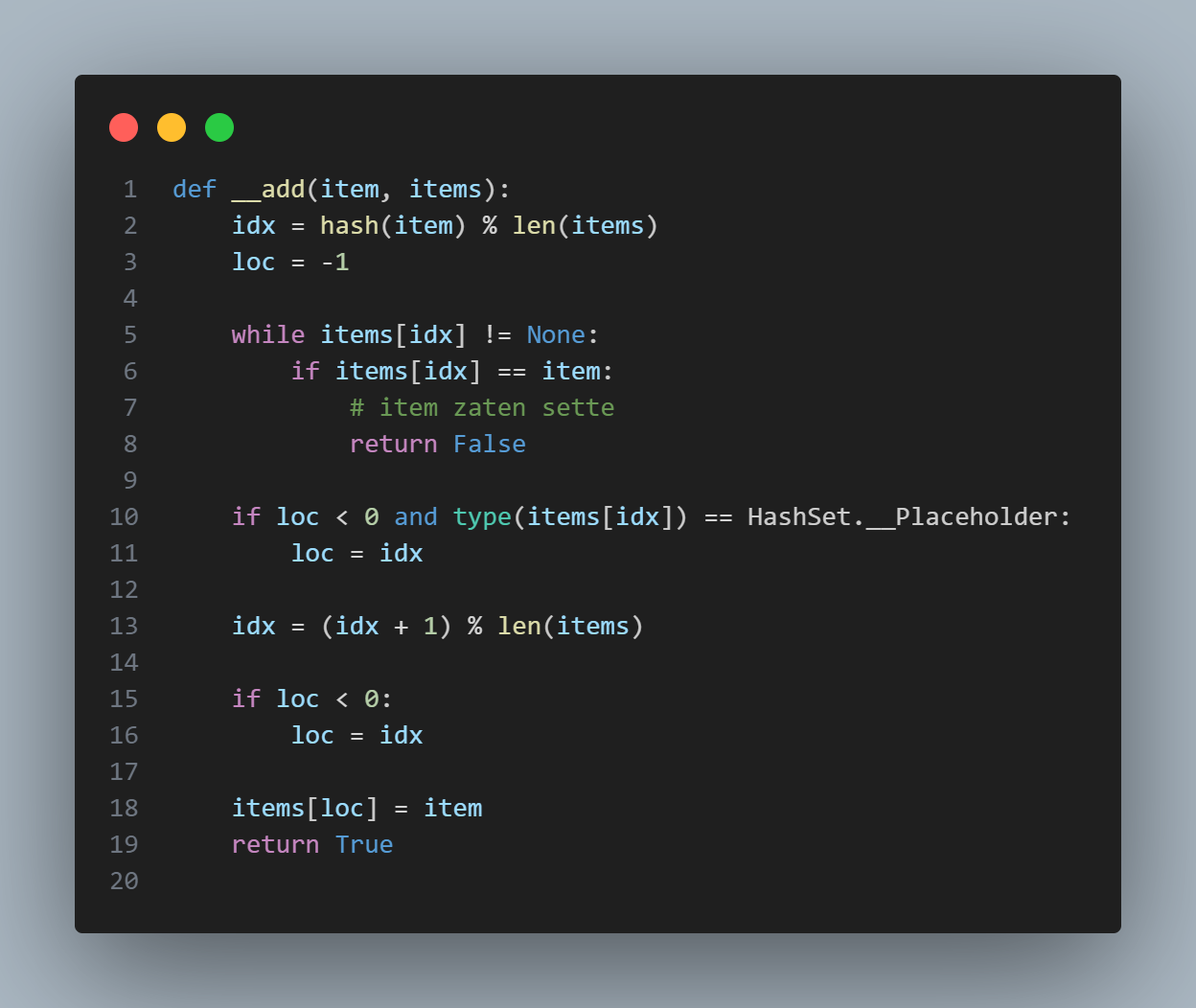
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.4 HashSet Add Yardımcı Fonksiyonu**



Bölüm 5.5.4'teki kod, zaten listede bulunan bir öğeyi eklemez. While döngüsü, kodun doğrusal adresleme (linear probing) kısmıdır. Öğe bulunana ya da bir None değeriyle karşılaşılana kadar idx indeksi, listenin uzunluğu modunda artırılır. None değerinin bulunması, doğrusal zincirin sonunu ve dolayısıyla öğenin zaten kümede olup olmadığını belirlemek için yapılan doğrusal aramanın tamamlandığını gösterir. Eğer öğe listede yoksa, öğe ya arama sırasında bulunan ilk \_\_Placeholder nesnesinin konumuna ya da zincirin sonundaki None değerinin konumuna eklenir.

Bir değer eklerken dikkate alınması gereken bir konu daha vardır. Hash küme listesinde yalnızca bir boş konum kaldığını düşünelim. Yukarıdaki kodda ne olurdu? Doğrusal arama, tüm listenin taranmasıyla sonuçlanırdı. Eğer liste dolu olsaydı, sonuç sonsuz bir döngü olurdu. Bu durumların hiçbirinin yaşanmasını istemiyoruz. Aslında, bir öğeyi amortize O(1) zamanda ekleyebilmek istiyoruz. Amortize edilmiş O(1) zaman karmaşıklığını sağlamak için listenin asla tamamen dolu veya neredeyse dolu olmaması gerekir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.5 Yük Faktörü**

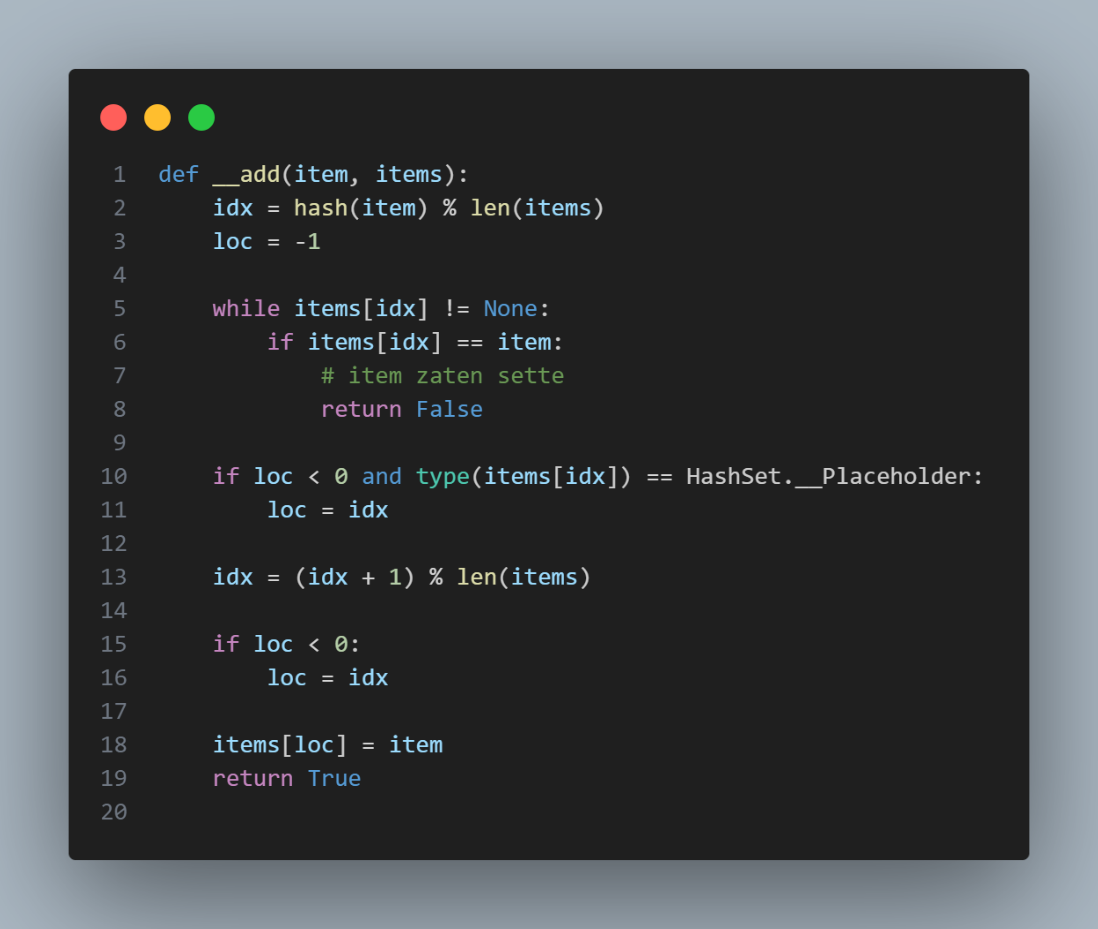
Karma küme listesinin doluluk oranına **yük faktörü** denir. Bir hash kümesinin yük faktörünü, listede saklanan öğe sayısını listenin uzunluğuna bölerek hesaplayabiliriz. Çok düşük bir yük faktörü, listenin içindeki öğe sayısına kıyasla çok daha büyük olduğu ve çarpışma olasılığının düşük olduğu anlamına gelir. Yüksek bir yük faktörü ise daha verimli alan kullanımı sağlarken çarpışma olasılığını artırır. Deneyler, en uygun yük faktörlerinin belirlenmesine yardımcı olabilir, ancak makul bir maksimum yük faktörü **%75** doluluktur.

Listeye bir değer eklerken, yük faktörü %75’i aşarsa, listedeki tüm değerler yeni bir listeye aktarılmalıdır. Değerleri yeni bir listeye aktarırken, yeni listenin uzunluğu farklı olacağından değerlerin tekrar **hashlenmesi** gerekir. Bu işleme **rehashing** denir. Hash kümesi uygulamasında, yeniden hashleme gerektiğinde listenin boyutunu **iki katına çıkarmayı** tercih ettik.

Bölüm 5.5.6'daki kod, Bölüm 5.5.4'teki \_\_add fonksiyonunu çağırır. Bu kod ve \_\_add yöntemi **HashSet** sınıfında yer almaktadır. \_\_add ve \_\_rehash fonksiyonları, genel erişime açık add metodunun kullandığı gizli yardımcı fonksiyonlardır.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.6 HashSet Ekle**



Yük faktörü kontrol altında tutulduğundan, listeye bir değer eklemenin amortize edilmiş karmaşıklığı **O(1)**'dir. Bu, liste içindeki herhangi bir zincirin uzunluğunun, hash kümesindeki öğe sayısından bağımsız olarak **sonlu** olacağı anlamına gelir.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

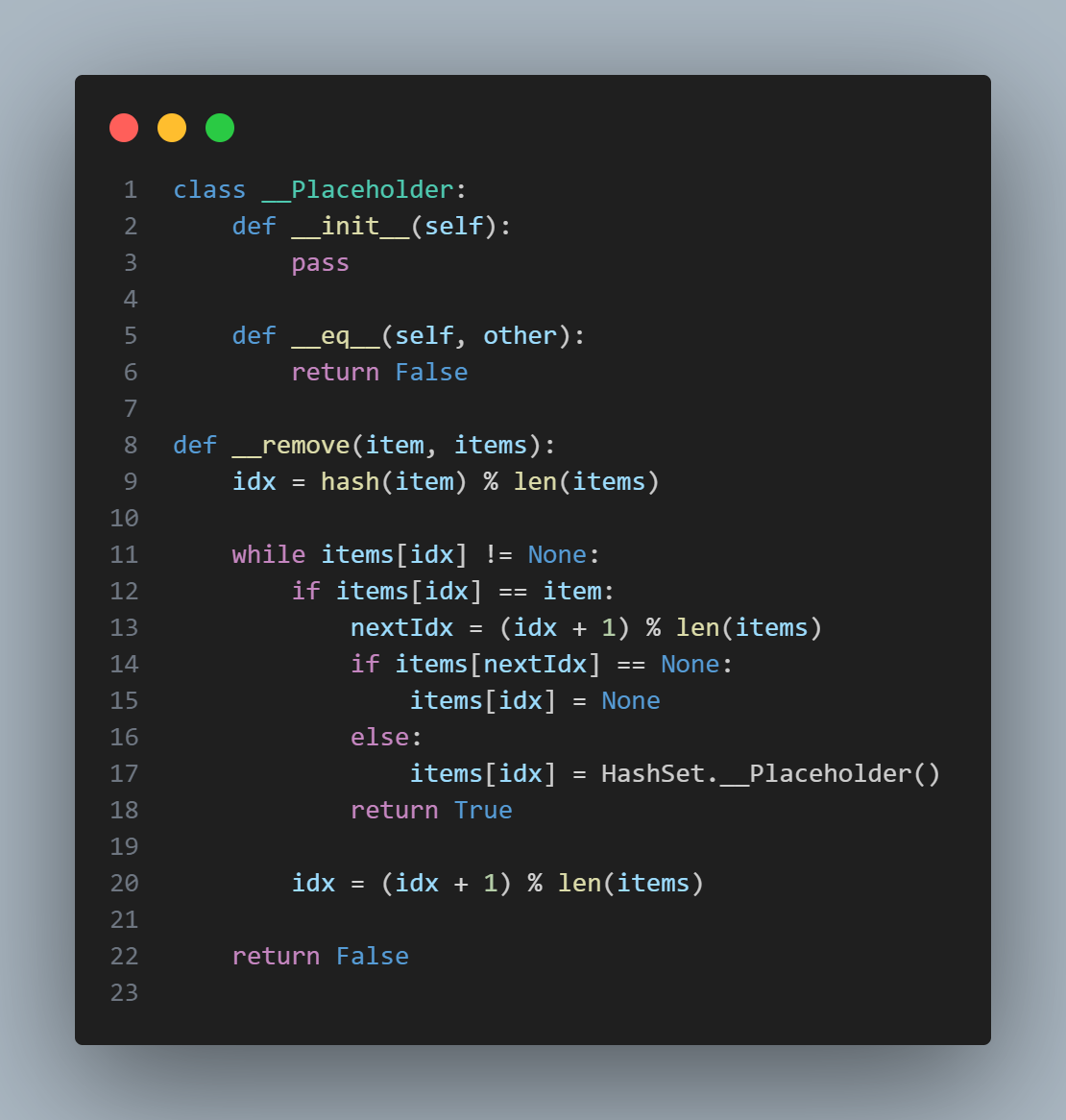
**5.5.7 Bir Öğeyi Silme**

Bir hash kümesinden bir değeri silmek, öncelikle öğeyi bulmayı gerektirir. Bu, listedeki belirli bir konumda bulunan değerler zincirinde **doğrusal bir arama** yapmayı içerebilir. Eğer silinecek değer zincirdeki **son öğeyse**, bir None ile değiştirilebilir. Ancak, öğe zincirin **ortasında** bulunuyorsa, onu None ile değiştiremeyiz çünkü bu, zinciri keserek bağlantıyı koparır. Bunun yerine, öğe bir \_\_Placeholder nesnesiyle değiştirilir.

Bir **yer tutucu** nesne zinciri bozmaz ve doğrusal adresleme gerektiğinde, arama işlemi **yer tutucu nesneleri atlayarak** devam eder. **remove** yardımcı fonksiyonu, Bölüm 5.5.8'de verilmiştir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.8 HashSet Kaldırma Yardımcı Fonksiyonu**



Bir öğeyi kaldırırken, yük faktörü çok düşebilir ve bellekte alan kullanımının verimli olmamasına neden olabilir. **Yük faktörü %25’in altına düştüğünde**, yük faktörünü artırmak amacıyla **liste yeniden rehash edilir** ve boyutu **yarıya indirilir**.

**remove** yöntemi, Bölüm 5.5.9’da verilmiştir.

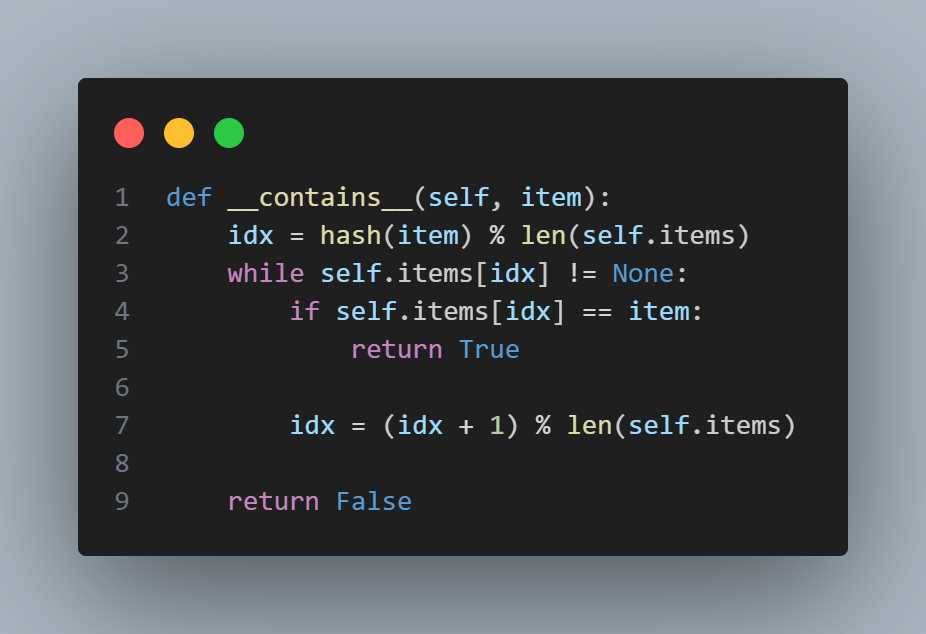
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.9 HashSet Kaldır**



Bir değeri **O(1) zamanda** ekleyebilme sebebiyle, bir değerin silinmesi de **O(1) amortize karmaşıklık** ile gerçekleştirilebilir. **discard** yöntemi, Bölüm 5.5.9’da sunulan **remove** yöntemiyle neredeyse aynıdır, ancak **öğe kümede yoksa herhangi bir istisna fırlatılmaz**.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.10 Bir Oğe Bulma**

Bir hash kümesindeki bir öğeyi bulmak, öğenin adresini bulmak üzere **hashlenmesini** ve ardından olası değerler zincirinin **arama yapılmasını** gerektirir. Zincir, bir **None** ile sonlanır. Öğe zincirde bir yerdeyse, **contains** yöntemi **True** döndürecektir; aksi takdirde **False** döndürülür. Kümedeki öğe bir programa yazıldığında, Bölüm 5.5.11'deki yöntem çağrılır.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.11 HashSet Üyelik**



Bir öğeyi bulmak da O(1) amortize karmaşıklık ile sonuçlanır. Zincirler, çoğu hash değeri eşit olarak dağıtıldığı ve yük faktörü 1’e yaklaşmadığı sürece kısa tutulur.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.12 Bir Küme Üzerinde Yineleme**

Bir kümenin öğeleri üzerinde yineleme yapmak için, HashSet’in öğelerini **yield** ederek döndüren **iter** yöntemini tanımlamamız gerekir. Yöntem, listeyi gezerek yer tutucu öğeleri ve None referanslarını atlar. İşte   
yineleyici için kod.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.13 Diğer Set İşlemleri**

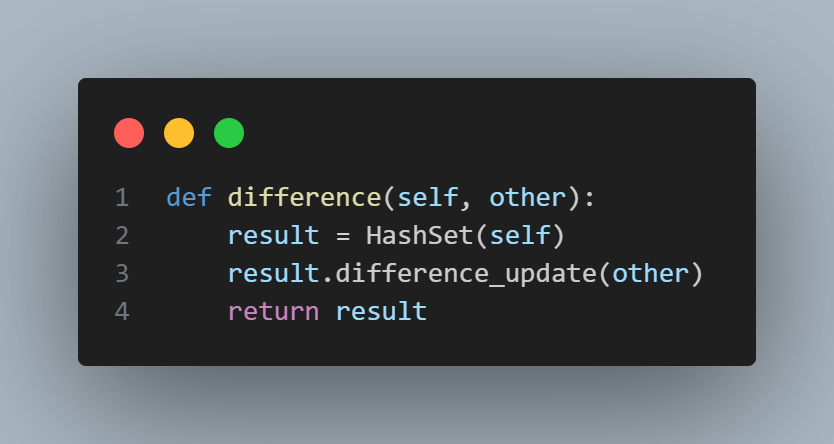
HashSet üzerindeki diğer küme işlemlerinin birçoğu okuyucu için bir alıştırma olarak bırakılmıştır. Ancak, bunların çoğu bu bölümde daha önce sunulan yöntemler açısından uygulanabilir. difference\_update yöntemini ele alalım. Yineleyici, üyelik testi ve ıskarta yöntemi kullanılarak uygulanabilir. Bölüm 5.5.14'teki kod, difference\_update yöntemi için bir uygulama sunmaktadır

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.14 HashSet Fark Güncellemesi**



Bölüm 5.5.14'te sunulan difference\_update yöntemi, self tarafından referans verilen diziyi değiştirdiği için bir mutatör yöntemidir. Bunu, self tarafından referans verilen nesneyi değiştirmeyen Bölüm 5.5.15'teki difference yöntemi ile karşılaştırın. Bunun yerine, fark yöntemi self ve diğer kümenin farkından oluşan yeni bir küme döndürür.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.15 HashSet Fark**

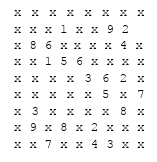


Fark yöntemi, sonuç HashSet üzerinde difference\_update yöntemi kullanılarak uygulanır. Yeni bir kümenin döndürüldüğüne dikkat edin. self tarafından başvurulan hash kümesi güncellenmez. Kod basittir ve difference\_update doğru yazılırsa bu yöntemin de doğru yazılacağı gibi bir avantajı vardır. Programcılar mümkün olduğunca yinelenen kodlar yazmaktan kaçınmalıdır. difference ve difference\_update yöntemleri, difference yönteminin self'in referans aldığı küme yerine yeni oluşturulan bir küme üzerinde fark işlemi gerçekleştirmesi dışında neredeyse aynıdır.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.6 Sudoku Çözme**

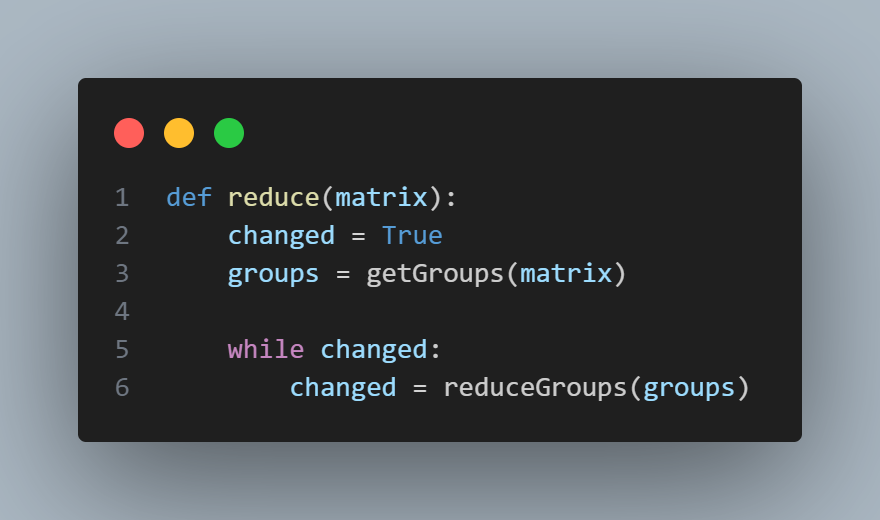
Bir küme veya HashSet veri türü kullanarak, artık çoğu Sudoku bulmacasını çözmek için gerekli araçlara sahibiz. Bir bulmaca, bilinen değerlerin rakamlarıyla temsil edildiği ve bilinmeyen değerlerin aşağıdaki bulmacada olduğu gibi X'ler olduğu bir dosyadan okunabilir.



Dateable'ı okumak her seferinde bir satır yapılabilir. Satırı bölmek, bilinen ve bilinmeyen her değeri listede ayrı bir öğe olarak içeren bir dize sağlayacaktır. Bir X ile karşılaşıldığında, tıpkı Sudoku'yu elle çözerken yaptığınız gibi, 1-9 arası tüm değerleri içeren bir küme oluşturulabilir. Bilinen bir değer bulunduğunda, içinde bilinen sayının bulunduğu bir küme oluşturulabilir. Kümeler iki boyutlu bir matrise eklenir. Matris, listelerden oluşan bir listedir. Yani bir satırı okumak matrisin bir satırını okumaya karşılık gelir. Her satır bir kümeler listesi haline gelir. Her set listesi matris adını vereceğimiz bir listeye eklenir. Yani, matrix[row][col] Sudoku bulmacası içindeki bir settir. Bir Sudoku bulmacasında 81 küme vardır. Ancak her küme üç grubun üyesidir: satır, sütun ve içinde bulunduğu kare. Yukarıdaki Sudoku bulmacası açıklamasında sunulan kuralların her biri bu gruplardan biri içindeki bir kümeyi indirgemekle ilgilidir. Dosyadan girdiyi okuduktan ve 81 set ile matrisi oluşturduktan sonra, bir grup listesi oluşturularak 27 grup oluşturulur. Her satırın yüzeysel bir kopyası ilk olarak gruplar listesine eklenir. Sığ bir kopya, kümelerin her birinin bulmaca okunduğunda oluşturulan kümeyle aynı olduğu anlamına gelir.

Derin bir kopya 81 kümenin her birinin bir kopyasını oluşturur. Bir listenin sığ kopyası, liste içindeki kümeleri kopyalamaz. Bir liste üzerinde list çağrısı yapmak sığ bir kopya oluşturacaktır. Her sütun için başka bir grup oluşturulur ve bu gruplar gruplar listesine eklenir. Son olarak, her kare için bir grup oluşturulur ve bu gruplar gruplar listesine eklenir. Hepsi tamamlandığında, içinde 27 grup bulunan bir gruplar listesi oluşur. Bu grupları oluştururken, üç grubun her birinde aynı kümenin görünmesi çok önemlidir. Bunun nedeni, bir satır küçültüldüğünde, bu satırdaki değişikliklerin satırın elemanlarının göründüğü sütunlara ve karelere de yansımasını istememizdir. Bir Sudoku bulmacasını çözmek, Bölüm 5.2'de sunulan iki kurala göre bir grubun her bir kümesindeki öğe sayısını azaltmak anlamına gelir. İndirgenecek 9 kümelik bir liste verilen reduceGroup adında bir fonksiyon yazmak yardımcı olabilir. reduceGroup fonksiyonu, grubu azaltabildiyse True, azaltamadıysa False döndürmelidir. Bu reduceGroup fonksiyonu verildiğinde, reduce fonksiyonu Bölüm 5.6.1'de tanımlandığı gibidir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.6.1 Sudoku Azaltma Fonksiyonu**



Bu algoritma oldukça basit ve yine çok güçlüdür. Şimdiye kadar bu metinde sunulan diğer algoritmalardan farklıdır. Kavram oldukça basittir: Daha fazla indirgeme mümkün olmayana kadar indirgemeye devam edin. Algoritmanın her iterasyonu, bulmacanın bazı kümelerindeki öğe sayısını azalttığı için, sonlanacağı garanti edilir. Bu kümelerin boyutunu asla artırmayız. Bu fonksiyondan döndüğümüzde, bir çözümümüz olup olmadığı belirsizdir. Bu Sudoku çözümleyicisiyle çözülemeyen bazı bulmacalar vardır çünkü yukarıda sunulan iki kural, tüm bulmacalar için yeterince güçlü değildir. Bazı durumlarda, bir kümedeki öğe sayısı sadece bir gruba bakarak indirgenemez. Öğelerin sayısını azaltmak için birden fazla gruba aynı anda bakmak gerekebilir. Ama durun, başka kurallar bulmanıza gerek yok. Bir sonraki bölüm, tüm Sudoku bulmacalarını, en zor olanları bile çözmek için bir algoritma sunacaktır.

Bu bölümde sunulan kurallar, bu bölümde verilen Sudoku bulmacasını ve daha birçok bulmacayı çözecektir. Metnin web sitesindeki birden altıya kadar olan Sudoku bulmacaları, bu Sudoku çözümleyicisiyle çözülebilir. Yukarıdaki reduceGroups fonksiyonu, muhtemelen listesinde bulunan her bir grup için reduceGroup fonksiyonunu çağırır ve gruplardan herhangi biri indirgenirse True, aksi takdirde False döndürür.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.7 Haritalar**

Bilgisayar bilimlerinde harita, arabanızla bir yere giderken kullandığınız harita gibi değildir. Harita terimi, bir alanı bir aralığa eşleyen bir fonksiyona atıfta bulunan daha matematiksel bir terimdir. Python'da zaten bir harita kullanmış olabilirsiniz. Haritalar, sözlükler, hash tabloları ve hash haritaları gibi birçok isimle anılır. Hepsi aynı veri yapısıdır.

Bir harita veya sözlük, bir fonksiyonun etki alanındaki bir değeri aralığa eşlemesi gibi, bir dizi benzersiz anahtarı ilişkili değerlerle eşler. Anahtar, anahtar/değer çiftini aramak istediğimizde bir haritaya sağladığımız şeydir. Bir haritanın anahtarları benzersizdir. Bir sözlükte, belirli bir anahtarın sadece bir kopyası olabilir. Birinci bölümde gördüğümüz gibi, Python, sözlükler veya haritalar için yerleşik destek sunar. İşte Python kabuğunda bir sözlükle yapılan bazı örnek etkileşimler.



Bir harita veya sözlük, bir kümeye çok benzer. Küme ve sözlüğün her ikisi de benzersiz değerler içerir. Küme veri türü, bir grup benzersiz değer içerir. Bir harita ise ilişkili değerlerle eşlenen benzersiz anahtarlar kümesi içerir. Kümelerde olduğu gibi, haritadaki bir anahtarı ve ilişkili değerini O(1) sürede arayabiliriz. Tahmin edebileceğiniz gibi, kümeler gibi haritalar da hashing kullanılarak uygulanır. Temel uygulama aynı olsa da, haritalar ve kümeler farklı şekilde kullanılır. Aşağıdaki tabloda, haritaların veya sözlüklerin yöntemleri ve operatörleri ile bunlara karşılık gelen karmaşıklıkları verilmiştir. Yukarıdaki tabloda yer alan işlemler, Bölüm 5.5'te sunulduğu gibi bir hash uygulaması göz önüne alındığında beklenen karmaşıklıklara sahiptir. İlginç olan fark, sözlükte sadece bir kümenin öğeleri yerine anahtar/değer çiftlerinin saklanmasıdır. Anahtar/değer çiftinin anahtar kısmı, beklediğiniz gibi bir anahtarın sözlükte olup olmadığını belirlemek için kullanılır. Uygun olduğunda değer döndürülür.

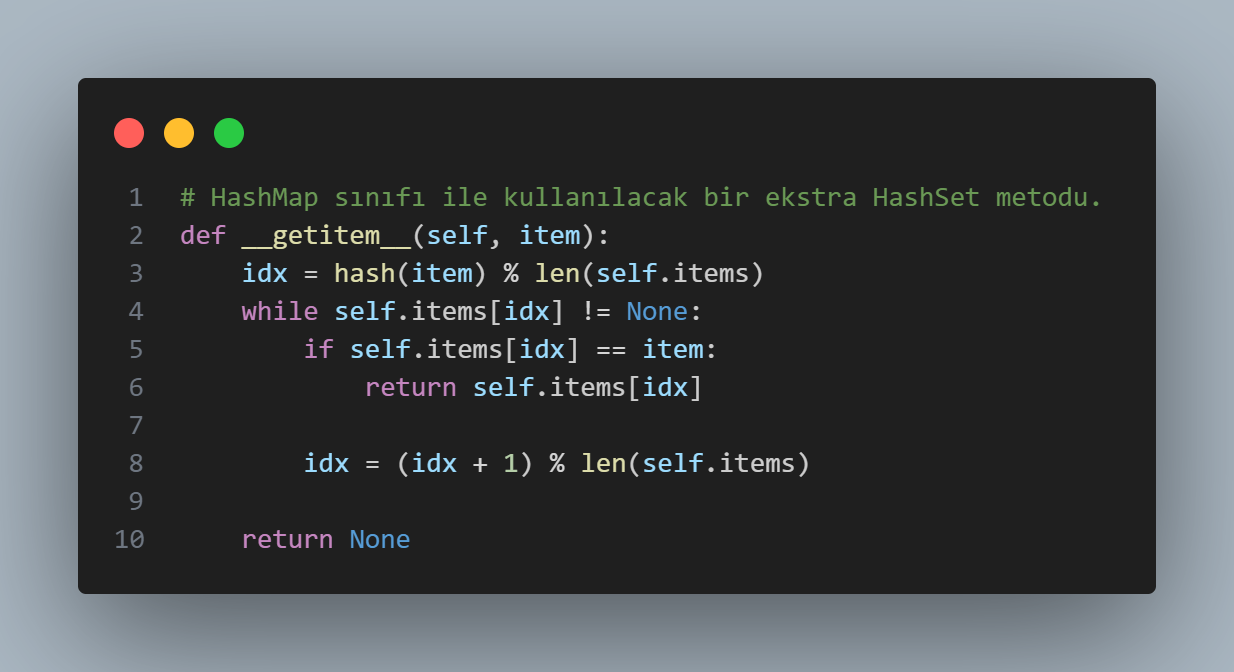
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

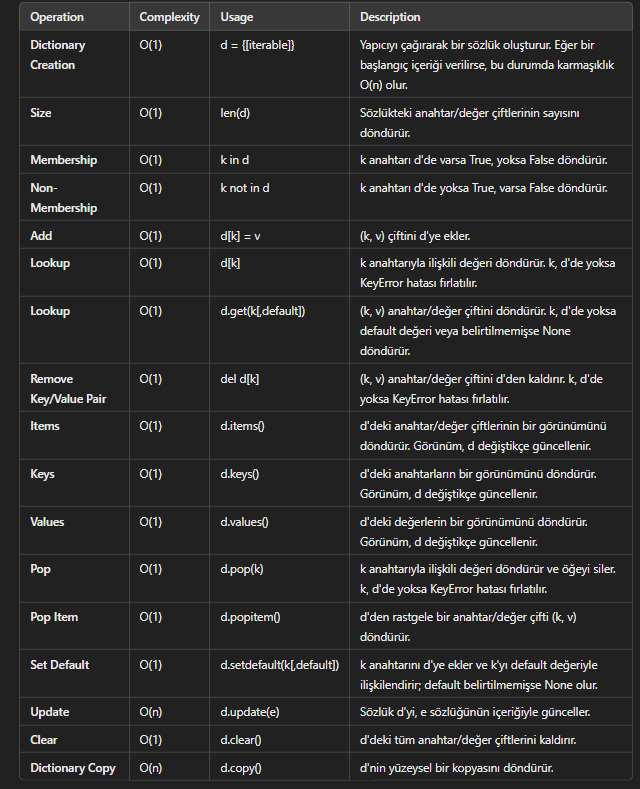
**5.7.1 HashMap Sınıfı**

Python'daki dict sınıfı gibi bir HashMap sınıfı, Şekil 5.6'daki tabloda özetlenen karmaşıklıkları elde etmek için hashing kullanır. Özel bir \_\_KVPair sınıfı tanımlanmıştır. \_\_KVPair örnekleri, HashMap nesnesine eklendikçe anahtar/değer çiftlerini tutar. HashSet sınıfına bir **getitem** yöntemi eklenerek, HashSet sınıfı HashMap sınıfı uygulaması için kullanılabilir. HashSet için ek **getitem** yöntemi Bölüm 5.7.3'te verilmiştir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.7.2 HashSet Öğeyi Al**





Daha sonra, HashMap'i uygulamak için Bölüm 5.7.3'te gösterildiği gibi bir HashSet kullanabiliriz. \_\_KVPair sınıf tanımında, hash haritasındaki iki öğeyi karşılaştırırken anahtarların karşılaştırılması için **eq** yöntemini tanımlamak gerekir. \_\_KVPair'in **hash** yöntemi yalnızca anahtar değerini hash eder, çünkü anahtarlar hash haritasında anahtar/değer çiftlerini aramak için kullanılır. Bölüm 5.7.3'te verilen uygulama kısmi bir uygulamadır. Diğer yöntemler okuyucu için bir alıştırma olarak bırakılmıştır.