabilmiştir. yield deyiminden bir değerin elde edilebilmesi için generator nesnesinin send isimli metodu kullanılmalıdır. send metoduna geçirilen parametre akış yield deyiminden devam ettirilirken oluşturulacak değeri belirtmektedir. Örneğin:

```
def foo():
    x = yield
    print(x)
    x = yield
    print(x)
    x = yield
    print(x)

iterator = foo()

try:
    next(iterator)
    iterator.send(10)
    iterator.send(20)
    iterator.send(30)
except StopIteration:
    pass
```

Biz şimdide kadar akış yield deyiminde durdurulduğunda yield deyiminde belirtilen değeri elde etmişik. Oysa burada tam ters bir işlem uygulanmıştır. Yukarıdaki örnekte yield deyimi bize bir değer vermemekte bir ona bir değer vermektedir. generator nesnesinin send isimli örnek metodu akışı en son durdurulmuş olan yield deyiminden bir değer göndererek devam ettirmektedir. Dolaşımı devam ettiren next işlemi ile send işlemi arasında şöyle bir fark vardır: next uygulandığında akış kalınan yerden sonraki yield deyimine kadar devam ettirilir. Akış sonraki yield deyiminde durdurulur ve o yield deyiminde belirtilen ifade next işleminden elde edilir. send metodu da benzer şeyi yapmasına karşın send metodu akışı kalınan yerden devam ettirirken üretici fonksiyonda değer olmasını sağlamaktadır. Başka bir deyişle hem next hem de send işlemi akışı kalınan yerden devam ettirip sonraki yield deyiminde durdurmaktadır. Ancak next sonraki yield deyiminde belirtilen değerin elde edilmesine yol açarken, send akışın devam ettirildiği yerde değer oluşturulmasına yol açmaktadır.

Şimdi yukarıdaki örnekte adım adım nelerin gerçekleştiğine bakalım. Önce üretici fonksiyon çağrılarak dolaşım nesnesi elde edilmiştir. Sonra next işlemi uygulanarak akış ilk yield deyiminde durdurulmuştur. Sonra send işlemi ile akış kaldığı yerden çalıştırılırken 10 değerinin oluşturulması sağlanmıştır. Dolayısıyla ekran'a 10 değeri basılmıştır. Sonra aynı işlem 20 ve 30 değerlerinin send işlemiyle oluşturulması biçiminde devam ettirilmiştir. Böylece yukarıdaki programda ekran çıktısı şöyle olacaktır:

```
10  
20  
30
```

Tabii aslında yield işleminde hem bir değer verilip hem de bir değer oluşturulabilir. Örneğin:

```
x = yield 10
```

Bu durumda send metodu da aslında değer oluşturuluktan sonra akışı sonraki yield deyimine kadar çalıştmakla birlikte tıpkı next işlemind eolduğu gibi akışın durdurulduğu yield deyiminde belirtilen değeri de geri döndürmektedir. Yani send metodu yalnızca değer göndermekte değil aynı zamanda sonraki yield deyimindeki değeri de tıpkı next gibi almaktadır. Örneğin:

```
def foo():  
    x = yield 10  
    print(x)  
    x = yield 20  
    print(x)  
    x = yield 30  
    print(x)  
  
iterator = foo()  
  
try:  
    val = next(iterator)  
    val = iterator.send(val * val)  
    val = iterator.send(val * val)  
    iterator.send(val * val)  
except StopIteration:  
    pass
```

Burada ilk next çağrı ile akış ilk yield deyiminde durdurulmuş ve next çağrılarından 10 değeri elde edilmiştir. Sonra send metodu elde edilen 10 değerinin karesi olan 100 değeri argüman yapılarak çağrılmıştır. Böylece üretici fonksiyonun içerisindeki x değişkenine 100 atanmıştır. Bu send çağrı ile akış sonraki yield deyiminde durdurulmuş ve o yield deyiminde belirtilen 20 değeri send metodunun geri dönüş değeri olarak elde edilmiştir. Sonra bu 20 değerinin karesi olan 400 ile send metodu çağrılmış bu kez de bu çağrıdan 30 değeri elde edilmiştir. Nihayet son olarak send metodu 30'un karesi olan 900 değeri ile çağrılmıştır. Yukarıdaki kod çalıştırıldığında şöyle bir çıktı elde edilmelidir:

```
100  
400  
900
```

Pekiyi yukarıdakine benzer üretici fonksiyonlarda ilk yinelemeye nasıl başlanmalıdır? send yerine next, next yerine send kullanılırsa ne olur? send metodu değer oluşturarak kalınan yerden devam etmeye sağladığına göre işin başında send metodunun kullanılmasının bir anlamı yoktur. Çünkü işin başında zaten akış fonksiyonun başından başlatılıp ilk yield deyiminde durdurulmaktadır. O halde başlangıçta bir kez next çağrı yaparak işe başlamak uygun olur. Tabii dolaşım bunun yerine işin başında da send metodu herhangi bir parametreyle (tipik olarak None) çağrılarak da başlatılabilir. Bu dueurmda akış yeni başlatıldığı için send metodunun parametresinde belirtilen değer dikkate alınmayacaktır. Aslında send metodu tamamen next yerine kullanılabilmektedir. Ancak next metodu send yerine kullanılamamaktadır. Eğer send metodu yerine next fonksiyonu kullanılırsa üretici fonksiyonda None değeri oluşturmaktadır. Başka bir deyişle aslında:

```
val = next(iterator)
```

Bu çağrı ile aşağıdaki eşdeğerdir:

```
val = iterator.send(None)
```

Örneğin:

```
def foo():
    x = yield
    print(x)

try:
    iterator = foo()
    val = next(iterator)      # val = iterator.send(None)
    print(val)
    next(iterator)          # iterator.send(None)
except StopIteration:
    pass
```

Bu kodun çalıştırılması sonucunda şu çıktı elde edilecektir:

```
None
None
```

Üretici Fonksiyonlara Neden Gereksinim Duyulmaktadır?

Üretici fonksiyon mekanizması fonksiyonu durdurup bir sonuç elde edip onun kalınan yerden çalışmasına devam etmesini sağlayan bir mekanizmadır. Dolayısıyla elde edilen değerler bir hamlede değil adım adım elde edilmektedir. Bu da bellek kullanımı bakımından ve organizasyon bakımından faydalı sağlamaktadır. Örneğin os.walk fonksiyonu tüm ağacı dolaşıp ağacı bize bir liste olarak verseydi bunun ne dezavantajı olurdu? Birincisi bu kadar dosyanın yol ifadelerinin yerleştirileceği liste çok büyük olurdu. (Belki de programcı bunun içerisinde bir dosyayı aramaktadır. Bu dosyayı bulunca işleme devam etmek istemeyebilir. Bu basit işlem için bile bütün ağacın bir listeye çekilmesi verimsiz bir yöntemdir.) İkincisi dizin ağacının dolaşılması uzun bir zaman alır. Bu işlem istenildiği zaman da kesilemezdi. Halbuki üretici fonksiyonlar sayesinde hem önceden bir bellek tahsisatının yapılmasına gerek kalmamaktadır hem de bu işlem parça parça yapılmaktadır. Üstelik de işlem folaşım sırasında istenildiği zaman kesilebilmektedir.

Peki C, C++, Java gibi derleyici temelli dillerde üretici fonksiyonlar dile dahil edilemez miydi? Derleyici temelli dillerde üretici fonksiyonların derleyici tarafından gerçekleştirilmesi oldukça zordur. Python, Ruby gibi bir yorumlayıcı biçiminde çalışması bu gerçekleştirimin yapılmasını kolaylaştırmaktadır. Diğer dillerde daha çok bu tür işlemler "geri çağrılan fonksiyonlar (callback functions)" yoluyla yapılmaktadır. Ancak derleyici temelli bazı programlama dillerine de (örneğin C#'a) üretici fonksiyonlar sokulmaya başlanmıştır.

Burada bir noktayı yinelemek istiyoruz. Şüphesiz üretici fonksiyonlar ile yapılan bazı işlemler dolaşılabilir sınıflar yoluyla da yapılmaktadır. Örneğin range işlemini yapan mekanizma hep dolaşılabilir bir sınıf yoluyla hem de üretici fonksiyon yoluyla oluşturulabilir. Şüphesiz üretici fonksiyon yazmak daha pratiktir. Ancak dolaşılabilir sınıflar neticede sınıf oldukları için soyutlamaya daha elverişlidir. Dolaşılabilir sınıfların akışı durdurup yeniden kalınan yerden çalıştırıldığına dikkat ediniz. Dolaşılabilir sınıflarda durum bilgisi sınıfın örnek özniteliklerinde saklanmaktadır. Halbuki üretici fonksiyonlarda durum bilgisi doğrudan fonksiyonun yerel değişkenlerinde saklanır. Örneğin üretici fonksiyonlar bir iç fonksiyon durumunda olabilir ve dış fonksiyonun yerel değişkenlerini kullanabilir. Dolaşılabilir sınıflar genel olarak üretici fonksiyonlardan daha hızlı çalışma eğilimindedir. Yorumlayıcının üretici fonksiyonu durdurup yeniden çalıştırması daha fazla bilgisayar zamanına yol açmaktadır.

Üretici İfadeler (Generator Expressions)

İçlemler konsusunda liste içemlerinin, küme ve sözlük içemlerinin nasıl oluşturulduğunu görmüştük. O bölümde demetler için bir içem mekanizmasının olmadığına dikkat etmişsinizdir. Aslında sentaksı demet içemi gibi olan fakat içem değil üretici fonksiyonlar biçiminde işlev gören bir ifade türü bulunmaktadır. Bunlara "üreticili ifadeler (generator expressions)" denilmektedir. Üretici ifadeler tamamen bir demet içemi gibi kullanılmaktadır. Ancak elde edilen ürün bir demet değil üretici bir fonksiyondur. Üretici ifadelerin genel biçim şöyledir:

(<iclem sentaksı>)

Üretici ifadelerden bir üretici nesnesi (generator) elde edilmektedir. Bu üretici nesnesi de bir dolaşım nesnesidir. Bu nesne dolaşıldığında içlem sentaksındaki sıradaki ifadenin değeri elde edilmektedir.

Örneğin:

```
g = (i * i for i in range(10))

for i in g:
    print(i, end=' ')
```

Göründüğü gibi burada bir demet içlemi değil üretici bir ifade oluşturulmuştur. Üretici ifadelerden elde edilen üretici dolaşıldığında her dolaşımda önce for döngüsü çalıştırılır, varsa if koşulu dikkate alınır ve soldaki ifadenin değeri hesaplanır. Dolaşımdan bu değer elde edilmektedir. Yukarıdaki kodun işlevsel eşdeğeri aşağıdaki gibidir:

```
def some_func():
    for i in range(10):
        yield i * i

g = some_func()
x = list(g)
print(x)
```

Örneğin:

```
g = (i for i in range(100) if i % 7 == 0)

a = list(g)
print(a)
```

Üretici ifadeler birer ifade durumunda olduğu için başka ifadelerde ve deyimlerde onların çeşitli parçalarında kullanılabilir. Örneğin:

```
for i in (i for i in range(100) if i % 7 == 0):
    print(i, end=' ')
```

Örneğin:

```
def foo(g):
    for i in g:
        print(i, end=' ')
```

```
foo((i for i in range(10)))
```

Pekiyi üretici fonksiyonlar varken üretici ifadelere neden gereksinim duyulmaktadır? İşte kısa üretici fonksiyonlar söz konusu olduğunda bunun fonksiyon olarak yazılması yerine üretici ifade biçiminde yazılması daha pratiktir. Tabii kapsamlı üreticiler ancak fonksiyon biçiminde yazılabilirler. Başka bir deyişle tüm üretici ifadeleri üretici fonksiyonlar biçiminde yazabilirdir ancak tüm üretici fonksiyonları üretici ifadeler biçiminde yazamayız. Üretici ifadeler tamamen içlem sentaksına sahiptir.

Lambda İfadeleri (Lambda Expression)

Bazen bazı küçük fonksiyonların yazılarak hemen işleme sokulması gerekebilmektedir. Bu tür durumlarda Python'da önce def deyişi ile fonksiyonu başka yerde oluşturup kullanmak gereklidir. İşte lambda ifadeleri bir fonksiyonu bir ifadenin bir parçası olarak hem tanımlayıp hem de kullanma kolaylığı sağlamaktadır. Gerçekten de fonksiyonların birer ifade gibi hemen tanımlanıp işleme sokulması artık pek çok dile sokulmuştur. Örneğin C++ 2011 versiyonıyla "lambda ifadelerine" sahip olmuştur. Benzer biçimde bu mekanizma Java ve C# gibi dillere de sokulmuş durumdadır.

Lambda ifadelerinin genel biçimini söyleyelim:

```
lambda [parametre listesi] : <ifade>
```

Lambda ifadeleri tamamen bir fonksiyon gibidir. Bunların normal fonksiyonlardan tek farkı bir deyim gibi değil bir ifade gibi işleme sokulmasıdır. Anımsanacağı gibi def anahtar sözcüğüyle bir fonksiyon bildirildiğinde fonksiyonun ismi zaten bu bildirimde yer almaktadır. Halbuki lambda ifadelerinde fonksiyonun bir ismi yoktur. Tabii lambda ifadesi bir değişkene de atanabilir. Bu durumda bunun def ile tanımlanan bir fonksiyondan farklı kalmaz. Örneğin:

```
foo = lambda x, y: x + y
print(type(foo))
result = foo(10, 20)
print(result)
```

lambda ifadelerinde return anahtar sözcüğü kullanılmaz. Zaten ':' atomunun yanındaki ifade return edilecek ifadeyi belirtir. Yukarıdaki örnekte x ve y fonksiyonunun parametreleri, x + y ise geri dönüş değeridir. Bunun eşdeğeri söyleyelim:

```
def foo(x, y):
    return x + y

result = foo(10, 20)
print(type(foo))
print(result)
```

lambda ifadeleri tek bir ifadeden oluşmak zorundadır. Yani biz ':' ayıracının sağına tek bir ifade yazabilirmiz. Örneğin biz bir dolaşılabilir nesneyi ve bir fonksiyonu alıp, dolaşılabilir nesnedeki her bir elemanı bool bir fonksiyona sokan ve bu fonksiyondan True ile dönen değerleri bize bir liste olarak veren bir fonksiyon yazmak isteyelim:

```
def myfilter(f, iterable):
    return [i for i in iterable if f(i)]
```

Şimdi bu fonksiyonu aşağıdaki gibi kullanalım:

```
def iseven(n):
    return n % 2 == 0

a = [3, 67, 4, 90, 45]
b = myfilter(iseven, a)
print(b)
```

Aynı işlemi lambda ifadesiyle çok daha kolay yapabiliyoruz:

```
a = [3, 67, 4, 90, 45]
b = myfilter(lambda n: n % 2 == 0, a)
print(b)
```

Aslında standart kütüphanede bu işlemi yapan filter isimli built-in bir fonksiyon zaten bulunmaktadır. Ancak standart kütüphanedeki filter fonksiyonu bize bir liste değil dolaşılabilir bir nesne vermektedir. Örneğin:

```
a = [3, 67, 4, 90]
b = list(filter(lambda n: n % 2 == 0, a))
print(b)
```

Sınıf Çalışması: Yukarıdaki select fonksiyonunu ya da filter fonksiyonunu kullanarak son iki basamağı aynı olan sayıları (77, 66, 33 gibi) bulan lambda ifadesini yazınız.

Çözüm:

```

a = [3, 77, 4, 90, 99]
b = select(a, lambda n: n // 10 % 10 == n % 10)
print(b)

```

Şimdi de standart kütüphanedeki filter fonksiyonunu yine myfilter ismiyle üretici fonksiyon biçiminde, üretici ifade biçiminde ve dolaşılabilir bir sınıf biçiminde yazalım:

```

def myfilter(f, iterable):
    for i in iterable:
        if f(i):
            yield i

def myfilter(f, iterable):
    return (i for i in iterable if f(i))

class myfilter:
    def __init__(self, f, iterable):
        self.f = f
        self.iterator = iter(self.iterator)

    def __iter__(self):
        return self

    def __next__(self):
        while True:
            val = next(self.iterator)
            if self.f(val):
                return val

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
iterable = myfilter(lambda i: i % 3 == 0, a)
for i in iterable:
    print(i, end=' ')

```

Her elemanı bir fonksiyon ya da lambda ifadesi olan liste oluşturabiliriz. Örneğin:

```

def foo(x):
    return x ** 2

def bar(x):
    return x ** 3

def tar(x):
    return x ** 4

ff = [foo, bar, tar]

```

Bunu daha pratik lambda ifadeleriyle şöyle de yapabildik:

```
ff = [lambda x: x ** 2, lambda x: x ** 3, lambda x: x ** 4]
```

Bu diziyi şöyle kullanabiliriz:

```

ff = [lambda x: x ** 2, lambda x: x ** 3, lambda x: x ** 4]

for f in ff:
    print(f(2))

```

Lambda parametreleri lambda ifadesi içerisinde faaliyet gösterir. Yani adeta bunlar lambda ifadesini kapsayan yerel değişkenler gibidir.

Tıpkı iç içe fonksiyonlarda olduğu gibi bir fonksiyonun içerisindeki lambda ifadelerinde içinde bulunulan fonksiyonun yerel değişkenleri ve global değişkenler kullanılabilir. Örneğin:

```
def foo(n):
    return lambda x: x * n

f = foo(10)
result = f(20)
print(result)      # 200
```

Bu tür ifadelerde en çok merak edilen nokta dış fonksiyon bittiği halde onun yerel değişkenlerinin nasıl daha sonra kullanılabildiğidir. Yani yukarıdaki örnekte foo çağrımı bitince n değişkeni yok edileceğine göre nasıl lamda içerisinde kullanılabilmektedir? İşte bu tür dillerde bu n değeri lambda fonksiyonuna sanki bir parametre gibi geçirilmektedir. Böylece dış fonksiyondaki n'in yok edilmiş olması lamdayı etkilememektedir.

Lambda fonksiyonlarının parametreleri de default değer alabilir. Örneğin:

```
f = lambda x = 1, y = 2: x + y

result = f()
print(result)      # 3

result = f(10)
print(result)      # 12

result = f(10, 20)
print(result)      # 30
```

Lambda parametreleri default değer alıyorsa yine fonksiyonlardaki kural uygulanmaktadır. Yani default değer alan bir parametreden sonra default değer almayan bir parametre kullanılamaz.

Yine lambda parametreleri '*'lı ve '**'lı olabilmektedir. '*'lı ve '**'lı parametrelerin bulundurulmasına ilişkin genel kural normal fonksiyonlarda olduğu gibidir. Örneğin:

```
f = lambda *args: sum(args)

result = f(10, 20, 30)
print(result)
```

Tabii lambda ifadelerinin parametresi olması da zorunlu değildir. Örneğin:

```
f = lambda: 100

print(f())      # 100
```

Lambda fonksiyonunu çağırmak için aslında onu geçici bir değişkene atamamız da gerekmekz. Örneğin:

```
result = (lambda x: x * x)(10)
print(result)      # 100
```

Burada parantezlerin gerekli olduğuna dikkat ediniz. Örneğin sözlüğün elemanı bir lambda ifadesi de olabilir.

```
n = int(input('Bir değer giriniz:'))

val = {10: lambda: n * 2, 20: lambda: n * 3, 30: lambda: n * 4}[n]()
print(val)
```

Sınıfların, Sınıflar Türünden Nesnelerin `__dict__` Özellikleri

Python'da sınıf, sınıflar türünden nesne, fonksiyon ve metot gibi faaliyet alanlarındaki elemanlar ilgili nesnelerin `__dict__` isimli sözlük öznitelikleri içerisinde saklanmaktadır. `__dict__` sözlüğünün anahtarı elemanların str türünden isimlerinden, değerleri ise onların gerçek değerlerinden oluşmaktadır. Bu sayede biz bu faaliyet alanlarındaki elemanlara `__dict__` sözlüğü yoluyla erişebiliriz. Örneğin:

```
class Sample:  
    def foo(self):  
        print('foo')  
  
    def bar(self):  
        print('bar')  
  
x = 10
```

Buradaki Sample sınıfına biz üç eleman yerleştirmiştir durumdayız. Bunlardan foo ve bar birer metot x ise bir sınıf değişkenidir. Tabii bir sınıfın içi boş olsa bile aslında Python yorumlayıcısı o sınıfı birkaç eleman eklemektedir. İşte biz bu elemanlara istersek `__dict__` sözlüğü yoluyla da erişebiliriz. Örneğin:

```
d = Sample.__dict__  
  
f = d['foo']  
b = d['bar']  
  
s = Sample()  
  
f(s)  
b(s)  
val = d['x']  
print(val)
```

Biz burada `Sample.__dict__` ifadesi ile Sample sınıfı içerisindeki elemanları bir sözlük olarak elde ettik. Bu sözlüğe anahtar olarak sınıf elemanın ismini verdığımızda özlükten değer olarak onun gerçek değeri elde edilmektedir. Örneğin `d['foo']` ifadesi ile biz foo ismindeki elemana karşık gelen fonksiyon nesnesini yani metodu elde etmiş olduk. Sınıfların `__dict__` elemanları yoluyla sınıflara eleman eklemek mümkün değildir. Çünkü sınıfların `__dict__` elemanları `read/only` sözlükler gibi ele alınmaktadır. Tabii biz sınıflara normal yolla eleman eklediğimizde eklenen bu elemanlar `__dict__` sözlüğünde görünecektir. Örneğin:

```
>>> class Sample:  
    x = 10  
  
>>> Sample.__dict__  
mappingproxy({'_module_': '__main__', 'x': 10, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None})  
>>> Sample.y = 20  
>>> Sample.__dict__  
mappingproxy({'_module_': '__main__', 'x': 10, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None, 'y': 20})  
>>> Sample.__dict__['z'] = 30  
Traceback (most recent call last):  
  File "<pyshell#6>", line 1, in <module>  
    Sample.__dict__['z'] = 30  
TypeError: 'mappingproxy' object does not support item assignment
```

Yukarıda da gördüğünüz gibi bir sınıfın içi boş olsa bile aslında Python yorumlayıcısı o sınıfın içine birkaç eleman yerleştirmektedir.

Sınıfın `__dict__` isimli sözlük nesnesinde sınıfın taban sınıf elemanlarının bulunmadığına dikkat ediniz.

Sınıf nesneleri için de Python yorumlayıcısı `__dict__` sözlüğünü oluşturmaktadır. Bu sözlüğün içerisinde nesnenin örnek öznitelikleri bulunmaktadır. Örneğin:

```
class Sample:  
    def __init__(self):  
        self.a = 10  
        self.b = 20  
  
    def foo(self):  
        print('foo')  
  
    def bar(self):  
        print('bar')  
  
s = Sample()  
print(s.__dict__)
```

Bu koddan şöyle bir çıktı elde edilecektir:

```
{'a': 10, 'b': 20}
```

Yine sözlüğün anahtarları str türünden nesnenin özniteliklerinden sözlüğün değerleri ise o örnek özniteliklerinin gerçek değerlerinden oluşmaktadır.

Sınıf nesnelerinin `__dict__` sözlükleri read/write biçimdedir. Yani biz bu sözlüklerle eleman eklediğimizde sanki nesneye eleman eklemiş gibi oluruz. Örneğin:

```
s = Sample()  
s.__dict__['c'] = 30  
print(s.c)
```

Sınıflara ilişkin `__dict__` sözlükleri eskiden read/write biçimdeydi. Daha sonra read only yapılmıştır.

Sınıfların `__new__` Metotları

Python'da sınıflar konusunun başlarında bir sınıf nesnesi yaratıldığında sınıfın aslında önce `__new__` metodunun çağrıldığını nesnenin yaratımının bu fonksiyon tarafından yapıldığını belirtmiştik. Anımsayacağınız gibi `__init__` metod `__new__` metodundan sonra ve eğer yaratılan nesne ilgili sınıf türünden ya da ondan türemiş bir sınıf türündense çağrılmaktadır. Python'daki `__init__` metodu diğer nesne yönelik dillerdeki "başlangıç fonksiyonlarına (constructors)" benzemektedir.

Sınıfların `__new__` metotları her ne kadar `@staticmethod` ile dekore edilmemiş olsalar da static bir metot biçimindedir. Dolayısıyla `__new__` metodunun çağrımları tipik olarak sınıf ismiyle (yani tür nesnesiyle) yapılmaktadır. `__new__` metodunun birinci parametresi hangi sınıf türü ile nesne yaratılmak istenmişse o sınıfın type nesne referansını alan bir tür parametresi olmak zorundadır. Bu parametre geleneksel olarak "cls" biçiminde isimlendirilmektedir. Metodun diğer parametreleri herhangi bir biçimde organize edilebilir. Ancak buradaki parametrelerin daha sonra `__init__` metoduna aktarılabilmesi için genel bir biçimde olması uygundur. Bildiğiniz gibi her sayıda ve türde argümanı kabul edebilen bir fonksiyon ya da metodun `*args` ve `**kwargs` parametrelerine sahip olması gereklidir. Bu nedenle `__new__` metodunun parametrik yapısı tipik olarak şöyledir:

```
class Sample:  
    def __new__(cls, *args, **kwargs):  
        pass
```

Tahsisat işlemleri en genel olarak eninde sonunda object sınıfının `__new__` metodu ile yapılmaktadır. Örneğin biz object sınıfının `__new__` metodu ile Sample türünden bir nesneyi şöyle tahsis edebiliriz:

```
instance = object.__new__(Sample)
```

Programcılar seyrek olarak kendi sınıfları için `__new__` metodunu yazarlar. Eğer bir sınıf için `__new__` metodu yazılmamışsa onun ilk taban sınıfının yazılmış olan `__new__` metodunun çağrılacagina dikkat ediniz. `__new__` metodu yazılmış ilk taban sınıf da genellikle `object` sınıfı olmaktadır. Bu durumda biz bir sınıf türünden nesne yarattığımızda aslında tahsisat genellikle `object` sınıfının `__new__` metoduyla yapılmış olur.

Python'da tüm sınıfların aslında `type` isimli bir sınıf türünden nesneler biçiminde oluşturulduğunu anımsayınız. İlgili sınıfa ilişkin tüm bilgiler `type` nesnesinin içerisinde saklanmaktadır. O halde aslında biz bir sınıf nesnesini `Sample(...)` biçiminde yarattığımızda aslında `type` sınıfının `__call__` metodu çağrılmaktadır. İşte `type` sınıfının `__call__` metodu aşağıdakine benzer bir biçimde gerçekleştirilmiştir:

```
class type:
    def __call__(self, *args, **kwargs):
        instance = self.__new__(self, *args, **kwargs)

        if isinstance(instance, self):
            instance.__init__(*args, **kwargs)

        return instance

    # ...
```

Bu temsili koddan şunu anlayabiliriz: Biz bir sınıf türünden nesne yarattığımızda aslında önce o sınıfın `__new__` metodu çağrılmakta, eğer bu `__new__` metodu o sınıf türünden ya da o sınıfından türemiş bir sınıf türünden bir nesne ile geri dönmüşse bu kez o sınıfın `__init__` metodu çağrılmaktadır. Bu temsili koddan догorduguñuz gibi eğer `__new__` ilgili sınıf türünden ya da o sınıfından türetilmiş olan bir sınıf türünden bir nesne ile geri dönmezse ilgili sınıfın `__init__` metodu çağrılmamaktadır. Örneğin:

```
class Mample:
    pass

class Sample:
    def __new__(cls, *args, **kwargs):
        return object.__new__(Mample)

    def __init__(self):
        print('Sample.__init__')

s = Sample()
print(type(s))
```

Burada `Sample` nesnesi yaratıldığından `Sample` sınıfındaki `__new__` metodu çağrılmacaktır. Bu metot `Sample` sınıfı ya da ondan türetilmiş olan bir sınıf nesnesi ile geri dönmediğinden `Sample` sınıfının `__init__` metodu çağrılmayacaktır. Programdan şöyle bir çıktı elde edilmiş:

```
<class '__main__.Mample'>
```

Şimdi de `__new__` içerisinde `Sample` sınıfı türünden bir nesne tahsis edelim:

```
class Sample:
    def __new__(cls, *args, **kwargs):
        return object.__new__(cls)

    def __init__(self):
        print('Sample.__init__')
```

```
s = Sample()  
print(type(s))
```

Burada artık `__new__` `Sample` sınıfı türünden bir nesne tahsis ettiği için (`cls` parametresinin `Sample` olduğuna dikkat ediniz) `__init__` metodu çağrılmacaktır. Programın çıktısı şöyledir:

```
Sample.__init__  
<class '__main__.Sample'>
```

Pekiyi mademki biz `__new__` metodunu yazmazsa `object` sınıfının `__new__` metodu çağrılıyor o da bizim sınıfımız türünden bir nesne tahsis ediyor, o halde kendi sınıfımız için `__new__` metodounu yazmamızın ne anlama olabilir? İşte programcılar kendi sınıfları için `__new__` metodunu birkaç nedenden dolayı yazmak isteyebilmektedir. Bunları sırasıyla açıklayalım.

1) Singleton nesne yaratımını sağlamak için. Bilindiği gibi bir tanınıftan tek bir nesne yaratımına nesne yönelimli tasarım kalıplarında "singleton kalıbı" denilmektedir. Singleton kalıbında programcı kendisi farklı nesneler yaratmış gibi işlemlerini yapsa da aslında toplamda ilgili sınıf türünden tek bir yaratılır. `__new__` operatörü kullanılarak singleton kalibini söyle oluşturabiliriz:

Sınıfın Özniteliklerine Aracılıklı Erişimler (Managed Attributes)

Python'da sınıfın olmayan özniteligi erişilmek istendiğinde eğer sınıfın `__getattr__` isimli bir metodu varsa yorumlayıcı tarafından o metot çağrılmaktadır. Eğer sınıfın `__getattr__` metodu yoksa `AttributeError` exception fırlatılmaktadır. Böylece programcı olmayan özniteliklere erişimlerde aracılık işlemleri uygulayabilir. `__getttr__` metodunun `self` dışında bir parametresi daha vardır. Bu parametreye yorumlayıcı tarafından erişilmek istenen özniteligin ismi str nesnesi olarak geçirilmektedir. Başka bir deyişle `x` bir sınıf türünden değişken eğer nesnenin a isimli bir örnek özniteligi yoksa ve sınıfı `__getattr__` isimli metot bulunuyorsa yorumlayıcı `s.a` işlemini `s.__getattr__('a')` biçiminde gerçekleştirir. Örneğin:

```
class Sample:  
    def __init__(self, a):  
        self.a = a  
  
    def __getattr__(self, attr):  
        print(f'__getattr__: {attr}')  
        return 0  
  
s = Sample(10)
```

Burada `Sample` nesnesi yaratıldığında `a` isimli bir örnek özniteligi oluşturulmuştur. Var olan bir örnek özniteligi erişim yapılırken `__getattr__` metodu çağrılmaz. Örneğin:

```
x = s.a  
print(x)
```

Burada `s.a` erişiminden 100 değeri elde dilektir. Dolayısıyla ekrana 100 değeri yazdırılacaktır. Ancak olmayan bir örnek özniteligi erişim yapıldığında sınıfın `__getattr__` isimli metodu çağrılmaktadır. Örneğin:

```
x = s.b  
print(x)
```

Burada `s.b` ifadesiyle sınıfın olmayan bir örnek özniteligi erişim yapılmaktadır. Bu durumda sınıfın `__getattr__` metodu çağrılmaktadır. `__getattr__` metodunun geri döndürdüğü 0 değeri `x` değişkenine atanmaktadır. Bu iki satırı çalıştırığınızda ekranda şunları görmelisiniz:

```
__getattr__: b  
0
```

Örneğin:

```
class Sample:  
    def __init__(self):  
        self.a = 10  
        self.b = 20  
  
    def __getattr__(self, name):  
        if name == 'c':  
            return 100  
        if name == 'd':  
            return 200  
        if name == 'e':  
            return 300  
  
        raise AttributeError(f'Sample object has no attribute \'{name}\'')  
  
s = Sample()  
  
x = s.a  
print(x)  
  
x = s.b  
print(x)  
  
x = s.c  
print(x)  
  
x = s.d  
print(x)  
  
x = s.e  
print(x)
```

Nesnede yeni bir örnek özniteliği oluşturulduğunda `__getattr__` metodu çağrılmamaktadır. Örneğin:

```
s.x = 10
```

Burada sınıfın x isimli yeni bir örnek özniteliği yaratılmıştır. Bu nedenle `__getattr__` metodu çağrılmayacaktır.

Sınıf türünden bir değişkenler sınıfın olmayan bir metodu çağrıldığında da `__getattr__` metodu çağrılmaktadır. Örneğin:

```
s.foo()
```

Burada foo metodu çağrıldığında s.foo biçiminde bir erişim söz konusu olduğu için sınıfın `__getattr__` metodu çağrılacaktır. Ancak bu metot 0 değerine geri döndüğü için ve int bir değişken fonksiyon çağrıma operatörü ile kullanılmayacağından dolayı exception oluşacaktır. Yukarıdaki çağrı sonucunda ekranda aşağıdakine benzerşeyler göreceksiniz:

```
__getattr__: foo  
Traceback (most recent call last):  
  File "/Users/KaanAslan/Dropbox/Study/Python/PyCharm-Sample/sample.py", line 19, in <module>  
    s.foo()  
TypeError: 'int' object is not callable
```

Tabii aynı metodun sınıf ismiyle çağırırsaydık bu durumda `__getattr__` metodu çağrılmazdı. Örneğin:

```
Sample.foo(s)
```

Bu çağrı doğrudan exception olmasına yol açacaktır. Sample.foo işleminde artık değişkenin Sample olduğuna ve bunun türünün de type olduğuna dikkat ediniz.

Pekiyi `__getattr__` metoduna neden gereksinim duyulmaktadır? Bazen olmayan bir örnek özniteligi erişilirken aslında başka işlemler yapılabilir. Böylece bir çeşit property erişimi mümkün hale getirilebilmektedir. (Property erişimi Python'da dekoratörler yoluyla yapılmaktadır. Örneğin:

```
import math

class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def __repr__(self):
        return f'center: ({self.x}, {self.y}), radius: {self.radius}'

    def __getattr__(self, attr):
        if attr == 'area':
            return math.pi * self.radius ** 2

        raise AttributeError(f'Circle object has no attribute \'{attr}\'')

c = Circle(10, 12, 3)
print(c)
print(c.area)
```

Burada Circle sınıfın aslında area diye bir örnek özniteligi yoktur. Bu nedenle bu örnek özniteligi erişilmek istediğiinde `__getattr__` metodu çağrılacaktır. Bu metot içerisinde eğer metot area örnek özniteligi ile çağrılmışsa çemberin alanı geri döndürülmektedir. Dolayısıyla bu örnekte area bir örnek özniteligi gibi kullanıldığında radius değerinden hareketle çemberin alanını vermektedir. Örneğin:

```
c = Circle(10, 12, 3)

print(c.area)
```

Bu işlemden şöyle bir çıktı elde edeceksiniz:

```
28.27433882308138
```

Yukarıdaki örnekte area dışında olmayan örnek özniteliklerine erişildiğinde `AttributeError` exception'ının fırlatıldığına da dikkat ediniz.

Sınıflar için `__getattr__` metodunun yanı sıra işlev olarak ona benzeyen bir de `__getattribute__` metodu vardır. `__getattribute__` yalnızca olmayan örnek öznitelikleri için değil tüm örnek öznitelikleri için çağrılmaktadır. Başka bir deyişle x bir sınıf türünden değişken olmak üzere eğer sınıfın `__getattribute__` isimli metodu varsa x.a ifadesini yorumlayıcı x.`__getattribute__`('a') biçiminde gerçekleştirilmektedir. Örneğin:

```
class Sample:
    def __init__(self, a):
        self.a = a

    def __getattribute__(self, attr):
```

```
print(f'__getattribute__: {attr}')
return 0
```

Şimdi aşağıdaki gibi bir kod çalıştırıralım:

```
s = Sample(10)

x = s.a
print(x)

x = s.b
print(x)
```

Burada hem var olan a örnek özniteligiine hem de var olmayan b örnek özniteligiine erişilmiştir. İşte __getattribute__ metodu hem olan hem de olmayan örnek özniteliklerine erişimde çağrılır. Yukarıdaki kod çalıştırıldığında aşağıdaki çıktı elde edilkecektir:

```
__getattribute__: a
0
__getattribute__: b
0
```

Pekiyi sınıfta hem __getattr__ hem de __getattribute__ metodu birlikte bulunursa ne olur? İşte bu durumda olmayan örnek özniteliklerine erişim sırasında __getattr__ metodu çağrılmaz yalnızca __getattribute__ metodu çağrılır. (Tabii bu iki metodun sınıfta birlikte bulundurulması bu nedenden dolayı anlamsızdır.)

__getattribute__ metodu içerisinde sınıfın örnek özniteliklerine erişimde özyinelemeli sonsuz döngü oluşabilir. Çünkü bu erişimde de yine __getattribute__ metodunun kendisi çağrılmaktadır. Örneğin:

```
class Sample:
    def __init__(self, a):
        self.a = a

    def __getattribute__(self, attr):
        if attr == 'a':
            return self.a

        raise AttributeError(Sample object has no attribute '{attr}'')
```

Burada __getattribute__ metodunda erişilen örnek özniteliği a ise self.a ile sınıfın a örnek özniteligiine geri dönülmek istenmiştir. Ancak bu erişiminb kendisi de __getattribute__ metodunun çağrımasına yol açacağından dolayı özyinelemeli sonsuz döngü oluşacaktır. Bu durumu aşağıdaki kodla test edebilirsiniz:

```
s = Sample(10)

x = s.a      # bu noktada özyinelemeli sonsuz döngü oluşacaktır
print(x)
```

Pekiyi bu durumda gerçekten __getattribute__ metodunda sınıfın örnek özniteliklerine nasıl erişebiliriz? Bu işlemin ilk bakışta sanki __dict__ örnek özniteliği ile yapabileceğini düşünebilirsiniz:

```
class Sample:
    def __init__(self, a):
        self.a = a

    def __getattribute__(self, attr):
        if attr == 'a':
            return self.__dict__[attr]
```

```
raise AttributeError(f'Sample object has no attribute \'{attr}\'')
```

Ama self.__dict__ işleminde de yine __getattribute__ metodu çağrılmadan dolayı yine özyinelemeli sonsuz döngü oluşacaktır. Bu işlem ancak object sınıfının __getattribute__ metodu yoluyla yapılabilir. object sınıfının __getattribute__ metodu nesnenin tüm örnek özniteliklerine erişebilmektedir. O halde erişim şöyle olmalıdır:

```
object.__getattribute__(self, attr)
```

Bu ifade object sınıfının __getattribute__ metodunun çağrımasına yol açar. Bu metoda self olarak kendi nesnemiz geçirilecektir. Örneğin:

```
class Sample:
    def __init__(self, a):
        self.a = a

    def __getattribute__(self, attr):
        if attr == 'a':
            return object.__getattribute__(self, attr)

    raise AttributeError(f'Sample object has no attribute \'{attr}\'')
```

Örneğin:

```
import math

class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def __repr__(self):
        return f'center: ({self.x}, {self.y}), radius: {self.radius}'

    def __getattribute__(self, attr):
        if name == 'area':
            return math.pi * object.__getattribute__(self, 'radius')

        return object.__getattribute__(self, attr)

c = Circle(10, 12, 3)

print(c.area)
print(c.x, c.y)
print(c.radius)
```

Sınıfın bir örnek özniteliğine atama yapılırken (ya da bu örnek özniteliği ilk kez yaratılırken) sınıfın __setattr__ isimli metodu çağrılmaktadır. Bu metot daha önce aynı isimli bir örnek özniteliği yaratılmış olsa da yaratılmamış olsa da her atama işleminde çağrılmaktadır. __setattr__ metodunu __getattribute__ metodunun atama yapıldığında çağrılan biçimi olarak düşünebilirsiniz. Python'da ayrıca __setattribute__ isimli özel bir metot yoktur.

__setattr__ metodu self parametresinin dışında iki parametreye daha sahiptir. Bunlardan ilki özniteliğin ismini ikincisi ise bunun değerini almaktadır. Bu durumda eğer sınıfta __setattr__ isimli bir metot varsa x.a = y gibi bir işlemi yorumlayıcı x.__setattr__('a', y) biçiminde gerçekleştirilmektedir. Örneğin aşağıdaki gibi bir Sample sınıfı tanımlamış olalım:

```
class Sample:  
    def __setattr__(self, attr, val):  
        print(f'__setattr__: {attr}, {val}')
```

Şimdi şöyle bir kod çalıştıralım:

```
s = Sample()  
s.a = 10
```

Burada s.a = 10 işleminde __setattr__ metodunun self parametresine s, attr parametresine 'a' ve val parametresine de 10 değeri aktarılacaktır. Yani bu işlem aşağıdaki ile eşdeğer olacaktır:

```
s.__setattr__('a', 10)
```

Sınıfımızda eğer __setattr__ metodu varsa __init__ içerisindeki atamalarda da bu metodun çağrılacağına dikkat ediniz.

__setattr__ metodu dicerisinde sınıfın ilgili elemanına atama yapmaya çalışırsak yine özyinelemeli sonsuz döngü oluşacaktır. Örneğin:

```
class Sample:  
    def __setattr__(self, attr, val):  
        if attr == 'a':  
            self.a = val  
  
s = Sample()  
s.a = 10      # özyinelemeli sonsuz döngü
```

Fakat bu atamayı eğer __dict__ yoluyla yapabildik:

```
class Sample:  
    def __setattr__(self, attr, val):  
        if attr == 'a':  
            self.__dict__[attr] = val
```

Buradaki self.__dict__ ifadesinde __dict__ değişkenine atama yapılmadığına __dict__ sözlüğünün attr anahtarına atama yapıldığına dikkat ediniz. Tabii atamayı yine object sınıfı yardımıyla da yapabildik:

```
class Sample:  
    def __setattr__(self, attr, val):  
        if attr == 'a':  
            object.__setattr__(self, attr, val)
```

object sınıfında da nesnenin elemanlarına atama yapıldığında çağrılan bir __setattr__ metodu olduğuna dikkat ediniz.

Aşağıda __getattr__ ve __setattr__ metodları kullanılarak aslında olmayan radius ve area özniteliklerine hesaplama yapılarak erişilmektedir:

```
import math  
  
class Circle:  
    def __init__(self, x, y, radius):  
        self.x = x  
        self.y = y  
        self.radius = radius  
  
    def __repr__(self):  
        return f'center: ({self.x}, {self.y}), radius: {self.radius}'
```

```

def __getattr__(self, attr):
    if attr == 'area':
        return math.pi * self.radius ** 2

    raise AttributeError(f'Sample object has no attribute \'{attr}\'')

def __setattr__(self, attr, value):
    if attr == 'area':
        self.__dict__['radius'] = math.sqrt(value / math.pi)
    else:
        self.__dict__[attr] = value

c = Circle(10, 12, 3)

print(c.area)
c.area = 5
print(c.area)

```

Örnek Özniteliklerine Erişimde Property Kullanımı

Nesne yönelimli programlama tekniğinin temel prensiplerinden birinin "sınıfın veri elemanlarının gizlenmesi (data hiding)" olduğundan bahsetmiştik. Anımsayacağınız gibi sınıfların örnek öznitelikleri o sınıfların içsel işleyişiyle ilgilidir. Sınıfların örnek öznitelikleri isim bakımından, tür bakımından zaman içerisinde değiştirilme eğilimindedir.

Nesne yönelimli programlama tekniğinde sınıfların örnek özniteliklerine (genel terim olarak vriteri elemanlarına") değer atanırken ya da onların değerleri alırken birtakım işlemlerin yapılması gerekebilmiştir. Örneğin Date isimli bir sınıf nesnesi bir tarih bilgisini tutuyor olabilir. Bu sınıfın örnek öznitelikleri tutulan tarih bilgisinin gün, ay, yıl bileşenleri olabilir. Dolayısıyla bu örnek özniteliklerine atanınan değerlerin belli sınırlarda olması gerekebilmiştir. Örneğin programcı tarihin gün bileşenine yanlışlıkla çok büyük bir değer atarsa tüm sınıfın çalışması bozulabilir. Pek çok dilde bu tür geçerlilik kontrolleri tipik olarak veri elemanlarına erişimlerde kullanılan get ve set metodları yoluyla yapılmaktadır. Bazı programlama dillerinde ise veri elemanlarına erişme işlemi metot çağrıma sentaksiyle değil, veri elemanına erişme sentaksiyle yapılmaktadır. İşte bazı programlama dillerinde bir veri elemanından değer alırken ya da ona değer yerleştirirken arka planda çalıştırılan metodlara "property" denilmektedir.

Önceki bölümde Python'da veri elemanlarına aracılık erişimin dolayısıyla property etkisinin Python'da sınıfların `__getattribute__`, `__getattribute__` ve `__setattribute__` metodlarıyla yapılabildiğini gördük. Ancak Python'da veri elemanlarına aracılık erişimi kolaylaştmak için isimli özel bir sınıf da tasarlanmıştır. property sınıfı aynı zamanda bir dekoratör olarak da kullanılabilmektedir. property sınıfı Python dokümanlarında "built-in" bir fonksiyon biçiminde ele alınmaktadır.

property sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Metodun dört parametresi olduğunu görüyorsunuz. Bu parametreler sırasıyla veri elemanından değer alındığında çağrılacak fonksiyonu, veri elemanına değer yerleştirilirken çağrılacak fonksiyonu, veri elemanı del deyiği ile silinmeye çalışıldığında çağrılacak fonksiyonu ve veri elemanı için oluşturulacak doküman yazısı belirtmektedir. (Doküman yazısı sonraki bölümlerde ele alınmaktadır.)

property sınıfı tipik olarak şöyle kullanılmaktadır: Programcı sınıf bildirimini içerisinde property sınıfı türünden bir nesne yaratır. Bu nesneyi (yani onun adresini) sınıfın property olarak kullanılacak değişkenine atar. Böylece artık bu değişkene örnek özniteligi sentaksi ile erişildiğinde property nesnesini yaratırken belirlenen fonksiyonlar çağrılacaktır. Örneğin:

```

class Sample:
    def __init__(self, a):
        self._a = a

```

```

def get_a(self):
    return self._a

def set_a(self, value):
    self._a = value

def del_a(self):
    del self._a

a = property(get_a, set_a, del_a)

s = Sample(10)
x = s.a          # x = s.get_a()
print(x)
s.a = 20         # s.set_a(20)
x = s.a          # x = s.get_a()
print(x)
del s.a          # s.del_a()

```

Örneğin:

```

import math

class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def __repr__(self):
        return f'center: ({self.x}, {self.y}), radius: {self.radius}'

    def _get_area(self):
        return math.pi * self.radius ** 2

    def _set_area(self, value):
        self.radius = math.sqrt(value / math.pi)

area = property(_get_area, _set_area)

c = Circle(10, 12, 3)

print(c.area)
c.area = 5
print(c.area)

```

Property'ler dekoratör biçiminde de kullanılabilmektedir. Örneğin:

```

class Sample:
    def __init__(self, a):
        self._a = a

    @property
    def a(self):
        return self._a
    # a = property(a)

    @a.setter
    def a(self, value):
        self._a = value
    # a = a.setter(a)

```

```

@a.deleter
def a(self):
    del self._a
# a = a.deleter(a)

s = Sample(10)
x = s.a
print(x)
s.a = 20
x = s.a
print(x)
del s.a

```

Örneğin:

```

import math

class Circle:
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def __repr__(self):
        return f'center: ({self.x}, {self.y}), radius: {self.radius}'

    @property
    def area(self):
        return math.pi * self.radius ** 2

    # area = property(area)

    @area.setter
    def area(self, value):
        self.radius = math.sqrt(value / math.pi)

    # area = area.setter(area)

c = Circle(10, 12, 3)

print(c.area)
c.area = 5
print(c.area)

```

Pekiyi acaba `property` oluşturan dekoratörler nasıl yazılmıştır? Aynı etkiyi yaratacak biçimde biz bu dekoratörleri yazabilir miydi? İşte bu tür dekoratörler "betimleyici (descriptor)" denilen özel sınıflar kullanılarak etkin biçimde yazılmaktadır. Aslında `property` sınıfı da bu anlamda bir betimleyici sınıfıdır.

Betimleyici Sınıflar

Betimleyici sınıflar Python'a sonradan eklenmiştir. Betimleyici sınıflar daha işlevsel dekoratörlerin yazılmasına olanak vermektedir. Önceki konuda gördüğümüz `property` isimli dekoratör sınıfını yazmaya çalışmak mevcut bilgilerimizle bunu gerçekleştirememeyiz.

`__get__`, `__set__` ve `__delete__` isimli metodlardan herhangi birine sahip olan sınıflara betimleyici sınıf denilmektedir. Sınıfın bu özel metodlarının parametrik yapıları aşağıdaki gibi olmalıdır:

```

class SampleDescriptor:
    def __get__(self, instance, owner):
        pass

```

```

def __set__(self, instance, value):
    pass

def __delete__(self, instance):
    pass

```

Betimleyici sınıflar türünden nesneler sınıf değişkenlerine atanırsa o değişkenlere erişimde betimleyici sınıflardaki bu özel `__get__`, `__set__` ve `__del__` metotları devreye girmektedir. Örneğin:

```

class SampleDescriptor:
    def __get__(self, instance, owner):
        print('__get__')

    def __set__(self, instance, value):
        print('__set__')

    def __delete__(self, instance):
        print('__delete__')

class Sample:
    a = SampleDescriptor()

s = Sample()
x = s.a      # SampleDescriptor sınıfının __get__ metodu çağrılacak
s.a = 10     # SampleDescriptor sınıfının __set__ metodu çağrılacak
del s.a     # SampleDescriptor sınıfının __delete__ metodu çağrılacak

```

Bu örnekte `Sample` sınıfının içerisinde `a` isimli sınıf değişkeni şöyle yaratılmıştır:

```

class Sample:
    a = SampleDescriptor()

```

İşte artık bu işlemden sonra sınıfın `a` değişkenine bu sınıf türünden bir değişkenle değer alma amaçlı erişildiğinde `SampleDescriptor` sınıfının `__get__` metodu, sınıfın `a` değişkenine atama yapıldığında `SampleDescriptor` sınıfının `__set__` metodu ve bu `a` değişkeni `del` deyimi ile silinmeye çalışıldığında `SampleDescriptor` sınıfının `__del__` metodu çağrılacaktır.

Peki bu metodlara hangi değerler parametre olarak aktarılmaktadır. İşte sınıf değişkenine okuma amaçlı erişildiğinde `__get__` metodunun `self` dışındaki ilk parametresine erişilen nesne (örneğimizde `s` nesnesi), ikinci parametresine ise erişilen sınıf geçirilmektedir. Sınıf değişkenine yazma amaçlı erişildiğinde ise `__set__` metodunun `self` dışındaki ilk parametresine yine erişilen nesne (örneğimizde `s`), ikinci parametresine ise ona atanınan değer (örneğimizde 10 değeri) geçirilmektedir. `__delete__` metodunun `self` dışındaki ilk parametresine ise yine silinmek istenen nesne (örneğimizde `s`) geçirilmektedir. Örneğin:

```

class MyDescriptor:
    def __get__(self, instance, owner):
        print(self)
        print(instance)
        print(owner)

    def __set__(self, instance, value):
        print(self)
        print(instance)
        print(value)

    def __delete__(self, instance):
        print(self)
        print(instance)

class Sample:

```

```

a = MyDescriptor()

s = Sample()

s.a
print('-----')
s.a = 10
print('-----')
del s.a

```

Bu kodun çalıştırılması sonucunda aşağıdaki çıktı elde edilecektir:

```

<__main__.MyDescriptor object at 0x0000019D087EE310>
<__main__.Sample object at 0x0000019D087EE0A0>
<class '__main__.Sample'>
-----
<__main__.MyDescriptor object at 0x0000019D087EE310>
<__main__.Sample object at 0x0000019D087EE0A0>
10
-----
<__main__.MyDescriptor object at 0x0000019D087EE310>
<__main__.Sample object at 0x0000019D087EE0A0>

```

Biz artık betimleyici sınıfı yoluya sınıfın bir örnek özniteligi erişebiliriz. Örneğin:

```

class SampleDescriptor:
    def __get__(self, instance, owner):
        return instance._a

    def __set__(self, instance, value):
        instance._a = value

    def __delete__(self, instance):
        del instance._a

class Sample:
    def __init__(self, a):
        self._a = a

    a = SampleDescriptor()

s = Sample(10)
x = s.a
print(x)
s.a = 20
x = s.a
print(x)
del s.a

```

Aynı betimleyici sınıfı isterseniz aracılıklı erişimde birden fazla değişken için de kullanabiliriz. Bunun için betimleyici sınıfı nesnesi oluşturulurken ona nesnenin hangi değişken ile ilişkili olduğunu anlatan bir parametre geçirmemiz gereklidir.. Örneğin:

```

class MyDescriptor:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

```

```

def __delete__(self, instance):
    del instance.__dict__[self.name]

class Sample:
    def __init__(self):
        self._a = 0
        self._b = 0

    a = MyDescriptor('_a')
    b = MyDescriptor('_b')

s = Sample()
s.a = 10
print(s.a)

s.b = 20
print(s.b)

```

Bu durumda aslında property sınıfını kendimiz de şöyle yazabiliriz:

```

class myproperty:
    def __init__(self, fget = None, fset = None, fdel = None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel

    def __get__(self, instance, owner):
        if self.fget is None:
            raise AttributeError('cannot get value!..')
        return self.fget(instance)

    def setter(self, f):
        self.fset = f
        return self

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError('cannot set value!..')
        return self.fset(instance, value)

    def deleter(self, f):
        self.fdel = f
        return self

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError('cannot set value!..')
        return self.fdel(instance)

class Sample:
    def __init__(self, a):
        self._a = a

    def get_a(self):
        return self._a

    def set_a(self, value):
        self._a = value

    def del_a(self):
        del self._a

```

```

a = myproperty(get_a, set_a, del_a)

s = Sample(10)
x = s.a
print(x)
s.a = 20
x = s.a
print(x)
del s.a

```

Yazdığımız betimleyici sınıfını dekoratör olarak da kullanabiliriz:

```

class Sample:
    def __init__(self, a):
        self._a = a

    @myproperty
    def a(self):
        return self._a
    # a = myproperty(a)

    @a.setter
    def a(self, value):
        self._a = value
    # a = a.setter(a)

    @a.deleter
    def a(self):
        del self._a
    # a = a.deleter(a)

s = Sample(10)
x = s.a
print(x)

s.a = 20
x = s.a
print(x)

del s.a

```

Bazı Önemli Built-In Fonksiyonlar

Bu bölümde şimdije kadar görülmemiş olan bazı önemli built-in fonksiyonlar ele alınacaktır.

filter Fonksiyonu

Bu fonksiyon bir fonksiyon ve bir de dolaşılabilir nesneyi parametre olarak alır. Dolaşılabilir nesnenin her elemanını fonksiyona sokar; fonksiyonun True geri döndürdüğü elemanları verecek bir dolaşım nesnesine geri döner. Örneğin:

```

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = filter(lambda x: x % 2 == 0, a)
print(list(b))

```

Sınıf Çalışması: Daha önce üretici fonksiyon biçiminde yazmış olduğumuz primes fonksiyonunu kullanarak ilk 100 asal sayı içerisinde basamaklarından birinde 7 olan değerleri filter fonksiyonuyla elde ediniz.

Çözüm:

```

def is_prime(val):
    if val % 2 == 0:
        return val == 2
    for i in range(3, val, 2):
        if val % i == 0:
            return False
    return True

def primes(count):
    i = 2
    while count > 0:
        if is_prime(i):
            count -= 1
            yield i
        i += 1

result = filter(lambda x: str(x).find('7') != -1, primes(100))
print(list(result))

```

filter fonksiyonunu basit bir biçimde üretici fonksiyon olarak yazabiliriz:

```

def myfilter(f, iterable):
    for i in iterable:
        if f(i):
            yield i

a = [7, 9, 2, 1, 10]

for i in myfilter(lambda i: i > 5, a):
    print(i, end=' ')

```

Benzer biçimde filter fonksiyonu dolaşılabilir bir sınıf biçiminde de şöyle yazılabildi:

```

class myfilter:
    def __init__(self, f, iterable):
        self.f = f
        self.iterable = iterable
        self.it = iter(self.iterable)

    def __iter__(self):
        return self

    def __next__(self):
        while True:
            val = next(self.it)
            if self.f(val):
                return val

a = [7, 9, 2, 1, 10]

for i in myfilter(lambda i: i > 5, a):
    print(i, end=' ')

```

eval Fonksiyonu

eval Fonksiyonu bizden str türünden bir parametre ister. Bizden aldığı yazıyı bir Python ifadesi olarak yorumlar ve bunun sonucunu geri dönüş değeri biçiminde verir. Örneğin:

```
x = 10
y = eval('x + 10')
print(y)      # 20
```

Örneğin:

```
y = eval('[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]')
print(y)
```

Ancak biz eval parametresi olarak tek bir ifade verebiliriz. Birden fazla ifadeyi bu biçimde veremeyiz. Örneğin:

```
s = input('Bir liste giriniz:')
l = eval(s)
for i in l:
    print(i, end=' ')
print()
```

eval fonksiyonu ile bir hesap makinesini basit bir biçimde şöyle oluşturabiliriz:

```
from math import *

while True:
    try:
        exp = input('calc>')
        if exp.strip() == 'exit':
            break
        result = eval('(' + exp + ')')
        print(result)
    except Exception as e:
        print(e)
```

Örnek bir kullanım şöyle olabilir:

```
calc>2 * (3 + 2)
10
calc>sqrt(10) + 2
5.16227766016838
calc>a := 10
10
calc>a ** 2
100
calc>
```

exec Fonksiyonu

Bu fonksiyon bizden bir Python kodunu alır ve onu çalıştırır. eval fonksiyonundan farkı tek bir ifadeyi değil bir kod bloğunu çalıştırabilmesidir. Örneğin:

```
s = 'x = 10; a = [1, 2, 3, 4, 5]; print(x); print(a)'
exec(s)
```

Örneğin:

```
s = input('Bir program parçası giriniz:')
exec(s)
```

exec ona parametre olarak verdığımız yazının içerisindeki kodu sanki normal bir kod gibi çalıştmaktadır. Örneğin:

```
def foo(a):
    return a * a
```

```
s = 'x = foo(10)'  
exec(s)  
print(x)
```

Python'da Paketler (Packages)

Bir grup modülü (yani Python dosyasını) barındıran dizinlere Python'da paket (package) denilmektedir. Modüller (yani Python dosyaları) dosyalara benzettirilse paketler de dizinlere benzetebilir. Bir paket kendi içerisinde başka paketleri ve modülleri içerebilmektedir. Paketler de yine import deyimi ile yüklenirler. Onlar da birer modül statüsündedir.

Pyton'da bir dizinin bir paket olması için `__init__.py` isimli dosyanın o dizinde bulunuyor olması gereklidir. (Python yorumlayıcıları bunu zorunlu tutmamaktadır. Ancak programcının içi boş olsa da `__init__.py` dosyasını dizin içerisinde bulundurması şiddetle tavsiye edilmektedir. O halde biz içerisinde `__init__.py` isimli dosya bulunan bir dizini sanki bir modülmüş gibi import deyimi ile yükleyebiliriz. Tabii bu dizinin daha önceden de beliretilmiş gibi sys.path dizininde bulunuyor olması gereklidir. Yani Python'da içerisinde `__init__.py` dosyası bulunan normal bir dizine paket, bir Python dosyasına da modül denilmektedir. Ancak paketler de import edildikten sonra birer modül haline gelirler. Örneğin A isimli bir dizini yaratıp bunun içersine içi boş bir `__init__.py` dosyası eklemiş olalım. Sonra bu A dizinini sys.path listesine ekleyip import edelim:

```
>>> sys.path.append('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src')  
>>> import A  
>>> type(A)  
<class 'module'>
```

Bir paket import ile yükleniğinde onun içerisindeki `__init__.py` dosyası çalıştırılır. Örneğin `__init__.py` dosyasının içeriği şöyle olsun:

```
>>> import A  
this is the A package
```

Bir paket içerisinde modüller ve başka paketler bulunabilir. Örneğin A dizinin içerisinde x.py isimli aşağıdaki gibi bir dosya bulunuyor olsun:

```
print('I am x module')  
  
class Sample:  
    def __init__(self, a):  
        self.a = a  
    def __str__(self):  
        return str(self.a)
```

Biz bir paket içerisindeki bir modülü paket ismini belirterek import edebiliriz. Örneğin:

```
>>> import A.x  
I am x module
```

Şimdi de bunun içerisindeki Sample sınıfını kullanmak isteyelim. Bu durumda bunu da yine modül ismiyle kombine etmemiz gereklidir:

```
>>> s = A.x.Sample(10)  
>>> print(s)  
10
```

Tabii biz bu Sample ismini niteliksiz olarak kullanabilmek için from import deyiminden faydalananabiliriz:

```
>>> from A.x import Sample
```

```
>>> s = Sample(10)
>>> print(s)
10
```

A paketinin içerisindeki x modülünü biz tek hamlede import edemeyiz. Önce A paketini import edip sonra A.x deme hakkına sahip olarak bu x modülünü import etmemiz gereklidir. import edilen isimlerin birer değişken statüsünde olduğunu anımsayınız. Aynı durum from import deyimi için de geçerlidir.

Pekiyi paketi temsil eden dizinin içerisindeki `__init__.py` dosyasının işlevi nedir? İşte bir paket import edildiğinde bu dosya otomatik olarak çalıştırılmaktadır. Programcı da genellikle burada diğer import işlemlerini yapar. Örneğin:

```
# __init__.py
print('this is the A package')
import A.x
```

Şimdi biz A modülünü import ettiğimizde `__init__.py` modülü çalıştırılacağı için x modülü de import edilmiş olacaktır. Tabii biz yine x modülüne A.x biçiminde erişiriz. Örneğin:

```
>>> import sys
>>> sys.path.append('D:\Dropbox\Kurslar\Python-Subat-2018\Src')
>>> import A
this is the A package
I am x module
>>> s = A.x.Sample(10)
>>> print(s)
10
```

Daha programcılar `__init__.py` içerisinde form import deyimi ile alt modüllerdeki isimleri yüklemektedir. Böylece dışarıdan kullanım daha kolay olur. Örneğin:

```
# __init__.py
print('this is the A package')
from A.x import *
```

Şimdi artık paket içerisindeki modüldeki isimleri şöyle kullanabiliriz:

```
>>> import sys
>>> sys.path.append('D:\Dropbox\Kurslar\Python-Subat-2018\Src')
>>> import A
this is the A package
I am x module
>>> s = A.Sample(10)
>>> print(s)
10
```

Burada önemli bir noktaya değinmek istiyoruz. Paketin içerisindeki `__init__.py` dosyasında biz `from import` deyimi ile isimleri çektiğimiz zaman bu isimler global isim alanına değil o paketin isim alanına alınmaktadır. Dolayısıyla örneğimizde biz import işleminden sonra dışarıdan x.py içerisindeki Sample ismini doğrudan Sample olarak değil A.Sample biçiminde kullandık.

Aslında Python'in standart modüllerinin çoğu birer modül dosyası değil pakettir. O paketlerin içerisindeki `__init__.py` dosyasında `from import` deyimi ile isimler paketin içerisinde aktarılmıştır. Şüphesiz biz global isim alanında da yeniden `from import` komutunu kullanarak bu isimleri yine global alana taşıyabiliriz. Örneğin:

```
>>> from A import *
```

```
>>> s = Sample(10)
>>> print(s)
10
```

Bir paket başka paketleri de içerebilir. Bu durumda iç paketler de yine dışarıdan hareketle ayrı import deyimleriyle yüklenebilir. Örneğin A dizinin içerisinde bir de B dizini olsun. Onun içerisinde de yine bir `__init__.py` dosyası bir de `y.py` modülü olsun:

```
A
  __init__.py
  x.py
B
  __init__.py
  y.py
```

Şimdi biz buradaki `y.py` modülünü adım adım yükleyip içerisindeki `Mample` sınıfını kullanmak isteyelim:

```
>>> import sys
>>> sys.path.append('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src')
>>> import A
this is the A package
I am x module
>>> import A.B
this is the B package
>>> import A.B.y
>>> m = A.B.y.Mample(10)
>>> print(m)
10
```

Modüllerdeki `__all__` Değişkeni

Bir modülün içerisinde `__all__` isimli bir str listesi bulundurulabilir. Bu liste modül aşağıdaki gibi `from import *` ile kullanıldığından modülden dışarıya aktarılacak isimleri belirtmektedir:

```
from <modül ismi> import *
```

Yani biz bir modüldeki tüm isimleri niteliksiz olarak kullanmak için `from import *` deyimini uyguladığımızda aslında yalnızca `__all__` listesi içerisindeki isimler bize verilmektedir. Tabii bu `__all__` listesi modül dosyasında hiç belirtilmeyebilir. Bu durumda tüm isimler dışarıya verilmektedir. Örneğin `sample.py` dosyasının içeriği şöyle olsun:

```
# sample.py

__all__ = ['foo', 'x']

def foo():
    print('foo')

def bar():
    print('bar')

def tar():
    print('tar')

x = 10
```

Şimdi bu `sample.py` modülünü aşağıdaki `from import` işlemine sokalım:

```
>>> import sys
>>> sys.path.append('D:\\Dropbox\\Kurslar\\Python-Subat-2018\\Src')
>>> from sample import *
```

```

>>> foo()
foo
>>> x
10
>>> bar()
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    bar()
NameError: name 'bar' is not defined
>>>

```

Burada görüldüğü gibi foo fonksiyonu ve x değişkeni dışarıya verilmiştir. Ancak bar ve tar fonksiyonları dışarıya verilmemiştir. Tabii bu `__all__` listesi yalnızca yıldızlı import deyiminde etki göstermektedir. Yoksa biz `__all__` listesi içerisinde olmayan isimleri de kullanabiliriz. Örneğin:

```

>>> from sample import bar
>>> from sample import tar
>>> bar()
bar
>>> tar()
tar

```

Dokümantasyon Yazıları (Documentation Strings)

Python dünyasına adım atanların önemli bir bölümü modüllerin, fonksiyonların ve sınıfların yetersiz biçimde dokümant edildiğini düşünmektedir. Maalesef dokümantasyon zaafı bu programlama dilinin neredeyse bir özelliği haline gelmiştir.

Python'da en önemli dokümantasyon unsurlarından biri dokümantasyon yazılarıdır. Dokümantasyon yazıları modül, fonksiyon, metot gibi öğelerden hemen sonra yerleştirilen bir string ile oluşturulmaktadır. Örneğin:

```

def square(a):
    "return the square of x"
    return a * a

```

Düğüman yazısı bir str nesnesi olacak biçimde yerleştirilmeldir. Yani yazındaki stringtek tırnak, çift tırnak, üç tek tırnak ya da üç çift tırnak biçiminde olabilir. Üç tırnaklarla oluşturulan string'lerin birden fazla satırda bölünebildiğini anımsayınız. Bu durumda eğer düğüman yazısı birden çok satırda oluşturulacaksa üç tırnak kullanılmalıdır. Örneğin:

```

def square(a):
    """
    square(a)

    return the square of x
    """
    return a * a

```

Üç tırnaklı string'lerden sonraki yeni satır karakterlerinin (newline) ve tab karakterlerinin yazında bulundurulduğuna dikkat ediniz. Eğer bunu istemiyorsanız sola dayalı bir biçimde de yazıyı düzenleyebilirsiniz.

```

def square(a):
    """
    square(a)

    return the square of x"""

    return a * a

```

Bir düğüman yazısı ilgili nesnenin `__doc__` isimli elemanından elde edilebilir. Örneğin:

Döküman yazıları modüllere, fonksiyonlara, sınıflara ve metodlara yerleştirilebilir. Modüllere yerleştirileceğse ilgili kaynak dosyanın başında bulunmalıdır. Örneğin:

```
# test.py

"""this is a test module
this module contains...."""

def foo():
    """this is a foo function"""

class Sample:
    """this is a Sample class"""

    def bar(self):
        "this is a bar method"
```

Şimdi başka bir modülden bu modüldeki doküman yazılarını elde edelim:

```
import test

print(test.__doc__)
print(test.foo.__doc__)
print(test.Sample.__doc__)
print(test.Sample.bar.__doc__)
```

Döküman yazılarının nasıl oluşturulması gereği konusunda standart bir yöntem kullanılmamaktadır. Değişik kütüphaneler kendilerine göre farklı bir format kullanmaktadır. Örneğin standart kütüphane içerisindeki sqrt fonksiyonunun doküman yazısını yazdırıralım:

```
import math

print(math.sqrt.__doc__)
```

Şöyledir bir yazı görüyoruz:

```
sqrt(x)

Return the square root of x.
```

Şimdi de numpy kütüphanesindeki bir doküman yazısını görüntüleyelim:

```
import numpy

print(numpy.sqrt.__doc__)
```

Buradaki doküman yazısının oldukça ayrıntılı olduğunu görüyoruz:

```
sqrt(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature,
extobj])
```

```
Return the non-negative square-root of an array, element-wise.
```

Parameters

```
-----
x : array_like
    The values whose square-roots are required.
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or `None`,
    a freshly-allocated array is returned. A tuple (possible only as a
```

```
keyword argument) must have length equal to the number of outputs.  
where : array_like, optional  
    Values of True indicate to calculate the ufunc at that position, values  
    of False indicate to leave the value in the output alone.  
**kwargs  
    For other keyword-only arguments, see the  
    :ref:`ufunc docs <ufuncs.kwargs>`.
```

Returns

y : ndarray

```
An array of the same shape as `x`, containing the positive  
square-root of each element in `x`. If any element in `x` is  
complex, a complex array is returned (and the square-roots of  
negative reals are calculated). If all of the elements in `x`  
are real, so is `y`, with negative elements returning ``nan``.  
If `out` was provided, `y` is a reference to it.  
This is a scalar if `x` is a scalar.
```

See Also

`lib.scimath.sqrt`

```
A version which returns complex numbers when given negative reals.
```

Notes

```
*sqrt* has--consistent with common convention--as its branch cut the  
real "interval" [-inf, 0), and is continuous from above on it.  
A branch cut is a curve in the complex plane across which a given  
complex function fails to be continuous.
```

Examples

```
>>> np.sqrt([1,4,9])  
array([ 1.,  2.,  3.])
```

```
>>> np.sqrt([4, -1, -3+4j])  
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, numpy.inf])  
array([ 2.,  NaN,  Inf])
```

Yukarıdaki doküman yazı formatı şu bölümlerden oluşmaktadır:

```
<Fonksiyonun parametrik yapısı>  
<Kısa açıklama>  
<Fonksiyonun parametreleri, onların türleri ve anlamı>  
<Fonksiyon geri dönüş değeri, i türü ve anlamı>  
<<Örnekler>
```

Numpy'da kullanılan doküman yazısı formatını aşağıdaki bağlantıdan inceleyebilirsiniz:

<https://numpydoc.readthedocs.io/en/latest/format.html>

Google tarafından kullanılan doküman yazısı formatını da aşağıdaki bağlantıdan inceleyebilirsiniz:

<https://numpydoc.readthedocs.io/en/latest/format.html>

Python'daki help isimli built-in fonksiyon bir modül, sınıf fonksiyon, metot hakkında bilgi edinmek için kullanılmaktadır. Bu fonksiyon ilgili elemanın doküman yazısının yanı sıra aynı zamanda bazı başka kaynaklardan elde edilen bilgileri de ekranaya yazdırmaktadır dolayısıyla bu fonksiyonla görüntüleyeceğiniz yazı doküman yazısından daha ayrıntılı olabilmektedir. Örneğin:

```
>>> import math  
>>> help(math.sqrt)
```

Help on built-in function sqrt in module math:

```
sqrt(x, /)
    Return the square root of x.
```