Bilgisayar Bilimcisi gibi Düşünmek

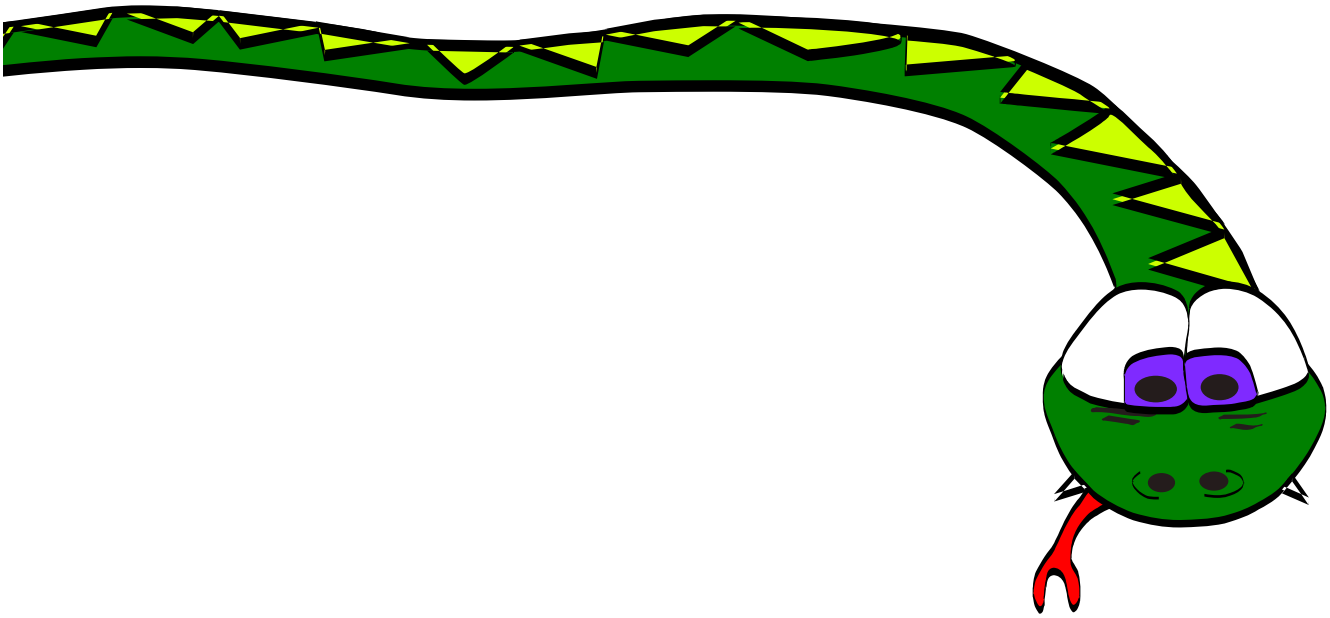
Python ile Öğrenme 2. Baskı

Yazarlar: Jeffrey Elkner, Allen B. Downey ve Chris Meyers

Görseller: Udit Bhatnager ve Chris Schmeelk

Çeviren: **Tahir Emre Kalaycı**

İngilizce Aslı İçin



Anasayfa David Cox tarafından tasarlanmıştır.

İçindekiler

1. Programlama Yolu

Bu kitabın amacı size bir bilgisayar bilimcisi gibi düşünmeyi öğretmektir. Bu düşünme şekli matematiğin, mühendisliğin ve doğal bilimlerin bazı en iyi özelliklerini birleştirir. Matematikçiler gibi, bilgisayar bilimcileri fikirleri (özellikle hesaplamalar) göstermek için biçimsel dilleri kullanırlar. Mühendisler gibi şeyleri tasarlar, bileşenleri (parçaları) sistemler şeklinde birleştirir ve farklı alternatiflerin avantaj ve dezavantajlarını değerlendirirler. Bilim insanları gibi, karmaşık sistemlerin davranışlarını gözlemler, hipotezler oluşturur ve varsayımları (tahminleri) sınarlar.

Bir bilgisayar bilimcisi için tek ve en önemli yetenek **problem çözme**dir. Problem çözme problemleri formüle edebilme, çözümler hakkında yaratıcı düşünme ve bir çözümü açık ve kesin olarak ifade edebilme yeteneğidir. Görüleceği gibi, programlama öğrenme süreci problem çözme yeteneklerinin uygulamasını yapmak için mükemmel bir fırsattır. Bu bölüme bu yüzden, "Programlama yolu" adı verilmiştir.

Bir yönden programlamayı öğrenirken - ki kendisi yararlı bir yetenektir - bir başka yönden programlamayı sonuç için bir yol olarak kullanmayı öğreneceksiniz. İlerledikçe bu daha açık bir hale gelecektir.

1.1 Python programlama dili

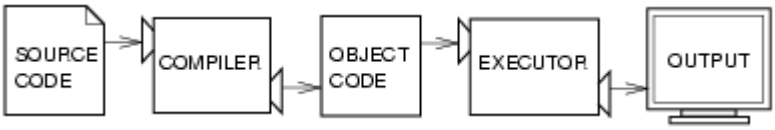
Öğreneceğiniz programlama dili Python olacaktır. Python bir **yüksek seviyeli dil** örneğidir; yüksek seviyeli dillere örnek olarak daha önce duymuş olabileceğiniz C++, PHP ve Java verilebilir.

"Yüksek seviyeli dil" ifadesinden çıkarabileceğiniz gibi, ayrıca **düşük seviyeli dil**ler de vardır, bunlar bazı durumlarda "makine dili" veya "birleştirici dili" şeklinde isimlendirilir. Basitçe söylersek, bilgisayarlar sadece düşük seviyeli dillerde yazılmış programları çalıştırabilirler. Bu yüzden, yüksek seviyeli dillerde yazılmış programlar çalıştırılmadan önce bir işlemden geçmelidir. Bu ek işlem biraz zaman alır, bu da yüksek seviyeli dillerin dezavantajıdır.

Ancak avantajları oldukça fazladır. İlk olarak, yüksek seviyeli dillerde programlamak oldukça kolaydır. Yüksek seviyeli dilde yazılmış programlar daha az sürede yazılır, daha kısadır, okuması daha kolay ve doğru olma ihtimalleri daha yüksektir. İkinci olarak, **yüksek seviyeli diller**dir, bunun anlamı farklı bilgisayarlarda az değişiklik veya değişiklik yapmadan çalıştırılabilirler. Düşük seviyeli programlar sadece tek bir bilgisayar çeşidinde çalışabilirler ve başka bir bilgisayarda çalışabilmesi için tekrar yazılmalıdır.

Yüksek seviyeli dilleri, düşük seviyeli dillere çevirmek için iki çeşit program kullanılmaktadır: **yorumlayıcılar** ve **derleyiciler**.Yorumlayıcılar yüksek seviyeli programı okur ve işletir, bunun anlamı program ne diyorsa onu yapar. Programı biraz biraz işler, satırları okur ve ilgili hesaplamaları gerekleştirir.

Derleyici programı okur ve programı çalıştırmadan önce tamamen başka bir şeye dönüştürür. Bu durumda yüksek seviyeli programa **kaynak kod**, çevrildiği programa ise **hedef kod** veya **çalıştırılabilir** denir. Program derlendikten sonra, daha fazla çevirme yapmadan tekrar tekrar çalıştırılabilir.



Çoğu modern dil her iki işlemi de kullanır. Programlar ilk önce daha düşük seviyeli bir dile derlenir, **byte kod**, ve daha sonra **sanal makine** adı verilen bir program tarafından yorumlanırlar. Python her iki işlemi de kullanır, ancak programcıların kendisiyle etkileşim yöntemi nedeniyle, daha çok **yorumlamalı dil** olarak kabul edilir.

Python yorumlayıcısının kullandığı iki yöntem vardır: *kabuk modu* ve *betik modu*. Kabuk modunda **Python kabuğu**na Python cümlelerini yazıp, yorumlayıcının hemen sonuçları üretmesi sağlanıyor:

```
$ python
Python 2.5.1 (r251:54863, May  2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print 1 + 1
2
```

Bu örnekteki ilk satır, bir Unix komut satırında Python yorumlayıcısını başlatan komuttur. Sonraki satır yorumlayıcının mesajlarıdır. >>> ile başlayan dördüncü satır, **Python bilgi istemi**(prompt)dir. Yorumlayıcı bilgi istemini komutlar için hazır olduğunu belirtmek için kullanmaktadır.print 1+1 yazdık ve yorumlayıcı yanıt olarak 2 döndürdü.

Alternatif olarak, programı bir dosya içine yazıp yorumlayıcının dosya içeriğini çalıştırmasını sağlayabiliriz. Bu tip dosyaya **betik** adı verilmektedir. Örneğin, bir metin dzenleyicisi yardımıyla firstprogram.py adlı dosyayı aşağıdaki içerikle yarattığımızı varsayalım:

```
print 1 + 1
```

Bir kural (gelenek) olarak, Python programlarını içeren dosyalar .py uzantısıyla biter.

Programı çalıştırmak için, yorumlayıcıya dosyanın ismini vermemiz gerekir:

```
$ python firstprogram.py
2
```

Bu örnekler Python'un Unix komut satırından çalıştırılmasını göstermektedir. Diğer geliştirme ortamlarında, programları çalıştırma ayrıntıları farklılıklar gösterebilir. Ayrıca çoğu program bu verilen örnekten çok daha ilginçtir.

Bu kitaptaki örnekler Python yorumlayıcısı ve betikleri kullanır. Kabuk modu örnekleri hep bilgi istemiyle başlayacağı için örneklerin alıştırılmasında hangi yöntemin kullanılacağını rahatlıkla ayırdedebileceksiniz. Kabuk modunda çalışmak küçük kod parçalarını sınamak için geleneksel yöntemdir, çünkü anında geri beslemeyi almış olursunuz. Kabuk modunu karalama için kullanılan bir kağıt parçası gibi düşünebilirsiniz. Bir kaç satırdan uzun herşey betik içerisine koyulmalıdır.

1.2 Program nedir?

Program, hesaplamayı gerekleştirmek için gereken birbirini izleyen yönergelerden (komutlardan) oluşan yapıdır. Hesaplama matematiksel olabilir, örneğin bir eşitlikler sisteminin çözülmesi veya bir polinomun kökünü bulmak gibi, ama ayrıca sembolik bir hesaplama, bir belge içinde metin arama ve değiştirme veya program derleme (yeterince ilginç) gibi, olabilir.

Ayrıntılar farklı programlama dillerinde farklı gözükebilir, ancak bazı basit temel yönergeler her dilde gözükür:

girdi:
: Klavyeden, dosyadan veya başka bir aygıttan veriyi alma.

çıktı:
: Ekranda veriyi görüntüleme veya veriyi bir dosya ya da başka bir aygıta gönderme.

matematik:
: Toplama, çarpma gibi bazı temel matematiksel işlemleri gerçekleştirme.

koşullu yürütme:
: Belirli durumlar için sınama yapma ve uygun cümle sırasını çalıştırma.

tekrarlama:
: Bazı eylemleri genellikle ufak tefek bazı değişikliklerle tekrar tekrar yürütme.

İnanın veya inanmayın, var olanların hepsi bu kadardır. Şu ana kadar kullandığınız her program, ne kadar karmaşık olursa olsun, yukarıdakilere benzeyen komutlarla yapılmıştır. Bu yüzden, programlamayı büyük ve karmaşık bir görevi, yukarıdaki temel komutlarla gerçekleştirilebilecek kadar basit olan daha küçük alt görevler şeklinde parçalama süreci olarak tanımlayabiliriz.

Bu biraz belirsiz gelebilir, ancak bu konuya **algoritmalar** hakkında konuşurken tekrar geleceğiz.

1.3 Hata ayıklama (Debugging) nedir?

Programlama karmaşık bir süreçtir, ve insanlar tarafından yapıldığı için hatalara yol açabilir. Garip nedenlerden dolayı, programlama hatalarına bug adı

verilmektedir, ve bu hataları belirleme ve düzeltme işlemine **debugging** adı verilmektedir (çn. Bug böcek anlamına gelir. Eski dönemlerde odayı kaplayan bilgisayarlarda böcek görülmesi üzerine kullanıldığına dair iddialar var. Türkçe'de hata ayıklama ifadesini kullanacağımız için aslında bu açıklamanın sadece ingilizce için geçerli olduğunu belirtmek gerekiyor).

Bir programda üç tür hata oluşabilir : sözdizimsel hatalar, çalışma zamanı hataları ve anlambilimsel hatalar. Bu hataları daha hızlı belirleyebilmek için ayırt edici özelliklerinin anlatılması önemlidir.

1.4 Sözdizimsel hatalar

Python bir programı eğer sözdizimsel olarak doğruysa çalıştırabilir; aksi halde, işlem başarısız olur ve bir hata mesajı döndürür. Sözdizimi programın yapısını kasteder ve bu yapı hakkındaki kuralları tanımlar. Örneğin, Türkçe'de bir cümle büyük harfle başlamalı ve nokta işaretiyle tamamlanmalıdır. bu cümle bir **sözdizimi hatası** içermektedir. Ayrıca bu cümle de

Çoğu okuyucu için, bazı sözdizimi hataları çok önemli değildir, bu yüzden bazı şiir örneklerini hata mesajları vermeden okuyabiliriz. Python bu kadar affedici değildir. Programın herhangi bir yerinde tek bir sözdizimi hatası olduğunda bile, bir hata mesajı verip çıkacaktır ve programı çalıştırmanız mümkün olmayacaktır. Programlama kariyerinizin ilk bir kaç haftasında, bu sözdizimi hatalarını belirlemek ve düzeltmek için oldukça fazla zaman harcayacaksınız. Deneyim kazandıkça, daha az hata yapmaya ve hataları daha hızlı belirleyip düzeltmeye başlayacaksınız.

1.5 Çalışma zamanı hataları

İkinci tür hatalar, **çalışma zamanı hataları**dır, bu ismin verilmesinin nedeni program çalıştırılana kadar bu hataların ortaya çıkmamasıdır. Bu hatalara ayrıca **istisna** adı da verilmektedir, çünkü istisnai (ve kötü) bir durumun ortaya çıktığını belirtirler. ilk bölümlerdeki basit programlarda çalışma zamanı hatalarıyla karşılaşmanız oldukça düşük ihtimallidir, bu hatalarla karşılaşana kadar biraz zaman geçebilir.

1.6 Anlambilimsel hatalar

Üçüncü hata tipi **anlambilimsel hata**lardır. Programda bir anlambilimsel hata varsa, başarılı bir şekilde çalışmayacaktır, bundan kastedilen bilgisayarın herhangi bir hata mesajı üretmeyeceği, ancak doğru şeyi yapmayacağıdır. Beklenilenden farklı bir şey yapacaktır. Bilgisayarlar siz ona ne derseniz, onu yapan aygıtlardır.

Sorun, yazdığınız program yazmak istediğiniz program değildir. Programın anlamı (anlambilimi) hatalıdır. Anlambilimsel hataları tanımlamak ustalık (beceri) ister, çünkü programın çıktısını inceleyerek geriye yönelik olarak takip etmenizi ve ne yaptığını anlamaya çalışmanızı gerektirir.

1.7 Deneysel hata ayıklama

Kazanabileceğiniz en önemli yeteneklerden biri hata ayıklamadır. Sinir bozucu olmasına rağmen, hata ayıklama programlamanın zihinsel zenginlik gerektiren, zorlayıcı ve ilginç bir parçasıdır.

Bazı durumlarda, hata ayıklama dedektif çalışması gibidir. İpuçlarıyla karşı karşıya kalmışsınızdır ve gördüğünüz sonuçlara yol açan süreçlerle eylemlerden sonuçlar ve anlamlar çıkarmanız gerekir.

Hata ayıklama deneysel bilim gibidir. Neyin hatalı gittiğine dair fikriniz oluştuğunda, programı değiştirerek tekrar tekrar denersiniz. Hipoteziniz doğru ise, değişikliğin sonucunu tahminleyebilir ve çalışan bir programa bir adım daha yaklaşmış olursunuz. Hipoteziniz yanlış ise, yeni bir hipotez üretmeniz gerekir. Sherlock Holmes'un söylediği gibi "imkansızı elediğinizde, elinizde kalan şey, her ne kadar görünmese de gerçek olmak zorundadır" (A. Conan Doyle, The Sign of Four)

Bazı kişiler için, programlama ve hata ayıklama aynı şeydir, programlama istediğiniz işi yapana kadar bir programda hata ayıklama sürecidir. Bu fikir programı yazmaya basit bir şekilde, temel bir şeyler yapan bir örnekle başlamanız gerektiğini söyler, ilerledikçe programda küçük değişiklikler yaparak, hataları ayıklayarak ilerlemenizi tavsiye eder. Bu yöntemle her zaman bir çalışan sürüme sahip olabilirsiniz.

Örneğin, Linux bir işletim sistemi çekirdeğidir ve binlerce satır kaynak kod içerir, ama Linus Torvalds tarafından Intel 80386 yongasını keşfetmek için basit bir program olarak başlatılmıştır. Larry Greenfield'a göre, "Linus'un erken projelerinden biri, AAAA ve BBBB arasında seçim yaparak ekrana yazdıran bir programdır, bu program daha sonra Linux'e evrilmiştir" ("The Linux Users' Guide version 1")

Kitabın sonraki bölümleri hata ayıklama ve diğer programlama pratikleri hakkında daha fazla önerilerde bulunacaktır.

1.8 Biçimsel (Formal) ve doğal diller

Doğal diller konuştuğumuz dillerdir, Türkçe, İngilizce, İspanyolca ve Fransızca gibi. İnsanlar tarafından tasarlanmamıştır (her ne kadar insanlar bunlar için kurallar koymaya çalışsa da), doğal olarak evrilmişlerdir.

Biçimsel diller insanlar tarafından bazı özel uygulamalar için geliştirilmiş dillerdir. Örneğin, matematikçilerin kullandığı gösterim sayılar ve semboller arasındaki ilişkiyi göstermek için iyi bir biçimsel dildir. Kimyacılar moleküllerin kimyasal yapısını göstermek için biçimsel bir dil kullanırlar. Ve en önemlisi:

> Programlama dilleri hesaplamaları ifade etmek için tasarlanmış biçimsel dillerdir.

Biçimsel diller sözdizimi açısından katı kurallara sahip olma eğilimindedir. rneğin, 3+3=6 ifadesi matematiksel olarak sözdizimsel olarak doğru bir ifadedir, ancak 3=+6$ değildir. H_2O sözdizimsel olarak doğru bir kimyasal isimdir, $_2Zz$ ifadesi değildir.

Sözdizimsel kurallar **token**lere ve yapıya ait olmak zere iki şekildedir. Tokenler dilin temel ğeleridir, kelimeler, sayılar ve kimyasal elementler gibi. 3=+6$ ifadesiyle ilgili bir sorun $ işaretinin matematikte (bildiğimiz kadarıyla) geçerli bir token olmamasıdır. Benzer olarak $_2Zz$ ifadesi de geçerli değildir çünkü Zz kısaltmasına sahip bir element yoktur.

İkinci tür sözdizim kuralı cümlelerin yapısıyla - tokenlerin nasıl dizildiği - ilgili kurallardır. 3=+6$ yapısal olarak geçersizdir çünkü eşittir işaretinden hemen sonra artı işareti gelemez. Benzer olarak, molekül formüllerinde de alt işaretler element isminden sonra gelmelidir, önce gelemezler.

Biçimsel bir dildeki veya Türkçe'deki bir cümleyi okuduğunuzda, cümlenin yapısını çözmek gerekir (doğal dillerde bunu bilinçsiz bir şekilde yaparız). Bu sürece **ayrıştırma** (parsing) adı verilmektedir.

Örneğin, "Ayaklarıma kara sular indi" cümlesini duyduğunuzda, "ayaklarıma"nın özne, "indi"'nin de yüklem olduğunu algılarsınız. Cümleyi ayrıştırdığınızda, ne anlama geldiğini (cümlenin anlambilimini) çözersiniz. "Ayak"ın ne olduğunu ve "indi" eylemini bildiğinizi varsayarsak, bu cümleyle kastedilen anlamı ortaya çıkarabilirsiniz.

Her ne kadar biçimsel ve doğal diller bir çok ortak özelliğe - tokenler, yapı, sözdizimi, anlambilim - sahip olsa da, bir çok farklılıkları da vardır:

belirsizlik:
: Doğal diller belirsizlikle doludur, kişiler bağlamsal ipuçları ve diğer bilgilerden yararlanarak bu belirsizliği aşarlar. Biçimsel diller neredeyse veya tamamen belirli olmak üzere tasarlanmıştır. Bunun anlamı cümlenin sadece bir anlamı vardır, bağlam ne olursa olsun.

fazlalık (redundancy):
: Belirsizliği önlemek ve yanlış anlamaları azaltmak için, doğal diller bir çok gereksiz içeriğe sahiptir. Bu yüzden bir çok fazlalık barındırır. Biçimsel diller fazlalık içermez, az ve öz olmalıdır.

gerçekçilik:
: Doğal diller deyim ve mecazlarla doludur. Eğer biri "Ayaklarıma kara sular indi" diyorsa, bu ayaklarına kara sular indi anlamına gelmez, gizli bir anlamı vardır ve yorgun olduğunu belirtir. Biçimsel dillerde cümlelerin gerçeği aynen yansıtması gerekir, anlamı yazılanla aynı olmalıdır.

Doğal bir dili konuşarak büyüyen kişiler - herkes - biçimsel dillere alışmakta zorlanabilirler. Bazı durumlarda, biçimsel dillerle doğal diller arasındaki farklar dzyazı ile şiir arasındaki farklar gibidir, ancak;

Şiir:
Kelimeler anlamları olduğu kadar sesleri için de kullanılır, ve tüm şiir bir etki veya duygusal bir tepki yaratır. Belirsizlik yaygın olmasının yanında sıklıkla bir gerekliliktir.

Düzyazı:
Kelimelerin gerçek anlamı daha önemlidir, ve yapı anlama daha fazla katkı sağlar. Düzyazının şiire göre çözümlemesi daha kolaydır ancak yine de belirsizlikler içerebilir.

Programlar:
Bilgisayar programının anlamı belirli (tek anlamlı) ve gerçek olmalıdır, ve token ile yapının çözümlenmesiyle tamamen anlaşılmalıdır.

Programları (ve diğer biçimsel dilleri) okumak için bazı öneriler şu şekildir. İlk olarak, biçimsel dillerin doğal dillerden daha yoğun olduğunu hatırlamak gerekir, böylece onları okumak daha fazla zaman alacaktır. Ayrıca, yapı oldukça önemlidir, bu yüzden yukarıdan aşağıya soldan sağa okumak iyi bir fikir değildir, programı kafanızda ayrıştırmayı, tokenleri ve yapıyı belirleme ve tanımlamayı öğrenmeniz gerekir. Son olarak, ayrıntılar önemlidir. yazım yanlışları ve noktalama hataları gibi küçük şeyler, doğal dillerde bunları yok sayabilirsiniz, biçimsel dillerde büyük farklar ve sorunlar yaratabilir.

1.9 İlk program

Geleneksel olarak, yeni bir dilde yazılan ilk program "Merhaba, Dünya!" adı verilen programdır, çünkü ekranda "Merhaba, Dünya!" kelimelerini gösterir. Python'da aşağıdaki şekilde yazılmaktadır:

```
print "Merhaba, Dünya!"
```

Bu **print cümlesinin** bir örneğidir. Bu cümle kağıt üstüne bir şey yazmaz, ekranda bir değer gösterir. Yukarıdaki örnekte, sonuç aşağıdaki kelimelerin ekranda gözükmesidir:

```
Merhaba, Dünya!
```

Tırnak işaretleri programda bir değerin baş ve sonunu gösterir, ekrandaki sonuçta bu karakterler gözükmez. Bazı kişiler programlama dilinin kalitesini "Merhaba, Dünya!" programının basitliğiyle değerlendirirler. Bu standarta göre, Python oldukça iyi sonuç üretir.

1.10 Sözlük

algoritma:
Bir problem kategorisini çözmek için bir genel süreç

bug:
Programdaki bir hata

byte kod:

Kaynak kod ve hedef kod arasındaki ara dildir. Çoğu modern programlama dili ilk olarak kaynak kodu, byte koda derler ve daha sonra bu byte kodu *sanal makine* adı verilen bir program ile yorumlarlar.

derleme:
Yüksek seviyeli bir dilde yazılmış programı, daha sonra çalıştırmak üzere daha düşük seviyeli bir dile çevirme işlemidir.

hata ayıklama:
Herhangi bir programlama hatasını bulma ve ortadan kaldırma sürecidir.

istisna:
Çalışma zamanı hatasının bir diğer adı

çalıştırılabilir:
Çalıştırılmaya hazır hedef kodun bir diğer adı

biçimsel dil:
Bazı özel amaçlar için , matematiksel fikirlerin temsili veya bilgisayar programları gibi, insanlar tarafından tasarlanmış herhangi bir dildir. Tüm programlama dilleri biçimsel dillerdir.

yüksek seviyeli dil:
İnsanlar tarafından rahatlıkla okunabilmesi, yazılması ve anlaşılması için geliştirilmiş Python benzeri programlama dili

yorumlama:
Yüksek seviyeli dilde yazılmış bir programı satır satır çevirerek yürütme

düşük seviyeli dil:
Bilgisayarın kolay bir şekilde yürütmesi için tasarlanmış programlama dili, "makine dili" veya "birleştirici dil" adı da verilmektedir.

doğal dil:
Doğal olarak evrimleşmiş ve insanların konuşurken kullandığı herhangi bir dil

hedef kod:
Derleyicinin programı (kaynak kodu) çevirmesiyle ortaya çıkan çıktı

ayrıştırma:
Bir programı inceleme ve sözdizimsel yapısını çözümleme işlemi

taşınabilirlik:
Bir programın birden fazla bilgisayar türünde yürütülebilmesi özelliği

print cümlesi:
Python yorumlayıcısının bir değeri ekranda göstermesini sağlayan yönerge (komut) cümlesi

problem çözme:
Problemi formüle etme, çözüm bulma ve çözümü ifade etme süreci

program:
Bilgisayar tarafından gerçekleştirilecek eylemleri ve hesaplamaları belirten art arda gelen komutlar

Python kabuğu:
Python yorumlayıcısının etkileşimli kullanıcı arayüzü. Python kabuğu kullanıcısı bilgi isteminde (>>>) komutları yazar ve Giriş (Enter) tuşuna basarak komutları anında yorumlayıcıya işlenmeleri için gönderir.

çalışma zamanı hatası:

Program çalıştırılana kadar ortaya çıkmayan, çalıştırıldıktan sonra oluşan ve programın çalışmasına devam etmesini engelleyen hata

betik:
Bir dosyada saklanmış program (genellikle yorumlayıcıya verilecek olan)

anlambilimsel hata:
Programın, programcının istediğinden farklı bir şey yapmasına neden olan hata

anlambilim:
Programın anlamı

kaynak kod:
Yüksek seviyeli bir dildeki programın derlemeden önceki hali

sözdizimi:
Programın yapısı

sözdizimi hatası:
Programın ayrıştırılmasını imkansız hale getiren hata (dolayısıyla yorumlanmasını da imkansız hale getirir)

token:
Programın sözdizimsel yapısının temel öğelerinden biri, doğal dillerdeki kelimeye benzetilebilir.

1.11 Alıştırmalar

1. Anlaşılabilir anlama sahip ama sözdizimsel olarak yanlış olan Türkçe bir cümle yazınız. Sözdizimsel olarak doğru ancak anlambilimsel hatalara sahip bir başka cümle yazınız.
2. Python kabuğunu başlatın. 1 + 2 yazıp Giriş tuşuna basın. Python bu *ifadeyi değerlendirir*, sonucu ekranda gösterir ve başka bir bilgi istemi yazar. * *çarpma işleci*dir, ve ** *üs alma işleci*dir. Farklı ifadeler girerek deneyin, Python yorumlayıcısının sonuçlarını kaydedin. */ işleci*ni kullandığınızda ne oluyor? Sonuçlar beklediğiniz gibi mi? Açıklayınız.
3. 1 2 yazıp Giriş tuşuna basın. Python bu ifadeyi değerlendirmeye çalışır, ancak yapamaz çünkü bu ifade sözdizimsel olarak geçerli değildir. Yerine aşağıdaki hata mesajını gösterir:

```
  File "<stdin>", line 1
    1 2
      ^
SyntaxError: invalid syntax
```

Çoğu durumda, Python sözdizimi hatasının nerede olduğunu belirtir, ancak bu her zaman için doğru değildir, ve neyin yanlış olduğuna dair yeterince bilgi vermeyebilir. Bu yüzden, genellikle, sözdizimi kurallarını öğrenmek üzerinizdeki yüktür.

Bu örnekte, Python numaralar arasında işleç olmadığı için şikayet etmektedir.

Python bilgi isteminde yazdığınızda hata mesajına neden olacak üç ifade örneği daha üretiniz. Her ürettiğiniz ifadenin Python sözdizimi açısından neden geçerli olmadığını açıklayınız.

4. print 'merhaba' yazınız. Python bu cümleyi yürütecektir, m-e-r-h-a-b-a harflerini ekranda göstermek gibi bir etkisi olacaktır. Ekranda gösterilen ifadede, cümlede kullandığınız tırnak işaretlerinin olmadığına dikkat edin. Şimdi print '"merhaba"' yazıp, sonucu tanımlayıp, açıklayınız.
5. print peynir ifadesini tırnak işaretleri olmadan yazın. Çıktı aşağıdaki gibi bir şey olacaktır:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'peynir' is not defined
```

Bu bir çalışma zamanı hatasıdır, *NameError* hatasıdır, isimsel bir hatadır çünkü peynir ismi tanımlanmamıştır. Bu ne anlama geliyor bilmiyorsanız, yakında öğreneceksiniz.

6. Python bilgi isteminde 'Bu bir testtir...' yazıp Giriş tuşuna basın. Ne olduğunu kaydedin. Şimdi test1.py isminde bir betik oluşturun ve aşağıdaki içeriği betiğin içine yazın (çalıştırmadan önce kaydetmeyi unutmayın):

```
'Bu bir testtir...'
```

Bu betiği çalıştırdığınızda ne oluyor? Şimdi içeriği aşağıdaki şekilde değiştirin:

```
print 'Bu bir testtir...'
```

ve tekrar çalıştırın. Bu sefer ne oldu?

Ne zaman bir deyim Python bilgi istemine yazılırsa, değerlendirilir ve sonuç ekranda bir satır aşağıya yazılır. 'Bu bir testtir...' bir deyimdir, 'Bu bir testtir...' şeklinde değerlendirilir (42 deyiminin 42 şeklinde değerlendirilmesi gibi). Bir betikte ise, deyimlerin değerlendirilmesi program çıktısına gnderilmez, bu yüzden açıkça ekranda gösterilmesinin gerektiği belirtilmelidir.

2. Değişkenler, deyimler ve cümleler

2.1 Değerler ve tipler

Değer programın işlediği temel şeylerden - harf veya rakam gibi - biridir. Şimdiye kadar gördüğümüz değerler 2 (1 + 1 işleminin sonucu), ve"Merhaba, Dünya!".

Bu değerler farklı **tiplere** aittir: 2 bir tamsayı (integer)dır, ve "Merhaba, Dünya!" bir karakter dizisi (string)dir, çünkü karakterlerden oluşan bir dizidir. Siz (ve

yorumlayıcı) karakter dizilerini ayırtedebilir, çünkü tırnak işaretleri arasında yazılmıştır.

print cümlesi tamsayılar için de çalışır.

```
>>> print 4
4
```

Eğer bir değerin tipinden emin değilseniz, yorumlayıcı bunu size söyleyebilir.

```
>>> type("Merhaba, Dünya!")
<type 'str'>
>>> type(17)
<type 'int'>
```

Sürpriz olmayacak şekilde, karakter dizisi **str** tipine ve tamsayılar da **int** tipine aittir. Ayrıca, ondalık basamağa sahip sayılar **float** adında bir tipe ait olacaktır, çünkü bu sayılar kayan noktalı (floating point) biçiminde temsil edilmektedir.

```
>>> type(3.2)
<type 'float'>
```

Peki "17" ve "3.2" şeklinde ifadelerin tipi nedir? Sayı gibi gözükmektedirler, ancak karakter dizileri gibi tırnak işaretleri arasındalar.

```
>>> type("17")
<type 'str'>
>>> type("3.2")
<type 'str'>
```

Onlar da karakter dizisidir.

Python'da karakter dizileri tek tırnak (') veya çift tırnak (") arasına alınabilir:

```
>>> type('Bu bir karakter dizisidir.')
<type 'str'>
>>> type("Bu da öyle.")
<type 'str'>
```

Çift tırnaklı karakter dizileri "Emre'nin sakalı" örneğinde olduğu gibi tek tırnak, ve tek tırnaklı karakter dizileri '"Ni!" söyleyen şövalyeler'örneğindeki gibi çift tırnak içerebilir.

Büyük bir tamsayı yazdığınızda, üçlü rakam grupları arasında 1,000,000 örneğinde olduğu gibi virgül kullanmak isteyebilirsiniz. Bu Python için geçerli bir tamsayı gösterimi değildir, ancak geçerli bir ifadedir:

```
>>> print 1,000,000
1 0 0
```

Bu beklediğimiz bir sonuç değil! Python 1,000,000 ifadesini üç öğe içeren bir liste şeklinde yorumlar ve o şekilde ekranda görüntüler. Bu yüzden tamsayılarda rakamlar arasında virgül kullanmamayı unutmamalısınız.

2.2 Değişkenler

Programlama dilinin en güçlü özelliklerinden birisi **değişkenleri** değiştirebilme (adından anlaşılacağı gibi) yeteneğidir. Bir değişken bir değeri tutan bir isimdir.

Atama cümlesi yeni bir değişken yaratır ve değerlerini atar:

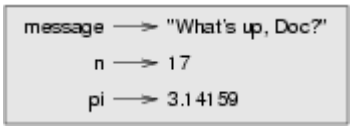
```
>>> mesaj = "Naber, doktor?"
>>> n = 17
>>> pi = 3.14159
```

Bu örnek üç atama gerçekleştirmektedir. İlk atama mesaj isimli yeni yarattığı bir değişkene "Naber, doktor?" karakter dizisini atar. İkinci atama ndeğişkenine, 17 tamsayı değerini ve son atama pi değişkenine 3.14159 kayan noktalı sayı değerini verir.

Atama işleci, =, "eşittir" işaretiyle (aynı karakteri kullansa da) karıştırılmamalıdır. Atama işleçleri isim , işlecin sol tarafı, ve değer , işlecin sağ tarafı, ifadelerini birbirine bağlar. Bu yüzden aşağıdaki ifadeyi yazdığınızda hata mesajıyla karşılaşırsınız:

```
>>> 17 = n
```

Değişkenleri kağıt üstünde göstermenin genel bir yolu değişken isminden bir ok çıkarıp değerini işaret etmektir. Bu çeşit gösterime **durum diyagramı** denilir, çünkü her bir değişkenin durumunu gösterir (değişkenin ruh hali olarak da düşünebilirsiniz). Bu diyagram atama cümlelerinin sonuçlarını gösterir:



print cümlesi değişkenlerle de çalışır.

```
>>> print mesaj
Naber, doktor?
>>> print n
17
>>> print pi
3.14159
```

Her bir durumda sonuç değişkenin o an ki değeridir. Değişkenler de tiplere sahiptir; yorumlayıcıya tiplerini sorabiliriz.

```
>>> type(mesaj)
```

```
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Değişkenin tipi referans ettiği değerin tipidir.

2.3 Değişken isimleri ve anahtar kelimeler

Programcılar değişkenleri için genellikle anlamlı isimler seçerler, böylece kullandıkları değişkenleri ne için kullanıldıklarına dair belgelendirmiş olurlar.

Değişken isimleri isteğe bağlı olarak uzun olabilir. Hem harf hem de rakam içerebilir, ancak mutlaka bir harfle başlamaları gerekir. Her ne kadar büyük harf kullanmak geçerli olsa da, geleneksel olarak kullanmayız. Unutulmaması gereken harfin büyük veya küçük olması farklı şeylerdir.Emreve emre farklı değişkenlerdir.

Altçizgi karakteri (_) bir isimde yeralabilir. Genellikle birden fazla harf içeren kelimelerde kullanılmaktadır, benim_ismim veya cinde_cayin_fiyatişeklinde.

Eğer değişkene geçerli olmayan bir isim verirseniz, bir sözdizimi hatasıyla karşılaşırsınız:

```
 >>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

76trombones geçersizdir çünkü harf ile başlamamaktadır. more$ geçersizdir çünkü geçersiz bir karakter içermektedir, dolar işareti. Peki classisminin sorunu nedir?

class isminin Python tarafından kullanılan **anahtar kelimeler**den biri olduğu ortadadır. Anahtar kelimeler genellikle dilin kural ve yapısını tanımlarlar, ve değişken isimleri olarak kullanılamazlar.

Python otuz bir adet anahtar kelimeye sahiptir (bu kelimeleri değişken ismi olarak kullanamazsınız):

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

def for lambda try

Bu listeyi akılda tutmalı veya gerektiğinde erişebileceğiniz bir yerde olmalıdır. Eğer yorumlayıcı değişkenlerinizden birinin isminden şikayet eder ve siz nedenini bulamazsanız, bir de değişkenlerinizi bu listeye göre kontrol edersiniz.

2.4 Cümleler

Cümle Python yorumlayıcısı tarafından işlenebilecek bir yönergedir (komuttur). Şimdiye kadar iki cümle gördük: print ve atama cümleleri.

Bir cümleyi komut satırına yazdığınızda, Python cümleyi yürütür ve sonucu gösterir, eğer bir sonuç varsa. print cümlesinin sonucu bir değerdir. Atama cümleleri herhangi bir sonuç üretmez.

Betik genellikle peşpeşe gelen cümlelerden oluşur. Eğer birden fazla cümle varsa, sonuçlar cümleler teker teker işlendiği için teker teker gözükürler (ilgili cümle yürütülünce ilgili sonuç - eğer varsa - görüntülenir)

Örneğin, aşağıdaki betik:

```
print 1
x = 2
printx
```

şu çıktıyı üretir:

```
1
2
```

Yine tekrar etmek gerekir, atama cümlesi herhangi bir çıktı üretmez.

2.5 Deyimleri değerlendirme

Bir **deyim** değerlerden, değişkenlerden ve işleçlerden oluşan bir yapıdır. Eğer bir deyimi komut satırına yazarsanız, yorumlayıcı deyimi**değerlendirir** ve sonucu gösterir:

```
>>> 1 + 1
2
```

Deyimi değerlendirme bir değer üretir, bu nedenle deyimler atama işlecinin sağ tarafından kullanılabilir. Değer de basit bir deyimdir, değişken de öyle.

```
>>> 17
17
>>> x
2
```

Karıştırmamak gerekir, bir deyimi değerlendirmek, değeri ekranda göstermek ile aynı şey değildir.

```
>>> message = "Naber, doktor?"
>>> mesaj
"Naber, doktor?"
>>> print mesaj
Naber, doktor?
```

Python kabuğu bir deyimin değerini ekranda gösterirken, değer girmek için kullanacağınız aynı biçimi kullanır. Karakter dizileri kullandığınızı varsayarsak, görüntülemede tırnak işaretlerini kullandığını görebiliriz. Ancak print cümlesi deyimin değerini ekranda gösterir, bu durumda karakter dizisinin içeriğini gösterecektir.

Betik içerisinde, deyim kendi başına geçerli bir cümledir, ancak hiçbir şey yapmaz. Betik

```
17
3.2
"Merhaba, Dünya!"
1 + 1
```

herhangi bir çıktı üretmez. Bu betiği nasıl değiştirmeliyiz ki, bu dört deyimin değeri ekranda gösterilsin?

2.6 İşleçler ve işlenenler

İşleçler toplama, çarpma, vb. hesaplamaları temsil eden özel sembollerdir. İşleçler tarafından kullanılan değerler **işlenen** adını almaktadır.

Aşağıdakilerin hepsi geçerli (anlamı açık veya tam açık olmayan) Python deyimleridir:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

+, -, ve / sembolleri ve gruplama için parantez kullanımı Python'da da matematikteki anlamlarıyla kullanılmaktadır. Yıldız işareti (*) çarpmanın ve** işareti de üs almanın sembolleridir.

Bir işlenenin yerinde değişkenin ismi yer aldığında, işlem yapılmadan önce bu değişken değeriyle değiştirilir.

Toplama, çıkarma, çarpma ve üs alma işlemleri neyi bekliyorsanız onu yapan işlemlerdir. Ancak bölme işlemine şaşırabilirsiniz. Aşağıdaki örnek beklemediğiniz bir sonuç üretecektir:

```
>>> dakika = 59
>>> dakika/60
0
```

dakika'nın değeri 59'dur ve 59, 60 ile bölündüğünde sonuç 0.98333 olmalıdır, 0 değil. Python'da bu farklı sonucun ortaya çıkmasının nedeni yukarıdaki örnekte **tamsayı bölme** işleminin uygulanmasıdır.

Eğer her iki işlenen de tamsayı ise, sonuçta bir tamsayı olmalıdır, ve geleneksel olarak tamsayı bölme işlemi her zaman aşağı yuvarlanır, hatta örnekte olduğu gibi tamsayıların birbirine çok yakın olduğu durumlarda dahi.

Bu soruna olası bir çözüm kesir yerine yüzdeyi hesaplamaktır:

```
>>> dakika*100/60
98
```

Yine görüleceği üzere, sonuç aşağı yuvarlanmıştır ama en azından şimdi yaklaşık olarak doğru bir sonuç elde edebildik. Bir başka çözüm noktalı kayan (floating-point) bölme işlemini kullanmaktır. Dördüncü bölümde tamsayı değerleri ve değişkenleri kayan noktalı değerlere dönüştürmeyi göreceğiz.

2.7 İşleçlerin sırası

Eğer bir deyim içerisinde birden fazla işleç varsa, bu işleçlerin değerlendirilme sırası **öncelik kurallarına** göre belirlenir. Python matematikteki öncelik sırası kurallarının aynısını kullanır. PÜÇBTÇ (komik olduğunu biliyorum) kısaltması bu kuralları hatırlamak için kullanılabilecek bir kısaltmadır:

1. **P**arantezler en yüksek önceliğe sahiptir ve deyimlerin hangi sırada değerlendirilmesine yönelik ayarlamaları yapmanızı sağlarlar. Parantez içerisindeki deyimler daha önce değerlendirildiği için, 2 * (3-1) 4, ve (1+1)**(5-2) 8'dir. Parantezleri ayrıca deyimleri daha kolay okumak için kullanabilirsiniz, (minute * 100) / 60 örneğinde olduğu gibi, sonucu da değiştirmemiş olursunuz.
2. **Ü**s alma daha az önceliğe sahiptir, 2**1+1 ifadesi 3'tür 4 değil, ve 3*1**3 ifadesi de 3'tür 27 değil.
3. **Ç**arpma ve **B**ölme aynı önceliğe sahiptir, ve **T**oplama ve **Ç**ıkarmadan (ki bunlar da aynı önceliğe sahiptir) yüksek önceliklidir. 2*3-1ifadesi 4 yerine 5 üretir, ve 2/3-1 ifadesi -1'dir, 1 değil (tamsayı bölmede 2/3=0 olduğunu hatırlayalım).
4. Aynı önceliğe sahip işleçlerin değerlendirilmesinde soldan sağa kuralı izlenir. dakika*100/60 ifadesinde, çarpma işlemi önceliklidir 5900/60 sonucuna yol açar, bu ifade de 98'i üretir. Eğer işleçler sağdan sola değerlendirilecek olsaydı sonuç 59*1 olacaktır, bu da 59'u, yanlış bir sonucu üretecekti.

2.8 Karakter dizisi üzerindeki işlemler

Genel olarak, karakter dizileri üzerinde matematiksel işlemler uygulanamaz, her ne kadar karakter dizileri sayı gibi gözükse de. Aşağıdaki deyimler geçersiz deyimlerdir (mesaj'ın karakter dizisi (string) tipinde olduğunu varsayalım):

```
mesaj-1   "Merhaba"/123   mesaj*"Merhaba"   "15"+2
```

Ancak, + işleci karakter dizileriyle çalışmaktadır, ancak beklediğiniz sonucu üretmemektedir. Karakter dizileri için, + işleci **birleştirme (concatenation)** , bunun anlamı iki karakter dizisini uç uca bağlamaktır, işlemini yapacaktır. Örneğin:

```
meyve = "muz"
iyi_pisirilmis = " fındık ekmeği"
print meyve + iyi_pisirilmis
```

Bu programın çıktısı muz fındık ekmeği olacaktır. fındık'tan önceki boşluk karakter dizisinin bir parçasıdır, ve birleştirme işleminde birleştirilen karakter dizileri arasında boşluk olmasını sağlıyor.

* işleci de karakter dizileri üzerinde çalışır. Tekrarlama işlemini gerçekleştirir. Örneğin 'Eğlence'*3 ifadesi 'EğlenceEğlenceEğlence' sonucunu üretecektir. İşlenenlerden biri karakter dizisi, diğeri de tamsayı olmalıdır.

Diğer taraftan, + ve * işleçleri toplama ve çarpma benzerliğiyle daha rahat anlaşılabilir. 4*3 örneğinin 4+4+4 ifadesine eşit olması gibi, karakter dizilerinde de 'Eğlence'*3 ifadesinin 'Eğlence'+'Eğlence'+'Eğlence' ifadesine eşit olmasını bekliyoruz ve eşittir. Ancak, karakter dizisi birleştirme ve tekrarlamasının tamsayı toplama ve çarpmasına göre önemli bir farkı vardır. Toplama ve çarpma işleminin sahip olduğu ancak karakter dizisi birleştirme ve tekrarlama işleminin sahip olmadığı bir özellik düşünebiliyor musunuz?

2.9 Girdi

Klavyeden girdi alabilmek için Python içerisinde tanımlı iki fonksiyon vardır:

```
n = raw_input("Lütfen isminizi giriniz: ")
print n
n = input("Nümerik bir ifade giriniz: ")
print n
```

Bu betiğin çalıştırılması aşağıdaki gibi bir sonuç üretecektir:

```
$ python tryinput.py
Lütfen isminizi giriniz: Arthur, İngiltere Kralı
Arthur, İngiltere Kralı
Nümerik bir ifade giriniz: 7 * 3
```

```
21
```

Her bir fonksiyon parantez içerisinde verilmiş ifadenin gösterilmesini sağlar.

2.10 Kompozisyon

Şimdiye kadar programın öğelerini - değişkenler, deyimler, ve cümleler - ayrı ayrı inceledik, bunları nasıl birleştireceğimizden bahsetmedik.

Programlama dillerinin en kullanışlı özelliklerinden biri küçük yapısal blokları üretmek ve bunları birleştirebilme (kompozisyon) özelliğidir. Örneğin, sayıları nasıl ekleyeceğimizi ve ekranda değeri nasıl göstereceğimizi biliyoruz; bu ikisini birleştirerek aynı anda gerçekleştirebiliriz:

```
>>>  print 17 + 3
20
```

Gerçekte, toplama işlemi görüntülemeden önce yapılmalıdır, bu yüzden eylemler aynı anda olmamaktadır. Buradaki önemli nokta; sayıları, karakter dizilerini ve değişkenleri içeren herhangi bir deyim print cümlesi içinde kullanılabilir. Bunun örneklerini daha önce görmüştünüz:

```
      print "Geceyarısından beri geçen dakika: ", saat*60+dakika
```

İsteğe bağlı olarak bazı deyimleri atama cümlesinin sağında kullanabilirsiniz:

```
      yuzde = (dakika * 100) / 60
```

Bu özellik şu an etkileyici görünmeyebilir, ilerleyen süreçte kompozisyonun ne kadar etkili olabileceğini karmaşık hesaplamaları temiz ve kısaca yapabildiğinizde göreceksiniz.

Uyarı: Bazı deyimleri kullanmanıza yönelik olarak kısıtlamalar vardır. Örneğin, atama cümlesinin sol tarafı mutlaka değişken ismi olmalıdır, deyim olamaz. Bu yüzden, aşağıdaki geçersiz bir ifadedir: dakika + 1 = saat.

2.11 Yorumlar

Programlar büyüyüp, karmaşıklaştıkça okunması zorlaşmaktadır. Biçimsel diller yoğundur ve kaynak kodun bir parçasına bakıp ne yaptığını veya neden bunu yaptığını anlamak oldukça zordur.

Bu nedenle, programlara kodu açıklayan doğal dilde notlar ve açıklamalar yazmak iyi bir fikirdir. Bu notlara **yorumlar** adı verilmektedir, ve #işaretiyle belirtilir:

```
      # gecen saatin yuzdesini hesaplayalim
```

```
yuzde = (dakika * 100) / 60
```

Bu durumda yorum bir satır şeklinde gözükür. Yorumlar ayrıca satır sonuna da yerleştirilebilir:

```
yuzde = (dakika * 100) / 60 # dikkat: tamsayi bolme
```

işaretinden itibaren satır sonuna kadar her şey yoksayılır - program üzerinde bir etkisi yoktur-. Yorumların amacı ileride programı okuyacak ve inceleyecek programcılar içindir. Bu durumda, okuyucuya tamsayı bölme durumunu hatırlatma görevi üstlenmiştir.

2.12 Sözlük

değer:
Bir değişkende saklanabilecek veya bir deyimde hesaplanacak bir sayı veya karakter dizisidir (veya daha sonra isimlendirilecek bir şey).

tip:
Değerler kümesidir. Bir değerin tipi, deyimler içerisinde nasıl kullanabileceğini belirler. Şu ana kadar gördüğünüz değerler tamsayılar (inttipi), kayan noktalı sayılar (float tipi) ve karakter dizileridir (string tipi).

int:
Pozitif ve negatif tam sayıları tutan Python veri tipidir.

str:
Karakter dizisi (string) tutan Python veri tipidir.

float:
Kayan noktalı sayıları saklayan Python veri tipidir. Kayan noktalı sayılar içsel olarak iki parça şeklinde saklanır: bir taban ve bir üst. Standart biçimde görüntülendiğinde, ondalık sayılar gibi gözükürler. float kullandığınızda yuvarlama hatalarına dikkat etmeniz gerekmektedir, ve yaklaşık değer barındırırlar.

değişken:
Bir değeri temsil eden isimdir.

atama cümlesi:
Bir isime (değişkene) değer atayan cümledir. Atama işlecinin , =, sol tarafı bir isimdir. Atama işlecinin sağ tarafı ise Python yorumlayıcısı tarafından değerlendirilecek ve isime atanacak değeri üretecek deyimdir. Atama işlecinin sol ve sağ taraflar arasındaki fark genellikle yeni programcıların kafasını karıştırabilir. Aşağıdaki atamada:

```
n = n + 1
```

n değişkeni = işlecinin iki tarafında farklı görevler üstlenir. Sağ tarafta bir değerdir ve Python yorumlayıcısı tarafından sol taraftaki isme atanmadan önce değerlendirilecek deyimin bir kısmını oluşturur.

atama işleci:
= Python'un temel atama işlecidir, aynı işareti kullanan matematiksel karşılaştırma işleci ile karıştırılmamalıdır.

durum diyagramı:
Değişkenlerin ve referans ettikleri değerlerin grafiksel bir gösterimidir.

değişken ismi:
Bir değişkene verilen isimdir. Python'da değişken isimleri bir harf ile başlayan harf (a..z, A..Z ve _) ve sayılardan (0..9) serilerinden oluşur. En iyi programlama pratiğinde değişken isimleri program içerisindeki kullanımlarını anlatacak şekilde seçilir, böylece program için öz belgeleme yapmış olur.

anahtar kelime:
Derleyici tarafından programı ayrıştırmak için ayrılmış kelimelerdir; anahtar kelimelerini - if, def, ve while gibi - değişken isimleri olarak kullanamazsınız.

cümle:
Python yorumlayıcı tarafından yürütülebilecek yönergedir (komuttur). Cümlelerin örnekleri atama cümleleri ve print cümlesini içerir.

deyim:
Tek bir sonuç değeri temsil eden değişken, işleç ve değerlerin bir kombinasyonudur.

değerlendirme:
Bir deyimi basitleştirmek ve tek bir değer üretmek için sırasıyla işlemleri gerçekleştirmektir.

işleç:
Toplama, çarpma veya karakter dizisi birleştirme gibi tek bir hesaplamayı temsil eden özel işarettir.

işlenen:
İşlecin üzerinde işlem yaptığı değerlerden biridir.

tamsayı bölme:
Bir tamsayıyı başka bir tamsayı ile bölüp, tamsayı değer üreten işlemdir. Tamsayı bölme sadece bölünenin bölende kaç kere geçtiğini bulan tam sayılar üretir, kalanı yoksayar.

öncelik kuralları:
Çok sayıda işleç ve işlenen içeren deyimlerin değerlendirilme sırasını belirleyen kurallar kümesidir.

birleştirme:
İki işlenen uç uca eklemedir.

kompozisyon:
Basit deyimleri ve cümleleri, karmaşık hesaplamaları temsil etmek için bir araya getirip bileşik cümleler ve deyimler oluşturma özelliğidir.

yorum:
Program içerisinde diğer programcılar (veya kaynak kodu okuyan herhangi biri için) yazılmış olan bilgilerdir ve programın çalışması üzerinde bir etkisi yoktur.

2.13 Alıştırmalar

1. Aşağıdaki atama cümlelerini çalıştırdığınızda olanları kaydedin:

```
>>> print n = 7
```

Peki bu?

```
>>> print 7 + 5
```

Ya bu?

```
>>> print 5.2, "bu", 4 - 2, "şu", 5/2.0
```

print cümlesi için genel bir kural düşünebiliyor musunuz? print cümlesi ne döndürmektedir?

2. Tüm iş ve oyun oynamama Mustafa'yı anlayışsız yapmaktadır cümlesini ele alın ve her kelimeyi farklı bir değişkende saklayın, son olarak cümleyi tek bir print cümlesi yardımıyla tek satırda yazdırın.
3. 6*1-2 ifadesine parantez ekleyerek değerini 4'ten -6'ya dönüştürün.
4. Daha önce çalışmış bir koda yorum ekleyerek tekrar çalıştırın, ve sonucu inceleyin.
5. input ve raw_input arasındaki fark input girdi karakter dizisini değerlendirirken, raw_input değerlendirmemektedir. Aşağıdakileri yorumlayıcıda sınayın ve sonucu inceleyin:

```
 >>> x = input()
3.14
>>> type(x)


 >>> x = raw_input()
3.14
>>> type(x)


 >>> x = input()
'The knights who say "ni!"'
>>> x
```

Aşağıdakini tırnak işaretleri olmadan denerseniz ne olur?

```
 >>> x = input()
The knights who say "ni!"
>>> x
```

```
 >>> x = raw_input()
'The knights who say "ni!"'
>>> x
```

Her sonucu anlatın ve açıklayın.

6. Python yorumlayıcısını başlatın ve bruce+4 ifadesini bilgi istemine girin. Bu size bir hata verecektir:

```
 NameError: name 'bruce' is not defined
```

bruce ismine bir değer atayın, böylece bruce + 4 10 değerini üretebilsin.

7. İsmi madlib.py bir program (Python betiği) yazın, bu program kullanıcıdan peşpeşe isimler, yüklemler, tümleçler, sıfatlar, çoğul isimler, geçmiş zaman yüklemleri, vb. cümle öğeleri istesin ve daha sonra bu girilen ifadeleri sözdizimi olarak doğru ancak anlambilimsel olarak yanlış bir paragraf üretsin (örnekler için http://madlibs.org adresini inceleyebilirsiniz).

3. Fonksiyonlar

3.1 Tanımlar ve kullanım

Programlama bağlamın, bir **fonksiyon (işlev)** belli bir işlemi gerçekleştirmek üzere isimlendirilmiş cümle (komut) serisidir. Bu işlem **fonksiyon tanımında** belirlenmiştir. Python'da, fonksiyon tanımı için sözdizimi şu şekildedir:

```
def ISIM( PARAMETRE LISTESI ):
    CUMLELER
```

Fonksiyonlar için dilediğiniz ismi kullanabilirsiniz, ancak değişkenlerde de olduğu gibi Python anahtar kelimelerini kullanamazsınız. Parametre listesi fonksiyon tarafından kullanılması gereken, varsa, bilgileri belirtmek için kullanılır.

Fonksiyon içerisinde herhangi sayıda cümle bulunabilir, ancak def'e göre daha içerden başlamaları gerekiyor. Bu kitaptaki örneklerde standart içeriden başlama (indentation - girinti) olan dört boşluk kullanılacaktır. Fonksiyon tanımlamaları **bileşik cümlelerin** ilk örneğidir, ileride de anlaşılacağı gibi hepsinin kalıbı aynıdır:

1. Bir **başlık**, bir anahtar kelime ile başlar ve iki nokta üst üste ile biter
2. Bir **gövde**, bir veya daha fazla Python cümlesi içerir ve herbiri başlığa göre eşit oranda içeriden - Python standartı 4 boşluk karakteridir - başlar.

Bir fonksiyon tanımlamasında, başlıktaki anahtar kelime deftir, bu ifade daha sonra fonksiyon ismi ve parantezlerle sınırlanmış parametrelerle izlenir. Tanımın sonunda iki nokta üst üste vardır. Parametre listesi boş olabilir veya herhangi bir sayıda parametre içerebilir. Her iki durumda da parantezler olmazsa olmazdır.

Yazacağımız ilk fonksiyon örnekleri parametre içermeyecektir, bu yüzden sözdizimi şu şekildedir:

```
def yeni_satir():
    print           # herhangi bir arguman almayan print cumlesi yeni
bir satir yazar
```

Bu fonksiyon yeni_satir ismindedir. Boş parantezler parametre almadığını gösterir. Gövdesinde sadece bir cümle vardır, bu cümle yeni satır karakterini ekranda göstermektedir (bunun anlamı bir boş satır bırakılmasıdır, boş bir print cümlesi kullanıldığında oluşan durumdur).

Yeni bir fonksiyon tanımlamak o fonksiyonun çalışmasını sağlamaz. Fonksiyonu çalıştırabilmemiz için bir **fonksiyon çağrısı** yapmamız gerekir. Fonksiyon çağrıları, çalıştırılacak fonksiyonun ismi ve onu takip eden parantez içerisindeki fonksiyona aktarılacak olan değer listesinden - bunlara argüman adı verilir - oluşur. Bu argümanlar fonksiyon tanımındaki parametrelere eşlenir. İlk örneklerimiz, boş parametre listesine sahip olduğu için, çağrılarımızda argüman göndermeyeceğiz. Ancak yine de parantezleri - içi boş olsa da - yazmaya devam edeceğiz:

```
print "İlk satır."
yeni_satir()
print "İkinci satır."
```

Programın çıktısı:

```
İlk satır.

İkinci satır.
```

İki satır arasındaki ek boşluk, yeni_satir() fonksiyon çağrısının bir sonucudur. Eğer satırlar arasında daha fazla boşluk isteseydik ne yapacaktık? Aynı fonksiyonu tekrar tekrar çağıracaktık:

```
print "İlk satır."
yeni_satir()
yeni_satir()
yeni_satir()
print "İkinci satır."
```

Veya üç yeni satır yazan uc_satir isminde yeni bir fonksiyon yazabilirdik:

```
def uc_satir():
    yeni_satir()
    yeni_satir()
    yeni_satir()

print "İlk satır."
uc_satir()
```

```
print "İkinci satır."
```

Bu fonksiyon üç cümle içermektedir, ve herbir cümle dört boşluk karakteri içeriden başlamaktadır. Sonraki satır içeriden başlamadığı için Python onun fonksiyonun bir parçası olmadığını anlayacaktır.

Bu program hakkında bir kaç şey dikkatinizi çekmiş olmalı:

- Aynı yordamı tekrar tekrar çağırabilirsiniz. Gerçekte, bu çok olağandır ve yapmanız yararlıdır.
- Bir fonksiyon çağrısının, bir başka fonksiyonu çağırmasını sağlayabilirsiniz; bu durumda uc_satir fonksiyonu yeni_satir fonksiyonunu çağırmıştır.

Şimdiye kadar bu yeni fonksiyonları yazmak için neden bu kadar zahmete katlandığımızı anlamamış olabilirsiniz. Aslında oldukça fazla neden vardır, aşağıdaki örnek iki nedeni gösterecektir:

1. Yeni bir fonksiyon yaratma, size cümle gruplarına bir isim verme şansı verir. Fonksiyonlar karmaşık hesaplamaları tek bir komutun arkasına saklayarak ve adı karışık komutlar yerine anlamlı isimlere (Türkçe, İngilizce) sahip komutlar kullanmanızı sağlayarak programı basitleştirirler.
2. Yeni fonksiyonlar yaratmak programınızı küçültebilir. Fonksiyonlar tekrar eden kodların elenmesini sağlar. Örneğin, dokuz kere ekrana boş satır yazacak komut yerine uc_satir fonksiyonunu üç kere çağırarak daha az kod yazmış olursunuz.

Önceki kod parçalarını biraraya getirip tryme1.py isimli bir betik içerisine yazdığımızda, tüm program aşağıdaki gibi gözükecektir:

```
def yeni_satir():
    print

def uc_satir():
    yeni_satir()
    yeni_satir()
    yeni_satir()

print "İlk satır."
uc_satir()
print "İkinci satır."
```

Bu program iki fonksiyon tanımlaması içermektedir: yeni_satir ve uc_satir. Fonksiyon tanımlamaları diğer cümleler gibi çalıştırılırlar. Bu etkisi yeni bir fonksiyon yaratılması şeklindedir. Fonksiyon içerisindeki cümleler fonksiyon çağrımı yapmadan işletilmezler ve fonksiyon tanımlaması herhangi bir çıktı üretmez.

Tahmin edebileceğiniz gibi, bir fonksiyonu çalıştırmadan önce yaratmanız gerekir. Başka bir şekilde açıklayacak olursak, fonksiyon tanımlaması fonksiyonun ilk çağrısından önce çalışmalıdır.

3.2 Yürütme akışı

Fonksiyonun ilk kullanımından önce tanımlandığından emin olmak için, cümlelerin yürütülme sırasını bilmeniz gerekir, bu sıraya **yürütme akışı (flow of execution)** adı verilmektedir.

Yürütme her zaman programın ilk satırıyla başlar. Cümleler her seferinde biri olmak üzere yukarıdan aşağıya doğru sırayla çalıştırılır.

Fonksiyon tanımlamaları programın yürütme akışını değiştirmezler, ancak fonksiyon içerisindeki cümlelerin fonksiyon çağrılana kadar yürütülmediğini unutmayın. Her ne kadar sıklıkla kullanılan bir yöntem olmasa da, bir fonksiyon içerisinde başka bir fonksiyon tanımlayabilirsiniz. Bu durumda içerideki fonksiyon tanımı, içerisinde tanımlandığı fonksiyon çağrılana kadar yürütülmeyecektir.

Fonksiyon çağrımları yürütme akışındaki sapmalar gibidir. Sonraki cümleye gitmek yerine, akış çağrılan fonksiyon içerisindeki ilk cümleye atlar, fonksiyondaki tüm cümleleri çalıştırır daha sonra bıraktığı yere (fonksiyonun çağrıldığı satır) döner.

Bu her ne kadar kolaymış gibi görünse de, bir fonksiyonun başka bir fonksiyonu çağırabildiğini düşündüğünüzde bu sapmaların bir süre sonra takibi zorlaşabilir. Düşünün ki, bir fonksiyonun ortasındayken bir başka fonksiyon çağrılabilir. Ayrıca bu yeni çağrılan fonksiyon içerisinde de bir başka fonksiyon içerisindeki cümlelerin yürütülmesine ihtiyaç duyulabilir. Bu böyle gider!

Şansımız var ki, Python nerede kaldığını tutmada oldukça beceriklidir, böylece her fonksiyon yürütülmesi bittiğinde, program kaldığı yere geri döner. Programın sonuna geldiğinde programı sonlandırır.

Bu sefil hikayenin anafikri nedir? Bir programı okuduğunuzda, ne yukarıdan aşağıya doğru ne aşağıdan yukarıya doğru okumayın. Bunun yerine, yürütme akışını takip edin.

3.3 Parametreler, argümanlar, ve import cümlesi

Çoğu fonksiyon argümanlara gereksinimi duyar, bu değerler fonksiyonun görevini yaparken kullandığı ve bir bakıma bu görevi nasıl yapacağını belirlerler. Örneğin, bir sayının mutlak değerini bulmak istiyorsanız, hangi sayının mutlak değerini bulmak istediğinizi belirtmeniz gerekir. Python bu iş için içerisinde mutlak değer hesaplayan bir fonksiyon barındırır:

```
>>> abs(5)
5
>>> abs(-5)
5
```

Bu örnekte, abs fonksiyonuna gönderdiğimiz argümanlar 5 ve -5'tir.

Bazı fonksiyonlar birden fazla argüman alabilir. Örneğin Python içerisinde tanımlı pow fonksiyonu iki argüman alır, taban ve üs. Fonksiyonun içerisinde bu değerler **parametre** adı verilen değişkenlere atanırlar.

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

Birden fazla argüman alan bir başka varolan fonksiyon max'tır.

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3*11, 5**3, 512-9, 1024**0)
503
```

max fonksiyonuna virgüllerle ayrılmış dilediğiniz sayıda argüman gönderebilirsiniz. Gönderilen argümanlar arasından en büyük sayıyı bulup geri döndürecektir. Argümanlar basit değerler olabildiği gibi deyim de olabilirler. Son örnekte 503 döndürüldü, çünkü 33, 125 ve 1'den daha büyüktür (deyimler değerlendirilerek sonuçlar fonksiyona geçiyor).

Kullanıcı tanımlı ve tek parametre içeren bir fonksiyon örneği:

```
def iki_kere_yaz(bruce):
    print bruce, bruce
```

Bu fonksiyon tek bir **argüman** alır ve onu bruce isimli parametreye atar. Parametrenin değeri (bu noktada ne olduğuna dair bir fikrimiz yok) iki kere yazılır, sonrasında bir yeni satır ekranda gösterilir. bruce isminin seçilmesinin nedeni parametrelere dilediğiniz ismi verebileceğinizi göstermek içindir. Elbette bruce isminden daha yaratıcı ve anlamlı isimler seçmeniz kaynak kodun okunabilirliği açısından önemlidir.

Etkileşimli Python kabuğu bize fonksiyonlarımızı sınamak için elverişli bir yol sağlarlar. **import cümlesini** bir betik içerisinde tanımladığımız fonksiyonları yorumlayıcı oturumuna getirmek için kullanabiliriz. Bunun nasıl çalıştığını anlamak için, iki_kere_yaz fonksiyonunun chap03.pyisimli bir betik içerisinde tanımlı olduğunu varsayalım. Etkileşimli olarak sınamak için betiği kendi Python kabuk oturumumuza aktarırız::

```
>>> from chap03 import *
>>> iki_kere_yaz('Spam')
Spam Spam
>>> iki_kere_yaz(5)
5 5
>>> iki_kere_yaz(3.14159)
3.14159 3.14159
```

Bir fonksiyon çağrımında, argümanın değeri ilgili parametreye fonksiyon tanımlamasında atanmaktadır. Gerçekte bruce = 'Spam'

atamasıiki_kere_yaz('Spam') çağrımı yapılınca, bruce = 5 iki_kere_yaz(5) ve bruce = 3.14159 iki_kere_yaz(3.14159) çağrımı yapıldığında gerçekleşir.

Yazılabilir herhangi bir argüman iki_kere_yaz fonksiyonuna gönderilebilir. İlk fonksiyon çağrımında, argüman bir karakter dizisidir. İkinci çağrımda tamsayı, üçüncü çağrımda kayan float tipindedir.

İçsel (varolan) fonksiyonlar gibi iki_kere_yaz argümanı yerine deyim de kullanabiliriz:

```
>>> iki_kere_yaz('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
```

'Spam'*4 deyimi ilk önce değerlendirilip 'SpamSpamSpamSpam' şeklinde iki_kere_yaz fonksiyonuna argüman olarak geçirilmektedir.

3.4 Kompozisyon

Matematiksel fonksiyonlarda olduğu gibi, Python fonksiyonları da **dizilebilir**, bunun anlamı bir fonksiyonun sonucu bir başka fonksiyona girdi olarak verilebilir.

```
>>> iki_kere_yaz(abs(-7))
7 7
>>> iki_kere_yaz(max(3, 1, abs(-11), 7))
11 11
```

İlk örnekte, abs(-7) deyimi 7 şeklinde değerlendirilir ve iki_kere_yaz fonksiyonuna argüman olarak verilir. İkinci örnekte iki seviyeli bir dizme sözkonusudur. İlk olarak abs(-11) ifadesi 11 sonucunu ürettiği için max(3,1,11,7) fonksiyonu 11 değerini üretir. Sonrasında bu 11 değeriiki_kere_yaz(11) fonksiyonuna argüman olarak geçirilir ve sonuç ekranda görüntülenir.

Ayrıca değişkenleri de argüman olarak kullanabiliriz:

```
>>> michael = 'Erik, yarım arı.'
>>> print_twice(michael)
Erik, yarım arı. Erik, yarım arı.
```

Burada dikkat etmeniz gereken önemli bir nokta var. Argüman olarak geçirdiğimiz değişkenin isminin (michael) parametre ismi (bruce) ile bir ilgisi yok. Çağrıldığı yerde o değişkenin isminin ne olduğunun bir önemi yok, fonksiyon içerisinde (iki_kere_yaz) parametre ismi (bruce) kullanılmaktadır, herkesi bu isim (bruce) ile sesleniriz.

3.5 Değişkenler ve parametreler yereldir

Bir fonksiyon içerisinde bir **yerel değişken** yarattığınızda, o değişken sadece o fonksiyon içerisinde varolur, fonksiyon dışında o değişkeni kullanamazsınız. Örneğin:

```
def iki_kere_birlestir(part1, part2):
    cat = part1 + part2
    iki_kere_yaz(cat)
```

Bu fonksiyon iki argüman alır, bu argümanları birleştirir ve sonucu iki kere ekranda görüntüler. Bu fonksiyonu iki karakter dizisi ile çağırabiliriz:

```
>>> chant1 = "Pie Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> iki_kere_birlestir(chant1, chant2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

When iki_kere_birlestir bittiğinde, cat değişkeni yokedilir. Onu yazdırmaya kalkışırsak bir hata ile karşılaşırız:

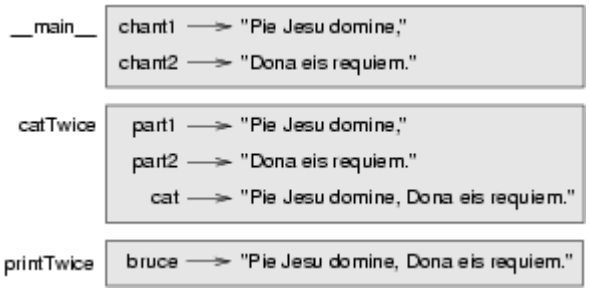
```
>>> print cat
NameError: name 'cat' is not defined
```

Parametreler de yereldir. Örneğin, iki_kere_yaz fonksiyonu dışında bruce ismiyle tanımlı bir değişken yoktur. Kullanmaya kalkışırsanız, Python yine şikayet edecektir.

3.6 Yığıt diyagramları

Hangi değişkenlerin nerede kullanıldığını takip edebilmek için, **yığıt diyagramları** çizmek faydalı olabilir. Durum diyagramları gibi, yığıt diyagramları da her değişkenin değerini gösterir, ancak ayrıca her değişkenin ait olduğu fonksiyonu da gösterirler.

Her fonksiyon bir **çerçeve** ile temsil edilir. Bir çerçeve yan tarafında fonksiyonun ismi ve içerisinde parametre ve değişkenlerin bulunduğu bir kutudur. Önceki kod örneğinin yığıt diyagramı aşağıdaki gibidir:



Yığıtın sırası yürütme akışını gösterir. iki_kere_yaz iki_kere_birlestir tarafından çağrılmıştır ve iki_kere_birlestir __main__ (en üstteki fonksiyonun özel ismidir) tarafından çağrılmıştır. Herhangi bir fonksiyonun dışında tanımladığınız değişkenler __main__'e aittir.

Her parametre ilgili argümanıyla aynı değeri gösterir. Bu yüzden, part1 chant1 ile aynı değere, part2 chant2 ile aynı değere ve bruce cat ile aynı değere sahiptir.

Bir fonksiyon çağrımı esnasında bir hata oluşursa Python fonksiyon ismini ve o fonksiyonu çağıran fonksiyonun ismini yazar, o fonksiyonu da çağıran fonksiyon ismini, ... en üstteki fonksiyona ulaşana kadar tüm çağrım yapan fonksiyonları yazar.

Bunun nasıl çalıştığını görmek için tryme2.py isminde ve içeriği aşağıdaki gibi olan bir betik yaratalım:

```
def iki_kere_yaz(bruce):
    print bruce, bruce
    print cat

def iki_kere_birlestir(part1, part2):
    cat = part1 + part2
    iki_kere_yaz(cat)

chant1 = "Pie Jesu domine, "
chant2 = "Dona eis requim."
iki_kere_birlestir(chant1, chant2)
```

iki_kere_yaz fonksiyonu içerisine print cat cümlesini ekledik, ancak cat değişkeni orada tanımlı olmadığı için bu betik hata üretecektir:

```
Traceback (innermost last):
  File "tryme2.py", line 11, in <module>
    cat_twice(chant1, chant2)
  File "tryme2.py", line 7, in cat_twice
    print_twice(cat)
  File "tryme2.py", line 3, in print_twice
    print cat
NameError: global name 'cat' is not defined
```

Bu fonksiyon listesine **geri izleme (traceback)** adı verilmektedir. Hangi program dosyasında, hangi satırda ve hangi fonksiyonları işletirken hatanın oluştuğuna dair bilgi verir. Ayrıca hataya neden olan kod satırını da gösterir.

Geri izleme ile yığıt diyagramı arasındaki benzerliğe dikkat edin. Bu bir rastlantı değil. Gerçekte geri izlemenin bir başka ismi de yığıt izlemedir.

3.7 Sözlük

fonksiyon (işlev):
Bazı faydalı işler başarabilen isimlendirilmiş cümle dizisidir. Fonksiyonlar parametre alabilir veya almayabilir, sonuç üretebilir veya üretmeyebilir.

fonksiyon tanımı:
Yeni bir fonksiyon yaratan cümledir, fonksiyonun ismini, parametrelerini ve yürüteceği cümleleri belirtir.

bileşik cümle:
İki parçadan oluşan cümledir:

1. başlık - cümlenin tipini belirleyen bir anahtar kelimeyle başlar ve iki nokta üstüste ile biter.
2. gövde - başlıktan eşit oranda içeride yazılmış olan bir veya daha fazla cümledir

Bileşik cümlenin sözdizimi aşağıdaki gibidir:

```
anahtar_kelime deyim :
    cümle
    cümle ...
```

başlık:
Bileşik cümlenin ilk kısmıdır. Bir anahtar kelime ile başlar ve iki nokta üst üste (:) ile biter.

gövde:
Bileşik cümlenin ikinci kısmıdır. Gövde başlıktan eşit derecede içeride başlayan cümle dizilerinden oluşur. Python topluluğu tarafından kullanılan standart girinti 4 boşluktur.

fonksiyon çağrımı:
Fonksiyonu yürütmeye başlatan cümledir. Fonksiyonun ismiyle başlar ve parantezler içerisindeki argümanlar (eğer varsa) ile devam eder.

yürütme akışı:
Bir programın çalışması sırasında cümlelerin yürütülmesi sırasıdır.

parametre:
Bir fonksiyon içerisinde kullanılan ve fonksiyona geçirilen argümanı referans eden isim.

içeri aktarma (import):

Bir Python betiği içerisinde tanımlanmış fonksiyon ve değişkenlerin başka bir betik ortamına veya Python kabuğuna getirilmesi için kullanılan cümledir.

Örneğin, aşağıdaki kodun tryme.py isimli bir betik içerisinde bulunduğunu düşünelim:

```
def print_thrice(thing):
    print thing, thing, thing

n = 42
s = "And now for something completely different..."
```

Şimdi tryme.py dosyasının bulunduğu dizin içerisinde bir Python kabuğu başlatalım:

```
$ ls
tryme.py  <ve diğer şeyler...>
$ python
>>>
```

tryme.py içerisinde üç isim tanımlanmıştır: print_thrice, n, and s. Eğer içe aktarmadan (import) bu isimleri kullanmaya çalışırsak, bir hata oluşur:

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
>>> print_thrice("ouch!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print_thrice' is not defined
```

tryme.py betiğinden herşeyi içe aktarırsak, içerisinde tanımlı herşeyi kullanabiliriz:

```
>>> from tryme import *
>>> n
42
>>> s
'And now for something completely different...'
>>> print_thrice("Yipee!")
Yipee! Yipee! Yipee!
>>>
```

Dikkat etmeniz gereken nokta import cümlesinde .py uzantısını kullanmıyorsunuz.

argüman:
Bir fonksiyon çağrıldığında, bu fonksiyona aktarılan değerdir. Bu değer fonksiyonda ilgili parametreye atanır.

fonksiyon dizilimi (kompozisyon):
Bir fonksiyon çıktısını bir başka fonksiyona girdi olarak kullanmaktır.

yerel değişken:
Bir fonksiyon içerisinde tanımlı değişkendir. Yerel değişken sadece kendi fonksiyonu içerisinde kullanılabilir.

yığıt diyagramı:
Fonksiyonların, değişkenlerinin ve değişkenlerin gösterdiği değerlerin grafiksel gösterimidir.

çerçeve:
Yığıt diyagramında bir fonksiyon çağrısını temsil eden çerçevedir. Fonksiyonun yerel değişkenleri ve parametrelerini içerir.

geri izleme:
Bir çalışma zamanı hatası oluştuğunda çalışan fonksiyonların bir listesidir. Geri izleme ayrıca yığıt izleme olarak ta adlandırılır. Bu listede fonksiyonlar çalışma zamanı yığıtı içerisindeki sıraya göre bulunurlar.

3.8 Alıştırmalar

1. Bir metin düzenleyici kullanarak tryme3.py bir Python betiği yazın. Bu dosya içerisinde three_lines fonksiyonunu kullanarak dokuz boş satır üreten nine_lines isimli bir fonksiyon yazın. Daha sonra clear_screen isimli, ekrana 25 adet boş satır yazıp, ekranı temizleyen fonksiyonu yazın. Programınızın son satırında clear_screen fonksiyonuna bir çağrı yapmanız gerekiyor.
2. tryme3.py programının son satırını en üste taşıyın, clear_screen fonksiyonuna yapılan fonksiyon çağrısı böylece fonksiyon tanımlamasından önce yapılmış olacak. Programı çalıştırın ve ortaya çıkan hata mesajını kaydedin. Fonksiyon tanımlamaları ve fonksiyon çağrıları ile ilgili birbirlerine bağlı olarak kod içerisinde nerede bulunabileceklerine dair bir kural oluşturabilir misiniz?
3. Hatasız çalışan bir tryme3.py kullanarak, new_line fonksiyon tanımlamasını three_lines tanımlamasından sonraya taşıyın. Bu programı çalıştırdığınızda ne olduğunu gözlemleyin. Şimdi new_line tanımlamasını three_lines() çağrısının altına taşıyalım ve çalıştıralım. Ortaya çıkan sonucu bir önceki örnekte tanımladığınız kural açısından değerlendirin.
4. s karakter dizisini n kere yazacak şekilde cat_n_times fonksiyon tanımlamasının gövdesini doldurun:

```
def cat_n_times(s, n):
    <kodu buraya yazın>
```

Bu kodu import_test.py betiği içerisine kaydedin. Şimdi bir unix bilgi isteminde bu betik ile aynı dizinde olduğunuzdan emin olduktan sonra (ls komutu ile import_test.py dosyasıyla aynı dizinde olup olmadığınızı kontrol edebilirsiniz). Bir Python kabuğu başlatın ve aşağıdakileri deneyin:

```
>>> from import_test import *
>>> cat_n_times('Spam', 7)
SpamSpamSpamSpamSpamSpamSpam
```

Eğer herşey düzgünse sizin oturumunuz da yukarıdaki gibi çalışacaktır. cat_n_times fonksiyonuna farklı çağrıları deneyerek nasıl çalıştığını iyice anlamaya çalışın.

4. Koşul ifadeleri

1. Modül işleci
2. Boolean değerler ve deyimler
3. Mantıksal işleçler
4. Koşullu yürütme
5. Alternatif yürütme
6. Zincirleme koşul ifadeleri
7. İçiçe koşul ifadeleri
8. Geri dönüş cümlesi

4.1 Modül (Kalan) işleci

Modül işleci tamsayılarla (ve tamsayı deyimlerle) çalışan ve ilk işlenen ikinci işlenene bölünmesiyle oluşan kalanı veren bir işleçtir. Python'da, modül işleci yüzde işareti(%)'dir. Sözdizimi diğer işleçlerle aynıdır:

```
>>> bolum = 7 / 3
>>> print bolum
2
>>> kalan = 7 % 3
>>> print kalan
1
```

7'yi 3 ile böldüğümüzde sonuç 2, ve kalan 1'dir.

Modül işlecinin faydalı olduğunu görebilirsiniz. Örneğin bir sayının bir başka sayı ile bölünebilir olup olmadığını -- x % y sıfıra eşit ise x y ile bölünebilirdir-- sınayabilirsiniz.

Ayrıca sayının en sağ basamağındaki rakam veya rakamları elde edebilirsiniz. Örneğin, x % 10 deyimi x sayısının en sağındaki rakamı (onluk sisteme göre) üretir. Benzer olarak x % 100 en sağdaki iki rakamı döndürür.

4.2 Boolean değerler ve deyimler

Python'da doğru ve yanlış değerleri saklamak için kullanılan tipe bool adı verilmektedir, bu isim (tüm dillerde olduğu gibi) Boolean Cebirini yaratan İngiliz matematikçi George Boole'dan dolayı verilmiştir. Boole cebiri tüm modern bilgisayar aritmetiğinin temelidir.

Sadece iki **boolean değer** vardır: True(doğru) ve False(yanlış). Python açısından büyük harfle başlamaları önemlidir. true ve false boolean değerler değildir.

```
>>> type(True)
<type 'bool'>
>>> type(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

Boolean deyimi, boolean değer şeklinde değerlendirilen (sonucu boolean değer olan) bir deyimdir. == işleci iki değeri karşılaştırıp boolean değer üretir:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

İlk cümlede, iki işlenen eşittir, bu yüzden deyim True sonucunu üretir; ikinci cümlede ise 5, 6'ya eşit değil, bu yüzden False sonucu elde ederiz.== işleci **karşılaştırma işleçleri**nden biridir; diğerleri şunlardır:

```
x != y               # x, y'e eşit değil
x > y                # x, y'den büyük
x < y                # x, y'den küçük
x >= y               # x, y'den büyük veya eşit
x <= y               # x, y'den küçük veya eşit
```

Her ne kadar bu işlemler size tanıdık gelse de, Python tarafından kullanılan işaretler matematiksel işaretlerden farklıdır. En çok karşılaşılan hata çift eşit işareti (==) yerine tek eşit (=) işaretini kullanmaktır. Unutulmaması gereken = işareti atama işlecidir ve == karşılaştırma işlecidir. Ayrıca =<ve => işaretleri tanımlı değildir.

4.3 Mantıksal işleçler

Üç adet **mantıksal işleç** vardır: and (ve), or (veya), ve not (değil). Bu işleçlerin anlamları, parantez içinde yazılmış olan Türkçe anlamlarıyla benzerdir. Örneğin x > 0 and x < 10 ifadesi sadece x 0'dan büyük ve x 10'dan küçük olduğunda doğrudur.

n % 2 == 0 or n % 3 == 0 ifadesi iki koşuldan biri doğru olduğunda (veya'dan dolayı)doğrudur. Bunun anlamı sayı 2 ile bölünebilir veya 3 ile bölünebildiğinde doğrudur.

Son olarak, not işleci bir boolean deyimin zıttını (negatifi) üretmede kullanılır. Böylece not(x > y) ifadesi (x > y) ifadesi yanlış olduğunda doğru olacaktır. Bu ifade x y'den daha küçük veya eşit olduğunda doğrudur.

4.4 Koşullu yürütme

Yararlı programlar yazabilmek için, neredeyse her zaman koşulları sınamamız ve programın davranışlarını bu sınamaların sonucuna göre değiştirebilmemiz gerekir. **Koşul cümleleri** bize bu yeteneği kazandırır. En basit örneği **if cümlesidir**:

```
if x > 0:
    print "x pozitiftir"
```

if cümlesinden sonra yazılmış olan boolean deyim'e **koşul** adı verilir. Eğer bu koşul doğru ise alttaki cümle çalışır. Eğer değilse, herhangi bir şey olmaz.

if cümlesinin sözdizimi şu şekildedir:

```
if BOOLEAN DEYIM:
    CUMLELER
```

Bir önceki bölümde gördüğümüz fonksiyon tanımlaması ve diğer bileşik cümleler gibi, if cümlesi bir başlık ve gövdeden oluşur. Başlık ifanahtar kelimesi ile başlar ve bir boolean deyimle devam eder. Sonu yine iki nokta üstüste (:) ile belirlenmiştir.

Takip eden girintili cümlelere **blok** adı verilir. İlk girintisiz cümle bloğun sonunu belirler. Bileşik cümlelerdeki cümle bloğuna cümlenin **gövde**si adı verilir.

Gövdedeki her bir cümle eğer boolean deyim doğruysa (True) sırasıyla yürütülür. Eğer boolean deyim yanlışsa (False) bütün blok yoksayılır ve çalıştırılmaz.

if cümlesinin gövdesinde bulunabilecek cümle sayısında bir sınır yoktur, ama en azından bir cümlenin olması gerekir. Bazen boş if cümlesi yazmak (mesela daha sonra yazmak üzere alanı ayırmak için) gerekebilir. Bu durumda pass cümlesini - ki bu cümle hiç bir şey yapmaz - kullanabilirsiniz.

```
if True:           # Bu her zaman dogrudur
    pass           # bu cumle her zaman yurutulecektir,ancak bir sey
yapmamaktadir
```

4.5 Alternatif yürütme

if cümlesinin bir biçimi de alternatif yürütmeye imkan sağlayan, iki yürütme olasılığı sağlayan ve buna koşulun sonucuyla karar veren cümledir. Sözdizimi şu şekildedir:

```
if x % 2 == 0:
    print x, "çifttir"
else:
    print x, "tektir"
```

Eğer x sayısının 2 ile bölünmesinden kalan 0 ise, x'in çift olduğunu biliriz, ve program bunu bize bildiren bir mesaj yazar. Eğer koşul yanlış ise (x'in 2'ye bölümünden kalan 0 değil ise) else'in altındaki cümleler çalışacaktır. Her halükarda boolean deyimi doğru veya yanlış olabileceğine göre bu iki alternatif yürütmeden biri mutlaka çalışacaktır. Bu alternatiflere **dal**(yol - branch) adı verilmektedir, çünkü yürütme akışındaki farklı dalları temsil ederler.

Şu ana kadar söylediklerimizi bir kenara bırakırsak, eğer sayıların çift veya tek olduğunu kontrol etmek isterseniz, yukarıdaki kodu bir fonksiyon içerisine koymak(wrap) isteyebilirsiniz:

```
def print_parity(x):
    if x % 2 == 0:
        print x, "çifttir"
    else:
        print x, "tektir"
```

x'in herhangi bir değeri için print_parity fonksiyonu uygun mesajı gösterecektir. Fonksiyonu çağırdığınızda, argüman olarak herhangi bir tamsayı değer gönderebilirsiniz.

```
>>> print_parity(17)
17 tektir.
>>> y = 41
>>> print_parity(y+1)
42 çifttir.
```

4.6 Zincirleme koşul ifadeleri

Bazen ikiden fazla olasılık vardır ve iki dallanmadan fazlasına gereksinim duyarız. Bunun gibi bir hesaplamayı ifade etmek için **zincirleme koşul ifadeleri**ni kullanırız:

```
if x < y:
    print x, "küçüktür", y
elif x > y:
    print x, "büyüktür", y
else:
    print x, "ve", y, "eşittir"
```

elif else if'in bir kısaltmasıdır. Daha öncede olduğu gibi sadece bir dal (yol) yürütülecektir. elif cümlelerinin sayısında bir sınırlama yoktur ancak sadece bir adet (isteğe bağlı olmak üzere) else cümlesine izin vardır ve bunun son cümle (son dal) olması gereklidir:

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
elif choice == 'c':
    function_c()
else:
    print "Yanlış seçim."
```

Her koşul sırasıyla sınanır. Eğer ilki yanlış ise, sonraki kontrol edilir, ve bu böyle gider. Eğer koşullardan biri doğru ise, ilgili dal yürütülür ve cümlenin işlevi biter. Eğer birden fazla koşul doğru olsa bile, sadece ilk karşılaşılan doğru dal çalışır.

4.7 İçiçe koşul deyimleri

Bir koşul deyimi bir başka koşul deyimiyle **içiçe** olabilir. Üç kısma bölünen örneğimizi aşağıdaki gibi yazabilirdik:

```
if x == y:
    print x, "ve", y, "eşittir"
```

```
else:
    if x < y:
        print x, "küçüktür", y
    else:
        print x, "büyüktür", y
```

Dışarıdaki koşul deyimi iki dal içermektedir. İlk dal bir adet basit bir görüntüleme cümlesi içerir. İkinci dal başka bir if cümlesi içerir ve bu cümle iki dal barındırır. Bu iki dalda basit görüntüleme cümleleridir, bunlar ayrı koşul cümleleri de olabilirlerdi.

Her ne kadar cümlelerin girintisi yapıyı düzgün bir şekilde gösterse de, içiçe koşul deyimleri hızlı bir şekilde okunması zor deyimlerdir. Genellikle bu tür deyimlerden mümkün olduğunca kaçınmak gerekir.

Mantıksal işleçler içiçe koşul cümlelrini basitleştirmek için bir yöntem sağlarlar. Örneğin aşağıdaki kodu tek bir koşul cümlesiyle yazabilirsiniz:

```
if 0 < x:
    if x < 10:
        print "x pozitif ve tek basamaklıdır."
```

print cümlesi sadece iki koşulu da geçersek çalışır, bu yüzden bu örnekte and işlecini kullanabiliriz:

```
if 0 < x and x < 10:
    print "x pozitif ve tek basamaklıdır."
```

Bu tip koşullar sık kullanılır ve Python bunlar için matematiksel gösterime benzeyen alternatif biz sözdizimi sağlar:

```
if 0 < x < 10:
    print "x pozitif ve tek basamaklıdır."
```

Bu koşul cümlesi yukarıdaki bileşik boolean deyimi ve içiçe koşul cümlesiyle aynı anlambilimine sahiptir.

4.8 Geri dönüş (return) cümlesi

return cümlesi bir fonksiyonun sona ulaşmadan bitirilmesini sağlar. Bu cümleyi kullanmanın bir nedeni bir hata durumuyla karşılaşmanız olabilir:

```
def print_square_root(x):
    if x <= 0:
        print "Sadece pozitif sayılar lütfen."
        return

    result = x**0.5
    print "x'in kare kökü", result
```

print_square_root fonksiyonu x isimli bir parametreye sahiptir. İlk yaptığı işlem bu parametrenin (x) 0'dan küçük veya eşit olduğunu sınamaktır. Eğer 0 veya daha küçük ise (negatif) bir hata mesajı verip fonksiyonu sonlandırıp ve çağrıldığı yere dönmektedir, bunu return cümlesi ile yapmaktadır. Yürütme akışı hemen çağıran yere geri döner ve fonksiyonun geri kalan satırları çalıştırılmaz.

4.9 Klavye girdisi

2. Bölümde Python tarafından sağlanan ve klavyeden girdi almamızı sağlayan fonksiyonlarını görmüştük: raw_input ve input. Şimdi bu fonksiyonları derinlemesine inceleyeceğiz.

Bu fonksiyonların herhhangi birisi çağrıldığında, program durup kullanıcının klavyeden bir şey girmesini bekler. Kullanıcı Giriş (Enter) tuşuna bastığında program çalışmasına devam eder ve raw_input kullanıcının girdiğini karakter dizisi (string) olarak döndürür:

```
>>> my_input = raw_input()
Ne için bekliyorsun?
>>> print my_input
Ne için bekliyorsun?
```

raw_input fonksiyonunu çağırmadan önce, kullanıcıya ne girmesi gerektiğini açıklayan bir mesaj göstermek faydalıdır. Bu mesaja **bilgi istemi (prompt)** adı verilir. raw_input fonksiyonuna bilgi istemini argüman olarak aktarabiliriz:

```
>>> name = raw_input("İsmin nedir? ")
İsmin nedir? Arthur, İngilizlerin kralı!
>>> print name
Arthur, İngilizlerin kralı!
```

Farkettiğiniz üzere bilgi istemi bir karakter dizisidir, bu yüzden tırnak işaretleri arasında alınmalıdır.

Eğer kullanıcının yanıtının tamsayı olmasını istiyorsak, yanıtı Python deyimi olarak değerlendiren input fonksiyonunu kullanabiliriz:

```
prompt = "Yüksüz bir kırlangıcın havadaki hızı nedir?\n"
speed = input(prompt)
```

Eğer kullanıcı rakamlardan oluşan bir karakter dizisi girdiğinde, bu girdi bir tamsayıya dönüştürülüp speed değişkenine atanacaktır. Ancak eğer kullanıcı geçerli bir Python deyimi girmezse program hata verecektir:

```
>>> speed = input(prompt)
Yüksüz bir kırlangıcın havadaki hızı nedir?
Neyi kastediyorsun, Afrika mı yoksa Avrupa kırlangıcı mı?
...
SyntaxError: invalid syntax
```

Son örnekte kullanıcı eğer ifadesinin etrafına tırnak işaretlerini koyup geçerli bir Python deyimini yanıt olarak verseydi, bir hata oluşmayacaktı:

```
>>> speed = input(prompt)
Yüksüz bir kırlangıcın havadaki hızı nedir?
"Neyi kastediyorsun, Afrika mı yoksa Avrupa kırlangıcı mı?"
>>> speed
'Neyi kastediyorsun, Afrika mı yoksa Avrupa kırlangıcı mı?'
>>>
```

Bu tür hatalardan uzak durmak için karakter dizisini almak üzere raw_input fonksiyonunu kullanmak ve diğer tiplere dönüşümü komutları kullanarak yapmak daha uygun bir yöntemdir.

4.10 Tip dönüşümü

Her Python tipi için tanımlı bu tipi diğer tiplere dönüştürmeye yarayan bir yerleşik komut vardır. Örneğin int(ARGUMAN) komutu, herhangi bir değeri alıp eğer yapılabiliyorsa tamsayıya (int) dönüştürür veya hata mesajı üretir:

```
>>> int("32")
32
>>> int("Merhaba")
ValueError: invalid literal for int() with base 10: 'Merhaba'
```

int ayrıca kayan noktalı değerleri tamsayılara çevirebilir, fakat unutulmaması gereken bu çevirme işleminde kesir kısmını kaldıracağıdır:

```
>>> int(-2.3)
-2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

float(ARGUMAN) komutu tamsayı ve karakter dizilerini kayan noktalı sayılara çevirir:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

Python'un tamsayı olan 1 ve kayan noktalı sayı olan 1.0'ı birbirinden farklı değerlendirmesi garip gelebilir. İkisi aynı değeri temsil edebilir ama farklı tiptedirler. Bunun nedeni bilgisayarda farklı şekillerde temsil edilmeleridir.

str(ARGUMAN) herhangi bir argümanı karakter dizisine (string) çevirir:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

str(ARGUMAN) herhangi bir değerle çalışabilir ve o değeri karakter dizisine çevirir. Daha önce de bahsedildiği gibi True boolean bir değerdir, ancak true değildir.

Boolean değerler için durum özellikle ilgi çekicidir:

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool("Ni!")
True
>>> bool("")
False
>>> bool(3.14159)
True
>>> bool(0.0)
False
```

Python boolean değerleri diğer tiplerin değerlerine atar. Sayısal değerler için (tamsayı ve kayan noktalı sayılar) sıfır değerler yanlıştır ve diğer değerler doğrudur. Karakter dizileri için boş diziler yanlıştır, diğerleri ise doğrudur.

4.11 Gasp

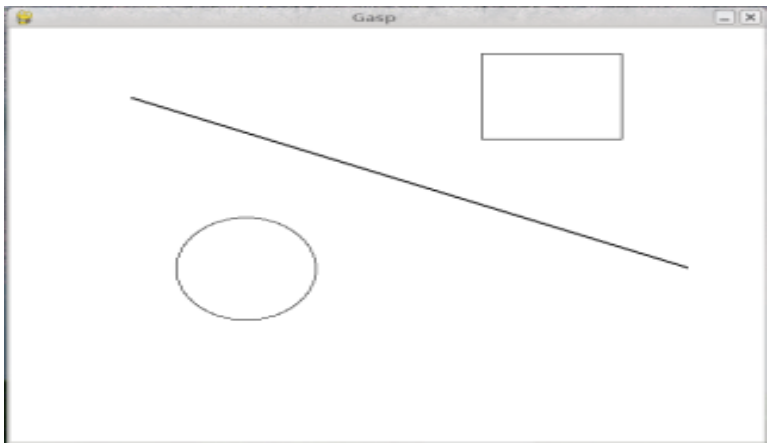
Gasp (**G**raphics **A**PI for **S**tudents of **P**ython - Python için Grafik Uygulama Geliştirme Arayüzü) grafiksel öğeler içeren programlar yazmanızı sağlar.

GASP'ı başlatmak için, aşağıdakileri deneyin:

```
>>> from gasp import *
>>> begin_graphics()
>>> Circle((200, 200), 60)
Circle instance at (200, 200) with radius 60
```

```
>>> Line((100, 400), (580, 200))
Line instance from (100, 400) to (590, 250)
>>> Box((400, 350), 120, 100)
Box instance at (400, 350) with width 120 and height 100
>>> end_graphics()
>>>
```

Son komuttan önce, tuvali kapatan komut, şu şekilde bir grafik penceresi görmeniz lazım:



GASP'ı bilgisayar programlama kavramlarını görselleştirmek ve öğrenirken eğlenmek için kullanacağız.

4.12 Sözlük

modül işleci:
Yüzde işaretiyle temsil edilen (%) ve tamsayılar üzerinde çalışıp, birbirine bölünen iki sayının bölmede kalanını üreten işleçtir.

boolean değer:
Sadece iki adet boolean değer vardır: True ve False. Boolean değerler Python yorumlayıcısı tarafından değerlendirilen bir boolean deyim sonucunda oluşurlar. bool tipindedirler.

boolean deyim:
Sonucu doğru veya yanlış olan deyim

karşılaştırma işleci:
İki değeri karşılaştıran işleçlerdir: ==, !=, >, <, >=, ve <=.

mantıksal işleç:
Boolean deyimleri birleştiren işleçlerdir: and, or, ve not.

koşul cümlesi:
Koşullara göre yürütme akışını kontrol eden cümlelerdir. Python'da if, elif, ve else anahtar kelimeleri koşul cümleleri için kullanılmaktadır.

koşul:

Koşul cümlesindeki boolean deyimdir, ve hangi dalın (yolun) yürütüleceğine karar verir.

blok:
Aynı girintiye sahip ardışık cümleler grubudur.

gövde:
Bileşik cümledeki başlığı takip eden cümle bloğudur.

dal:
Koşul cümlesi sonucu belirlenen yürütme akışındaki olası yollardan biridir.

zincirleme koşul ifadesi:
İkiden fazla yürütme akışı olasılığına sahip koşul dallanmasıdır. Python'da zincirleme koşul ifadeleri if ... elif ... else cümleleri şeklinde yazılmaktadır.

içiçe geçme:
Bir program yapısının bir başka program yapısı içerisinde bulunmasıdır, örnek olarak bir koşul cümlesinin bir başka koşul cümlesi dallanması içerisinde bulunmasını verebiliriz.

bilgi istemi:
Kullanıcıya veri girmesini söyleyen görsel ipuçu

tip dönüşümü:
Bir tipteki değeri alıp başka bir tip değerine dönüştüren cümle.

4.13 Alıştırmalar

1. Aşağıdaki nümerik ifadeleri kafanızdan değerlendirmeye çalışın, daha sonra Python yorumlayıcısı yardımıyla bulduğunuz sonuçları karşılaştırın:

    1. >>> 5 % 2
    2. >>> 9 % 5
    3. >>> 15 % 12
    4. >>> 12 % 15
    5. >>> 6 % 6
    6. >>> 0 % 7
    7. >>> 7 % 0

    Son örnekte ne oldu? Neden? Bilgisayarın yanıtı sizi tatmin ettiyse ve anlayabildiyeseniz ilerleyebilirsiniz. Eğer böyle bir şey sözkonusu değilse kendi kendinize örnekleri yapmanız gerekir. Modül işlecini anladığınıza emin olana kadar araştırmaya devam etmelisiniz.

2.
```
if x < y:
    print x, "küçüktür", y
elif x > y:
    print x, "büyüktür", y
else:
    print x, "ve", y, "eşittir"
```

Bu kodu compare(x,y) fonksiyonu içine koyun(wrap). compare fonksiyonunu üç kere, ilk argümanın küçük olduğu, ilk argümanın büyük olduğu ve iki argümanın eşit olduğu, çağırın.

3. Boolean deyimleri daha iyi anlamak için, doğruluk tabloları üretmek yararlıdır. İki boolean deyim ancak ama ancak aynı doğruluk tablosuna sahipse mantıksal olarak eşittir.

   Aşağıdaki Python betiği p ve q değişkenlerine sahip herhangi bir boolean ifadesinin doğruluk tablosunu görüntüler:

```
expression = raw_input("p ve q değişkenleriyle tanımlanmış
herhangi bir boolean ifade giriniz : ")

print " p      q      %s"  % expression
length = len( " p      q      %s"  % expression)
print length*"="

for p in True, False:
    for q in True, False:
        print "%-7s %-7s %-7s" % (p, q, eval(expression))
```

   Bu betiğin nasıl çalıştığını ileriki bölümlerde öğreneceksiniz. Şimdilik bu betiği boolean ifadeleri anlamak için kullanacağız. Bu programıp_and_q.py isimli bir dosyaya yazıp kaydedelim, daha sonra komut satırından çalıştıralım ve p or q ifadesini bilgi isteminde bulunduğu zaman verelim. Aşağıdaki gibi bir çıktıyla karşılaşmanız gerekiyor:

```
 p      q      p or q
====================
True    True    True
True    False   True
False   True    True
False   False   False
```

   Nasıl çalıştığını anlamak için, kullanımı kolaylaştırmak üzere bir fonksiyon içine yazalım:

```
def truth_table(expression):
    print " p      q      %s"  % expression
    length = len( " p      q      %s"  % expression)
    print length*"="

    for p in True, False:
        for q in True, False:
            print "%-7s %-7s %-7s" % (p, q, eval(expression))
```

   Bu betiği bir Python kabuğunda içeri aktabilir ve truth_table fonksiyonunu kendi boolean ifademizi (p ve q değişkenleriyle oluşturulmuş) içeren bir karakter dizisi argümanı ile çağırabiliriz:

```
>>> from p_and_q import *
>>> truth_table("p or q")
 p      q      p or q
====================
True    True    True
True    False   True
False   True    True
False   False   False
>>>
```

truth_table fonksiyonunu aşağıdaki deyimlerle kullanıp, her seferinde üretilen doğruluk tablosunu inceleyin:

1. not(p or q)
2. p and q
3. not(p and q)
4. not(p) or not(q)
5. not(p) and not(q)

Yukarıdaki deyimlerden hangileri mantıksal olarak eşittir?

4. Aşağıdaki deyimleri Python kabuğuna giriniz:

```
True or False
True and False
not(False) and True
True or 7
False or 7
True and 0
False or 8
"mutlu" and "üzgün"
"mutlu" or "üzgün"
"" and "üzgün"
"mutlu" and ""
```

Sonuçları çözümleyin. Farklı tiplerin değerleri ve mantıksal işleçler ile ilgili olarak ne tür gözlemleriniz var? Bu gözlemleri basit and ve ordeyimleri şeklindeki kurallar halinde yazabilir misiniz?

```
5.if choice == 'a':
    function_a()
  elif choice == 'b':
    function_b()
  elif choice == 'c':
    function_c()
  else:
    print "Hatalı seçim."
```

Bu kodu dispatch(choice) fonksiyonu içerisine yerleştirelim. Daha sonra function_a, function_b ve function_c fonksiyonlarını yazarak bu fonksiyonların çağrıldığını ekranda gösterelim. Örneğin:

```
def function_a():
    print "function_a was çağrıldı..."
```

Fonksiyonları (dispatch, function_a, function_b, ve function_c) ch4prob4.py isimli bir betik içerisine koyun. Bu betiğin en altınadispatch('b') çağrısını ekleyin. Programı çağırdığınızda çıktınız şu şekilde olmalıdır:

```
function_b çağrıldı...
```

Son olarak betiği değiştirerek kullanıcının 'a', 'b', veya 'c' girebileceği şekile getirin. Python kabuğunda içe aktararak betiği sınayın.

6. is_divisible_by_3 fonksiyonu yazın. Bu fonksiyon tek bir tamsayı argüman alsın ve eğer argüman üç ile bölünebiliyorsa "Bu sayı üç ile bölünebilir" mesajını ekranda göstersin. Eğer bölünemeyen bir sayı ise "Bu sayı üç ile bölünemez" mesajını ekranda göstersin.

   Benzer şekilde is_divisible_by_5 fonksiyonunu yazın.

7. Bir önceki alıştırmada yazdığınız fonksiyonları is_divisible_by_n(x,n) şeklinde iki tamsayı argüman alan ve ilk argümanın ikinci argüman tarafından bölünüp bölünemediğini sınamak üzere genelleştirin. Bu programı ch04e06.py dosyasında saklayın. Bir kabuğa aktarın ve sınayın. Örnek bir oturum şu şekilde olacaktır:

```
>>> from ch04e06 import *
>>> is_divisible_by_n(20, 4)
Yes, 20 is divisible by 4
>>> is_divisible_by_n(21, 8)
No, 21 is not divisible by 8
```

8. Aşağıdakinin çıktısı ne olacaktır?

```
if "Ni!":
    print 'We are the Knights who say, "Ni!"'
else:
    print "Stop it! No more of this!"

if 0:
    print "And now for something completely different..."
else:
    print "What's all this, then?"
```

   Ne olduğunu ve neden olduğunu açıklayın.

9. Aşağıdaki GASP betiği, house.py dosyasındadır, GASP tuvali üzerine basit bir ev çizer:

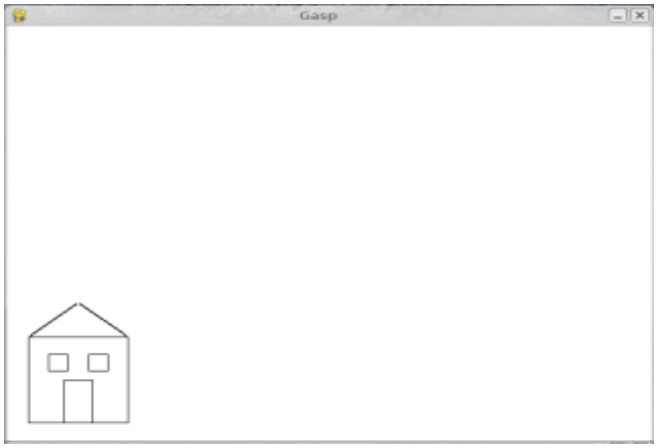
```
from gasp import *                  # import everything from the gasp
library

begin_graphics()                    # open the graphics canvas

Box((20, 20), 100, 100)             # the house
Box((55, 20), 30, 50)               # the door
Box((40, 80), 20, 20)               # the left window
Box((80, 80), 20, 20)               # the right window
Line((20, 120), (70, 160))          # the left roof
Line((70, 160), (120, 120))         # the right roof

pause()                             # keep the canvas open until a
key is pressed
end_graphics()                      # close the canvas (which would
happen anyway,
                                # since the script ends here, but it
is better
                                # to be explicit).
```

1. Bu betiği çalıştırıp aşağıdaki gibi bir pencere elde ediniz ve doğru çalıştığından emin olun:



2. Ev kodunu draw_house() isimli bir fonksiyon içerisine taşıyın.
3. Betiği çalıştırın. Evi görebiliyor musunuz? Neden?
4. Betiğin en altına draw_house() fonksiyon çağrısını ekleyin, böylece ev ekranda gözükmeye başlar
5. Fonksiyon x ve y parametreleriyle parametreleştirin -- başlık def draw_house(x, y): şekline gelecektir --, böylece evin tuval üzerinde çizileceği konumu argüman şeklinde vermiş olacaksınız.
6. draw_house fonksiyonunu tuvalde farklı konumlarda beş farklı ev çizmek için kullanın.

5. Ürün veren fonksiyonlar

5.1 Geri dönüş değerleri

Şimdiye kadar kullandığımız abs, pow ve max benzeri içsel olarak tanımlı fonksiyonlar sonuç ürettiler. Her bu fonksiyonları çağırmamız bir değer üretmektedir, bu üretilen değeri genellikle bir değişkene atama veya bir deyimin parçası haline getirdik.

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

Ancak şimdiye kadar kendi yazdığımız fonksiyonların hiçbiri bir değer üretip, döndürmedi.

Bu bölümde değer döndüren fonksiyonlar yazacağız, bunlara da **ürün veren (ürünlü) fonksiyonlar** adını, daha iyi anlamlı bir isim gereksinimden dolayı, vereceğiz. İlk örneğimiz area örneği, bu fonksiyon yarıçapı verilen bir dairenin alanını bize döndürecek:

```
def area(radius):
    temp = 3.14159 * radius**2
    return temp
```

return cümlesini daha önce görmüştür, ancak ürün veren bir fonksiyonda return cümlesi bir **geri dönüş değeri** içerir. Bu cümlenin anlamı:Hemen çağrıldığın yere geri dön ve devamımdaki ifadeyi geri dönüş değeri olarak kullandır. Deyim isteğimize bağlı olarak karmaşık olabilir, örneğin yukarıdaki fonksiyonu aşağıdaki gibi de yazabiliriz:

```
def area(radius):
    return 3.14159 * radius**2
```

Ancak, diğer taraftan temp gibi **geçici değerleri** kullanmak hata ayıklamayı kolaylaştırır.

Bazı durumlarda birden fazla geri dönüş cümlesine sahip olmak faydalıdır, her bir koşullu dallanmada bir geri dönüş yapılabilir. Bunu daha önce içsel tanımlı abs fonksiyonunda görmüştük, şimdi kendi örneğimizi yazalım::

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

return cümleleri farklı alternatif koşullarda olduğu için, aralarından sadece biri yürütülecektir. Biri yürütülür yürütülmez fonksiyon geri kalan cümlelerini çalıştırmadan sonlanacaktır.

Yukarıdaki fonksiyonu yazmanın başka bir yolu da else'i kaldırıp koşulunu ikinci return cümlesi olarak yazmaktır:

```
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

Yukarıdaki örnek hakkında düşünüp, aynen ilki gibi çalıştığına kendinizi ikna edin.

return cümlesinden sonra gelen herhangi bir kod veya yürütme akışı açısından erişilmez durumda olan kodlar **ölü kod** adını almaktadır.

Bir ürünlü fonksiyonda olası her yolda bir return cümlesinin olması iyi bir fikirdir. Aşağıdaki absolute_value sürümü bunu gerçekleştirmemektedir:

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

Bu sürüm doğru bir sürüm değildir çünkü x 0 olduğunda hiç bir koşul doğru olmayacağı için, fonksiyon herhangi bir return cümlesine rastlamayacaktır. Bu durumda geri dönüş değeri özel bir değer olan **None** olacaktır:

```
>>> print absolute_value(0)
None
```

None, NoneType tipinin tekil değeridir.

```
>>> type(None)
<type 'NoneType'>
```

Tüm Python fonksiyonları başka bir değer döndürmüyorlarsa, None değerini döndürürler.

5.2 Program geliştirme

Bu noktada, tamamlanmış fonksiyonlara bakarak ne yaptıklarını anlayabilmeniz ve anlatabilmeniz gerekmektedir. Ayrıca, eğer alıştırmaları yapıyorsanız, bazı küçük fonksiyonlar da yazmış olmanız gerekiyor. Daha büyük fonksiyonlar yazdıkça zorlanmaya başlarsınız, özellikle çalışma zamanı ve anlambilimsel hatalar artmaya başlar.

Artan bir şekilde karmaşık programlarla başa çıkmak için, bir teknik önereceğiz. Bu tekniğe **arttırımsal geliştirme** adı verilmektedir. Arttırımsal geliştirmenin

hedefi uzun hata ayıklama süreçlerini kısaltmak için bir anda küçük bir kod parçası eklemek ve bu parçayı sınamaktır.

Örnek olarak iki nokta arasındaki uzaklığı bulmak istediğinizi varsayalım. Koordinatları verilmiş olan (x_1,y_1) ve (x_2,y_2) noktalarının arasındaki uzaklığı Pisagor teoremine göre şu şekilde hesaplarız:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

İlk adım distance fonksiyonunun Python'da nasıl gözükeceğini düşünmektir. Başka bir ifadeyle, girdiler (parametreler) nedir ve çıktı (geri dönüş değeri) nedir sorularına cevap bulmaktır?

Bu örnekte, iki nokta girdilerimizdir ve bu girdileri dört parametre ile temsil edebiliriz. Geri dönüş değerimiz uzaklıktır, bu da kayan noktalı bir sayıdır.

Bu cevaplardan sonra fonksiyonumuzun ana hatlarını yazabiliriz:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Açıkça görülmektedir, fonksiyonumuzun bu sürümü uzaklıkları hesaplamamaktadır; her zaman sıfır değerini döndürecektir. Ancak sözdizimsel olarak doğrudur ve çalışabilir. Bunun anlamı bu fonksiyonu daha fazla karmaşıklaştırmadan önce sınayabiliriz.

Yeni fonksiyonu sınamak için, örnek değerlerle çağıracağız:

```
>>> distance(1, 2, 4, 6)
0.0
```

Bu değerleri seçmemizin nedeni yatay uzaklığın 3, dikey uzaklığın 4 olması sonucu iki nokta arasındaki uzaklığın 5 olmasıdır (3-4-5 üçgeninde hipotenüs). Fonksiyonu sınarken, doğru cevabı bilmek yararlıdır.

Bu noktada fonksiyonun sözdizimsel olarak doğru olduğunu onaylamış olduk, yeni kod satırları ekleyebiliriz. Her bir arttırımsal değişiklikten sonra fonksiyonu tekrar sınarız. Eğer herhangi bir noktada hata ile karşılaşırsak bulmamız kolay olacaktır. En son eklediğimiz kod olmalıdır (satır satır ilerlersek hatayı en son eklediğimiz satır yaratmıştır diyebiliriz).

Hesaplamadaki mantıksal ilk adım x_2 - x_1 ve y_2 - y_1 farklarını bulmaktır. Bu değerleri geçici değişkenlerde (dx, dy) saklarız ve ekranda görüntüleriz.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print "dx is", dx
    print "dy is", dy
    return 0.0
```

Eğer fonksiyon çalışıyorsa, çıktılar 3 ve 4 olmalıdır. Eğer sonuç böyleyse fonksiyon parametreleri doğru alıp, ilk hesaplamayı doğru yapıyordur. Eğer bir hata varsa kontrol edilecek satır sayısı azdır.

Daha sonra dx ve dy değişkenlerinin karelerini toplarız:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print "dsquared is: ", dsquared
    return 0.0
```

Dikkat ederseniz print cümlelerini kaldırdık. Bu tip kodlara **iskele (scaffolding)** adı verilmektedir, son ürünümüzün bir parçası değildir ancak programı yazarken yararlıdır.

Bu aşamada programı tekrar çalıştırıp çıktıyı kontrol ederiz (25 olmalıdır).

Son olarak, kesirsel üs 0.5'i kullanarak kare kökü hesaplar ve sonuç olarak geri döndürürüz:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result
```

Eğer bu doğru çalışırsa, işimiz bitmiştir. Çalışmazsa, result değişkenini geri döndürmeden önce ekranda görüntülememiz gerekebilir.

Başladığınızda, bir anda sadece bir iki satırlık kod eklemeniz gerekir. Deneyim kazandıkça kendinizi daha büyük parçalar yazıp, sınayıp, hata ayıklar durumda bulacaksınız. Her şekilde, arttırımlı geliştirme süreci sizi sıkıntılı fazla hata ayıklama zamanından kurtaracaktır.

Sürecin anahtar hatları şunlardır:

1. Çalışan bir programla başlayıp küçük artımlı değişikler yapın. Herhangi bir noktada bir hata varsa hatanın nerede olduğunu bileceksiniz.
2. Ara değerleri tutmak için geçici değişkenler kullanın böylece bu değerleri ekran gösterip kontrol edebilirsiniz.
3. Program çalışır hale gelince, iskelet kodlarının bazılarını kaldırmanız ve birden fazla cümleyi bileşik deyimler haline getirmeniz gerekebilir, elbette bu birleştirme işlemini programın okunabilirliğini zorlaştırmıyorsa yapmanız gerekir.

5.3 Kompozisyon

Şimdiye kadar farkedeceğiniz gibi, bir fonksiyonu bir başka fonksiyondan çağırabilirsiniz. Bu yeteneğe **kompozisyon** (bileşim) adı verilmektedir.

Örnek olarak, iki nokta alan (dairenin merkezi ve çevrenin üzerinde bir nokta) bir fonksiyon yazıp dairenin alanını hesaplayacağız.

Merkez noktanın xc ve yc değişkenlerinde, ve çevre üzerindeki noktada xp ve yp değişkenlerinde saklandığını varsayalım. İlk adım dairenin yarıçapını bulmaktır. Bu da verilen iki nokta arasındaki uzaklıktır. Daha önce yazmış olduğumuz distance fonksiyonunu bu iki nokta arasındaki uzaklığı hesaplamak için kullanabiliriz. Ne de olsa bu iş yazılmış, sadece kullanmamız yeterli:

```
radius = distance(xc, yc, xp, yp)
```

İkinci adım ise bulduğumuz bu yarıçapı kullanarak dairenin alanını hesaplayıp geri döndürmektir. Yine daha önce yazdığımız fonksiyonları kullanacağız:

```
result = area(radius)
return result
```

Bu kodları bir fonksiyon içine yazarsak:

```
def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Bu fonksiyona area2 adını area fonksiyonundan ayırtetmek için kullandık. Bir modül içerisinde bir isimde sadece tek bir fonksiyon olabilir. Aynı modülde aynı isimden fonksiyonlar yeralamaz.

Geçici radius ve result değişkenleri geliştirme ve hata ayıklama için yararlıdır, ancak fonksiyon çağrılarını birleştirirsek daha kısa ve öz bir ifade elde ederiz:

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

5.4 Boolean fonksiyonlar

Fonksiyonlar boolean değerler de döndürebilir, bu kullanışlı karmaşık sınamaları fonksiyonların içerisine saklamak için yararlıdır. Örneğin:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Bu fonksiyonun ismi is_divisible'dır. **Boolean fonksiyon**lara evet/hayır cevapları gibi isimler vermek yaygındır. is_divisible, x'in y tarafından bölünebilir olup olmadığına dair True veya False döndürecektir.

Fonksiyonu if cümlesinin de boolean bir deyim olduğu gerçeğinden yararlanarak daha kısa ve öz bir hale getirebiliriz. Doğrudan deyimi döndürerek, if cümlesinin kendisinden kurtulabiliriz:

```
def is_divisible(x, y):
    return x % y == 0
```

Aşağıda bu yeni fonksiyonun nasıl kullanıldığını görebilirsiniz:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Boolean fonksiyonlar genellikle koşul cümlelerinde kullanılır:

```
if is_divisible(x, y):
    print "x y tarafından bölünebilir"
else:
    print "x y tarafından bölünemez"
```

Aşağıdaki gibi bir şey yazmak baştan çıkarıcı olabilir:

```
if is_divisible(x, y) == True:
```

Ancak ek karşılaştırma tamamen gereksizdir.

5.5 function tipi

Fonksiyonlar Python'da farklı bir tiptir. Bu tip int, float, str, bool ve NoneType'i kapsar.

```
>>> def func():
...     return "fonksiyon func çağrıldı..."
...
>>> type(func)
<type 'function'>
>>>
```

Diğer tiplerde olduğu gibi, fonksiyonlar başka fonksiyonlara argüman olarak geçirilebilir:

```
def f(n):
    return 3*n - 6

def g(n):
```

```
    return 5*n + 2

def h(n):
    return -2*n + 17

def doto(value, func):
    return func(value)

print doto(7, f)
print doto(7, g)
print doto(7, h)
```

doto üç kere çağrılmıştır. 7 her sefer için fonksiyonun value argümanıdır; f, g ve h func argümanı olarak geçirilmektedir. Bu betiğin çıktısı şu şekildedir:

```
15
37
3
```

Bu örnek her ne kadar yapmacık bir örnek olsa da, ileride bir fonksiyona fonksiyonun geçirilmesinin yararlı olduğu durumlar karşımıza çıkacaktır.

5.6 Program yazım kuralları

Okunabilirlik programcılar açısında çok önemlidir, çünkü pratikte programların yazılmasından ziyade okunup değiştirildikleri daha sık karşılaşılan bir durumdur. Bu kitaptaki tüm örnekler Python geliştirme önerisi 8 (PEP 8), Python topluluğu tarafından geliştirilen yazım rehberi ile tutarlı olacaktır.

Yazım kuralları konusunda programlarımız karmaşıklaştıkça söyleyeceklerimiz artacaktır, ancak şimdiden bir kaç noktayı açıklamak yararlı olacaktır:

- girintiler için 4 boşluk karakteri kullanın
- içe aktarmalar (import) dosyaların en üstünde yer almalıdır
- fonksiyon tanımlamaları arasında iki boş satır olmalıdır
- fonksiyon tanımlamaları bir arada olmalıdır
- en üst seviye cümleler, fonksiyon çağrımları da dahil olmak üzere, birlikte programın en altında tutun

5.7 Üçlü tırnaklı karakter dizileri

İlk olarak 2. Bölümde gördüğümüz tek ve çift tırnaklı karakter dizilerine ek olarak, üç tırnaklı karakter dizileri'da Python'da vardır:

```
>>> type("""Bu bir üç çift tırnaklı karakter dizisidir""")
<type 'str'>
>>> type('''Bu bir üç tek tırnaklı karakter dizisidir''')
<type 'str'>
>>>
```

Üç tırnaklı karakter dizileri hem tek hem de çift tırnakları içerisinde barındırabilir:

```
>>> print '''"Oh hayır", diye bağırdı, "Ben'in bisikleti bozuldu!"'''
"Oh hayır", diye bağırdı, "Ben'in bisikleti bozuldu!"
>>>
```

Son olarak, üç tırnaklı karakter dizileri bir kaç satıra bölünebilir:

```
>>> message = """Bu mesaj
... bir kaç satıra
... bölünecektir."""
>>> print message
Bu mesaj
bir kaç satıra
bölünecektir.
>>>
```

5.8 doctest ile birim sınama (unit test)

Bu günlerde yazılım geliştirmede kaynak kodun otomatik **birim sınama**sını yapmak yaygın bir pratiktir. Birim sınama, fonksiyonlar gibi bağımsız kod parçalarının otomatik olarak doğru çalıştığını onaylamak için bir yol sağlar. Bu daha sonra fonksiyonun gerçekleştirimini değiştirmeyi ve yine de bekleneni yapmasını olanaklı kılar.

Python doctest isimli kolay birim sınama için bir modül barındırır. Doctestler fonksiyon gövdesinin veya betiğin ilk satırında üç tırnaklı karakter dizileri içerisinde yazılabilir. Bunlar bir Python bilgi istemine girdileri ve beklenen çıktıyı örnekleyen yorumlayıcı oturumları şeklindedir.

doctest modülü >>> ile başlayan herhangi bir cümleyi otomatik olarak çalıştırarak, takip eden cümleyi yorumlayıcının çıktısıyla karşılaştırır.

Bunun nasıl çalıştığını görmek için aşağıdaki kodları myfunctions.py isimli bir betiğe koyup deneyiniz:

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
    """


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Son üç satır doctestin çalışmasını sağlayan satırlardır. Bu satırları doctestleri içeren herhangi bir dosyanın en altına koymalısınız. Nasıl çalıştıklarını 10. bölümde modülleri anlatırken açıklayacağız.

Yukarıdaki betiği çalıştırmanız aşağıdaki çıktıyı üretecektir:

```
$ python myfunctions.py
**********************************************************************
*
File "myfunctions.py", line 3, in __main__.is_divisible_by_2_or_5
Failed example:
    is_divisible_by_2_or_5(8)
Expected:
    True
Got nothing
**********************************************************************
*
1 items had failures:
   1 of   1 in __main__.is_divisible_by_2_or_5
***Test Failed*** 1 failures.
$
```

Yukarıdaki bir çöken test örneğidir. Bu test "eğer is_divisible_by_2_or_5(8)'i çalıştırırsanız sonuç True olmalıdır" demektedir. Yazılmış olanis_divisible_by_2_or_5 fonksiyonu herhangi bir şey döndürmediği için, test çöker ve doctest bize beklenenin True olduğunu ama bir şey dönmediğini bildirir.

Bu testi, fonksiyonu True döndürecek şekilde yazarak geçerli hale getirebiliriz:

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
    """
    return True


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Şimdi çalıştırdığımız, herhangi bir çıktı olmayacaktır. Bunun anlamı tüm testlerin başarılı olduğudur. Tekrar etmek gerekir ki, doctest karakter dizileri hemen fonksiyon tanımlaması başlığından sonra yazılmalıdır.

Daha ayrıntılı çıktı görebilmek için betiği -v komut satırı seçeneğiyle çalıştırabilirsiniz:

```
$ python myfunctions.py -v
Trying:
```

```
    is_divisible_by_2_or_5(8)
Expecting:
    True
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.is_divisible_by_2_or_5
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
$
```

Her ne kadar testimiz geçerli olsa da, bizim yazdığımız testler yetersizdir. Çünkü is_divisible_2_or_5 herşey için True döndürmektedir. Aşağıda daha kapsamlı testler içeren ve bu testleri geçerli hale getiren tamamlanmış bir sürüm görebilirsiniz:

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
      >>> is_divisible_by_2_or_5(7)
      False
      >>> is_divisible_by_2_or_5(5)
      True
      >>> is_divisible_by_2_or_5(9)
      False
    """
    return n % 2 == 0 or n % 5 == 0


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Bu betiği -v komut satırı seçeneğiyle çalıştırın ve sonucu inceleyin.

5.9 Sözlük

ürün veren fonksiyon:
Geri dönüş değeri üreten fonksiyondur.

geri dönüş değeri:
Bir fonksiyon çağrısı sonucu olarak sağlanan değer.

geçici değişken:
Karmaşık hesaplamalarda ara değerleri saklamak için kullanılan değişken.

ölü kod:

Programların, genellikle return cümlesinden sonra geldiği için hiç bir zaman çalıştırılamaz olan parçaları.

None:
Geri dönüş cümleleri içermeyen veya argümansız geri dönüş cümleleri içeren fonksiyonlar tarafından döndürülen özel Python değeri. NoneNoneType'ın tek değeridir.

arttırımlı geliştirme:
Hata ayıklamayı kolaylaştırmak ve mümkün olduğunca azaltmak için yapılan ve bir anda az kod ekleme ve bu ekleneni hemen sınama şeklinde gerçekleştirilen bir programlama geliştirme yöntemidir.

iskele:
Program geliştirmede kullanılan ama son program sürümünün parçası olmayan kod.

boolean fonksiyon:
Boolean değer döndüren fonksiyon.

kompozisyon (fonksiyonların):
Bir fonksiyon gövdesinden başka bir fonksiyon çağırmak veya bir fonksiyonun geri dönüş değerini diğerinin çağrımına argüman olarak aktarmak.

birim sınama:
Bağımsız kod parçalarını doğrulamak için kullanılan otomatik yordamlar. Python doctest modülünü bu amaç için barındırmaktadır.

5.10 Alıştırmalar

Buradaki tüm alıştırmalar ch05.py isimli aşağıdaki içeriğe sahip dosyaya eklenmelidir:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Her alıştırmayı bitirdikten sonra, programı çalıştırarak yeni fonksiyonunuzun doctestleri geçerli kıldığını onaylayınız.

1. İki değeri birbiriyle karşılaştıran ve a > b ise 1, a == b ise 0 ve a < b ise -1 değerlerini döndüren compare isminde bir fonksiyon yazınız.

```
def compare(a, b):
    """
      >>> compare(5, 4)
      1
      >>> compare(7, 7)
      0
      >>> compare(2, 3)
      -1
      >>> compare(42, 1)
      1
```

```
    """
    #  Fonksiyon govdesi burada baslamalidir
```

Fonksiyon gövdesini doldurarak dostestlerin geçerliliğini sağlayın.

2. Arttırımsal geliştirme yöntemini kullanarak iki kenarının uzunluğu verilen bir dik üçgenin hipotenüsünü bulan hypotenuse isimli bir fonksiyon yazın. İlerledikçe yaptığınız arttırımlı geliştirme sürecinin her aşamasını kaydedin.

```
def hypotenuse(a, b):
    """
      >>> hypotenuse(3, 4)
      5.0
      >>> hypotenuse(12, 5)
      13.0
      >>> hypotenuse(7, 24)
      25.0
      >>> hypotenuse(9, 12)
      15.0
    """
```

Kodu geliştirmeyi tamamladıktan sonra, oluşan kodunuzu aşağıdaki doctestlerle birlikte ch05.py dosyası içerisine kaydedin ve doctestlerin geçerliliğini doğrulayın.

3. (x1,y1) ve (x2,y2) noktalarından geçen doğrunun eğimini hesaplayan slope fonksiyonunu yazın. Gerçekleştiriminizin aşağıdaki doctestleri geçerli yaptığını doğrulayın:

```
def slope(x1, y1, x2, y2):
    """
      >>> slope(5, 3, 4, 2)
      1.0
      >>> slope(1, 2, 3, 2)
      0.0
      >>> slope(1, 2, 3, 3)
      0.5
      >>> slope(2, 4, 1, 2)
      2.0
    """
```

Daha sonra slope fonksiyonuna intercept(x1,y1,x2,y2) fonksiyonu içerisinde bir çağrı yaparak, doğrunun (x1,y1) ve (x2,y2)noktalarından geçen y-kesişimini bulun.

```
def intercept(x1, y1, x2, y2):
    """
      >>> intercept(1, 6, 3, 12)
      3.0
      >>> intercept(6, 1, 1, 6)
```

```
        7.0
        >>> intercept(4, 6, 12, 8)
        5.0
    """
```

intercept fonksiyonu yukarıdaki doctestleri geçmelidir.

4. İsmi is_even(n) olan ve tamsayı argüman alan bir fonksiyon yazın. Bu fonksiyon argüman **çift sayı** ise True ve **tek sayı** ise Falsedöndürmelidir. Bu fonksiyona kendi doctestlerinizi de ekleyip doğrulamaları yapın.
5. Şimdi de is_odd(n) isimli, n tek sayı ise True, çift sayı ise False değer döndüren fonksiyonu yazınız. Fonksiyonu yazma sürecinde doctestleri de ekleyin. Daha sonra fonksiyonu girilen argümanı değerlendirmek için kullanmak üzere is_even fonksiyonunu çağıracak şekilde değiştirin.

```
6.def is_factor(f, n):
    """
        >>> is_factor(3, 12)
        True
        >>> is_factor(5, 12)
        False
        >>> is_factor(7, 14)
        True
        >>> is_factor(2, 14)
        True
        >>> is_factor(7, 15)
        False
    """
```

is_factor fonksiyonuna bir gövde yazarak yukarıdaki doctestlerin geçmesini sağlayın.

```
7.def is_multiple(m, n):
    """
        >>> is_multiple(12, 3)
        True
        >>> is_multiple(12, 4)
        True
        >>> is_multiple(12, 5)
        False
        >>> is_multiple(12, 6)
        True
        >>> is_multiple(12, 7)
        False
    """
```

is_multiple fonksiyonuna doctestleri geçebilmesi için gereken gövdeyi yazın. is_multiple fonksiyonu tanımlaması içerisinde is_factorfonksiyonunu kullanmanın bir yolunu bulabilir misiniz?

```
8.def f2c(t):
```

```
    """
      >>> f2c(212)
      100
      >>> f2c(32)
      0
      >>> f2c(-40)
      -40
      >>> f2c(36)
      2
      >>> f2c(37)
      3
      >>> f2c(38)
      3
      >>> f2c(39)
      4
    """
```

f2c fonksiyon tanımlaması için bir gövde yazın. Fonksiyon Fahrenheit olarak verilmiş olan sıcaklığı en yakın Celcius tamsayı değerini döndürecek şekilde çalışmalıdır. (ipucu: Python tarafından sağlanan round fonksiyonunu kullanmak isteyebilirsiniz. Python kabuğundaround.__doc__ yazarak ve iyice anlayana kadar round fonksiyonunu deneyerek denemeler yapabilirsiniz.)

```
9.def c2f(t):
    """
      >>> c2f(0)
      32
      >>> c2f(100)
      212
      >>> c2f(-40)
      -40
      >>> c2f(12)
      54
      >>> c2f(18)
      64
      >>> c2f(-48)
      -54
    """
```

c2f fonksiyonu gövdesini Celcius'tan Fahrenheit'a çevirim yapacak şekilde doldurun.

6. Yineleme

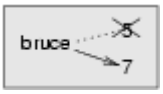
6.1 Birden fazla atama

Keşfetmiş olabileceğiniz gibi, aynı değişkene birden fazla atama yapılması geçerli bir yöntemdir. Yeni bir atama varolan değişkeninin yeni bir değeri temsil etmesini sağlar (ve önceki-eski değeri temsiliyetini ortadan kaldırır).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

Yukarıdaki programın çıktısı 5 7 şeklindedir, çünkü bruce değişkenini ilk görüntülememizden önce,değeri 5'tir, ve daha sonra ikinci görüntülemede 7 olmaktadır. İlk printcümlesindeki bruce değişkeninden sonra gelen virgül işareti yeni bir satır yaratılmasını engeller, bu yüzden iki print cümlesinin çıktısı da aynı satırdadır

Aşağıda **birden fazla atama**'nın bir durum diyagramında nasıl gözüktüğünü inceleyebilirsiniz:



Birden fazla atamalarda, atama işlemi ve eşitlik cümlesini birbirinden ayırmak oldukça önemlidir. Çünkü Python eşit işaretini (=) atama için kullanmaktadır. a=b şeklindeki bir cümleyi eşitlik olarak algılamak çekici gelebilir, ancak bu ifadenin eşitlik ifadesi olmadığı unutulmamalıdır!

İlk olarak, eşitlik simetriktir and atama değildir. Örneğin, matematikte, eğer a=7 ise 7=a'dır. Python'da a = 7 cümlesi geçerli bir ifade olmasına rağmen, 7 = a geçerli bir ifade değildir.

Ayrıca, matematikte eşitlik cümlesi her zaman doğrudur. Eğer şimdi a = b ise, daha sonra da a her zaman b'ye eşit olacaktır. Python'da, bir atama işleci iki değişkeni eşitleyebilir ancak sürekli bu şekilde kalmaları gerekli değil:

```
a = 5
b = a    # a ve b şimdi eşitler
a = 3    # a ve b artık eşit değiller
```

Üçüncü satır a değişkeninin değerini değiştirir ama b değişkeninin değerini değiştirmez, bu yüzden o satırın işletilmesinden sonra artık eşit değillerdir. (Bazı programlama dillerinde, karmaşayı önlemek için atama işleminde farklı bir işaret, <- veya := gibi, karmaşayı önlemek için kullanılmaktadır)

6.2 Değişkenleri güncelleme

Birden fazla atamanın en sık biçimi güncellemedir, bir değişkenin değerini eskisine bağlı olarak değiştirmektir.

```
x = x + 1
```

Bu ifadenin anlamı x'in değerini al, bu değere 1 ekle ve x'i yeni değerle güncelle

Eğer varolmayan bir değişkeni güncellemeye çalışırsanız, bir hata oluşur, çünkü Python atama işleminde öncelikle sağ taraftaki deyimi değerlendirdikten sonra oluşan sonucu sol taraftaki isme (değişkene) atar.:

```
>>> x = x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Bir değişkeni güncellemeden önce, o değişkeni **ilklemeniz** gerekir, bu da oldukça basit bir atamayla yapılabilir:

```
>>> x = 0
>>> x = x + 1
>>>
```

Bir değişkeni üzerine 1 ekleyerek güncelleme işlemine **artırma**; değerinden 1 çıkarma işlemine çıkarma adı verilir.

6.3 while cümlesi

Bilgisayarlar genellikle tekrarlayan görevleri otomatikleştirmek için kullanılmaktadır. Aynı veya benzer görevleri hatasız bir şekilde tekrarlama işlemi bilgisayarların iyi yaptığı, insanların zorlandığı bir şeydir.

Bir cümle kümesinin yinelemeli (tekrarlı) olarak yürütülmesi işlemine **yineleme (iteration)** adı verilir. Yineleme çok yaygın olduğu için, Python işlemi kolaylaştırmak için bir çok dil özelliği sağlamıştır. İnceleyeceğimiz ilk özellik while cümlesidir.

countdown isimli aşağıda tanımlanmış olan fonksiyon while cümlesinin nasıl kullanılacağını göstermektedir:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print "Yokol!"
```

while cümlesini neredeyse İngilizce bir cümleymiş gibi okuyabilir ve yaptığı iş anlayabilirsiniz. Cümlenin anlamı n değişkeni sıfırdan (0) büyük olduğu sürece

değerini ekranda görüntüle ve görüntülemeden sonra n'in değerini 1 azalt. Sıfır değerine ulaştığında (döngüden çıkma) ekranda Yokol! kelimesini görüntüle

Daha biçimsel olarak, aşağıda while cümlesinin yürütme akışını inceleyebilirsiniz:

1. Koşulu değerlendir, False veya True üret
2. Eğer koşul yanlış ise, while cümlesinden çıkıp bir sonraki satırdan yürütmeye devam et.
3. Eğer koşul doğru ise, cümlenin gövdesindeki her cümleyi çalıştır ve 1. adıma geri dön.

Gövde başlığın altında eşit girintiye sahip olan tüm cümleleri içermektedir.

Bu şekildeki akışa **döngü** adı verilmektedir, çünkü üçüncü adımda işlem başa dönmektedir. Eğer koşul ilk seferinde yanlış ise, döngünün içerisindeki cümleler hiç bir zaman çalıştırılmazlar.

Döngünün gövdesi bir veya daha fazla değişkenin değerini değiştirmelidir ki, koşul yanlışlanabilsin ve döngünün bitmesi garanti edilsin. Eğer koşul yanlışlanamazsa döngü sonsuza kadar çalışabilir, bu şekildeki döngülere **sonsuz döngü** adı verilir. Bir bilgisayar bilimcisi için örneğin, şampuanların üzerindeki köpürtün, durulayın, tekrarlayın işlemi başlı başına bir sonsuz döngü eğlencesi yaratabilir.

countdown örneğinde,bu döngünün sonlandırıldığını ispatlayabiliriz, çünkü n değişkeninin sonlu olduğunu biliyoruz ve n değişkeninin her seferinde küçüldüğünü görebiliriz. Bu nedenlerden döngünün eninde sonunda biteceği kesindir. Ancak bazı durumlarda bunu görmek bu kadar kolay olmayabilir:

```
def sequence(n):
    while n != 1:
        print n,
        if n % 2 == 0:        # n çifttir
            n = n / 2
        else:                 # n tektir
            n = n * 3 + 1
```

Bu döngüdeki koşul n != 1'dir, döngü koşulu yanlışlayan n 1 değerine sahip olana kadar devam edecektir.

Döngüdeki her seferde, program n'nin değerini ekranda görüntüler ve değişkenin tek veya çift olup olmadığını sınar. Eğer çift ise değerini ikiye böler. Eğer tek ise değeri n * 3 + 1 ile değiştirilir. Örneğin, başlangıç argümanı olarak 3 verdiğimizi düşünelim, ortaya çıkan seri 3, 10, 5, 16, 8, 4, 2, 1.'dir.

n artabildiği ve azalabildiği için değerinin 1'e erişip erişmeyeceğine dair bir garanti yoktur. n'nin bazı belirgin değerleri için yürütmenin biteceğini garantileyebiliriz. Örneğin, ikinin kuvvetlerini başlangıç değeri olarak kabul edersek n'nin döngüde her seferinde ikiye bölünerek çift sayı olup 1'e ulaşacağını biliriz. Önceki örnek böyle bir seri ile (16 ile başlayan) bitmektedir.

Bazı özel değerleri bir kenara koyarsak, bu programın yürütmesinin tüm giriş değerleri için bitip bitmeyeceğini ispatlayabilir miyiz sorusu başlı başına ilginç bir sorudur. Şu ana kadar, bunu ispatlayan veya ispatlayamayan çıkmamıştır!

6.4 Programı izlemek

Etkili bilgisayar programları yazabilmek için, bir programcının bilgisayar programlarının yürütmesini **izleme** yeteneğini geliştirmesi gereklidir. İzleme bilgisayar olmayı ve bir örnek çalıştırarak yürütme akışını takip etmeyi gerektirir. Bu takipte her yürütülen bir satırdan sonra bütün değişkenlerin durumları ve programın ürettiği çıktıları kaydedilir.

Bu süreci anlamak için, sequence(3) çağrımını izlemeye çalışalım. İzlemenin başında n adında 3 ilk değerine sahip bir yerel değişkenimiz (başlangıç parametresi) vardır. 3, 1'e eşit olmadığı için while döngüsünün gövdesi çalıştırılır. 3 ekranda görüntülenir ve 3 % 2 == 0 deyimi değerlendirilir, sonuç False olduğundan, else dalı yürütülür ve3 * 3 + 1 işlemi yapılır sonucu n değişkenine atanır.

Bütün bunları izleyebilmek için, bir kağıt parçasının üstüne yaratılan her değişken için başlık ve çıktı için bir başlık ekleriz. İzlememiz aşağıdaki gibi bir şey olacaktır:

```
 n              çıktı
---             ------
 3                 3
 10
```

10 != 1 ifadesinin değeri True olduğundan, döngü gövdesi tekrar yürütülecektir ve 10 ekranda görüntülenecektir. 10 % 2 == 0 doğru olduğundan, if dalı yürütülecek ve n5 olacaktır. Bu izlemenin sonunda aşağıdakine sahip olmamız gerekir:

```
 n              çıktı
---             ------
 3                 3
 10                10
 5                 5
 16                16
 8                 8
 4                 4
 2                 2
 1
```

İzleme can sıkıcı ve hataya eğilimli olabilir (bu yüzden bilgisayarları bu tür işler için kullanıyoruz), ancak bir programcının sahip olması gereken zorunlu yeteneklerden biridir. İzlemeyle kodumuzun nasıl çalıştığına dair bir çok şey öğrenebiliriz. n ikinin kuvveti olur olmaz, programın $\log_2(n)$ yürütme sonunda döngü gövdesinden çıkacağını izleme sayesinde gözlemleyebiliriz. Ayrıca son 1'in çıktı olarak ekranda görüntülenmeyeceğini de bulabiliriz.

6.5 Basamakları sayma

Aşağıdaki fonksiyon bir sayının onluk basamaklarını pozitif tamsayı şeklinde onluk biçimde sayar:

```
def num_digits(n):
    count = 0
    while n:
        count = count + 1
        n = n / 10
    return count
```

Fonksiyonu num_digits(710) şeklinde çağırdığımızda 3 değerini ürettiğini görebilirsiniz. Bu fonksiyon çağrımının yürütmesini izleyerek çalıştığına dair kendinizi ikna edin.

Bu fonksiyon hesaplamanın sayaç adı verilen başka bir şablonunu göstermektedir. count değişkeni 0 ile ilklenmekte ve daha sonra döngü gövdesinde artırrılmaktadır. Döngü tamamlandıktan sonra, count sonucu -- döngünün çalışma sayısı, ki bu toplam basamak sayısına eşittir -- içerir.

Eğer sadece 0 veya 5 olan basamakları saymak isteseydik, arttırımdan önce bir koşul cümlesi koymamız işimize yarardı:

```
def num_zero_and_five_digits(n):
    count = 0
    while n:
        digit = n % 10
        if digit == 0 or digit == 5:
            count = count + 1
        n = n / 10
    return count
```

num_zero_and_five_digits(1055030250) ifadesinin 7 döndürdüğünü doğrulayın.

6.6 Kısaltılmış atama

Bir değişkeni arttırma Python'da sık karşılaşılan bir şeydir, bu nedenle bir kısaltması vardır:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
>>>
```

count += 1 ifadesi count = count + 1 ifadesi yerine geçen bir kısaltmadır. Arttırma değeri 1 olmak zorunda da değildir:

```
>>> n = 2
>>> n += 5
>>> n
7
>>>
```

Diğer işleçler (-=, *=, /=, ve %=) için de kısaltmalar vardır:

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
>>> n
6
>>> n /= 2
>>> n
3
>>> n %= 2
>>> n
1
```

6.7 Tablolar

Döngülerle ilgili olarak güzel şeylerden biri tablo biçiminde veri üretmede yararlı olmasıdır. Bilgisayarlar yok iken logaritma, sinüs, cosinüs ve diğer matematiksel fonksiyonların değerlerini insanlar el ile hesaplamak zorundaydı. Bunu kolaylaştırmak için matematik kitapları bu değerleri barındıran büyük değer tabloları içeriyordu. Bu tabloları hazırlamak yavaş ve sıkıcı bir işti, ve genellikle hatalar barındırıyordu.

Bilgisayarlar sahneye çıktıktan sonraki ilk tepkilerden biri Bu çok iyi birşey! Bilgisayarları tabloları üretmek için kullanabiliriz, böylece herhangi bir hata olmaz. idi. Bu tepkinin (çoğunlukla) doğru olduğu ama öngörüsüz olduğu ortaya çıktı. Bilgisayarların ve hesap makinelerinin her tarafa yayılmasından sonra, artık tablolara gerek kalmadı.

Tabi hemen hemen demek gerekiyor. Bazı işlemler için, bilgisayarlar yaklaşık yanıtı üretebilmek ve yaklaşık yanıtı iyileştirmek için hesaplamalar yapmak üzere tabloları kullanıyor. Bazı durumlarda, kaynak olarak kullanılan tablolarda bazı hatalar vardı, bunun en çok bilinen örneği Intel Pentium tarafından kayan noktalı sayılarda bölme işleminde kullanılan tabloydu.

Her ne kadar bir log tablosu eskisi kadar yararlı olmasa da, yinelemenin güzel bir örneğidir. Aşağıdaki program sol sütundaki değer dizisini ve sağ sütunda 2 sayısının bu değerden kuvvetini bir tablo şeklinde gösterir:

```
x = 1
while x < 13:
    print x, '\t', 2**x
    x += 1
```

'\t' ifadesi **tab** karakterisini temsil eder. '\t' ifadesindeki '\' karakteri **kaçış serisi (escape sequence)**nin başlangıcını belirtir. Kaçış serileri tab ve yeni satır gibi görünmez karakterleri göstermek için kullanılırlar. '\n' serisi **yeni satır**ı betimler.

Bir kaçış serisi bir karakter dizisinin herhangi bir yerinde bulunabilir; bu örnekte, tab kaçış serisi karakter dizisindeki tek şeydir. Bir karakter dizisinde ters bölü ('\') karakterini nasıl temsil edersiniz?

Karakter ve karakter dizileri ekranda görüntülendikçe, **imleç (cursor)** görünmeyen bir işaretçi bir sonraki karakterin nereye koyulacağını takip eder. print cümlesinden sonra, imleç normal olarak bir sonraki satırın başına gider.

Tab karakteri imleçi bir tab sonu ile karşılaşana kadar sağa kaydırır. Tablar metinleri sutünlar şeklinde hizalamak için yararlıdır, aşağıda önceki programın çıktısını inceleyebilirsiniz:

```
1       2
2       4
3       8
4       16
5       32
6       64
7       128
8       256
9       512
10      1024
11      2048
12      4096
```

Sütunlar arasındaki tab karakterlerinden dolayı, ikinci sütunun konumu birinci sütundaki sayıların basamak sayısına bağlı değildir.

6.8 İki boyutlu tablolar

İki boyutlu bir tablo satır ve sütun kesişimindeki değeri okuduğunuz bir tablodur. Çarpım tablosu bu tür tablolar için iyi bir örnektir. 1'den 6'ya kadar çaprımları yazmak istediğinizi varsayalım.

Başlamak için iyi bir yol, tek bir kod satırında 2'nin çarpımlarını ekranda görüntüleyen bir döngü yazmaktır:

```
i = 1
while i <= 6:
    print 2*i, '    ',
    i += 1
```

```
print
```

İlk satır i ismindeki değişkeni ilkler, bu değişken sayaç veya **döngü değişkeni** şeklinde davranacaktır. Döngü işletildikçe i'nin değeri 1'den 6'ya artar. i 7 olduğunda, döngü sonlandırılır. Döngünün her seferinde 2*i'nin değeri boşluklardan sonra ekranda görüntülenir.

Yine, print cümlesindeki virgül yeni satıra başlamayı engeller. Döngü bittikten sonra, ikinci print cümlesi yeni bir satır başlatır.

Programın çıktısı:

```
2       4       6       8       10      12
```

Buraya kadar herşey güzel. Bir sonraki adım **sarma (encapsulate)** ve **genelleştirme (generalization)**dir.

6.9 Sarma (Encapsulation) ve genelleştirme

Sarma (encapsulation) bir kod parçasını fonksiyon içerisine koymadır. Böylece fonksiyonlar tarafından sağlanan tüm avantajlardan yararlanmış olacağız. Şimdiye kadar iki sarma örneği gördük: 4. bölümdeki print_parity ve 5. bölümdeki is_divisible fonksiyonları.

Genelleştirme özgü bir şeyi alıp, örneğin ikinin çarpım katlarını yazmak gibi, onu daha genel bir hale getirmektir, herhangi bir tamsayının tüm çarpım katlarını yazmak gibi.

Aşağıdaki fonksiyon bir önceki döngüyü sarıp genelleştirerek n tamsayısının çarpım katlarını ekranda görüntüler:

```
def print_multiples(n):
    i = 1
    while i <= 6:
        print n*i, '\t',
        i += 1
    print
```

Sarmak için yapmamız gereken tek şey fonksiyon ismini ve parametre listesini tanımladığımız başlığı ilk satır olarak eklemektir. Genelleştirmek için, yapmamız gereken tek şey 2 değerini parametre olarak tanımladığımız n ile değiştirmektir.

Eğer bu fonksiyonu 2 argümanı ile çağırırsak, daha önceki çıktıyla aynı sonucu elde ederiz. Eğer argüman olarak 3 verirsek, çıktı aşağıdaki gibi olur:

```
3       6       9       12      15      18
```

Argüman olarak 4 verirsek, çıktımız:

```
4       8       12      16      20      24
```

Artık tahmin edeceğiniz gibi çarpım tablosunu nasıl ekranda görüntüleyeceğinizi -- print_multiples fonksiyonunu farklı değerler için tekrar tekrar çağırarak -- tahmin etmişsinizdir. Aslında başka bir döngü de kullanabiliriz:

```
i = 1
while i <= 6:
    print_multiples(i)
    i += 1
```

Bu yeni döngünün print_multiples içindeki döngüye ne kadar benzediğine dikkat edin. Yaptığımız tek şey print cümlesi yerine bir fonksiyon çağrımı -- print_multiples-- koymak oldu.

Bu programın çıktısı çarpım tablosudur:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

6.10 Daha fazla sarma

Sarmayı tekrar göstermek için önceki bölümden bir kod alıp fonksiyon şekline getirelim:

```
def print_mult_table():
    i = 1
    while i <= 6:
        print_multiples(i)
        i += 1
```

Bu süreç yaygın karşılaşılan bir **geliştirme plan**ıdır. Kodu herhangi bir fonksiyon dışında yazarak veya yorumlayıcıya doğrudan yazarak geliştiririz, tam olarak çalışan koda ulaştığımızda bu kodu bir fonksiyon içerisine alıp, fonksiyon haline getiririz.

Bu geliştirme planı eğer programı hangi fonksiyonlara parçalayacağınızı bilmiyorsanız oldukça yararlıdır. Bu yaklaşım programı geliştirdikçe tasarlamanıza olanak sağlar.

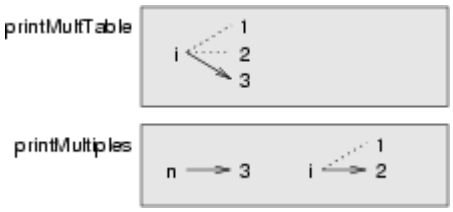
6.11 Yerel değişkenler

Aynı değişkeni, i, hem print_multiples hem de print_mult_table fonksiyonlarının ikisinde nasıl kullanabildiğimizi merak ediyor olabilirsiniz. Bu fonksiyonlardan birii'nin değerini değiştirirse bu sorunlara yol açmaz mı?

Yanıt "hayır"dır, çünkü print_multiples ve print_mult_table fonksiyonlarındaki i değişkeni aynı değişken değildir.

Bir fonksiyon tanımlaması içerisinde yaratılan değişkenler yereldir; tanımlandığı ev fonksiyonu dışından bir yerden değişkene erişemezsiniz. Bunun anlamı aynı fonksiyon içerisinde tanımlı olmayan, aynı isme sahip birden fazla değişkene sahip olabilirsiniz.

Bu programdaki i isimli iki değişkeni gösteren yığıt diagramını aşağıda inceleyebilirsiniz. Farklı değerleri gösterebilirler ve birinin değişmesi diğerini etkilemez.



print_mult_table fonksiyonundaki i değişkeninin değeri 1'den 6'ya artar. Şekilde 3'tür. Bir sonraki döngüde 4 olacaktır. Döngüdeki her seferde, print_mult_tablefonksiyonu print_multiples fonksiyonunu i değişkeninin o anki değerini argüman olarak kullanarak çağırmaktadır. Bu değer n parametresine atanmaktadır.

print_multiples içerisinde, i değişkeninin değeri 1'den 6'ya artmaktadır. Şekilde 2'dir. Bu değişkeni değiştirmenin print_mult_table fonksiyonu içerisindeki i üzerinde bir etkisi yoktur.

Aynı isimde farklı yerel değişkenlere sahip olmak, sık karşılaşılan ve tamamen geçerlidir. Aslında i ve j gibi isimler döngülerde değişken ismi olarak sıklıkla kullanılmaktadır. Eğer başka bir fonksiyon içerisinde kullandığınız için kullanmamazlık edersseniz, programın okunmasını zorlaştırırsınız.

6.12 Daha fazla genelleştirme

Genelleştirmeye başka bir örnek olarak sadece 6x6 değil herhangi bir boyutta çarpım tablosu yazdırma isteğimizi verebiliriz, print_mult_table fonksiyonuna bir parametre ekleyerek bunu gerçekleştirebiliriz:

```
def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i)
        i += 1
```

6 değerini high parametresi ile değiştirdik. Eğer print_mult_table fonksiyonunu 7 argümanı ile çağırırsak, aşağıdaki sonucu üretecektir:

```
1       2       3       4       5       6
2       4       6       8       10      12
```

```
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

Tablonun kare olmasını (satır ve sütun sayısının eşit olması) isteyebileceğimiz dışında bu yeterli bir çözüm. Tabloyu kare haline getirmek için print_multiplesfonksiyonuna sütun sayısını belirleyecek olan başka bir parametre ekleriz.

Can sıkmak için, bu parametreyi de high olarak adlandıralım, böylece farklı fonksiyonların aynı isme sahip (yerel değişkenler gibi) parametrelere sahip olabileceğini göstermiş oluruz. Programın son hali şekildedir:

```
def print_multiples(n, high):
    i = 1
    while i <= high:
        print n*i, '\t',
        i += 1
    print

def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i, high)
        i += 1
```

Yeni bir parametre eklediğimizde fonksiyonun ilk satırını (fonksiyon başlığı) değiştirdiğimiz farketmişsinizdir. Buna ek olarak fonksiyonun çağrıldığı yerleri (örneğinprint_mult_table içinde çağrılan yer) de değiştirmemiz gerekmektedir.

Beklediğimiz gibi, bu program tabloyu 7x7 boyutlarında kare şeklinde üretir:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

Bir fonksiyonu uygun şekilde genelleştirdiğinizde, planlamadığınız yeteneklere sahip bir program elde edersiniz. Örneğin ab = ba olması dolayısıyla tablodaki her bir girdinin iki kere görüntülendiğini farketmişsinizdir. Mürekkep harcamasını azaltmak için tablonun sadece yarısını görüntülemek isteyebilirsiniz. Bunu yapmak içinprint_mult_table fonksiyonunda bir satır değiştirmeniz gerekir. Aşağıdaki satırı

```
print_multiples(i, high)
```

aşağıdaki şekilde

```
print_multiples(i, i)
```

değiştirdiğinizde şu sonucu üretebilirsiniz:

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

6.13 Fonksiyonlar

Şimdiye kadar bir kaç kere fonksiyonlar için iyi olan şeylerden bahsettik. Bu noktada, iyi olan bu şeylerden bir kısmını toplu bir şekilde vermemiz aydınlatıcı olacaktır:

1. Bir cümle dizisine bir isim vermeniz programınızın okunabilirliğini arttıracak, hata ayıklamayı kolaylaştıracaktır.
2. Büyük bir programı fonksiyonlara parçalamanız, programda parçaları birbirinden ayırmanızı sağlayacaktır. Böylece izole bir şekilde hataları ayıklayabilecek, bu farklı parçaların bir bütün olarak davranmasını sağlayabileceksiniz.
3. Fonksiyonlar yinelemenin kullanımını kolaylaştırır.
4. İyi tasarlanmış fonksiyonlar, yazılıp iyi bir şekilde hatalardan arındırıldıktan sonra tekrar kullanılabildiği için, bir çok program için yararlıdır.

6.14 Newton yöntemi

Döngüler, bir yaklaşık değer ile başlayıp yineli olarak bu yaklaşık değerin iyileştirildiği nümerik hesaplamalarda sıklıkla kullanılmaktadır.

Örnek olarak, kare kök hesaplamada kullanılan Newton yöntemini verebiliriz. Farzedelim ki, n değişkeninin kare kökünü bulmak istiyoruz. Herhangi bir yaklaşık değer ile başlarsak, aşağıdaki formülü kullanarak daha iyi bir yaklaşık değer (başladığımız yaklaşık değeri iyileştirerek) hesaplayabiliriz:

```
beter =  (approx + n/approx)/2
```

Bu formülü, yeni elde ettiğimiz yaklaşık değer önceki değerden farklı olduğu sürece (better!=approx) yineleyerek kullanan bir kare kök hesaplama fonksiyonu yazabiliriz:

```
def sqrt(n):
    approx = n/2.0
    better = (approx + n/approx)/2.0
```

```
    while better != approx:
        approx = better
        better = (approx + n/approx)/2.0
    return approx
```

Bu fonksiyonu 25 değeri için çağırın ve fonksiyonun 5.0 sonucu üretip üretmediğini sınayarak doğru çalışıp çalışmadığını kontrol edin.

6.15 Algoritmalar

Newton yöntemi bir **algoritma** örneğidir: belli bir kategorideki (bu durumda kare kökleri hesaplama kategorisi) problemleri çözmeye yönelik mekanik bir süreçtir.

Bir algoritmayı tanımlamak kolay değildir. İlk olarak algoritma olmayan bir şey ile başlamak yardımcı olacaktır. Tek basamaklı sayıları çarpmayı öğrendiğinizde, çarpım tablosunu ezberlemişsinizdir. Bunun sonucunda, 100 belli çözümü ezberlediniz. Bu tür bilgi algoritmik değildir.

Eğer tembel biriyseniz, bir kaç ipucu kullanarak öğrenip hile yapmışsınızdır. Örneğin n ve 9'un çarpımını bulmak için n-1 ilk basamak, 10-n ikinci basamak olacak şekilde yazarak sonucu üretmişsinizdir (7x9 = 63'ü düşünün). Bu "hile" herhangi tek basamaklı bir sayısı 9 ile çarpmak için genel çözümdür. Bu bir algoritmadır!

Benzer olarak, eldeli toplama, borçlu çıkarma ve uzun bölme işlemleri için öğrendiğiniz tekniklerin hepsi algoritmalardır. Algoritmaların temel özelliği uygulanabilmeleri için özel bir zeka gerektirmemeleridir. Basit kurallara göre her bir adımın birbirini izlediği tamamen mekanik süreçlerdir.

Bizim düşüncemize göre, insanların okulda herhangi bir zeka gerektirmeyen algoritmaları yürütmeyi öğrenmek için çok fazla zaman harcamaları utandırıcı bir durumdur. Diğer taraftan, algoritma tasarımı süreci ilginç, zihni zorlayıcı ve programlamanın temel parçası olan bir süreçtir.

İnsanların zorlanmadan veya bilinç dışı olarak doğallıkla yaptığı bazı şeyleri algoritmik olarak ifade etmek zordur. Doğal dili anlamak bunun güzel bir örneğidir. Hepimiz bunu yapıyoruz ama nasıl yaptığımızı şimdiye kadar kimse tam olarak anlatamamıştır. En azından bu işlemi herhangi bir algoritmik biçimde yapmıyoruz.

6.16 Glossary

birden fazla atama:
Programın yürütülmesi esnasına bir değişkene birden fazla atama yapılması

(değişken) ilkleme:
Bir değişkeni ilklemek (initialize), değişkene bir başlangıç değeri verilmesidir, genellikle birden fazla atama durumlarında yapılır. Python'da değişkenler atama yapılmadığı sürece varolmadıkları için, yaratıldıkları zaman ilklenirler. Diğer programlama dillerinde durum bu şekilde değildir, ve değişkenler

ilklenmeden de yaratılabilir, ilk değer olarak ya varsayılan değeri, ya da çöp, anlamsız değeri barındırırlar.

arttırma
İsim ve sıfat olarak, arttırma bir değere 1 ekleme anlamına gelir.

azaltma
1 çıkarma.

yineleme:
Programlama cümlesi kümesinin tekrar tekrar çalıştırılması.

döngü:
Bitiş kriteri karşılanana kadar tekrar tekrar çalıştırılan bir cümle veya cümle grubu.

sonsuz döngü:
Bitiş kriterinin hiç bir zaman karşılanamadığı döngü.

izleme:
Bir programın yürütme akışını elle takip etme, değişken durumlarındaki değişimleri ve üretilen çıktıları kaydetme işlemidir.

sayaç
Bir şeyi saymak için kullanılan değişken, genellikle sıfır olarak ilklenip bir döngü gövdesinde arttırılır.

gövde:
Bir döngü içerisindeki cümleler.

döngü değişkeni:
Bir döngünün bitiş kriterinin bir kısmı olarak kullanılan değişken.

tab:
İmlecin bulunulan satırda bir sonraki tab durma noktasına hareket etmesini sağlayan özel karakter.

yeni satır:
İmlecin bir sonraki satırın başına konumlanmasını sağlayan özel karakter.

imleç:
Bir sonraki karakterin nereye yazılacağını tutan görünmez işaretçi.

kaçış dizisi:
Bir kaçış karakteri, \, ve onu takip eden yazılabilir bir veya daha fazla karakterden oluşan yazılmayan bir karakteri belirtmek için kullanılan karakter dizisi.

sarma (encapsulation):
Büyük karmaşık bir programı bileşenlere (fonksiyonlar gibi) bölme (parçalama, ayırma) ve her bir bileşeni diğerlerinden yalıtma (örneğin yerel değişkenleri kullanarak) işlemidir.

genelleştirme:
Gereksiz yere özel olan bir ifadeyi (sabit bir değer gibi), uygun bir şekilde genel olan bir ifadeyle (bir değişken veya parametre gibi) değiştirme işlemidir. Genelleştirme kodu çok yönlüleştirir, tekrar kullanılabilirliğini arttırır, hatta yazılması işlemini kolaylaştırabilir.

geliştirme planı:
Bir programı geliştirme sürecidir. Bu bölümde basit, belli şeyleri yapıp bunları sarıp, genelleştirilmesine dayanan bir kod geliştirme süreci anlatılmıştır.

algoritma:
Belli bir kategorideki problemleri adım adım çözme sürecidir.

6.17 Alıştırmalar

1. Aşağıdaki çıktıyı üreten tek bir karakter dizisi yazın

```
bu
çıktıyı
üretmeli.
```

2. 6.14 bölümünde tanımlanan sqrt fonksiyonuna better değişkeni her hesaplandığında ekranda görüntüleyen bir print cümlesi ekleyin. Fonksiyonun değiştirdiğiniz halini 25 argümanı ile çağırıp, sonuçları kaydedin.
3. print_mult_table fonksiyonunun son sürümünün yürütmesini izleyip, nasıl çalıştığını anlamaya çalışın.
4. Adı print_triangular_numbers(n) olan ve ilk üçgen sayıları ekranda görüntüleyen bir fonksiyon yazın. print_triangular_numbers(5) çağrımı aşağıdaki çıktıyı üretmelidir:

```
1       1
2       3
3       6
4       10
5       15
```

   (ipucu: bir web araması yaparak üçgensel sayının ne olduğunu bulabilirsiniz.)
5. Adı ch06.py olan bir dosya yaratıp içerisine aşağıdakileri ekleyin:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

   is_prime isminde, tek bir tamsayı argüman alan ve eğer argüman **asal sayı** ise True döndüren, değilse False döndüren bir fonksiyon yazın. Fonksiyonunu geliştirirken doctestlerini de ekleyin.
6. num_digits(0) ne döndürecektir? Bu durum için 1 döndürecek şekilde fonksiyonu düzeltin. Neden num_digits(-24) çağrımı sonsuz bir döngü oluşturmaktadır (ipucu: -1/10 değeri -1 olarak değerlendirilmektedir)? num_digits fonksiyonunu herhangi bir tamsayı değerler doğru çalışacak şekilde düzeltin.

   Daha önceki alıştırmada hazırladığınız ch06.py dosyasına aşağıdakileri ekleyin:

```
def num_digits(n):
```

```
    """
      >>> num_digits(12345)
      5
      >>> num_digits(0)
      1
      >>> num_digits(-12345)
      5
    """
```

Fonksiyon gövdenizi num_digits fonksiyonuna koyup tüm dostestleri geçtiğini doğrulayın.

7. Aşağıdakileri ch06.py içerisine koyun:

```
def num_even_digits(n):
    """
      >>> num_even_digits(123456)
      3
      >>> num_even_digits(2468)
      4
      >>> num_even_digits(1357)
      0
      >>> num_even_digits(2)
      1
      >>> num_even_digits(20)
      2
    """
```

num_even_digits fonksiyonuna beklendiği gibi çalışacak bir gövde yazın.

8. Aşağıdakiler ch06.py dosyasına ekleyin:

```
def print_digits(n):
    """
      >>> print_digits(13789)
      9 8 7 3 1
      >>> print_digits(39874613)
      3 1 6 4 7 8 9 3
      >>> print_digits(213141)
      1 4 1 3 1 2
    """
```

print_digits fonksiyona tüm verilen doctestleri geçebileceği bir gövde yazın.

9. Adı sum_of_squares_of_digits olan bir fonksiyon yazın. Bu fonksiyon argüman olarak verilen tamsayının tüm basamaklarının karelerini toplasın. Örneğin,sum_of_squares_of_digits(987) 194 döndürmelidir, çünkü 9**2 + 8**2 + 7**2 == 81 + 64 + 49 == 194'tür.

```
def sum_of_squares_of_digits(n):
    """
      >>> sum_of_squares_of_digits(1)
```

```
    1
    >>> sum_of_squares_of_digits(9)
    81
    >>> sum_of_squares_of_digits(11)
    2
    >>> sum_of_squares_of_digits(121)
    6
    >>> sum_of_squares_of_digits(987)
    194
  """
```

Çözümünüzü yukarıdaki doctestler bağlamında sınayın.

7. Karakter dizileri

7.1 Bileşik veri tipi

Şimdiye kadar beş tip gördük:int, float, bool, NoneType ve str. Karakter dizileri (str) diğer dört tipten nitelik olarak farklıdır çünkü daha küçük parçalardan (karakterler) oluşmuştur.

Daha küçük parçalardan oluşan tiplere **bileşik veri tipleri** adı verilir. Ne yaptığımıza bağlı olarak, bileşik veri tipine tek bir şeymiş gibi davranabilir veya parçalarına erişmek isteyebiliriz. Bu belirsizlik yararlıdır.

Köşeli parantez işleci bir karakter dizisinden tek bir karakter seçmeye yarar:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

fruit[1] deyimi fruit değişkeninden 1 numaralı karakteri seçer. değişkeni sonucu referans eder. letter değişkenini ekranda görüntülediğimizde, bir sürprizle karşılaşırız:

```
a
```

"banana" kelimesinin ilk harfi a değildir, eğer bilgisayar bilimcisi değilseniz. Sapkın nedenlerden dolayı, bilgisayar bilimcileri saymaya her zaman sıfırdan başlarlar."banana" kelimesinin Sıfırıncı (0.) harfi b'dir. Birinci (1.)harfi a ve ikinci (2.) harfi n'dir.

Bir karakter dizisinin sıfırıncı karakterini istiyorsanız, köşeli parantezler içerisine 0 veya sonucu 0 olan bir deyim koyarsınız:

```
>>> letter = fruit[0]
>>> print letter
b
```

Köşeli parantezler içerisindeki deyim **indis (index)** adını alır. Bir dizin sıralı bir kümedeki belli bir öğeyi tanımlar, bizim durumumuzda karakter dizisi içerisindeki karakterlerin kümesini hedeflemektedir. İndis, isimden de anlaşılacağı gibi hangi öğeyi istediğinizi belirtir. Herhangi bir tamsayı ifade olabilir.

7.2 Uzunluk

len fonksiyonu bir karakter dizisindeki karakterlerin sayısını döndürür:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

Bir karakter dizisinin son harfine ulaşmak için, aşağıdaki gibi bir şeyi kullanmak isteyebilirsiniz:

```
length = len(fruit)
last = fruit[length]        # ERROR!
```

Bu çalışmaz, çalışma zamanı hatası üretir IndexError: string index out of range. Bunun nedeni sıfırdan saymaya başladığımız için, altı tane harf 0 ve 5 arasında numaralanacaktır. Son karakteri seçmek için, uzunluktan bir çıkarmamız gerekir:

```
length = len(fruit)
last = fruit[length-1]
```

Alternatif olarak, negatif indisleri de kullanabiliriz, bunlar karakter dizisinin sonundan itibaren saymaya başlarlar. fruit[-1] deyimi son harfi getirir, fruit[-2] sondan ikinci harfi getirir ve böyle devam eder.

7.3 Gezinme ve for döngüsü

Bir çok hesaplama bir karakter dizisinin bir anda tek bir karakterini işlemeyi barındırır. Genellikle baştan başlar ve sona kadar devam edr. Bu şekildeki işleme yöntemine**gezinme (traversal)** adı verilir. Bir gezinmeyi kodlamanın bir yolu while cümlesi kullanmaktır:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index += 1
```

Bu döngü karakter dizisini dolaşır ve her seferinde bir harfi ekranda görüntüler. Döngü koşulu index < len(fruit), böylece index karakter dizisinin uzunluğuna eriştiğinde koşul yanlışlanacak ve döngünün gövdesi yürütülmeyecektir. En son erişilen karakter indisi len(fruit) - 1 olan karakter olacaktır, bu da karakter dizisindeki son karakterdir.

Bir değer kümesini gezinmede indis kullanımı çok yaygın kullanılan bir yöntem olduğu için Python bir alternatif sağlamaktadır, daha kolay sözdizimine sahip for döngüsü:

```
for char in fruit:
   print char
```

Döngüde her anda, karakter dizisindeki bir sonraki karakter char değişkenine atanıyor. Döngü hiç karakter kalmayana kadar devam eder.

Aşağıdaki örnek birleştirme işlemi ve for döngüsünü kullanarak abecedarian serisi üretmeyi göstermektedir. Abecedarian öğelerin alfabetik bir şekilde sıralanmış seri veya listeyi temsil eder. Örneğin, Robert McCloskey'in kitabı Make Way for Ducklings, ördek yavrularının isimleri Jack, Kack, Lack, Mack, Nack, Ouack, Pack ve Quack'tır. Aşağıdaki döngü bu isimleri sıralı olarak ekranda görüntüler:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
   print letter + suffix
```

Programın çıktısı:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

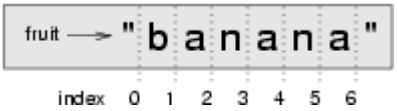
Elbete, bu o kadar doğru değildir çünkü Ouack ve Quack yanlış yazılmıştır. Bunu aşağıdaki alıştırmaların birinde düzelteceksiniz.

7.4 Karakter dizisi dilimleri

Bir karakter dizisinin her bir parçasına (alt karakter dizisi - substring) **dilim** adı verilmektedir. Bir dilimi seçmek bir karakteri seçmeye benzer:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

[n:m] işleci bir karakter dizisinin n. ve m. karakterleri arasındaki parçasını, ilk karakteri içerir ancak son karakteri içermez, döndürür. Bu davranış düşülenin aksine bir davranıştır; eğer indisleri aşağıdaki resimdeki gibi karakter aralarını işaret ediyor diye düşünürseniz daha anlamlı gelecektir:



Eğer ilk indisi yazmazsanız (iki nokta üstüsteden önceki), dilim karakter dizisinin başından başlar. Eğer ikinci indisi yazmazsanız, dilim karakter dizisinin sonuna kadar gider. Böylece:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Bu durumda s[:] ifadesinin ne olduğunu düşünüyorsunuz?

7.5 Karakter dizisi karşılaştırma

Karşılaştırma işlecleri karakter dizileri için de çalışır. İki karakter dizisinin eşit olup olmadığını kontrol etmek için:

```
if word == "muz":
    print  "Evet, iki muzumuz var!"
```

Diğer karşılaştırma işleçleri kelimeleri alfabetik sıraya koymak için yararlıdır:

```
if word < "muz":
    print "Kelimeniz," + word + ", muzdan önce gelir."
elif word > "muz":
    print "Kelimeniz," + word + ", muzdan sonra gelir."
else:
    print "Evet, hiç muzumuz yok!"
```

Python'un küçük ve büyük harflere insanların davrandığı gibi davranmadığını unutmamalısınız. Tüm büyük harfler küçük harflerden önce gelir. Bu yüzden:

```
Kelimeniz, Zebra, muzdan önce gelir.
```

Bu probleme yönelik yaygın bir kullanım karakter dizilerini standart bir biçime dönüştürmektir. Örneğin herhangi bir karşılaştırma yapmadan önce hepsini küçük harfe çevirebilirsiniz. Daha büyük problem ise programın zebranın bir meyve olmadığını anlamamasıdır.

7.6 Karakter dizileri değişmez

Sadece bir karakteri değiştirmek amacıyla [] işlecini atamanın solunda kullanmak isteyebilirsiniz. Örneğin:

```
greeting = "Merhaba, dünya!"
greeting[0] = 'N'             # ERROR!
print greeting
```

Nerhaba, dünya! çıktısını üretmek yerine yukarıdaki kod bir çalışma zamanı hatası TypeError: 'str' object doesn't support item assignment üretmektedir.

Karakter dizileri **değişmez (immutable)**, bunun anlamı var olan bir karakter dizisini değiştiremezsiniz. En iyi yapabileceğiniz şey var olan karakter dizisi üzerinde bir değişiklik yapıp yeni bir karakter dizisi yaratmaktır:

```
greeting = "Merhaba, dünya!"
newGreeting = 'N' + greeting[1:]
print newGreeting
```

Buradaki çözüm değiştirmek istediğimiz ilk karakterin yeni halini, greeting değişkeninde değişmemesi gereken kısımla birleştirmedir. Bu işlemin özgün karakter dizisi üzerinde bir etkisi yoktur.

7.7 in işleci

in işleci bir karakter dizisinin başka bir karakter dizisinde alt karakter dizisi olarak varolup olmadığını sınar:

```
>>> 'p' in 'apple'
True
>>> 'i' in 'apple'
False
>>> 'ap' in 'apple'
True
>>> 'pa' in 'apple'
False
```

Karakter dizisinin kendisinin de bir alt karakter dizisi olduğunu unutmamak gerekir:

```
>>> 'a' in 'a'
True
>>> 'apple' in 'apple'
True
```

in işlecini karakter dizisi birleştirme (+ işleci) ile bir araya getirdiğimizde, bir karakter dizisinden tüm sesli harfleri çıkaran bir fonksiyon yazabiliriz:

```
def remove_vowels(s):
```

```
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    for letter in s:
        if letter not in vowels:
            s_without_vowels += letter
    return s_without_vowels
```

Bu fonksiyonu istediğimizi yapıp yapmadığına dair bir sınamadan geçirin.

7.8 Bir bulma(find) fonksiyonu

Aşağıdaki fonksiyon ne yapar?

```
def find(strng, ch):
    index = 0
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

Bir bakıma, find [] işlecinin karşıtıdır. İndisi alıp bu indise karşılık gelen karakter yerine, bir karakter alıp o karakterin bulunduğu indisi döndürmektedir. Eğer karakter yok ise, -1 değerini döndürür.

Bu return cümlesini bir döngü içerisinde gördüğünüz ilk örnektir. Eğer strng[index] == ch ise, fonksiyon hemen dönecektir, döngü erken bir şekilde sonlandırılmış olacaktır.

Eğer karakter, karakter dizisinin içerisinde bulunamazsa döngü normal bir şekilde tamamlanacak ve fonksiyon -1 değerini döndürecektir.

Bu tarz hesaplama şablonuna euraka gezinimi denilmektedir çünkü sonucu bulur bulmaz, Euraka! diye bağırıp aramayı bırakabiliriz.

7.9 Döngü ve sayma

Aşağıdaki program a harfinin bir karakter dizisinde kaç kere gözüktüğünü sayar, ve 6. bölümde anlattığımız sayma şablonunun bir başka örneğidir:

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

7.10 İsteğe bağlı parametreler

Bir karakterin bir karakter dizisinde ikinci veya üçüncü konumlarını bulmak istersek, find fonksiyonunu bir üçüncü parametre -- aramanın yapıldığı karakter dizisinde aramaya başlama indisi -- ekleyerek değiştirebiliriz:

```
def find2(strng, ch, start):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

find2('banana','a',2) şimdi 3 değerini döndürmektedir, bu değer 'a' karakterinin 2. indisten sonraki ilk bulunduğu yerin indisidir. find2('banana','n',3) ne döndürecektir? Eğer, 4 dediyseniz find2'nin nasıl çalıştığına dair bir fikriniz oluşmuş demektir.

Daha iyisi, find ve find2 fonksiyonlarını **isteğe bağlı (optional) parametre** kullanarak birleştirebiliriz:

```
def find(strng, ch, start=0):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

Fonksiyonunun bu sürümüne yapılan find('banana', 'a', 2) çağrımı aynen find2 gibi davranacaktır, find('banana','a') çağrımında ise start **varsayılan (default) değer**e yani 0'a değerine sahip olacaktır.

find fonksiyonuna yeni bir isteğe bağlı parametre ekleyerek kodumuzun her iki yönde (ileri, geri) arama yapmasını sağlayabiliriz:

```
def find(strng, ch, start=0, step=1):
    index = start
    while 0 <= index < len(strng):
        if strng[index] == ch:
            return index
        index += step
    return -1
```

step için -1 değeri geçirmemiz karakter dizisinin sonu yerine başından arama yapmamızı sağlayacaktır. Unutulmaması gereken, index değişkenini alabileceği en yüksek değer olduğu kadar bu değişikliği karşılayabilmek için en düşük değerler için de kontrol etmemiz gerekmektedir.

7.11 string modülü

string modülü karakter dizileri üzerinde işlemler yapmamız için gereken yararlı fonksiyonlar içermektedir. Her modül için olduğu gibi kullanmadan önce bu modülü de içe aktarmamız gerekmektedir:

```
>>> import string
```

Bu modülün içerdiklerini görmek için, dir fonksiyonunu argüman olarak modül ismini vererek kullanabiliriz.

```
>>> dir(string)
```

bu deyim string modülü içerisindeki öğelerin listesini döndürecektir:

['Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__', '__name__', '_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'atof', 'atof_error', 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize', 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find', 'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase', 'lstrip', 'maketrans', 'octdigits', 'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase', 'translate', 'upper', 'uppercase', 'whitespace', 'zfill']

Bu listedeki herhangi bir öğe hakkında daha fazla bilgi almak istersek, type komutunu kullanabiliriz. Modül ismini ve **nokta işareti**nden sonra da öğe ismini yazarak sonucu inceleyebiliriz.

```
>>> type(string.digits)
<type 'str'>
>>> type(string.find)
<type 'function'>
```

string.digits bir karakter dizisi olduğuna göre, içeriğini ekranda görüntüleyebiliriz:

```
>>> print string.digits
0123456789
```

Şaşırtmayacak şekilde, onluk sistemdeki rakamları içeriyor.

string.find bizim yazdığımız fonksiyonla neredeyse aynı şeyi yapan bir fonksiyon. Hakkında daha fazla bilgi almak için **docstring**'ini (__doc__) -- fonksiyonun belgelendirilmesidir -- ekranda görüntüleyebiliriz:

```
>>> print string.find.__doc__
find(s, sub [,start [,end]]) -> in

   Return the lowest index in s where substring sub is found,
   such that sub is contained within s[start,end].  Optional
   arguments start and end are interpreted as in slice notation.
```

```
   Return -1 on failure.
```

Köşeli parantez içerisindeki parametreler isteğe bağlı parametrelerdir. string.find fonksiyonunu bizim yazdığımız find fonksiyonu ile aynı şekilde kullanabiliriz:

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

Bu örnek, modüllerin bir yararını göstermektedir, var olan (built-in) fonksiyonlar ile kullanıcı tanımlı fonksiyonların isimden dolayı karışmasını önler. Nokta işaretini kullanarak hangi find sürümünü kullanmak istediğimizi belirtebiliriz.

Aslında, string.find bizim fonksiyondan daha genel bir sürümdür. Sadece karakterleri değil, alt karakter dizilerini de bulabilir:

```
>>> string.find("banana", "na")
2
```

Bizimki gibi, başlaması gereken indisi bir ek argüman olarak alabilir:

```
>>> string.find("banana", "na", 3)
4
```

Bizimkinden farklı olarak ikinci isteğe bağlı parametresi, aramayı bitirmesi gereken indisi vermek için:

```
>>> string.find("bob", "b", 1, 2)
-1
```

Bu örnekte arama başarısız olur çünkü b harfi verdiğimiz indis aralığında (1'den 2'ye - 2 dahil değil) yoktur.

7.12 Karakter sınıflandırma

Bir karakteri incelemek ve büyük-küçük olduğunu kontrol etmek veya harf mi rakam mı olduğunu belirlemek genellikle yararlıdır. string modülü bu iş için bir çok sabit sağlamaktadır. Bu sabitlerden biri, string.digits, daha önce içeriğini gördüğümüz bir sabittir.

string.lowercase sistemin küçük harf olarak tanımladığı tüm harfleri içerir. Benzer bir şekilde string.uppercase bütün büyük harfleri içerir. Aşağıdakini deneyip sonucunu inceleyebilirsiniz:

```
print string.lowercase
print string.uppercase
print string.digits
```

Bu sabitleri ve find fonksiyonunu karakterleri sınıflandırmak için kullanabiliriz. Örneğin, eğer find(lowercase,ch) fonksiyon çağrımı -1'den farklı bir değer döndürürse, chküçük harf olmalıdır:

```
def is_lower(ch):
    return string.find(string.lowercase, ch) != -1
```

Bir alternatif olarak, in işlecinin avantajını kullanabiliriz:

```
def is_lower(ch):
    return ch in string.lowercase
```

Başka bir alternatif olarak, karşılaştırma işlecini kullanabiliriz:

```
def is_lower(ch):
    return 'a' <= ch <= 'z'
```

Eğer ch a ve z arasında ise, küçük harf olmak zorundadır.

string modülünde bulunan başka bir sabit ekranda görüntülediğiniz zaman sizi şaşırtabilir:

```
>>> print string.whitespace
```

Boş karakterler (whitespace) imleci herhangi bir şey görüntülemeden hareket ettiren karakterlerdir. Görünür karakterler arasında beyaz boşluklar (en azından beyaz kağıt üstünde) yaratırlar. string.whitespace sabiti tüm boş karakterleri -- boşluk, tab (\t) ve yeni satır (\n) -- içerir:

string modülü içerisinde bir sürü başka yararlı fonksiyon vardır, ancak bu kitap referans el kitabı amacı gütmediği için tüm bu yararlı fonksiyonları burada anlatmayacağız.Python Kütüphane Referansı (Python Library Reference), referans kitabıdır. Diğer bir çok yararlı belgeyle birlikte Python web sitesinden erişilebilir, http://www.python.org

7.13 Karakter dizisi biçimlendirme

Python programlamada bir karakter dizisini biçimlendirmenin en kesin ve güçlü yolu karakter dizisi biçimlendirme işlecini, %, Python'un karakter dizisi biçimlendirme işlemleriyle birlikte kullanmaktır. Bunun nasıl çalıştığını görmek için, aşağıdaki bir kaç örnekle başlayalım:

```
>>> "His name is %s."  % "Arthur"
'His name is Arthur.'
>>> name = "Alice"
>>> age = 10
>>> "I am %s and I am %d years old." % (name, age)
'I am Alice and I am 10 years old.'
>>> n1 = 4
>>> n2 = 5
```

```
>>> "2**10 = %d and %d * %d = %f" % (2**10, n1, n2, n1 * n2)
'2**10 = 1024 and 4 * 5 = 20.000000'
>>>
```

Karakter dizisi biçimlendirme işleminin sözdizimi şu şekildedir:

```
"<BICIM>" % (<DEGERLER>)
```

Peşpeşe gelen karakterler ve dönüştürme belirtimlerinden oluşan biçim ile başlar. Dönüştürme belirtimleri % işleci ile başlar. Biçim karakter dizisini tek bir % işareti ve her bir dönüşüm belirtimi için peşpeşe gelen değerler izler, değerler virgül ile ayrılmıştır ve parantez içerisine alınmıştır. Tek bir değer olması durumunda parantezler isteğe bağlıdır.

Yukarıdaki ilk örnekte, tek bir dönüştürme değeri vardır, %s, bu bir karakter dizisini gösterir. Ve tek bir değer, "Arthur" bu işaretten etkilenir, dikkat edilirse parantez kullanılmamıştır.

İkinci örnekte, name karakter dizisi değerine sahiptir, "Alice"; age tamsayı değerine sahiptir, 10. Bu iki dönüştürme belirtimine eşlenir, %s ve %d. İkinci dönüştürme belirtimindeki d karakteri değerin bir onluk tamsayı olduğunu belirtir.

Üçüncü örnekte n1 ve n2 değişkenleri 4 ve 5 tamsayı değerlerine sahipler. Biçim karakter dizisinde dört dönüştürme belirtimi var: üç %d ve bir %f. f karakteri değerin kayan noktalı sayı olarak temsil edilmesi gerektiğini belirtir. Dört dönüştürme belirtimine eşlenen dört değer ise: 2**10, n1, n2 ve n1 * n2'dir.

s, d, ve f karakterlerinin hepsi bu kitap için gereksinim duyacağımız dönüştürme tipleridir. Dönüştürme karakterlerinin tüm listesini incelemek için Python Library Reference kitabının String Formatting Operations (Karakter Dizisi Biçimlendirme İşlemleri) bölümüne bakabilirsiniz.

Aşağıdaki örnek karakter dizisi biçimlendirmenin gerçek kullanımını göstermektedir:

```
i = 1
print "i\ti**2\ti**3\ti**5\ti**10\ti**20"
while i <= 10:
    print i, '\t', i**2, '\t', i**3, '\t', i**5, '\t', i**10, '\t',
i**20
    i += 1
```

Bu program 1'den 10'a kadar sayıların farklı üslerinin değerlerini ekranda bir tablo olarak görüntüler. Değerleri hizalamak için tab (\t) karakterinden yararlanır, ancak bu tab genişliğinden, 8 karakterden büyük bir sonuç oluştuğunda şekli bozmaktadır:

```
i       i**2    i**3    i**5    i**10   i**20
1       1       1       1       1       1
2       4       8       32      1024    1048576
3       9       27      243     59049   3486784401
```

```
4        16       64       1024     1048576          1099511627776
5        25       125      3125     9765625          95367431640625
6        36       216      7776     60466176         3656158440062976
7        49       343      16807    282475249        79792266297612001
8        64       512      32768    1073741824       1152921504606846976
9        81       729      59049    3486784401       12157665459056928801
10       100      1000     100000   10000000000      100000000000000000000
```

Olası bir çözüm tab genişliğini değiştirmektir, ancak ilk sütun hali hazırda gerekenden fazla boşluğa sahiptir. En iyi çözüm her bir sütunun genişliğini birbirlerinden bağımsız, farklı boyutlar halinde ayarlamaktır. Tahmin edebileceğiniz gibi, bu çözümü karakter dizisi biçimlendirmeyle sağlıyoruz:

```
i = 1
print "%-4s%-5s%-6s%-8s%-13s%-15s" %
     ('i', 'i**2', 'i**3', 'i**5', 'i**10', 'i**20')
while i <= 10:
   print "%-4d%-5d%-6d%-8d%-13d%-15d" % (i, i**2, i**3, i**5, i**10,
i**20)
   i += 1
```

Bu sürümü çalıştırdığımızda aşağıdaki çıktıyı elde ederiz:

```
i   i**2 i**3  i**5    i**10        i**20
1   1    1     1       1            1
2   4    8     32      1024         1048576
3   9    27    243     59049        3486784401
4   16   64    1024    1048576      1099511627776
5   25   125   3125    9765625      95367431640625
6   36   216   7776    60466176     3656158440062976
7   49   343   16807   282475249    79792266297612001
8   64   512   32768   1073741824   1152921504606846976
9   81   729   59049   3486784401   12157665459056928801
10  100  1000  100000  10000000000  100000000000000000000
```

Dönüştürme belirtimindeki her bir % karakterinden sonra gelen - karakteri sol dayalı hizalamayı belirtir. Sayısal değerler en küçük uzunluğu tanımlar, böylece %-13d sola dayalı hizalanmış ve en az 13 karakter uzunluğunda bir tamsayıdır.

7.14 Sözlük

bileşik veri tipi:
Değerleri kendileri de bir değer olan bileşenlerden veya öğelerden üretilmiş olan veri tipidir.

indis:
Sıralı bir kümenin (karakter dizisindeki karakterler) bir üyesini seçmek için kullanılan değer veya değişkendir.

gezinme:

Bir kümenin öğeleri arasında sırayla dolaşma ve her bir öğe üzerinde benzer işlemleri gerçekleştirmedir.

dilim:
Bir karakter dizisinin indis aralığıyla belirlenmiş bir parçasıdır. Daha genel olarak, Pyton içerisinde bulunan herhangi bir dizinin dilim işleci ile (dizi[başlangıç:bitiş]) alt dizisidir.

değişmez (immutable):
Öğelerine ayrı ayrı yeni bir değer atanamayan bileşik veri tipidir.

isteğe bağlı parametre:
Fonksiyon başlığında kendisine varsayılan değerin atandığı ve eğer fonksiyon çağrımında ilgili parametre için bir argüman verilmezse bu varsayılan değeri kullanacak olan parametredir.

varsayılan (default) değer:
İsteğe bağlı bir parametreye verilen ve fonksiyon çağrımında argüman verilmemesi durumunda kullanılacak olan değerdir.

nokta gösterimi:
Bir modül içerisindeki fonksiyonlara erişmek için **nokta işlecinin** (.) kullanılmasıdır.

docstring:
Bir fonksiyon veya modül tanımlamasındaki ilk satırda bulunan karakter dizisi şeklindeki sabittir (ileride göreceğimiz gibi sınıf ve metot tanımlamalarında da bulunacak). Docstringler kod ile belgelendirmeyi birleştirmek için kullanılan geleneksel bir yöntemdir. Docstringler ayrıca doctest modülü tarafından otomatik sınama için kullanılmaktadır.

boş karakterler (whitespace):
İmleci bir şey görüntülemeden hareket ettiren herhangi bir karakterdir. string.whitespace sabiti tüm boş karakterleri içerir.

7.15 Alıştırmalar

1. Aşağıdaki kodu Ouack ve Quack kelimeleri doğru bir şekilde yazılacak şekilde düzeltin:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
    print letter + suffix
```

2. Aşağıdaki fonksiyonu count_letters isminde ve karakter dizisi ile karakteri argüman alacak şekilde genelleştirip bir fonksiyon şekline getirin:

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
```

```
print count
```

3. count_letters fonksiyonunu tekrar yazarak, karakter dizisini gezinmesi yerine find fonksiyonunu (8.10 bölümündeki sürümü) sayılan harfin yeni bulunuşları için kullanılacak isteğe bağlı bir parametre ile tekrar tekrar çağırsın.
4. is_lower fonksiyonunun hangi sürümünün daha hızlı olduğunu düşünüyorsunuz? Bir sürümü diğerine tercih etmek için hızdan başka nedenler düşünebiliyor musunuz?
5. stringtools.py isminde bir dosya yaratıp aşağıdakiler içerisine koyun:

```
def reverse(s):
    """
      >>> reverse('happy')
      'yppah'
      >>> reverse('Python')
      'nohtyP'
      >>> reverse("")
      ''
      >>> reverse("P")
      'P'
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

   Doctestlerin başarılı olması için reverse fonksiyon gövdesini yazın.
6. stringtools.py içerisine mirror kodunu koyun.

```
def mirror(s):
    """
      >>> mirror("good")
      'gooddoog'
      >>> mirror("yes")
      'yessey'
      >>> mirror('Python')
      'PythonnohtyP'
      >>> mirror("")
      ''
      >>> mirror("a")
      'aa'
    """
```

   Doctestler ile tanımlanmış işlemi yapabilmesi için fonksiyon gövdesini yazın.
7. stringtools.py içerisine remove_letter fonksiyonunu ekleyin.

```
def remove_letter(letter, strng):
    """
```

```
    >>> remove_letter('a', 'apple')
    'pple'
    >>> remove_letter('a', 'banana')
    'bnn'
    >>> remove_letter('z', 'banana')
    'banana'
    >>> remove_letter('i', 'Mississippi')
    'Mssssppp'
  """
```

Doctestler ile tanımlanmış görevi yapabilmesi için fonksiyon gövdesini yazın.

8. Son olarak, aşağıdaki fonksiyonların gövdesini her seferinde biri olacak şekilde doldurun.

```
def is_palindrome(s):
  """
    >>> is_palindrome('abba')
    True
    >>> is_palindrome('abab')
    False
    >>> is_palindrome('tenet')
    True
    >>> is_palindrome('banana')
    False
    >>> is_palindrome('straw warts')
    True
  """

def count(sub, s):
  """
    >>> count('is', 'Mississippi')
    2
    >>> count('an', 'banana')
    2
    >>> count('ana', 'banana')
    2
    >>> count('nana', 'banana')
    1
    >>> count('nanan', 'banana')
    0
  """

def remove(sub, s):
  """
    >>> remove('an', 'banana')
    'bana'
    >>> remove('cyc', 'bicycle')
    'bile'
```

```
      >>> remove('iss', 'Mississippi')
      'Missippi'
      >>> remove('egg', 'bicycle')
      'bicycle'
    """

def remove_all(sub, s):
    """
      >>> remove_all('an', 'banana')
      'ba'
      >>> remove_all('cyc', 'bicycle')
      'bile'
      >>> remove_all('iss', 'Mississippi')
      'Mippi'
      >>> remove_all('eggs', 'bicycle')
      'bicycle'
    """
```

Tüm doctestlerin başarılı olması gerektiğini unutmayın.

9. Aşağıdaki biçimlendirilmiş karakter dizilerin Python kabuğunda sınayıp sonuçları inceleyin:
    1. "%s %d %f" % (5, 5, 5)
    2. "%-.2f" % 3
    3. "%-10.2f%-10.2f" % (7, 1.0/2)
    4. print " $%5.2f\n $%5.2f\n $%5.2f" % (3, 4.5, 11.2)
10. Aşağıdaki biçimlendirilmiş karakter dizilerinde hata var, hataları düzeltin:
    1. "%s %s %s %s" % ('this', 'that', 'something')
    2. "%s %s %s" % ('yes', 'no', 'up', 'down')
    3. "%d %f %f" % (3, 3, 'three')

8. Örnek Çalışma: Catch

8.1 Başlangıç

İlk örnek çalışmamızda GASP paketi yardımıyla basit bir oyun geliştireceğiz. Oyunda bir top pencerenin solundan sağına hareket edecek ve siz bu topu yakalamak için sağ taraftaki bir eldiveni hareket ettireceksiniz.

8.2 Topu hareket ettirmek için while kullanımı

while cümeleleri gasp ile birlikte programa hareket katmak için kullanılabilir. Aşağıdaki program siyah bir topu 800 x 600 piksel boyutlarında bir grafik tuvalinde hareket ettirmektedir. Bu programı pitch.py isminde bir dosyaya koyun:

```
from gasp import *
```

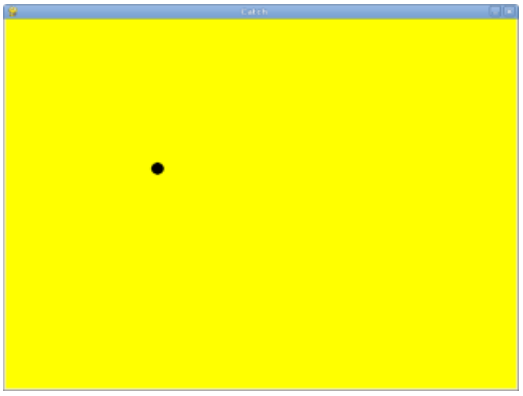
```
begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)

ball_x = 10
ball_y = 300
ball = Circle((ball_x, ball_y), 10, filled=True)
dx = 4
dy = 1

while ball_x < 810:
    ball_x += dx
    ball_y += dy
    move_to(ball, (ball_x, ball_y))
    update_when('next_tick')

end_graphics()
```

Top ekran boyunca hareket ettikçe, aşağıdaki gibi bir grafik penceresi göreceksiniz:



Programdaki ilk bir kaç döngüyü inceleyerek x ve y değişkenlerine ne olduğunu anlamaya çalışın.

Bu örnekten GASP hakkında öğrenilecek yeni şeyler:

- begin_graphics fonksiyonu grafik tuvali için genişlik, yükseklik, başlık ve arka plan rengi argümanlarını alıyor.
- set_speed kare (çerçeve) sayısını (Frame Rate) saniyedeki kare sayısı (fps - Frame Per Second) olarak alıyor.
- Circle(...) fonksiyonuna filled=True eklediğimizde ortaya çıkan çemberin dolu olması sağlanıyor.
- ball = Circle çemberin (çemberin gerçekte ne olduğunu daha sonra konuşacağız) ball isimli bir değişkende saklanmasını sağlıyor, böylece daha sonra da erişilebiliyor.
- GASP'taki move_to fonksiyonu programcının bir şekli (bu örnekte top şekli) verilen konum parametresine göre konumlandırmasını sağlıyor.

- update_when fonksiyonu GASP'ın belli bir olay olana kadar eylemi geciktirmesini sağlıyor. 'next_tick' olayı bir sonraki çerçeveye kadar bekler, bu da set_speed ile belirlenmiş olan çerçeve sayısıdır. update_when için diğer geçerli argümanlar 'key_pressed' (tuşa basıldı) ve 'mouse_clicked' (fare tıklandı) argümanlarıdır.

8.3 Aralıkları değiştirmek

Oyunumuzu daha ilginç kılmak için, topumuzun hız ve doğrultusunu değiştirmek isteyebiliriz. GASP random_between(alt, ust) isminde ve alt ve ust aralığında **rastgele**tamsayı değer üreten bir fonksiyona sahiptir. Bunun nasıl çalıştığını görmek için, aşağıdaki programı çalıştırın:

```
from gasp import *

i = 0
while i < 10:
    print random_between(-5, 5)
    i += 1
```

Fonksiyon her çağrıldığında -5 ve 5 aralığında küçük veya büyük bir rastgele tamsayı değer seçilmektedir. Bu programı çalıştırdığımızda aşağıdaki sonuçları görüyoruz:

```
-2
-1
-4
1
-2
3
-5
-3
4
-5
```

Siz muhtemelen daha farklı sayılar göreceksiniz.

random_between fonksiyonunu topun yönünü değiştirmek için kullanalım. pitch.py programında y değişkenine 1 atayan kısmı değiştirelim:

```
dy = 1
```

yukarıdaki kısım yerine -4 ve 4 arasında rastgele bir değer üreten aşağıdaki satırı yazalım:

```
dy = random_between(-4, 4)
```

8.4 Topun sekmesini sağlamak

Bu programın yeni halini çalıştırdığınızda, topun arada sırada ekranın üst veya alt sınırını aştığını göreceksiniz. Bunu önlemek için topun kenarlardan sekmesini, dy'nin işaretini değiştirip topu zıt dikey yöne gönderelim

Aşağıdaki kodu pitch.py programında while döngüsünün gövdesinde ilk satıra ekleyin:

```
if ball_y >= 590 or ball_y <= 10:
    dy *= -1
```

Nasıl davrandığını iyice anlamak için programı bir kaç kez çalıştırın.

8.5 break cümlesi

break cümlesi bir döngünün gövdesini hemen terketmek için kullanılır. Aşağıdaki program basit bir tahmin oyununu gerçekleştirmektedir:

```
from gasp import *

number = random_between(1, 1000)
guesses = 1
guess = input("Guess the number between 1 and 1000: ")

while guess != number:
    if guess > number:
        print "Too high!"
    else:
        print "Too low!"
    guess = input("Guess the number between 1 and 1000: ")
    guesses += 1

print "\n\nCongratulations, you got it in %d guesses!\n\n" % guesses
```

break cümlesi yazarak bu programda input cümlesinin tekrar tekrar kullanımının önüne geçebiliriz:

```
from gasp import *

number = random_between(1, 1000)
guesses = 0

while True:
    guess = input("Guess the number between 1 and 1000: ")
    guesses += 1
    if guess > number:
        print "Too high!"
    elif guess < number:
        print "Too low!"
```

```
    else:
        print "\n\nCongratulations, you got it in %d guesses!\n\n" %
guesses
        break
```

Bu program matematiksel bir yasa olan **üçe bölünme (trichotomy)**yi (verilen gerçel sayılar a ve b için a > b, a < b, veya a = b) kullanır. Her iki sürümde her ne kadar 15 satır olsa da, ikinci sürümdeki mantığın daha anlaşılır olduğunu söyleyebiliriz.

Bu programı guess.py isminde bir dosyaya koyun.

8.6 Klavyeyi yanıtlamak

Aşağıdaki program bir çember (veya eldiven) yaratmaktadır, bu çember klavye girdisine göre tepki vermektedir. j veya k tuşlarına basmak eldiveni sırasıyla yukarı ve aşağı hareket ettirmektedir. Bu programı mitt.py isminde bir dosyaya yazın:

```
from gasp import *

begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)

mitt_x = 780
mitt_y = 300
mitt = Circle((mitt_x, mitt_y), 20)

while True:
    if key_pressed('k') and mitt_y <= 580:
        mitt_y += 5
    elif key_pressed('j') and mitt_y >= 20:
        mitt_y -= 5

    if key_pressed('escape'):
        break

    move_to(mitt, (mitt_x, mitt_y))
    update_when('next_tick')

end_graphics()
```

mitt.py'yi çalıştırın, j ve k tuşlarına basarak eldiveni ekranda yukarı aşağı hareket ettirin.

8.7 Çarpışma kontrolü

Aşağıdaki program iki topu birbirine doğru ekranın karşıt yönlerinden hareket ettirir. Çarpıştıklarında, her iki top kaybolur ve program sonlanır:

```
from gasp import *

def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)

ball1_x = 10
ball1_y = 300
ball1 = Circle((ball1_x, ball1_y), 10, filled=True)
ball1_dx = 4

ball2_x = 790
ball2_y = 300
ball2 = Circle((ball2_x, ball2_y), 10)
ball2_dx = -4

while ball1_x < 810:
    ball1_x += ball1_dx
    ball2_x += ball2_dx
    move_to(ball1, (ball1_x, ball1_y))
    move_to(ball2, (ball2_x, ball2_y))
    if distance(ball1_x, ball1_y, ball2_x, ball2_y) <= 20:
        remove_from_screen(ball1)
        remove_from_screen(ball2)
        break
    update_when('next_tick')

sleep(1)
end_graphics()
```

Bu programı collide.py isminde bir dosyaya koyup, çalıştırın.

8.8 Parçaları birleştirelim

Hareket eden topu, eldiveni, ve çarpışma kontrolünü birleştirmek için tek bir while döngüsüne ihtiyaç duyuyoruz. Bu döngüde her bir olay sırasıyla gerçekleşir:

```
from gasp import *

def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)

ball_x = 10
ball_y = 300
```

```
ball = Circle((ball_x, ball_y), 10, filled=True)
dx = 4
dy = random_between(-4, 4)

mitt_x = 780
mitt_y = 300
mitt = Circle((mitt_x, mitt_y), 20)

while True:
    # move the ball
    if ball_y >= 590 or ball_y <= 10:
        dy *= -1
    ball_x += dx
    if ball_x > 810:              # the ball has gone off the screen
        break
    ball_y += dy
    move_to(ball, (ball_x, ball_y))

    # check on the mitt
    if key_pressed('k') and mitt_y <= 580:
        mitt_y += 5
    elif key_pressed('j') and mitt_y >= 20:
        mitt_y -= 5

    if key_pressed('escape'):
        break

    move_to(mitt, (mitt_x, mitt_y))

    if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:  # ball is
caught
        remove_from_screen(ball)
        break

    update_when('next_tick')

end_graphics()
```

Bu programı catch.py isminde bir dosyaya koyup, bir kaç kez çalıştırın. Bazı durumlarda topu yakalamayı, diğer bazı durumlarda da kaçırmayı deneyin.

8.9 Metin gösterimi

Bu program her kullanıcı hem de bilgisayar için ekran üzerinde skoru gösterir. 0 veya 1 olan rastgele bir sayı üretir (yazı tura gibi) ve eğer değer 1 ise kullanıcıya 1 puan ekler, aksi durumda bilgisayara bir puan ekler. Daha sonra ekrandaki metini günceller.

```
from gasp import *
```

```
begin_graphics(800, 600, title="Catch", background=color.YELLOW)
set_speed(120)

player_score = 0
comp_score = 0

player = Text("Player: %d Points" % player_score, (10, 570), size=24)
computer = Text("Computer: %d Points" % comp_score, (640, 570),
size=24)

while player_score < 5 and comp_score < 5:
    sleep(1)
    winner = random_between(0, 1)
    if winner:
        player_score += 1
        remove_from_screen(player)
        player = Text("Player: %d Points" % player_score, (10, 570),
size=24)
    else:
        comp_score += 1
        remove_from_screen(computer)
        computer = Text("Computer: %d Points" % comp_score, (640,
570), size=24)

if player_score == 5:
    Text("Player Wins!", (340, 290), size=32)
else:
    Text("Computer Wins!", (340, 290), size=32)

sleep(4)

end_graphics()
```

Bu programı scores.py isminde bir dosyaya yazıp, çalıştırın.

catch.py programını kazananı gösterecek şekilde güncelleyebiliriz. if ball_x > 810: koşulundan hemen sonra, aşağıdakileri ekleyin:

```
Text("Bilgisayar kazandı!", (340, 290), size=32)
sleep(2)
```

Oyuncunun kazandığını gösterme işi size bir alıştırma olarak bırakılmıştır.

8.10 Soyutlama

Programımız biraz karmaşıklaşmaya başladı. Durumu daha da kötü kılmaz için, karmaşasını daha da arttıracağız. Bir sonraki geliştirme aşaması **içiçe döngü** gerektiriyor. Dış döngü bilgisayar veya kullanıcı kazanma sayısına ulaşana kadar

turları tekrarlayacak. İç döngü ise halihazırda sahip olduğumuz döngü olacak, bu döngü tek bir tur oynanmasını sağlıyor, topu ve eldiveni hareket ettiriyor ve yakalama veya kaçırma gerçekleşti mi diye kontrol ediyor.

Araştırmacılar bilişsel görevlerimizi işleme yeteneğimizin açık sınırları olduğunu gösteriyor (George A. Miller tarafından yazılmış olan The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information incelenebilir). Program karmaşıklaştıkça, deneyimli bir programcının bile onu geliştirmesi, anlaması ve bakımını yapması zorlaşıyor.

Artan karmaşıklığı karşılayabilmek için, ilişkili cümleleri fonksiyonlar şeklinde ayrıştırmak, program ayrıntılarını gizleyerek **soyutlama** yapmak gerekir. Bu aynı kavramdaki, kapsamdaki programlama cümlelerine bir grup şeklinde davranmamızı sağlar, zihinsel **veri yolu**muzu boşaltarak sonraki görevlere bizi hazırlar. Soyutlamayı kullanma yeteneği bilgisayar programlamadaki en güçlü fikirlerden biridir.

Aşağıdaki catch.py programının bitmiş halini inceleyebilirsiniz:

```
from gasp import *

COMPUTER_WINS = 1
PLAYER_WINS = 0
QUIT = -1


def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5


def play_round():
    ball_x = 10
    ball_y = random_between(20, 280)
    ball = Circle((ball_x, ball_y), 10, filled=True)
    dx = 4
    dy = random_between(-5, 5)

    mitt_x = 780
    mitt_y = random_between(20, 280)
    mitt = Circle((mitt_x, mitt_y), 20)

    while True:
        if ball_y >= 590 or ball_y <= 10:
            dy *= -1
        ball_x += dx
        ball_y += dy
        if ball_x >= 810:
            remove_from_screen(ball)
            remove_from_screen(mitt)
            return COMPUTER_WINS
```

```
        move_to(ball, (ball_x, ball_y))

        if key_pressed('k') and mitt_y <= 580:
            mitt_y += 5
        elif key_pressed('j') and mitt_y >= 20:
            mitt_y -= 5

        if key_pressed('escape'):
            return QUIT

        move_to(mitt, (mitt_x, mitt_y))

        if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:
            remove_from_screen(ball)
            remove_from_screen(mitt)
            return PLAYER_WINS

        update_when('next_tick')


def play_game():
    player_score = 0
    comp_score = 0

    while True:
        pmsg = Text("Player: %d Points" % player_score, (10, 570),
size=24)
        cmsg = Text("Computer: %d Points" % comp_score, (640, 570),
size=24)
        sleep(3)
        remove_from_screen(pmsg)
        remove_from_screen(cmsg)

        result = play_round()

        if result == PLAYER_WINS:
            player_score += 1
        elif result == COMPUTER_WINS:
            comp_score += 1
        else:
            return QUIT

        if player_score == 5:
            return PLAYER_WINS
        elif comp_score == 5:
            return COMPUTER_WINS


begin_graphics(800, 600, title="Catch", background=color.YELLOW)
```

```
set_speed(120)

result = play_game()

if result == PLAYER_WINS:
    Text("Player Wins!", (340, 290), size=32)
elif result == COMPUTER_WINS:
    Text("Computer Wins!", (340, 290), size=32)

sleep(4)

end_graphics()
```

Bu örnekten öğrenilecek yeni şeyler:

- İyi düzenleme (örgütleme) pratiklerini izlemek programların okunurluğunu arttıracaktır. Programlarınızda aşağıdaki düzenleme yaklaşımını kullanın:
    - import cümleleri
    - genel (global) sabitler
    - fonksiyon tanımlamaları
    - programın ana gövdesi
- COMPUTER_WINS, PLAYER_WINS, ve QUIT gibi sembolik **sabitler** programın okunurluğunu arttırmak için kullanılabilir. İsim sabitlerinin tüm harflerini büyük yazmak gelenekselleşmiş bir yaklaşımdır. Python'da sabite yeni değer atamak (ve atamamak) tamamen programcının kontrolündedir, dil bunu kısıtlamak için bir kolay yol sunmamaktadır (çoğu diğer programlama dilleri bu kolay yolu sunar).
- 8.8 nolu bölümde geliştirilen programı aldık ve play_round() isminde bir fonksiyon içerisine yerleştirdik. play_round fonksiyonu programın üstünde tanımladığımız sabitleri kullanmaktadır. Kendisine atanan sayısal değeri hatırlamaktansa COMPUTER_WINS sabit ismini hatırlamak daha kolaydır.
- play_game() ismindeki yeni fonksiyon player_score ve comp_score için değişkenler yaratır. while döngüsü kullanarak play_round fonksiyonunu tekrar tekrar çağırır. Her bir çağrımda sonucu kontrol edip, skoru uygun şekilde günceller. Son olarak eğer bilgisayar veya kullanıcı 5 puana ulaşırsa, play_game fonksiyonu programın ana gövdesine kazananı geri döndürür, ana program gövdesi de kazananı görüntüleyip programı sonlandırır.
- İki result değişkeni vardır --- bir tanesi play_game fonksiyonunda, diğeri de programın ana gövdesinde. Her ikisi de aynı isme sahip olsa da, farklı isim uzayındalar, birbirleriyle bir ilişkileri yoktur. Her fonksiyon kendi isim uzayını yaratır ve fonksiyon gövdesinde tanımlı isimler, fonksiyon gövdesinin dışında görünür değildir. İsim uzayları sonraki bölümde ayrıntılı olarak anlatılacaktır.

8.11 Sözlük

rastgele:

Herhangi bir desene sahip olmayan. Tahmin edilemeyen. Bilgisayarlar tahmin edilebilir olarak tasarlanmıştır, ve bilgisayardan gerçekten rastgele bir değer almak mümkün değildir. Bazı fonksiyonlar rastgeleymiş gibi görünen bazı sayı dizileri üretir, bunlar, Python'da da elde ettiğimiz yalancı rastgele (pseudorandom)değerlerdir.

üçe bölünme (trichotomy):
Verilen gerçel sayılar a ve b için, takip eden ilişkilerden sadece bir tanesinin sağlanmasıdır: a < b, a > b, veya a = b. Böylece eğer ilişkilerden ikisinin yanlış olduğunu belirlediğinizde, diğer ilişkinin doğru olduğunu kabul edebilirsiniz.

iç döngü:
Bir başka döngü içinde yer alan döngü.

soyutlama:
Bir kavramın bilgi içeriğinin azaltılmasıyla sağlanan genelleştirme. Python'daki fonksiyonlar cümleleri gruplamak tek bir isim altında gruplamak için, ayrıntıları soyutlamak ve programın anlaşılırlığını kolaylaştırmak için kullanılabilir.

sabit:
Program çalışması sırasında değişmeyen nümerik değer. Sabitleri temsil etmek için tamamen büyük harflerden oluşan isimler kullanmak gelenekselleşmiştir. Python programlarında sabitlerin değişmemesi programcıya bırakılmıştır, Python'da gerçek sabitleri sağlamak üzere herhangi bir dil düzeneği yoktur.

8.12 Alıştırmalar

1. mitt.py çalışırken <Escape> tuşuna basmanız durumunda ne olur? Programda bu davranışı yaratan iki satırı bulup nasıl çalıştığını anlatın.
2. guess.py'deki sayaç değişkeninin ismi nedir? Uygun bir strateji ile, doğruya ulaşmak için gereken maksimum tahmin sayısı 11'dir. Bu strateji nedir?
3. mitt.py programındaki eldiven grafik penceresinin en üst veya en altına gelirse ne olur? Programda bu davranışı kontrol eden satırları bulun ve satırın nasıl çalıştığını ayrıntılı olarak anlatın.
4. collide.py programındaki ball1_dx değişkeninin değerini 2 olarak değiştirin. Program hangi farklı şekilde davranmaktadır? Şimdi ball1_dx değişkenini tekrar 4 olarak değiştirin, ve ball2_dx değerini -2 olarak değiştirin. Ayrıntılı olarak bu değişikliklerin davranışta yarattığı farklılıkları anlatın.
5. collide.py programındaki break cümlesini yorum haline (başına # koyarak yapabilirsiniz) getirin. Programın davranışında herhangi bir fark görüyor musunuz? Şimdi remove_from_screen(ball1) cümlesini de yorum haline getirin. Şimdi ne oluyor? Bu iki satırın birlikte gerekli davranışı nasıl ortaya çıkardığını iyice anlamak için yorum haline getirme ve normal hale getirme denemeleri yapın, ayrıntılı olarak bu cümlelerin nasıl çalıştığını anlatın.
6. Aşağıdaki satırları 8.8 bölümündeki catch.py programında nereye eklemelisiniz ki top yakalandığında programda mesaj görüntülensin?

```
Text("Player Wins!", (340, 290), size=32)
sleep(2)
```

7. catch.py programının son sürümünde play_round fonksiyonunun işlenmesi esnasında <Escape> tuşuna basıldığındaki çalışma akışını izleyin. Bu tuşa bastığınızda ne oluyor? Neden?
8. catch.py programının son sürümündeki ana gövdeyi yazın. Her bir satırın ne yaptığını ayrıntılı olarak anlatın. Oyunu başlatan fonksiyonu hangi cümle çağırmaktadır?
9. Top ve eldiveni göstermekle sorumlu fonksiyonu belirtin. Bu fonksiyon tarafından sağlanan diğer işlemler nelerdir?
10. Hangi fonksiyon skorun takibini yapmaktadır? Bu ayrıca skoru gösteren fonksiyon mudur? Yanıtınızı kanıtlamak için kodun bu işlemleri gerçekleştiren ilgili kısımlarını anlatın.

8.13 Proje: pong.py

Pong ilk ticari bilgisayar oyunlarından biriydi. Büyük P ile kendisi bir kayıtlı markadır, ama pong ifadesi herhangi bir masa tenisi benzeri raketli, toplu oyunları tanımlamak için kullanılır.

catch.py kendi pong sürümümüzü geliştirmek için gereken programlama araçlarını hali hazırda içermektedir. Artırımsal olarak catch.py programını pong.py programına değiştirmek bu projenin hedefidir, aşağıdaki alıştırmaları tamamladığınızda bu hedefe ulaşmış olacaksınız:

1. catch.py programını pong1.py programı olarak kopyalayın ve eldiveni Circle yerine Box kullanarak bir rakete çevirin. Box hakkında ayrıntılı bilgi için GASP ekine bakabilirsiniz. raketin ekranda kalması için gereken ayarlamaları yapın.
2. pong1.py programını pong2.py programı şeklinde kopyalayın. distance fonksiyonunu boolean bir fonksiyon olan hit(bx, by, r, px, py, h) ile değiştirin, bu fonksiyon topun dikey konumu (by) raketin altında veya üstünde kaldığında ve topun yatay konumu (bx) raketin önünden çapa (r) eşit veya daha küçük uzaklıkta iseTrue döndürür. hit fonksiyonunu topun rakete değmesini sınamak için kullanın, top rakete değdiğinde (hit sonuç olarak True döndürdüğünde) topu zıt yatay yönde sektirin. Bitmiş fonksiyonunuz aşağıda doctestleri geçmelidir:

```
def hit(bx, by, r, px, py, h):
    """
      >>> hit(760, 100, 10, 780, 100, 100)
      False
      >>> hit(770, 100, 10, 780, 100, 100)
      True
      >>> hit(770, 200, 10, 780, 100, 100)
      True
      >>> hit(770, 210, 10, 780, 100, 100)
```

```
        False
    """
```

Son olarak, skoru top sol taraftan çıktığında kullanıcıya bir puan verecek şekilde değiştirin.

3. pong2.py programını pong3.py programı olarak kopyalayın. Ekranın sol tarafına yeni bir raket ekleyin. Bu raket 'a' tıklandığında yukarı, 's' tıklandığında aşağı hareket etsin. Top için başlangıç noktasını ekranın ortası (400, 300) olarak değiştirin, her turun başında sol veya sağa rastgele hareket edecek şekilde değiştirin.

9. Listeler

Liste (list) sıralı değer kümesidir, her bir değer bir indis ile erişilmektedir. Listeyi oluşturan değerlere **öğe**ler denilmektedir. Listeler karakter dizilerine, karakterlerden oluşan sıralı kümeler, benzer, tek farkı listelerin öğeleri herhangi bir tipte olabilir. Listeler ve karakter dizileri -- ve sıralı küme şeklinde davranan diğer şeyler -- **diziler (sequences)**adını almaktadır.

9.1 Liste değerleri

Liste yaratmanın bir çok yolu vardır; en kolayı öğeleri köşeli parantezlerle ([ve]) kapsamaktır:

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

İlk örnek dört tamsayıdan oluşan bir listedir. İkincisi ise üç karakter dizisinin bir listesidir. Listenin öğeleri aynı tipten olmak zorunda değildir. Aşağıdaki liste bir karakter dizisi, bir kayan noktalı sayı, bir tamsayı ve bir başka liste (mirabile dictu) içermektedir:

```
["hello", 2.0, 5, [10, 20]]
```

Liste içerisindeki listeye **içiçe** adı verilir.

Son olarak, öğe içermeyen özel bir liste vardır. Boş liste adını alır ve [] ile temsil edilir.

Nümerik 0 değeri ve boş karakter dizisi gibi boş liste de boolean değer olarak yanlıştır ("false"):

```
>>> if []:
...     print 'This is true."
... else:
...     print 'This is false."
...
This is false.
```

```
>>>
```

Liste yaratmanın bütün bu yöntemleri yanında, liste değerlerini başka değişkenlere atayamasak veya listeleri fonksiyonlara parametre olarak geçiremezsek bu bir hayal kırıklığı yaratır. Ancak bunları yapabiliyoruz:

```
>>> vocabulary = ["ameliorate", "castigate", "defenestrate"]
>>> numbers = [17, 123]
>>> empty = []
>>> print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
```

9.2 Öğelere erişme

Listenin öğelerine erişme sözdizimi, karakter dizisindeki karakterlere erişimin sözdizimiyle aynıdır. Köşeli parantez ([], boş liste ile karıştırılmamalıdır) işleci kullanılmaktadır. Köşeli parantezlerin içerisindeki ifade indisi belirtmektedir. İndislerin 0'la başladığını hatırlayın:

```
>>> print numbers[0]
17
```

Her hangi bir tamsayı ifade indis olarak kullanılabilir:

```
>>> numbers[9-8]
5
>>> numbers[1.0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers
```

Olmayan bir öğeyi okumak ve yazmak isterseniz, bir çalışma zamanı hatasıyla karşılaşırsınız:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Eğer indis negatif bir sayı ise, listenin sonundan geriye doğru sayacaktır:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

numbers[-1] listenin son öğesi, numbers[-2] sondan önceki öğe, ve numbers[-3] yok.

Döngü değişkenini liste indisi olarak kullanmak sık yapılan bir şeydir.

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
    print horsemen[i]
    i += 1
```

Bu while döngüsü 0'dan 4'e kadar saymaktadır. Döngü değişkeni i 4 olduğunda, koşul sağlanmayacak ve döngü bitecektir. Döngünün gövdesi *i* değişkeni 0,1,2, ve 3 iken işletilecektir.

Döngüdeki her bir turda, i değişkeni listede indis olarak kullanılmakta, ve listenin i. öğesi ekranda görüntülenmektedir. Bu şekildeki kullanıma **liste dolaşımı** adı verilmektedir.

9.3 Liste boyutu

len fonksiyonu listenin boyutunu döndürür, boyut öğelerin sayısına eşittir. Bu değeri bir döngünün üst değeri olarak kullanmak, sabit bir sayı kullanmaktan daha mantıklıdır. Bu şekilde listenin boyutu değiştiğinde, programdaki tüm döngüleri değiştirmeniz gerekmez. Hepsi doğru bir şekilde her türlü boyutta liste için çalışır:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
num = len(horsemen)
while i < num:
    print horsemen[i]
    i += 1
```

Döngü gövdesinin son çalışmasına, i len(horsemen) - 1 eşittir, bu da son öğenin indisidir. i'nin değeri len(horsemen)'e eşit olduğunda, koşul sağlanmayacak ve döngü gövdesi işletilmeyecektir, bu iyi bir şeydir çünkü len(horsemen) geçerli bir indis değildir.

Her ne kadar liste başka bir liste içerse de, içteki liste tek bir öğe olarak sayılacaktır. Aşağıdaki listenin boyutu 4'tür:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

9.4 Liste üyeliği

in işleci bir dizi içerisindeki üyeliği sınayan bir boolean işleçtir. Daha önce karakter dizileriyle kullanmıştır, bu işleç liste ve diğer dizilerde de işe yarar:

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
True
>>> 'debauchery' in horsemen
False
```

pestilence öğesi horsemen listesinin bir üyesi olduğundan, in işleci True sonucunu döndürmektedir. Ancak debauchery listede olmadığı için, in işleci Falsedöndürmektedir.

not işlecini in işleciyle birlikte bir öğe listenin üyesi mi değil mi sınaması için kullanabiliriz:

```
>>> 'debauchery' not in horsemen
True
```

9.5 Liste işlemleri

The + işleci listeleri birleştirir:

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

Benzer şekilde, * işleci verilen sayıda tekrarlar:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

İlk örnek [0] listesini dört kere tekrarlamıştır. İkinci örnek ise [1, 2, 3] listesini üç kere tekrarlamaktadır.

9.6 Liste dilimleri

Karakter dizilerinde gördüğümüz dilim işlemleri listelerde de işe yaramaktadır:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

9.7 The range fonksiyonu

Sıralı tamsayı içeren listeler sıklıkla karşılaştığımız bir durumdur, bu yüzden Python bunları yaratmak için kolay bir yol sunmaktadır:

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

range fonksiyonu iki argüman alır ve ilk argümandan ikinci argümana kadar ilk argümanı içeren, ancak ikinci argümanı içermeyen tüm tamsayıları içeren bir liste döndürür..

range fonksiyonunun iki biçimi daha vardır. Tek parametre alan biçimi 0 ile başlayan bir liste döndürür:

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

Eğer üçüncü bir parametre alıyorsa, bu parametre peşpeşe değerler arasındaki artış miktarını, **adım sayısını** belirler. Aşağıdaki örnek 1'den 10'a kadar 2'şerli artan bir listeyi göstermektedir:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Eğer adım sayısı negatif ise, başlangıç parametresi bitiş parametresinden büyük olmalıdır.

```
>>> range(20, 4, -5)
[20, 15, 10, 5]
```

yoksa sonuç boş bir liste olacaktır.

```
>>> range(10, 20, -5)
[]
```

9.8 Listeler değiştirilebilir

Karakter dizilerinin aksine listeler **değiştirilebilir (mutable)**, yani öğelerini değiştirebildiğimiz yapılardır. Köşeli parantez işlecini sol tarafta kullanarak, listenin öğelerini güncelleyebiliriz:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

Listeye uyguladığımız köşe parantezi işleci ifadenin herhangi bir yerinde yer alabilir. Atamanın sol tarafından yer aldığında, listenin bir öğesini değiştirir,

yukarıdaki örnekte gördüğümüz üzere fruit listesinin ilk öğesi 'banana''dan 'pear''a, ve son öğesi 'quince''tan 'orange''a değiştirilmiştir. Bir listenin öğesine yapılan atamaya **öğe ataması** adı verilmektedir. Öğe atamaları karakter dizilerinde işe yaramamaktadır:

```
>>> my_string = 'TEST'
>>> my_string[2] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

ancak listelerde işe yaramaktadır:

```
>>> my_list = ['T', 'E', 'S', 'T']
>>> my_list[2] = 'X'
>>> my_list
['T', 'E', 'X', 'T']
```

Dilim işleciyle listenin birden fazla öğesini tek seferde güncelleyebiliriz:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = ['x', 'y']
>>> print a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

Ayrıca ilgili öğelerin yerine boş liste atayarak öğeleri silebiliriz:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = []
>>> print a_list
['a', 'd', 'e', 'f']
```

Ve boş bir dilimin arasına yeni öğeler sıkıştırarak listede istediğimiz konuma yeni öğeler ekleyebiliriz:

```
>>> a_list = ['a', 'd', 'f']
>>> a_list[1:1] = ['b', 'c']
>>> print a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ['e']
>>> print a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

9.9 Liste silme

Dilimleri kullanarak liste öğelerini silmek uygunsuz ve hataya yatkın olacaktır. Python daha okunaklı bir alternatif yöntem sunmaktadır.

del listeden öğe silmekte kullanılır:

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

Tahmin edebileceğiniz gibi del negatif indisleri destekler ve varolmayan bir indis kullanıldığında çalışma zamanı hatası üretir.

Dilim işlemini del için bir indis olarak kullanabilirsiniz:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del a_list[1:5]
>>> print a_list
['a', 'f']
```

Her zamanki gibi, dilimler ikinci indise kadar tüm öğeleri seçer, ama ikinci indisteki öğeyi seçmez.

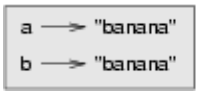
9.10 Nesneler ve değerler

Eğer aşağıdaki atama cümlelerini çalıştırırsak,

```
a = "banana"
b = "banana"
```

a ve b'nin "banana" karakterlerini içeren karakter dizilerini gösterdiğini biliriz. Ancak aynı karakter dizisine gösterip göstermediklerini bilemeyiz.

İki olası durum vardır:



Bir durumda, a ve b farklı şeyleri gösterir ama değerleri aynıdır. İkinci durumda aynı şeyi gösterirler. Bu şeylerin bir ismi vardır --- **nesneler** olarak adlandırılmaktadır. Ve nesne bir değişken tarafından gösterilebilen (referans edilen) bir şeydir.

Her nesne bir tekil **tanımlayıcı (identifier)**ya sahiptir, bu tanımlayıcıyı id fonksiyonuyla öğrenebiliriz. a ve b'nin tanımlayıcılarını ekranda görüntüleyerek aynı nesneyi gösterip göstermektediklerini anlayabiliriz.

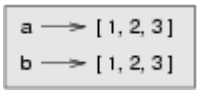
```
>>> id(a)
135044008
>>> id(b)
135044008
```

Gerçekte, aynı tanımlayıcıyı iki kere görüntülemiş olduk, bunun anlamı Python a ve b'nin her ikisinin göstermesi için tek bir karakter dizisi yaramıştır. Sizin id değeriniz muhtemelen farklı olacaktır.

İlginç bir şekilde, listeler farklı şekilde davranmaktadır. İki liste yarattığımızda, iki nesne elde ederiz:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Diyagram şu şekilde olacaktır:



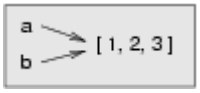
a ve b aynı değere sahiptir, fakat aynı nesneyi göstermemektedir.

9.11 Takma isimler

Değişkenler nesneleri gösterdiğine göre, eğer bir değişkeni bir başkasına atarsak, her iki değişken de aynı nesneyi gösterecektir:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a) == id(b)
True
```

Bu durumda, durum diyagramı şu şekilde olacaktır:



Aynı liste farklı isimlere sahip olduğu için, a ve b, **takma isimlendirilmiş (aliased)** olduğunu söyleriz. Bir takma isime yapılan değişiklikler diğerini de etkiler:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Bu davranış yararlı olsa da, bazı durumlarda beklenmeyen ve istenmeyen bir davranıştır. Genel olarak, değiştirilebilir nesnelerle çalışırken takma isimlerden uzak durmak daha güvenli olacaktır. Elbette, değiştirilemeyen nesneler için, herhangi bir problem yoktur. Bu Python'un karakter dizilerini ekonomi için takma isimlerle (farklı değişkenler için aynı karakter dizilerini kullanmasının) isimlendirmesinin bir nedenidir.

9.12 Listeleri klonlama

Eğer bir listeyi değiştirmek ama asıl kopyayı da korumak istiyorsak, listenin sadece referansının değil kendisinin bir kopyasını oluşturabilmemiz gerekmektedir. Bu sürece bazen kopyalama ile oluşan karışıklığı önlemek için **klonlama** adı verilmektedir.

Bir listeyi klonlamanın en kolay yolu dilim işlecini kullanmaktır:

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

a'nın herhangi bir dilimini almak yeni bir liste yaratır. Yukarıdaki örnekteki durumda dilimde listenin hepsi yer almaktadır.

Şimdi a için endişelenmeden b'ye istediğimiz değişiklikleri yapmakta özgürüz:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

9.13 Listeler ve for döngüleri

for döngüsü listeler için de çalışmaktadır. for döngüsünün genel söz dizimi şu şekildedir:

```
for DEĞİŞKEN in LİSTE:
    GÖVDE
```

Bu cümle aşağıdakiyle aynı işlevi görür:

```
i = 0
while i < len(LİSTE):
    DEĞİŞKEN = LİSTE[i]
    GÖVDE
    i += 1
```

for döngüsü daha sadedir çünkü döngü değişkeni, i'den bizi kurtarmaktadır. Daha önceki döngünün for döngüsüyle yazılmış halini aşağıda inceleyebilirsiniz.

```
for horseman in horsemen:
    print horseman
```

Neredeyse İngilizce dilindeki gibi okunacaktır: For (every) horseman in (the list of) horsemen, print (the name of the) horseman. (atlılar listesindeki her bir atlı için atlının adını yazdır)

for döngüsünde herhangi bir liste ifadesi kullanılabilir:

```
for number in range(20):
   if number % 3 == 0:
      print  number

for fruit in ["banana", "apple", "quince"]:
   print "I like to eat " + fruit + "s!"
```

İlk örnek 0 ile 19 arasındaki 3'ün katlarını yazdırır. İkinci örnek değişik meyveler için ilgiyi ekranda görüntüler.

Listeler değiştirilebilir olduğu için, bir listeyi dolaşarak her bir öğesini değiştirmek istenen bir özelliktir. Aşağıdaki örnek 1'den 5'e tüm sayıların karesini almaktadır:

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

for index in range(len(numbers)):
   numbers[index] = numbers[index]**2
```

Bir süre range(len(numbers)) hakkında düşünüp nasıl çalıştığını anlamaya çabalayın. Burada listedeki hem değer hem de indise ihtiyaç duyuyoruz, bu yüzden kendisine yeni bir değer atayabiliriz.

Aşağıdaki kalıp bunu gerçekleştirmek için Python'un sağladığı ve sıklıkla kullanılan güzel bir yoldur:

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

for index, value in enumerate(numbers):
   numbers[index] = value**2
```

enumerate listeyi dolaşırken hem indisi hem de onunla ilişkili değeri üretir. Aşağıdaki örneği çalıştırarak enumerate'in nasıl çalıştığını daha açık bir şekilde anlamaya çalışın:

```
>>> for index, value in enumerate(['banana', 'apple', 'pear',
'quince']):
...    print index, value
...
0 banana
1 apple
2 pear
3 quince
>>>
```

9.14 Liste parametreler

Bir listeyi argüman olarak geçirmek gerçekte listeye bir referans geçirir, listenin bir kopyasını değil. Listeler değiştirilebilir olduğu için parametreye yapılan bir

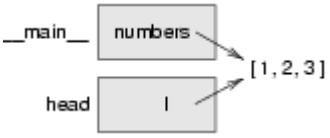
değişiklik asıl listeyi, argümanı da değiştirecektir. Örneğin, aşağıdaki fonksiyon bir listeyi argüman olarak alır ve her öğesini 2 ile çarpar:

```
def double_stuff(a_list):
    for index, value in enumerate(a_list):
        a_list[index] = 2 * value
```

Eğer double_stuff fonksiyonunu ch09.py isminde bir dosyaya koyarsak, aşağıdaki şekilde çalışmasını sınayabiliriz:

```
>>> from ch09 import double_stuff
>>> things = [2, 5, 'Spam', 9.5]
>>> double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

a_list parametresi ve things değişkeni aynı nesnenin takma isimleridir. Durum diyagramı şu şekilde olacaktır:



Liste nesnesi iki çerçeveyle paylaşıldığı için, aralarında çizdik.

Eğer bir fonksiyon liste parametreyi değiştirirse, çağıran kısım da değişikliği görür.

9.15 Saf (pure) fonksiyonlar ve değiştiriciler (modifiers)

Listeleri argüman olarak alıp çalışma esnasında onları değiştiren fonksiyonlara **değiştiriciler** adı verilir ve oluşan değişiklikler **yan etkiler (side effects)** olarak isimlendirilir.

Saf fonksiyon yan etkiler üretmez. Çağıran programla sadece parametrelerle haberleşir, parametreleri değiştirmez, bir dönüş değeri döndürür. Aşağıdaki double_stufffonksiyonunun saf fonksiyon şeklinde yazılmış halini görebilirsiniz:

```
def double_stuff(a_list):
    new_list = []
    for value in a_list:
        new_list += [2 * value]
    return new_list
```

double_stuff fonksiyonunun bu sürümü argümanları değiştirmez:

```
>>> from ch09 import double_stuff
>>> things = [2, 5, 'Spam', 9.5]
>>> double_stuff(things)
```

```
[4, 10, 'SpamSpam', 19.0]
>>> things
[2, 5, 'Spam', 9.5]
>>>
```

things'i değiştirmek için double_stuff saf fonksiyonunu kullanabilmek için, geri dönüş değerini tekrar things'e atamanız gerekir:

```
>>> things = double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

9.16 Hangisi daha iyi?

Değiştiricilerle yapabildiğiniz herşeyi saf fonksiyonlarla da yapabilirsiniz. Gerçekte bazı programlama dilleri sadece saf fonksiyonlara izin vermektedir. Saf fonksiyonları kullanan programların değiştiricileri kullanan programlara göre daha hızlı geliştirildiği ve daha az hataya yatkın olduklarına dair bazı kanıtlar vardır. Yine de değiştiricilerin uygun olduğu bazı durumlar vardır, ve bazı durumlarda fonksiyonel programlar daha az verimlidir.

Genel olarak, mantıklı olduğu her durumda saf fonksiyonları kullanmanızı, çetin bir avantaj olduğunda değiştiricilere başvurmanızı öneriyoruz. Bu yaklaşım fonksiyonel programlama tarzı olarak adlandırılabilir.

9.17 İçiçe listeler

İç liste bir başka listenin öğesi olarak gözüken listedir. Aşağıdaki listede, 3 indisine sahip öğe bir iç listedir:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

Eğer nested[3] ifadesini ekrana yazdırırsak, [10, 20] sonucunu elde ederiz. İçteki listeden bir öğe elde etmek için iki adım izlememiz gerekir:

```
>>> elem = nested[3]
>>> elem[0]
10
```

Veya bu adımları birleştiririz:

```
>>> nested[3][1]
20
```

Köşeli parantez işleçleri soldan sağa doğru işlenir, bu ifade içteki nested listesinin üçüncü öğesinin ilk öğesini döndürür.

9.18 Matrisler

İçiçe listeler genellikle matrisleri temsil etmek için kullanılır. Örneğin, şu matris:

şu şekilde temsil edilebilir:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

matrix üç öğesi olan, ve her bir öğesi matrisin bir satırı olan bir listedir. Köşeli parantez işleciyle matrisin bir satırını seçebiliriz:

```
>>> matrix[1]
[4, 5, 6]
```

Veya matrisin tek bir öğesini çift indis biçimini kullanarak seçebiliriz:

```
>>> matrix[1][1]
5
```

İlk indis saturu seçer, ikinci indis sütunu seçer. Bu şekilde matris temsili sık kullanılan bir yöntem olsa da, tek yöntem değildir. Basit bir değişiklik satır listesi yerine sütun listesi kullanımıdır. Daha sonra sözlük aracılığıyla daha radikal bir yöntemi göreceğiz.

9.19 Test güdümlü geliştirme (TDD)

Test güdümlü geliştirme (TDD) bir yazılım geliştirme pratiğidir. Bu pratikte küçük peşpeşe adımlar kullanılır. Bu adımlar otomatikleştirilmiş testlerle güdülenmiştir. İstenen özelliği veya ilerlemeyi sınayan testler önce yazılır.

Doctest TDD kullanımını göstermemizi oldukça kolaylaştırmaktadır. Verilen rows ve columns argümanlarından yararlanarak satır ve sütun matrisi yaratan bir fonksiyona ihtiyaç duyduğumuzu varsayalım.

İlk önce ismi matrices.py olan bir dosyada bu fonksiyon için bir test hazırlıyoruz:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Bu programı çalıştırdığımız çöken bir testle karşılaşırız:

```
**********************************************************************
*
File "matrices.py", line 3, in __main__.make_matrix
Failed example:
    make_matrix(3, 5)
Expected:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
Got nothing
**********************************************************************
*
1 items had failures:
   1 of   1 in __main__.make_matrix
***Test Failed*** 1 failures.
```

Test çöker çünkü henüz fonksiyonun gövdesi sadece üç tırnakla sınırlanmış bir karakter dizisi içerir, bir geri dönüş cümlesi yoktur. Bu yüzden None döndürür. Testimizde sıfırlardan oluşan 3 satır ve 5 sütun içeren bir matris döndürmesini bekliyoruz.

TDD'de testi geçecek çözümü yazarken kural çalışan en basit şeyi kullanmaktır, bu durumda beklenen sonucu döndüren bir şey yazabiliriz:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """
    return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Programı çalıştırdığımızda testleri geçer, ama bu gerçekleştirimimiz make_matrix fonksiyonunun hep aynı değeri döndürmesini, yani istediğimiz şeyi yapmamasını sağlamaktadır. Bunu düzeltmek için, önce iyileştirmemizi bir test ekleyerek güdülüyoruz:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

beklediğimiz gibi test çöker:

```
**********************************************************************
*
File "matrices.py", line 5, in __main__.make_matrix
Failed example:
```

```
    make_matrix(4, 2)
Expected:
    [[0, 0], [0, 0], [0, 0], [0, 0]]
Got:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
**********************************************************************
*
1 items had failures:
   1 of   2 in __main__.make_matrix
***Test Failed*** 1 failures.
```

Bu tekniğe test güdümlü adı verilmektedir çünkü sadece çöken test varken, bu testlerin başarılı olması için kod yazılması gerektiğini söyler. Çöken testle güdülenmiş bir şekilde, daha genel bir çözüm üretebiliriz:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0] * columns] * rows
```

Bu çözüm çalışıyor gibi gözüküyor, işimizin bittiğini düşünebiliriz, ancak yeni fonksiyonu daha sonra denediğimizde bir hata keşfederiz:

```
>>> from matrices import *
>>> m = make_matrix(4, 3)
>>> m
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> m[1][2] = 7
>>> m
[[0, 0, 7], [0, 0, 7], [0, 0, 7], [0, 0, 7]]
>>>
```

İkinci satır ve üçüncü sütundaki öğeye 7 değeriniz atamak istedik, ama tüm üçüncü sütundaki öğelerin 7 olduğunu görüyoruz!

İncelediğimizde, çözümümüzün her bir satırın diğer satırların bir takma ismi olduğunu farkederiz. Bu gerçekte yapmak istediğimiz bir şey değil, bu yüzden problemi çözmek için çökecek olan testi yazarız:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
```

```
    >>> m = make_matrix(4, 2)
    >>> m[1][1] = 7
    >>> m
    [[0, 0], [0, 7], [0, 0], [0, 0]]
  """
  return [[0] * columns] * rows
```

Çözülecek çöken testle birlikte, daha iyi bir çözüm için güdülenmiş oluruz:

```
def make_matrix(rows, columns):
  """
    >>> make_matrix(3, 5)
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    >>> make_matrix(4, 2)
    [[0, 0], [0, 0], [0, 0], [0, 0]]
    >>> m = make_matrix(4, 2)
    >>> m[1][1] = 7
    >>> m
    [[0, 0], [0, 7], [0, 0], [0, 0]]
  """
  matrix = []
  for row in range(rows):
     matrix += [[0] * columns]
  return matrix
```

TDD kullanımının bizim yazılım geliştirme sürecine bir çok yararı vardır. TDD:

- çözmeye çalıştığımız problemi çözmeden önce problem hakkında somut bir şekilde düşünmemize yardımcı olur.
- karmaşık problemleri daha küçük, basit problemlere parçalamamız ve büyük problemi çözmeyi adım adım gerçekleştirmemiz için bizi cesaretlendirir
- yazılımımızın iyi geliştirilmiş bir otomatik test kümesine sahip olmasını sağlar, böylece daha sonraki eklemeler ve geliştirmeler daha güvenli bir şekilde yapılır.

9.20 Karakter dizileri ve listeler

Python list adı verilen bir komuta sahiptir, bu komut bir dizi tipini alıp onun öğelerinden liste oluşturur:

```
>>> list("Crunchy Frog")
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

Ayrıca bir str komutu vardır, herhangi bir Python değerini argüman olarak alıp o değerin karakter dizisi temsilini döndürür.

```
>>> str(5)
'5'
>>> str(None)
```

```
'None'
>>> str(list("nope"))
"['n', 'o', 'p', 'e']"
```

Son örnekten gördüğümüz üzere, str karakterlerden oluşan bir listeyi birleştirmek için kullanılamamaktadır. Bunu yapabilmek için string modülündeki join fonksiyonunu kullanmamız gerekir:

```
>>> import string
>>> char_list = list("Frog")
>>> char_list
['F', 'r', 'o', 'g']
>>> string.join(char_list, '')
'Frog'
```

string modülünün en kullanışlı iki fonksiyonu karakter dizilerinin listesiyle ilgilidir. split fonksiyonu bir karakter dizisini kelimelerden oluşan bir listeye ayrıştırır. Varsayılan olarak beyaz boşluk karakterleri kelime sınırları olarak kabul edilir:

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

İsteğe bağlı bir argüman olan **ayırıcı (delimeter)** kelime sınırlarını belirleyecek karakterleri ifade etmek için kullanılabilir. Aşağıdaki örnekler ai karakter dizisini ayırıcı olarak kullanmaktadır:

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...']
```

Ayırıcının listede yer almadığına dikkat edin.

string.join fonksiyonu string.split'in karşıtıdır. İki argüman alır: karakter dizisi listesi ve sonuç karakter dizisinde görünecek listedeki her bir öğe arasında koyulacak birayırıcı (separator).

```
>>> import string
>>> words = ['crunchy', 'raw', 'unboned', 'real', 'dead', 'frog']
>>> string.join(words, ' ')
'crunchy raw unboned real dead frog'
>>> string.join(words, '**')
'crunchy**raw**unboned**real**dead**frog'
```

9.21 Sözlük

liste:
Nesnelerin isimlendirilmiş bir kolleksiyonu, her bir öğe bir indis ile tanımlanır.
indis:

Bir listenin öğesini gösteren, temsil eden tamsayı değişken veya değer

öğe:
Listedeki (veya diğer dizilerdeki) değerlerden biri. Köşeli parantez işleci lsitedeği öğeleri seçer.

dizi:
Sıralı öğe kümesinden oluşan herhangi bir veri tipi, her bir öğe bir indis ile tanımlanır.

iç liste:
Bir başka listenin öğesi olan liste

adım sayısı:
Lineer bir dizideki peşpeşe öğeler arasındaki aralık miktarıdır. range fonksiyonunun üçüncü (veya isteğe bağlı) argümanına adım sayısı adı verilir. Eğer belirtilmezse varsayılan değeri 1'dir.

liste dolaşma:
Listedeki her bir öğeye sıralı bir şekilde erişmedir.

değiştirilebilir (mutable) tip:
Öğeleri değiştirilebilen veri tipidir. Tüm değiştirilebilir tipler bileşik tiplerdir. Listeler değiştirilebilir veri tipidir; karakter dizileri değildir.

nesne:
Değişkenlerin gösterdiği şey

takma isimler:
Aynı nesneyi gösteren farklı değişkenlerdir.

klonlama:
Var olan nesneyle aynı değere farklı yeni bir nesne yaratma. Nesneye olan referansı kopyalama işlemi takma isim oluştururken nesneyi klonlamaz.

değiştirici:
Fonksiyon gövdesi için argümanları değiştiren fonksiyonlardır. Sadece değiştirilebilir tipler değiştiricilerle değiştirilebilir.

yan etki:
Programın durumunda bir fonksiyonu çağırarak yapılan ancak fonksiyonun geri dönüş değerini okuyarak gerçekleşmeyen değişikliktir. Yan etkiler sadece değiştiricilerle üretilebilir.

saf fonksiyon:
Yan etkisi olmayan fonksiyondur. Saf fonksiyonlar sadece geri dönüş değerlerini döndürerek çağıran programa değişiklik yaparlar.

test güdümlü geliştirme (TDD):
Küçük, arttırımlı adımlar şeklinde istenen özelliğin geliştirildiği, her bir adımın önce yazılan otomatik testlerle güdülendiği yazılım geliştirme pratiğidir. Önce testler yazılarak istenen özellik ve geliştirmeler için güdü sağlanır.(daha ayrıntılı bilgi için Test-driven development wikipedia yazısını inceleyebilirsiniz.)

ayırıcı:
Karakter dizisinin nereden ayrılacağını belirten bir karakter veya karakter dizisi.

9.22 Alıştırmalar

1. Aşağıdaki listeyi dolaşan ve her öğenin boyutunu yazan bir döngü yazın:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

   len'e tamsayı gönderince ne oluyor? 1'i 'one''a çevirerek çözümünüzü tekrar çalıştırın.

2. ch09e02.py ismindeki bir dosyaya aşağıdaki içeriği yazın:

```
#  doctestleri buraya ekleyin:
"""
"""

# Python kodunuzu buraya ekleyin:


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

   Aşağıdaki her bir dostest kümesini dosyanın üstündeki docstringe koyup ve bu testlerin geçmesi için gereken Python kodlarını yazın.

```
1."""
    >>> a_list[3]
    42
    >>> a_list[6]
    'Ni!'
    >>> len(a_list)
    8
  """

2."""
    >>> b_list[1:]
    ['Stills', 'Nash']
    >>> group = b_list + c_list
    >>> group[-1]
    'Young'
  """

3."""
    >>> 'war' in mystery_list
    False
    >>> 'peace' in mystery_list
    True
    >>> 'justice' in mystery_list
    True
    >>> 'oppression' in mystery_list
    False
```

```
    >>> 'equality' in mystery_list
    True
  """

4."""
    >>> range(a, b, c)
    [5, 9, 13, 17]
  """
```

Her adımda bir doctest kümesini ekleyin. Önceki eklenen test başarılı olmadan, yeni doctestler eklenmemelidir.

3. Python yorumlayıcısının aşağıdakilere tepkisi nedir?

```
>>> range(10, 0, -2)
```

range fonksiyonunun üç argümanı sırasıyla başlangıç, bitiş ve adımdır. Bu örnekte başlangıç, bitiş'ten daha büyüktür. başlangıç < bitiş ve adım < 0 durumunda ne olur? başlangıç, bitiş ve adım arasındaki ilişki hakkında bir kural yazınız.

```
4.a = [1, 2, 3]
  b = a[:]
  b[0] = 5
```

a ve b için üçüncü satır işlenmeden ve işlendikten sonra durum diyagramı çiziniz.

5. Aşağıdaki programın çıktısı ne olur?

```
this = ['I', 'am', 'not', 'a', 'crook']
that = ['I', 'am', 'not', 'a', 'crook']
print "Test 1: %s" % (id(this) == id(that))
that = this
print "Test 2: %s" % (id(this) == id(that))
```

Sonucun ayrıntılı bir açıklamasını yapınız.

6. ch09e06.py isminde bir dosya açıp, 2. alıştırmadaki aynı prosedürü aşağıdaki doctestler başarılı olacak şekilde tekrarlayın:

```
1."""
    >>> 13 in junk
    True
    >>> del junk[4]
    >>> junk
    [3, 7, 9, 10, 17, 21, 24, 27]
    >>> del junk[a:b]
    >>> junk
    [3, 7, 27]
  """

2."""
```

```
    >>> nlist[2][1]
    0
    >>> nlist[0][2]
    17
    >>> nlist[1][1]
    5
  """

3."""
    >>> import string
    >>> string.split(message, '??')
    ['this', 'and', 'that']
  """
```

7. Aynı uzunlukta olan ve sayılardan oluşan iki liste argümanı alan ve her bir listede birbirine karşılık gelen sayıların toplamını yeni bir liste olarak döndürenadd_lists(a, b) fonksiyonunu yazınız.

```
def add_lists(a, b):
    """
      >>> add_lists([1, 1], [1, 1])
      [2, 2]
      >>> add_lists([1, 2], [1, 4])
      [2, 6]
      >>> add_lists([1, 2, 1], [1, 4, 3])
      [2, 6, 4]
    """
```

add_lists yukarıdaki doctestleri geçmelidir.

8. Aynı uzunlukta olan ve sayılardan oluşan iki liste argümanı alan ve her listede birbirine karşılık gelen sayılarının çarpımlarını yeni bir liste olarak döndürenmult_lists(a, b) fonksiyonunu yazınız.

```
def mult_lists(a, b):
    """
      >>> mult_lists([1, 1], [1, 1])
      2
      >>> mult_lists([1, 2], [1, 4])
      9
      >>> mult_lists([1, 2, 1], [1, 4, 3])
      12
    """
```

mult_lists yukarıdaki doctestleri geçmelidir.

9. Test güdümlü geliştirme bölümünde anlatılan matrices.py programına aşağıdaki iki fonksiyonu ekleyin:

```
def add_row(matrix):
    """
```

```
      >>> m = [[0, 0], [0, 0]]
      >>> add_row(m)
      [[0, 0], [0, 0], [0, 0]]
      >>> n = [[3, 2, 5], [1, 4, 7]]
      >>> add_row(n)
      [[3, 2, 5], [1, 4, 7], [0, 0, 0]]
      >>> n
      [[3, 2, 5], [1, 4, 7]]
    """

def add_column(matrix):
    """
      >>> m = [[0, 0], [0, 0]]
      >>> add_column(m)
      [[0, 0, 0], [0, 0, 0]]
      >>> n = [[3, 2], [5, 1], [4, 7]]
      >>> add_column(n)
      [[3, 2, 0], [5, 1, 0], [4, 7, 0]]
      >>> n
      [[3, 2], [5, 1], [4, 7]]
    """
```

Yeni fonksiyonlarınız doctestleri geçmelidir. Her fonksiyondaki son doctest add_row ve add_column fonksiyonlarının saf fonksiyon olduğunu garantilemek için yazılmıştır.

10. m1 ve m2'yi toplayan ve toplamlarını yeni bir matris olarak döndüren add_matrices(m1, m2) fonksiyonunu yazınız. m1 ve m2 aynı büyüklükte olduğunu kabul edebilirsiniz. Matrislerini birbirine karşılık gelen değerlerini toplayarak toplarsınız.

```
def add_matrices(m1, m2):
    """
      >>> a = [[1, 2], [3, 4]]
      >>> b = [[2, 2], [2, 2]]
      >>> add_matrices(a, b)
      [[3, 4], [5, 6]]
      >>> c = [[8, 2], [3, 4], [5, 7]]
      >>> d = [[3, 2], [9, 2], [10, 12]]
      >>> add_matrices(c, d)
      [[11, 4], [12, 6], [15, 19]]
      >>> c
      [[8, 2], [3, 4], [5, 7]]
      >>> d
      [[3, 2], [9, 2], [10, 12]]
    """
```

Yeni fonksiyonunuzu matrices.py programına ekleyin, ve yukarıdaki doctestleri geçmesini sağlayın. Son iki doctest add_matrices fonksiyonunun saf fonksiyon olmasını sağlamak içindir.

11. Bir matrisi,m, bir büyüklük, n ile çarpan scalar_mult(n, m) fonksiyonunu yazın.

```
def scalar_mult(n, m):
    """
      >>> a = [[1, 2], [3, 4]]
      >>> scalar_mult(3, a)
      [[3, 6], [9, 12]]
      >>> b = [[3, 5, 7], [1, 1, 1], [0, 2, 0], [2, 2, 3]]
      >>> scalar_mult(10, b)
      [[30, 50, 70], [10, 10, 10], [0, 20, 0], [20, 20, 30]]
      >>> b
      [[3, 5, 7], [1, 1, 1], [0, 2, 0], [2, 2, 3]]
    """
```

Fonksiyonunuzu matrices.py programına ekleyin ve yukarıdaki doctestleri geçmesini sağlayın.

12.
```
def row_times_column(m1, row, m2, column):
    """
      >>> row_times_column([[1, 2], [3, 4]], 0, [[5, 6], [7, 8]],
0)
      19
      >>> row_times_column([[1, 2], [3, 4]], 0, [[5, 6], [7, 8]],
1)
      22
      >>> row_times_column([[1, 2], [3, 4]], 1, [[5, 6], [7, 8]],
0)
      43
      >>> row_times_column([[1, 2], [3, 4]], 1, [[5, 6], [7, 8]],
1)
      50
    """

def matrix_mult(m1, m2):
    """
      >>> matrix_mult([[1, 2], [3,  4]], [[5, 6], [7, 8]])
      [[19, 22], [43, 50]]
      >>> matrix_mult([[1, 2, 3], [4,  5, 6]], [[7, 8], [9, 1],
[2, 3]])
      [[31, 19], [85, 55]]
      >>> matrix_mult([[7, 8], [9, 1], [2, 3]], [[1, 2, 3], [4,
5, 6]])
      [[39, 54, 69], [13, 23, 33], [14, 19, 24]]
    """
```

Yeni fonksiyonlarını matrices.py programına ekleyin ve yukarıdaki doctestleri geçmesini sağlayın.

```
13.import string

  song = "The rain in Spain..."
```

string.join(string.split(song)) ve song arasındaki ilişkiyi tanımlayın. Tüm karakter dizileri için aynı mıdır? Ne zaman farklıdırlar?

14. s karakter dizisindeki tüm old olan yerleri new ile değiştiren replace(s, old, new) fonksiyonunu yazınız.

```
  def replace(s, old, new):
      """
        >>> replace('Mississippi', 'i', 'I')
        'MIssIssIppI'
        >>> s = 'I love spom!  Spom is my favorite food.  Spom,
  spom, spom, yum!'
        >>> replace(s, 'om', 'am')
        'I love spam!  Spam is my favorite food.  Spam, spam, spam,
  yum!'
        >>> replace(s, 'o', 'a')
        'I lave spam!  Spam is my favarite faad.  Spam, spam, spam,
  yum!'
      """
```

Çözümünüz yukarıdaki doctestleri sağlamalıdır. İpucu: string.split ve string.join kullanın.

10. Modüller ve dosyalar

10.1 Modüller

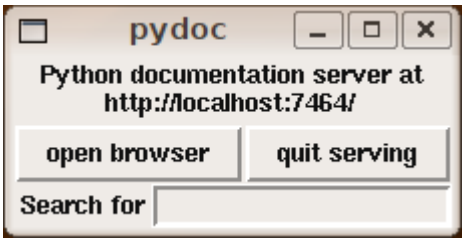
Modül, diğer Python programları tarafından kullanılmak üzere Python tanımlamaları ve cümlelerini içeren bir dosyadır. Python'la birlikte **standart kütüphane**nin parçası olarak gelen bir çok Python modülü vardır. Bunlardan iki tanesini daha önce gördük, doctest ve string modülleri.

10.2 pydoc

pydoc modülünü sistemde kurulu olan Python kütüphanelerini içerisinde arama yapmak için kullanabilirsiniz. **Komut satırı**nda aşağıdakini yazın:

```
$ pydoc -g
```

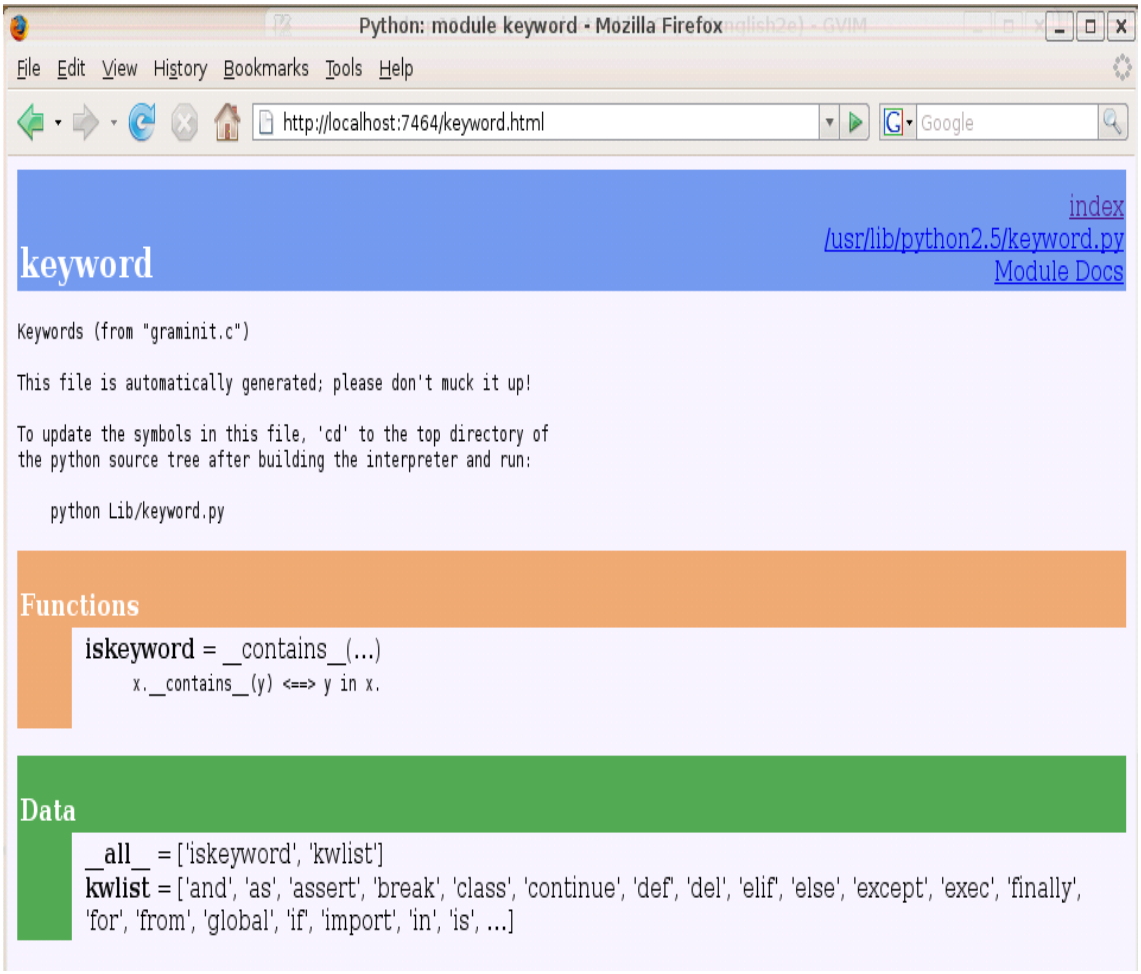
ve aşağıdaki görüntülenecektir:



(not: bir hata ile karşılaşırsanız 2. alıştırmaya bakın)

pydoc ile üretilmiş belgeleri ağ tarayıcı penceresinde görüntülemek için open browser düğmesine basın:

Bu sistemdeki Python tarafından bulunan bütün python kütüphanelerinin bir listesidir. Bir modül ismine tıkladığınızda, o modülün belgelemesi açılacaktır. Bir keyworde tıkladığınızda, örneğin, aşağıdaki pencere açılacaktır:



Çoğu modülün belgeleri üç renkli kod kısımları içermektedir:

- Sınıflar pembe
- Fonksiyonlar turuncu
- Veri yeşil

Sınıflar daha sonraki bölümlerde anlatılacaktır, ama şimdi pydocu modüller içerisindeki fonksiyonları ve verileri görmek için kullanabiliriz.

keyword modülü tek bir fonksiyon içermektedir, iskeyword, isminden de anlaşılacağı gibi parametre olarak aldığı karakter dizisi bir anahtar kelime ise True değerini döndüren boolean bir fonksiyondur:

```
>>> from keyword import *
>>> iskeyword('for')
True
>>> iskeyword('all')
False
>>>
```

Veri öğesi, kwlist Python'da yer alan tüm anahtar kelimelerin bir listesini içerir:

```
>>> from keyword import *
```

```
>>> print kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif',
'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if',
'import',
'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise',
'return', 'try',
'while', 'with', 'yield']
>>>
```

Sizi pydoc'u Python ile birlikte gelen kapsamlı kütüphaneleri araştırmanız için kullanmaya teşvik ediyoruz. Keşfedilecek bir çok hazine var!

10.3 Modül yaratma

Modül yaratmak için bütün yapılması gereken uzantısı .py olan bir metin dosyasıdır:

```
#  seqtools.py
#
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

Modülümüzü hem betiklerde hem de Python kabuğunda kullanabiliriz. Bunu yapabilmek için öncelikle modülümüzü içe aktarmak (import) gerekmektedir. Bunu yapmanın iki yolu vardır:

```
>>> from seqtools import remove_at
>>> s = "A string!"
>>> remove_at(4, s)
'A sting!'
```

ve:

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

İlk örnekte, remove_at daha önce gördüğümüz fonksiyonlar gibi çağrıldı. İkinci örnekte modülün ismi ve bir nokta (.) fonksiyon isminden önce yazıldı.

İki durumda da dosyayı içe aktarırkan .py uzantısını yazmadığımıza dikkat edin. Python, Python modüllerinin .py uzantısı ile bitmesini beklemektedir, bu nedenle dosya uzantısı **içe aktarma cümlesi** yer almaz.

Modül kullanımı çok büyük programları yönetilebilir büyüklükte parçalara bölmemize ve ilişkili parçaları birlikte tutmamıza yaramaktadır.

10.4 İsim uzayları

İsim uzayı sözdizimsel bir kaptır. Sayesinde aynı ismin farklı modül veya fonksiyonlarda (yakında göreceğimiz gibi sınıf ve metotlarda) kullanılmasına izin verir.

Her modül kendi isim uzayını belirler, böylece aynı ismi farklı modüllerde bir tanımlama problemiyle karşılaşmadan kullanabiliriz.

```
# module1.py

question = "What is the meaning of life, the Universe, and
everything?"
answer = 42

# module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

Şimdi her iki modülü de içe aktarıp içerilerindeki question ve answer erişebiliriz:

```
>>> import module1
>>> import module2
>>> print module1.question
What is the meaning of life, the Universe, and everything?
>>> print module2.question
What is your quest?
>>> print module1.answer
42
>>> print module2.answer
To seek the holy grail.
>>>
```

Eğer from module1 import * ve from module2 import * kullansaydık bir **isimlendirme çakışması**yla karşılaşabilirdik ve module1deki question ve answer'a erişemezdik.

Fonksiyonlar da kendi isim uzaylarına sahiptir:

```
def f():
    n = 7
    print "printing n inside of f: %d"  % n

def g():
    n = 42
    print "printing n inside of g: %d"  % n

n = 11
print "printing n before calling f: %d"  % n
```

```
f()
print "printing n after calling f: %d"  % n
g()
print "printing n after calling g: %d"  % n
```

Bu programı çalıştırmak aşağıdaki çıktıyı üretir:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

Üç n burada çakışmaz çünkü her biri ayrı isim uzayındadır.

İsim uzayları birden fazla programcının aynı projede isim çakışmalarıyla karşılaşmadan birlikte çalışmasına olanak sağlar.

10.5 Özellikler ve nokta işleci

Modül içerisinde tanımlanmış olan değişkenlere modülün **özellikler**i denir. Bu özelliklere **nokta işleci** (.) ile erişilir. module1 ve module2'nin question özelliklerinemodule1.question ve module2.question şeklinde erişilmektedir.

Modüller özellikler gibi, fonksiyonlar da içermektedir ve nokta işleci aynı şekilde fonksiyonlara erişmek için de kullanılmaktadır. seqtools.remove_at ifadesi seqtoolsmodülünün remove_at fonksiyonunu temsil etmektedir.

Bölüm 7'de string modülünün find fonksiyonunu tanıtmıştık. string modülü daha başka bir çok yararlı fonksiyon içermektedir:

```
>>> import string
>>> string.capitalize('maryland')
'Maryland'
>>> string.capwords("what's all this, then, amen?")
"What's All This, Then, Amen?"
>>> string.center('How to Center Text Using Python', 70)
'                   How to Center Text Using Python
'
>>> string.upper('angola')
'ANGOLA'
>>>
```

string modülündeki diğer fonksiyon ve özellikleri öğrenmek için pydoc'u kullanmalısınız.

10.6 Karakter dizisi ve liste metotları

Python dili geliştikçe, string modülündeki fonksiyonların bir çoğu ayrıca karakter dizisi nesnesi **metotları** olarak eklendi. Metot bir fonksiyona oldukça benzer, ama çağırma sözdizimi biraz farklıdır:

```
>>> 'maryland'.capitalize()
'Maryland'
>>> "what's all this, then, amen?".title()
"What'S All This, Then, Amen?"
>>> 'How to Center Text Using Python'.center(70)
'                   How to Center Text Using Python
'
>>> 'angola'.upper()
'ANGOLA'
>>>
```

Karakter dizisi metotları string nesneleri içerisinde yer alırlar, ve nesneden sonra nokta işleci, sonrasında metot ismiyle çağrılırlar (invoke).

Kendi nesnelerimizi, kendi metotlarımızla nasıl oluşturacağımızı ileriki bölümlerde öğreneceğiz. Şimdilik Python'un yerleşik nesneleriyle gelen metotları nasıl kullanacağımızı göreceğiz.

Nokta işleci liste nesnelerinin yerleşik metotlarına erişmek için de kullanılmaktadır:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
>>>
```

append verilen parametreyi listenin sonuna ekleyen bir liste metotudur. Bu örnekle devam edecek olursak, bir çok diğer liste metotu şunlardır:

```
>>> mylist.insert(1, 12)
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)
2
>>> mylist.extend([5, 9, 5, 11])
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)
6
>>> mylist.count(5)
```

```
3
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
>>>
```

Bu örnekteki liste metotlarıyla denemeler yapmaya, nasıl çalıştıklarını gerçekten anladığınıza güvenene kadar devam edin.

10.7 Metin dosyalarını okuma ve yazma

Bir program çalışırken, verisi rastgele erişimli bellekte (RAM) saklanmaktadır, ancak bu bellek **geçici** (volatile)dir, bunun anlamı programın çalışması sonlandığında, veya bilgisayar kapandığında, RAM'deki veri kaybolur. Bilgisayarı açtığınızda ve programınızı çalıştırdığınızda bu veriye tekrar ulaşmak istiyorsanız, veriyi **kalıcı** (non volatile) bir depo ortamına - sabit disk, usb disk, yazılabilir CD, vb. - yazmanız gerekir.

Kalıcı depo ortamındaki veri **dosyalar** adı verilen isimlendirilmiş konumlarda saklanmaktadır. Dosyaları okuyarak ve yazarak, programlar çalıştırmalar arasında bilgi saklayabilir.

Dosyalarla çalışmak not defterleriyle çalışmaya benzer. Bir not defteri kullanabilmek için öncelikle açmanız gerekir. İşiniz bittikten sonra da kapatmanız gerekir. Not defteri açıkken yazabilir veya okuyabilirsiniz. Her iki durumda da not defterinde nerede kaldığınızı bilirsiniz. Not defterinin hepsini doğal sırasında okuyabilirsiniz veya bazı kısımları atlayabilirsiniz.

Bütün bunlar dosyalar için de geçerlidir. Bir dosyayı açmak için ismini belirtir ve hangi amaçla - okuma veya yazma - açtığınızı söylersiniz.

Bir dosya açmak bir file nesnesi yaratır. Aşağıdaki örnekte myfile yeni dosya nesnesini göstermektedir.

```
>>> myfile = open('test.dat', 'w')
>>> print myfile
<open file 'test.dat', mode 'w' at 0x2aaaaab80cd8>
```

open fonksiyonu iki argüman almaktadır. İlk argüman dosyanın ismidir, ikinci argümanda **mod**tur. 'w' modunun anlamı dosyayı yazmak için açtığımızdır.

Eğer test.dat isminde bir dosya yoksa, yaratılacaktır. Eğer hali hazırda bir tane varsa, yeni yazacağımız dosyayla değiştirilerecektir.

Dosya nesnesini yazarken, dosyanın ismini, modunu, ve nesnenin konumunu görürüz.

Dosyaya veri koymak için dosya nesnesindeki write metotunu çalıştırırız:

```
>>> myfile.write("Now is the time")
>>> myfile.write("to close the file")
```

Dosyayı kapatmak sisteme yazma işimizin bittiğini söyler, böylece dosyamız okumak için elverişli hale gelir:

```
>>> myfile.close()
```

Şimdi dosyayı tekrar açabiliriz, ancak bu sefer okumak için, ve içeriği bir karakter dizisine okuruz. Okumak için mod argümanı 'r'dir:

```
>>> myfile = open('test.dat', 'r')
```

Eğer olmayan bir dosyayı açmaya çalışırsak, bir hatayla karşılaşırız:

```
>>> myfile = open('test.cat', 'r')
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Tahmin edebileceğiniz gibi, read metotu dosyadaki tüm veriyi okumaktadır. Argümansız işletilirse, dosyanın tüm içeriğini tek bir karakter dizisine okuyacaktır:

```
>>> text = myfile.read()
>>> print text
Now is the timeto close the file
```

Gördüğünüz üzere time ve to arasında boşluk yok, çünkü dosyaya yazarken karakter dizileri arasına boşluk koymadık.

read ayrıca okunacak karakter sayısını belirleyen bir argüman da alabilir:

```
>>> myfile = open('test.dat', 'r')
>>> print myfile.read(5)
Now i
```

Eğer dosyada yeterince karakter kalmazsa, read kalan karakterleri döndürecektir. Dosyanın sonuna geldiğimizde read boş bir karakter dizisi okuyacaktır:

```
>>> print myfile.read(1000006)
s the timeto close the file
>>> print myfile.read()

>>>
```

Aşağıdaki fonksiyon bir dosyayı kopyalar, tek bir seferde elli karakter okuyup yazarak kopyalamayı gerçekleştirir. İlk argüman kaynak olan asıl dosyamızın, ikinci argüman da yeni yaratılacak olan hedef dosyamızın ismidir.

```
def copy_file(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.read(50)
        if text == "":
            break
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

Bu fonksiyon infile'dan 50 karakter okur ve outfile'a 50 karakter yazar, döngüye infile'ın sonuna erişene kadar devam eder, sonuna eriştiğimizde text boştur ve breakcümlesi işletilir.

10.8 Metin dosyaları

Metin dosyası yazdırılabilir karakterler ve beyaz boşluklar içeren, satırlar şeklinde düzenlenmiş, satırlar birbirinden yeni satır karakterleriyle ayrılmış dosyadır. Python özellikle metin dosyalarını işlemek için tasarlandığı için, bu işi kolaylaştırmak için metotlar sunar.

Gösterim için, yeni satır karakteriyle ayrılmış üç satır metinden oluşan bir metin dosyası yaratacağız:

```
>>> outfile = open("test.dat","w")
>>> outfile.write("line one\nline two\nline three\n")
>>> outfile.close()
```

readline metotu bütün karakterleri, yeni satır karakteri de dahil olmak üzere, okur:

```
>>> infile = open("test.dat","r")
>>> print infile.readline()
line one

>>>
```

readlines kalan bütün satırları karakter dizisi listesi olarak döndürür:

```
>>> print infile.readlines()
['line two\012', 'line three\012']
```

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence \\012.

At the end of the file, readline returns the empty string and readlines returns the empty list:

```
>>> print infile.readline()

>>> print infile.readlines()
[]
```

Aşağıdaki satır-işleme programı örneğidir. filter metotu oldfile'ın # ile başlayan satırları olmayacak şekilde yeni bir kopyasını yaratır:

```
def filter(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.readline()
        if text == "":
            break
        if text[0] == '#':
            continue
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

continue cümlesi döngünün o anki adımını bitirir ama döngüye devam eder. İşleme akışı döngünün başına gider, koşulu kontrol eder ve uygun şekilde süreç devam eder.

Böylece, eğer text boşsa döngüden çıkılır. Ama eğer text'in ilk karakteri diyez işareti ise, işlem akışı döngünün başına gider. Her iki koşul da geçersiz olursa text'i yeni dosyaya yazarız.

10.9 Dizinler

Kalıcı depo ortamındaki dosyalar **dosya sistemi** adı verilen kural kümeleriyle düzenlenmiştir. Dosya sistemleri dosyalardan ve **dizinlerden** oluşmaktadır, dizinler hem dosyalar hem de diğer dizinler için bir kap görevi görürler.

Bir dosyayı açıp ve yazarak yeni bir dosya yarattığınızda, yeni dosya o an ki dizinde (programı çalıştırdığınız dizin) yer alır. Benzer olarak bir dosyayı okumak için açtığınızda, Python o an ki dizinde dosyayı arar.

Eğer başka bir yerdeki bir dosyayı açmak istiyorsanız, o dosyanın **konumunu** (path), hangi dizinde bulunduğunu belirtmeniz gerekmektedir:

```
>>> wordsfile = open('/usr/share/dict/words', 'r')
>>> wordlist = wordsfile.readlines()
>>> print wordlist[:5]
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

Bu örnek, / adı verilen sistemin en üst dizini altındaki usr dizini altındaki share dizini altındaki dict dizini içerisindeki words isminde bir dosyayı açmaktadır. Daha sonra tüm satırları karakter dizisi listesine readlines ile okumakta ve listenin ilk 5 öğesini ekranda görüntülemektedir.

/ karakterini dosya isminin bir parçası olarak kullanamazsınız; çünkü dizin ve dosya isimleri arasında **ayırıcı** (delimiter) karakter olarak rezerve edilmiştir.

/usr/share/dict/words dosyası Unix tabanlı sistemlerde var olmalıdır ve alfabetik sıralı kelime listesi içerir.

10.10 Counting Letters

ord fonksiyonu bir karakterin tamsayı temsilini döndürmektedir:

```
>>> ord('a')
97
>>> ord('A')
65
>>>
```

Bu örnek neden 'Apple' < 'apple' ifadesi True döndürür açıklamaktadır.

chr fonksiyonu ord fonksiyonunun tersidir. Bir tamsayı argüman alır ve bu argümanın karakter temsilini döndürür:

```
>>> for i in range(65, 71):
...     print chr(i)
...
A
B
C
D
E
F
>>>
```

Aşağıdaki countletters.py programı Alice in Wonderland kitabındaki her bir karakterin kaç kere geçtiğini sayar:

```
#
# countletters.py
#

def display(i):
    if i == 10: return 'LF'
    if i == 13: return 'CR'
    if i == 32: return 'SPACE'
    return chr(i)
```

```
infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()

counts = 128 * [0]

for letter in text:
    counts[ord(letter)] += 1

outfile = open('alice_counts.dat', 'w')
outfile.write("%-12s%s\n" % ("Character", "Count"))
outfile.write("================\n")

for i in range(len(counts)):
    if counts[i]:
        outfile.write("%-12s%d\n" % (display(i), counts[i]))

outfile.close()
```

Bu programı çalıştırıp ürettiği çıktı dosyasını bir metin düzenleyici ile inceleyin. Sondaki alıştırmalarda bu programı çözümlemeniz istenecek.

10.11 sys modülü ve argv

sys modülü python yorumlayıcının çalıştığı ortama erişmek için fonksiyonlar ve değişkenler içermektedir.

Aşağıda örnek bizim sistemdeki bazı değişkenlerin değerini görüntülemektedir:

```
>>> import sys
>>> sys.platform
'linux2'
>>> sys.path
['', '/home/jelkner/lib/python', '/usr/lib/python25.zip',
'/usr/lib/python2.5',
'/usr/lib/python2.5/plat-linux2', '/usr/lib/python2.5/lib-tk',
'/usr/lib/python2.5/lib-dynload', '/usr/local/lib/python2.5/site-
packages',
'/usr/lib/python2.5/site-packages', '/usr/lib/python2.5/site-
packages/Numeric',
'/usr/lib/python2.5/site-packages/gst-0.10',
'/var/lib/python-support/python2.5', '/usr/lib/python2.5/site-
packages/gtk-2.0',
'/var/lib/python-support/python2.5/gtk-2.0']
>>> sys.version
'2.5.1 (r251:54863, Mar  7 2008, 04:10:12) \n[GCC 4.1.3 20070929
(prerelease)
(Ubuntu 4.1.2-16ubuntu2)]'
>>>
```

Aynı makinede **Jython** başlatmak aynı değişkenler için farklı değerleri üretir:

```
>>> import sys
>>> sys.platform
'java1.6.0_03'
>>> sys.path
['', '/home/jelkner/.', '/usr/share/jython/Lib',
'/usr/share/jython/Lib-cpython']
>>> sys.version
'2.1'
>>>
```

Sizin makinenizdeki sonuçlar da farklı olacaktır elbette.

argv değişkeni Python betiği çalıştırıldığında **komut satırı**ndan okunan karakter dizilerinin listesini tutmaktadır. Bu **komut satırı argümanları** program başlarken programa bilgi geçirmeye yardımcı olur.

```
#
# demo_argv.py
#
import sys

print sys.argv
```

Bu programı unix komut satırından çalıştırmanız sys.argv'nin nasıl çalıştığını gösterecektir:

```
$ python demo_argv.py this and that 1 2 3
['demo_argv.py', 'this', 'and', 'that', '1', '2', '3']
$
```

argv karakter dizisi listesidir. İlk öğenin programın ismi olduğuna dikkat edin. Argümanlar beyaz boşluklarla liste şeklinde ayrılmıştır, aynı string.split fonksiyonun işlemesi gibi. Eğer bir argümanın beyaz boşluk içermesini istiyorsanız, o argümanın etrafına tırnak işaretleri koyun:

```
$ python demo_argv.py "this and" that "1 2" 3
['demo_argv.py', 'this and', 'that', '1 2', '3']
$
```

argv ile girdilerini doğrudan komut satırından alan yararlı programlar yazabiliriz. Örneğin aşağıdaki bir sayı dizisinin toplamını alan bir programdır:

```
#
# sum.py
#
from sys import argv

nums = argv[1:]
```

```
for index, value in enumerate(nums):
    nums[index] = float(value)

print sum(nums)
```

Bu programda from <module> import <attribute> şeklinde içe aktarmayı kullanmaktayız, böylece argv modülün isim uzayına alınmış oluyor.

Şimdi programı komut satırından aşağıdaki şekilde çalıştırabiliriz:

```
$ python sum.py 3 4 5 11
23
$ python sum.py 3.5 5 11 100
119.5
```

Alıştırma olarak benzer programlar yazmanız istenmektedir.

10.12 Sözlük

modül:
Başka Python programlarında kullanılmak üzere Python tanımlama ve cümleleri içeren bir dosyadır. Modülün içerikleri başka programlarda kullanılmak üzere importcümlesiyle içe aktarılmaktadır.

standart kütüphane:
Kütüphane başka yazılımların geliştirilmesinde kullanılmak üzere araçlar içeren yazılım kolleksiyonudur. Bir programlama dilinin standart kütüphanesi bu tür araçların çekirdek programlama diliyle dağıtılanlarıdır. Python geniş bir standart kütüphaneyle birlikte gelir.

pydoc:
Python standart kütüphanesiyle birlikte gelen bir belge üreticidir.

komut istemi:
Komut satırı arayüzü tarafından görüntülen bir karakter dizisidir, komutların girilebileceğini belirtir.

import cümlesi:
Bir modül içerisindeki nesnelerin kullanıma uygun hale gelmesini sağlayan cümledir. İki biçimi vardır. mymod modül ismi, içerdiği fonksiyonlar f1 ve f2, içerdiği değişkenler v1 ve v2 olsun, iki biçimin örnekleri şunlardır:

```
import mymod
```

İlk biçim

```
from mymod import f1, f2, v1, v2
```

Tüm nesneleri içe aktar.

isim uzayı:

Aynı isimin farklı isim uzaylarında karışıklık olmadan yer almasına olanak sağlayan sözdizimsel kaptır. Python'da, modüller, sınıflar, fonksiyonlar, metotların hepsi bir isim uzayı oluşturur.

isim çakışması:
Bir isim uzayında iki veya daha fazla ismin karışması durumudur.

```
from string import *
```

yerine

```
import string
```

kullanımı isim çakışmasını önleyecektir.

özellik:
Bir modül içerisinde tanımlanmış (veya sınıf veya örnek -- daha sonra göreceğimiz gibi) değişkendir. Modül özelliklerine **nokta işleci** kullanılarak erişilmektedir.(.).

nokta işleci:
Nokta işleci (.) bir modülün özellik ve fonksiyonlarına (veya sınıf - örneğin özellik ve metotlarına) erişmeyi sağlar.

metot:
Bir nesnenin fonksiyon benzeri özelliğidir. Metotlar nesne üzerinde nokta işleci kullanılarak çağrılırlar (invoke). Örneğin:

```
>>> s = "this is a string."
>>> s.upper()
'THIS IS A STRING.'
>>>
```

Deriz ki, s karakter dizisinin upper metotu çağrıldı. s upper metotunun ilk argümanıdır.

geçici bellek:
Durumunu korumak için elektrik akımına ihtiyaç duyan bellektir. Bilgisayarın Ana bellek veya RAMi geçicidir. RAMde saklanan bilgi bilgisayar kapandığında kaybolur.

kalıcı bellek:
Güç olmadan da durumunu koruyan hafızadır. Sabit diskler, flash sürücüler, tekrar yazılabilir CDlerin her biri kalıcı belleğin örnekleridir.

dosya:
Genellikle sabit disk, floppy disk veya CD-ROM'da saklanan isimli varlıklardır, karakter dizileri, akımları içerir.

mod:
Bilgisayar programında ayrı işlem metotlarıdır. Python'da dosyalar üç modta açılabilirler: okuma ('r'), yazma ('w') ve ekleme ('a').

yol (konum) (path):
Bir dosyanın gerçek konumunu belirten dizin isimleri serisidir.

metin dosyası:
Yazdırılabilir karakterler içeren ve yeni satır karakterleriyle ayrılmış satırlar içeren dosyadır.

continue cümlesi:
Döngünün adımını sonlandıran cümledir. Akış döngünün başına gider, koşulu değerlendirir, ve uygun şekilde döngüye devam eder.

dosya sistemi:
İçerdikleri veriyi ve dosyaları isimlendirmek, erişmek ve düzenlemek için bir yöntem.

dizin:
Dosyaların isimlendirilmiş kolleksiyonudur. Dizinler başka dizinleri - dizinin alt dizinleri adını alır - ve dosyaları barındırırlar.

yol (konum) (path):
Bir dosya sisteminde bir dosyanın isim konumudur. Örneğin:

```
/usr/share/dict/words
```

/ kök dizini altındaki usr altdizininin share altdizininin dict altdizini altındaki words dosyasını belirtmektedir.

ayırıcı:
Bir metinin farklı parçaları arasındaki sınırı belirleyen tek veya daha fazla karakter serisidir.

komut satırı:
Komut satırı arayüzünde komut yorumlayıcıya yazılacak karakter serisidir (Ayrıntılı bilgi için komut satırı.

komut satırı argümanı:
Program başlatılırken programa komut satırı arayüzünün komut isteminden geçirilen değerdir.

Jython:
Python programlama dilinin Java'da yazılmış bir gerçekleştirimidir.(Daha fazla bilgi için Jython ana sayfası http://www.jython.org adresini tıklayın)

argv:
argv argüman vektörünün kısaltmasıdır ve sys modülü içerisinde komut satırı argümanlarının listesini tutan bir değişkendir.

10.13 Alıştırmalar

1. Aşağıdakileri yapın:
   - pydoc sunucusunu komut satırında pydoc -g yazarak başlatın.
   - pydoc tk penceresinde open browser düğmesine tıklayın
   - calendar modülünü bulup tıklayın
   - Functions bölümüne bakarken, aşağıdakileri Python kabuğunda deneyin:

```
>>> import calendar
>>> year = calendar.calendar(2008)
```

```
>>> print year                          # Burada ne olur?
```

- calendar.isleap ile deneyler yapın. Argüman olarak ne almaktadır? Sonuç olarak ne döndürmektedir? Bu ne tür bir fonksiyondur?

Bu alıştırmadan öğrendiklerinize dair ayrıntılı notlar oluşturun.

2. Eğer bilgisayarınızda Tkinter kurulu değilse, pydoc -g hata döndürecektir. Çünkü grafik penceresi Tkinter'e ihtiyaç duymaktadır. Bir alternatif çözüm web sunucusu ile başlatmaktır:

```
$ pydoc -p 7464
```

Bu pydoc web sunucusunu 7464 portunda başlatır. Ağ tarayıcınızda aşağıdaki adresi giriniz:

```
http://localhost:7464
```

bu adresle sisteminizde kurulu Python kütüphanelerini gezebilirsiniz.

Bu yaklaşımı kullanarak pydoc'u başlatın ve math modülünü gözden geçirin.

- math modülünde kaç fonksiyon vardır?
- math.ceil ne yapar? Peki math.floor?(ipucu: hem floor hem de ceil kayan noktalı argüman kabul etmektedir)
- math modülünü kullanmadan math.sqrt ile aynı değeri nasıl hesaplayabileceğimizi açıklayın?
- math modülündeki iki veri sabiti nedir?

Bu alıştırmadaki keşiflerinizin ayrıntılı notlarını tutun.

3. pydoc ile copy mdülünü inceleyin. deepcopy ne işe yaramaktadır? Önceki bölümdeki hangi alıştırmada deepcopy işimize yarar?
4. mymodule1.py isminde bir modül yaratın. Modüle şu anki yaşınızı tutan myage ve şu anki yılı tutan year özelliklerini ekleyin. mymodule2 isminde başka bir modül yaratın, bu modüle de 0'a eşit myage ve doğduğunuz yıla eşit year özelliklerini ekleyin.

Şimdi namespace_test.py isminde bir dosya yaratın. İki modülü de içe aktarın ve aşağıdaki cümleleri yazın:

```
print (mymodule2.myage - mymodule1.myage) == (mymodule2.year - mymodule1.year)
```

namespace_test.py'i çalıştırdığınızda, bu yıl doğum gününüzü kutlayıp kutlamadığınıza bağlı olarak True veya False değerlerinden birini göreceksiniz.

5. Aşağıdaki cümleyi mymodule1.py, mymodule2.py, ve namespace_test.py dosyalarına ekleyin:

```
print "My name is %s" % __name__
```

namespace_test.py'ı çalıştırın. Ne oldu? Neden? Şimdi aşağıdakini mymodule1.py'nin en altına ekleyin:

```
if __name__ == '__main__':
    print "This won't run if I'm  imported."
```

mymodule1.py ve namespace_test.py'i tekrar çalıştırın. Hangi durumda print cümlesinin yazdırdığını görüyoruz?

6. Bir Python kabuğunda aşağıdakileri deneyin:

```
>>> import this
```

İsim uzayları hakkında Tim Peter bize ne söylüyor?

7. pydoc'u kullanarak string modülündeki diğer üç fonksiyonu bulup, sınayın. Bulgularınızı kaydedin.
8. Önceki bölümdeki matrix_mult fonksiyonunu öğrendiğiniz liste metotlarıyla tekrar yazın.
9. dir fonksiyonu, daha önce 7. bölümde görmüştük, argüman olarak aktarılan nesnenin özelliklerinin listesini ekranda görüntülemektedir. Başka bir deyişle, dirargümanının isim uzayının içeriğini döndürmektedir.

   dir(str) ve dir(list) komutlarını kullanarak bu bölümde anlatılmamış olan en az üç tane metot bulun. İki alt çizgi ile başlayan (__) herşeyi şimdilik görmezden gelin. Bulgularınızın isimleri ve kullanım örnekleriyle birlikte ayrıntılı notlarını oluşturmayı unutmayın.

   (ipucu: incelemek istediğini fonksiyonunun docstring'ini yazdırın. Örneğin str.join'un nasıl çalıştığını görmek için print str.join.__doc__ cümlesini kullanın.)

10. Bir yorumlayıcı oturumunda aşağıdaki komutların ne çıktı vereceğini yazınız:

```
>>> s = "If we took the bones out, it wouldn't be
crunchy, would it?"
>>> s.split()

>>> type(s.split())

>>> s.split('o')

>>> s.split('i')

>>> '0'.join(s.split('o'))
```

Her sonucu neden aldığınızı mutlaka anlayın. Aşağıdaki fonksiyonun gövdesini doldurmak için str nesnelerinin split ve join metotlarını kullanarak öğrendiklerinizi uygulayın:

```
def myreplace(old, new, s):
    """
    Replace all occurances of old with new in the string s.

      >>> myreplace(',', ';', 'this, that, and, some, other,
thing')
      'this; that; and; some; other; thing'
      >>> myreplace(' ', '**', 'Words will now be separated by
stars.')
      'Words**will**now**be**separated**by**stars.'
    """
```

Çözümünüz tüm doctestleri geçmelidir.

11. En altında aşağıdakileri içeren wordtools.py isminde bir modül yaratın:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Bu cümlenin nasıl hem sınamak hem de kullanmak için kullanılabildiğini açıklayınız. Eğer wordtools.py başka bir modülde içe aktarılırsa __name__'in değeri ne olur? Ana program olarak çalışırsa ne olur? Hangi durumda doctestler çalışır?

Şimdi aşağıdaki fonksiyonları doctestlerin geçmesi için ekleyin ve içlerini doldurun:

```
def cleanword(word):
    """
      >>> cleanword('what?')
      'what'
      >>> cleanword('"now!"')
      'now'
      >>> cleanword('?+="word!,@$()"')
      'word'
    """

def has_dashdash(s):
    """
      >>> has_dashdash('distance--but')
      True
      >>> has_dashdash('several')
      False
      >>> has_dashdash('critters')
      False
```

```
      >>> has_dashdash('spoke--fancy')
      True
      >>> has_dashdash('yo-yo')
      False
    """

def extract_words(s):
    """
      >>> extract_words('Now is the time!  "Now", is the time?
Yes, now.')
      ['now', 'is', 'the', 'time', 'now', 'is', 'the', 'time',
'yes', 'now']
      >>> extract_words('she tried to curtsey as she spoke--
fancy')
      ['she', 'tried', 'to', 'curtsey', 'as', 'she', 'spoke',
'fancy']
    """

def wordcount(word, wordlist):
    """
      >>> wordcount('now', ['now', 'is', 'time', 'is', 'now',
'is', 'is'])
      ['now', 2]
      >>> wordcount('is', ['now', 'is', 'time', 'is', 'now',
'is', 'the', 'is'])
      ['is', 4]
      >>> wordcount('time', ['now', 'is', 'time', 'is', 'now',
'is', 'is'])
      ['time', 1]
      >>> wordcount('frog', ['now', 'is', 'time', 'is', 'now',
'is', 'is'])
      ['frog', 0]
    """

def wordset(wordlist):
    """
      >>> wordset(['now', 'is', 'time', 'is', 'now', 'is', 'is'])
      ['is', 'now', 'time']
      >>> wordset(['I', 'a', 'a', 'is', 'a', 'is', 'I', 'am'])
      ['I', 'a', 'am', 'is']
      >>> wordset(['or', 'a', 'am', 'is', 'are', 'be', 'but',
'am'])
      ['a', 'am', 'are', 'be', 'but', 'is', 'or']
    """

def longestword(wordset):
    """
      >>> longestword(['a', 'apple', 'pear', 'grape'])
```

```
    5
    >>> longestword(['a', 'am', 'I', 'be'])
    2
    >>> longestword(['this', 'that',
'supercalifragilisticexpialidocious'])
    34
   """
```

Bu modülü kaydedin, daha sonra programlarınızda bu araçları kullanabilirsiniz.

12. unsorted_fruits.txt dosyası her biri farklı bir karakterle başlayan 26 tane meyve içermektedir.

    Bu dosyayı (unsorted_fruits.txt) okuyan ve içerisindeki meyveleri alfabetik olarak sıralı bir şekilde sorted_fruits.txt dosyasına yazan sort_fruits.py isminde bir program yazın.

13. countletters.py hakkındaki aşağıdaki soruları yanıtlayın:
    - Aşağıdaki üç satırın ne yaptığını ayrıntılı olarak açıklayın:

```
infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()
```

    Bu satırlar çalıştırıldıktan sonra type(text) ne döndürür?
    - 128 * [0] ifadesinin sonucu ne olur? evaluate to? ASCII hakkında Wikipedia'dan bilgi alın ve neden counts değişkenine neden 128 * [0] atanmıştır açıklayın.
    - Aşağıdaki ifade

```
for letter in text:
    counts[ord(letter)] += 1
```

    counts'a ne yapar?
    - display fonksiyonunun amacını açıklayın. Neden 10, 13, ve 32 değerlerini sınar? Bu değerler neden özeldir?
    - Aşağıdaki satırların ne yaptığını ayrıntılı olarak anlatın

```
outfile = open('alice_counts.dat', 'w')
outfile.write("%-12s%s\n" % ("Character", "Count"))
outfile.write("================\n")
```

    Çalışma tamamlandığında alice_counts.dat'da ne olur?
    - Son olarak, aşağıdakileri ayrıntılı olarak açıklayın

```
for i in range(len(counts)):
    if counts[i]:
        outfile.write("%-12s%d\n" % (display(i), counts[i]))
```

if counts[i]'in amacı nedir?

14. Write a program named mean.py that takes a sequence of numbers on the command line and returns the mean of their values.

```
$ python mean.py 3 4
3.5
$ python mean.py 3 4 5
4.0
$ python mean.py 11 15 94.5 22
35.625
```

A session of your program running on the same input should produce the same output as the sample session above.

15. Komut satırından verilmiş olan sayıların medyan değerini bulan median.py isminde bir program yazınız.

```
$ python median.py 3 7 11
7
$ python median.py 19 85 121
85
$ python median.py 11 15 16 22
15.5
```

Programınızın bir oturumunda verilen aynı girdiler için aynı çıktılar üretilmelidir, yukarıdaki örnekte görüldüğü gibi çalışmalı.

16. countletters.py programını dosyayı komut satırı argümanı olarak alacak şekilde değiştirin. Çıktı dosyasını isimlendirmeyi nasıl çözersiniz?

11. Özyineleme ve istisnalar

11.1 Tuplelar ve değişebilirlik

Şimdiye kadar iki bileşik tip gördünüz: karakter dizileri, karakterlerden oluşmaktadır; ve listeler, herhangi bir tipte öğelerden oluşmaktadır. Bu iki bileşik tip arasındaki farklardan biri listenin öğelerinin değiştirilebilir olduğu, karakter dizisindeki karakterlerin değiştirilemez olduğuydu. Başka bir deyişle, karakter dizileri **değiştirilemez (immutable)** ve listeler **değiştirilebilir (mutable)** tiplerdir.

Tuple, liste gibi herhangi bir tipteki öğelerin ardışıklığıdır. Ancak listelerden farklı olarak tuplelar değiştirilemez. Sözdizim olarak tuplelar virgül ile ayrılmış değerlerin sıralanmasıdır:

```
>>> tup = 2, 4, 6, 8, 10
```

Her ne kadar gerekmese de, tupleları parantezlerin arasına yazmak gelenekselleşmiştir:

```
>>> tup = (2, 4, 6, 8, 10)
```

Tek öğeli bir tuple yaratmak için son virgülü yazmamız gerekmektedir:

```
>>> tup = (5,)
>>> type(tup)
<type 'tuple'>
```

Virgülsüz Python (5) ifadesine parantezler içerisinde yer alan bir tamsayı olarak davranır:

```
>>> tup = (5)
>>> type(tup)
<type 'int'>
```

Sözdizim farklarını bir kenara bırakırsak, tuplelar karakter dizileri ve listelerdeki aynı dizi işlemlerini desteklerler. İndis işleci tupledan belirli bir öğeyi seçmek için kullanılır.

```
>>> tup = ('a', 'b', 'c', 'd', 'e')
>>> tup[0]
'a'
```

Dilim işleci de belirli bir aralıktaki öğeleri seçer.

```
>>> tup[1:3]
('b', 'c')
```

Ancak eğer atamayla bir öğeyi değiştirmek istersek, bir hatayla karşılaşırız:

```
>>> tup[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Elbette, tuple öğelerini değiştiremesek te, tupleı başka bir tuple ile değiştirebiliriz:

```
>>> tup = ('X',) + tup[1:]
>>> tup
('X', 'b', 'c', 'd', 'e')
```

Alternatif olarak, ilk önce listeye çevirip, değiştirip, tekrar bir tuplea dönüştürebiliriz:

```
>>> tup = ('X', 'b', 'c', 'd', 'e')
>>> tup = list(tup)
>>> tup
['X', 'b', 'c', 'd', 'e']
>>> tup[0] = 'a'
>>> tup = tuple(tup)
>>> tup
('a', 'b', 'c', 'd', 'e')
```

11.2 Tuple ataması

Arada sırada, iki değişkenin değerlerini karşılıklı olarak aktarmak yararlıdır. Geleneksel atama cümleleriyle, bir geçici değişken kullanmamız gerekir. Örneğin, a ve b'yi karşılıklı değiştirmek için:

```
gecici = a
a = b
b = gecici
```

Eğer bunu sıkça yapmamız gerekirse, bu işlem yorucu olacaktır. Python **tuple ataması**nın bir biçimini bu sorunu çözmek üzere sağlamaktadır:

```
a, b = b, a
```

Sol taraf değişkenlerin tupleıdır, sağ taraf ise değerlerin. Her bir değer ilgili değişkene atanmaktadır. Sağ taraftaki tüm deyimler herhangi bir atama olmadan önce işlenmektedir. Bu özellik tuple atamasını çok amaçlı kılmaktadır.

Doğal olarak soldaki değişkenlerin ve sağdaki değerlerin sayısı aynı olmalıdır:

```
>>> a, b, c, d = 1, 2, 3
ValueError: need more than 3 values to unpack
```

11.3 Geri dönüş değerleri olarak Tuplelar

Fonksiyonlar tupleları geri dönüş değeri olarak döndürebilirler. Örneğin, iki parametreyi birbiriyle değiştiren bir fonksiyon yazabiliriz:

```
def swap(x, y):
    return y, x
```

Daha sonra geri dönüş değerini iki değişkenli bir tuplea atayabiliriz:

```
a, b = swap(a, b)
```

Bu durumda, swapi bir fonksiyon yapmanın çok bir avantajı yoktur. Gerçekte, swapi sarmanın (encapsulate) bir tehlikesi vardır, aşağıdaki cezbedici bir yanlıştır:

```
def swap(x, y):       # yanlis surum
    x, y = y, x
```

Eğer fonksiyonu şu şekilde çağırırsak:

```
swap(a, b)
```

a ve x aynı değerin rumuzudur. x'i swap içerisinde değiştirmek x'in başka bir değeri göstermesini, ancak a'nın __main__ içinde etkilenmemesini sağlar. Benzer olarak y'i değiştirmenin de b üzerinde etkisi olmayacaktır.

Bu fonksiyon hata mesajı üretmeden çalışacaktır, ama yapmak istediğimizi yapmayacaktır. Bu tipik bir anlambilim hatasıdır.

11.4 Saf fonksiyonlar ve değiştiriciler - gözden geçirme

9. bölümde listelerle ilişkili olarak saf fonksiyonları ve değiştiricileri (modifier) tartışmıştık. Tuplelar değiştirilemez olduklarına göre üzerlerine bir değiştirici yazamayız.

Burada değiştirici listenin ortasına yeni bir değer eklemektedir:

```
#
# seqtools.py
#

def insert_in_middle(val, lst):
   middle = len(lst)/2
   lst[middle:middle] = [val]
```

Nasıl çalıştığını görmek için yürütelim:

```
>>> from seqtools import *
>>> my_list = ['a', 'b', 'd', 'e']
>>> insert_in_middle('c', my_list)
>>> my_list
['a', 'b', 'c', 'd', 'e']
```

Tuple ile kullanmak istersek, bir hatayla karşılaşırız:

```
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_tuple)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "seqtools.py", line 7, in insert_in_middle
    lst[middle:middle] = [val]
TypeError: 'tuple' object does not support item assignment
>>>
```

Problem tupleların değiştirilemez olması ve dilim atamasını desteklememesidir. insert_in_middle fonksiyonunu saf fonksiyon yapmanın basit bir yöntemi şudur:

```
def insert_in_middle(val, tup):
    middle = len(tup)/2
    return tup[:middle] + (val,) + tup[middle:]
```

Bu sürüm tuplelar için şimdi çalışacaktır, ama liste veya karakter dizileri için çalışmayacaktır. Eğer tüm dizi tipleri için çalışan bir sürüm istiyorsak, kendi değerimizi uygun dizi tipine saran bir yol bulmalıyız. Basit bir yardımcı fonksiyon işimizi görecektir:

```
def encapsulate(val, seq):
    if type(seq) == type(""):
        return str(val)
    if type(seq) == type([]):
        return [val]
    return (val,)
```

Şimdi insert_in_middle fonksiyonunu tüm yerleşik dizi tipleriyle çalışacak şekilde yazabiliriz:

```
def insert_in_middle(val, seq):
    middle = len(seq)/2
    return seq[:middle] + encapsulate(val, seq) + seq[middle:]
```

insert_in_middle fonksiyonunun son iki sürümü saf fonksiyondur. Herhangi bir yan etkileri yoktur. encapsulate ve insert_in_middle'ı seqtools.py modülüne ekleyerek, sınayabiliriz:

```
>>> from seqtools import *
>>> my_string = 'abde'
>>> my_list = ['a', 'b', 'd', 'e']
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_string)
'abcde'
>>> insert_in_middle('c', my_list)
['a', 'b', 'c', 'd', 'e']
>>> insert_in_middle('c', my_tuple)
('a', 'b', 'c', 'd', 'e')
>>> my_string
'abde'
```

my_string, my_list, ve my_tuple değerleri değişmedi. insert_in_middle fonksiyonunun onları değiştirmesini istiyorsak, ürettiğimiz değeri geri dönüş değeri olarak fonksiyondan döndürmeliyiz:

```
>>> my_string = insert_in_middle('c', my_string)
>>> my_string
'abcde'
```

11.5 Özyineli veri yapıları

Şu ana kadar gördüğümüz tüm Python veri tipleri listelerde ve tuplelarda değişik yollarla gruplanabilir. Listeler ve tuplelar da içiçe yerleştirilebilir, verinin düzenlenmesi için sayısız olasılık sunar. Kullanımını kolaylaştırmak için verinin düzenlenmesine **veri yapısı** adı verilir.

Seçim zamanı ve biz de oylar geldikçe hesaplanmasına yardımcı oluyor. Oylar farklı şehir, köy, kasaba, ilçe ve illerden geliyor. Bazen toplu oylar, bazen de alt toplamlardan oluşan listeler şeklinde gönderiliyor. Sayımların en iyi ne şekilde saklanacağına dair kafa patlattıktan sonra içiçe sayı listesi kullanmaya karar veriyoruz. İçiçe sayı listesini şu şekilde tanımlıyoruz:

içiçe sayı listesi aşağıdakilerden herhangi biri öğesi olan bir listedir:

1. sayılar
2. içiçe sayı listesi

Terimin, içiçe sayı listesi kendi tanımını barındırdığına dikkat edin. Bunun gibi **özyineli tanımlamalar** matematik ve bilgisayar bilimlerinde yaygındır. **Özyineli veri yapıları**nı tanımlamak için öz ve güçlü bir yol sağlarlar. Özyineli veri yapıları kendilerinin daha küçük ve basit örneklerinden oluşmaktadır. Tanım döngüsel değildir, eninde sonunda öğesi olarak herhangi bir liste içermeyen bir listeye erişeceğiz.

Şimdi içiçe sayı listesindeki tüm değerleri toplayan bir fonksiyon yazmamız gerektiğini varsayalım. Python bir dizideki sayıların toplamını bulmaya yarayan yerleşik bir fonksiyona sahiptir:

```
>>> sum([1, 2, 8])
11
>>> sum((3, 5, 8.5))
16.5
>>>
```

Ancak bizim içiçe sayı listemiz için sum çalışmayacaktır:

```
>>> sum([1, 2, [11, 13], 8])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>
```

Problem şu ki, listenin üçüncü öğesi, [11, 13], bir listedir ve 1, 2 ve 8 değerlerine eklenemez.

11.6 Özyineleme

Özyineli içiçe sayı listemizdeki tüm sayıları toplamak için listeyi dolaşmak, ve içiçe geçmiş yapısındaki tüm öğeleri dolaşmak, karşılaştığımız her nümerik sayıyı toplamımıza eklemek ve herhangi bir liste öğesi için bu süreci tekrarlamak gerekmektedir.

Modern programlama dilleri genellikle **özyineleme**yi desteklemektedir, özyineleme fonksiyonların kendi tanımlarında kendilerini çağırmasıdır. Özyineleme sayesinde, içiçe sayı listesindeki değerlerin toplamını bulacağımız Python kodu şaşırtıcı derecede kısadır:

```
def recursive_sum(nested_num_list):
    sum = 0
    for element in nested_num_list:
        if type(element) == type([]):
            sum = sum + recursive_sum(element)
        else:
            sum = sum + element
    return sum
```

recursive_sum fonksiyonunun gövdesi ağırlıklı olarak nested_num_list'i dolaşan bir for döngüsünden oluşmaktadır. Eğer element nümerik bir sayı ise (else dalı), basitçe değeri sum'a eklenmektedir. Eğer element liste ise, recursive_sum fonksiyonu element argümanı ile tekrar çağrılmaktadır. Fonksiyon tanımlaması içerisindeki kendisini çağıran cümle **özyineli çağrı** olarak bilinir.

Özyineleme gerçekten bilgisayar bilimlerindeki en güzel ve en şık araçlardan biridir.

Biraz daha karmaşık bir problem olan içiçe sayı listesindeki en büyük değeri bulmak:

```
def recursive_max(nested_num_list):
    """
      >>> recursive_max([2, 9, [1, 13], 8, 6])
      13
      >>> recursive_max([2, [[100, 7], 90], [1, 13], 8, 6])
      100
      >>> recursive_max([2, [[13, 7], 90], [1, 100], 8, 6])
      100
      >>> recursive_max([[[13, 7], 90], 2, [1, 100], 8, 6])
      100
    """
    largest = nested_num_list[0]
    while type(largest) == type([]):
```

```
        largest = largest[0]

    for element in nested_num_list:
        if type(element) == type([]):
            max_of_elem = recursive_max(element)
            if largest < max_of_elem:
                largest = max_of_elem
        else:                               # element is not a list
            if largest < element:
                largest = element

    return largest
```

Doctestler recursive_max çalışmasının örneklerini sağlamak için verilmiştir.

Bu probleme nümerik bir değeri bulmak için eklenmiş olan kıvırma largest'in ilklenmesidir. nested_num_list[0]'ı kullanamayız, çünkü bu değer sayı da olabilir, bir liste de. Bu sorunu çözmek için largest'a hangi derinliğe inerse insin ilk nümerik değeri atayan bir while döngüsü kullanırız.

Yukarıdaki iki örnekte özyineli çağrıya neden olmayan bir **temel durum**a sahiptir: öğenin bir sayı olduğu, liste olmadığı durum. Temel durum olmadan, **sonsuz özyineleme**ye girersiniz ve programınız çalışmaz. Python belirli bir maksimum özyineleme derinliğine ulaştığında çalışmayı durdurup, bir çalışma zamanı hatası döndürür.

Aşağıdaki kodu infinite_recursion.py isimli bir dosyaya yazın:

```
#
# infinite_recursion.py
#
def recursion_depth(number):
    print "Recursion depth number %d." % number
    recursion_depth(number + 1)

recursion_depth(0)
```

Programı kaydettiğiniz dizinde komut satırında aşağıdaki komutu çalıştırın:

```
python infinite_recursion.py
```

Mesajları geçişini izledikten sonra, uzun bir mesaj sonrasında aşağıdakiyle karşılaşacaksınız:

```
  ...
  File "infinite_recursion.py", line 3, in recursion_depth
    recursion_depth(number + 1)
RuntimeError: maximum recursion depth exceeded
```

Bunun gibi bir durumun programlarımızdan birinde oluşmasını hiç bir zaman istemeyiz. Özyinelemeyi anlatmayı bitirmeden önce, Python'da hataların nasıl kotarıldığını öğrenelim.

11.7 İstisnalar (Exceptions)

Her ne zaman bir çalışma zamanı hatası oluşursa, bir **istisna** ortaya çıkar. Program bu noktada çalışmasını durdurur ve hatanın dökümünü verir, bu döküm ortaya çıkan istisnayla biter.

Örneğin sıfırla bölme aşağıdaki istisnayı yaratır:

```
>>> print 55/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Var olmayan bir liste öğesine erişmeye çalışmak:

```
>>> a = []
>>> print a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Veya tupleda öğe ataması yapmaya çalışmak:

```
>>> tup = ('a', 'b', 'd', 'd')
>>> tup[2] = 'c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Her bir durumda, sondaki hata mesajları iki parçadır: iki nokta üstüsteden önce hatanın tipi, ve iki nokta üstüstüden sonra hatanın ayrıntıları.

Bazı durumlarda istisna yaratabilecek işlemler yapmak isteyebiliriz, ancak programın çalışmasını durdurmasını istemeyiz. Bu gibi durumlarda **istisna işleme (kotarma, handling)** yapmamız gerekir. Bunu da try ve except cümleleriyle yaparız.

Örneğin, kullanıcıdan bir dosya ismi sorarız ve daha sonra bu dosyayı açmaya çalışırız. Eğer dosya yoksa, programın çökmesini değil, istisnayı işlemeyi isteriz:

```
filename = raw_input('Enter a file name: ')
try:
    f = open (filename, "r")
```

```
except:
    print 'There is no file named', filename
```

try cümlesi ilk bloktaki cümleleri işletir. Eğer istisna oluşmazsa, except bloğunu yoksayar. Eğer herhangi bir istisna oluşursa, except dalı içerisindeki cümleleri çalıştırır ve devam eder.

Bu yeteneği bir fonksiyon içerisine sarabiliriz: exists fonksiyonu bir dosya ismi alır ve eğer dosya varsa doğru, yoksa yanlış döndürür:

```
def exists(filename):
    try:
        f = open(filename)
        f.close()
        return True
    except:
        return False
```

Farklı istisnaları işlemek için birden fazla except kullanabilirsiniz (istisnalar hakkında ayrıntılı bilgi için Python'u yaratan Guido Van Rossum'un Python Öğrencesindeki Hatalar ve İstisnalar dersini inceleyebilirsiniz).

Eğer programınız hata durumunu tespit ederse, bir istisna **tetikleyebilirsiniz**. Aşağıda kullanıcıdan girdi alan ve sayının negatif olmadığını kontrol eden bir örnek görebilirsiniz:

```
#
# learn_exceptions.py
#
def get_age():
    age = input('Please enter your age: ')
    if age < 0:
        raise ValueError, '%s is not a valid age' % age
    return age
```

raise cümlesi iki argüman alır: istisna tipi, ve hata hakkında ayrıntılı bilgi. ValueError tetiklemek istediğimiz istisnaya en yakın yerleşik istisnadır. Yerleşik istisnaların bütün listesine Python kütüphane referansının 2.3 bölümünden ulaşabilirsiniz (yine Guido Van Rossum'un yazdığı).

Eğer çağırdığı fonksiyon get_age hatayı kotarırsa, program çalışmaya devam edecektir, diğer durumda Python hata dökümünü yapıp çıkacaktır:

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
Please enter your age: -2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError, '%s is not a valid age' % age
ValueError: -2 is not a valid age
>>>
```

Hata mesajı bizim sağladığımız istisna tipini ve ek bilgiyi içermektedir.

İstisna işlemeyi kullanarak, infinite_recursion.py'yi değiştirip, izin verilen maksimum özyineleme derinliğine ulaştığında durmasını sağlayabiliriz:

```
#
# infinite_recursion.py
#
def recursion_depth(number):
    print "Recursion depth number %d." % number
    try:
        recursion_depth(number + 1)
    except:
        print "Maximum recursion depth exceeded."

recursion_depth(0)
```

Bu sürümü çalıştırıp, sonuçları gözlemleyin.

11.8 Kuyruk özyineleme

Fonksiyon tanımlamasının sonunda bir özyineli çağrı oluşursa, buna **kuyruk özyineleme** adı verilir.

Aşağıda 6. bölümde yazdığımız gerisayim fonksiyonunun kuyruk özyineleme ile tekrar yazılmış sürümünü görebilirsiniz:

```
def gerisayim(n):
    if n == 0:
        print "Defol!"
    else:
        print n
        gerisayim(n-1)
```

Yineleme ile yapılan herhangi bir hesaplama, özyineleme ile de yapılabilir.

Bir çok bilinen matematiksel fonksiyon özyineli olarak tanımlanmıştır. Faktöryel, örneğin, özel bir işleçe, !, sahiptir ve şu şekilde tanımlanmıştır:

```
0! = 1
n! = n(n-1)
```

Bunu Pythonda kolayca kodlayabiliriz:

```
def faktoryel(n):
    if n == 0:
```

```
        return 1
    else:
        return n * faktoryel(n-1)
```

Matematikte çok bilinen diğer bir özyineli ilişki fibonacci serisidir, şu şekilde tanımlanmıştır:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

Bu da Pythonda kolayca yazılabilir:

```
def fibonacci (n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Hem factorial hem de fibonacci kuyruk özyinelemenin örnekleridir.

Kuyruk özyineleme Python gibi dillerde kötü bir alışkanlık olarak kabul edilir, çünkü eşlenik yinelemeli çözüme göre daha fazla sistem kaynağı kullanmaktadır.

faktoryel(1000)i çalıştırmak maksimum özyineleme derinliğini aşacaktır. Ve fibonacci(35) çalıştırmayı deneyin, ve hesaplamanın ne kadar uzun sürdüğünü gözlemleyin (sabırla bekleyin, hesaplama bitecektir).

faktoryelin yinelemeli bir sürümünü yazmanız alıştırma olarak sorulacak, ve bir sonraki bölümde fibonacciyi daha iyi bir yolla çözeceğiz.

11.9 Liste kavraması

Liste kavraması kısa, matematiksel bir sözdizimi kullanarak diğer listelerden yeni listeler yaratmaya yarayan sözdizimsel bir yapıdır:

```
>>> sayilar = [1, 2, 3, 4]
>>> [x**2 for x in sayilar]
[1, 4, 9, 16]
>>> [x**2 for x in sayilar if x**2 > 8]
[9, 16]
>>> [(x, x**2, x**3) for x in sayilar]
[(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
>>> dosyalar = ['bin', 'Data', 'Desktop', '.bashrc', '.ssh',
'.vimrc']
>>> [isim for isim in dosyalar if isim[0] != '.']
['bin', 'Data', 'Desktop']
>>> harfler = ['a', 'b', 'c']
>>> [n*harf for n in sayilar for harf in harfler]
```

```
['a', 'b', 'c', 'aa', 'bb', 'cc', 'aaa', 'bbb', 'ccc', 'aaaa',
'bbbb', 'cccc']
>>>
```

Liste kavraması için genel sözdizimi şu şekildedir:

```
[expr for item1 in seq1 for item2 in seq2 ... for itemx in seqx if
condition]
```

Bu liste deyimi aşağıdakiyle aynı etkiye sahiptir:

```
output_sequence = []
for item1 in seq1:
    for item2 in seq2:
        ...
            for itemx in seqx:
                if condition:
                    output_sequence.append(expr)
```

Gördüğünüz gibi, liste kavraması çok daha kısa ve özdür.

11.10 Mini örnek çalışma: tree

Aşağıdaki program Unix'teki tree programının davranışının bir alt kümesini gerçekleştirmektedir.

```
#!/usr/bin/env python

import os
import sys


def getroot():
    if len(sys.argv) == 1:
        path = ''
    else:
        path = sys.argv[1]

    if os.path.isabs(path):
        tree_root = path
    else:
        tree_root = os.path.join(os.getcwd(), path)

    return tree_root


def getdirlist(path):
    dirlist = os.listdir(path)
    dirlist = [name for name in dirlist if name[0] != '.']
```

```
    dirlist.sort()
    return dirlist


def traverse(path, prefix='|--', s='.\n', f=0, d=0):
    dirlist = getdirlist(path)

    for num, file in enumerate(dirlist):
        lastprefix = prefix[:-3] + '`--'
        dirsize = len(dirlist)

        if num < dirsize - 1:
            s += '%s %s\n' % (prefix, file)
        else:
            s += '%s %s\n' % (lastprefix, file)
        path2file = os.path.join(path, file)

        if os.path.isdir(path2file):
            d += 1
            if getdirlist(path2file):
                s, f, d = traverse(path2file, '|    ' + prefix, s, f, d)
        else:
            f += 1

    return s, f, d


if __name__ == '__main__':
    root =  getroot()
    tree_str, files, dirs = traverse(root)

    if dirs == 1:
        dirstring = 'directory'
    else:
        dirstring = 'directories'
    if files == 1:
        filestring = 'file'
    else:
        filestring = 'files'

    print tree_str
    print '%d %s, %d %s' % (dirs, dirstring, files, filestring)
```

Alıştırmaların bir çoğunda bu programı incelemeniz istenecek.

11.11 Sözlük

değiştirilemez veri tipi:

Değiştirilemeyen veri tipidir. Değiştirilemez tiplerin öğelerine veya dilimlerine yapılan atamalar çalışma zamanı hatası yaratır.

değiştirilebilir veri tipi:
Değiştirilebilen veri tipleridir. Tüm değiştirilebilen tipler bileşik tiplerdir. Listeler ve sözlükler değiştirilebilir veri tipidir; karakter dizileri ve tuplelar değildir.

tuple:
Herhangi bir tipte öğe dizisi içeren veri tipidir. Liste gibidir ama değiştirilemez. Tuplelar değiştirilemeyen tiplerin gerektirdiği durumlarda kullanılabilir, örneğin sözlükte (sonraki bölüme bakınız) bir anahtar gibi.

tuple ataması:
Tek bir atama ifadesiyle tupledaki tüm öğelere atamadır. Tuple ataması sıralı yerine paralel bir şekilde gerçekleşir, değerlerin birbirleriyle değiştirilmesi için yararlıdır.

veri yapısı
Kullanımı kolaylaştırmak amacıyla verinin düzenlenmesidir.

özyineli tanımlama
Kendisini kendi terimleriyle tanımlayan ifadedir. Yararlı olabilmesi için özyineli olmayan temel durumları içermelidir. Bu şekilde döngüsel tanımlamadan farklıdır. Özyineli tanımlamalar karmaşık veri yapılarını ifade etmenin şık bir yoludur.

özyineleme:
Hali hazırda çalışan fonksiyonun kendisini çağırmasıdır.

özyineli çağrı:
Özyineli bir fonksiyonda kendisini çağıran ifade.

temel durum:
Özyineli fonksiyondaki koşul cümlesinin özyineli çağrı şeklinde sonuçlanmayan bir dalı.

sonsuz özyineleme:
Temel duruma ulaşmadan sürekli olarak kendisini çağıran fonksiyon. Sonsuz özyineleme çalışma zamanı hatası üretir.

istisna:
Çalışma zamanında oluşan hata.

istisna işleme:
Bir istisnanın programın çalışmasını durdurmasını try ve except cümleleriyle önlemek.

tetikleme:
Bir istisnayı raise cümlesiyle üretmek.

kuyruk özyineleme:
Fonksiyon tanımlamasının son cümlesi olarak (kuyrukta) yer alan özyineli çağrı. Kuyruk özyineleme Python programlarında kötü alışkanlık olarak değerlendirilir. Çünkü mantıksal olarak eş fonksiyonlar yineleme kullanılarak daha etkin bir şekilde yazılabilir (ayrıntılı bilgi için tail recursion).

liste kavrama:
Başka listelerden liste yaratmaya yarayan, matematiksel küme yapım gösterimine benzer sözdizimsel yapı.

11.12 Alıştırmalar

```
1.def swap(x, y):       # yanlis surum
      print  "before swap statement: id(x):", id(x), "id(y):",
  id(y)
      x, y = y, x
      print  "after swap statement: id(x):", id(x), "id(y):",
  id(y)

  a, b = 0, 1
  print  "before swap function call: id(a):", id(a), "id(b):",
  id(b)
  swap(a, b)
  print  "after swap function call: id(a):", id(a), "id(b):",
  id(b)
```

Bu programı çalıştırın ve sonuçları tanımlayın. Sonuçları neden bu swap sürümünün istendiği gibi çalışmadığını açıklamak için kullanın. swap çağrımından sonra a veb'nin değerleri ne olacaktır?

2. seqtools.py bir modül yaratın. Bu bölümden encapsulate ve insert_in_middle fonksiyonlarını ekleyin.Her üç dizi tipi için doğru çalışıp çalışmadığını sınayan doctestleri ekleyin.
3. Aşağıdaki fonksiyonların herbirini seqtools.py dosyasına ekleyin:

```
  def make_empty(seq):
      """
        >>> make_empty([1, 2, 3, 4])
        []
        >>> make_empty(('a', 'b', 'c'))
        ()
        >>> make_empty("No, not me!")
        ''
      """

  def insert_at_end(val, seq):
      """
        >>> insert_at_end(5, [1, 3, 4, 6])
        [1, 3, 4, 6, 5]
        >>> insert_at_end('x', 'abc')
        'abcx'
        >>> insert_at_end(5, (1, 3, 4, 6))
        (1, 3, 4, 6, 5)
      """

  def insert_in_front(val, seq):
      """
        >>> insert_in_front(5, [1, 3, 4, 6])
        [5, 1, 3, 4, 6]
```

```
      >>> insert_in_front(5, (1, 3, 4, 6))
      (5, 1, 3, 4, 6)
      >>> insert_in_front('x', 'abc')
      'xabc'
    """

def index_of(val, seq, start=0):
    """
      >>> index_of(9, [1, 7, 11, 9, 10])
      3
      >>> index_of(5, (1, 2, 4, 5, 6, 10, 5, 5))
      3
      >>> index_of(5, (1, 2, 4, 5, 6, 10, 5, 5), 4)
      6
      >>> index_of('y', 'happy birthday')
      4
      >>> index_of('banana', ['apple', 'banana', 'cherry',
'date'])
      1
      >>> index_of(5, [2, 3, 4])
      -1
      >>> index_of('b', ['apple', 'banana', 'cherry', 'date'])
      -1
    """

def remove_at(index, seq):
    """
      >>> remove_at(3, [1, 7, 11, 9, 10])
      [1, 7, 11, 10]
      >>> remove_at(5, (1, 4, 6, 7, 0, 9, 3, 5))
      (1, 4, 6, 7, 0, 3, 5)
      >>> remove_at(2, "Yomrktown")
      'Yorktown'
    """

def remove_val(val, seq):
    """
      >>> remove_val(11, [1, 7, 11, 9, 10])
      [1, 7, 9, 10]
      >>> remove_val(15, (1, 15, 11, 4, 9))
      (1, 11, 4, 9)
      >>> remove_val('what', ('who', 'what', 'when', 'where',
'why', 'how'))
      ('who', 'when', 'where', 'why', 'how')
    """

def remove_all(val, seq):
    """
```

```
      >>> remove_all(11, [1, 7, 11, 9, 11, 10, 2, 11])
      [1, 7, 9, 10, 2]
      >>> remove_all('i', 'Mississippi')
      'Mssssppp'
    """

def count(val, seq):
    """
      >>> count(5, (1, 5, 3, 7, 5, 8, 5))
      3
      >>> count('s', 'Mississippi')
      4
      >>> count((1, 2), [1, 5, (1, 2), 7, (1, 2), 8, 5])
      2
    """

def reverse(seq):
    """
      >>> reverse([1, 2, 3, 4, 5])
      [5, 4, 3, 2, 1]
      >>> reverse(('shoe', 'my', 'buckle', 2, 1))
      (1, 2, 'buckle', 'my', 'shoe')
      >>> reverse('Python')
      'nohtyP'
    """

def sort_sequence(seq):
    """
      >>> sort_sequence([3, 4, 6, 7, 8, 2])
      [2, 3, 4, 6, 7, 8]
      >>> sort_sequence((3, 4, 6, 7, 8, 2))
      (2, 3, 4, 6, 7, 8)
      >>> sort_sequence("nothappy")
      'ahnoppty'
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Her zamanki gibi, bu fonksiyonları doctestlerin hepsini geçecek şekilde düzeltin.

4. Bir fonksiyon yazın, recursive_min, içiçe sayı listesindeki en küçük değeri döndürsün:

```
def recursive_min(nested_num_list):
    """
      >>> recursive_min([2, 9, [1, 13], 8, 6])
```

```
    1
    >>> recursive_min([2, [[100, 1], 90], [10, 13], 8, 6])
    1
    >>> recursive_min([2, [[13, -7], 90], [1, 100], 8, 6])
    -7
    >>> recursive_min([[[-13, 7], 90], 2, [1, 100], 8, 6])
    -13
    """
```

Fonksiyonunuz doctestleri geçmelidir.

5. Bir fonksiyon yazın, recursive_count, nested_number_list içerisinde target sayısını döndürsün:

```
def recursive_count(target, nested_num_list):
    """
    >>> recursive_count(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]])
    4
    >>> recursive_count(7, [[9, [7, 1, 13, 2], 8], [7, 6]])
    2
    >>> recursive_count(15, [[9, [7, 1, 13, 2], 8], [2, 6]])
    0
    >>> recursive_count(5, [[5, [5, [1, 5], 5], 5], [5, 6]])
    6
    """
```

Yine fonksiyonunuz tüm doctestleri geçmelidir.

6. Bir fonksiyon yazın, flatten, nested_number_list içerisindeki tüm değerleri barındıran basit bir sayı listesi döndürsün:

```
def flatten(nested_num_list):
    """
    >>> flatten([2, 9, [2, 1, 13, 2], 8, [2, 6]])
    [2, 9, 2, 1, 13, 2, 8, 2, 6]
    >>> flatten([[9, [7, 1, 13, 2], 8], [7, 6]])
    [9, 7, 1, 13, 2, 8, 7, 6]
    >>> flatten([[9, [7, 1, 13, 2], 8], [2, 6]])
    [9, 7, 1, 13, 2, 8, 2, 6]
    >>> flatten([[5, [5, [1, 5], 5], 5], [5, 6]])
    [5, 5, 1, 5, 5, 5, 5, 6]
    """
```

Doctestleri geçtiğini doğrulamak için fonksiyonunuzu çalıştırın.

7. readposint isminde bir fonksiyon yazın. Fonksiyon kullanıcıdan bir pozitif tamsayı alsın ve girdiyi gereksinimleri karşılayıp karşılamadığına göre sınasın. Örnek bir oturum aşağıdaki gibi olacaktır:

```
>>> num = readposint()
Please enter a positive integer: yes
yes is not a positive integer.  Try again.
```

```
Please enter a positive integer: 3.14
3.14 is not a positive integer.  Try again.
Please enter a positive integer: -6
-6 is not a positive integer.  Try again.
Please enter a positive integer: 42
>>> num
42
>>> num2 = readposint("Now enter another one: ")
Now enter another one: 31
>>> num2
31
>>>
```

Kullanıcının girdisinin yanlış olduğunu doğrulamak için Python'un istisna işleme düzeneklerini kullanın.

8. Aşağıdakilerin herbiri için Python yorumlayıcının üreteceği sonuçları yazın:

```
1.>>> nums = [1, 2, 3, 4]
  >>> [x**3 for x in nums]

2.>>> nums = [1, 2, 3, 4]
  >>> [x**2 for x in nums if x**2 != 4]

3.>>> nums = [1, 2, 3, 4]
  >>> [(x, y) for x in nums for y in nums]

4.>>> nums = [1, 2, 3, 4]
  >>> [(x, y) for x in nums for y in nums if x != y]
```

Yorumlayıcıda denemeden önce sonuçları kavramanız gerekir.

9. pydoc veya http://pydoc.org adresindeki çevrimiçi belgeleri kullanarak sys.getrecursionlimit() ve sys.setrecursionlimit(n) fonksiyonlarının ne iş yaptığını öğrenin. Bu modül fonksiyonlarının nasıl çalıştığını iyice anlamak için infinite_recursion.py'de olduğu gibi bir çok deney yaratın.
10. faktoryel fonksiyonunu özyineleme yerine yineleme (iteration) kullanarak tekrar yazın. Fonksiyonunuzu 1000 argümanıyla çağırın ve ne kadar sürede sonucu ürettiğini not edin.
11. İsmi litter.py olan bir program yazın, argüman olarak verilen ağacın kökündeki (veya varsayılan olarak verilen o anki dizin) her bir altdizinde trash.txt isminde boş bir dosya yaratsın. Şimdi de cleanup.py isminde bir program yazın, bütün bu dosyaları (dikkat edin, trash.txt dosyalarını sadece) silsin.

    İpucu: Mini örnek çalışmada verdiğimiz tree programını bu iki özyineli program için temel olarak kullanın.

12. Sözlükler

Şimdiye kadar incelediğimiz tüm bileşik tipler -- karakter dizileri, listeler, ve çok öğeliler (tuple) -- ardışık tiplerdir, içerdikleri değerlere erişmek için tamsayılar indis olarak kullanılır.

Sözlükler farklı bileşik tiplerdir. Python'un yerleşik **eşleştirme tipi**dir. Değişmez, sabit olan **Anahtarları (key)** herhangi bir tipteki değerlere eşler, aynı çok öğeli veya listelerdeki değerler gibi.

Örnek olarak, Türkçe kelimeleri İspanyolca'ya çeviren bir sözlük yaratacağız. Bu sözlük için anahtarlar karakter dizisidir.

Sözlük yaratmanın bir yolu boş bir sözlükle başlamak ve daha sonra **anahtar-değer çiftleri** eklemektir. Boş sözlük {} ile gösterilmektedir:

```
>>> tr2sp = {}
>>> tr2sp['bir'] = 'uno'
>>> tr2sp['ikıi'] = 'dos'
```

İlk atama tr2sp adında bir sözlük yaratır; diğer atamalar sözlüğe yeni anahtar-değer çiftlerini ekler. Sözlüğün geçerli değerini her zamanki gibi yazdırabiliriz:

```
>>> print tr2sp
{'iki': 'dos', 'bir': 'uno'}
```

Sözlükteki anahtar-değer çiftleri virgülle ayrılmıştır. Her çift iki nokta üstüste ile ayrılmış bir anahtar ve değer içerir.

Çiftlerin sırası beklendiği gibi olmayabilir. Python sözlükte anahtar-değer çiftlerinin nerede saklanacağını belirlemek için karmaşık algoritmalar kullanır. Amaçlarımız açısından bu sıralamanın önceden kestirilemeyen olduğunu düşünebiliriz

Sözlük yaratmanın bir diğer yolu önceki çıktıda gördüğümüz sözdiziminde anahtar-değer çiftlerinden oluşan bir liste sağlamaktır::

```
>>> tr2sp = {'bir': 'uno', 'iki': 'dos', 'üç': 'tres'}
```

Çiftlerin yazılış sırasının önemi yoktur. Sözlükteki değerlere anahtarlarla erişilir, indislerle değil. Bu yüzden sıralamayla uğraşmaya gerek yoktur.

Aşağıdaki ilgili değeri bulmak için anahtarın nasıl kullanıldığını görebilirsiniz:

```
>>> print tr2sp['iki']
'dos'
```

'iki' anahtarı 'dos' değerini döndürmektedir.

12.1 Sözlük işlemleri

del ifadesi sözlükten anahtar-değer çiftini siler. Örneğin aşağıdaki sözlük meyve isimlerini ve her meyvenin stoktaki miktarını tutmaktadır:

```
>>> stok = {'elma': 430, 'muz': 312, 'portakal': 525, 'erik': 217}
>>> print stok
{'portakal': 525, 'elma': 430, 'erik': 217, 'muz': 312}
```

Eğer birisi tüm erikleri satın alırsa, ilgili kaydı sözlükten silebiliriz:

```
>>> del stok['erik']
>>> print stok
{'portakal': 525, 'elma': 430, 'muz': 312}
```

Eğer yakın zamanda daha fazla erik gelmesini bekliyorsak, sadece eriklerin değerini değiştiririz:

```
>>> stok['erik'] = 0
>>> print stok
{'portakal': 525, 'elma': 430, 'erik': 0, 'muz': 312}
```

len işlemi sözlüklerde de çalışır; anahtar-değer çiftlerinin sayısını döndürür:

```
>>> len(stok)
4
```

12.2 Sözlük metotları

Sözlükler bazı yararlı yerleşik metotlara sahiptir.

keys metotu bir sözlük parametresi alır ve o sözlüğün anahtarlarını döndürür.

```
>>> tr2sp.keys()
['üç', 'iki', 'bir']
```

Karakter dizileri ve listelerde gördüğümüz gibi, sözlük metotları da nokta gösterimini kullanır. Noktanın sağına metotun ismi yazılır ve noktanın solundaki değişkene ilgili metotu uygular. Parantezler bu metotun parametre almadığını belirtir.

Metot çağrımına **çağırma (invocation)** adı verilir, tr2sp nesnesinde keys metodunu çağırdığımızı söyleriz. Nesne yönelimli programlama ile ilgili bölümü incelediğimizde, nesnenin çağrılan metotunun, metotun ilk argümanı olduğunu da göreceğiz.

values metotu benzerdir; sözlükteki değerlerin bir listesini döndürür

```
>>> tr2sp.values()
['tres', 'dos', 'uno']
```

items metotu hem anahtarları hem de değerleri, çok öğeli (tuple) liste şeklinde döndürür:

```
>>> tr2sp.items()
[('üç', 'tres'), ('iki', 'dos'), ('bir', 'uno')]
```

has_key metotu argüman olarak bir anahtar alır ve eğer sözlük anahtarı içeriyorsaTrue içermiyorsa False değerini döndürür:

```
>>> tr2sp.has_key('bir')
True
>>> tr2sp.has_key('deux')
False
```

Bu metot çok yararlı olabilir, çünkü sözlükte olmayan bir anahtara erişmek çalışma zamanı hatasına yol açar:

```
>>> tr2esp['dog']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'dog'
>>>
```

12.3 Rumuz ve kopyalama

Sözlükler değiştirilebilir olduğu için rumuz kullanımına karşı dikkatli olmalısınız. Ne zaman iki değişken aynı nesneyi gösterirse, herhangi birine yapılan değişiklikler diğerini de etkiler.

Eğer sözlüğünüzü değiştirmek ama asılın bir kopyasını saklamak istiyorsanız, copy metotunu kullanmalısınız. Örneğin, karsitlar karşıt çiftleri içeren bir sözlük olsun:

```
>>> karsitlar = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> rumuz = karsitlar
>>> kopya = karsitlar.copy()
```

rumuz ve karsitlar aynı nesneyi göstermektedir; kopya ise aynı sözlüğün yeni bir kopyasını göstermektedir. Eğer rumuzu değiştirirsek, karsitlar da değişecektir:

```
>>> rumuz['right'] = 'left'
>>> karsitlar['right']
'left'
```

Eğer kopyayı değiştirirsek, karsitlar değişmeyecektir:

```
>>> kopya['right'] = 'privilege'
>>> karsitlar['right']
'left'
```

12.4 Dağınık matrisler

Daha önce bir matrisi temsil etmek için listelerden oluşan liste kullanmıştık. Bu genellikle sıfırdan farklı değerler içeren matrisler için iyi bir çözümdür, ancak aşağıdaki gibi bir dağınık matrisi düşünün:

Liste temsili çok sayıda sıfır içerecektir:

```
matris = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

Alternatif bir yöntem sözlük kullanmaktır. Anahtarlar için satır ve sütun numalarını içeren tuple kullanabiliriz. Aşağıda aynı matrisin sözlük gösterimini görebilirsiniz:

```
matris = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

Sadece üç anahtar-değer çiftine sahibiz, matriste sıfır olmayan her bir öğe için. Her bir anahtar iki tamsayı içeren bir tuple'dan oluşuyor.

Matrisin öğelerine erişmek için, [] işlecini kullanabiliriz:

```
matris[(0, 3)]
1
```

Dikkat ederseniz sözlük gösterimi söz dizimi içiçe liste sözdiziminden farklıdır. İki farklı tamsayı indis yerine, bir indisimiz, tamsayılardan oluşan tuple indisimiz var.

Ancak bir sorunumuz var. Eğer sıfır olan bir öğeyi belirtmek istersek, hatayla karşılaşırız, çünkü sözlükte o anahtarın girdisi yok:

```
>>> matris[(1, 3)]
KeyError: (1, 3)
```

get metotu bu sorunu çözer:

```
>>> matris.get((0, 3), 0)
1
```

İlk argüman anahtardır; ikinci argüman ise anahtarın sözlükte olmaması durumunda get tarafından geri döndürülecek değerdir:

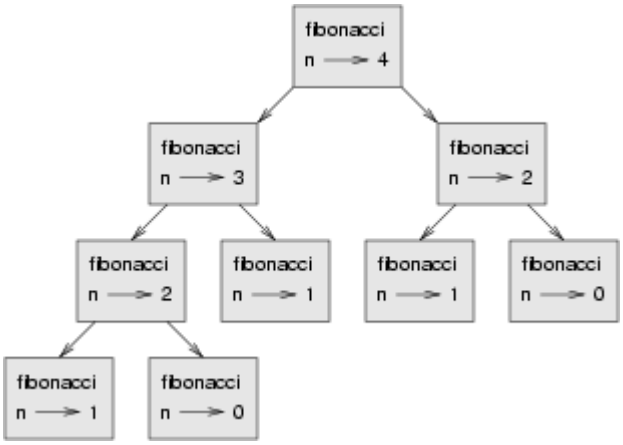
```
>>> matris.get((1, 3), 0)
0
```

get dağınık matrise erişmenin sözdizimini bariz bir şekilde iyileştirir.

12.5 İpuçları

Son bölümdeki fibonacci fonksiyonuyla biraz uğraşırsanız, daha büyük argümanların fonksiyonun çalışmasını daha da uzattığını görürsünüz. Çalışma zamanı oldukça hızlı bir şekilde artmaktadır. Bir makinemize, fibonacci(20) anında biterken, fibonacci(30) yaklaşık olarak 1 sn, ve fibonacci(40) ise neredeyse sonsuz süre almaktadır.

Bunun neden olduğunu anlamak için, fibonacci fonksiyonunun n = 4 için **çağrı çizge**sini incelemek lazım:



Çağrı çizgesi fonksiyon çerçeve kümeleriyle, her çerçeveyi çağırdığı fonksiyonun çerçevelerine bağlayan doğruları gösterir. Çizgenini tepesinde fibonacci n = 4 ve fibonacci n = 3'ü çağırmaktadır. fibonacci n = 3'te fibonnaci n = 2 ve fibonnaci n = 1'i çağırmaktadır. Bu böyle devam eder.

fibonacci(0) ve fibonacci(1)'in kaç kere çağrıldığını sayın. Bu problemin verimsiz bir çözümüdür, ve problemin argümanı büyüdükçe çözüm kötüleşmektedir.

İyi bir çözüm daha önceden hesaplanmış değerlerin bir sözlükte saklanmasıdır. Sonradan çağrılmak üzere saklanmış önceden hesaplanmış bir değer **ipucu (hint)** adını almaktadır. Aşağıda fibonacci fonksiyonunun ipuçlarıyla gerçekleştirimini görebilirsiniz:

```
onceki = {0: 0, 1: 1}

def fibonacci(n):
    if onceki.has_key(n):
        return onceki[n]
    else:
        yeni_deger = fibonacci(n-1) + fibonacci(n-2)
        onceki[n] = yeni_deger
        return yeni_deger
```

onceki adını verdiğimiz sözlük, hali hazırda bildiğimiz Fibonacci sayılarını tutmaktadır. Sadece iki çiftle başlarız: 0 1'e eşlenir ve 1 1'e eşlenir.

Her ne zamanfibonacci çağrılırsa, önce sözlükte değer var mı diye kontrol eder. Eğer değer varsa, herhangi özyineli bir çağrım yapmadan fonksiyon değeri döndürebilir. Eğer yoksa, yeni değeri hesaplaması gerekir. Yeni değer fonksiyon değer döndürmeden önce sözlüğe eklenir.

Bu fibonacci sürümünü kullanarak, makinelerimizin bir göz kırpımında fibonacci(100)'ü hesaplamasını sağlayabiliriz.

```
>>> fibonacci(100)
354224848179261915075L
```

Numaranın sonundaki L ifadesi değerin bir uzun tamsayı olduğunu belirtmektedir.

12.6 Uzun tamsayılar

Python herhangi bir büyüklükteki (elbette makinenizin belleğiyle sınırlıdır :)) tamsayıyı desteklemek için long (uzun tamsayı) tipini sunmaktadır.

Uzun tamsayı değeri oluşturmanın üç yolu vardur. İlki normal bir tamsayıya sığmayacak çok büyük değer üreten bir aritmetik deyim hesaplamaktadır. Bunu daha öncefibonacci(100) örneğinde gördük. Bir başka yöntem tamsayının sağına L harfini koymaktır:

```
>>> type(1L)
<type 'long'>
```

Üçüncü yöntem çevrilecek olan argümanla long'u çağırmaktır. long aynı int ve float't olduğu gibi tamsayıları, kayan noktalıları ve hatta rakamlardan oluşan karakter dizisini uzun tamsayıya çevirir:

```
>>> long(7)
7L
>>> long(3.9)
3L
>>> long('59')
59L
```

12.7 Harfleri saymak

7. bölümde, bir karakter dizisindeki harfleri sayan bir fonksiyon yazmıştık. Bu problemin daha genel bir biçimi karakter dizisindeki her bir harfin ne kadar yer aldığını gösteren bir sıklık grafiğinin (histogram) oluşturulmasıdır.

Bu şekildeki bir sıklık grafiği bir metin dosyasının sıkıştırılmasında yararlı olacaktır. Çünkü farklı harfler farklı sıklıklarla yer alacaktır, böylece sık karşılaşılan harfler için daha kısa kodlar, seyrek harfler için daha uzun kodlar kullanabiliriz.

Sıklık grafiği oluşturmak için sözlükler şık bir yol sunarlar:

```
>>> harf_sayilari = {}
>>> for harf in "Mississippi":
...   harf_sayilari[harf] = harf_sayilari.get(harf, 0) + 1
...
>>> harf_sayilari
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Boş bir sözlükle başlarız. Karakter dizisindeki her bir harf için, sözlükteki o anki sayıyı bulur (eğer yoksa sıfırdır), o sayıyı arttırırız. Sonda, sözlüğümüzde harflerin anahtar, değerlerin de harf sayıları olduğu durumu elde ederiz.

Sıklık grafiğini alfabetik sıraya göre görüntülemek daha çekici olabilir. Bunu items ve sort metotlarıyla yapabiliriz:

```
>>> harfler = harf_sayilari.items()
>>> harfler.sort()
>>> print harfler
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

12.8 Örnek çalışma: Robotlar

Oyun

Bu örnek çalışmada klasik konsol oyunlarından robotlar oyununun bir sürümünü yazacağız.

Robotlar sıralı hamleli, kahraman olarak sizin aptal ancak acımasız robotlara karşı hayatta kalmaya çalıştığınız bir oyun. Her robot her hareket ettiğinizde (her hamlede) size sadece bir kare yaklaşabilir. Eğer sizi yakalarlarsa, ölürsünüz, çarpışırlarsa arkalarında bir robot çöplüğü bırakarak onlar ölür. Diğer robotlar bu çöplüğe çarptığında ölürler.

Temel strateji kendinizi robotlar birbirleriyle veya çöplüklerle çarpışacak şekilde konumlandırmaktır. Oyunu daha oynanabilir kılmak için size ayrıca başka bir konuma ışınlanma yeteneği verilmiştir -- 3 kere güvenli ışınlanma, diğerleri rastgele ışınlanma.

Dünyayı, oyuncuyu ve ana döngüyü ayarlama

Oyuncuyu ekrana yerleştiren ve klavye ile hareket ettiren fonksiyonları içerecek bir şekilde programa başlayalım:

```
#
# robots.py
#
from gasp import *
```

```
SCREEN_WIDTH = 640
SCREEN_HEIGHT = 480
GRID_WIDTH = SCREEN_WIDTH/10 - 1
GRID_HEIGHT = SCREEN_HEIGHT/10 - 1


def place_player():
    x = random.randint(0, GRID_WIDTH)
    y = random.randint(0, GRID_HEIGHT)
    return {'shape': Circle((10*x+5, 10*y+5), 5, filled=True), 'x':
x, 'y': y}


def move_player(player):
    update_when('key_pressed')
    if key_pressed('escape'):
        return True
    elif key_pressed('4'):
        if player['x'] > 0: player['x'] -= 1
    elif key_pressed('7'):
        if player['x'] > 0: player['x'] -= 1
        if player['y'] < GRID_HEIGHT: player['y'] += 1
    elif key_pressed('8'):
        if player['y'] < GRID_HEIGHT: player['y'] += 1
    elif key_pressed('9'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
        if player['y'] < GRID_HEIGHT: player['y'] += 1
    elif key_pressed('6'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
    elif key_pressed('3'):
        if player['x'] < GRID_WIDTH: player['x'] += 1
        if player['y'] > 0: player['y'] -= 1
    elif key_pressed('2'):
        if player['y'] > 0: player['y'] -= 1
    elif key_pressed('1'):
        if player['x'] > 0: player['x'] -= 1
        if player['y'] > 0: player['y'] -= 1
    else:
        return False

    move_to(player['shape'], (10*player['x']+5, 10*player['y']+5))

    return False


def play_game():
    begin_graphics(SCREEN_WIDTH, SCREEN_HEIGHT)
    player = place_player()
```

```
    finished = False
    while not finished:
        finished = move_player(player)
    end_graphics()


if __name__ == '__main__':
    play_game()
```

Kullanıcıyla etkileşim içeren bunun gibi programlar etkileşimi **olaylar (event)** (tuş basma, fare tıklama, vb.) aracılığıyla gerçekleştirir. Bunlara **olay güdümlü programlar**adı verilir.

Ana döngü **olay döngüsü** bu kademede kolaydır:

```
while not finished:
        finished = move_player(player)
```

Olay kotarma işlemi move_player fonksiyonu içinde yapılmaktadır. update_when('key_pressed') bir tuşa basılana kadar bekler, tuşa basıldıktan sonra sonraki cümleye geçer. Çoklu dallandırma cümlesi daha sonra oyunla ilgili tüm tuşları kotarır.

ESC tuşuna basmak move_player'in True döndürmesine, böylece not finished'i yanlışlayıp ana döngüden ve oyundan çıkılmasına neden olur. 4, 7, 8, 9, 6, 3, 2, ve 1 tuşlarının hepsi oyuncunun uygun yönde hareket etmesini sağlar, eğer oyuncu pencere kenarına gelmemişse.

Robot ekleme

Şimdi oyuncu her hareket ettiğinde oyuncuya doğru hareket eden bir robot ekleyelim.

Aşağıdaki place_robot fonksiyonunu place_player ve move_player arasına yerleştirin:

```
def place_robot():
    x = random.randint(0, GRID_WIDTH)
    y = random.randint(0, GRID_HEIGHT)
    return {'shape': Box((10*x, 10*y), 10, 10), 'x': x, 'y': y}
```

move_robot fonksiyonunu hemen move_player'in ardına ekleyin:

```
def move_robot(robot, player):
    if robot['x'] < player['x']: robot['x'] += 1
    elif robot['x'] > player['x']: robot['x'] -= 1

    if robot['y'] < player['y']: robot['y'] += 1
    elif robot['y'] > player['y']: robot['y'] -= 1

    move_to(robot['shape'], (10*robot['x'], 10*robot['y']))
```

Robot ve oyuncuyu bu fonksiyona geçirmemiz gerekir, böylece konumları karşılaştırılıp robot oyuncuya doğru yönlendirilebilir.

Şimdi robot = place_robot() satırını programın ana gövdesinde player = place_player()'ın ardına yerleştirelim, ve move_robot(robot, player) satırını da ana döngünün içerisinde finished = move_player(player) satırının ardına ekleyelim.

Şimdi oyuncuya acımasızca ilerleyen bir robota sahibiz, ama oyuncuyu yakaladığında onunla birlikte aynı yöne ilerler. Yapmamız gereken oyuncu yakalanır yakalanmaz oyunu bitirmektir. Aşağıdaki fonksiyon yakalanmanın olup olmadığına karar verir:

```
def collided(robot, player):
    return player['x'] == robot['x'] and player['y'] == robot['y']
```

Bu fonksiyonu move_player fonksiyonunun hemen altına yerleştirelim. Şimdi play_game fonksiyonunu çarpışmaları denetlemek üzere değiştirelim:

```
def play_game():
    begin_graphics(SCREEN_WIDTH, SCREEN_HEIGHT)
    player = place_player()
    robot = place_robot()
    defeated = False

    while not defeated:
        quit =  move_player(player)
        if quit:
            break
        move_robot(robot, player)
        defeated = collided(robot, player)

    if defeated:
        remove_from_screen(player['shape'])
        remove_from_screen(robot['shape'])
        Text("Yakalandın!", (240, 240), size=32)
        sleep(3)

    end_graphics()
```

Yeni bir değişken yarattık, defeated, bu değişken collided fonksiyonun sonucudur. Ana döngü defeated yanlış olana kadar devam edecektir. ESC tuşuna basmak hala programı sonlandırmaktadır, quit sınaması yapıp doğru ise ana döngüden çıktığımız için. Son olarak ana döngüden hemen sonra defeated kontrolü yapıp eğer doğruysa uygun bir mesajı ekranda görüntülüyoruz.

Daha fazla robot ekleme

Şimdi yapabileceğimi bir çok şey var:

- oyuncunun bir robotun üstünde çıkmaması için güvenli yerleştirilmesi.

- oyuncuya ışınlanma yeteneğinin verilmesi, böylece takipten başka bir konuma kaçabilir.
- daha fazla robot eklenmesi.

Bu görevlerden herhangi biri ilk önce yapılabilir olsa da, biz sonuncuyla daha fazla robot eklemeyle ilgileneceğiz.

İkinci bir robot eklemek için sadece place_robot çağrımını yapan robot2 isminde yeni bir değişken yaratabiliriz. Ancak daha fazla robot eklemek istedikçe ve robotların sayısı arttıkça bu yöntem gereksiz bir yük getirecektir. Daha şık bir yöntem tüm robotları bir listede tutmaktır:

```
def place_robots(numbots):
    robots = []
    for i in range(numbots):
        robots.append(place_robot())
    return robots
```

Şimdi place_robot fonksiyonunu play_game içinde çağırmak yerine, place_robots fonksiyonunu çağırırız, bu bize tüm robotları içeren bir liste döndürecektir:

```
robots = place_robots(2)
```

Birden fazla robotun eklendiği durumda her bir robotun hareketiyle ayrıca ilgilenmemiz gerekecek. Bunu tek bir robot için yapmıştık, şimdi yapmamız gereken tek şey tüm listedeki robotları dolaşmak ve hareket ettirmek:

```
def move_robots(robots, player):
    for robot in robots:
        move_robot(robot, player)
```

move_robots fonksiyonunu move_robot fonksiyonundan sonraya ekliyoruz, ve play_game fonksiyonunu move_robot fonksiyonu yerine move_robots fonksiyonunu çağıracak şekilde düzeltiyoruz.

Şimdi de her bir robotun oyuncuyla çarpışıp çarpışmadığını incelememiz gerekiyor:

```
def check_collisions(robots, player):
    for robot in robots:
        if collided(robot, player):
            return True
    return False
```

Son olarak play_game içerisindeki defeated'i ayarlayan satırı collided yerine check_collisions fonksiyonunu çağıracak şekilde düzeltiriz.

12.9 Sözlük

sözlük:

Anahtar-değer çiftlerinde oluşan, anahtarları değerlere eşleyen veri tipidir. Anahtarlar herhangi bir değiştirilemez tip olabilir, değerler ise herhangi bir tip olabilir.

eşleme tipi:
Anahtar ve ilişkili değerlerin kolleksiyonlarından oluşan veri tipidir. Python'un yerleşik tep eşleme tipi sözlüktür. Sözlükler ilişkili dizi soyut veri tipini gerçekleştirmektedir.

anahtar:
Sözlükte bir değere eşlenen veri öğesidir. Anahtarlar sözlükteki değerlere bakmak, erişmek için kullanılmaktadır.

anahtar-değer çifti:
Sözlükteki öğelerin bir çiftidir. Sözlükteki değerlere anahtarlarla erişilir.

ipucu:
Tekrar hesaplamayı önlemek üzere önceden hesaplanmış değerlerin geçici saklanmasıdır.

olay:
Klavye tuşuna basma, fare tıklaması veya bir başka programdan mesaj gibi sinyallerdir.

olay güdümlü program:
Olayları algılayan, işleyen ve kotaran programlardır. Olayları işlemek için kotarıcılar (handler) vardır.

olay döngüsü:
Olayları bekleyen ve onları işleyen programlama yapısı

taşma:
Nümerik biçimde gösterilemeyecek kadar büyük nümerik sonuçtur.

12.10 Alıştırmalar

1. Komut satırından karakter dizisi okuyan ve alfabedeki harflerin tablosunu girilen karakter dizisindeki bulunma sayısıyla birlikte listeleyen bir program yazınız. Küçük-büyük farkı yok sayılmalıdır. Programın örnek bir çalıştırması aşağıdaki olmalıdır:

```
$ python harf_sayilari.py "ThiS is String with Upper and lower
case Letters."
a  2
c  1
d  1
e  5
g  1
h  2
i  4
l  2
n  2
o  1
p  2
```

```
r  4
s  5
t  5
u  1
w  2
$
```

2. Aşağıdaki yorumlayıcı oturumlarındaki sorgular için sonuçları üretin:

```
1.  >>> d = {'apples': 15, 'bananas': 35, 'grapes': 12}
    >>> d['banana']


2.  >>> d['oranges'] = 20
    >>> len(d)


3.  >>> d.has_key('grapes')


4.  >>> d['pears']


5.  >>> d.get('pears', 0)


6.  >>> fruits = d.keys()
    >>> fruits.sort()
    >>> print fruits


7.  >>> del d['apples']
    >>> d.has_key('apples')
```

Her bir sonucun neden olduğunu da iyi anlamayı unutmayın. Öğrendiklerinizi aşağıda fonksiyon gövdesini doldurmak için kullanın:

```
def add_fruit(inventory, fruit, quantity=0):
    """
    Adds quantity of fruit to inventory.

      >>> new_inventory = {}
      >>> add_fruit(new_inventory, 'strawberries', 10)
      >>> new_inventory.has_key('strawberries')
      True
      >>> new_inventory['strawberries']
      10
      >>> add_fruit(new_inventory, 'strawberries', 25)
      >>> new_inventory['strawberries']
```

```
    """
```

Çözümünüz doctestleri geçmelidir.

3. alice_words.py isminde bir program yazın, programınız alice_in_wonderland.txt dosyasındaki tüm kelimelerin alfabetik listesini her bir kelimenin kaç kere yer aldığıyla birlikte alice_words.txt adındaki metin dosyasına yazsın. Çıktınızın ilk 10 satırı aşağıdakine benzeyecektir:

```
Kelime              Adet
======================
a                   631
a-piece             1
abide               1
able                1
about               94
above               3
absence             1
absurd              2
```

alice kelimesi kitapta kaç kere yer almaktadır?

4. Alice in Wonderland (Alice Harikalar Diyarında)'ki en uzun kelime hangisidir? Kelime kaç karakter içermektedir??
5. DÜnyayı, oyuncuyu, ve ana döngüyü ayarlamak kısmındaki kodu robots.py dosyasına yazın ve çalıştırın. Oyuncuyu ekranda nümerik tuş takımı yardımıyla hareket ettirebiliyor olmanız gerekiyor, ayrıca ESC tuşuyla program çıkabilirsiniz.
6. Dizüstü bilgisayarlar genellikle ayrı bir nümerik tuş takımı içermeyen ve masaüstü bilgisayarlara göre daha küçük bir klavye içerirler. Robotlar programını '4', '7', '8', '9', '6', '3', '2', ve '1' yerine 'a','q','w','e','d','c','x', ve 'z' tuşlarını kullanacak şekilde değiştirin. Böylece tipik bir dizüstü klavyede de çalışacaktır.
7. Add all the necessary code from the Adding a robot section in the places indicated. Make sure the program works and that you now have a robot following around your player.
8. Daha fazla robot ekleme kısmındaki ilgili yerlere eklenecek tüm kodları ekleyin. Programınız çalıştığından ve oyuncunuzu iki robotun izlediğinden emin olun.robots = place_robots(2)'deki argümanı 2'den 4'e değiştirin ve 4 robotun oluşturulduğunu ve istendiği gibi çalıştığını doğrulayın.

13. Sınıflar ve nesneler

13.1 Nesne yönelimli programlama

Python **nesne yönelimli programlama dilidir**, bunun anlamı which **nesne yönelimli programlamanın** (Object Oriented Programming - OOP) desteklediği özellikleri sağlar.

Nesne yönelimli programlama 1960larda ortaya çıkmasına rağmen 1980lerin ortasına kadar yeni yazılım üretmede ana **programlama paradigması** haline gelmedi. Hızlı bir şekilde büyüyen ve karmaşıklaşan yazılım sistemlerini kotarmak ve bu büyük ve karmaşık sistemleri zaman içerisinde daha kolay değiştirmek için geliştirilmiştir.

Şu ana kadar programları **yordamsal (procedural) programlama** paradigmasını kullanarak yazdık. Yordamsal programlamada fonksiyonları veya yordamları yazmaya odaklanma sözkonusudur. Fonksiyonlar/yordamlar veri üzerinde işlem yaparlar. Nesne yönelimli programlamada odak **nesnelerin** yaratılmasındadır, nesneler hem veriyi hem de işlevselliği birlikte barındırır.

13.2 Kullanıcı tanımlı bileşik tipler

Sınıf esas olarak yeni bir **veri tipi** tanımlar. Şu ana kadar kitapta Python'un içinde tanımlı tipleri kullandık, şimdi kendi kullanıcı tanımlı tiplerimizi yaratmaya başlayabiliriz:Nokta.

Matematiksel nokta kavramını düşünelim. İki boyutlu bir uzayda, nokta iki sayıdan (koordinatlar) oluşan ve birlikte tek bir nesne olarak ele alınır. Matematiksel gösterimde, noktalar genellikle parantez içerisinde virgül ile ayrılmış iki sayı şeklinde gösterilir. Örneğin (0,0) başlangıç (origin) noktasını, (x,y) ise başlangıçtan x birim kadar sağdaki ve y birim kadar yukarıdaki bir noktayı gösterir.

Python'da bir noktayı doğal olarak temsil etmenin bir yolu iki nümerik kullanmaktır. Soru, bu iki nümerik değeri nasıl gruplayacağız sorusudur. Hızlı ve kötü bir çözüm bir liste veya tuple kullanmaktır ve bazı uygulamalar için en iyi çözüm olabilir.

Bir alternatif yöntem ise kullanıcı tanımlı bileşik tip tanımlamaktır, bu bileşik tipe **sınıf** adı verilmektedir. Bu yaklaşım biraz çaba gerektirir, ancak avantajları ortaya çıkacaktır.

Bir sınıf tanımı aşağıdaki gibidir:

```
class Nokta:
    pass
```

Sınıf tanımları programın içerisinde herhangi bir yerde bulunabilir, ancak genellikle başlangıçta kullanılırlar (import cümlelerinden sonra). Sınıf tanımı için sözdizimi kuralları bileşik cümleler için olanlarla aynıdır. Bir başlık vardır, class anahtar

kelimesiyle tanımlı, sonrasında sınıfın ismi bulunur ve cümle iki nokta üstüste ile bitirilir.

Yukarıdaki tanımlama Nokta olarak çağrılan bir yeni sınıf yaratır. pass cümlesi etkisizdir; sadece bileşik cümlenin gövdesinde bir şey olması gerektiği için yazılmıştır.

Noktasınıfını yaratarak, Nokta isminde yeni bir tip oluşturmuş olduk. Bu tipin üyeleri, tipin **örnekleri** veya **nesneler** adını almaktadır. Yeni bir örnek yaratma işlemine**örnekleme (instantiation)** denilir. Bir Nokta nesnesini örneklemek için Nokta isminde (tahmin ettiğiniz gibi) bir fonksiyon çağırırız:

```
>>> type(Nokta)
<type 'classobj'>
>>> p = Nokta()
>>> type(p)
<type 'instance'>
```

p değişkeni yeni Nokta nesnesine bir referans olarak atanmıştır. Yeni nesneler yaratan Nokta şeklindeki fonksiyon **yapıcı (constructor)**dır.

13.3 Özellikler

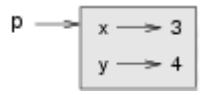
Gerçek dünya nesneleri gibi, nesne örnekleri de biçim ve işlevlere sahiptir. Biçim örnek içinde yer alan veri öğelerinden oluşur.

Bir örneğe nokta işaretini kullanarak yeni veri öğeleri ekleyebiliriz:

```
>>> p.x = 3
>>> p.y = 4
```

Bu sözdizimi, bir modülden değişken seçme işine yarayan, örneğin math.pi veya string.uppercase gibi, sözdizimi ile benzerdir. Hem modüller hem de örnekler kendi isim alanlarını (namespace) oluştururlar, ve bu içerdikleri isimlere, bunlara **özellik** adı verilmektedir, erişmek için kullanılan sözdizimleri aynıdır. Sınıflar için geçerli olan durumda, özellik bir örnekten seçtiğimiz veriyi kastetmektedir.

Aşağıdaki durum diyagramları yukarıdaki atamaların sonuçlarını göstermektedir::



p değişkeni bir Nokta nesnesini referans etmekte, bu nesnede iki özellik içermektedir. Her özellik bir sayıya işaret etmektedir.

Bir özelliğin değerini aynı sözdizimini kullanarak okuyabiliriz:

```
>>> print p.y
4
>>> x = p.x
>>> print x
```

```
3
```

p.x ifadesinin anlamı, p'nin referans ettiği nesneye git ve x'in değerini getirdir. x=p.x ifadesinde, x isimli değişkene p.x değerini atıyoruz. x değişkeni ve x özelliği arasında herhangi bir çakışma yoktur. Nokta gösteriminin amacı hangi değişkeni kullandığımızı belirsizlik yaratmadan tanımlamaktır.

Nokta gösterimini herhangi bir deyimin parçası olarak kullanabilirsiniz, bu yüzden aşağıdaki cümlelerin hepsi geçerlidir:

```
print '(%d, %d)' % (p.x, p.y)
uzaklıkKare = p.x * p.x + p.y * p.y
```

İlk satır (3, 4) çıktısını gösterir; ikinci satır 25 değerini hesaplar.

13.4 **İlkleme (initialization) metotu** ve self

Nokta sınıfımız iki boyutlu bir matematiksel noktayı temsil edeceğine göre, tüm nokta örnekleri x ve y özelliklerine sahip olmalıdır, ancak bu henüz bizim Nokta nesneleri için geçerli değildir.

```
>>> p2 = Nokta()
>>> p2.x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Nokta instance has no attribute 'x'
>>>
```

Bu sorunu çözmek için sınıfımıza bir **ilkleme metotu** ekleriz

```
class Nokta:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

13.5 Parametreler şeklinde örnekler

Bir örneği (burada kastedilen sınıf örneğidir-instance) olağan şekilde parametre olarak geçirebilirsiniz. Örneğin:

```
def noktayi_yaz(p):
    print '(%s, %s)' % (str(p.x), str(p.y))
```

noktayi_yaz bir noktayı argüman olarak alır ve standart bir biçimde ekranda değerini görüntüler. Eğer noktayi_yaz(blank) şeklinde çağrılırsa, çıktı (3, 4) olacaktır.

13.6 Aynılık

Aynı kelimesinin anlamı, üzerinde biraz düşünene kadar yeterince açıktır. Ancak bir süre sonra daha fazlasının olması gerektiğini düşünmeye başlamanız gerekiyor.

Örneğin, Mustafa ve ben aynı arabaya sahibiz dediğinizde, bunun anlamı sizin ve Mustafa'nın arabasının aynı model ve üretim olduğu, ancak farklı iki araç olduğudur. Eğer Mustafa ve ben aynı anneye sahibiz dediğinizde, bunun anlamı ikinizin annesinin aynı kişi olduğudur.

Nesneler hakkında konuşurken, benzer bir belirsizlik sözkonusudur. Örneğin, eğer iki Nokta aynı ise, bunun anlamı aynı veriye (koordinatlar) sahip olmaları mı yoksa ikisinin gerçekten aynı nesne olması mıdır?

İki referansın aynı nesneyi gösterip göstermediğini anlamak için, == işleci kullanılır. Örneğin:

```
>>> p1 = Nokta()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Nokta()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
False
```

p1 ve p2 her ne kadar aynı koordinatlara sahip olsa da, ikisi aynı nesne değildir. Eğer p1'i p2'ye atarsak, o zaman iki değişken aynı nesneyi gösterecektir:

```
>>> p2 = p1
>>> p1 == p2
True
```

Bu tip eşitliğe **yüzeysel eşitlik (shallow equality)** denir, çünkü sadece referansları karşılaştırır, içeriği değil.

Nesnelerin içeriğini karşılaştırmak için -- **derin eşitlik** -- ayni_nokta isminde bir fonksiyon yazabiliriz:

```
def ayni_nokta(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)
```

Eğer şimdi içeriği aynı olan iki farklı nesne yaratırsak, ayni_nokta metotunu kullanarak aynı noktayı temsil edip etmediklerini kontrol edebiliriz.

```
>>> p1 = Nokta()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Nokta()
>>> p2.x = 3
>>> p2.y = 4
```

```
>>> ayni_nokta(p1, p2)
True
```

Elbette, iki değişken de aynı nesneyi gösteriyorsa, her ikisi de hem yüzeysel hem de derin eşitliğe sahiptir.

13.7 Dikdörtgenler

Varsayalım ki dikdörtgeni temsil eden bir sınıf istiyoruz. Soru, bir dikdörtgeni temsil etmek için ne tür bilgiye ihtiyacımız vardır? İşi kolay tutmak için, farzedelim ki dikdörtgen sadece dikey veya yatay, açısız bir şekilde olsun.

Birkaç olasılık mevcuttur: dikdörtgenin merkezini (iki koordinat) ve boyutlarını (genişlik, yükseklik) belirtebiliriz; veya köşelerden birini ve boyutlarını belirtebiliriz; veya iki çapraz köşeyi belirtebiliriz. Geleneksel yöntem sol üst köşeyi ve boyutları belirtmektir.

Tekrar, yeni bir sınıf tanımlarız:

```
class Dikdortgen:
    pass
```

Ve örnekleriz:

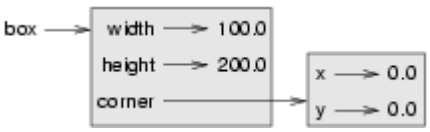
```
box = Dikdortgen()
box.genislik = 100.0
box.yukseklik = 200.0
```

Yukarıdaki kod, iki kayan noktalı özelliğe sahip yeni bir Dikdortgen nesnesi yaratır. Sol üst köşeyi tanımlamak için nesne içerisine nesne gömebiliriz!

```
box.kose = Nokta()
box.kose.x = 0.0;
box.kose.y = 0.0;
```

Nokta işareti düzeni sağlar. box.kose.x ifadesi box değişkeninin gösterdiği nesnenin kose isimli özelliğiyle tanımlı nesnenin x isimli özelliğinin değerini seç anlamına gelir.

Aşağıdaki şekilde bu nesnenin durumunu görebilirsiniz:



13.8 Dönüş değeri olarak örnekler

Fonksiyonlar örnekleri döndürebilir. Örneğin, merkezi_bul bir Dikdortgen argümanı alır ve bu argümanın tanımladığı dikdörtgenin merkezinin koordinatlarını Nokta örneği olarak döndürür:

```
def merkezi_bul(box):
    p = Nokta()
    p.x = box.kose.x + box.genislik/2.0
    p.y = box.kose.y - box.yukseklik/2.0
    return p
```

Bu fonksiyonu çağırmak için box değişkenini argüman olarak geçirip, sonucu bir değişkene atarız::

```
>>> merkez = merkezi_bul(box)
>>> noktayi_yaz(merkez)
(50.0, 100.0)
```

13.9 Nesneler değiştirilebilir

Bir nesnenin durumunu özelliklerinden birine atama yaparak değiştirebiliriz. Örneğin bir dikdörtgenin konumunu değiştirmeden boyutlarını değiştirmek istersek, genislikve yukseklik özelliklerini değiştirebiliriz. :

```
box.genislik = box.genislik + 50
box.yukseklik = box.yukseklik + 100
```

13.10 Kopyalama

İsim takma (aliasing) programın okunabilirliğini zorlaştırabilir, çünkü bir yerde yapılan bir değişiklik başka bir yeri etkileyebilir. Bir nesneye referans veren tüm değişkenleri takip etmek zordur.

Kopyalama işlemi isim takmanın bir alternatifidir. copy modülü copy isminde bir fonksiyon içerir ve bu fonksiyon herhangi bir nesnenin kopyasını oluşturur:

```
>>> import copy
>>> p1 = Nokta()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
False
>>> ayni_nokta(p1, p2)
True
```

copy modülünü içeri aktardıktan sonra, copy fonksiyonunu yeni Noktalar üretmek için kullanabiliriz. p1 ve p2 aynı noktalar değildir, ancak aynı veriyi içerirler.

Nokta gibi basit bir nesneyi, herhangi bir gömülü nesneye sahip olmayan, kopyalamak için copy fonksiyonu yeterlidir. Bu **yüzeysel kopyalama**dır.

Dikdortgen benzeri, içerisinde Nokta nesnesine referans içermektedir, sınıfları kopyalamak için copy bekleneni yapmayacaktır. Sadece Point nesnesinin referansını kopyalayacaktır, böylece eski ve yeni Dikdortgen nesneleri aynı Nokta nesnesini gösterecektir.

Eğer olağan şekilde bir b1 kutu nesnesi oluşturup, copy fonksiyonu ile kopyasını oluşturursak, oluşan durum diyagramı aşağıdaki gibi olacaktır::

Bu kesinlikle bizim istediğimiz sonuç değil. Bu durumda, herhangi bir dikdörtgen nesnesinde çağırdığımız kareyi_genislet fonksiyonu diğer nesneyi etkileyemeyecek, ancak konum_degistir fonksiyonu her iki nesneyi de etkileyecektir! Bu davranış kafa karıştırıcı ve hatalara müsaittir.

Şanslıyız ki, copy modülü deepcopy isminde bir fonksiyon içermektedir. Bu fonksiyon sadece nesneyi değil, ayrıca gömülü tüm nesneleri de kopyalayacaktır. Bu işleme**derin kopyalama** denmesi sizi şaşırtmayacaktır.

```
>>> b2 = copy.deepcopy(b1)
```

Şimdi b1 ve b2 tamamen farklı nesnelerdir.

deepcopy fonksiyonunu kareyi_genislet ve konum_degistir fonksiyonlarını değiştirmek için kullanabiliriz. Böylece bu fonksiyonlar varolan bir Dikdortgeni değiştirmek yerine, aynı konuma sahip fakat yeni boyutlarda yeni bir Dikdortgen oluşturacaktır:

```
def kareyi_genislet(box, dwidth, dheight):
    import copy
    new_box = copy.deepcopy(box)
    new_box.width = new_box.width + dwidth
    new_box.height = new_box.height + dheight
    return new_box
```

13.11 Sözlük

sınıf:
Kullanıcı tanımlı bileşik tiptir. Bir sınıf ayrıca, kendisinden oluşturulan örnekler için nesnelerin bir şablonu gibi de düşünülebilir.

örnekleme:
Bir sınıfın bir örneğini yaratma.

örnek:
Bir sınıfa ait olan bir nesne.

nesne:
Gerçek dünyadaki bir şeyi veya kavramı modellemek için kullanılan bileşik veri tipidir.

yapıcı:
Yeni nesneler yaratmak için kullanılan bir metot.

özellik:
Bir örneği oluşturan isimlendirilmiş veri öğelerinden biri.

yüzeysel eşitlik:
Referansların eşitliği, veya iki referansın aynı nesneyi işaret etmesidir.

derin eşitlik:
Değerlerin eşitliği, veya iki referansın aynı değerlere sahip nesneleri işaret etmesidir.

yüzeysel kopyalama:
Bir nesnenin içeriklerini, içerdiği gömülü nesne referansları da dahil olmak üzere, kopyalamak. copy modülündeki copy fonksiyonunda gerçekleştirilmiştir.

derin kopyalama:
Bir nesnenin içeriklerini, içeridği gömülü nesnelerin içerikleri de olmak üzere, kopyalanmasıdır. copy modülündeki deepcopy fonksiyonunda gerçekleştirilmiştir.

13.12 Alıştırmalar

1. Bir Nokta nesnesi yaratıp ekrana yazdırın, daha sonra id kullanarak nesnenin tekil belirtecini ekranda gösterin. Onaltılık biçimi onluk biçime çevirip ikisinin uyuştuğunu doğrulayın.
2. 5. bölümdeki uzaklik fonksiyonunu dört sayı yerine iki Nokta parametresi alacak şekilde tekrar yazın.
3. kareyi_yurut isminde ve dx ile dy isminde iki parametre alan fonksiyon yazınız. Bu fonksiyon konum değiştirmeyi, dikdörtgenin köşesinin ilgili özelliğine bu aldığı parametreleri (x'e dx, y'e dy) ekleyerek gerçekleştirmelidir.
4. konum_degistir fonksiyonunu varolan dikdörtgeni değiştirmek yerine yeni bir Rectangle nesnesi oluşturup döndürecek şekilde tekrar yazın.

A. Ubuntu'yu Python geliştirme için yapılandırma

Aşağıdaki yönergeler Ubuntu 8.04 ("Hardy Heron") işletim sistemini bu kitap için kullanıma hazırlamak içindir. Geliştirme ve sınama için Ubuntu GNU/Linux kullanmam nedeniyle burada yapılandırmasını anlatacağım tek sistem olacaktır.

Yazılım özgürlüğü ve açık işbirliği ruhu adına, eğer başka bir işletim sistemi için buna benzer bir ek hazırlamak isterseniz benimle bağlantıya geçin lütfen. O eki Açık Kitap Proje sitesine koymaktan veya bağlantı vermekten mutlu olurum, bağlantılı kullanıcı geri bildirimlerini size sağlamam koşuluyla.

Teşekkürler!

Jeffrey Elkner

Arlington Public Schools Arlington, Virginia

A.1 Vim

Vim Python geliştirmesi için çok verimli bir şekilde kullanılabilir, ancak Ubuntu sadece vim-tiny paketiyle beraber geldiği için sözdizimi renklendirme veya otomatik girintilemeyi desteklemiyor.

Vim'i kullanabilmek için, önce aşağıdaki şekilde kurmalıyız:

1. Komut satırından aşağıdaki komutu çalıştırın:

```
sudo apt-get install vim-gnome
```

2. Kullanıcı ev dizininizde .vimrc isimli dosyayı aşağıdakileri içerecek şekilde oluşturun:

```
syntax enable
filetype indent on
set et
set sw=4
set smarttab
map <f2> :w\|!python %<cr>
```

.py uzantılı bir dosyayı vim ile düzenlemek istediğinizde sözdizimi renklendirme ve otomatik girintileme özelliklerine artık sahip olmanız gerekiyor. <F2> tuşuna bastığınızda programınızın çalışması ve programın çalışması bittikten sonra sizi tekrar düzenleyiciye döndürmesi gerekiyor.

Vim'i öğrenmek için aşağıdaki komutu bir komut satırında çalıştırın:

```
vimtutor
```

A.2 GASP

Bir çok durum çalışmaları GASP (Graphics API for Students for Python)'ı kullanıyor. Bu ayrıca kurulması gereken ek bir kütüphanedir.

GASP'ı kurmak için aşağıdakileri yapmanız gerekir:

1. Mathew Gallagher'ın özel paket arşivini apt kaynakları arasında ekleyin:
   - System -> Administration -> Software Sources yolunu izleyerek paket yöneticisini açın
   - Third-Party Software sekmesini açın
   - + Add düğmesine basın
   - aşağıdaki kısmı APT line: metin kutusuna girin:
     `deb http://ppa.launchpad.net/mattva01/ubuntu hardy main restricted universe multiverse`

- Close düğmesine tıklayın
- The information about available software is out-of-date mesaj kutusunda Reload seçeneğini seçin
- Software Sources penceresini Close düğmesiyle kapatın

2. GASP'ı aşağıdaki komutu komut satırına girerek kurun:

```
sudo apt-get install python-gasp
```

A.3 $HOME ortamı

Aşağıdaki kullanıcı dizininizde kendi Python kütüphane ve çalıştırılabilir betikleriniz için kullanışlı bir ortam hazırlar:

1. Komut satırında kullanıcı dizininde bin ve lib/python alt dizinlerini aşağıdaki komutlar ile oluşturun:

```
mkdir bin lib
mkdir lib/python
```

2. Aşağıdaki satırları kullanıcı dizininizdeki .bashrc dosyasının sonuna ekleyin:

```
PATH=$HOME/bin:$PATH
PYTHONPATH=$HOME/lib/python
EDITOR=vim

export PATH PYTHONPATH EDITOR
```

   Bu tercih ettiğiniz metin düzenleyici olarak Vim'i ayarlar, kendi lib/python alt dizininizdeki kütüphaneleri Python yoluna ekler ve bin alt dizinini çalıştırılabilir betikler için bir yer olarak ekler.

A.4 Bir Python betiğini heryerden çalıştırılabilir ve yürütülebilir yapma

Unix tabanlı sistemlerde, Python betikleri aşağıdaki işlemlerle çalıştırılabilir hale getirilirler:

1. Aşağıdaki satırı betiğin ilk satırı olarak ekleyin:

```
#!/usr/bin/env python
```

2. Unix komut satırında, aşağıdaki komutu myscript.py betiğini çalıştırılabilir yapmak için çalıştırın:

```
chmod +x myscript.py
```

3. myscript.py betiğini bin dizinine taşıyın, böylece heryerden çalıştırılabilir hale gelecektir.

GNU Free Documentation License Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant

Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms

defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License

or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.