Python Tutorial

Yayım 3.13.0

Guido van Rossum and the Python development team

Kasım 15, 2024

Python Software Foundation
Email: docs@python.org

İçindekiler

1 İştahınızı Kabartma **3**

2 Python Yorumlayıcısını Kullanma **5**
2.1 Yorumlayıcıyı Çağırma . . . 5
2.1.1 Değişken Geçirme . . . 6
2.1.2 Etkileşimli Mod . . . 6
2.2 Yorumlayıcı ve Çevresi . . . 6
2.2.1 Kaynak Kodu Şeması . . . 6

3 Python'a Resmi Olmayan Bir Giriş **9**
3.1 Python'ı Hesap Makinesi Olarak Kullanmak . . . 9
3.1.1 Sayılar . . . 9
3.1.2 Metin . . . 11
3.1.3 Listeler . . . 15
3.2 Programlamaya Doğru İlk Adımlar . . . 16

4 Daha Fazla Kontrol Akışı Aracı **19**
4.1 `if` İfadeleri . . . 19
4.2 `for` İfadeleri . . . 19
4.3 `range()` Fonksiyonu . . . 20
4.4 `break` and `continue` Statements . . . 21
4.5 `else` Clauses on Loops . . . 22
4.6 `pass` İfadeleri . . . 22
4.7 `pass` İfadeleri . . . 23
4.8 Fonksiyonların Tanımlanması . . . 25
4.9 İşlev Tanımlama hakkında daha fazla bilgi . . . 27
4.9.1 Varsayılan Değişken Değerleri . . . 27
4.9.2 Anahtar Kelime Değişkenleri . . . 28
4.9.3 Özel parametreler . . . 29
4.9.4 Keyfi Argüman Listeleri . . . 32
4.9.5 Argüman Listelerini Açma . . . 32
4.9.6 Lambda İfadeleri . . . 33
4.9.7 Dokümantasyon Stringler'i . . . 33
4.9.8 Fonksiyon Ek Açıklamaları . . . 34
4.10 Intermezzo: Kodlama Stili . . . 34

5 Veri Yapıları **37**
5.1 Listeler Üzerine . . . 37
5.1.1 Listeleri Yığın Olarak Kullanma . . . 38
5.1.2 Listeleri Kuyruk Olarak Kullanma . . . 39
5.1.3 Liste Kavramaları . . . 39

5.1.4 İç İçe Liste Kavramaları . . . 40
5.2 del ifadesi . . . 41
5.3 Veri Grupları ve Diziler . . . 42
5.4 Kümeler . . . 43
5.5 Sözlükler . . . 43
5.6 Döngü Teknikleri . . . 44
5.7 Koşullar Üzerine . . . 46
5.8 Diziler ile Diğer Veri Tiplerinin Karşılaştırılması . . . 46

6 Modüller **49**
6.1 Modüller hakkında daha fazla . . . 50
6.1.1 Modülleri komut dosyası olarak yürütme . . . 51
6.1.2 Modül Arama Yolu . . . 51
6.1.3 "Derlenmiş" Python dosyaları . . . 52
6.2 Standart modüller . . . 52
6.3 dir() Fonksiyonu . . . 53
6.4 Paketler . . . 54
6.4.1 Bir Paketten * İçe Aktarma . . . 55
6.4.2 Paket İçi Referanslar . . . 56
6.4.3 Birden Çok Dizindeki Paketler . . . 57

7 Girdi ve Çıktı **59**
7.1 Güzel Çıktı Biçimlendirmesi . . . 59
7.1.1 Biçimlendirilmiş Dize Değişmezleri . . . 60
7.1.2 String format() Metodu . . . 61
7.1.3 Manuel Dize Biçimlendirmesi . . . 62
7.1.4 Eski dize biçimlendirmesi . . . 63
7.2 Dosyaları Okuma ve Yazma . . . 63
7.2.1 Dosya Nesnelerinin Metotları . . . 64
7.2.2 Yapılandırılmış verileri json ile kaydetme . . . 65

8 Hatalar ve Özel Durumlar **67**
8.1 Söz Dizimi Hataları . . . 67
8.2 Özel Durumlar . . . 67
8.3 Özel Durumları İşleme . . . 68
8.4 Hata Yükseltme . . . 70
8.5 İstisna Zincirleme . . . 71
8.6 Kullanıcı Tanımlı İstisnalar . . . 72
8.7 Temizleme Eylemlerini Tanımlama . . . 72
8.8 Önceden Tanımlanmış Temizleme Eylemleri . . . 74
8.9 Birden Fazla Alakasız İstisna Oluşturma ve İşleme . . . 74
8.10 İstisnaları Notlarla Zenginleştirme . . . 76

9 Sınıflar **79**
9.1 İsim ve Nesneler Hakkında Birkaç Şey . . . 79
9.2 Python Etki Alanları ve Ad Alanları . . . 80
9.2.1 Kapsamlar ve Ad Alanları Örneği . . . 81
9.3 Sınıflara İlk Bakış . . . 82
9.3.1 Sınıf Tanımlama Söz Dizimi . . . 82
9.3.2 Sınıf Nesneleri . . . 82
9.3.3 Örnek Nesneleri . . . 83
9.3.4 Metot Nesneleri . . . 83
9.3.5 Sınıf ve Örnek Değişkenleri . . . 84
9.4 Rastgele Açıklamalar . . . 85
9.5 Kalıtım . . . 86
9.5.1 Çoklu Kalıtım . . . 87
9.6 Özel Değişkenler . . . 87
9.7 Oranlar ve Bitişler . . . 88
9.8 Yineleyiciler . . . 89

9.9 Üreteçler . . . 90
9.10 Üreteç İfadeleri . . . 91

10 Standart Kütüphanenin Özeti **93**
10.1 İşletim Sistemi Arayüzü . . . 93
10.2 Dosya Joker Karakterleri . . . 94
10.3 Komut Satırı Argümanları . . . 94
10.4 Hata Çıktısının Yeniden Yönlendirilmesi ve Programın Sonlandırılması . . . 94
10.5 String Örüntü Eşlemesi . . . 94
10.6 Matematik . . . 95
10.7 İnternet Erişimi . . . 95
10.8 Tarihler ve Saatler . . . 96
10.9 Veri Sıkıştırma . . . 96
10.10 Performans Ölçümü . . . 97
10.11 Kalite Kontrolü . . . 97
10.12 Bataryalar Dahildir . . . 98

11 Standart Kütüphanenin Kısa Özeti — Bölüm II **99**
11.1 Çıktı Biçimlendirmesi . . . 99
11.2 Şablonlamak . . . 100
11.3 İkili Veri Kaydı Düzenleriyle Çalışma . . . 101
11.4 Çoklu iş parçacığı . . . 101
11.5 Günlükleme . . . 102
11.6 Zayıf Başvurular . . . 103
11.7 Listelerle Çalışma Araçları . . . 103
11.8 Decimal Floating-Point Arithmetic . . . 104

12 Sanal Ortamlar ve Paketler **107**
12.1 Tanıtım . . . 107
12.2 Sanal Ortamlar Oluşturma . . . 107
12.3 Paketleri pip ile Yönetme . . . 108

13 Sırada Ne Var? **111**

14 Etkileşimli Girdi Düzenleme ve Geçmiş İkame **113**
14.1 Tab Tamamlama ve Geçmiş Düzenleme . . . 113
14.2 Etkileşimli Yorumlayıcıya Alternatifler . . . 113

15 Floating-Point Arithmetic: Issues and Limitations **115**
15.1 Temsil Hatası . . . 118

16 Ek Bölüm **121**
16.1 Etkileşimli Mod . . . 121
16.1.1 Hata İşleme . . . 121
16.1.2 Yürütülebilir Python Komut Dosyaları . . . 121
16.1.3 Etkileşimli Başlangıç Dosyası . . . 122
16.1.4 Özelliştirme Modülleri . . . 122

A Sözlük **123**

B Bu dokümanlar hakkında **141**
B.1 Python Dokümantasyonuna Katkıda Bulunanlar . . . 141

C Tarihçe ve Lisans **143**
C.1 Yazılımın tarihçesi . . . 143
C.2 Python'a erişmek veya başka bir şekilde kullanmak için şartlar ve koşullar . . . 144
C.2.1 PYTHON İÇİN PSF LİSANS ANLAŞMASI 3.13.0 . . . 144
C.2.2 PYTHON 2.0 İÇİN BEOPEN.COM LİSANS SÖZLEŞMESİ . . . 145
C.2.3 PYTHON 1.6.1 İÇİN CNRI LİSANS ANLAŞMASI . . . 145
C.2.4 0.9.0 ARASI 1.2 PYTHON İÇİN CWI LİSANS SÖZLEŞMESİ . . . 146

C.2.5 PYTHON 3.13.0 BELGELERİNDEKİ KOD İÇİN SIFIR MADDE BSD LİSANSI 147
C.3 Tüzel Yazılımlar için Lisanslar ve Onaylar 147
C.3.1 Mersenne Twister'ı 147
C.3.2 Soketler 148
C.3.3 Asenkron soket hizmetleri 149
C.3.4 Çerez yönetimi 149
C.3.5 Çalıştırma izleme 150
C.3.6 UUencode ve UUdecode fonksiyonları 150
C.3.7 XML Uzaktan Yordam Çağrıları 151
C.3.8 test_epoll 151
C.3.9 kqueue seçin 152
C.3.10 SipHash24 152
C.3.11 strtod ve dtoa 153
C.3.12 OpenSSL 153
C.3.13 expat 156
C.3.14 libffi 157
C.3.15 zlib 157
C.3.16 cfuhash 158
C.3.17 libmpdec 158
C.3.18 W3C C14N test paketi 159
C.3.19 mimalloc 160
C.3.20 asyncio 160
C.3.21 Global Unbounded Sequences (GUS) 160

D Telif Hakkı **163**

Dizin **165**

Python öğrenmesi kolay, güçlü bir yazılım dilidir. Verimli üst düzey veri yapılarına ve nesne yönelimli programlamaya basit ama etkili bir yaklaşıma sahiptir. Python'un zarif sözdizimi ve dinamik yazımı, yorumlanmış doğasıyla birlikte, onu çoğu platformda birçok alanda komut dosyası oluşturma ve hızlı uygulama geliştirme için ideal bir dil haline getirir.

Python yorumlayıcısı ve kapsamlı standart kütüphane, Python web sitesinde, https://www.python.org/ tüm büyük platformlar için kaynak veya ikili biçimde ücretsiz olarak mevcuttur ve ücretsiz olarak dağıtılabilir. Aynı site ayrıca birçok ücretsiz üçüncü taraf Python modülü, programı ve aracının dağıtımlarını ve bunlara yönelik yönlendirmeleri ve ek belgeleri içerir.

Python yorumlayıcısı, C veya C++'da (veya C'den çağrılabilen diğer dillerde) uygulanan yeni işlevler ve veri türleri ile kolayca genişletilebilir. Python, özelleştirilebilir uygulamalar için bir uzantı dili olarak da kullanılabilir.

Bu öğretici, okuyucuyu Python dilinin ve sisteminin temel kavramlarını ve özelliklerini gayriresmi olarak tanıtır. Uygulamalı deneyim için kullanışlı bir Python yorumlayıcıya sahip olmaya yardımcı olur, ancak tüm örnekler bağımsızdır, böylece öğretici de çevrimdışı olarak okunabilir.

Standart nesnelerin ve modüllerin açıklaması için library-index 'e bakınız. reference-index dilin daha resmi bir tanımını verir. Uzantıları C veya C++'ta yazmak için extending-index ve c-api-index 'i okuyun. Python'ı derinlemesine kapsayan birkaç kitap da vardır.

Bu öğretici kapsamlı olmaya ve her bir özelliği, hatta yaygın olarak kullanılan her özelliği bile kapsamaya çalışmaz. Bunun yerine, Python'un en dikkat çekici özelliklerinin çoğunu sunar ve size dilin tarzı hakkında iyi bir fikir verecektir. Okuduktan sonra, Python modüllerini ve programlarını okuyabilecek ve yazabileceksiniz ve library-index bölümünde açıklanan çeşitli Python kütüphanesi modülleri hakkında daha fazla bilgi edinmeye hazır olacaksınız.

Ayrıca *Sözlük* de göz atmaya değer.

BÖLÜM 1

İştahınızı Kabartma

Bilgisayarlarda çok fazla iş yaparsanız, sonunda otomatikleştirmek istediğiniz bazı görevler olduğunu bulursunuz. Örneğin, çok sayıda metin dosyası üzerinde arama ve değiştirme gerçekleştirmek veya bir grup fotoğraf dosyasını karmaşık bir şekilde yeniden adlandırmak ve yeniden düzenlemek isteyebilirsiniz. Belki küçük bir özel veritabanı, özel bir GUI uygulaması veya basit bir oyun yazmak istersiniz.

Profesyonel bir yazılım geliştiricisiyseniz, birçok C/C++/Java kütüphanesiyle çalışmanız gerekebilir, ancak normal yazma/derleme/test etme/yeniden derleme döngüsünü çok yavaş buluyorsunuz. Belki de böyle bir kütüphane için bir test paketi yazıyorsunuz ve test kodunu yazmayı sıkıcı bir görev olarak buluyorsunuz. Veya eklenti dili kullanabilecek bir program yazdınız ve uygulamanız için yepyeni bir dil tasarlamak ve uygulamak istemiyorsunuz.

Python tam size göre bir dil.

Bu görevlerden bazıları için bir Unix kabuk komut dosyası veya Windows toplu iş dosyaları yazabilirsiniz, ancak kabuk komut dosyaları, dosyaların etrafında hareket etmek ve metin verilerini değiştirmekte en iyisi olmakla beraber GUI uygulamaları veya oyun yapımı için pek öyle olmayabilir. Bir C/C++/Java programı yazabilirsiniz, ancak taslak programını almak bile çok zaman alabilir. Python'un kullanımı daha kolaydır, Windows, macOS ve Unix işletim sistemlerinde kullanılabilir ve işi daha hızlı bir şekilde halletmenize yardımcı olur.

Python kolay olmakla birlikte, kabuk komut dosyalarının veya toplu iş dosyalarının sunabileceğinden çok daha fazla yapı ve büyük programlar için destek sunan gerçek bir programlama dilidir. Öte yandan, Python C'den çok daha fazla hata denetimi sunar ve *çok üst düzey bir dil (very-high-level language)* olarak, esnek diziler ve sözlükler gibi yerleşik üst düzey veri türlerine sahiptir. Python daha genel veri türleri nedeniyle, Awk ve hatta Perl'den çok daha büyük bir kullanım alanına uygulanabilir, ancak Python'da birçok şey en az bu dillerdeki kadar kolaydır.

Python, programınızı diğer Python programlarında yeniden kullanılabilecek modüllere bölmenizi sağlar. Python, programlayı öğrenmeye başlarken veya programlarınızda temel olarak kullanabileceğiniz geniş bir standart modül koleksiyonuyla birlikte gelir. Bu modüllerden bazıları dosya giriş/çıkışı (I/O), sistem çağrıları, soketler ve hatta Tk gibi GUI araç setlerini içerir.

Python, derleme ve bağlama gerekmediğinden program geliştirme sırasında size önemli ölçüde zaman kazandırabilecek yorumlanmış bir dildir. Yorumlayıcı etkileşimli olarak kullanılabilir, bu da dilin özellikleriyle deneme yapmayı, atma programları yazmayı veya aşağıdan yukarıya program geliştirme sırasında işlevleri test etmeyi kolaylaştırır. Ayrıca kullanışlı bir masa üstühesap makinesidir.

Python, programların kompakt ve okunabilir bir şekilde yazılmasını sağlar. Python'da yazılan programlar genellikle çeşitli nedenlerden dolayı aynı görevi gören C, C++ veya Java programlarına göre çok daha kısadır:

- üst düzey veri türleri karmaşık işlemleri tek bir deyimde ifade etmenizi sağlar;
- ifade gruplandırması, başlangıç ve bitişe koyulan köşeli ayraçlar yerine girintileme kullanılarak yapılır;

- değişken veya bağımsız değişken bildirimleri gerekli değildir.

Python *genişletilebilir*: C'de programlamayı biliyorsanız, kritik işlemleri maksimum hızda gerçekleştirmek veya Python programlarını yalnızca ikili biçimde (satıcıya özgü grafik kitaplığı gibi) kullanılabilen kütüphanelere bağlamak için yorumlayıcıya yeni bir yerleşik işlev veya modül eklemek kolaydır. Gerçekten bağlandıktan sonra, Python yorumlayıcısını C ile yazılmış bir uygulamaya bağlayabilir ve bu uygulama için bir uzantı veya komut dili olarak kullanabilirsiniz.

Bu arada, dil adını BBC şovu "Monty Python'un Uçan Sirki"nden almıştır ve sürüngenlerle hiçbir ilgisi yoktur. Belgelerde Monty Python skeçlerine atıfta bulunmaya sadece izin verilmez, aynı zamanda teşvik edilir!

Artık hepiniz Python için heyecanlı olduğunuza göre, biraz daha ayrıntılı olarak incelemek isteyeceksiniz. Bir dili öğrenmenin en iyi yolu kullanmak olduğundan, öğretici sizi okurken Python yorumlayıcısıyla oynamaya davet eder.

Bir sonraki bölümde, yorumlayıcıyı kullanma mekaniği açıklanmıştır. Bu oldukça sıradan bir bilgidir, ancak daha sonra gösterilen örnekleri denemek için gereklidir.

Öğreticinin geri kalanı, basit ifadeler, ifadeler ve veri türlerinden başlayarak, işlevler ve modüller aracılığıyla ve son olarak istisnalar ve kullanıcı tanımlı sınıflar gibi gelişmiş kavramlara dokunarak örnekler aracılığıyla Python dilinin ve sisteminin çeşitli özelliklerini tanıtır.

BÖLÜM 2

Python Yorumlayıcısını Kullanma

2.1 Yorumlayıcıyı Çağırma

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

kabuğa.[1] Yorumlayıcının bulunduğu dizinin seçimi bir yükleme seçeneği olduğundan, Python dizini başka bir yerde de olabilir; yerel Python gurunuza veya sistem yöneticinize danışın. (Örneğin, `/usr/local/python` popüler bir alternatif konumdur.)

On Windows machines where you have installed Python from the Microsoft Store, the `python3.13` command will be available. If you have the py.exe launcher installed, you can use the `py` command. See setting-envvars for other ways to launch Python.

Dosya sonu karakteri (Unix'te `Control-D`, Windows'ta `Control-Z`) yazılması, yorumlayıcının sıfır durumuyla (zero exit status) sonlanmasına neden olur. Bu işe yaramazsa, aşağıdaki komutu yazarak yorumlayıcıdan çıkabilirsiniz: `quit()`.

Yorumlayıcının satır düzenleme özellikleri arasında etkileşimli düzenleme, geçmiş değiştirme ve GNU Readline kütüphanesini destekleyen sistemlerde kod tamamlama bulunur. Komut satırı düzenlemenin desteklenip desteklenmediğini görmek için belki de en hızlı denetim, elde ettiğiniz ilk Python istemine `Control-P` yazmaktır. Bip sesi çıkarsa komut satırı düzenlemeniz vardır; Tuşların tanıtımı için *Etkileşimli Girdi Düzenleme ve Geçmiş İkame* 'e göz atabilirsiniz. Hiçbir şey görünmüyorsa veya `^P` yankılanıyorsa, komut satırı düzenlemesi kullanılamaz; yalnızca geçerli satırdaki karakterleri kaldırmak için geri alabiliyorsunuz.

Yorumlayıcı bir şekilde Unix kabuğu gibi çalışır: bir tty aygıtına bağlı standart girdi ile çağrıldığında, komutları etkileşimli olarak okur ve yürütür; bir dosya adı argümanıyla veya standart girdi olarak bir dosyayla çağrıldığında, o dosyadan bir *komut dosyası* okur ve yürütür.

Yorumlayıcıyı başlatmanın ikinci bir yolu, kabuğun `-c` seçeneğine benzer şekilde *command* içindeki deyimleri çalıştıran `python -c command [arg] ...` komutudur. Python deyimleri genellikle boşluk veya kabuğa özel başka karakterler içerdiğinden, genellikle *command* ifadesinin tamamının alıntılanması tavsiye edilir.

Bazı Python modülleri komut dosyası olarak da yararlıdır. Bunlar, eğer tam adını komut satırına yazarsanız *modül* için kaynak dosyası `python -m *modül* [argüman] ...` kullanılarak çağrılabilir.

[1] Unix'te, Python 3.x yorumlayıcısı varsayılan olarak `python` adlı yürütülebilir dosyayla yüklenmez, böylece aynı anda yüklenen bir Python 2.x yürütülebilir dosyasıyla çakışmaz.

Bir komut dosyası kullanıldığında, bazen komut dosyasını çalıştırabilmek ve daha sonra etkileşimli moda girebilmek yararlıdır. Bu komut dosyasından önce `-i` geçirilerek yapılabilir.

Tüm komut satırı seçenekleri using-on-general bölümünde açıklanmıştır.

2.1.1 Değişken Geçirme

Yorumlayıcı tarafından bilindiğinde, komut dosyası adı ve bundan sonraki ek argüman dizelerin listesine dönüştürülür ve `sys` modülündeki `argv` değişkenine atanır. Bu listeye `import sys` öğesini yürüterek erişebilirsiniz. Listenin uzunluğu en az birdir; komut dosyası ve argüman verilmediğinde `sys.argv[0]` boş bir dizedir. Komut dosyası adı `'-'` (standart giriş anlamına gelir) olarak verildiğinde, `sys.argv[0] '-'` olarak ayarlanır. `-c` *komut* kullanıldığında, `sys.argv[0]`, `-c` olarak ayarlanır. `-m` *modül* kullanıldığında, `sys.argv[0]` bulunan modülün tam adına ayarlanır. `-c` *komut* veya `-m` *modül* 'den sonra bulunan seçenekler, Python yorumlayıcısının seçenek işlemesi tarafından tüketilmez, ancak komut veya modülün işlemesi için `sys.argv` içinde bırakılır.

2.1.2 Etkileşimli Mod

Komutlar bir tty'den okunduğunda, yorumlayıcının *etkileşimli modda* olduğu söylenir. Bu modda, genellikle üç büyük işaret olan *öncelikli bilgi istemi* ile bir sonraki komutu ister (`>>>`); devam satırları için *ikincil istem* ile sorar, varsayılan olarak üç nokta (`...`). Yorumlayıcı, ilk istemi yazdırmadan önce sürüm numarasını ve telif hakkı bildirimini belirten bir karşılama iletisi yazdırır:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Çok satırlı bir yapıya girilirken devamlılık satırları gereklidir. Örnek olarak, `if` ifadesine bir göz atın:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Etkileşimli mod hakkında daha fazlası için bkz. *Etkileşimli Mod*.

2.2 Yorumlayıcı ve Çevresi

2.2.1 Kaynak Kodu Şeması

Varsayılan olarak, Python kaynak dosyaları UTF-8'de kodlanmış olarak kabul edilir. Bu kodlamada, dünyadaki çoğu dilin karakterleri dize değişmezlerinde, tanımlayıcılarda ve yorumlarda aynı anda kullanılabilir — standart kütüphane tanımlayıcılar için yalnızca ASCII karakterleri kullansa da, herhangi bir taşınabilir kodun izlemesi gereken bir kuraldır. Tüm bu karakterleri düzgün görüntülemek için, düzenleyicinizin dosyanın UTF-8 olduğunu tanıması ve dosyadaki tüm karakterleri destekleyen bir yazı tipi kullanması gerekir.

Varsayılan dil şeması dışında bir şema bildirmek için, dosyanın *ilk* satırı olarak özel bir yorum satırı eklenmelidir. Sözdizimi aşağıdaki gibidir:

```
# -*- coding: encoding -*-
```

burada *kodlama*, Python tarafından desteklenen geçerli `codec` bileşenlerinden biridir.

Örneğin, Windows-1252 şemasının kullanılacağını bildirmek için, kaynak kod dosyanızın ilk satırı şu olmalıdır:

```
# -*- coding: cp1252 -*-
```

İlk satır kuralının bir istisnası, kaynak kodun *UNIX "shebang" line* satırı ile başlamasıdır. Bu durumda, şema bildirimi dosyanın ikinci satırı olarak eklenmelidir. Örneğin:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

BÖLÜM 3

Python'a Resmi Olmayan Bir Giriş

İlerleyen örneklerde; giriş ve çıkış, bilgi istemlerinin olup olmamasına göre ayırt edilir (»> ve …): örneği tekrarlamak için bilgi isteminden sonra her şeyi yazmalısınız, istem göründüğünde bir bilgi istemi ile başlamayan satırlar yorumlayıcıdan çıkar. Bir örnekte tek başına bir satırdaki ikincil istemin boş bir satır yazmanız gerektiği anlamına geldiğini unutmayın; bu, çok satırlı bir komutu sonlandırmak için kullanılır.

Bu kılavuzdaki örneklerin çoğu, etkileşimli komut isteminde girilenler dahil, yorumlar içerir. Python'da yorumlar, # hash karakteriyle başlar ve fiziksel satırın sonuna kadar uzanır. Bir satırın başında veya boşluk veya kodun ardından bir yorum görünebilir, ancak bir dize sabiti içinde değil. Bir dize sabiti içindeki bir hash karakteri, yalnızca bir hash karakterdir. Yorumlar kodu netleştirmek için olduğundan ve Python tarafından yorumlanmadığından örnekler yazarken atlanabilirler.

Bazı örnekler:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Python'ı Hesap Makinesi Olarak Kullanmak

Bazı basit Python komutlarını deneyelim. Yorumlayıcıyı başlatın ve >>> birincil istemini bekleyin. (Uzun sürmemelidir.)

3.1.1 Sayılar

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, –, * and / can be used to perform arithmetic; parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> 8 / 5  # division always returns a floating-point number
1.6
```

Tam sayıların (örneğin `2`, `4`, `20`) türü `int` olup, kesirli kısmı olanlar (örneğin `5.0`, `1.6`) `float` türüne sahiptir. Sayısal türler hakkında sonrasında daha fazlasını göreceğiz.

Bölme (`/`) her zaman bir ondalıklı sayı döndürür. *floor division* yapmak ve bir tam sayı sonucu almak için `//` operatörünü kullanabilirsiniz; kalanı hesaplamak içinse `%` operatörünü kullanabilirsiniz:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

Python ile üslü sayıları hesaplamak için `**` operatörünü kullanmak mümkündür[1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

Bir değişkene değer atamak için eşittir işareti (=) kullanılır. Daha sonra, bir sonraki etkileşimli komut isteminden önce hiçbir sonuç görüntülenmez:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Bir değişken "tanımlı" değilse (bir değer atanmamışsa), onu kullanmaya çalışmak size bir hata verecektir:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Ondalıklı sayı için tam destek var; karışık türde işlenenlere sahip operatörler, tam sayı işlenenini ondalıklı sayıya dönüştürür:

```
>>> 4 * 3.75 - 1
14.0
```

Etkileşimli modda, son yazdırılan ifade _ değişkenine atanır. Bu, Python'ı bir masa hesap makinesi olarak kullandığınızda, hesaplamalara devam etmenin biraz daha kolay olduğu anlamına gelir, örneğin:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

(sonraki sayfaya devam)

[1] `**`, `-` 'den daha yüksek önceliğe sahip olduğundan, `-3**2`, `-(3**2)` olarak yorumlanacak ve dolayısıyla `-9` ile sonuçlanacaktır. Bundan kaçınmak ve `9` elde etmek için `(-3)**2` kullanabilirsiniz.

(önceki sayfadan devam)

```
>>> round(_, 2)
113.06
```

Bu değişken, kullanıcı tarafından salt okunur olarak ele alınmalıdır. Açıkça ona bir değer atamayın — sihirli davranışları olan bu gömülü değişkeni maskeleyen aynı ada sahip bağımsız bir yerel değişken yaratırsınız.

`int` ve `float` 'a ek olarak Python, `Decimal` ve `Fraction` gibi diğer sayı türlerini de destekler. Python ayrıca karmaşık sayılar için gömülü desteğe sahiptir ve hayali kısmı belirtmek için `j` veya `J` son ekini kullanır (ör. `3+5j`).

3.1.2 Metin

Python sayıların yanı sıra metinleri de ("string" olarak adlandırılan `str` türü ile temsil edilir) işleyebilir. Bu karakterleri "`!`", kelimeleri "`tavşan`", isimleri "`Paris`", cümleleri "`Arkanı kolluyorum.`", vb. içerir. "`Yay! :)`". Tek tırnak (`'...'`) veya çift tırnak (`"..."`) içine alınabilirler ve aynı sonucu verirler[2].

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

Bir alıntıyı alıntılamak için, önüne `\` koyarak "kaçmamız"(escape) gerekir. Alternatif olarak, diğer tırnak işareti türlerini de kullanabiliriz:

```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Python kabuğunda(shell), string tanımı ve çıktı stringi farklı görünebilir. `print()` fonksiyonu, tırnak işaretlerini atlayarak ve kaçan ve özel karakterleri yazdırarak daha okunabilir bir çıktı üretir:

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are interpreted, so \n produces␣
↪new line
First line.
Second line.
```

`\` ile başlayan karakterlerin özel karakterler olarak yorumlanmasını istemiyorsanız, ilk alıntıdan önce bir `r` ekleyerek *ham dizeleri (raw strings)* kullanabilirsiniz:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

[2] Diğer dillerden farklı olarak, `\n` gibi özel karakterler hem tek (`'...'`) hem de çift (`"..."`) tırnak işaretleri ile aynı anlama sahiptir. İkisi arasındaki tek fark, tek tırnak içinde `"` dan kaçmanıza gerek olmamasıdır (ancak `\'` dan kaçmanız gerekir) ve bunun tersi de geçerlidir.

Ham dizelerin ince bir yönü vardır: ham bir dize tek sayıda \ karakterle bitmeyebilir; daha fazla bilgi ve geçici çözümler için SSS 'e bakın.

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...      -h                        Display this usage message
...      -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to

>>>
```

Dizeler + operatörüyle birleştirilebilir (birbirine yapıştırılabilir) ve * ile tekrarlanabilir:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Yan yana iki veya daha fazla *dize sabiti* (yani, tırnak işaretleri arasına alınanlar) otomatik olarak birleştirilir.

```
>>> 'Py' 'thon'
'Python'
```

Bu özellik, özellikle uzun dizeleri kırmak istediğinizde kullanışlıdır:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Bu, değişkenler veya ifadelerle değil, yalnızca iki sabit değerle çalışır:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

Değişkenleri veya bir değişkeni ve bir sabiti birleştirmek istiyorsanız, + kullanın:

```
>>> prefix + 'thon'
'Python'
```

Dizeler, ilk karakterin indeksi 0 olacak şekilde *dizine eklenebilir* (abone olabilir). Karakterler için ayrı bir tür yoktur; karakterler yalnızca *bir* uzunluğunda dizelerdir:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Sağdan saymaya başlamak için indeksler negatif sayılar da olabilir:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

-0 ile 0 aynı olduğundan, negatif endekslerin -1'den başladığını unutmayın.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Dilim indekslerinin kullanışlı varsayılanları vardır; atlanmış bir ilk dizin varsayılanı sıfırdır, atlanmış bir ikinci dizin varsayılanı dilimlenmekte olan dizenin boyutudur.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Başlangıcın her zaman dahil edildiğine ve sonun her zaman hariç tutulduğuna dikkat edin. Bu, `s[:i] + s[i:]` değerinin her zaman `s` değerine eşit olmasını sağlar:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Dilimlerin nasıl çalıştığını hatırlamanın bir yolu, dizinleri ilk karakterin sol kenarı 0 ile *arasındaki* karakterleri işaret ediyor olarak düşünmektir. Ardından, *n* karakterli bir dizenin son karakterinin sağ kenarında *n* dizini vardır, örneğin:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

İlk sayı satırı, dizideki 0…6 endekslerinin konumunu verir; ikinci satır, karşılık gelen negatif endeksleri verir. *i* ile *j* arasındaki dilim, sırasıyla *i* ve *j* etiketli kenarlar arasındaki tüm karakterlerden oluşur.

Negatif olmayan indeksler için, her ikisi de sınırlar içindeyse, bir dilimin uzunluğu indekslerin farkıdır. Örneğin, `kelime[1:3]` 'ün uzunluğu 2'dir.

Çok büyük bir dizin kullanmaya çalışmak bir hataya neden olur:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Ancak, aralık dışı dilim endeksleri, dilimleme için kullanıldığında zarif bir şekilde işlenir:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python dizeleri değiştirilemez — bunlar *immutable* 'dır. Bu nedenle, dizide dizine alınmış bir konuma atamak bir hatayla sonuçlanır:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Farklı bir dizeye ihtiyacınız varsa, yeni bir tane oluşturmalısınız:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Yerleşik işlev `len()`, bir dizenin uzunluğunu döndürür:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Ayrıca bakınız

textseq
: Dizeler, *sıra türlerinin* örnekleridir ve bu türler tarafından desteklenen genel işlemleri destekler.

dize-yöntemleri
: Dizeler, temel dönüşümler ve arama için çok sayıda yöntemi destekler.

f-strings
: Gömülü ifadelere sahip dize sabitleri.

formatstrings
: `str.format()` ile dize biçimlendirme hakkında bilgi.

old-string-formatting
: Dizeler `%` operatörünün sol işleneni olduğunda çağrılan eski biçimlendirme işlemleri burada daha ayrıntılı olarak açıklanmaktadır.

3.1.3 Listeler

Python, diğer değerleri gruplamak için kullanılan bir dizi *bileşik* veri türünü bilir. En çok yönlü olanı, köşeli parantezler arasında virgülle ayrılmış değerlerin (ögelerin) bir listesi olarak yazılabilen *liste*'dir. Listeler farklı türde ögeler içerebilir, ancak genellikle ögelerin tümü aynı türdedir.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Dizeler gibi (ve diğer tüm yerleşik *sequence* türleri), listeler dizine alınabilir ve dilimlenebilir:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Ayrıca listeler birleştirme gibi işlemleri de destekler:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

immutable olan dizelerin aksine, listeler *mutable* türündedir, yani içeriklerini değiştirmek mümkündür:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` *method* (we will see more about methods later):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it.:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # they reference the same object
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

Tüm dilim işlemleri, istenen ögeleri içeren yeni bir liste döndürür. Bu, aşağıdaki dilimin listenin bir shallow copy döndürdüğü anlamına gelir:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

Dilimlere atama da mümkündür ve bu, listenin boyutunu bile değiştirebilir veya tamamen temizleyebilir:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Yerleşik işlev `len()` ayrıca listeler için de geçerlidir:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Listeleri iç içe yerleştirmek (diğer listeleri içeren listeler oluşturmak) mümkündür, örneğin:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Programlamaya Doğru İlk Adımlar

Elbette Python'ı iki ile ikiyi toplamaktan daha komplike görevler için kullanabiliriz. Örneğin, Fibonacci serisinin ilk alt dizisini aşağıdaki gibi yazabiliriz:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
5
8
```

Bu örnek, birkaç yeni özellik sunar.

- İlk satır bir *çoklu atama*: `a` ve `b` değişkenleri aynı anda yeni 0 ve 1 değerlerini alır. Tarafların tümü, herhangi bir görev yapılmadan önce değerlendirilir. Sağ taraftaki ifadeler soldan sağa doğru değerlendirilir.
- `while` döngüsü, koşul (burada: `a < 10`) doğru kaldığı sürece yürütülür. Python'da, C'de olduğu gibi, sıfır olmayan herhangi bir tam sayı değeri doğrudur; sıfır yanlıştır. Koşul ayrıca bir dizi veya liste değeri, aslında herhangi bir dizi olabilir; uzunluğu sıfır olmayan her şey doğrudur, boş diziler yanlıştır. Örnekte kullanılan test basit bir karşılaştırmadır. Standart karşılaştırma işleçleri C'dekiyle aynı şekilde yazılır: < (küçüktür), > (büyüktür), == (eşittir), < = (küçük veya eşit), > = (büyük veya eşit) ve != (eşit değil).
- Döngünün *gövdesi girintilidir*: girinti, Python'un ifadeleri gruplama şeklidir. Etkileşimli komut isteminde, girintili her satır için bir sekme veya boşluk(lar) yazmanız gerekir. Pratikte, bir metin düzenleyici ile Python için daha karmaşık girdiler hazırlayacaksınız; tüm düzgün metin editörlerinin otomatik girinti özelliği vardır. Bir bileşik deyim etkileşimli olarak girildiğinde, tamamlandığını belirtmek için boş bir satırdan sonra gelmelidir (çünkü ayrıştırıcı son satırı ne zaman yazdığınızı tahmin edemez). Bir temel blok içindeki her satırın aynı miktarda girintili olması gerektiğini unutmayın.
- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

end anahtar sözcüğü argümanı, çıktıdan sonra yeni satırı önlemek veya çıktıyı farklı bir dizeyle bitirmek için kullanılabilir:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

BÖLÜM 4

Daha Fazla Kontrol Akışı Aracı

As well as the `while` statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1 `if` İfadeleri

Belki de en iyi bilinen deyim türü `if` deyimidir. Örneğin:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Sıfır veya daha fazla `elif` bölümü olabilir ve `else` bölümü isteğe bağlıdır. 'elif' anahtar sözcüğü 'else if' ifadesinin kısaltmasıdır ve aşırı girintiden kaçınmak için kullanışlıdır. Bir `if` … `elif` … `elif` … dizisi, diğer dillerde bulunan `switch` veya `case` deyimlerinin yerine geçer.

Aynı değeri birkaç sabitle karşılaştırıyorsanız veya belirli türleri veya nitelikleri kontrol ediyorsanız, `match` deyimini de yararlı bulabilirsiniz. Daha fazla ayrıntı için *pass İfadeleri* bölümüne bakınız.

4.2 `for` İfadeleri

Python'daki `for` deyimi, C veya Pascal'da alışkın olduğunuzdan biraz farklıdır. Her zaman sayıların aritmetik ilerlemesi üzerinde yineleme yapmak (Pascal'daki gibi) veya kullanıcıya hem yineleme adımını hem de durma koşulunu tanımlama yeteneği vermek (C gibi) yerine, Python'un `for` deyimi, herhangi bir dizinin (bir liste veya bir dize) öğeleri üzerinde, dizide göründükleri sırayla yineler. Örneğin (kelime oyunu yapmak istemedim):

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Aynı koleksiyon üzerinde yineleme yaparken bir koleksiyonu değiştiren kodun doğru yazılması zor olabilir. Bunun yerine, koleksiyonun bir kopyası üzerinde döngü yapmak veya yeni bir koleksiyon oluşturmak genellikle daha kolaydır:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '□□□': 'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 `range()` Fonksiyonu

Bir sayı dizisi üzerinde yineleme yapmanız gerekiyorsa, yerleşik `range()` fonksiyonu kullanışlı olur. Aritmetik ilerlemeler üretir:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Verilen bitiş noktası asla oluşturulan dizinin bir parçası değildir; `range(10)` 10 değer üretir, 10 uzunluğundaki bir dizinin öğeleri için yasal indisler. Aralığın başka bir sayıdan başlamasına izin vermek veya farklı bir artış (negatif bile olsa; bazen buna 'adım' denir) belirtmek mümkündür:

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Bir dizinin indisleri üzerinde yineleme yapmak için `range()` ve `len()` öğelerini aşağıdaki gibi birleştirebilirsiniz:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Ancak bu tür durumların çoğunda `enumerate()` fonksiyonunu kullanmak uygundur, bkz *Döngü Teknikleri*.

Sadece bir aralık yazdırırsanız garip bir şey olur:

```
>>> range(10)
range(0, 10)
```

Birçok yönden `range()` tarafından döndürülen nesne bir listeymiş gibi davranır, ancak aslında öyle değildir. Üzerinde yineleme yaptığınızda istenen dizinin ardışık öğelerini döndüren bir nesnedir, ancak listeyi gerçekten oluşturmaz, böylece yerden tasarruf sağlar.

Böyle bir nesnenin *iterable* olduğunu, yani arz tükenene kadar ardışık öğeler elde edebilecekleri bir şey bekleyen fonksiyonlar ve yapılar için bir hedef olarak uygun olduğunu söylüyoruz. Daha önce `for` deyiminin böyle bir yapı olduğunu görmüştük, bir yinelenebilir alan bir fonksiyon örneği ise `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Daha sonra yinelenebilirleri döndüren ve argüman olarak yinelenebilirleri alan daha fazla fonksiyon göreceğiz. *Veri Yapıları* bölümünde, `list()` hakkında daha ayrıntılı olarak tartışacağız.

4.4 `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 `else` Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

One way to think of the else clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like if/if/if/else. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see *Özel Durumları İşleme*.

4.6 `pass` İfadeleri

`pass` deyimi hiçbir şey yapmaz. Sözdizimsel olarak bir deyim gerektiğinde ancak program hiçbir eylem gerektirmediğinde kullanılabilir. Örneğin:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Bu genellikle minimal sınıflar oluşturmak için kullanılır:

```
>>> class MyEmptyClass:
...     pass
...
```

`pass` 'in kullanılabileceği bir başka yer de, yeni kod üzerinde çalışırken bir fonksiyon veya koşul gövdesi için bir yer tutucu olarak daha soyut bir düzeyde düşünmeye devam etmenizi sağlamaktır. `pass` sessizce göz ardı edilir:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.7 `pass` İfadeleri

`match` bir ifadeyi alır ve değerini bir veya daha fazla case bloğu olarak verilen ardışık kalıplarla karşılaştırır. Bu, C, Java veya JavaScript'teki (ve diğer birçok dildeki) bir switch ifadesine yüzeysel olarak benzer, ancak Rust veya Haskell gibi dillerdeki kalıp eşleştirmeye daha çok benzer. Yalnızca eşleşen ilk kalıp yürütülür ve ayrıca bileşenleri (sıra öğeleri veya nesne nitelikleri) değerden değişkenlere çıkarabilir.

En basit form, bir konu değerini bir veya daha fazla sabitle karşılaştırır:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Son bloğa dikkat edin: "değişken adı" _ bir *wildcard* görevi görür ve asla eşleşmez. Hiçbir durum eşleşmezse, dallardan hiçbiri yürütülmez.

| ("or") kullanarak birkaç sabiti tek bir kalıpta birleştirebilirsiniz:

```
case 401 | 403 | 404:
    return "Not allowed"
```

Kalıplar paket açma atamaları gibi görünebilir ve değişkenleri bağlamak için kullanılabilir:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y ={y}")
    case (x, 0):
        print(f"X ={x}")
    case (x, y):
        print(f"X ={x}, Y ={y}")
    case _:
        raise ValueError("Not a point")
```

Bunu dikkatle inceleyin! İlk kalıpta iki sabit vardır ve yukarıda gösterilen sabit kalıbının bir uzantısı olarak düşünülebilir. Ancak sonraki iki kalıp bir sabit ve bir değişkeni birleştirir ve değişken özneden (`point`) bir değer *bağlar*. Dördüncü kalıp iki değeri yakalar, bu da onu kavramsal olarak `(x, y) = point` paket açma atamasına benzer hale getirir.

Verilerinizi yapılandırmak için sınıfları kullanıyorsanız, sınıf adını ve ardından bir yapıcıya benzeyen, ancak nitelikleri değişkenlere yakalama yeteneğine sahip bir argüman listesi kullanabilirsiniz:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y ={y}")
        case Point(x=x, y=0):
            print(f"X ={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Konumsal parametreleri, nitelikleri için bir sıralama sağlayan bazı yerleşik sınıflarla (örneğin veri sınıfları) kullanabilirsiniz. Ayrıca sınıflarınızda `__match_args__` niteliğini ayarlayarak kalıplardaki nitelikler için belirli bir konum tanımlayabilirsiniz. Bu özellik ("x", "y") olarak ayarlanırsa, bahsi geçen kalıpların hepsi eş değerdir (ve hepsi `y` niteliğini `var` değişkenine bağlar):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Kalıpları okumak için önerilen bir yol, hangi değişkenlerin neye ayarlanacağını anlamak için onlara bir atamanın soluna koyacağınız şeyin genişletilmiş bir biçimi olarak bakmaktır. Yalnızca bağımsız isimler (yukarıdaki `var` gibi) bir eşleştirme deyimi tarafından atanır. Noktalı isimlere (`foo.bar` gibi), nitelik isimlerine (yukarıdaki `x =` ve `y =` gibi) veya sınıf isimlerine (yukarıdaki `Point` gibi yanlarındaki "(…)" ile tanınan) asla atama yapılmaz.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

Bir kalıba "guard" olarak bilinen bir `if` cümlesi ekleyebiliriz. Eğer guard yanlış ise, `match` bir sonraki case bloğunu denemeye devam eder. Değer yakalamanın koruma değerlendirilmeden önce gerçekleştiğine dikkat edin:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y =X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Bu açıklamanın diğer bazı kilit özellikleri:

- Paket açma atamaları gibi, tuple ve liste kalıpları da tamamen aynı anlama sahiptir ve aslında rastgele dizilerle eşleşir. Önemli bir istisna, yineleyicilerle veya string'lerle eşleşmezler.
- Sıra kalıpları genişletilmiş paket açmayı destekler: `[x, y, *rest]` ve `(x, y, *rest)` paket açma atamalarına benzer şekilde çalışır. `*` öğesinden sonraki ad _ de olabilir, bu nedenle `(x, y, *_)` öğesi, kalan öğeleri bağlamadan en az iki öğeden oluşan bir diziyle eşleşir.
- Eşleme kalıpları: `{"bandwidth": b, "latency": l}` bir sözlükten `"bandwidth"` ve `"latency"` değerlerini yakalar. Sıra kalıplarının aksine, ekstra anahtarlar göz ardı edilir. `**rest` gibi bir paket açma da desteklenir. (Ancak `**_` gereksiz olacağından buna izin verilmez)
- Alt kalıplar `as` anahtar sözcüğü kullanılarak yakalanabilir:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  girdinin ikinci elemanını `p2` olarak yakalayacaktır (girdi iki noktadan oluşan bir dizi olduğu sürece)
- Çoğu sabit eşitlikle karşılaştırılır, ancak `True`, `False` ve `None` tekilleri özdeşlikle karşılaştırılır.
- Kalıplar adlandırılmış sabitler kullanabilir. Bunlar, yakalama değişkeni olarak yorumlanmalarını önlemek için noktalı isimler olmalıdır:

  ```
  from enum import Enum
  class Color(Enum):
      RED = 'red'
      GREEN = 'green'
      BLUE = 'blue'

  color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

  match color:
      case Color.RED:
          print("I see red!")
      case Color.GREEN:
          print("Grass is green")
      case Color.BLUE:
          print("I'm feeling the blues :(")
  ```

Daha ayrıntılı bir açıklama ve ek örnekler için, öğretici bir formatta yazılmış olan **PEP 636** sayfasına bakabilirsiniz.

4.8 Fonksiyonların Tanımlanması

Fibonacci serisini rastgele bir sınıra kadar yazan bir fonksiyon oluşturabiliriz:

```
>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Anahtar kelime `def` bir fonksiyon *tanımını* tanıtır. Bunu fonksiyon adı ve parantez içine alınmış resmi parametreler listesi takip etmelidir. Fonksiyonun gövdesini oluşturan ifadeler bir sonraki satırdan başlar ve girintili olmalıdır.

Fonksiyon gövdesinin ilk ifadesi isteğe bağlı olarak bir string literal olabilir; bu string literal fonksiyonun dokümantasyon stringi veya *docstring* 'dir. (Docstringler hakkında daha fazla bilgi *Dokümantasyon Stringler'i* bölümünde bulunabilir.) Otomatik olarak çevrimiçi veya basılı dokümantasyon üretmek veya kullanıcının etkileşimli olarak kodda gezinmesini sağlamak için docstringleri kullanan araçlar vardır; yazdığınız koda docstringler eklemek iyi bir uygulamadır, bu yüzden bunu alışkanlık haline getirin.

Bir fonksiyonun *çalıştırılması*, fonksiyonun yerel değişkenleri için kullanılan yeni bir sembol tablosu ortaya çıkarır. Daha açık bir ifadeyle, bir fonksiyon içindeki tüm değişken atamaları değeri yerel sembol tablosunda saklar; oysa değişken referansları önce yerel sembol tablosuna, sonra çevreleyen fonksiyonların yerel sembol tablolarına, daha sonra global sembol tablosuna ve son olarak da yerleşik isimler tablosuna bakar. Bu nedenle, global değişkenlere ve çevreleyen fonksiyonların değişkenlerine bir fonksiyon içinde doğrudan değer atanamaz (global değişkenler için bir `global` deyiminde veya çevreleyen fonksiyonların değişkenleri için bir `nonlocal` deyiminde isimlendirilmedikçe), ancak bunlara referans verilebilir.

Bir fonksiyon çağrısının gerçek parametreleri (argümanları), çağrıldığında çağrılan fonksiyonun yerel sembol tablosunda tanıtılır; bu nedenle, argümanlar *call by value* (burada *value* her zaman bir nesne *referans*'dır, nesnenin değeri değildir) kullanılarak aktarılır.[1] Bir fonksiyon başka bir fonksiyonu çağırdığında veya kendini tekrarlı olarak çağırdığında, bu çağrı için yeni bir yerel sembol tablosu oluşturulur.

Bir fonksiyon tanımı, fonksiyon adını geçerli sembol tablosundaki fonksiyon nesnesiyle ilişkilendirir. Yorumlayıcı, bu adın işaret ettiği nesneyi kullanıcı tanımlı bir fonksiyon olarak tanır. Diğer isimler de aynı fonksiyon nesnesine işaret edebilir ve fonksiyona erişmek için kullanılabilir:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Diğer dillerden geliyorsanız, `fib` 'in bir fonksiyon değil, değer döndürmediği için bir prosedür olduğuna itiraz edebilirsiniz. Aslında, `return` ifadesi olmayan fonksiyonlar bile, oldukça sıkıcı olsa da, bir değer döndürürler. Bu değer `None` olarak adlandırılır (yerleşik bir isimdir). Normalde `None` değerinin yazılması, yazılan tek değer olacaksa yorumlayıcı tarafından bastırılır. Eğer gerçekten istiyorsanız `print()` kullanarak görebilirsiniz:

```
>>> fib(0)
>>> print(fib(0))
None
```

Fibonacci serisindeki sayıların listesini döndürebilecek bir fonksiyon yazmak gayet basittir, onun yerine şunu yazdırarak:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
```

(sonraki sayfaya devam)

[1] Aslında, *nesne referansı ile çağırma* daha iyi bir tanımlama olacaktır, çünkü değiştirilebilir bir nesne aktarılırsa, çağıran, çağırılanın üzerinde yaptığı tüm değişiklikleri (bir listeye eklenen öğeler) görecektir.

(önceki sayfadan devam)

```
...
>>> f100 = fib2(100)     # call it
>>> f100                 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Bu örnek, her zamanki gibi, bazı yeni Python özelliklerini göstermektedir:

- Bir `return` deyimi bir fonksiyondan bir değerle döner. `return` deyimi bir ifade argümanı olmadan `None` döndürür. Bir fonksiyonun sonundan düşmek de `None` değerini döndürür.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see *Sınıflar*) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9 İşlev Tanımlama hakkında daha fazla bilgi

Değişken sayıda argüman içeren fonksiyonlar tanımlamak da mümkündür. Birleştirilebilen üç form vardır.

4.9.1 Varsayılan Değişken Değerleri

En kullanışlı biçim, bir veya daha fazla bağımsız değişken için varsayılan bir değer belirtmektir. Bu, izin vermek üzere tanımlandığından daha az sayıda bağımsız değişkenle çağrılabilen bir fonksiyon oluşturur. Örneğin:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Bu fonksiyon çeşitli yollarla çağrılabilir:

- sadece zorunlu argümanı vererek: `ask_ok('Gerçekten çıkmak istiyor musun?')`
- isteğe bağlı değişkenlerden birini vermek: `ask_ok('OK to overwrite the file?', 2)`
- ya da bütün değişkenleri vermek: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Bu örnek ayrıca `in` anahtar sözcüğünü de tanıtır. Bu, bir dizinin belirli bir değer içerip içermediğini test eder.

Varsayılan değerler *tanımlayan* kapsamdaki fonksiyon tanımlama noktasında değerlendirilir, böylece

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

5 çıktısını verecektir.

Önemli uyarı: Varsayılan değer yalnızca bir kez değerlendirilir. Varsayılan değer liste, sözlük veya çoğu sınıfın örnekleri gibi değiştirilebilir bir nesne olduğunda bu durum fark yaratır. Örneğin, aşağıdaki fonksiyon sonraki çağrılarda kendisine aktarılan argümanları biriktirir:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Bu şu çıktıyı verecektir

```
[1]
[1, 2]
[1, 2, 3]
```

Varsayılan değerin sonraki çağrılar arasında paylaşılmasını istemiyorsanız, bunun yerine fonksiyonu şu şekilde yazabilirsiniz:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2 Anahtar Kelime Değişkenleri

Fonksiyonlar ayrıca `kwarg =value` şeklinde *anahtar kelime argümanları* kullanılarak da çağrılabilir. Örneğin, aşağıdaki fonksiyon:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

bir gerekli argüman (`voltage`) ve üç isteğe bağlı argüman (`state`, `action` ve `type`) kabul eder. Bu fonksiyon aşağıdaki yollardan herhangi biriyle çağrılabilir:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

ancak aşağıdaki tüm çağrılar geçersiz olacaktır:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Bir fonksiyon çağrısında, anahtar kelime argümanları konumsal argümanları takip etmelidir. Aktarılan tüm anahtar sözcük argümanları fonksiyon tarafından kabul edilen argümanlardan biriyle eşleşmelidir (örneğin `actor` `parrot`

fonksiyonu için geçerli bir argüman değildir) ve sıraları önemli değildir. Buna isteğe bağlı olmayan argümanlar da dahildir (örneğin `parrot(voltage =1000)` da geçerlidir). Hiçbir argüman birden fazla değer alamaz. İşte bu kısıtlama nedeniyle başarısız olan bir örnek:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

`**name` biçiminde bir son biçimsel parametre mevcut olduğunda, biçimsel parametreye karşılık gelenler dışındaki tüm anahtar kelime argümanlarını içeren bir sözlük alır (bkz. typesmapping). Bu, biçimsel parametre *tuple* listesinin ötesindeki konumsal argümanları içeren bir `*name` biçimindeki bir biçimsel parametre ile birleştirilebilir (bir sonraki alt bölümde açıklanmıştır). (`*name`, `**name` 'den önce gelmelidir.) Örneğin, aşağıdaki gibi bir fonksiyon tanımlarsak:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Şöyle denebilir:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

ve tabii ki yazdıracaktır:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Anahtar sözcük bağımsız değişkenlerinin yazdırılma sırasının, fonksiyon çağrısında sağlandıkları sırayla eşleşmesinin garanti edildiğini unutmayın.

4.9.3 Özel parametreler

Varsayılan olarak, argümanlar bir Python fonksiyonuna ya pozisyona göre ya da açıkça anahtar kelimeye göre aktarılabilir. Okunabilirlik ve performans için, argümanların geçirilme şeklini kısıtlamak mantıklıdır, böylece bir geliştiricinin öğelerin konumla mı, konumla ya da anahtar sözcükle mi yoksa anahtar sözcükle mi geçirildiğini belirlemek için yalnızca fonksiyon tanımına bakması gerekir.

Bir fonksiyon tanımı aşağıdaki gibi görünebilir:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        |             |                   |
        |        Positional or keyword   |
        |                                 - Keyword only
         -- Positional only
```

burada / ve * isteğe bağlıdır. Kullanılırsa, bu semboller, argümanların fonksiyona nasıl geçirilebileceğine göre parametre türünü gösterir: yalnızca konumsal, konumsal veya anahtar sözcük ve yalnızca anahtar sözcük. Anahtar sözcük parametreleri, adlandırılmış parametreler olarak da adlandırılır.

Konumsal veya Anahtar Kelime Argümanları

Eğer / ve * fonksiyon tanımında mevcut değilse, argümanlar bir fonksiyona pozisyon veya anahtar kelime ile aktarılabilir.

Yalnızca Konumsal Parametreler

Bu konuya biraz daha detaylı bakacak olursak, belirli parametreleri *positional-only* olarak işaretlemek mümkündür. Eğer *konumsal-sadece* ise, parametrelerin sırası önemlidir ve parametreler anahtar kelime ile aktarılamaz. Yalnızca konumsal parametreler bir / (ileri eğik çizgi) önüne yerleştirilir. / sadece konumsal parametreleri diğer parametrelerden mantıksal olarak ayırmak için kullanılır. Fonksiyon tanımında / yoksa, sadece konumsal parametre yoktur.

/ işaretini takip eden parametreler *konumsal veya anahtar sözcük* veya *sadece anahtar sözcük* olabilir.

Yalnızca Anahtar Sözcük İçeren Değişkenler

Parametrelerin anahtar sözcük argümanıyla geçirilmesi gerektiğini belirterek parametreleri *anahtar sözcüğe özel* olarak işaretlemek için, argüman listesine ilk *anahtar sözcüğe özel* parametreden hemen önce bir * yerleştirin.

Fonksiyon Örnekleri

/ ve * işaretlerine çok dikkat ederek aşağıdaki örnek fonksiyon tanımlarını göz önünde bulundurun:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

İlk fonksiyon tanımı, `standard_arg`, en bilinen biçimdir, çağırma kuralına herhangi bir kısıtlama getirmez ve argümanlar konum veya anahtar kelime ile aktarılabilir:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

İkinci fonksiyon `pos_only_arg`, fonksiyon tanımında bir / olduğu için sadece konumsal parametreleri kullanacak şekilde sınırlandırılmıştır:

```
>>> pos_only_arg(1)
1
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

Sonuncusu ise aynı fonksiyon tanımında üç çağrı kuralını da kullanır:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword␣
↪arguments: 'pos_only'
```

Son olarak, `name` konumsal argümanı ile `name` anahtarına sahip `**kwds` arasında potansiyel bir çakışma olan bu fonksiyon tanımını düşünün:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Anahtar kelime `'name'` her zaman ilk parametreye bağlanacağı için `True` döndürmesini sağlayacak olası bir çağrı yoktur. Örneğin:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Ancak `/` (yalnızca konumsal argümanlar) kullanıldığında, `name` bir konumsal argüman olarak ve `'name'` anahtar kelime argümanlarında bir anahtar olarak izin verdiği için mümkündür:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> foo(1, **{'name': 2})
True
```

Başka bir deyişle, yalnızca konumsal parametrelerin adları `**kwds` içinde belirsizlik olmadan kullanılabilir.

Özet

Kullanım durumu, fonksiyon tanımında hangi parametrelerin kullanılacağını belirleyecektir:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Rehber olarak:

- Parametrelerin adının kullanıcı tarafından kullanılamamasını istiyorsanız sadece pozisyonel seçeneğini kullanın. Bu, parametre adlarının gerçek bir anlamı olmadığında, fonksiyon çağrıldığında bağımsız değişkenlerin sırasını zorlamak istediğinizde veya bazı konumsal parametreler ve rastgele anahtar sözcükler almanız gerektiğinde kullanışlıdır.
- Adların bir anlamı olduğunda ve fonksiyon tanımının adlarla açık olmasıyla daha anlaşılır olduğunda veya kullanıcıların geçirilen argümanın konumuna güvenmesini önlemek istediğinizde yalnızca anahtar sözcük kullanın.
- Bir API için, parametrenin adı gelecekte değiştirilirse API değişikliklerinin bozulmasını önlemek için yalnızca konumsal kullanın.

4.9.4 Keyfi Argüman Listeleri

Son olarak, en az kullanılan seçenek, bir fonksiyonun rastgele sayıda argümanla çağrılabileceğini belirtmektir. Bu argümanlar bir tuple içinde paketlenecektir (bkz *Veri Grupları ve Diziler*). Değişken argüman sayısından önce, sıfır veya daha fazla normal argüman olabilir.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalde *variadic* argümanlar biçimsel parametreler listesinde en sonda yer alır, çünkü fonksiyona aktarılan geri kalan tüm girdi argümanlarını toplarlar. `*args` parametresinden sonra gelen tüm biçimsel parametreler 'keyword-only' (yalnızca-anahtar-kelime) argümanlarıdır, yani konumsal argümanlar yerine sadece anahtar kelimeler olarak kullanılabilirler.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.9.5 Argüman Listelerini Açma

Tersi durum, argümanlar zaten bir liste veya tuple içinde olduğunda, ancak ayrı konumsal argümanlar gerektiren bir fonksiyon çağrısı için paketten çıkarılması gerektiğinde ortaya çıkar. Örneğin, yerleşik `range()` fonksiyonu ayrı *start* ve *stop* argümanları bekler. Eğer bunlar ayrı olarak mevcut değilse, argümanları bir listeden veya tuple'dan çıkarmak için fonksiyon çağrısını `*`-operatörü ile yazın:

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

Aynı şekilde, sözlükler `**`-operatörü ile anahtar sözcük argümanları sunabilir:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin
↪' demised !
```

4.9.6 Lambda İfadeleri

Küçük anonim fonksiyonlar `lambda` anahtar sözcüğü ile oluşturulabilir. Bu fonksiyon iki argümanının toplamını döndürür: `lambda a, b: a+b`. Lambda fonksiyonları, fonksiyon nesnelerinin gerekli olduğu her yerde kullanılabilir. Sözdizimsel olarak tek bir ifadeyle sınırlıdırlar. Anlamsal olarak, normal bir fonksiyon tanımı için sadece sözdizimsel şekerdirler. İç içe işlev tanımları gibi, lambda işlevleri de içeren kapsamdaki değişkenlere başvurabilir:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Yukarıdaki örnekte bir fonksiyon döndürmek için bir lambda ifadesi kullanılmıştır. Başka bir kullanım da küçük bir fonksiyonu argüman olarak geçirmektir:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7 Dokümantasyon Stringler'i

Belge dizelerinin içeriği ve biçimlendirilmesiyle ilgili bazı kurallar aşağıda verilmiştir.

İlk satır her zaman nesnenin amacının kısa ve öz bir özeti olmalıdır. Öz olması için, nesnenin adı veya türü açıkça belirtilmemelidir, çünkü bunlar başka yollarla elde edilebilir (adın bir fonksiyonun çalışmasını açıklayan bir fiil olması durumu hariç). Bu satır büyük harfle başlamalı ve nokta ile bitmelidir.

Belgeleme string'inde daha fazla satır varsa, ikinci satır boş olmalı ve özeti açıklamanın geri kalanından görsel olarak ayırmalıdır. Sonraki satırlar, nesnenin çağrı kurallarını, yan etkilerini vb. açıklayan bir veya daha fazla paragraftan oluşmalıdır.

Python ayrıştırıcısı, Python'daki çok satırlı dize değişmezlerinden girintiyi çıkarmaz, bu nedenle belgeleri işleyen araçların istenirse girintiyi çıkarması gerekir. Bu, aşağıdaki kural kullanılarak yapılır. Dizenin ilk satırından *sonraki* boş olmayan ilk satır, tüm dokümantasyon dizesi için girinti miktarını belirler. (İlk satırı kullanamayız, çünkü genellikle dizenin açılış tırnaklarına bitişiktir, bu nedenle girintisi dize değişmezinde belirgin değildir) Bu girintiye "eş değer" boşluk daha sonra dizenin tüm satırlarının başlangıcından çıkarılır. Daha az girintili satırlar oluşmamalıdır, ancak oluşurlarsa başlarındaki tüm boşluklar çıkarılmalıdır. Beyaz boşlukların eş değerliği sekmelerin genişletilmesinden sonra test edilmelidir (normalde 8 boşluğa kadar).

İşte çok satırlı bir docstring örneği:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.9.8 Fonksiyon Ek Açıklamaları

Fonksiyon ek açıklamaları kullanıcı tanımlı fonksiyonlar tarafından kullanılan tipler hakkında tamamen isteğe bağlı meta veri bilgileridir (daha fazla bilgi için **PEP 3107** ve **PEP 484** sayfalarına bakınız).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10 Intermezzo: Kodlama Stili

Artık daha uzun, daha karmaşık Python parçaları yazmak üzere olduğunuza göre, *kodlama stili* hakkında konuşmak için iyi bir zaman. Çoğu dil farklı stillerde yazılabilir (ya da daha özlü bir ifadeyle *biçimlendirilebilir*); bazıları diğerlerinden daha okunaklıdır. Başkalarının kodunuzu okumasını kolaylaştırmak her zaman iyi bir fikirdir ve güzel bir kodlama stili benimsemek buna çok yardımcı olur.

Python için **PEP 8**, çoğu projenin bağlı olduğu stil kılavuzu olarak ortaya çıkmıştır; okunabilir ve göze hoş gelen bir kodlama stilini teşvik eder. Her Python geliştiricisi bir noktada onu okumalıdır; işte sizin için çıkarılan en önemli noktalar:

- 4 aralıklı girinti kullanın ve sekme kullanmayın.

  4 boşluk, küçük girinti (daha fazla iç içe geçme derinliği sağlar) ve büyük girinti (okunması daha kolay) arasında iyi bir uzlaşmadır. Sekmeler karışıklığa neden olur ve en iyisi dışarıda bırakmaktır.

- Satırları 79 karakteri geçmeyecek şekilde sarın.

  Bu, küçük ekranlı kullanıcılara yardımcı olur ve daha büyük ekranlarda birkaç kod dosyasının yan yana olmasını mümkün kılar.

- Fonksiyonları ve sınıfları ve fonksiyonların içindeki büyük kod bloklarını ayırmak için boş satırlar kullanın.

- Mümkün olduğunda, yorumları kendi başlarına bir satıra koyun.

- Docstrings kullanın.

- Operatörlerin etrafında ve virgüllerden sonra boşluk kullanın, ancak doğrudan parantez yapılarının içinde kullanmayın: `a = f(1, 2) + g(3, 4)`.
- Sınıflarınızı ve fonksiyonlarınızı tutarlı bir şekilde adlandırın; buradaki kural, sınıflar için `UpperCamelCase`, fonksiyonlarını; metotlar için de `lowercase_with_underscores` kullanmaktır. İlk yöntem argümanının adı olarak her zaman `self` kullanın (sınıflar ve yöntemler hakkında daha fazla bilgi için *Sınıflara İlk Bakış* bölümüne bakın).
- Kodunuz uluslararası ortamlarda kullanılacaksa süslü kodlamalar kullanmayın. Python'un varsayılanı, UTF-8 veya hatta düz ASCII her durumda en iyi sonucu verir.
- Aynı şekilde, farklı bir dil konuşan kişilerin kodu okuması veya muhafaza etmesi için en ufak bir şans varsa, tanımlayıcılarda ASCII olmayan karakterler kullanmayın.

BÖLÜM 5

Veri Yapıları

Bu bölüm, daha önce öğrendiğiniz bazı şeyleri daha ayrıntılı olarak açıklamakta ve bazı yeni şeyler de eklemektedir.

5.1 Listeler Üzerine

Liste veri türünün bazı yöntemleri daha vardır. Liste nesnelerinin tüm metotları şunlardır:

`list.`**`append`**`(`*x*`)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.

`list.`**`insert`**`(`*i*, *x*`)`

Verilen pozisyona bir öğe ekleyin. İlk argüman, daha önce ekleyeceğiniz öğenin indeksidir, bu nedenle `a.insert(0, x)` listenin önüne ekler ve `a.insert(len(a), x)` komutu `a.append(x)` komutu ile eş değerdir.

`list.`**`remove`**`(`*x*`)`

Değeri *x*'e eşit olan ilk öğeyi listeden kaldırır. Böyle bir öğe yoksa `ValueError` hatası ortaya çıkar.

`list.`**`pop`**`(`[*i*]`)`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.

`list.`**`clear`**`()`

Remove all items from the list. Similar to `del a[:]`.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

Değeri *x*'e eşit olan ilk öğenin listedeki sıfır tabanlı indeksini döndürür. Böyle bir öğe yoksa `ValueError` hatası ortaya çıkar.

İsteğe bağlı *start* ve *end* argümanları dilim gösteriminde olduğu gibi yorumlanır ve aramayı listenin belirli bir alt dizisiyle sınırlamak için kullanılır. Döndürülen dizin, *start* bağımsız değişkeni yerine tam dizinin başlangıcına göre hesaplanır.

`list.`**`count`**`(`*x*`)`

Listede *x* öğesinin kaç kez göründüğünü döndürür.

`list.sort(*, key=None, reverse=False)`

Listenin öğelerini yerinde sıralayın (argümanlar sıralama özelleştirmesi için kullanılabilir, açıklamaları için bkz: `sorted()`).

`list.reverse()`

Listenin öğelerini yerinde ters çevirir.

`list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

Liste yöntemlerinin çoğunu kullanan bir örnek:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Listeyi yalnızca değiştiren `insert`, `remove` veya `sort` gibi yöntemlerin hiçbir dönüş değeri yazdırmadığını fark etmiş olabilirsiniz – onlar varsayılan `None` değerini döndürürler.[1] Bu, Python'daki tüm değiştirilebilir veri yapıları için bir tasarım ilkesidir.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.1.1 Listeleri Yığın Olarak Kullanma

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
```

(sonraki sayfaya devam)

[1] Diğer diller, `d->insert("a")->remove("b")->sort();` gibi yöntem zincirlemesine izin veren değiştirilmiş nesneyi döndürebilir.

(önceki sayfadan devam)

```
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Listeleri Kuyruk Olarak Kullanma

Bir listeyi, eklenen ilk elemanın alınan ilk eleman olduğu bir kuyruk olarak kullanmak da mümkündür ("ilk giren ilk çıkar"); ancak listeler bu amaç için verimli değildir. Listenin sonundan ekleme ve çıkarma işlemleri hızlıyken, listenin başından ekleme veya çıkarma yapmak yavaştır (çünkü diğer tüm öğelerin bir adım kaydırılması gerekir).

Bir kuyruk uygulamak için, her iki uçtan da hızlı ekleme ve çıkarma yapmak üzere tasarlanmış `collections.deque` kullanın. Örneğin:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Liste Kavramaları

Liste kavramaları, listeler oluşturmak için kısa bir yol sağlar. Yaygın uygulamalar, her bir öğenin başka bir dizinin veya yinelenebilir öğenin her bir üyesine uygulanan bazı işlemlerin sonucu olduğu yeni listeler oluşturmak veya belirli bir koşulu karşılayan öğelerin bir alt dizisini oluşturmaktır.

Örneğin, aşağıdaki gibi bir kareler listesi oluşturmak istediğimizi varsayalım:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Bunun, `x` adında bir değişken yarattığına (veya üzerine yazdığına) ve bu değişkenin döngü tamamlandıktan sonra da var olduğuna dikkat edin. Kareler listesini, şunu kullanarak herhangi bir yan etki olmadan hesaplayabiliriz:

```
squares = list(map(lambda x: x**2, range(10)))
```

veya aynı şekilde:

```
squares = [x**2 for x in range(10)]
```

ki bu daha kısa ve okunaklıdır.

Bir liste kavrayışı, bir ifade içeren parantezlerden ve ardından bir `for` cümlesinden, ardından sıfır veya daha fazla `for` veya `if` cümlesinden oluşur. Sonuç, ifadenin kendisini takip eden `for` ve `if` cümleleri bağlamında değerlendirilmesiyle elde edilen yeni bir liste olacaktır. Örneğin, listcomp, eşit değillerse iki listenin öğelerini birleştirir:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

ve şuna eş değerdir:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Her iki kod parçacığında da `for` ve `if` ifadelerinin sıralamasının aynı olduğuna dikkat edin.

Eğer ifade bir veri grubu ise (örneğin önceki örnekteki `(x, y)`), parantez içine alınmalıdır.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Liste kavramaları karmaşık ifadeler ve iç içe geçmiş fonksiyonlar içerebilir:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 İç İçe Liste Kavramaları

Bir liste kavrayışındaki ilk ifade, başka bir liste kavrayışı da dahil olmak üzere rastgele herhangi bir ifade olabilir.

Uzunluğu 4 olan 3 listeden oluşan bir liste olarak uygulanan 3x4'lük bir matrisin aşağıdaki örneğini düşünün:

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

Aşağıdaki liste kavraması satır ve sütunların yerlerini değiştirir:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Önceki bölümde gördüğümüz gibi, iç liste kavrayışı, onu takip eden `for` bağlamında değerlendirilir, bu nedenle bu örnek şuna eş değerdir:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

ki bu da şununla aynıdır:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Gerçek dünyada, karmaşık akışlı ifadeler yerine yerleşik işlevleri tercih etmelisiniz. Bu kullanım durumu için `zip()` fonksiyonu harika bir iş çıkaracaktır:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Bu satırdaki yıldız işaretiyle ilgili ayrıntılar için *Argüman Listelerini Açma* bölümüne bakın.

5.2 `del` ifadesi

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` değişkenlerin tamamını silmek için de kullanılabilir:

```
>>> del a
```

Bundan sonra `a` ismine referans vermek bir hatadır (en azından ona başka bir değer atanana kadar). Daha sonra `del` için başka kullanımlar bulacağız.

5.3 Veri Grupları ve Diziler

Listelerin ve dizelerin indeksleme ve dilimleme işlemleri gibi birçok ortak özelliğe sahip olduğunu gördük. Bunlar *dizi* veri tiplerinin iki örneğidir (bkz. typesseq). Python gelişen bir dil olduğu için başka dizi veri tipleri de eklenebilir. Ayrıca başka bir standart dizi veri tipi daha vardır: *tuple*.

Bir veri grubu, virgülle ayrılmış bir dizi değerden oluşur, örneğin:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Gördüğünüz gibi, çıktıda veri grupları her zaman parantez içine alınır, böylece iç içe geçmiş veri grupları doğru şekilde yorumlanır; parantezli veya parantezsiz olarak girilebilirler, ancak genellikle parantezler gereklidir (eğer grup daha büyük bir ifadenin parçasıysa). Bir veri grubunun öğelerine tek tek atama yapmak mümkün değildir, ancak listeler gibi değiştirilebilir nesneler içeren veri grupları oluşturmak mümkündür.

Veri grupları listelere benzer görünse de, genellikle farklı durumlarda ve farklı amaçlar için kullanılırlar. Veri grupları *immutable* 'dır ve genellikle paket açma (bu bölümün ilerleyen kısımlarına bakınız) veya indeksleme (hatta `namedtuples` durumunda öznitelik ile) yoluyla erişilen heterojen bir dizi eleman içerir. Listeler *mutable* 'dır ve elemanları genellikle homojendir ve listenin üzerinde yinelenerek erişilir.

Özel bir sorun, 0 veya 1 öğe içeren veri gruplarının oluşturulmasıdır: söz diziminin bunlar ile başa çıkan bazı ekstra tuhaflıkları vardır. Boş veri grupları boş bir parantez çifti ile oluşturulur; bir öğeli bir veri grubu, bir değeri virgülle takip ederek oluşturulur (tek bir değeri parantez içine almak yeterli değildir). Çirkin olsa da etkilidir. Örneğin:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

`t = 12345, 54321, 'hello!'` ifadesi bir *veri grubu paketleme* örneğidir: `12345`, `54321` ve `'hello!'` değerleri bir veri grubu içinde bir araya getirilmiştir. Bu işlemin tersi de mümkündür:

```
>>> x, y, z = t
```

Buna uygun bir şekilde *dizi açma (sequence unpacking)* denir ve sağ taraftaki herhangi bir dizi için çalışır. Dizi açma, eşittir işaretinin sol tarafında dizideki eleman sayısı kadar değişken olmasını gerektirir. Çoklu atamanın aslında tuple paketleme ve dizi açmanın bir kombinasyonu olduğunu unutmayın.

5.4 Kümeler

Python ayrıca *kümeler* için bir veri türü içerir. Bir küme, yinelenen öğeleri olmayan sırasız bir koleksiyondur. Temel kullanımları içerisinde üyelik testi ve yinelenen girdilerin elenmesi yer alır. Küme nesneleri ayrıca birleşim, kesişim, fark ve simetrik fark gibi matematiksel işlemleri de destekler.

Küme oluşturmak için küme parantezleri veya `set()` fonksiyonu kullanılabilir. Not: boş bir küme oluşturmak için `{}` değil `set()` kullanmanız gerekir, çünkü birincisi boş bir sözlük oluşturur, ki bu da bir sonraki bölümde tartışacağımız bir veri yapısıdır.

İşte kısa bir gösterim:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

liste kavramaları gibi küme kavramaları da desteklenmektedir:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5 Sözlükler

Another useful data type built into Python is the *dictionary* (see typesmapping). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

Bir sözlüğü *anahtar: değer* çiftleri olarak düşünmek en iyisidir. Bir sözlük içerisindeki her anahtarın benzersiz olması gerektiğini ise unutmayın. Bir çift parantez boş bir sözlük oluşturur: `{}`. Anahtar:değer çiftlerinin virgülle ayrılmış bir listesini parantezler içine yerleştirmek sözlüğe ilk anahtar:değer çiftlerini ekler; sözlükler çıktıda da aynı bu şekilde görünürler.

Bir sözlük üzerindeki ana işlemler, bir değeri bir anahtarla depolamak ve anahtarı verilen değeri geri çıkarmaktır. Ayrıca `del` ile bir anahtar:değer çiftini silmek de mümkündür. Zaten bir değeri kullanımda olan bir anahtar kullanarak saklarsanız, bu anahtarla ilişkili eski değer unutulur. Var olmayan bir anahtar kullanarak değer çıkarmak bir hatadır.

Bir sözlük üzerinde `list(d)` işlemini gerçekleştirmek, sözlükte kullanılan tüm anahtarların bir listesini eklenme sırasına göre döndürür (başka bir düzenle sıralanmasını istiyorsanız, bunun yerine `sorted(d)` kullanın). Tek bir anahtarın sözlükte olup olmadığını kontrol etmek için `in` anahtar sözcüğünü kullanın.

İşte sözlük kullanılan bir örnek:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

`dict()` yapıcısı, doğrudan anahtar-değer dizilerinden sözlükler oluşturur:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Buna ek olarak, sözlük kavramaları rastgele anahtar ve değer ifadelerinden sözlükler oluşturmak için de kullanılabilir:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Anahtarlar basit dizgiler olduğunda, anahtar sözcük argümanlarını kullanarak çiftleri belirtmek bazen daha kolaydır:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Döngü Teknikleri

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Bir dizi boyunca döngü yaparken, `enumerate()` fonksiyonu kullanılarak konum indeksi ve karşılık gelen değer aynı anda alınabilir.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
0 tic
1 tac
2 toe
```

Aynı anda iki veya daha fazla dizi üzerinde döngü yapmak için, girdiler `zip()` fonksiyonu ile eşleştirilebilir.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Bir dizi üzerinde ters yönde döngü yapmak için, önce diziyi ileri yönde belirtin ve ardından `reversed()` fonksiyonunu çağırın.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Bir dizi üzerinde sıralı olarak döngü yapmak için, listenin kaynağını değiştirmeksizin yeni bir sıralı liste döndüren `sorted()` fonksiyonunu kullanın.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Bir dizi üzerinde `set()` kullanmak yinelenen öğeleri ortadan kaldırır. Bir dizi üzerinde `set()` ile birlikte `sorted()` kullanımı, dizinin benzersiz öğeleri üzerinde sıralı olarak döngü oluşturmanın deyimsel bir yoludur.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Bazen bir liste üzerinde döngü yaparken değiştirmek cazip gelebilir; ancak bunun yerine yeni bir liste oluşturmak genellikle daha basit ve daha güvenlidir.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Koşullar Üzerine

`while` ve `if` deyimlerinde kullanılan koşullar sadece karşılaştırma değil, herhangi bir operatör içerebilir.

Karşılaştırma operatörleri `in` ve `not in` bir değerin bir dizide olup olmadığını kontrol eder. `is` ve `is not` operatörleri iki nesnenin gerçekten aynı nesne olup olmadığını karşılaştırır. Tüm karşılaştırma operatörleri aynı önceliğe sahiptir, bu öncelik ise tüm sayısal operatörlerden daha düşüktür.

Karşılaştırmalar art arda zincirlenebilir. Örneğin, `a < b == c` ifadesi `a` değerinin `b` değerinden küçük olup olmadığını test ederken aynı zamanda `b` değerinin `c` değerine eşit olup olmadığını test eder.

Karşılaştırmalar `and` ve `or` Boolean operatörleri kullanılarak birleştirilebilir ve bir karşılaştırmanın (veya başka bir Boolean ifadesinin) sonucu `not` ile olumsuzlanabilir. Bunlar karşılaştırma operatörlerinden daha düşük önceliklere sahiptir; aralarında, `not` en yüksek önceliğe ve `or` en düşük önceliğe sahiptir, böylece `A ve B değil veya C`, `(A ve (B değil)) veya C` ile eş değerdir. Her zaman olduğu gibi, istenen bileşimi ifade etmek için parantezler kullanılabilir.

Boolean operatörleri `and` ve `or` *kısa devre* operatörleri olarak adlandırılır: argümanları soldan sağa doğru değerlendirilir ve sonuç belirlenir belirlenmez değerlendirme durur. Örneğin, `A` ve `C` doğru ancak `B` yanlış ise, `A ve B ve C` `C` ifadesini değerlendirmez. Boolean olarak değil de genel bir değer olarak kullanıldığında, kısa devre işlecinin dönüş değeri son değerlendirilen bağımsız değişkendir.

Bir değişkene, bir karşılaştırmanın sonucunu veya başka bir Boolean ifadesini atamak mümkündür. Örneğin:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Python'da, C'den farklı olarak, ifadelerin içindeki atamanın walrus operatörü = ile açıkça yapılması gerektiğini unutmayın. Bu, C programlarında karşılaşılan yaygın bir sorunu önler: == yazmak isterken = yazmak.

5.8 Diziler ile Diğer Veri Tiplerinin Karşılaştırılması

Dizi nesneleri tipik olarak aynı dizi türüne sahip diğer nesnelerle karşılaştırılabilir. Karşılaştırmada *sözlüksel* sıralama kullanılır: önce ilk iki öğe karşılaştırılır ve farklılarsa bu karşılaştırmanın sonucunu belirler; eşitlerse, sonraki iki öğe karşılaştırılır ve her iki dizi de tükenene kadar böyle devam eder. Karşılaştırılacak iki öğenin kendileri de aynı türden diziler ise, sözlükbilimsel karşılaştırma özyinelemeli olarak gerçekleştirilir. İki dizinin tüm öğeleri eşit çıkarsa, diziler eşit kabul edilir. Eğer dizilerden biri diğerinin alt dizisi ise, daha kısa olan dizi daha küçük (küçük) olandır. Dizgiler için sözlükbilimsel sıralama, tek tek karakterleri sıralamak için Unicode kod noktası numarasını kullanır. Aynı türdeki diziler arasındaki karşılaştırmalara bazı örnekler:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Nesnelerin uygun karşılaştırma yöntemlerine sahip olması koşuluyla, farklı türlerdeki nesnelerin < veya > ile karşılaştırılabileceğini unutmayın. Örneğin, karışık sayısal türler sayısal değerlerine göre karşılaştırılır, bu nedenle 0 eşittir 0.0, vb. Bu türler uygun yöntemlere sahip değillerse, yorumlayıcı rastgele bir sıralama döndürmek yerine `TypeError` hatası verir.

BÖLÜM 6

Modüller

Python yorumlayıcısından çıkıp tekrar girerseniz, yaptığınız tanımlar (fonksiyonlar ve değişkenler) kaybolur. Bu nedenle, daha uzun bir program yazmak istiyorsanız, girdiyi yorumlayıcıya hazırlarken bir metin düzenleyicisi kullanmak ve o dosyayla girdi olarak çalıştırmak daha iyidir. Bu bir *script* oluşturma olarak bilinir. Programınız uzadıkça, daha kolay bakım için birkaç dosyaya bölmek isteyebilirsiniz. Ayrıca, tanımını her programa kopyalamadan, birkaç programda yazdığınız kullanışlı bir fonksiyonu kullanmak isteyebilirsiniz.

Bunu desteklemek için Python, tanımları bir dosyaya koymanın ve bunları bir komut dosyasında veya yorumlayıcının etkileşimli bir örneğinde kullanmanın bir yolunu sağlar. Böyle bir dosyaya *module* denir; bir modülden alınan tanımlar diğer modüllere veya *main* modülüne (en üst düzeyde ve hesap makinesi modunda yürütülen bir komut dosyasında erişiminiz olan değişkenlerin derlenmesi) aktarılabilir.

Modül, Python tanımlarını ve ifadelerini içeren bir dosyadır. Dosya adı, `.py` son ekini içeren modül adıdır. Bir modül içinde, modülün adı (dize olarak) `__name__` genel değişkeninin değeri olarak kullanılabilir. Örneğin, geçerli dizinde aşağıdaki içeriklerle `fibo.py` adlı bir dosya oluşturmak için en sevdiğiniz metin düzenleyicisini kullanın:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Şimdi Python yorumlayıcısına girin ve bu modülü aşağıdaki komutla içe aktarın:

```
>>> import fibo
```

Bu, `fibo` 'da tanımlanan işlevlerin adlarını doğrudan geçerli *namespace* 'e eklemez (daha fazla ayrıntı için bkz. *Python Etki Alanları ve Ad Alanları*); oraya sadece `fibo` modül adını ekler. Modül adını kullanarak şu işlevlere erişebilirsiniz:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Bir işlevi sık sık kullanmayı düşünüyorsanız, işlevi yerel bir ada atayabilirsiniz:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Modüller hakkında daha fazla

Bir modül, işlev tanımlarının yanı sıra çalıştırılabilir ifadeler de içerebilir. Bu ifadeler modülü başlatmayı amaçlamaktadır. Yalnızca bir import ifadesinde modül adıyla karşılaşıldığında *ilk* kez yürütülürler.[1] (Dosya komut dosyası olarak yürütülürse de çalıştırılırlar.)

Her modülün, modülde tanımlanan tüm işlevler tarafından genel ad alanı olarak kullanılan kendi özel ad alanı vardır. Böylece, bir modülün yazarı, bir kullanıcının genel değişkenleriyle yanlışlıkla çakışma endişesi duymadan, modüldeki genel değişkenleri kullanabilir. Öte yandan, ne yaptığınızı biliyorsanız, bir modülün global değişkenlerine, işlevlerine atıfta bulunmak için kullanılan `modname.itemname` notasyonuyla dokunabilirsiniz.

Modüller diğer modülleri içe aktarabilir. Tüm `import` ifadelerinin bir modülün (veya bu konuda betiğin) başına yerleştirilmesi alışılmış bir durumdur ancak gerekli değildir. İçe aktarılan modül adları, bir modülün en üst düzeyine yerleştirilirse (herhangi bir işlev veya sınıfın dışında), modülün genel ad alanına eklenir.

`import` ifadesinin, bir modülden adları doğrudan içe aktaran modülün ad alanına aktaran bir çeşidi vardır. Örneğin:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Bu, içe aktarmaların yerel ad alanında alındığı modül adını tanıtmaz(bu nedenle örnekte `fibo` tanımlanmamıştır).

Bir modülün tanımladığı tüm adları içe aktarmak için bir varyant bile vardır:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Bu, alt çizgiyle başlayanlar (_) dışındaki tüm isimleri alır. Çoğu durumda Python programcıları bu özelliği kullanmaz, çünkü yorumlayıcıya bilinmeyen bir ad kümesi ekler ve muhtemelen önceden tanımladığınız bazı şeyleri gizler.

Genel olarak, bir modülden veya paketten `*` içeri aktarma uygulamasının, genellikle okunamayan koda neden olduğundan hoş karşılanmadığına dikkat edin. Ancak, etkileşimli oturumlarda yazmayı kaydetmek için kullanmak sorun değildir.

Modül adının ardından `as` geliyorsa, `as` 'den sonraki ad doğrudan içe aktarılan modüle bağlanır.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Bu, modülün `import fibo` 'nun yapacağı şekilde etkin bir şekilde içe aktarılmasıdır, tek farkı `fib` olarak mevcut olmasıdır.

Benzer efektlere sahip `from` kullanılırken de kullanılabilir:

[1] Aslında işlev tanımları aynı zamanda 'çalıştırılan' 'ifadelerdir'; modül düzeyinde bir işlev tanımının çalıştırılması, işlev adını modülün genel ad alanına ekler.

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Not

Verimlilik nedeniyle, her modül yorumlayıcı oturumu başına yalnızca bir kez içe aktarılır. Bu nedenle, modüllerinizi değiştirirseniz, yorumlayıcıyı yeniden başlatmanız gerekir - veya etkileşimli olarak test etmek istediğiniz tek bir modülse, `importlib.reload()`, örneğin `importlib; importlib.reload(modulename)`.

6.1.1 Modülleri komut dosyası olarak yürütme

Bir Python modülünü :: ile çalıştırdığınızda:

```
python fibo.py <arguments>
```

modüldeki kod, içe aktardığınız gibi yürütülür, ancak `__name__` `"__main__"` olarak ayarlanır. Bu, modülünüzün sonuna bu kodu ekleyerek:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

dosyayı bir komut dosyası ve içe aktarılabilir bir modül olarak kullanılabilir hale getirebilirsiniz, çünkü komut satırını ayrıştıran kod yalnızca modül "main" dosya olarak yürütülürse çalışır:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Modül içe aktarılırsa kod çalıştırılmaz:

```
>>> import fibo
>>>
```

Bu genellikle bir modüle uygun bir kullanıcı arabirimi sağlamak veya test amacıyla kullanılır (modülü komut dosyası olarak çalıştırmak bir test paketi yürütür).

6.1.2 Modül Arama Yolu

`spam` adında bir modül içe aktarıldığında, yorumlayıcı ilk olarak bu ada sahip gömülü bir modül arar. Bu modül adları `sys.builtin_module_names` içinde listelenir. Bulamazsa, `sys.path` değişkeni tarafından verilen dizin listesinde `spam.py` adlı bir dosya arar. `sys.path` bu konumlardan başlatılır:

- Girdi komut dosyasını içeren dizin (veya dosya belirtilmediğinde geçerli dizin).
- `PYTHONPATH` (kabuk değişkeni `PATH` ile aynı sözdizimine sahip dizin adlarının listesi).
- Kuruluma bağlı varsayılan (kural olarak, `site` modülü tarafından işlenen bir "site-packages" dizini dahil).

Daha fazla ayrıntı sys-path-init adresindedir.

Not

Symlink'leri destekleyen dosya sistemlerinde, symlink izlendikten sonra girdi komut dosyasını içeren dizin hesaplanır. Başka bir deyişle, symlink içeren dizin modül arama yoluna **not** eklenir.

Başlatıldıktan sonra Python programları `sys.path` değiştirebilir. Çalıştırılmakta olan komut dosyasını içeren dizin, arama yolunun başına, standart kitaplık yolunun önüne yerleştirilir. Bu, kitaplık dizininde aynı ada sahip modüller

yerine bu dizindeki komut dosyalarının yüklanacağı anlamına gelir. Değiştirme amaçlanmadığı sürece bu bir hatadır. Daha fazla bilgi için *Standart modüller* bölümüne bakın.

6.1.3 "Derlenmiş" Python dosyaları

Modüllerin yüklenmesini hızlandırmak için Python, her modülün derlenmiş sürümünü `__pycache__` dizininde `module.version.pyc` adı altında önbelleğe alır; burada sürüm, derlenen dosyanın biçimini kodlar; genellikle Python sürüm numarasını içerir. Örneğin, CPython 3.3 sürümünde spam.py'nin derlenmiş sürümü `__pycache__/spam.cpython-33.pyc` olarak önbelleğe alınacaktır. Bu adlandırma kuralı, farklı sürümlerden ve Python'un farklı sürümlerinden derlenmiş modüllerin bir arada var olmasına izin verir.

Python, eski olup olmadığını ve yeniden derlenmesi gerekip gerekmediğini görmek için kaynağın değişiklik tarihini derlenmiş sürümle karşılaştırır. Bu tamamen otomatik bir işlemdir. Ayrıca, derlenen modüller platformdan bağımsızdır, bu nedenle aynı kitaplık farklı mimarilere sahip sistemler arasında paylaşılabilir.

Python iki durumda önbelleği kontrol etmez. İlk olarak, doğrudan komut satırından yüklenen modülün sonucunu her zaman yeniden derler ve saklamaz. İkincisi, kaynak modül yoksa önbelleği kontrol etmez. Kaynak olmayan (yalnızca derlenmiş) bir dağıtımı desteklemek için, derlenen modül kaynak dizinde olmalı ve bir kaynak modül olmamalıdır.

Uzmanlar için bazı ipuçları:

- Derlenmiş bir modülün boyutunu küçültmek için Python komutundaki `-O` veya `-OO` anahtarlarını kullanabilirsiniz. `-O` anahtarı, onaylama ifadelerini kaldırır, `-OO` anahtarı, hem assert ifadelerini hem de __doc__ dizelerini kaldırır. Bazı programlar bunların kullanılabilir olmasına güvenebileceğinden, bu seçeneği yalnızca ne yaptığınızı biliyorsanız kullanmalısınız. "Optimize edilmiş" modüller bir "opt-" etiketine sahiptir ve genellikle daha küçüktür. Gelecekteki sürümler, optimizasyonun etkilerini değiştirebilir.
- Bir program `.pyc` dosyasından okunduğunda, `.py` dosyasından okunduğundan daha hızlı çalışmaz; `.pyc` dosyaları hakkında daha hızlı olan tek şey, yüklenme hızlarıdır.
- `compileall` modülü, bir dizindeki tüm modüller için .pyc dosyaları oluşturabilir.
- **PEP 3147** 'de, kararların bir akış şeması da dahil olmak üzere, bu süreç hakkında daha fazla ayrıntı bulunmaktadır.

6.2 Standart modüller

Python, ayrı bir belge olan Python Kütüphane Referansında (buradan itibaren "Kütüphane Referansı") açıklanan standart modüllerden oluşan bir kütüphaneyle birlikte gelir. Bazı modüller yorumlayıcıda yerleşik olarak bulunur; bunlar dilin çekirdeğinin bir parçası olmayan ancak yine de verimlilik için veya sistem çağrıları gibi işletim sistemi ilkellerine erişim sağlamak için yerleşik olarak bulunan işlemlere erişim sağlar. Bu tür modüllerin kümesi, altta yatan platforma da bağlı olan bir yapılandırma seçeneğidir. Örneğin, `winreg` modülü yalnızca Windows sistemlerinde sağlanır. Belirli bir modül biraz ilgiyi hak ediyor: `sys`, her Python yorumlayıcısında yerleşik olarak bulunur. `sys.ps1` ve `sys.ps2` değişkenleri birincil ve ikincil istemler olarak kullanılan dizeleri tanımlar:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Bu iki değişken yalnızca yorumlayıcı etkileşimli moddaysa tanımlanır.

`sys.path` değişkeni, yorumlayıcının modüller için arama yolunu belirleyen bir dizeler listesidir. `PYTHONPATH` ortam değişkeninden veya `PYTHONPATH` ayarlanmamışsa yerleşik bir varsayılan değerden alınan varsayılan bir yola başlatılır. Standart liste işlemlerini kullanarak değiştirebilirsiniz:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 dir() Fonksiyonu

Yerleşik fonksiyon `dir()`, bir modülün hangi adları tanımladığını bulmak için kullanılır. Sıralanmış bir dize listesi döndürür:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Argümanlar olmadan, `dir()`, şu anda tanımladığınız adları listeler:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Her tür adın listelendiğini unutmayın: değişkenler, modüller, fonksiyonlar vb.

`dir()` yerleşik fonksiyonlarun ve değişkenlerin adlarını listelemez. Bunların bir listesini istiyorsanız, standart modül `builtins` 'de tanımlanırlar:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

6.4 Paketler

Paketler, Python'un modül isim alanını "noktalı modül isimleri" kullanarak yapılandırmanın bir yoludur. Örneğin, `A.B` modül adı, `A` adlı bir paketteki `B` adlı bir alt modülü belirtir. Modül kullanımının farklı modüllerin yazarlarını birbirlerinin global değişken isimleri hakkında endişelenmekten kurtarması gibi, noktalı modül isimlerinin kullanımı da NumPy veya Pillow gibi çok modüllü paketlerin yazarlarını birbirlerinin modül isimleri hakkında endişelenmekten kurtarır.

Ses dosyalarının ve ses verilerinin tek tip işlenmesi için bir modül koleksiyonu ("paket") tasarlamak istediğinizi varsayalım. Birçok farklı ses dosyası biçimi vardır (genellikle uzantılarıyla tanınır, örneğin: `.wav`, `.aiff`, `.au`) bu nedenle, çeşitli dosya biçimleri arasında dönüşüm için büyüyen bir modül koleksiyonu oluşturmanız ve sürdürmeniz gerekebilir. Ses verileri üzerinde gerçekleştirmek isteyebileceğiniz birçok farklı işlem de vardır (karıştırma, eko ekleme, ekolayzır işlevi uygulama, yapay bir stereo efekti oluşturma gibi) bu nedenle ek olarak, bu işlemleri gerçekleştirmek için hiç bitmeyen bir modül akışı yazıyor olacaksınız. İşte paketiniz için olası bir yapı (hiyerarşik bir dosya sistemi cinsinden ifade edilir):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Paketi içe aktarırken Python, paket alt dizinini arayan `sys.path` üzerindeki dizinleri arar.

`__init__.py` dosyaları, Python'un dosyayı içeren dizinleri paket olarak ele almasını sağlamak için gereklidir (nispeten gelişmiş bir özellik olan *namespace package* kullanılmadığı sürece). Bu, `string` gibi ortak bir ada sahip dizinlerin, modül arama yolunda daha sonra ortaya çıkan geçerli modülleri istemeden gizlemesini önler. En basit durumda, `__init__.py` sadece boş bir dosya olabilir, ancak aynı zamanda paket için başlatma kodunu çalıştırabilir veya daha sonra açıklanacak olan `__all__` değişkenini ayarlayabilir.

Paketin kullanıcıları, paketin içindeki ayrı modülleri içe aktarabilir, örneğin:

```
import sound.effects.echo
```

Bu, `sound.effects.echo` alt modülünü yükler. Tam adı ile referans verilmelidir.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alt modülü içe aktarmanın alternatif bir yolu:

```
from sound.effects import echo
```

Bu aynı zamanda `echo` alt modülünü yükler ve paket öneki olmadan kullanılabilir hale getirir, böylece aşağıdaki gibi kullanılabilir:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yine başka bir varyasyon, istenen işlevi veya değişkeni doğrudan içe aktarmaktır:

```
from sound.effects.echo import echofilter
```

Yine, bu, `echo` alt modülünü yükler, ancak bu, `echofilter()` fonksiyonunu doğrudan kullanılabilir hale getirir:

```
echofilter(input, output, delay=0.7, atten=4)
```

`from package import item` kullanırken, öğenin paketin bir alt modülü (veya alt paketi) veya pakette tanımlanmış bir fonksiyon, sınıf veya değişken gibi başka bir ad olabileceğini unutmayın. `import` ifadesi önce öğenin pakette tanımlanıp tanımlanmadığını test eder; değilse modül olduğunu varsayar ve yüklemeye çalışır. Onu bulamazsa, bir `ImportError` istisnası ortaya çıkar.

Aksine, `import item.subitem.subsubitem` gibi bir sözdizimi kullanırken, sonuncusu hariç her öğe bir paket olmalıdır; son öğe bir modül veya paket olabilir, ancak önceki öğede tanımlanan bir sınıf, fonksiyon veya değişken olamaz.

6.4.1 Bir Paketten * İçe Aktarma

Şimdi, kullanıcı `from sound.effects import *` yazdığında ne olur? İdeal olarak, bunun bir şekilde dosya sistemine gitmesi, pakette hangi alt modüllerin bulunduğunu bulması ve hepsini içe aktarması umulur. Bu uzun zaman alabilir ve alt modüllerin içe aktarılması, yalnızca alt modül açıkça içe aktarıldığında gerçekleşmesi gereken istenmeyen yan etkilere neden olabilir.

Tek çözüm, paket yazarının paketin açık bir dizinini sağlamasıdır. `import` ifadesi aşağıdaki kuralı kullanır: eğer bir paketin `__init__.py` kodu `__all__` adlı bir liste tanımlarsa, `from package import *` ile karşılaşıldığında

alınması gereken modül adlarının listesi olarak alınır. Paketin yeni bir sürümü yayınlandığında bu listeyi güncel tutmak paket yazarının sorumluluğundadır. Paket yazarları, paketlerinden * içe aktarmak için bir kullanım görmezlerse, onu desteklememeye de karar verebilirler. Örneğin, `sound/effects/__init__.py` dosyası şu kodu içerebilir:

```
__all__ = ["echo", "surround", "reverse"]
```

Bu, `from sound.effects import *` ifadesinin `sound.effects` paketinin üç adlandırılmış alt modülünü içe aktaracağı anlamına gelir.

Alt modüllerin yerel olarak tanımlanmış isimler tarafından gölgelenebileceğini unutmayın. Örneğin, `sound/effects/__init__.py` dosyasına bir `reverse` fonksiyonu eklediyseniz, `from sound.effects import *` sadece iki alt modül olan `echo` ve `surround``u içe aktarır, ancak *reverse` alt modülünü içe aktarmaz, çünkü yerel olarak tanımlanmış `reverse` fonksiyonu tarafından gölgelenir:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str):  # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]    #     in the case of a 'from sound.effects import *'
```

Eğer `__all__` tanımlanmamışsa, `from sound.effects import *` ifadesi `sound.effects` paketindeki tüm alt modülleri geçerli isim alanına import etmez; sadece `sound.effects` paketinin import edildiğinden emin olur (muhtemelen `__init__.py` içindeki herhangi bir başlatma kodunu çalıştırır) ve sonra pakette tanımlanan isimleri import eder. Bu, `__init__.py` tarafından tanımlanan (ve alt modülleri açıkça yüklenen) tüm isimleri içerir. Ayrıca, önceki `import` deyimleri tarafından açıkça yüklenmiş olan paketin tüm alt modüllerini de içerir. Bu kodu göz önünde bulundurun:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Bu örnekte, `echo` ve `surround` modülleri, `from...import` deyimi çalıştırıldığında `sound.effects` paketinde tanımlandıkları için geçerli ad alanında içe aktarılır. (Bu aynı zamanda `__all__` tanımlandığında da çalışır).

Bazı modüller, `import *` kullandığınızda yalnızca belirli kalıpları takip eden adları dışa aktarmak üzere tasarlanmış olsa da, üretim kodunda yine de kötü uygulama olarak kabul edilir.

Unutmayın, `from package import specific_submodule` kullanmanın yanlış bir tarafı yok! Aslında, içe aktarma modülünün farklı paketlerden aynı ada sahip alt modülleri kullanması gerekmedikçe, önerilen gösterim budur.

6.4.2 Paket İçi Referanslar

Paketler alt paketler halinde yapılandırıldığında (örnekteki `sound` paketinde olduğu gibi), kardeş paketlerin alt modüllerine başvurmak için mutlak içe aktarımları kullanabilirsiniz. Örneğin, `sound.filters.vocoder` modülünün `sound.effects` paketindeki `echo` modülünü kullanması gerekiyorsa, `from sound.effects import echo` kullanabilir.

Ayrıca, içe aktarma ifadesinin `from module import name` formuyla göreli içe aktarmaları(relative import) da yazabilirsiniz. Bu içe aktarmalar, göreli içe aktarmada(relative import) yer alan mevcut ve ana paketleri belirtmek için baştaki noktaları kullanır. Örneğin `surround` modülünden şunları kullanabilirsiniz:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Göreceli içe aktarmaların geçerli modülün adını temel aldığını unutmayın. Ana modülün adı her zaman `"__main__"` olduğundan, bir Python uygulamasının ana modülü olarak kullanılması amaçlanan modüller her zaman mutlak içe aktarma kullanmalıdır.

6.4.3 Birden Çok Dizindeki Paketler

Packages support one more special attribute, `__path__`. This is initialized to be a *sequence* of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

Bu özelliğe sıklıkla ihtiyaç duyulmasa da, bir pakette bulunan modül dizisini genişletmek için kullanılabilir.

BÖLÜM 7

Girdi ve Çıktı

Bir programın çıktısını sunmanın birkaç yolu vardır; veriler okunabilir bir biçimde yazdırılabilir veya ileride kullanılmak üzere bir dosyaya yazılabilir. Bu bölümde bazı olasılıklar tartışılacaktır.

7.1 Güzel Çıktı Biçimlendirmesi

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Genellikle, yalnızca boşlukla ayrılmış değerleri yazdırmaktansa, çıktınızın biçimlendirmesi üzerinde daha fazla denetim istersiniz. Çıktıyı biçimlendirmenin birkaç yolu vardır.

- *formatted string literals* kullanmak için, açılış tırnak işaretinden veya üç tırnak işaretinden önce `f` veya `F` ile bir dize başlatın. Bu dizenin içinde, değişkenlere veya hazır bilgi değerlerine başvurabilen `{` ve `}` karakterleri arasında bir Python ifadesi yazabilirsiniz.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

  Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see formatspec for details).

- Son olarak, hayal edebileceğiniz herhangi bir düzen oluşturmak için dize dilimleme ve birleştirme işlemlerini kullanarak tüm dize işlemeyi kendiniz yapabilirsiniz. Dize türü, dizeleri belirli bir sütun genişliğine doldurma için yararlı işlemler gerçekleştiren bazı yöntemlere sahiptir.

Daha güzel görünen bir çıktıya ihtiyacınız olmadığında, ancak hata ayıklama amacıyla bazı değişkenlerin hızlı bir şekilde görüntülenmesini istediğinizde, herhangi bir değeri `repr()` veya `str()` işlevleriyle bir dizeye dönüştürebilirsiniz.

`str()` işlevi açıkça okunabilir değerlerin gösterimlerini döndürmek için, `repr()` ise yorumlayıcı tarafından okunabilecek gösterimler oluşturmak içindir (veya eş değer bir sözdizimi yoksa `SyntaxError` 'ı zorlar). İnsan tüketimi için belirli bir temsili olmayan nesneler için `str()`, `repr()` ile aynı değeri döndürür. Sayılar veya listeler ve sözlükler benzeri yapılar gibi birçok değer, her iki işlevi de kullanarak aynı gösterime sahiptir. Özellikle dizelerin iki farklı gösterimi vardır.

Bazı örnekler:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

`string` modülü bir `Template` sınıfını içerir; bu sınıf, `$x` gibi yer tutucuları kullanarak ve bunları bir sözlükten değerlerle değiştirerek, değerleri dizelerle değiştirmenin başka bir yolunu sunar, ancak biçimlendirme üzerinde çok daha az kontrol sağlar.

7.1.1 Biçimlendirilmiş Dize Değişmezleri

Formatted string literals (kısaltmak için f-string olarak da adlandırılır), dizenin önüne `f` veya `F` yazarak ve ifadeleri `{expression}` olarak yazarak Python ifadelerinin değerini bir dizenin içine eklemenize olanak tanır.

Opsiyonel biçim belirleyicisi ifadeyi izleyebilir. Bu, değerin nasıl biçimlendirileceğini daha fazla denetlemenizi sağlar. Aşağıdaki örnek pi sayısını ondalık sayıdan sonra üç basamağa yuvarlar:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

`':'` öğesinin ardından bir tamsayı geçmek, bu alanın en az sayıda karakter genişliğinde olmasına neden olur. Bu, sütunların hizaya getirilmesi için yararlıdır.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Diğer değiştiriciler, değeri biçimlendirilmeden önce dönüştürmek için kullanılabilir. `'!a'` `ascii()`, `'!s'` `str()`, ve `'!r'` `repr()` uygular:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

= belirleyicisi, bir ifadeyi ifadenin metnine, eşittir işaretine ve ardından değerlendirilen ifadenin temsiline genişletmek için kullanılabilir:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs ='roaches' count =13 area ='living room'
```

= belirtici hakkında daha fazla bilgi için kendi kendini belgeleyen ifadeler konusuna bakın. Bu biçim belirtimleriyle ilgili bir referans için, formatspec için başvuru kılavuzuna bakın.

7.1.2 String format() Metodu

`str.format()` metodunun temel kullanımı şöyle görünür:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

İçlerindeki köşeli ayraçlar ve karakterler (format fields olarak adlandırılır) `str.format()` yöntemine geçirilen nesnelerle değiştirilir. Köşeli ayraçlardaki bir sayı, `str.format()` yöntemine geçirilen nesnenin konumuna başvurmak için kullanılabilir.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Anahtar sözcük argümanları `str.format()` yönteminde kullanılıyorsa, değerlerine argümanın adı kullanılarak başvurulmaktadır.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Konumsal ve anahtar sözcük argümanları isteğe bağlı olarak birleştirilebilir:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

Bölmek istemediğiniz gerçekten uzun biçimli bir dizeniz varsa, konuma göre değil de ada göre biçimlendirilecek değişkenlere başvurursanız iyi olur. Bu, sadece dict'i geçirerek ve tuşlara erişmek için `'[]'` köşeli ayraçları kullanarak yapılabilir.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Bu, `table` sözlüğünü `**` gösterimiyle anahtar kelime bağımsız değişkenleri olarak ileterek de yapılabilir.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k +'};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Örnek olarak, aşağıdaki satırlar, tamsayıları ve bunların karelerini ve küplerini veren düzenli bir şekilde hizalanmış bir sütun kümesi oluşturur:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

`str.format()` ile dize biçimlendirmeye tam bir genel bakış için bkz. formatstrings.

7.1.3 Manuel Dize Biçimlendirmesi

Manuel olarak biçimlendirilmiş aynı kare ve küp tablosu aşağıdadır:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Her sütun arasındaki tek boşluğun `print()` çalışma şekliyle ekli olduğunu unutmayın: her zaman argümanları arasına boşluk ekler.)

Dize nesnelerinin `str.rjust()` yöntemi, belirli bir genişlikteki bir alandaki dizeyi soldaki boşluklarla doldurmayı haklı hale getirir. Benzer yöntemler vardır `str.ljust()` ve `str.center()`. Bu yöntemler hiçbir şey yazmaz, yalnızca yeni bir dize döndürür. Giriş dizesi çok uzunsa, onu kesmezler, değiştirmeden döndürürler; bu, sütununuzu mahvedecektir, ancak bu genellikle bir değer hakkında yalan söylemek olan alternatiften daha iyidir. (Gerçekten kesilme istiyorsanız, `x.ljust(n)[:n]` gibi her zaman bir dilim işlemi ekleyebilirsiniz.)

Soldaki sayısal bir dizeyi sıfırlarla dolduran başka bir metot vardır: `str.zfill()`. Bu metot artı ve eksi işaretlerini anlar:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Eski dize biçimlendirmesi

The % operator (modulo) can also be used for string formatting. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Daha fazla bilgiyi old-string-formatting bölümünde bulabilirsiniz.

7.2 Dosyaları Okuma ve Yazma

`open()` bir *file object* döndürür ve en yaygın olarak iki konum argümanı ve bir anahtar sözcük argümanı ile kullanılır: `open(filename, mode, encoding =None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

İlk parametre dosya adını içeren bir dizedir. İkinci parametre dosyanın nasıl kullanılacağını açıklayan birkaç karakter içeren başka bir dizedir. *mode*, dosya yalnızca okunacağı zaman `'r'`, yalnızca yazmak için `'w'` olabilir (aynı ada sahip varolan bir dosya temizlenir) ve `'a'` dosyayı ekleme için açar; dosyaya yazılan tüm veriler otomatik olarak sonuna eklenir. `'r+'` dosyayı hem okumak hem de yazmak için açar. *mode* parametresi isteğe bağlıdır; verilmezse `'r'` varsayılacaktır.

Normalde dosyalar *text mode* 'unda açılır, yani belirli bir *kodlamada (encoding)* kodlanmış dizeleri dosyadan okur ve dosyaya yazarsınız. *kodlama* belirtilmezse, varsayılan değer platforma bağlıdır (bakınız `open()`). UTF-8 modern de-fakto standart olduğundan, farklı bir kodlama kullanmanız gerekmiyorsa `encoding ="utf-8"` kullanmanız önerilir. Moda `'b'` eklemek, dosyayı *binary* modunda açar. İkili mod verileri, `bytes` nesneleri olarak okunur ve yazılır. Dosyayı ikili modda açarken *kodlama* belirtemezsiniz.

Metin modunda, okurken varsayılan değer platforma özgü satır sonlarını (`\n` on Unix, `\r\n` on Windows) yalnızca `\n` olarak dönüştürmektir. Metin modunda yazarken, varsayılan değer `\n` oluşumlarını platforma özgü satır sonlarına geri dönüştürmektir. Dosya verilerinde yapılan bu sahne arkası değişikliği metin dosyaları için iyidir, ancak `JPEG` veya `EXE` dosyalarında bunun gibi ikili verileri bozacaktır. Bu tür dosyaları okurken ve yazarken ikili modu kullanmaya çok dikkat edin.

Dosya nesneleriyle uğraşırken `with` anahtar sözcüğünü kullanmak iyi bir uygulamadır. Avantajı, herhangi bir noktada bir hata oluşsa bile, paketi bittikten sonra dosyanın düzgün bir şekilde kapatılmasıdır. `with` kullanmak da eş değer `try-finally` blokları yazmaktançok daha kısadır:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

`with` anahtar sözcüğünü kullanmıyorsanız, dosyayı kapatmak ve kullandığı sistem kaynaklarını hemen boşaltmak için `f.close()` metodunu çağırmalısınız.

> **Uyarı**
>
> `with` anahtar sözcüğünü kullanmadan `f.write()` çağırmak veya `f.close()` çağırmak, program başarıyla çıksa bile `f.write()` parametrelerinin diske tamamen yazılmamasıyla sonuçlanabilir.

Bir dosya nesnesi kapatıldıktan sonra, bir `with` deyimiyle veya `f.close()` çağırarak dosya nesnesini kullanma girişimleri otomatik olarak başarısız olur.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Dosya Nesnelerinin Metotları

Bu bölümdeki örneklerin geri kalanı, `f` adlı bir dosya nesnesinin zaten oluşturulduğunu varsayar.

Bir dosyanın içeriğini okumak için, bir miktar veriyi okuyan ve dize (metin modunda) veya bayt nesnesi (ikili modda) olarak döndüren `f.read(size)` öğesini çağırın. *size* isteğe bağlı bir sayısal parametredir. *size* boş bırakıldığında veya negatif olduğunda, dosyanın tüm içeriği okunur ve döndürülür; dosya makinenizin belleğinden iki kat daha büyükse bu sizin sorununuzdur. Aksi takdirde, en fazla *size* karakterleri (metin modunda) veya *size* bayt (ikili modda) okunur ve döndürülür. Dosyanın sonuna ulaşıldıysa, `f.read()` boş bir dize (`''`) döndürür.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` dosyadan tek bir satır okur; dizenin sonunda bir newline karakteri (`\n`) bırakılır ve dosya yalnızca dosya yeni satırda bitmezse dosyanın son satırında atlanır. Bu, dönüş değerini netleştirir; `f.readline()` boş bir dize döndürürse, dosyanın sonuna ulaşılmış demektir, boş bir satır ise yalnızca tek bir yeni satır içeren bir dize olan `'\n'` ile temsil edilir.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Bir dosyadan satırları okumak için, dosya nesnesinin üzerinde döngü oluşturabilirsiniz. Bu bellek verimliliğine, hızlılığına ve basit koda yol açar:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Listedeki bir dosyanın tüm satırlarını okumak istiyorsanız, `list(f)` veya `f.readlines()` öğelerini de kullanabilirsiniz.

`f.write(string)` *string* içeriğini dosyaya yazar ve yazılan karakter sayısını döndürür.

```
>>> f.write('This is a test\n')
15
```

Diğer nesne türlerinin yazmadan önce bir dizeye (metin modunda) veya bayt nesnesine (ikili modda) dönüştürülmesi gerekir:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` dosya nesnesinin dosyadaki geçerli konumunu ikili moddayken dosyanın başından itibaren bayt sayısı ve metin modundayken opak bir sayı olarak veren bir tamsayı döndürür.

Dosya nesnesinin konumunu değiştirmek için `f.seek(offset, whence)` kullanın. Konum, bir referans noktasına *offset* eklenerek hesaplanır; referans noktası *whence* parametresi tarafından seçilir. *whence* değeri dosyanın başından itibaren 0 ölçerken, 1 geçerli dosya konumunu, 2 ise başvuru noktası olarak dosyanın sonunu kullanır. *whence* atlanabilir ve başvuru noktası için dosyanın başlangıcını kullanarak 0 olarak varsayılabilir.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Metin dosyalarında (mod dizesinde `b` olmadan açılanlar), yalnızca dosyanın başına göre aramalara izin verilir (dosyanın sonuna `seek(0, 2)` ile arayan özel durum) ve tek geçerli *offset* değerleri `f.tell()` veya sıfırdan döndürülen değerlerdir. Başka herhangi bir *offset* değeri tanımsız davranış üretir.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2 Yapılandırılmış verileri `json` ile kaydetme

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Python, kullanıcıların karmaşık veri türlerini dosyalara kaydetmek için sürekli olarak kod yazmasını ve hata ayıklamasını sağlamak yerine, JSON (JavaScript Object Notation) adlı popüler veri değişim biçimini kullanmanıza olanak tanır. `json` adlı standart modül, Python veri hiyerarşilerini alabilir ve bunları dize temsillerine dönüştürebilir; bu işleme *serializing* adı verilir. Dize gösteriminden verilerin yeniden yapılandırılmasına *deserializing* denir. Serileştirme ve seri durumdan çıkarma arasında, nesneyi temsil eden dizi bir dosyada veya veride saklanmış olabilir veya bir ağ bağlantısı üzerinden uzaktaki bir makineye gönderilmiş olabilir.

> **Not**
>
> JSON biçimi, veri alışverişine izin vermek için modern uygulamalar tarafından yaygın olarak kullanılır. Birçok programcı zaten buna aşinadır, bu da onu birlikte çalışabilirlik için iyi bir seçim haline getirir.

`x` nesneniz varsa, JSON dize gösterimini basit bir kod satırıyla görüntüleyebilirsiniz:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

`dumps()` işlevinin başka bir çeşidi, `dump()` adı verilen nesneyi bir *text file* (metin dosyası) olarak seri hale getirmektedir. Yani `f` bir *text file* nesnesi yazmak için açılmışsa, bunu yapabiliriz:

```
json.dump(x, f)
```

Nesnenin kodunu tekrar çözmek için, `f` okuma için açılmış bir *binary file* veya *text file* nesnesiyse:

```
x = json.load(f)
```

Not

JSON dosyaları UTF-8'de kodlanmalıdır. Hem okuma hem de yazma için JSON dosyasını *text file* olarak açarken `encoding ="utf-8"` kullanın.

Bu basit seri hale getirme tekniği listeleri ve sözlükleri işleyebilir, ancak JSON'da rasgele sınıf örneklerini seri hale getirmek biraz daha fazla çaba gerektirir. `json` modülü için olan örnek bunun bir açıklamasını içerir.

Ayrıca bakınız

`pickle` - pickle modülü

JSON ifadesinin aksine, *pickle*, gelişigüzel olarak karmaşık Python nesnelerinin seri hale getirilmesine izin veren bir protokoldür. Bu nedenle, Python'a özgüdür ve diğer dillerde yazılmış uygulamalarla iletişim kurmak için kullanılamaz. Varsayılan olarak da güvensizdir: güvenilmeyen bir kaynaktan gelen pickle verilerinin dizilerinin seri halden çıkarılması, veriler yetenekli bir saldırgan tarafından hazırlanmışsa rasgele kod yürütebilir.

BÖLÜM 8

Hatalar ve Özel Durumlar

Şimdiye kadar hata mesajlarından fazla bahsedilmedi, ancak örnekleri denediyseniz muhtemelen bazılarını görmüşsünüzdür. (En azından) iki ayırt edilebilir hata türü vardır: *söz dizimi hataları* (*syntax errors*) ve *özel durumlar* (*exceptions*).

8.1 Söz Dizimi Hataları

Ayrıştırma hataları olarak da bilinen söz dizimi hataları, Python öğrenirken belki de en sık karşılaşılan hatalardan biridir:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little 'arrow's pointing at the token in the line where the error was detected. The error may be caused by the absence of a token *before* the indicated token. In the example, the error is detected at the function `print()`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Özel Durumlar

Bir komut veya ifade söz dizimsel olarak doğru olsa bile, yürütülmeye çalışıldığında hataya neden olabilir. Yürütme sırasında algılanan hatalara *exceptions (özel durumlar)* denir ve bazıları programlar için kritik değildir: yakında Python programlarında bunların üstesinden gelmeyi öğreneceksiniz. Ancak, çoğu özel durum programlar tarafından önlenemez ve burada gösterildiği gibi hata iletileriyle sonuçlanır:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

Hata iletisinin son satırı hataya neyin sebep olduğu gösterir. Özel durumlar farklı türlerde gelir ve tür iletinin bir parçası olarak yazdırılır: örnekteki türler şunlardır `ZeroDivisionError`, `NameError` ve `TypeError`. Özel durum türü olarak yazdırılan dize, oluşan gömülü özel durumun adıdır. Bu, tüm gömülü özel durumlar için geçerlidir, ancak kullanıcı tanımlı özel durumlar için doğru olması gerekmez (yararlı bir kural olmasına rağmen). Standart özel durum adları gömülü tanımlayıcılardır (ayrılmış anahtar sözcükler değildir).

Satırın geri kalanı, özel durum türüne ve buna neyin neden olduğuna bağlı olarak ayrıntı gösterir.

Hata iletisinin önceki bölümü, özel durumun oluştuğu bağlamı yığın izleme geri dönüşü biçiminde gösterir. Genel olarak, kaynak satırları listeleyen bir yığın izleme listesi içerir; ancak, standart girişten okunan satırları görüntülemez.

bltin-exceptions yerleşik özel durumları ve anlamlarını listeler.

8.3 Özel Durumları İşleme

Seçili özel durumları işleyen programlar yazmak mümkündür. Geçerli bir tam sayı girilene kadar kullanıcıdan giriş isteyen, ancak kullanıcının programı kesmesine izin veren aşağıdaki örneğe bakın (`Control-C` veya işletim sistemi ne destekliyorsa onu kullanarak); kullanıcı kaynaklı kesintilerin `KeyboardInterrupt` özel durumu ile gösterildiğini unutmayın.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

`try` ifadesi aşağıdaki gibi çalışır.

- İlk olarak, *try yan tümcesi* (`try` ve `except` anahtar kelimeleri arasındaki ifadeler) yürütülür.
- Özel durum oluşmazsa, *except yan tümcesi* atlanır ve `try` ifadesinin yürütülmesi tamamlanır.
- `try` yan tümcesinin yürütülmesi sırasında bir özel durum oluşursa, yan tümcenin geri kalanı atlanır. Daha sonra belirtilen `except` türü ile karşılaşılırsa, *except yan tümcesi* yürütülür ve try/except bloğundan sonra yürütme devam eder.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

`try` ifadesi, farklı özel durumlar için işleyiciler belirtmek üzere birden fazla *except yan tümcesi* alabilir. Maksimum bir tane işleyici yürütülür. İşleyiciler yalnızca karşılık gelen *try yan tümcesinde* oluşan özel durumları işler, aynı `try` ifadesinin diğer işleyicilerinde işlemez. *except yan tümcesi* birden çok özel durumu parantezli demet olarak adlandırabilir, örneğin:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

except yan tümceleri tersine çevrilmişse (ilk olarak `except B` ile) B, B, B şeklinde yazdırılacaktır — ilk eşleşen *except yan tümcesi* tetiklenir.

Bir istisna oluştuğunda, istisnanın *argümanı* olarak da bilinen ilişkili bir değeri olabilir. Argümanın varlığı ve türü, istisna türüne bağlıdır.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException`, tüm istisnaların ortak temel sınıfıdır. Alt sınıflarından biri olan `Exception`, tüm önemli olmayan istisnaların temel sınıfıdır. `Exception` öğesinin alt sınıfları olmayan istisnalar genellikle işlenmez, çünkü bunlar programın sona ermesi gerektiğini belirtmek için kullanılır. Bunlar `sys.exit()` tarafından oluşturulan `SystemExit` ve bir kullanıcı programı kesmek istediğinde ortaya çıkan `KeyboardInterrupt` içerir.

`Exception` (neredeyse) her şeyi yakalayan bir joker karakter (wildcard) olarak kullanılabilir. Ancak, işlemeyi amaçladığımız istisna türleri konusunda mümkün olduğunca spesifik olmak ve beklenmeyen istisnaların yayılmasına izin vermek iyi bir uygulamadır.

`Exception` işlemek için en yaygın model, istisnayı yazdırmak veya log'a kaydetmek ve ardından yeniden yükseltmektir (arayanın istisnayı da işlemesine izin verir):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

`try` … `except` ifadesinin isteğe bağlı bir *else yan tümcesi* vardır, bu da mevcut olduğunda tüm *except yan tümcelerini* izlemelidir. *try yan tümcesi* bir özel durum oluşturmazsa yürütülmesi gereken kod için yararlıdır. Mesela:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

`else` yan tümcesinin kullanılması, `try` yan tümcesine ek kod eklemekten daha iyidir çünkü yanlışlıkla `try` tarafından korunan kod tarafından oluşturulmamış bir istisnayı yakalamayı önler. … `except` ifadesi.

İstisna işleyicileri, yalnızca *try yan tümcesinde* anında ortaya çıkan istisnaları değil, aynı zamanda *try yan tümcesinde* çağrılan (dolaylı olarak da olsa) işlevlerin içinde oluşan istisnaları da işler. Örneğin:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Hata Yükseltme

`raise` ifadesi, programcının belirli bir istisnanın gerçekleşmesini zorlamasını sağlar. Örneğin:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

`raise` için tek argüman, ortaya çıkacak istisnayı belirtir. Bu, bir istisna örneği veya bir istisna sınıfı (`BaseException` 'dan türetilen bir sınıf, örneğin `Exception` veya alt sınıflarından) olmalıdır. Bir istisna sınıfı iletilirse, yapıcısını hiçbir argüman olmadan çağırarak dolaylı olarak başlatılır:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

Bir istisnanın oluşturulup oluşturulmadığını belirlemeniz gerekiyorsa ancak onu işlemeyi düşünmüyorsanız, `raise` ifadesinin daha basit bir biçimi, istisnayı yeniden oluşturmanıza olanak tanır:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

8.5 İstisna Zincirleme

Bir `except` bölümü içinde işlenmeyen bir istisna meydana gelirse, istisnanın kendisine eklenmiş olarak işlenmesi ve şu hata mesajına dahil edilmesi gerekir:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

`raise` ifadesi, bir istisnanın başka bir istisnanın doğrudan sonucu olduğunu belirtmek için isteğe bağlı `from` yan tümcesine izin verir:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Bu, özel durumları dönüştürürken yararlı olabilir. Mesela:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

Ayrıca, `from None` deyimi kullanılarak otomatik istisna zincirlemenin devre dışı bırakılmasına izin verir:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

Zincirleme mekaniği hakkında daha fazla bilgi için bkz. bltin-exceptions.

8.6 Kullanıcı Tanımlı İstisnalar

Programlar yeni bir istisna sınıfı oluşturarak kendi özel durumlarını adlandırabilir (Python sınıfları hakkında daha fazla bilgi için bkz: *Sınıflar*). Özel durumlar genellikle doğrudan veya dolaylı olarak `Exception` sınıfından türetilmelidir.

İstisna sınıfları, başka herhangi bir sınıfın yapabileceği her şeyi yapan tanımlanabilir, ancak genellikle basit tutulur ve çoğu zaman yalnızca istisna için işleyiciler tarafından hatayla ilgili bilgilerin çıkarılmasına izin veren bir dizi öznitelik sunar.

Çoğu özel durum, standart özel durumların adlandırışına benzer şekilde "Hata" ile biten adlarla tanımlanır.

Birçok standart modül, tanımladıkları işlevlerde oluşabilecek hataları raporlamak için kendi istisnalarını tanımlar.

8.7 Temizleme Eylemlerini Tanımlama

`try` deyimi, her koşulda yürütülmesi gereken temizleme eylemlerini tanımlamayı amaçlayan başka bir opsiyonel yan tümceye sahiptir. Mesela:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

Bir `finally` yan tümcesi varsa, `finally` yan tümcesi `try` deyimi tamamlanmadan önceki son görev olarak yürütülür. `finally` yan tümcesi `try` deyiminin bir istisna oluşturup oluşturmadığından bağımsız çalışır. Aşağıdaki noktalarda, bir istisna oluştuğunda daha karmaşık durumlar anlatılmaktadır:

- `try` yan tümcesinin yürütülmesi sırasında bir istisna oluşursa, istisna bir `except` yan tümcesi tarafından işlenebilir. İstisna bir `except` yan tümcesi tarafından ele alınmıyorsa, istisna `finally` yan tümcesi yürütüldükten sonra yeniden oluşturulur.
- Bir `except` veya `else` yan tümcesinin yürütülmesi sırasında bir istisna oluşabilir. Yine, istisna, `finally` yan tümcesi yürütüldükten sonra yeniden oluşturulur.
- `finally` yan tümcesi bir `break`, `continue` veya `return` deyimini yürütürse, istisnalar yeniden oluşturulmaz.
- `try` ifadesi bir `break`, `continue` veya `return` ifadesine ulaşırsa, `finally` yan tümcesi `break`, `continue` veya `return` ifadesinin yürütülmesinin hemen öncesinde yürütülür.
- Bir `finally` yan tümcesi bir `return` ifadesini içeriyorsa, döndürülen değer, `finally` yan tümcesinin `return` ifadesindeki değer olacaktır, `try` yan tümcesinin `return` ifadesindeki değer değil.

Mesela:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Daha karmaşık bir örnek:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Gördüğünüz gibi, `finally` yan tümcesi her durumda yürütülür. İki dizeyi bölerek oluşturulan `TypeError`, `except` yan tümcesi tarafından işlenmez ve bu nedenle `finally` yan tümcesi yürütüldikten sonra yeniden yükseltilir.

Gerçek dünyadaki uygulamalarda `finally` yan tümcesi, kaynağın kullanımının başarılı olup olmadığına bakılmaksızın dış kaynakları (dosyalar veya ağ bağlantıları gibi) serbest bırakmak için yararlıdır.

8.8 Önceden Tanımlanmış Temizleme Eylemleri

Bazı nesneler, nesneyi kullanan işlemin başarılı veya başarısız olup olmadığına bakılmaksızın, nesne artık gerekli olmadığında gerçekleştirilecek standart temizleme eylemlerini tanımlar. Bir dosyayı açmaya ve içeriğini ekrana yazdırmaya çalışan aşağıdaki örneğe bakın.

```
for line in open("myfile.txt"):
    print(line, end="")
```

Bu kodla ilgili sorun, kodun bu bölümünün yürütülmesi tamamlandıktan sonra dosyayı belirsiz bir süre açık bırakmasıdır. Bu basit komut dosyalarında bir sorun değildir, ancak daha büyük uygulamalar için bir sorun olabilir. `with` ifadesi, dosyalar gibi nesnelerin her zaman hızlı ve doğru temizlenmesini sağlayacak şekilde kullanılmasına izin verir.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

İfade çalıştırıldıktan sonra, satırlar işlenirken bir sorunla karşılaşılsa bile *f* dosyası her zaman kapatılır. Dosyalar gibi önceden tanımlanmış temizleme eylemleri sağlayan nesneler dokümantasyonlarında bunu gösterir.

8.9 Birden Fazla Alakasız İstisna Oluşturma ve İşleme

Meydana gelen birkaç istisnanın rapor edilmesinin gerekli olduğu durumlar vardır. Bu, genellikle birkaç görevin paralel olarak başarısız olabileceği eşzamanlılık çerçevelerinde geçerlidir, ancak ilk istisnayı yükseltmek yerine yürütmeye devam etmenin ve birden çok hata toplamanın istendiği başka kullanım durumları da vardır.

Yerleşik `ExceptionGroup`, birlikte oluşturulabilmeleri için istisna örneklerinin bir listesini sarar. Kendisi bir istisnadır, bu nedenle herhangi bir istisna gibi yakalanabilir.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
  | ExceptionGroup: there were problems (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

`except` yerine `except*` kullanarak, yalnızca gruptaki belirli bir türle eşleşen istisnaları seçerek işleyebiliriz. İç içe geçmiş bir istisna grubunu gösteren aşağıdaki örnekte, her bir `except*` yan tümcesi, diğer tüm istisnaların diğer yan tümcelere yayılmasına ve sonunda yeniden ortaya çıkmasına izin verirken, belirli bir türdeki grup istisnalarını çıkarır.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Bir istisna grubunda iç içe geçmiş istisnaların türler değil, örnekler olması gerektiğini unutmayın. Bunun nedeni, pratikte istisnaların tipik olarak, aşağıdaki kalıp boyunca program tarafından önceden oluşturulmuş ve yakalanmış olanlar olmasıdır:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
...
```

8.10 İstisnaları Notlarla Zenginleştirme

Yükseltilmek üzere bir istisna oluşturulduğunda, genellikle meydana gelen hatayı açıklayan bilgilerle başlatılır. İstisna yakalandıktan sonra bilgi eklemenin yararlı olduğu durumlar vardır. Bu amaçla, istisnaların bir dizgiyi kabul eden ve onu istisnanın notlar listesine ekleyen `add_note(note)` metodu vardır. Standart geri işleme (traceback) oluşturma, tüm notları istisnadan sonra eklendikleri sırayla içerir.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

Örneğin, istisnaları bir istisna grubuna toplarken, bireysel hatalar için bağlam bilgisi eklemek isteyebiliriz. Aşağıda, gruptaki her özel durum, bu hatanın ne zaman oluştuğunu gösteren bir nota sahiptir.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('We have some problems', excs)
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

BÖLÜM 9

Sınıflar

Sınıflar, verileri ve işlevleri bir arada tutan bir araçtır. Yeni bir sınıf oluşturulurken, objeye ait yeni örnekler (instances) oluşturulur. Sınıf örnekleri, durumlarını korumak için onlara iliştirilmiş niteliklere sahip olabilir. Sınıfların durumunu modifiye etmek için ise metotlar kullanılabilir.

Diğer programlama dilleriyle karşılaştırıldığında, Python'un sınıf mekanizması olabildiğince az yeni sözdizimi ve semantik içeren sınıflar ekler. C++ ve Modula-3'te bulunan sınıf mekanizmalarının bir karışımıdır. Python sınıfları Nesne Yönelimli Programlama'nın tüm standart özelliklerini sağlar: sınıf devralma mekanizması birden çok temel sınıfa izin verir, türetilmiş bir sınıf temel sınıfının veya sınıflarının herhangi bir metodunu geçersiz kılabilir ve bir metot aynı ada sahip bir temel sınıfın metodunu çağırabilir. Nesneler farklı miktar ve türlerde veri içerebilir. Modüller için olduğu gibi, sınıflar da Python'un dinamik doğasına uygundur: çalışma sırasında oluşturulurlar ve oluşturulduktan sonra da değiştirilebilirler.

C++ terminolojisinde, normalde sınıf üyeleri (veri üyeleri dahil) *public* (aşağıdakine bakın *Özel Değişkenler*) ve tüm üye fonksiyonları *virtual*' dır. Modula-3'te olduğu gibi, nesnenin üyelerine metotlarından ulaşmak için bir kısayol yoktur: metodun işlevi, çağrı tarafından örtülü olarak sağlanan objeyi temsil eden açık bir argümanla bildirilir. Smalltalk'ta olduğu gibi, sınıfların kendileri de birer nesnedir. Bu sayede metotlar yeniden isimlendirilebilir veya içe aktarılabilir. C++ ve Modula-3'ün aksine, yerleşik türler kullanıcının üzerlerine yapacağı geliştirmeler için temel sınıflar olarak kullanılabilir. Ayrıca, C++'ta olduğu gibi, özel sözdizimine sahip çoğu yerleşik işleç (aritmetik işleçler, alt simgeleme vb.) sınıf örnekleri için yeniden tanımlanabilir, geliştirilebilir.

(Sınıflara dair evrensel terimlerin yanı sıra, nadiren Smalltalk ve C++ terimlerini kullanacağım. Onun yerine Modula-3 terimlerini kullanacağım çünkü Python'un nesne yönelimli semantiğine C++'tan daha yakın, yine de ancak çok az okuyucunun bunları duymuş olacağını tahmin ediyorum.)

9.1 İsim ve Nesneler Hakkında Birkaç Şey

Nesnelerin bireyselliği vardır ve birden çok ad (birden çok kapsamda) aynı nesneye bağlanabilir. Bu, diğer dillerde örtüşme (aliasing) olarak bilinir. Bu genellikle Python'a ilk bakışta takdir edilmeyen bir özellik olsa da değiştirilemeyen veri türleriyle (sayılar, dizeler, diziler) uğraşırken rahatlıkla göz ardı edilebilir. Ancak, örtüşmeler, listeler, sözlükler ve diğer türlerin çoğu gibi değişebilir nesneleri içeren Python kodunun semantiği üzerinde şaşırtıcı bir etkiye sahiptir. Diğer adlar bazı açılardan işaretçiler (pointers) gibi davrandığından, bu genellikle programın yararına kullanılır. Örneğin, bir nesneyi geçirmek kolaydır çünkü uygulama tarafından geçirilen şey yalnızca bir işaretçidir; bir fonksiyon argüman olarak geçirilen bir nesneyi değiştirirse, çağıran bu değişikliği görür — bu Pascal'daki iki farklı bağımsız değişken geçirme mekanizmasına olan gereksinimi ortadan kaldırır.

9.2 Python Etki Alanları ve Ad Alanları

Sınıflara başlamadan önce, size Python'un etki alanı kuralları hakkında bir şey söylemeliyim. Sınıf tanımlarında ad alanlarının kullanımının bazı püf noktaları vardır ve neler olduğunu tam olarak anlamak için etki alanlarının ve isimlerin nasıl çalıştığını bilmeniz gerekir. Bu arada, bu konuda bilgi edinmek herhangi bir ileri düzey Python programcısı için yararlıdır.

Haydi birkaç tanımlama ile başlayalım.

Ad alanları (namespace), adlardan nesnelere eşlemedir. Çoğu isim şu anda Python sözlükleri olarak uygulanmaktadır, ancak bu fark edilir çapta bir fark yaratmaz (performans hariç) ve gelecekte değişebilir. Ad alanlarına örnek olarak şunlar verilebilir: yerleşik isimler (`abs()` ve yerleşik özel durum adları gibi fonksiyonları içerir); modüldeki global adlar; ve bir fonksiyon çağırmadaki yerel adlar. Bir bakıma, bir nesnenin nitelik kümesi de bir ad alanı oluşturur. İsimler hakkında bilinmesi gereken önemli şey, farklı ad alanlarındaki adlar arasında kesinlikle bir ilişki olmamasıdır; örneğin, iki farklı modülün her ikisi de karışıklık olmadan `maximize` fonksiyonunu tanımlayabilir — modül kullanıcılarının modül adını örneklemesi gerekir.

Bu arada, *nitelik* sözcüğünü bir noktayı takip eden herhangi bir isim için kullanıyorum. Mesela, `z.real` ifadesinde `real`, `z` nesnesine ait bir niteliktir. Açıkçası, modüllerde isimlere yapılan referanslar nitelik referanslarıdır: `modname.funcname` ifadesinde `modname` bir modül nesnesi ve `funcname` onun bir niteliğidir. Bu durumda, modülün nitelikleri ve modüldeki global değişkenler arasında bir eşleşme olur: aynı ad alanını paylaşmaları![1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Ad alanları farklı anlarda oluşturulur ve farklı yaşam sürelerine sahiptir. Yerleşik adları içeren ad alanı, Python yorumlayıcısı başlatıldığında oluşturulur ve hiçbir zaman silinmez. Modül tanımı okunduğunda modül için genel ad alanı oluşturulur; normalde, modül ad alanları da yorumlayıcı çıkıp bitene kadar sürer. Bir komut dosyasından veya etkileşimli olarak okunan yorumlayıcının en üst düzey çağrısı tarafından yürütülen ifadeler `__main__` adlı bir modülün parçası olarak kabul edilir, bu nedenle kendi genel ad alanlarına sahiptirler. (Yerleşik isimler aslında bir modülde de yaşar; buna `builtins`.)

Bir işlevin yerel ad alanı, bir fonksiyon çağrıldığında oluşturulur ve fonksiyon başa çıkamadığı bir hata veya istisna ile karşılaştığında silinir. (Aslında, bir bakıma ad alanı unutulmaktadır diyebiliriz.) Tabii ki, özyinelemeli çağrıların her birinin kendi yerel ad alanı vardır.

scope, bir ad alanının doğrudan erişilebilir olduğu Python programının metinsel bölgesidir. Burada "Doğrudan erişilebilir", niteliksiz bir başvurunun hedeflediği ismi ad alanında bulmaya çalıştığı anlamına gelir.

Kapsamlar statik olarak belirlense de, dinamik olarak kullanılırlar. Yürütme sırasında herhangi bir zamanda, ad alanlarına doğrudan erişilebilen 3 veya 4 iç içe kapsam vardır:

- en içte bulunan kapsam, ilk aranan olmakla birlikte yerel isimleri içerir
- en yakın kapsayan kapsamdan başlayarak aranan kapsayan fonksiyonların kapsamları yerel olmayan, aynı zamanda genel olmayan adlar içerir
- sondan bir önceki kapsam, geçerli modülün genel adlarını içerir
- en dıştaki kapsam (en son aranan), yerleşik adlar içeren ad alanıdır

Bir ad global olarak bildirilirse, tüm başvurular ve atamalar doğrudan modülün genel adlarını içeren orta kapsama gider. En iç kapsamın dışında bulunan değişkenleri yeniden bağlamak için `nonlocal` ifadesi kullanılabilir; yerel olarak bildirilmezse de, bu değişkenler salt okunurdur (böyle bir değişken düzenlenirse, aynı adlı dış değişkeni değiştirmeden en iç kapsamda *yeni* bir yerel değişken oluşturur).

Genellikle, yerel kapsam (metinsel olarak) geçerli fonksiyonun yerel adlarına başvurur. Dış fonksiyonlarda, yerel kapsam genel kapsamla aynı ad alanına başvurur: modülün ad alanı. Sınıf tanımları yerel kapsamda başka bir ad alanı yerleştirir.

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

Kapsamların metinsel olarak belirlendiğini fark etmek önemlidir: bir modülde tanımlanan bir işlevin genel kapsamı, işlevin nereden veya hangi diğer adla adlandırıldığından bağımsız olarak modülün ad alanıdır. Öte yandan, gerçek ad araması dinamik olarak yapılır, çalışma zamanında — ancak, dil tanımı statik ad çözümlemelerine doğru, "derleme" zamanında gelişir, bu nedenle dinamik ad çözümlemesini güvenmeyin! (Aslında, yerel değişkenler zaten statik olarak belirlenir.)

Python'un özel bir cilvesi, eğer `global` veya `nonlocal` deyimi geçerli değilse – isimlere yapılan atamaların her zaman en içteki kapsama girmesidir. Atamalar verileri kopyalamaz — adları nesnelere bağlarlar. Aynı şey silme için de geçerlidir: `del x` deyimi, `x` bağlamasını yerel kapsam tarafından başvurulan ad alanından kaldırır. Aslında, yeni adlar tanıtan tüm işlemler yerel kapsamı kullanır: özellikle, `import` ifadeleri ve işlev tanımları modül veya işlev adını yerel kapsamda bağlar.

`global` deyimi, belirli değişkenlerin genel kapsamda yaşadığını ve orada geri alınması gerektiğini belirtmek için kullanılabilir; `nonlocal` deyimi, belirli değişkenlerin bir çevreleme kapsamında yaşadığını ve orada geri alınması gerektiğini gösterir.

9.2.1 Kapsamlar ve Ad Alanları Örneği

Bu, farklı kapsamlara ve ad alanlarına başvurmayı ve `global` ve `nonlocal` değişken bağlamasını nasıl etkileyeceğini gösteren bir örnektir:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Örnek kodun çıktısı şudur:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Varsayılan atama olan *local* atamasının *scope_test*'in *spam* bağlamasını nasıl değiştirmediğini unutmayın. `nonlocal` ataması *scope_test*'in *spam* bağlamasını değiştirdi, hatta `global` ataması modül düzeyindeki bağlamasını değiştirdi.

Ayrıca `global` atamasından önce *spam* için herhangi bir bağlama olmadığını görebilirsiniz.

9.3 Sınıflara İlk Bakış

Sınıflar biraz yeni söz dizimi, üç yeni nesne türü ve bazı yeni semantiklere sahiptir.

9.3.1 Sınıf Tanımlama Söz Dizimi

Sınıf tanımlamasının en basit biçimi şöyledir:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Sınıf tanımlamaları, fonksiyon tanımlamaları (`def` deyimleri) ile aynı şekilde, herhangi bir etkiye sahip olmadan önce yürütülmelidir. (Bir sınıf tanımını `if` ifadesinin bir dalına veya bir fonksiyonun içine yerleştirebilirsiniz.)

Uygulamada, bir sınıf tanımı içindeki ifadeler genellikle fonksiyon tanımlamaları olacaktır, ancak diğer ifadelere de izin verilir ve daha sonra bahsedeceğimiz üzere bazen yararlılardır. Bir sınıfın içindeki fonksiyon tanımları normalde, yöntemler için çağırma kuralları tarafından dikte edilen tuhaf bir bağımsız değişken listesi biçimine sahiptir — bu daha sonra açıklanacak.

Sınıf tanımı girildiğinde, yeni bir ad alanı oluşturulur ve yerel kapsam olarak kullanılır — böylece yerel değişkenlere yapılan tüm atamalar bu yeni ad alanına gider. Özellikle, fonksiyon tanımlamaları yeni fonksiyonların adını buraya bağlar.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

9.3.2 Sınıf Nesneleri

Sınıf nesneleri iki tür işlemi destekler: nitelik referansları ve örnekleme.

Nitelik referansları, Python'daki tüm nitelik referansları için kullanılan standart söz dizimi olan `obj.name` kullanır. Geçerli nitelik adları, sınıf nesnesi oluşturulduğunda sınıfın ad alanında bulunan tüm adlardır. Yani, sınıf tanımı şöyle görünüyorsa:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Sınıf *örnekleme* fonksiyon notasyonunu kullanır. Sınıf nesnesinin, sınıfın yeni bir örneğini döndüren parametresiz bir fonksiyon olduğunu varsayın. Örneğin (yukarıdaki sınıfı varsayarsak):

```
x = MyClass()
```

sınıfın yeni bir *örnek* öğesini oluşturur ve bu nesneyi `x` yerel değişkenine atar.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Örnek Nesneleri

Şimdi örnek nesnelerle ne yapabiliriz? Örnek nesneleri tarafından anlaşılan tek işlemler nitelik başvurularıdır. İki tür geçerli öznitelik adı vardır: veri nitelikleri ve metotları.

data attributes correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that "belongs to" an object.

Örnek nesnesinin geçerli metot adları nesnenin sınıfına bağlıdır. Fonksiyon nesneleri olan bir sınıfın tüm nitelikleri, örneklerinin karşılık gelen fonksiyonlarını tanımlar. Bu nedenle, örneğimizde, `x.f` geçerli bir metot referansıdır, ne de olsa `MyClass.f` bir fonksiyondur ancak `MyClass.i` fonksiyon olmadığından `x.i` değildir. Ancak `x.f`, `MyClass.f` ile aynı şey değildir çünkü bir fonksiyon nesnesi değil, *metot nesnesi* 'dir.

9.3.4 Metot Nesneleri

Genellikle, bir metot bağlandıktan hemen sonra çağrılır:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

daima `hello world` yazdırmaya devam edecek.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely

Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used…

Aslında, cevabı tahmin etmiş olabilirsiniz: yöntemlerle ilgili özel şey, örnek nesnesinin fonksiyonun ilk argüman olarak geçirilmesidir. Örneğimizde, `x.f()` çağrısı tam olarak `MyClass.f(x)` ile eş değerdir. Genel olarak, *n* tane argümana sahip bir metodu çağırmak, ilk argümandan önce metodun örnek nesnesi eklenerek oluşturulan bir argüman listesiyle karşılık gelen fonksiyonu çağırmaya eş değerdir.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5 Sınıf ve Örnek Değişkenleri

Genel olarak, örnek değişkenleri o örneğe özgü veriler içindir ve sınıf değişkenleri sınıfın tüm örnekleri tarafından paylaşılan nitelikler ile metotlar içindir:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

İsim ve Nesneler Hakkında Birkaç Şey 'te anlatıldığı gibi, paylaşılan veriler listeler ve sözlükler gibi *mutable* nesnelerini içeren şaşırtıcı etkilere sahip olabilir. Örneğin, aşağıdaki koddaki *tricks* listesi sınıf değişkeni olarak kullanılmamalıdır, çünkü yalnızca tek bir liste tüm *Dog* örnekleri tarafından paylaşılacaktır:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Doğru olan ise veriyi paylaşmak yerine bir örnek değişkeni kullanmaktır:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Rastgele Açıklamalar

Aynı nitelik adı hem bir örnekte hem de bir sınıfta oluşuyorsa, nitelik araması örneğe öncelik verir:

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Veri niteliklerine metotların yanı sıra bir nesnenin sıradan kullanıcıları ("istemciler") tarafından başvurulabilir. Başka bir deyişle, sınıflar saf soyut veri türlerini uygulamak için kullanılamaz. Aslında, Python'daki hiçbir şey, veri gizlemeyi zorlamaz. Bu durum geleneksel kullanımından dolayı bu şekildedir. (Öte yandan, C ile yazılan Python uygulamaları, uygulama ayrıntılarını tamamen gizleyebilir ve gerekirse bir nesneye erişimi kontrol edebilir; bu, C ile yazılmış Python uzantıları tarafından kullanılabilir.)

İstemciler veri niteliklerini dikkatle kullanmalıdır — istemciler veri özniteliklerini damgalayarak yöntemler tarafından tutulan değişmezleri bozabilir. İstemcilerin, metotların geçerliliğini etkilemeden bir örnek nesnesine kendi veri niteliklerini ekleyebileceğini unutmayın. Tabii ki burada da ad çakışmalarından kaçınılmalıdır, adlandırma kuralları ise kaçınmak için oldukça faydalı olabilir.

Yöntemlerin içinden veri niteliklerine (veya diğer metotlara!) başvurmak için kısa yol yoktur. Bu aslında metotların okunabilirliğini arttırıyor, çünkü bir metoda bakarken yerel değişkenleri ve örnek değişkenlerini karıştırma ihtimali bırakmamış oluyoruz.

Genellikle, bir metodun ilk bağımsız değişkenine `self` denir. Bu bir kullanım geleneğinden başka bir şey değildir, yani `self` adının Python için kesinlikle özel bir anlamı yoktur. Bununla birlikte, kurala uymadığınızda kodunuzun diğer Python programcıları tarafından daha az okunabilir olabileceğini ve yazılabilecek potansiyel bir *sınıf tarayıcısı* programının bu kurala dayanıyor olabileceğini unutmayın.

Sınıf niteliği olan herhangi bir fonksiyon nesnesi, bu sınıfın örnekleri için bir metot tanımlar. Fonksiyon tanımının metinsel olarak sınıf tanımına dahil olması gerekli değildir: sınıftaki yerel bir değişkene fonksiyon nesnesi atamak da uygundur. Mesela:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Metotlar, `self` bağımsız değişkeninin metot niteliklerini kullanarak diğer metotları çağırabilir:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Metotlar, global adlara sıradan işlevler gibi başvuru yapabilir. Bir metotla ilişkili genel kapsam, tanımını içeren modüldür. (Bir sınıf hiçbir zaman genel kapsam olarak kullanılmaz.) Global verileri bir metotta kullanmak için nadiren iyi bir nedenle karşılaşılsa da, genel kapsamın birçok meşru kullanımı vardır: bir kere, genel kapsamlıya içe aktarılan fonksiyon ve modüller, metotlar ve içinde tanımlanan fonksiyonlar ve sınıflar tarafından kullanılabilir. Genellikle, metodu içeren sınıfın kendisi bu genel kapsamda tanımlanır ve sonraki bölümde bir yöntemin kendi sınıfına başvurmak istemesinin bazı iyi nedenlerini bulacağız.

Her değer bir nesnedir ve bu nedenle bir *sınıf* (*type* olarak da adlandırılır) bulundurur. `object.__class__` olarak depolanır.

9.5 Kalıtım

Tabii ki, bir dil özelliği kalıtımı desteklemeden "sınıf" adına layık olmaz. Türetilmiş sınıf tanımının söz dizimi şöyle görünür:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Türetilmiş bir sınıf tanımının yürütülmesi, temel sınıfla aynı şekilde devam eder. Sınıf nesnesi inşa edildiğinde, temel

sınıf hatırlanır. Bu, nitelik başvurularını çözmek için kullanılır: istenen nitelik sınıfta bulunmazsa, arama temel sınıfa bakmaya devam eder. Temel sınıfın kendisi başka bir sınıftan türetilmişse, bu kural özyinelemeli olarak uygulanır.

Türetilmiş sınıfların somutlaştırılmasında özel bir şey yoktur: `DerivedClassName()` sınıfın yeni bir örneğini oluşturur. Metot başvuruları aşağıdaki gibi çözümlenir: ilgili sınıf niteliği aranır, gerekirse temel sınıflar zincirinin aşağısına inilir ve bu bir fonksiyon nesnesi veriyorsa metot başvurusu geçerlidir.

Türetilmiş sınıflar, temel sınıflarının metotlarını geçersiz kılabilir. Metotlar aynı nesnenin diğer yöntemlerini çağırırken özel ayrıcalıkları olmadığından, aynı temel sınıfta tanımlanan başka bir metodu çağıran bir temel sınıfın metodu, onu geçersiz kılan türetilmiş bir sınıfın metodunu çağırabilir. (C++ programcıları için: Python'daki tüm yöntemler etkili bir şekilde `sanal`.)

Türetilmiş bir sınıfta geçersiz kılma yöntemi aslında yalnızca aynı adlı temel sınıf yöntemini değiştirmek yerine genişletmek isteyebilir. Temel sınıf metodunu doğrudan çağırmanın basit bir yolu vardır: sadece `BaseClassName.methodname(self, arguments)` çağırın. Bu bazen müşteriler için de yararlıdır. (Bunun yalnızca temel sınıfa genel kapsamda `BaseClassName` olarak erişilebiliyorsa çalıştığını unutmayın.)

Python'un kalıtımla çalışan iki yerleşik fonksiyonu vardır:

- Bir örneğin türünü denetlemek için `isinstance()` kullanın: `isinstance(obj, int)` yalnızca `obj.__class__` `int` veya `int` sınıfından türetilmiş bir sınıfsa `True` olacaktır.
- Sınıf kalıtımını denetlemek için `issubclass()` kullanın: `issubclass(bool, int)` `True` 'dur, çünkü `bool` `int` 'in bir alt sınıfıdır. Ancak, `issubclass(float, int)` `False` olduğundan `float`, `int` alt sınıfı değildir.

9.5.1 Çoklu Kalıtım

Python, çoklu kalıtım biçimini de destekler. Birden çok temel sınıf içeren bir sınıf tanımı şöyle görünür:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

Aslında, durum bundan biraz daha karmaşıktır: yöntem çözümleme sırası, `super()` için işbirliği çağrılarını desteklemek için dinamik olarak değişir. Bu yaklaşım, diğer bazı çoklu kalıtım dillerinde sonraki çağrı yöntemi olarak bilinir ve tekli kalıtım dillerinde bulunan süper çağrıdan daha güçlüdür.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see python_2.3_mro.

9.6 Özel Değişkenler

Python'da bir nesnenin içinden erişilmesi dışında asla erişilemeyen "özel" örnek değişkenleri yoktur. Ancak, çoğu Python kodu tarafından izlenen bir kural vardır: alt çizgi (örneğin `_spam`) ile öneklenmiş bir ad API'nin genel olmayan bir parçası olarak kabul edilmelidir (bir fonksiyon, metot veya veri üyesi olsun). Bir uygulama detayıdır ve önceden haber verilmeksizin değiştirilebilir.

Sınıf-özel üyeler için geçerli bir kullanım örneği olduğundan (yani alt sınıflar tarafından tanımlanan adlara sahip adların ad çakışmasını önlemek için), *name mangling* adı verilen böyle bir mekanizma için sınırlı destek vardır. `__spam` formunun herhangi bir tanımlayıcısı (en az iki satır altı, en fazla bir alt çizgi) metinsel olarak `_classname__spam` ile değiştirilir; Bu mangling, bir sınıfın tanımı içinde gerçekleştiği sürece tanımlayıcının söz dizimsel konumuna bakılmaksızın yapılır.

Ayrıca bakınız

The private name mangling specifications for details and special cases.

Ad mangling, alt sınıfların sınıf içi metot çağrılarını kesmeden metotları geçersiz kılmasına izin vermek için yararlıdır. Mesela:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Yukarıdaki örnek, `MappingSubclass` sırasıyla `Mapping` sınıfında `_Mapping__update` ve `mappingSubclass` sınıfında `_MappingSubclass__update` ile değiştirildiği için `__update` tanımlayıcısı tanıtsa bile çalışır.

Mangling kurallarının çoğunlukla kazaları önlemek için tasarlandığını unutmayın; özel olarak kabul edilen bir değişkene erişmek veya değiştirmek hala mümkündür. Bu, hata ayıklayıcı gibi özel durumlarda bile yararlı olabilir.

`exec()` veya `eval()` koduna geçirilen kodun çağırma sınıfının sınıf adını geçerli sınıf olarak görmediğine dikkat edin; bu, etkisi aynı şekilde birlikte bayt derlenmiş kodla sınırlı olan `global` deyiminin etkisine benzer. Aynı kısıtlama `getattr()`, `setattr()` ve `delattr()` ve doğrudan `__dict__` atıfta bulunurken de geçerlidir.

9.7 Oranlar ve Bitişler

Bazen, birkaç adlandırılmış veri öğesini bir araya getirerek Pascal *record* 'u veya C *struct* 'ına benzer bir veri türüne sahip olmak yararlıdır. Deyimsel yaklaşım, bu amaç için `dataclasses` kullanmaktır:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
'computer lab'
>>> john.salary
1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

9.8 Yineleyiciler

Şimdiye kadar büyük olasılıkla çoğu kapsayıcı nesnenin bir `for` deyimi kullanılarak döngüye alınabileceğini fark etmişsinizdir:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Bu erişim tarzı açık, özlü ve kullanışlıdır. Yineleyicilerin kullanımı Python'u istila eder ve birleştirir. Perde arkasında `for` deyimi kapsayıcı nesne üzerinde `iter()` öğesini çağırır. Fonksiyon, kapsayıcıdaki öğelere teker teker erişen `__next__()` metodunu tanımlayan bir yineleyici nesnesi döndürür. Başka öğe olmadığında, `__next__()`, `for` döngüsünün sonlandırılacağını bildiren bir `StopIteration` hatası oluşturur. `next()` yerleşik fonksiyonunu kullanarak `__next__()` yöntemini çağırabilirsiniz; Bu örnek, her şeyin nasıl çalıştığını gösterir:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Üreteçler

Üreteçler yineleyiciler oluşturmak için basit ve güçlü bir araçtır. Normal fonksiyonlar gibi yazılırlar, ancak veri döndürmek istediklerinde `yield` deyimini kullanırlar. Üzerinde her `next()` çağrıldığı zaman, üreteç kaldığı yerden devam eder (tüm veri değerlerini ve hangi deyimin en son yürütüldüğını hatırlar). Bu örnek, üreteçlerin oluşturulmasının ne kadar da kolay olabileceğini gösterir:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Başka bir önemli özellik, yerel değişkenlerin ve yürütme durumunun çağrılar arasında otomatik olarak kaydedilmesidir. Bu, fonksiyonun yazılmasını kolaylaştırdı ve `self.index` ve `self.data` gibi değişkenleri kullanmaya kıyasla çok daha net hale getirdi.

Otomatik metot oluşturma ve kaydetme programı durumuna ek olarak, üreteçler sonlandırıldığında otomatik olarak `StopIteration` 'ı yükseltirler. Birlikte, bu özellikler normal bir işlev yazmaktan daha fazla çaba harcamadan yinelemeler oluşturmayı kolaylaştırır.

9.10 Üreteç İfadeleri

Bazı basit üreteçler, listelere benzer bir söz dizimi kullanılarak ve köşeli ayraçlar yerine parantezlerle kısaca kodlanabilir. Bu ifadeler, üreteçlerin kapsayıcı bir fonksiyon tarafından hemen kullanıldığı durumlar için tasarlanmıştır. Üreteç ifadeleri tam üreteç tanımlarından daha kompakt ancak daha az çok yönlüdür ve aynı özellikle liste anlamalarından daha bellek dostu olma eğilimindedir.

Örnekler:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

BÖLÜM 10

Standart Kütüphanenin Özeti

10.1 İşletim Sistemi Arayüzü

`os` modülü, işletim sistemiyle etkileşim kurmak için düzinelerce fonksiyon sağlar:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python313'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

`from os import *` yerine `import os` stilini kullandığınızdan emin olun. Yanlış kullanım olan ilk stili tercih etmek, `os.open()` 'ın çok daha farklı çalışan gömülü `open()` fonksiyonunu gölgelemesine neden olur.

Gömülü `dir()` ve `help()` fonksiyonları `os` gibi büyük modüllerle çalışma konusunda interaktif yardımcılar olarak kullanışlıdır:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Günlük dosya ve dizin yönetimi görevleri için `shutil` modülü kullanımı daha kolay olan daha yüksek düzeyli (higher level) bir arayüz sağlar:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Dosya Joker Karakterleri

`glob` modülü , klasör joker karakter aramalarından dosya listeleri oluşturabilmek için bir fonsiyon sağlar:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Komut Satırı Argümanları

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's *argv* attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

`argparse` modülü, komut satırı argümanlarını işlemek için daha sofistike bir yöntem sağlar. Aşağıdaki program, bir veya birden fazla dosya adını ve isteğe bağlı görüntülenecek satır sayısını ayıklar:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Komut satırında `python top.py --lines =5 alpha.txt beta.txt` çalıştırıldığı zaman program, `args.lines` ögesini `5` ve `args.filenames` ögesini `['alpha.txt', 'beta.txt']` olarak ayarlar.

10.4 Hata Çıktısının Yeniden Yönlendirilmesi ve Programın Sonlandırılması

`sys` modülü ayrıca *stdin*, *stdout* ve *stderr* özelliklerine sahiptir. İkincisi, *stdout* yeniden yönlendirilmiş olsa bile bunları görünür hale getirmek için uyarılar ve hata iletileri yayımlamak için yararlıdır:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Bir programı sonlandırmak için, en kısa yol olan `sys.exit()` komutunu kullanın.

10.5 String Örüntü Eşlemesi

`re` modülü, gelişmiş string işleme için kurallı ifade araçları sağlar. Karmaşık eşleme ve manipülasyon için, kurallı ifadeler kısa ve optimize edilmiş çözümler sunar:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Basit işlemlerde "string" metodu önerilir çünkü okuması ve hata ayıklaması daha kolaydır:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matematik

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

`random` modülü rastgele seçimler yapmak için araçlar sağlar:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

`statistics` modülü sayı içeren veriler için temel istatistiksel özellikleri hesaplar (ortalama, ortanca, fark, vb.):

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

SciPy projesi <https://scipy.org> sayısal hesaplamalar için daha fazla modül içerir.

10.7 İnternet Erişimi

İnternete bağlanmak ve internet protokollerini işlemek için bazı modüller var. Bunlardan en basit ikisi; `urllib.request` URL'lerden veri çekmek, ve `smtplib` ise mail göndermek için:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # Remove trailing newline
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(İkinci örnek için bir mail sunucusunun localhost'ta çalışması gerektiğini unutmayın.)

10.8 Tarihler ve Saatler

`datetime` modülü, tarihleri ve saatleri hem basit hem de karmaşık şekillerde işlemek için sınıflar sağlar. Tarih ve saat aritmetiği desteklenirken, uygulamanın odak noktası çıktı biçimlendirmesi ve düzenlemesi için verimli üye ayıklamadır. Modül ayrıca saat dilimi farkında olan nesneleri de destekler.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Veri Sıkıştırma

Genel veri arşivleme ve sıkıştırma formatları şu modüller tarafından desteklenir: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` ve `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Performans Ölçümü

Bazı Python kullanıcıları, aynı soruna farklı yaklaşımların göreli performansını bilmek konusunda derin bir ilgi geliştirir. Python, bu soruları hemen yanıtlayan bir ölçüm aracı sağlar.

Örneğin, argümanları değiştirmek için geleneksel yaklaşım yerine dizi paketleme ve açma özelliğini kullanmak cazip olabilir. `timeit` modülü hızla sade bir performans avantajı gösterir:

```
>>> from timeit import Timer
>>> Timer('t =a; a =b; b =t', 'a =1; b =2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a =1; b =2').timeit()
0.54962537085770791
```

`timeit` 'in ince ayrıntı düzeyinin aksine, `profile` ve `pstats` modülleri, daha büyük kod bloklarında zaman açısından kritik bölümleri tanımlamak için araçlar sağlar.

10.11 Kalite Kontrolü

Yüksek kalitede yazılımlar geliştirmek için her fonksiyon için testler yazılmalıdır ve bu testler geliştirirken sık sık çalıştırmalıdır.

`doctest` modülü, bir modülü taramak ve bir programın docstrings'ine gömülü testleri doğrulamak için bir araç sağlar. Test yapısı, sonuçlarıyla birlikte tipik bir çağrıyı docstring'e kesip yapıştırma kadar basittir. Bu, kullanıcıya bir örnek sunarak dokümantasyonu geliştirir ve doctest modülünün kodun dokümantasyona göre doğru olduğundan emin olmasını sağlar:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

`unittest` modülü, `doctest` modülü gibi çaba istemeyen bir modül değildir ama daha geniş kapsamlı test setlerinin ayrı dosyalarda sağlanmasına imkân verir:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 Bataryalar Dahildir

Python'un "bataryalar dahil" felsefesi vardır. Bu , büyük paketlerin en iyi şekilde sofistike ve sağlam kapabiliteleriyle görülür. Mesela:

- `xmlrpc.client` ve `xmlrpc.server` modülleri, uzak işlem grubu çağrılarını uygulamayı neredeyse önemsiz bir görev haline getirir. Modül adlarına rağmen, XML'nin doğrudan bilgisine veya işlenmesine gerek yoktur.
- `email` paketi e-mail mesajlarını işlemek için bir kütüphanedir.MIME ve diğer **RFC 2822**-tabanlı mesajların dokümanlarını içerir. Mesaj gönderip alan `smtplib` ve `poplib` 'in aksine, e-mail paketinin derleme işlemini, kompleks mesaj yapılarının (ekler dahil) decode edilebilmesini sağlayan, internet encode işlemini ve header protokollerini uygulamak için geniş kapsamlı bir toolkit'e sahiptir.
- `json` paketi, bu popüler veri değişim biçimini ayrıştırmak için sağlam destek sağlar. `csv` modülü, genellikle veritabanları ve elektronik tablolar tarafından desteklenen Virgülle-Ayrılmış Değer biçimindeki dosyaların doğrudan okunmasını ve yazılmasını destekler. XML işleme `xml.etree.ElementTree`, `xml.dom` ve `xml.sax` paketleri tarafından desteklenir. Birlikte, bu modüller ve paketler Python uygulamaları ve diğer araçlar arasındaki veri değişimini büyük ölçüde basitleştirir.
- `sqlite3` modülü , SQLite veritabanı kütüphanesi için bir wrapper'dır. Biraz standart dışı SQL syntax'ları kullanılarak güncellenebilen ve erişilebilen kalıcı bir veritabanı sağlanabilir.
- Uluslararasılaştırma `gettext`, `locale` ve `codecs` paketi dahil olmak üzere bir dizi modül tarafından desteklenir.

BÖLÜM 11

Standart Kütüphanenin Kısa Özeti — Bölüm II

Bu ikinci özet, profesyonel programlama ihtiyaçlarını destekleyen daha gelişmiş modülleri kapsar. Bu modüller nadiren küçük komut dosyalarında bulunur.

11.1 Çıktı Biçimlendirmesi

`reprlib` modülü, büyük veya derinlemesine iç içe kapların kısaltılmış gösterimleri için özelleştirilmiş bir `repr()` sürümünü sağlar:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

`pprint` modülü, hem yerleşik hem de kullanıcı tanımlı nesnelerin yorumlayıcı tarafından okunabilecek şekilde yazdırılması üzerinde daha karmaşık kontrol sunar. Sonuç bir satırdan uzun olduğunda, "pretty printer" veri yapısını daha net bir şekilde ortaya çıkarmak için satır sonları ve girintiler ekler:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

`textwrap` modülü, metin paragraflarını belirli bir ekran genişliğine uyacak şekilde biçimlendirir:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

`locale` modülü kültüre özgü veri biçimlerinden oluşan bir veritabanına erişmektedir. Yerel ortamın biçim işlevinin gruplandırma özniteliği, sayıları grup ayırıcılarıyla biçimlendirmek için doğrudan bir yol sağlar:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Şablonlamak

`string` modülü birçok yönlü kullanıcılar tarafından düzenlemeye uygun basitleştirilmiş sözdizimine sahip şablon sınıfı `Template` içerir. Bu, kullanıcıların uygulamayı değiştirmek zorunda kalmadan uygulamalarını özelleştirmelerini sağlar.

Format, geçerli Python tanımlayıcılarıyla (alfasayısal karakterler ve alt çizgiler) `$` tarafından oluşturulan yer tutucu adlarını kullanır. Yer tutucuyu ayraçla çevrelemek, onu araya giren boşluklar olmadan daha alfasayısal harflerle takip etmenizi sağlar. `$$` yazmak tek bir `$` oluşturur:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

`substitute()` yöntemi, bir sözlükte veya anahtar sözcük bağımsız değişkeninde yer tutucu sağlandığında `KeyError` öğesini yükseltir. Adres mektup birleştirme stili uygulamalar için, kullanıcı tarafından sağlanan veriler eksik olabilir ve `safe_substitute()` yöntemi daha uygun olabilir — veriler eksikse yer tutucuları değiştirmez:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Şablon alt sınıfları özel bir sınırlayıcı belirtebilir. Örneğin, bir fotoğraf tarayıcısı için toplu yeniden adlandırma yardımcı programı, geçerli tarih, görüntü sıra numarası veya dosya biçimi gibi yer tutucular için yüzde işaretlerini kullanmayı seçebilir:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Templating için başka bir uygulama, program mantığını birden çok çıktı biçiminin ayrıntılarından ayırmaktır. Bu, XML dosyaları, düz metin raporları ve HTML web raporları için özel şablonların değiştirilmesini mümkün kılar.

11.3 İkili Veri Kaydı Düzenleriyle Çalışma

`struct` modülü, değişken uzunluklu ikili kayıt formatlarıyla çalışmak için `pack()` ve `unpack()` işlevlerini sağlar. Aşağıdaki örnek, `zipfile` modülünü kullanmadan bir ZIP dosyasındaki başlık bilgilerinin nasıl döngüye alınacağını gösterir. Paket kodları `"H"` ve `"I"` sırasıyla iki ve dört baytlık işaretsiz sayıları temsil eder. `"<"`, standart boyutta ve küçük endian bayt düzeninde olduklarını gösterir:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4 Çoklu iş parçacığı

Diş açma, sıralı olarak bağımlı olmayan görevlerin ayrıştırılması için bir tekniktir. Diğer görevler arka planda çalışırken kullanıcı girdisini kabul eden uygulamaların yanıt verme hızını artırmak için iş parçacıkları kullanılabilir. İlgili bir kullanım durumu, başka bir iş parçacığındaki hesaplamalara paralel olarak I/O çalıştırmaktadır.

Aşağıdaki kod, ana program çalışmaya devam ederken üst düzey `threading` modülünün görevleri arka planda nasıl çalıştırabileceğini gösterir:

```
import threading, zipfile

class AsyncZip(threading.Thread):
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

Çok iş parçacıklı uygulamaların temel zorluğu, verileri veya diğer kaynakları paylaşan iş parçacıklarını koordine etmektir. Bu amaçla, iş parçacığı modülü, kilitler, olaylar, koşul değişkenleri ve semaforlar dahil olmak üzere bir dizi senkronizasyon ilkesi sağlar.

Bu araçlar güçlü olsa da, küçük tasarım hataları, yeniden üretilmesi zor sorunlara neden olabilir. Bu nedenle, görev koordinasyonuna yönelik tercih edilen yaklaşım, bir kaynağa tüm erişimi tek bir iş parçacığında yoğunlaştırmak ve ardından bu iş parçacığını diğer iş parçacıklarından gelen isteklerle beslemek için `queue` modülünü kullanmaktır. İş parçacıkları arası iletişim ve koordinasyon için `Queue` nesnelerini kullanan uygulamaların tasarımı daha kolay, daha okunaklı ve daha güvenilirdir.

11.5 Günlükleme

`logging` modülü, tam özellikli ve esnek bir kayıt sistemi sunar. En basit haliyle, günlük mesajları bir dosyaya veya `sys.stderr` adresine gönderilir:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Bu, aşağıdaki çıktıyı üretir:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Varsayılan olarak, bilgi ve hata ayıklama mesajları bastırılır ve çıktı standart hataya gönderilir. Diğer çıktı seçenekleri, mesajları e-posta, datagramlar, yuvalar veya bir HTTP Sunucusuna yönlendirmeyi içerir. Yeni filtreler mesaj önceliğine göre farklı yönlendirme seçebilir: `DEBUG`, `INFO`, `WARNING`, `ERROR` , ve `CRITICAL`.

Günlük kaydı sistemi, doğrudan Python'dan yapılandırılabilir veya uygulamayı değiştirmeden özelleştirilmiş günlük kaydı için kullanıcı tarafından düzenlenebilir bir yapılandırma dosyasından yüklenebilir.

11.6 Zayıf Başvurular

Python otomatik bellek yönetimi yapar (çoğu nesne için referans sayımı ve döngüleri ortadan kaldırmak için *garbage collection*). Hafıza, ona yapılan son başvurunun ortadan kaldırılmasından kısa bir süre sonra serbest bırakılır.

Bu yaklaşım çoğu uygulama için iyi sonuç verir ancak bazen nesneleri yalnızca başka bir şey tarafından kullanıldıkları sürece izlemeye ihtiyaç duyulur. Ne yazık ki, sadece onları izlemek onları kalıcı kılan bir referans oluşturur. `weakref` modülü, referans oluşturmadan nesneleri izlemek için araçlar sağlar. Nesneye artık ihtiyaç duyulmadığında, zayıf referans tablosundan otomatik olarak kaldırılır ve zayıf referans nesneleri için bir geri arama tetiklenir. Tipik uygulamalar, oluşturması pahalı olan nesneleri önbelleğe almayı içerir:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Listelerle Çalışma Araçları

Birçok veri yapısı ihtiyacı yerleşik liste türüyle karşılanabilir. Ancak bazen farklı performans ödünleşimleri ile alternatif uygulamalara ihtiyaç duyulmaktadır.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Alternatif liste uygulamalarına ek olarak, kütüphane ayrıca sıralanmış listeleri işlemek için işlevlere sahip `bisect` modülü gibi başka araçlar da sunar:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

`heapq` modülü, düzenli listelere dayalı yığınları uygulamak için işlevler sağlar. En düşük değerli giriş her zaman sıfır konumunda tutulur. Bu, en küçük öğeye tekrar tekrar erişen ancak tam liste sıralamasını çalıştırmak istemeyen uygulamalar için kullanışlıdır:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- tam ondalık gösterim gerektiren finansal uygulamalar ve diğer kullanımlar,
- hassasiyet üzerinde kontrol,
- yasal veya düzenleyici gereklilikleri karşılamak için yuvarlama üzerinde kontrol,
- önemli ondalık basamakların izlenmesi veya
- kullanıcının sonuçların elle yapılan hesaplamalarla eşleşmesini beklediği uygulamalar.

Örneğin, 70 sentlik bir telefon ücretinde 5% vergi hesaplamak ondalık kayan nokta ve ikili kayan nokta için farklı sonuçlar verir. Sonuçlar en yakın küsurata yuvarlanırsa fark önemli hale gelir:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

`Decimal` sonucu, iki basamaklı anlamlı çarpanlardan otomatik olarak dört basamaklı anlamlılık çıkaran, sonunda bir sıfır tutar. Ondalık, matematiği elle yapıldığı gibi yeniden üretir ve ikili kayan nokta ondalık miktarları tam olarak temsil edemediğinde ortaya çıkabilecek sorunları önler.

Tam gösterim, `Decimal` sınıfının, ikili kayan nokta için uygun olmayan modlo hesaplamaları ve eşitlik testleri gerçekleştirmesini sağlar:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

`decimal` modülü, aritmetikle birlikte gerektiği kadar hassasiyet sağlar:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

BÖLÜM 12

Sanal Ortamlar ve Paketler

12.1 Tanıtım

Python uygulamaları genellikle standart kütüphanenin bir parçası olmayan paketleri ve modülleri kullanır. Uygulama bazen kütüphanenin spesifik bir sürümüne ihtiyaç duyar, çünkü uygulama belirli bir hatanın düzeltilmiş olmasını veya uygulamanın kütüphanenin arabiriminin eski bir sürümü kullanılarak yazılmasını gerektirebilir.

Bu, bir Python yüklemesinin, her uygulamanın gereksinimlerini karşılamasının mümkün olmayabileceği anlamına gelir. Bir A uygulaması belirli bir modülün 1.0 sürümüne, bir B uygulaması ise 2.0 sürümüne ihtiyaç duyuyorsa, sürümlerin farklı olmasından dolayı versiyon 1.0 veya 2.0'ı yüklemek A veya B uygulamasından birini çalışmaz hale getirecektir.

Bu sorunun çözümü, spesifik bir Python sürümü için Python yüklemesi ve bir dizi ek paket içeren bağımsız bir dizin ağacı olan *virtual environment* (sanal ortam) oluşturmaktır.

Bu sayede farklı uygulamalar farklı sanal ortamlar kullanabilir. Çakışan gereksinimlerin önceki örneğini çözmek için, A uygulamasının sanal ortamında sürüm 1.0 yüklüyken, B uygulamasının sanal ortamında sürüm 2.0 yüklü olabilir. B uygulaması bir kitaplığın sürüm 3.0'a yükseltilmesini gerektiriyorsa, bu uygulama A'nın ortamını etkilemez.

12.2 Sanal Ortamlar Oluşturma

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

Sanal ortam oluşturmak için, yerleştirmek istediğiniz dizine karar verin ve `venv` modülünü dizin yolu ile bir komut dosyası olarak çalıştırın:

```
python -m venv tutorial-env
```

Bu, eğer yoksa `tutorial-env` dizinini oluşturur ve ayrıca Python derleyicisinin bir kopyasını ve çeşitli destekleyici dosyaları içeren dizinler oluşturur.

Sanal ortam için ortak bir dizin konumu `.venv` 'dir. Bu ad, dizini genellikle kabuğunuzda gizli tutar ve böylece dizinin neden var olduğunu açıklayan bir ad verirken aradan uzak tutar. Ayrıca, bazı araç çalıştırmanın desteklediği `.env` ortam değişkeni tanım dosyalarıyla çakışmayı önler.

Sanal bir ortam oluşturduktan sonra onu etkinleştirebilirsiniz.

Windows'da çalıştır:

```
tutorial-env\Scripts\activate
```

Unix veya MacOS'ta çalıştır:

```
source tutorial-env/bin/activate
```

(Bu komut dosyası bash kabuğu için yazılmıştır. Eğer **csh** veya **fish** kabuklarını kullanıyorsanız, bunun yerine `activate.csh` veya `activate.fish` komut dosyalarını kullanmanız gerekmektedir.)

Sanal ortamı etkinleştirmek, hangi sanal ortamı kullandığınızı göstermek için kabuğunuzun görünüşünü değiştirir ve ortamı değiştirerek `python` komutunun belirlediğiniz spesifik Python kurulumunu ve sürümünü çalıştırmasını sağlar. Mesela:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

Bir sanal ortamı devre dışı bırakmak için şunu yazın:

```
deactivate
```

terminalin içine.

12.3 Paketleri pip ile Yönetme

pip adlı bir program kullanarak paketleri yükleyebilir, yükseltebilir ve kaldırabilirsiniz. Varsayılan olarak `pip`, Python Paket Dizini 'nden paketler yükler. Python Paket Dizini'ne web tarayıcınızdan giderek göz atabilirsiniz.

`pip` bir dizi alt komut içerir: "install" (yükle), "uninstall" (kaldır), "freeze" (dondur), vb. (`pip` için eksiksiz dokümantasyon için installing-index rehberine bakın.)

Paketin adını belirterek paketin en son sürümünü yükleyebilirsiniz:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Paketin belirli bir sürümünü, paket adını ve ardından == ve sürüm numarasını vererek de yükleyebilirsiniz:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Bu komutu tekrar çalıştırırsanız, `pip` istenen sürümün kurulu olduğunu fark edecek ve hiçbir şey yapmayacaktır. Bu sürümü almak için farklı bir sürüm numarası sağlayabilir veya paketi en son sürüme yükseltmek için `python -m pip install --upgrade` komutunu çalıştırabilirsiniz:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` ve ardından gelen bir veya daha fazla paket adı, paketleri sanal ortamdan kaldıracaktır.

`python -m pip show` belirli bir paket hakkındaki bilgileri görüntüleyecektir:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` sanal ortamda yüklü olan tüm paketleri gösterecektir:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` yüklü paketlerin benzer bir listesini üretecektir, ancak çıktı `python -m pip install`'ın beklediği biçimi kullanır. Genel bir kullanım bu listeyi bir `requirements.txt` dosyasına koymaktır:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

`requirements.txt` daha sonra sürüm denetimine kaydedilebilir ve bir uygulamanın parçası olarak gönderilebilir. Kullanıcılar daha sonra gerekli tüm paketleri `install -r` ile yükleyebilir:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the installing-index guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the Python packaging user guide.

BÖLÜM 13

Sırada Ne Var?

Bu öğreticiyi okumak muhtemelen Python'u kullanmaya olan ilginizi pekiştirdi — Python'u gerçek dünyadaki sorunlarınızı çözmek için kullanmaya istekli olmalısınız. Daha fazla bilgi edinmek için nereye gitmelisiniz?

Bu öğretici Python'un dokümantasyon kümesinin bir parçasıdır. Kümedeki diğer bazı belgeler şunlardır:

- library-index:

  Standart kütüphanedeki türler, fonksiyonlar ve modüller hakkında eksiksiz (özlü de olsa) referans materyali sağlayan bu kılavuza göz atmalısınız. Standart Python dağıtımı *birçok* ek kod içerir. Unix posta kutularını okumak, belgeleri HTTP yoluyla almak, rastgele sayılar oluşturmak, komut satırı seçeneklerini ayrıştırmak, verileri sıkıştırmak ve diğer birçok görev için modüller vardır. Kütüphane Referansını gözden geçirmek size nelerin mevcut olduğu hakkında bir fikir verecektir.

- installing-index , diğer Python kullanıcıları tarafından yazılan ek modüllerin nasıl yükleneceğini açıklar.

- reference-index: Python sözdiziminin ve anlambiliminin ayrıntılı bir açıklaması. Ağır bir metin, ancak dilin kendisi için eksiksiz bir rehber olarak yararlıdır.

Diğer Python kaynakları:

- https://www.python.org: Başlıca Python web sitesi. Web'deki Python ile ilgili sayfalara yönelik kod, belgeler ve işaretçiler içerir.

- https://docs.python.org: Python belgelerine hızlı erişim.

- https://pypi.org: Daha önce Cheese Shop[1], olarak da adlandırılan Python Paket Dizini, kullanıcılar tarafından oluşturulan indirilebilir Python modülleri dizinidir. Kodlarınızı yayınlamaya başladıktan sonra, başkalarının bulabilmesi için buraya kaydedebilirsiniz.

- https://code.activestate.com/recipes/langs/python/: Python Yemek Tarifleri; kod örnekleri, daha geniş çaplı modüller ve kullanışlı kodların büyük bir koleksiyonudur. Özellikle dikkat çekici katkılar Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3) adlı bir kitapta toplanır.

- https://pyvideo.org, konferanslardan ve kullanıcı grubu toplantılarından Python ile ilgili videolara bağlantılar toplar.

- https://scipy.org: Scientific Python projesi, hızlı dizi hesaplamaları ve manipülasyonları için modüllerin yanı sıra doğrusal cebir, Fourier dönüşümleri, doğrusal olmayan çözücüler, rastgele sayı dağılımları, istatistiksel analiz ve benzeri şeyler için bir dizi paket içerir.

[1] "Cheese Shop" Monty Python'un bir çizimidir: bir müşteri peynir dükkanına girer, ancak istediği peynir ne olursa olsun, tezgahtar elinde olmadığını söyler.

Python ile ilgili sorular ve sorun raporları için haber grubu *comp.lang.python* adresine gönderebilir veya bunları python-list@python.org posta listesine gönderebilirsiniz. Haber grubu ve posta listesi aynı ağ içinde olduğundan, birine gönderilen iletiler otomatik olarak diğerine iletilir. Günde yüzlerce gönderi geliyor, sorular soruluyor (ve yanıtlanıyor), yeni özellikler öneriliyor ve yeni modüller duyuruluyor. Posta listesi arşivleri için https://mail.python.org/pipermail/.

Göndermeden önce, Sık Sorulan Sorular (SSS olarak da adlandırılır) listesini kontrol etmeyi unutmayın. SSS, tekrar tekrar gelen soruların çoğunu yanıtlar ve halihazırda sorununuzun çözümünü içeriyor olabilir.

BÖLÜM 14

Etkileşimli Girdi Düzenleme ve Geçmiş İkame

Python yorumlayıcısının bazı sürümleri, Korn kabuğunda ve GNU Bash kabuğunda bulunan tesislere benzer şekilde, mevcut girdi satırının ve geçmiş ikamesinin düzenlenmesini destekler. Bu, çeşitli düzenleme stillerini destekleyen GNU Readline kütüphanesi kullanılarak gerçekleştirilir. Bu kütüphanenin burada kopyalamayacağımız kendi belgeleri vardır.

14.1 Tab Tamamlama ve Geçmiş Düzenleme

Completion of variable and module names is automatically enabled at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2 Etkileşimli Yorumlayıcıya Alternatifler

Bu olanak, tercümanın önceki sürümleriyle karşılaştırıldığında ileriye doğru atılmış çok büyük bir adımdır; ancak, bazı eksiklikler var: Devam satırlarında uygun girinti önerilmiş olsaydı iyi olurdu (ayrıştırıcı, daha sonra bir girinti jetonunun gerekip gerekmediğini bilir).

Oldukça uzun bir süredir var olan alternatif geliştirilmiş etkileşimli yorumlayıcı, tab tamamlama, nesne keşfi ve gelişmiş geçmiş yönetimi özelliklerine sahip IPython 'dur. Ayrıca, tamamen özelleştirilebilir ve diğer uygulamalara gömülebilir. Bir başka benzer geliştirilmiş etkileşimli ortam da bpython 'dur.

BÖLÜM 15

Floating-Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction `0.625` has value 6/10 + 2/100 + 5/1000, and in the same way the **binary** fraction `0.101` has value 1/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Ne yazık ki, ondalık kesirlerin çoğu tam olarak ikili kesir olarak gösterilemez. Bunun bir sonucu olarak, genel olarak, girdiğiniz ondalık kayan noktalı sayılar, makinede gerçekte depolanan ikili kayan noktalı sayılar tarafından yalnızca yaklaşık olarak gösterilir.

Problem ilk başta 10 tabanında daha kolay anlaşılabilir. 1/3 kesrini düşünün. Bunu 10 tabanında bir kesir olarak yaklaşık olarak hesaplayabilirsiniz:

```
0.3
```

ya da daha iyisi

```
0.33
```

ya da daha iyisi

```
0.333
```

ve bunun gibi. Kaç basamak yazmak isterseniz isteyin, sonuç hiçbir zaman tam olarak 1/3 olmayacak, ancak 1/3'ün giderek daha iyi bir yaklaşımı olacaktır.

Aynı şekilde, kaç tane 2 tabanı basamağı kullanmak isterseniz isteyin, 0.1 ondalık değeri tam olarak 2 tabanı kesri olarak gösterilemez. Taban 2'de 1/10 sonsuza kadar tekrar eden bir kesirdir

```
0.0001100110011001100110011001100110011001100110011...
```

Herhangi bir sonlu bit sayısında durduğunuzda bir yaklaşık değer elde edersiniz. Bugün çoğu makinede, kayan sayılar, pay en anlamlı bitten başlayarak ilk 53 bit kullanılarak ve payda ikinin kuvveti olarak ikili bir kesir kullanılarak yaklaştırılır. 1/10 durumunda ikili kesir, 1/10'un gerçek değerine yakın ancak tam olarak eşit olmayan `3602879701896397 / 2 ** 55` şeklindedir.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead:

```
>>> 1 / 10
0.1
```

Unutmayın, yazdırılan sonuç 1/10'un tam değeri gibi görünse de, saklanan gerçek değer temsil edilebilir olan en yakın ikili kesirdir.

İlginç bir şekilde, aynı en yakın yaklaşık ikili kesri paylaşan birçok farklı ondalık sayı vardır. Örneğin, `0.1` ve `0.1000000000000001` ve `0.1000000000000000055511151231257827021181583404541015625` sayılarının tümü `3602879701896397 / 2 ** 55` ile yaklaştırılır. Bu ondalık değerlerin tümü aynı yaklaşımı paylaştığından, `eval(repr(x)) == x` değişmezi korunarak bunlardan herhangi biri görüntülenebilir.

Geçmişte, Python komut istemi ve yerleşik `repr()` işlevi 17 anlamlı basamağa sahip olanı, `0.1000000000000001` 'i seçerdi. Python 3.1'den itibaren, Python (çoğu sistemde) artık bunlardan en kısa olanı seçebilmekte ve basitçe `0.1` 'i görüntüleyebilmektedir.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Bunun gerçek anlamda bir yanılsama olduğunun farkına varmak önemlidir: gerçek makine değerinin *görüntüsünü* yuvarlıyorsunuz.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with `round()` function cannot help:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `math.isclose()` function can be useful for comparing inexact values:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the `round()` function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See Examples of Floating Point Problems for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see The Perils of Floating Point for a more complete account of other common surprises.

As that says near the end, "there are no easy answers." Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

Patolojik durumlar mevcut olsa da, kayan noktalı aritmetiğin sıradan kullanımı için, nihai sonuçlarınızın görüntüsünü beklediğiniz ondalık basamak sayısına yuvarlarsanız, sonunda beklediğiniz sonucu görürsünüz. `str()` genellikle yeterlidir ve daha ince kontrol için formatstrings içindeki `str.format()` yönteminin biçim belirleyicilerine bakın.

Tam ondalık gösterim gerektiren durumlar için, muhasebe uygulamaları ve yüksek hassasiyetli uygulamalar için uygun ondalık aritmetiği uygulayan `decimal` modülünü kullanmayı deneyin.

Kesin aritmetiğin bir başka biçimi, rasyonel sayılara dayalı aritmetik uygulayan `fractions` modülü tarafından desteklenir (böylece 1/3 gibi sayılar tam olarak temsil edilebilir).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Temsil tam olduğundan, değerleri Python'un farklı sürümleri arasında güvenilir bir şekilde taşımak (platform bağımsızlığı) ve aynı biçimi destekleyen diğer dillerle (Java ve C99 gibi) veri alışverişi yapmak için kullanışlıdır.

Another helpful tool is the `sum()` function which helps mitigate loss-of-precision during summation. It uses extended precision for intermediate rounding steps as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the "lost digits" as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.0032449368398082 27]
>>> float(sum(map(Fraction, arr)))   # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in standard precision
...
>>> total                            # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1 Temsil Hatası

Bu bölüm "0.1" örneğini ayrıntılı olarak açıklamakta ve bu gibi durumların tam analizini kendiniz nasıl yapabileceğinizi göstermektedir. İkili kayan nokta gösterimine temel düzeyde aşina olunduğu varsayılmaktadır.

Temsil hatası, bazı (aslında çoğu) ondalık kesirlerin tam olarak ikili (taban 2) kesirler olarak temsil edilemeyeceği gerçeğini ifade eder. Bu, Python'un (veya Perl, C, C++, Java, Fortran ve diğerlerinin) genellikle beklediğiniz tam ondalık sayıyı göstermemesinin başlıca nedenidir.

Why is that? 1/10 is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 "double precision" values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

şu şekilde

```
J ~= 2**N / 10
```

and recalling that *J* has exactly 53 bits (is `>= 2**52` but `< 2**53`), the best value for *N* is 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in IEEE 754 double precision is:

```
7205759403792794 / 2 ** 56
```

Hem pay hem de paydayı ikiye böldüğünüzde kesir şuna indirgenir:

```
3602879701896397 / 2 ** 55
```

Aslında bölümü yukarı yuvarladığımız için değerin 1/10'dan biraz daha büyük olduğuna dikkat edin; yukarı yuvarlamamış olsaydık, bölüm 1/10'dan biraz daha küçük olurdu. Ancak hiçbir durumda *tam olarak* 1/10 olamaz!

So the computer never "sees" 1/10: what it sees is the exact fraction given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by 10**55, we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value 0.1000000000000000055511151231257827021181583404541015625. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The `fractions` and `decimal` modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

BÖLÜM 16

Ek Bölüm

16.1 Etkileşimli Mod

There are two variants of the interactive *REPL*. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with `curses` support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see using-on-controlling-color for details. Function keys provide some additional functionality. `F1` enters the interactive help browser `pydoc`. `F2` allows for browsing command-line history with neither output nor the `>>>` and `...` prompts. `F3` enters "paste mode", which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the `PYTHON_BASIC_REPL` environment variable.

16.1.1 Hata İşleme

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Yarıda kesme karakterinin (genellikle `Control-C` veya `Delete`) birincil veya ikincil istemine yazılması girişi iptal eder ve birincil istemi döndürür.[1] Bir komut yürütülürken bir yarıda kesme karakteri yazmak `KeyboardInterrupt` özel durumuna neden olur ve bu özel durum `try` deyimi tarafından işlenebilir.

16.1.2 Yürütülebilir Python Komut Dosyaları

BSD türü Unix sistemlerinde Python komut dosyaları, :: satırı koyarak kabuk komut dosyaları gibi doğrudan yürütülebilir hale getirilebilir:

```
#!/usr/bin/env python3
```

[1] GNU Readline paketiyle ilgili bir sorun bunu engelleyebilir.

(yorumlayıcının kullanıcının PATH) komut dosyasının başında olduğunu ve dosyaya yürütülebilir bir mod verdiğini varsayarsak. #! dosyanın ilk iki karakteri olmalıdır. Bazı platformlarda, bu ilk satırın Windows ('\r\n') satır sonuyla değil, Unix stili bir satır sonuyla ('\n') bitmesi gerekir. Python'da yorum başlatmak için karma veya pound karakteri olan '#' kullanıldığını unutmayın.

Komut dosyasına **chmod** komutu kullanılarak yürütülebilir mod veya izin verilebilir.

```
$ chmod +x myscript.py
```

Windows sistemlerinde, "yürütülebilir mod" kavramı yoktur. Python yükleyicisi otomatik olarak .py dosyalarını python.exe ile ilişkilendirir, böylece python dosyasına çift tıklama komut dosyası olarak çalıştırılır. Uzantı .pyw 'de olabilir, bu durumda normalde görünen konsol penceresi bastırılır.

16.1.3 Etkileşimli Başlangıç Dosyası

Python'u etkileşimli olarak kullandığınızda, yorumlayıcı her başlatıldığında bazı standart komutların yürütülmesi genellikle kullanışlıdır. Bunu, başlatma komutlarınızı içeren bir dosyanın adına PYTHONSTARTUP adlı bir ortam değişkeni ayarlayarak yapabilirsiniz. Bu, Unix kabuklarının .profile özelliğine benzer.

Bu dosya Python komutları bir komut dosyasından okuduğunda değil, yalnızca etkileşimli oturumlarda okunur. Python komutları bir komut dosyasından okuduğunda, /dev/tty komutların açık kaynağı olarak verildiğinde değil (aksi takdirde etkileşimli bir oturum gibi davranır). Etkileşimli komutların yürütüldüğü aynı ad alanında yürütülür, böylece tanımladığı veya aldığı nesneler etkileşimli oturumda nitelik olmadan kullanılabilir. Bu dosyadaki sys.ps1 ve sys.ps2 istemlerini de değiştirebilirsiniz.

Geçerli dizinden ek bir başlangıç dosyası okumak istiyorsanız, bunu genel başlangıç dosyasında if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read() gibi bir kod kullanarak programlayabilirsiniz. Başlangıç dosyasını bir komut dosyasında kullanmak istiyorsanız, bunu komut dosyasında açıkça yapmalısınız:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 Özelliştirme Modülleri

Python provides two hooks to let you customize it: sitecustomize and usercustomize. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Artık bu dizinde usercustomize.py adlı bir dosya oluşturabilir ve içine istediğiniz her şeyi koyabilirsiniz. Otomatik içe aktarmayı devre dışı bırakmak için -s seçeneğiyle başlatılmadıkça Python'un her çağrısını etkiler.

sitecustomize works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before usercustomize. See the documentation of the site module for more details.

EK A

Sözlük

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Şunlara başvurabilir:

  - The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
  - `Elipsis` yerleşik sabiti.

soyut temel sınıf
: Soyut temel sınıflar *duck-typing* 'i, `hasattr()` gibi diğer teknikler beceriksiz veya tamamen yanlış olduğunda arayüzleri tanımlamanın bir yolunu sağlayarak tamamlar (örneğin sihirli yöntemlerle). ABC'ler, bir sınıftan miras almayan ancak yine de `isinstance()` ve `issubclass()` tarafından tanınan sanal alt sınıfları tanıtır; `abc` modül belgelerine bakın. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). `abc` modülü ile kendi ABC'lerinizi oluşturabilirsiniz.

dipnot
: Bir değişkenle, bir sınıf niteliğiyle veya bir fonksiyon parametresiyle veya bir dönüş değeriyle ilişkilendirilen, gelenek olarak *type hint* biçiminde kullanılan bir etiket.

  Yerel değişkenlerin açıklamalarına çalışma zamanında erişilemez, ancak global değişkenlerin, sınıf niteliklerinin ve işlevlerin açıklamaları, sırasıyla modüllerin, sınıfların ve işlevlerin `__annotations__` özel özelliğinde saklanır.

  Bu işlevi açıklayan *variable annotation*, *function annotation*, **PEP 484** ve **PEP 526**'e bakın. Ek açıklamalarla çalışmaya ilişkin en iyi uygulamalar için ayrıca bkz. annotations-howto.

argüman
: Fonksiyon çağrılırken bir *function* 'a (veya *method*) geçirilen bir değer. İki tür argüman vardır:

  - *keyword argument*: bir işlev çağrısında bir tanımlayıcının (ör. `ad =`) önüne geçen veya bir sözlükte `**` ile başlayan bir değer olarak geçirilen bir argüman. Örneğin, `3` ve `5`, aşağıdaki `complex()`: çağrılarında anahtar kelimenin argümanleridir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: anahtar kelime argümanı olmayan bir argüman. Konumsal argümanler, bir argüman listesinin başında görünebilir ve/veya `*` ile başlayan bir *iterable* öğesinin öğeleri olarak iletilebilir. Örneğin, `3` ve `5`, aşağıdaki çağrılarda konumsal argümanlerdir:

```
complex(3, 5)
complex(*(3, 5))
```

Argümanler, bir fonksiyon gövdesindeki adlandırılmış yerel değişkenlere atanır. Bu atamayı yöneten kurallar için calls bölümüne bakın. Sözdizimsel olarak, bir argümanı temsil etmek için herhangi bir ifade kullanılabilir; değerlendirilen değer yerel değişkene atanır.

Ayrıca *parameter* sözlüğü girişine, the difference between arguments and parameters hakkındaki SSS sorusuna ve **PEP 362** 'ye bakın.

asenkron bağlam yöneticisi
An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asenkron jeneratör
asynchronous generator iterator döndüren bir işlev. Bir `async for` döngüsünde kullanılabilen bir dizi değer üretmek için `yield` ifadeleri içermesi dışında `async def` ile tanımlanmış bir eşyordam işlevine benziyor.

Genellikle bir asenkron üreteç işlevine atıfta bulunur, ancak bazı bağlamlarda bir *asynchronous generator iterator* 'e karşılık gelebilir. Amaçlanan anlamın net olmadığı durumlarda, tam terimlerin kullanılması belirsizliği önler.

Bir asenkron üretici fonksiyonu, `await` ifadelerinin yanı sıra `async for` ve `async with` ifadeleri içerebilir.

asenkron jeneratör yineleyici
Bir *asynchronous generator* işlevi tarafından oluşturulan bir nesne.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

eşzamansız yinelenebilir
An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asenkron yineleyici
An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

nitelik
Noktalı ifadeler kullanılarak adıyla başvurulan bir nesneyle ilişkili değer. Örneğin, *o* nesnesinin *a* özniteliği varsa, bu nesneye *o.a* olarak başvurulur.

Bir nesneye, eğer nesne izin veriyorsa, örneğin `setattr()` kullanarak, adı identifiers tarafından tanımlandığı gibi tanımlayıcı olmayan bir öznitelik vermek mümkündür. Böyle bir özniteliğe noktalı bir ifade kullanılarak erişilemez ve bunun yerine `getattr()` ile alınması gerekir.

beklenebilir
An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL
Benevolent Dictator For Life, namı diğer Guido van Rossum, Python'un yaratıcısı.

ikili dosya
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

Ayrıca `str` nesnelerini okuyabilen ve yazabilen bir dosya nesnesi için *text file* 'a bakın.

ödünç alınan referans
In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

borrowed reference üzerinde `Py_INCREF()` çağırmak, nesnenin ödünç alınanın son kullanımından önce yok edilemediği durumlar dışında, onu yerinde bir *strong reference* 'a dönüştürmek için tavsiye edilir. referans. `Py_NewRef()` işlevi, yeni bir *strong reference* oluşturmak için kullanılabilir.

bayt benzeri nesne
bufferobjects 'i destekleyen ve bir C-*contiguous* arabelleğini dışa aktarabilen bir nesne. Bu, tüm `bytes`, `bytearray` ve `array.array` nesnelerinin yanı sıra birçok yaygın `memoryview` nesnesini içerir. Bayt benzeri nesneler, ikili verilerle çalışan çeşitli işlemler için kullanılabilir; bunlara sıkıştırma, ikili dosyaya kaydetme ve bir soket üzerinden gönderme dahildir.

Bazı işlemler, değişken olması için ikili verilere ihtiyaç duyar. Belgeler genellikle bunlara "okuma-yazma bayt benzeri nesneler" olarak atıfta bulunur. Örnek değiştirilebilir arabellek nesneleri `bytearray` ve bir `bytearray` `memoryview` içerir. Diğer işlemler, ikili verilerin değişmez nesnelerde ("salt okunur bayt benzeri nesneler") depolanmasını gerektirir; bunların örnekleri arasında `bytes` ve bir `bytes` nesnesinin `memoryview` bulunur.

bayt kodu
Python kaynak kodu, bir Python programının CPython yorumlayıcısındaki dahili temsili olan bayt kodunda derlenir. Bayt kodu ayrıca `.pyc` dosyalarında önbelleğe alınır, böylece aynı dosyanın ikinci kez çalıştırılması daha hızlı olur (kaynaktan bayt koduna yeniden derleme önlenebilir). Bu "ara dilin", her bir bayt koduna karşılık gelen makine kodunu yürüten bir *sanal makine* üzerinde çalıştığı söylenir. Bayt kodlarının farklı Python sanal makineleri arasında çalışması veya Python sürümleri arasında kararlı olması beklenmediğini unutmayın.

Bayt kodu talimatlarının bir listesi bytecodes dokümanında bulunabilir.

çağırılabilir
Bir çağrılabilir, muhtemelen bir dizi argümanla (bkz. *argument*) ve aşağıdaki sözdizimiyle çağrılabilen bir nesnedir:

```
callable(argument1, argument2, argumentN)
```

Bir *fonksiyon* ve uzantısı olarak bir *metot* bir çağrılabilirdir. `__call__()` yöntemini uygulayan bir sınıf örneği de bir çağrılabilirdir.

geri çağırmak
Gelecekte bir noktada yürütülecek bir argüman olarak iletilen bir alt program işlevi.

sınıf
Kullanıcı tanımlı nesneler oluşturmak için bir şablon. Sınıf tanımları normalde sınıfın örnekleri üzerinde çalışan yöntem tanımlarını içerir.

sınıf değişkeni
Bir sınıfta tanımlanmış ve yalnızca sınıf düzeyinde (yani sınıfın bir örneğinde değil) değiştirilmesi amaçlanan bir değişken.

closure variable
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

karmaşık sayı
Tüm sayıların bir reel kısım ve bir sanal kısım toplamı olarak ifade edildiği bilinen gerçek sayı sisteminin bir uzantısı. Hayali sayılar, hayali birimin gerçek katlarıdır (`-1` 'in karekökü), genellikle matematikte `i` veya mühendislikte `j` ile yazılır. Python, bu son gösterimle yazılan karmaşık sayılar için yerleşik desteğe sahiptir; hayali kısım bir `j` son ekiyle yazılır, örneğin `3+1j`. `math` modülünün karmaşık eş değerlerine erişmek için `cmath` kullanın. Karmaşık sayıların kullanımı oldukça gelişmiş bir matematiksel özelliktir. Onlara olan ihtiyacın farkında değilseniz, onları güvenle görmezden gelebileceğiniz neredeyse kesindir.

context
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

bağlam yöneticisi
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

bağlam değişkeni
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

bitişik
Bir arabellek, *C-bitişik* veya *Fortran bitişik* ise tam olarak bitişik olarak kabul edilir. Sıfır boyutlu arabellekler C ve Fortran bitişiktir. Tek boyutlu dizilerde, öğeler sıfırdan başlayarak artan dizinler sırasına göre bellekte yan yana yerleştirilmelidir. Çok boyutlu C-bitişik dizilerde, öğeleri bellek adresi sırasına göre ziyaret ederken son dizin en hızlı şekilde değişir. Ancak, Fortran bitişik dizilerinde, ilk dizin en hızlı şekilde değişir.

eşyordam
Eşyordamlar, altyordamların daha genelleştirilmiş bir biçimidir. Alt programlara bir noktada girilir ve başka bir noktada çıkılır. Eşyordamlar birçok farklı noktada girilebilir, çıkılabilir ve devam ettirilebilir. `async def` ifadesi ile uygulanabilirler. Ayrıca bakınız **PEP 492**.

eşyordam işlevi
Bir *coroutine* nesnesi döndüren bir işlev. Bir eşyordam işlevi `async def` ifadesiyle tanımlanabilir ve `await`, `async for` ve `async with` anahtar kelimelerini içerebilir. Bunlar **PEP 492** tarafından tanıtıldı.

CPython
Python programlama dilinin python.org üzerinde dağıtıldığı şekliyle kurallı uygulaması. "CPython" terimi, gerektiğinde bu uygulamayı Jython veya IronPython gibi diğerlerinden ayırmak için kullanılır.

current context
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

dekoratör
Genellikle `@wrapper` sözdizimi kullanılarak bir işlev dönüşümü olarak uygulanan, başka bir işlevi döndüren bir işlev. Dekoratörler için yaygın örnekler şunlardır: `classmethod()` ve `staticmethod()`.

Dekoratör sözdizimi yalnızca sözdizimsel şekerdir, aşağıdaki iki işlev tanımı anlamsal olarak eş değerdir:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Aynı kavram sınıflar için de mevcuttur, ancak orada daha az kullanılır. Dekoratörler hakkında daha fazla bilgi için function definitions ve class definitions belgelerine bakın.

tanımlayıcı
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Tanımlayıcıların yöntemleri hakkında daha fazla bilgi için, bkz. descriptors veya Descriptor How To Guide.

sözlük
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

sözlük anlama
Öğelerin tümünü veya bir kısmını yinelenebilir bir şekilde işlemenin ve sonuçları içeren bir sözlük döndürmenin kompakt bir yolu. `results = {n: n ** 2 for range(10)}`, `n ** 2` değerine eşlenmiş `n` anahtarını içeren bir sözlük oluşturur. Bkz. comprehensions.

sözlük görünümü
`dict.keys()`, `dict.values()` ve `dict.items()` 'den döndürülen nesnelere sözlük görünümleri denir. Sözlüğün girişleri üzerinde dinamik bir görünüm sağlarlar; bu, sözlük değiştiğinde görünümün bu değişiklikleri yansıttığı anlamına gelir. Sözlük görünümünü tam liste olmaya zorlamak için `list(dictview)` kullanın. Bakınız dict-views.

belge dizisi
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

ördek yazma
Doğru arayüze sahip olup olmadığını belirlemek için bir nesnenin türüne bakmayan bir programlama stili; bunun yerine, yöntem veya nitelik basitçe çağrılır veya kullanılır ("Ördek gibi görünüyorsa ve ördek gibi vaklıyorsa, ördek olmalıdır.") İyi tasarlanmış kod, belirli türlerden ziyade arayüzleri vurgulayarak, polimorfik ikameye izin vererek esnekliğini artırır. Ördek yazma, `type()` veya `isinstance()` kullanan testleri önler. (Ancak, ördek yazmanın *abstract base class* ile tamamlanabileceğini unutmayın.) Bunun yerine, genellikle `hasattr()` testleri veya *EAFP* programlamasını kullanır.

EAFP
Af dilemek izin almaktan daha kolaydır. Bu yaygın Python kodlama stili, geçerli anahtarların veya niteliklerin varlığını varsayar ve varsayımın yanlış çıkması durumunda istisnaları yakalar. Bu temiz ve hızlı stil, birçok `try` ve `except` ifadesinin varlığı ile karakterize edilir. Teknik, C gibi diğer birçok dilde ortak olan *LBYL* stiliyle çelişir.

ifade (değer döndürür)
Bir değere göre değerlendirilebilecek bir sözdizimi parçası. Başka bir deyişle, bir ifade, tümü bir değer döndüren sabit değerler, adlar, öznitelik erişimi, işleçler veya işlev çağrıları gibi ifade öğelerinin bir toplamıdır. Diğer birçok dilin aksine, tüm dil yapıları ifade değildir. Ayrıca `while` gibi kullanılamayan *ifadeler* de vardır. Atamalar da değer döndürmeyen ifadelerdir (statement).

uzatma modülü
Çekirdekle ve kullanıcı koduyla etkileşim kurmak için Python'un C API'sini kullanan, C veya C++ ile yazılmış bir modül.

f-string
Ön eki `'f'` veya `'F'` olan dize değişmezleri genellikle "f-strings" olarak adlandırılır; bu, formatted string literals 'ın kısaltmasıdır. Ayrıca bkz. **PEP 498**.

dosya nesnesi
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Aslında üç dosya nesnesi kategorisi vardır: ham *binary files*, arabelleğe alınmış *binary files* ve *text files*. Arayüzleri `io` modülünde tanımlanmıştır. Bir dosya nesnesi yaratmanın kurallı yolu `open()` işlevini kullanmaktır.

dosya benzeri nesne
dosya nesnesi ile eşanlamlıdır.

dosya sistemi kodlaması ve hata işleyicisi
Python tarafından işletim sistemindeki baytların kodunu çözmek ve Unicode'u işletim sistemine kodlamak için kullanılan kodlama ve hata işleyici.

Dosya sistemi kodlaması, 128'in altındaki tüm baytların kodunu başarıyla çözmeyi garanti etmelidir. Dosya sistemi kodlaması bu garantiyi sağlayamazsa, API işlevleri `UnicodeError` değerini yükseltebilir.

`sys.getfilesystemencoding()` ve `sys.getfilesystemencodeerrors()` işlevleri, dosya sistemi kodlamasını ve hata işleyicisini almak için kullanılabilir.

filesystem encoding and error handler Python başlangıcında `PyConfig_Read()` işleviyle yapılandırılır: bkz. `filesystem_encoding` ve `filesystem_errors` üyeleri `PyConfig`.

Ayrıca bkz. *locale encoding*.

bulucu
İçe aktarılmakta olan bir modül için *loader* 'ı bulmaya çalışan bir nesne.

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

kat bölümü
En yakın tam sayıya yuvarlayan matematiksel bölme. Kat bölme operatörü `//` şeklindedir. Örneğin, `11 // 4` ifadesi, gerçek yüzer bölme tarafından döndürülen `2.75` değerinin aksine `2` olarak değerlendirilir. `(-11) // 4` 'ün `-3` olduğuna dikkat edin, çünkü bu `-2.75` yuvarlatılmış *aşağı*. Bakınız **PEP 238**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

fonksiyon
Bir arayana bir değer döndüren bir dizi ifade. Ayrıca, gövdenin yürütülmesinde kullanılabilen sıfır veya daha fazla *argüman* iletilebilir. Ayrıca *parameter*, *method* ve function bölümüne bakın.

fonksiyon açıklaması
Bir işlev parametresinin veya dönüş değerinin *ek açıklaması*.

İşlev ek açıklamaları genellikle *type hints* için kullanılır: örneğin, bu fonksiyonun iki `int` argüman alması ve ayrıca bir `int` dönüş değerine sahip olması beklenir

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

İşlev açıklama sözdizimi function bölümünde açıklanmaktadır.

Bu işlevi açıklayan *variable annotation* ve **PEP 484** 'e bakın. Ek açıklamalarla çalışmaya ilişkin en iyi uygulamalar için ayrıca annotations-howto konusuna bakın.

__future__
Bir future ifadesi, `from __future__ import <feature>`, derleyiciyi, Python'un gelecekteki bir sürümünde standart hale gelecek olan sözdizimini veya semantiği kullanarak mevcut modülü derlemeye yönlendirir. `__future__` modülü, *feature*'ın olası değerlerini belgeler. Bu modülü içe aktararak ve değişkenlerini değerlendirerek, dile ilk kez yeni bir özelliğin ne zaman eklendiğini ve ne zaman varsayılan olacağını (ya da yaptığını) görebilirsiniz:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

çöp toplama
Artık kullanılmadığında belleği boşaltma işlemi. Python, referans sayımı ve referans döngülerini algılayıp kırabilen bir döngüsel çöp toplayıcı aracılığıyla çöp toplama gerçekleştirir. Çöp toplayıcı `gc` modülü kullanılarak kontrol edilebilir.

jeneratör
Bir *generator iterator* döndüren bir işlev. Bir for döngüsünde kullanılabilen bir dizi değer üretmek için `yield` ifadeleri içermesi veya `next()` işleviyle birer birer alınabilmesi dışında normal bir işleve benziyor.

Genellikle bir üretici işlevine atıfta bulunur, ancak bazı bağlamlarda bir *jeneratör yineleyicisine* atıfta bulunabilir. Amaçlanan anlamın net olmadığı durumlarda, tam terimlerin kullanılması belirsizliği önler.

jeneratör yineleyici
Bir *generator* işlevi tarafından oluşturulan bir nesne.

Her `yield`, konum yürütme durumunu hatırlayarak (yerel değişkenler ve bekleyen try ifadeleri dahil) işlemeyi geçici olarak askıya alır. *jeneratör yineleyici* devam ettiğinde, kaldığı yerden devam eder (her çağrıda yeniden başlayan işlevlerin aksine).

jeneratör ifadesi
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

genel işlev
Farklı türler için aynı işlemi uygulayan birden çok işlevden oluşan bir işlev. Bir çağrı sırasında hangi uygulamanın kullanılması gerektiği, gönderme algoritması tarafından belirlenir.

Ayrıca *single dispatch* sözlük girdisine, `functools.singledispatch()` dekoratörüne ve **PEP 443** 'e bakın.

genel tip
Parametrelendirilebilen bir *type*; tipik olarak bir konteyner sınıfı, örneğin `list` veya `dict`. *type hint* ve *annotation* için kullanılır.

Daha fazla ayrıntı için generic allias types, **PEP 483**, **PEP 484**, **PEP 585** ve `typing` modülüne bakın.

GIL
Bakınız *global interpreter lock*.

genel tercüman kilidi
CPython yorumlayıcısı tarafından aynı anda yalnızca bir iş parçacığının Python *bytecode* 'u yürütmesini sağlamak için kullanılan mekanizma. Bu, nesne modelini (`dict` gibi kritik yerleşik türler dahil) eşzamanlı erişime karşı örtük olarak güvenli hale getirerek CPython uygulamasını basitleştirir. Tüm yorumlayıcıyı kilitlemek, çok işlemcili makinelerin sağladığı paralelliğin çoğu pahasına, yorumlayıcının çok iş parçacıklı olmasını kolaylaştırır.

Bununla birlikte, standart veya üçüncü taraf bazı genişletme modülleri, sıkıştırma veya karma gibi hesaplama açısından yoğun görevler yaparken GIL'yi serbest bırakacak şekilde tasarlanmıştır. Ayrıca, GIL, G/Ç yaparken her zaman serbest bırakılır.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil =0` or after setting the `PYTHON_GIL =0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

karma tabanlı pyc
Geçerliliğini belirlemek için ilgili kaynak dosyanın son değiştirilme zamanı yerine karma değerini kullanan bir bayt kodu önbellek dosyası. Bakınız pyc-invalidation.

yıkanabilir
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability, bir nesneyi bir sözlük anahtarı ve bir set üyesi olarak kullanılabilir hale getirir, çünkü bu veri yapıları hash değerini dahili olarak kullanır.

Python'un değişmez yerleşik nesnelerinin çoğu, yıkanabilir; değiştirilebilir kaplar (listeler veya sözlükler gibi) değildir; değişmez kaplar (tüpler ve donmuş kümeler gibi) yalnızca öğelerinin yıkanabilir olması durumunda yıkanabilirdir. Kullanıcı tanımlı sınıfların örnekleri olan nesneler varsayılan olarak hash edilebilirdir. Hepsi eşit olmayanı karşılaştırır (kendileriyle hariç) ve hash değerleri `id()` 'lerinden türetilir.

BOŞTA
Python için Entegre Geliştirme Ortamı. idle, Python'un standart dağıtımıyla birlikte gelen temel bir düzenleyici ve yorumlayıcı ortamıdır.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

değişmez
Sabit değeri olan bir nesne. Değişmez nesneler arasında sayılar, dizeler ve demetler bulunur. Böyle bir nesne değiştirilemez. Farklı bir değerin saklanması gerekiyorsa yeni bir nesne oluşturulmalıdır. Örneğin bir sözlükte anahtar olarak, sabit bir karma değerinin gerekli olduğu yerlerde önemli bir rol oynarlar.

içe aktarım yolu
İçe aktarılacak modüller için *path based finder* tarafından aranan konumların (veya *path entries*) listesi. İçe aktarma sırasında, bu konum listesi genellikle `sys.path` adresinden gelir, ancak alt paketler için üst paketin `__path__` özelliğinden de gelebilir.

içe aktarma
Bir modüldeki Python kodunun başka bir modüldeki Python koduna sunulması süreci.

içe aktarıcı
Bir modülü hem bulan hem de yükleyen bir nesne; hem bir *finder* hem de *loader* nesnesi.

etkileşimli
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by

selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see *Etkileşimli Mod*.

yorumlanmış
Python, derlenmiş bir dilin aksine yorumlanmış bir dildir, ancak bayt kodu derleyicisinin varlığı nedeniyle ayrım bulanık olabilir. Bu, kaynak dosyaların daha sonra çalıştırılacak bir yürütülebilir dosya oluşturmadan doğrudan çalıştırılabileceği anlamına gelir. Yorumlanan diller genellikle derlenmiş dillerden daha kısa bir geliştirme/hata ayıklama döngüsüne sahiptir, ancak programları genellikle daha yavaş çalışır. Ayrıca bkz. *interactive*.

tercüman kapatma
Kapatılması istendiğinde, Python yorumlayıcısı, modüller ve çeşitli kritik iç yapılar gibi tahsis edilen tüm kaynakları kademeli olarak serbest bıraktığı özel bir aşamaya girer. Ayrıca *garbage collector* için birkaç çağrı yapar. Bu, kullanıcı tanımlı yıkıcılarda veya zayıf referans geri aramalarında kodun yürütülmesini tetikleyebilir. Kapatma aşamasında yürütülen kod, dayandığı kaynaklar artık çalışmayabileceğinden çeşitli istisnalarla karşılaşabilir (yaygın örnekler kütüphane modülleri veya uyarı makineleridir).

Yorumlayıcının kapatılmasının ana nedeni, `__main__` modülünün veya çalıştırılan betiğin yürütmeyi bitirmiş olmasıdır.

yinelenebilir
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, …). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

yineleyici
An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Daha fazla bilgi typeiter içinde bulunabilir.

CPython uygulama ayrıntısı: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

anahtar işlev
Anahtar işlevi veya harmanlama işlevi, sıralama veya sıralama için kullanılan bir değeri döndüren bir çağrılabilir. Örneğin, `locale.strxfrm()`, yerel ayara özgü sıralama kurallarının farkında olan bir sıralama anahtarı üretmek için kullanılır.

Python'daki bir dizi araç, öğelerin nasıl sıralandığını veya gruplandırıldığını kontrol etmek için temel işlevleri kabul eder. Bunlar `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` ve `itertools.groupby()`.

Bir tuş fonksiyonu oluşturmanın birkaç yolu vardır. Örneğin. `str.lower()` yöntemi, büyük/küçük harfe duyarlı olmayan sıralamalar için bir anahtar fonksiyonu işlevi görebilir. Alternatif olarak, `lambda r: (r[0], r[2])` gibi bir `lambda` ifadesinden bir anahtar işlevi oluşturulabilir. Ayrıca, `attrgetter()`, `itemgetter()` ve `methodcaller()` fonksiyonları üç anahtar fonksiyon kurucularıdır. Anahtar işlevlerin nasıl oluşturulacağı ve kullanılacağına ilişkin örnekler için Sorting HOW TO bölümüne bakın.

anahtar kelime argümanı
Bakınız *argument*.

lambda
İşlev çağrıldığında değerlendirilen tek bir *expression* 'dan oluşan anonim bir satır içi işlev. Bir lambda işlevi oluşturmak için sözdizimi `lambda [parametreler]: ifade` şeklindedir

LBYL
Zıplamadan önce Bak. Bu kodlama stili, arama veya arama yapmadan önce ön koşulları açıkça test eder. Bu stil, *EAFP* yaklaşımıyla çelişir ve birçok `if` ifadesinin varlığı ile karakterize edilir.

Çok iş parçacıklı bir ortamda, LBYL yaklaşımı "bakan" ve "sıçrayan" arasında bir yarış koşulu getirme riskini taşıyabilir. Örneğin, `if key in mapping: return mapping[key]` kodu, testten sonra, ancak aramadan önce başka bir iş parçacığı *eşlemeden key* kaldırırsa başarısız olabilir. Bu sorun, kilitlerle veya EAFP yaklaşımı kullanılarak çözülebilir.

liste
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

liste anlama
Bir dizideki öğelerin tümünü veya bir kısmını işlemenin ve sonuçları içeren bir liste döndürmenin kompakt bir yolu. `sonuç = ['{:#04x}'.format(x) for range(256) if x % 2 == 0]`, dizinde çift onaltılık sayılar (0x..) içeren bir diziler listesi oluşturur. 0 ile 255 arasındadır. `if` yan tümcesi isteğe bağlıdır. Atlanırsa, "aralık(256)" içindeki tüm öğeler işlenir.

yükleyici
An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

yerel kodlama
Unix'te, LC_CTYPE yerel ayarının kodlamasıdır. `locale.setlocale(locale.LC_CTYPE, new_locale)` ile ayarlanabilir.

Windows'ta bu, ANSI kod sayfasıdır (ör. `"cp1252"`).

Android ve VxWorks'te Python, yerel kodlama olarak `"utf-8"` kullanır.

`locale.getencoding()` can be used to get the locale encoding.

Ayrıca *filesystem encoding and error handler* 'ne bakın.

sihirli yöntem
special method için gayri resmi bir eşanlamlı.

haritalama
Keyfi anahtar aramalarını destekleyen ve `Mapping` veya `MutableMapping` collections-abstract-base-classes içinde belirtilen yöntemleri uygulayan bir kapsayıcı nesnesi. Örnekler arasında `dict`, `collections.defaultdict`, `collections.OrderedDict` ve `collections.Counter` sayılabilir.

meta yol bulucu
Bir *finder*, `sys.meta_path` aramasıyla döndürülür. Meta yol bulucular, *yol girişi bulucuları* ile ilişkilidir, ancak onlardan farklıdır.

Meta yol bulucuların uyguladığı yöntemler için `importlib.abc.MetaPathFinder` bölümüne bakın.

metasınıf
Bir sınıfın sınıfı. Sınıf tanımları, bir sınıf adı, bir sınıf sözlüğü ve temel sınıfların bir listesini oluşturur. Metasınıf, bu üç argümanı almaktan ve sınıfı oluşturmaktan sorumludur. Çoğu nesne yönelimli programlama dili, varsayılan bir uygulama sağlar. Python'u özel yapan şey, özel metasınıflar oluşturmanın mümkün olmasıdır. Çoğu kullanıcı bu araca hiçbir zaman ihtiyaç duymaz, ancak ihtiyaç duyulduğunda, metasınıflar güçlü ve zarif

çözümler sağlayabilir. Nitelik erişimini günlüğe kaydetmek, iş parçacığı güvenliği eklemek, nesne oluşturmayı izlemek, tekilleri uygulamak ve diğer birçok görev için kullanılmışlardır.

Daha fazla bilgi metaclasses içinde bulunabilir.

metot
Bir sınıf gövdesi içinde tanımlanan bir işlev. Bu sınıfın bir örneğinin özniteliği olarak çağrılırsa, yöntem örnek nesnesini ilk *argument* (genellikle `self` olarak adlandırılır) olarak alır. Bkz. *function* ve *nested scope*.

metot kalite sıralaması
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

modül
Python kodunun kuruluş birimi olarak hizmet eden bir nesne. Modüller, rastgele Python nesneleri içeren bir ad alanına sahiptir. Modüller, *importing* işlemiyle Python'a yüklenir.

Ayrıca bakınız *package*.

modül özelliği
Bir modülü yüklemek için kullanılan içe aktarmayla ilgili bilgileri içeren bir ad alanı. Bir `importlib.machinery.ModuleSpec` örneği.

See also module-specs.

MRO
Bakınız *metot çözüm sırası*.

değiştirilebilir
Değiştirilebilir (mutable) nesneler değerlerini değiştirebilir ancak `idlerini` koruyabilirler. Ayrıca bkz. *immutable*.

adlandırılmış demet
"named tuple" terimi, demetten miras alan ve dizinlenebilir öğelerine de adlandırılmış nitelikler kullanılarak erişilebilen herhangi bir tür veya sınıf için geçerlidir. Tür veya sınıfın başka özellikleri de olabilir.

Çeşitli yerleşik türler, `time.localtime()` ve `os.stat()` tarafından döndürülen değerler de dahil olmak üzere, tanımlama grupları olarak adlandırılır. Başka bir örnek `sys.float_info`:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

ad alanı
Değişkenin saklandığı yer. Ad alanları sözlükler olarak uygulanır. Nesnelerde (yöntemlerde) yerel, genel ve yerleşik ad alanlarının yanı sıra iç içe ad alanları vardır. Ad alanları, adlandırma çakışmalarını önleyerek modülerliği destekler. Örneğin, `builtins.open` ve `os.open()` işlevleri ad alanlarıyla ayırt edilir. Ad alanları, hangi modülün bir işlevi uyguladığını açıkça belirterek okunabilirliğe ve sürdürülebilirliğe de yardımcı olur. Örneğin, `random.seed()` veya `itertools.islice()` yazmak, bu işlevlerin sırasıyla `random` ve `itertools` modülleri tarafından uygulandığını açıkça gösterir.

ad alanı paketi
A **PEP 420** *package*, yalnızca alt paketler için bir kap olarak hizmet eder. Ad alanı paketlerinin hiçbir fiziksel temsili olmayabilir ve `__init__.py` dosyası olmadığından özellikle *regular package* gibi değildirler.

Ayrıca bkz. *module*.

iç içe kapsam
Kapsamlı bir tanımdaki bir değişkene atıfta bulunma yeteneği. Örneğin, başka bir fonksiyonun içinde tanımlanan bir fonksiyon, dış fonksiyondaki değişkenlere atıfta bulunabilir. İç içe kapsamların varsayılan olarak yalnızca başvuru için çalıştığını ve atama için çalışmadığını unutmayın. Yerel değişkenler en içteki kapsamda hem okur hem de yazar. Benzer şekilde, global değişkenler global ad alanını okur ve yazar. `nonlocal`, dış kapsamlara yazmaya izin verir.

yeni stil sınıf
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

obje
Durum (öznitelikler veya değer) ve tanımlanmış davranış (yöntemler) içeren herhangi bir veri. Ayrıca herhangi bir *yeni tarz sınıfın* nihai temel sınıfı.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

paket
Alt modüller veya yinelemeli olarak alt paketler içerebilen bir Python *module*. Teknik olarak bir paket, `__path__` özniteliğine sahip bir Python modülüdür.

Ayrıca bkz. *regular package* ve *namespace package*.

parametre
Bir *function* (veya yöntem) tanımında, işlevin kabul edebileceği bir *argument* (veya bazı durumlarda, argümanlar) belirten adlandırılmış bir varlık. Beş çeşit parametre vardır:

- *positional-or-keyword*: *pozisyonel* veya bir *keyword argümanı* olarak iletilebilen bir argüman belirtir. Bu, varsayılan parametre türüdür, örneğin aşağıdakilerde *foo* ve *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*: yalnızca konuma göre sağlanabilen bir argüman belirtir. Yalnızca konumsal parametreler, onlardan sonra fonksiyon tanımının parametre listesine bir `/` karakteri eklenerek tanımlanabilir, örneğin aşağıdakilerde *posonly1* ve *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: sadece anahtar kelime ile sağlanabilen bir argüman belirtir. Yalnızca anahtar kelime (keyword-only) parametreleri, onlardan önceki fonksiyon tanımının parametre listesine tek bir değişken konumlu parametre veya çıplak `*` dahil edilerek tanımlanabilir, örneğin aşağıdakilerde *kw_only1* ve *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: keyfi bir pozisyonel argüman dizisinin sağlanabileceğini belirtir (diğer parametreler tarafından zaten kabul edilmiş herhangi bir konumsal argümana ek olarak). Böyle bir parametre, parametre adının başına `*` eklenerek tanımlanabilir, örneğin aşağıdakilerde *args*:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: keyfi olarak birçok anahtar kelime argümanının sağlanabileceğini belirtir (diğer parametreler tarafından zaten kabul edilen herhangi bir anahtar kelime argümanına ek olarak). Böyle bir parametre, parametre adının başına `**`, örneğin yukarıdaki örnekte *kwargs* eklenerek tanımlanabilir.

Parametreler, hem isteğe bağlı hem de gerekli argümanleri ve ayrıca bazı isteğe bağlı bağımsız değişkenler için varsayılan değerleri belirtebilir.

Ayrıca bkz. *argüman*, argümanlar ve parametreler arasındaki fark, `inspect.Parameter`, function ve **PEP 362**.

yol girişi
path based finder içe aktarma modüllerini bulmak için başvurduğu *import path* üzerindeki tek bir konum.

yol girişi bulucu
Bir *finder* `sys.path_hooks` (yani bir *yol giriş kancası*) üzerinde bir çağrılabilir tarafından döndürülür ve *path entry* verilen modüllerin nasıl bulunacağını bilir.

Yol girişi bulucularının uyguladığı yöntemler için `importlib.abc.PathEntryFinder` bölümüne bakın.

yol giriş kancası
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

yol tabanlı bulucu
Modüller için bir *import path* arayan varsayılan *meta yol bulucular*dan biri.

yol benzeri nesne
Bir dosya sistemi yolunu temsil eden bir nesne. Yol benzeri bir nesne, bir yolu temsil eden bir `str` veya `bytes` nesnesi veya `os.PathLike` protokolünü uygulayan bir nesnedir. `os.PathLike` protokolünü destekleyen bir nesne, `os.fspath()` işlevi çağrılarak bir `str` veya `bytes` dosya sistemi yoluna dönüştürülebilir; `os.fsdecode()` ve `os.fsencode()`, bunun yerine sırasıyla `str` veya `bytes` sonucunu garanti etmek için kullanılabilir. **PEP 519** tarafından tanıtıldı.

PEP
Python Geliştirme Önerisi. PEP, Python topluluğuna bilgi sağlayan veya Python veya süreçleri ya da ortamı için yeni bir özelliği açıklayan bir tasarım belgesidir. PEP'ler, önerilen özellikler için özlü bir teknik şartname ve bir gerekçe sağlamalıdır.

PEP'lerin, önemli yeni özellikler önermek, bir sorun hakkında topluluk girdisi toplamak ve Python'a giren tasarım kararlarını belgelemek için birincil mekanizmalar olması amaçlanmıştır. PEP yazarı, topluluk içinde fikir birliği oluşturmaktan ve muhalif görüşleri belgelemekten sorumludur.

Bakınız **PEP 1**.

kısım
PEP 420 içinde tanımlandığı gibi, bir ad alanı paketine katkıda bulunan tek bir dizindeki (muhtemelen bir zip dosyasında depolanan) bir dizi dosya.

konumsal argüman
Bakınız *argument*.

geçici API
Geçici bir API, standart kitaplığın geriye dönük uyumluluk garantilerinden kasıtlı olarak hariç tutulan bir API'dir. Bu tür arayüzlerde büyük değişiklikler beklenmese de, geçici olarak işaretlendikleri sürece, çekirdek geliştiriciler tarafından gerekli görüldüğü takdirde geriye dönük uyumsuz değişiklikler (arayüzün kaldırılmasına kadar ve buna kadar) meydana gelebilir. Bu tür değişiklikler karşılıksız yapılmayacaktır - bunlar yalnızca API'nin eklenmesinden önce gözden kaçan ciddi temel kusurlar ortaya çıkarsa gerçekleşecektir.

Geçici API'ler için bile, geriye dönük uyumsuz değişiklikler "son çare çözümü" olarak görülür - tanımlanan herhangi bir soruna geriye dönük uyumlu bir çözüm bulmak için her türlü girişimde bulunulacaktır.

Bu süreç, standart kitaplığın, uzun süreler boyunca sorunlu tasarım hatalarına kilitlenmeden zaman içinde gelişmeye devam etmesini sağlar. Daha fazla ayrıntı için bkz. **PEP 411**.

geçici paket
Bakınız *provisional API*.

Python 3000
Python 3.x sürüm satırının takma adı (uzun zaman önce sürüm 3'ün piyasaya sürülmesi uzak bir gelecekte olduğu zaman ortaya çıktı.) Bu aynı zamanda "Py3k" olarak da kısaltılır.

Pythonic
Diğer dillerde ortak kavramları kullanarak kod uygulamak yerine Python dilinin en yaygın deyimlerini yakından takip eden bir fikir veya kod parçası. Örneğin, Python'da yaygın bir deyim, bir `for` ifadesi kullanarak

yinelenebilir bir öğenin tüm öğeleri üzerinde döngü oluşturmaktır. Diğer birçok dilde bu tür bir yapı yoktur, bu nedenle Python'a aşina olmayan kişiler bazen bunun yerine sayısal bir sayaç kullanır:

```
for i in range(len(food)):
    print(food[i])
```

Temizleyicinin aksine, Pythonic yöntemi:

```
for piece in food:
    print(piece)
```

nitelikli isim
: **PEP 3155** içinde tanımlandığı gibi, bir modülün genel kapsamından o modülde tanımlanan bir sınıfa, işleve veya yönteme giden "yolu" gösteren noktalı ad. Üst düzey işlevler ve sınıflar için nitelikli ad, nesnenin adıyla aynıdır:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Modüllere atıfta bulunmak için kullanıldığında, *tam nitelenmiş ad*, herhangi bir üst paket de dahil olmak üzere, modüle giden tüm noktalı yol anlamına gelir, örn. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

referans sayısı
: The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

sürekli paketleme
: `__init__.py` dosyası içeren bir dizin gibi geleneksel bir *package*.

Ayrıca bkz. *ad alanı paketi*.

REPL
: An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
: Örnek öznitelikleri için önceden yer bildirerek ve örnek sözlüklerini ortadan kaldırarak bellekten tasarruf sağlayan bir sınıf içindeki bildirim. Popüler olmasına rağmen, tekniğin doğru olması biraz zor ve en iyi, bellek açısından kritik bir uygulamada çok sayıda örneğin bulunduğu nadir durumlar için ayrılmıştır.

dizi
: An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

anlamak
Öğelerin tümünü veya bir kısmını yinelenebilir bir şekilde işlemenin ve sonuçlarla birlikte bir küme döndürmenin kompakt bir yolu. `results = {c for c in 'abracadabra' if c not in 'abc'}`, `{'r', 'd'}` dizelerini oluşturur. Bakınız comprehensions.

tek sevk
Uygulamanın tek bir argüman türüne göre seçildiği bir *generic function* gönderimi biçimi.

parçalamak
Genellikle bir *sequence* 'nin bir bölümünü içeren bir nesne. Bir dilim, örneğin `variable_name[1:3:5]` 'de olduğu gibi, birkaç tane verildiğinde, sayılar arasında iki nokta üst üste koyarak, `[]` alt simge gösterimi kullanılarak oluşturulur. Köşeli ayraç (alt simge) gösterimi, dahili olarak `slice` nesnelerini kullanır.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

özel metod
Toplama gibi bir tür üzerinde belirli bir işlemi yürütmek için Python tarafından örtük olarak çağrılan bir yöntem. Bu tür yöntemlerin çift alt çizgi ile başlayan ve biten adları vardır. Özel yöntemler specialnames içinde belgelenmiştir.

ifade (değer döndürmez)
Bir ifade, bir paketin parçasıdır (kod "bloğu"). Bir ifade, bir *expression* veya `if`, `while` veya `for` gibi bir anahtar kelimeye sahip birkaç yapıdan biridir.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

güçlü referans
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

`Py_NewRef()` fonksiyonu, bir nesneye güçlü bir başvuru oluşturmak için kullanılabilir. Genellikle `Py_DECREF()` fonksiyonu, bir referansın sızmasını önlemek için güçlü referans kapsamından çıkmadan önce güçlü referansta çağrılmalıdır.

Ayrıca bkz. *ödünç alınan referans*.

yazı çözümleme
Python'da bir dize, bir Unicode kod noktaları dizisidir (`U+0000–U+10FFFF` aralığında). Bir dizeyi depolamak veya aktarmak için, bir bayt dizisi olarak seri hale getirilmesi gerekir.

Bir dizeyi bir bayt dizisi halinde seri hale getirmek "kodlama (encoding)" olarak bilinir ve dizeyi bayt dizisinden yeniden oluşturmak "kod çözme (decoding)" olarak bilinir.

Toplu olarak "metin kodlamaları" olarak adlandırılan çeşitli farklı metin serileştirme kodekleri vardır.

yazı dosyası
A *file object* `str` nesnelerini okuyabilir ve yazabilir. Çoğu zaman, bir metin dosyası aslında bir bayt yönelimli veri akışına erişir ve otomatik olarak *text encoding* işler. Metin dosyalarına örnek olarak metin modunda açılan dosyalar (`'r'` veya `'w'`), `sys.stdin`, `sys.stdout` ve `io.StringIO` örnekleri verilebilir.

Ayrıca *ikili dosyaları* okuyabilen ve yazabilen bir dosya nesnesi için *bayt benzeri nesnelere* bakın.

üç tırnaklı dize
Üç tırnak işareti (") veya kesme işareti (') ile sınırlanan bir dize. Tek tırnaklı dizelerde bulunmayan herhangi bir işlevsellik sağlamasalar da, birkaç nedenden dolayı faydalıdırlar. bir dizeye çıkışsız tek ve çift tırnak eklemeniz gerekir ve bunlar, devam karakterini kullanmadan birden çok satıra yayılabilir, bu da onları özellikle belge dizileri yazarken kullanışlı hale getirir.

tip
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

tip takma adı
Bir tanımlayıcıya tür atanarak oluşturulan, bir tür için eş anlamlı.

Tür takma adları, *tür ipuçlarını* basitleştirmek için kullanışlıdır. Örneğin:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

bu şekilde daha okunaklı hale getirilebilir:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Bu işlevi açıklayan `typing` ve **PEP 484** bölümlerine bakın.

tür ipucu
Bir değişken, bir sınıf niteliği veya bir işlev parametresi veya dönüş değeri için beklenen türü belirten bir *ek açıklama*.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Genel değişkenlerin, sınıf özniteliklerinin ve işlevlerin tür ipuçlarına, yerel değişkenlere değil, `typing.get_type_hints()` kullanılarak erişilebilir.

Bu işlevi açıklayan `typing` ve **PEP 484** bölümlerine bakın.

evrensel yeni satırlar
Aşağıdakilerin tümünün bir satırın bitişi olarak kabul edildiği metin akışlarını yorumlamanın bir yolu: Unix satır sonu kuralı `'\n'`, Windows kuralı `'\r\n'`, ve eski Macintosh kuralı `'\r'`. Ek bir kullanım için **PEP 278** ve **PEP 3116** ve ayrıca `bytes.splitlines()` bakın.

değişken açıklama
Bir değişkenin veya bir sınıf özniteliğinin *ek açıklaması*.

Bir değişkene veya sınıf niteliğine açıklama eklerken atama isteğe bağlıdır:

```
class C:
    field: 'annotation'
```

Değişken açıklamaları genellikle *tür ipuçları* için kullanılır: örneğin, bu değişkenin `int` değerlerini alması beklenir:

```
count: int = 0
```

Değişken açıklama sözdizimi annassign bölümünde açıklanmıştır.

Bu işlevi açıklayan; *function annotation*, **PEP 484** ve **PEP 526** bölümlerine bakın. Ek açıklamalarla çalışmaya ilişkin en iyi uygulamalar için ayrıca bkz. annotations-howto.

sanal ortam
: Python kullanıcılarının ve uygulamalarının, aynı sistem üzerinde çalışan diğer Python uygulamalarının davranışına müdahale etmeden Python dağıtım paketlerini kurmasına ve yükseltmesine olanak tanıyan, işbirliği içinde yalıtılmış bir çalışma zamanı ortamı.

  Ayrıca bakınız `venv`.

sanal makine
: Tamamen yazılımla tanımlanmış bir bilgisayar. Python'un sanal makinesi, bayt kodu derleyicisi tarafından yayınlanan *bytecode* 'u çalıştırır.

Python'un Zen'i
: Dili anlamaya ve kullanmaya yardımcı olan Python tasarım ilkeleri ve felsefelerinin listesi. Liste, etkileşimli komut isteminde "`import this`" yazarak bulunabilir.

EK B

Bu dokümanlar hakkında

Bu dokümanlar, Python dokümanları için özel olarak yazılmış bir doküman işlemcisi olan Sphinx tarafından reStructuredText kaynaklarından oluşturulur.

Dokümantasyonun ve araç zincirinin geliştirilmesi, tıpkı Python'un kendisi gibi tamamen gönüllü bir çabadır. Katkıda bulunmak istiyorsanız, nasıl yapacağınıza ilişkin bilgi için lütfen reporting-bugs sayfasına göz atın. Yeni gönüllülere her zaman açığız!

Destekleri için teşekkürler:

- Fred L. Drake, Jr., orijinal Python dokümantasyon araç setinin yaratıcısı ve içeriğin çoğunun yazarı;
- Docutils projesi, reStructuredText ve Docutils paketini oluşturdukları için;
- Fredrik Lundh, Sphinx'in pek çok iyi fikir edindiği Alternatif Python Referansı projesi için.

B.1 Python Dokümantasyonuna Katkıda Bulunanlar

Birçok kişi Python diline, Python standart kütüphanesine ve Python dokümantasyonuna katkıda bulunmuştur. Katkıda bulunanların kısmi bir listesi için Python kaynak dağıtımında Misc/ACKS dosyasına bakın.

Python topluluğunun girdileri ve katkıları sayesinde böyle harika bir dokümantasyona sahibiz – Teşekkürler!

EK C

Tarihçe ve Lisans

C.1 Yazılımın tarihçesi

Python, 1990'ların başında Guido van Rossum tarafından Hollanda'da Stichting Mathematisch Centrum'da (CWI, bkz. https://www.cwi.nl/) ABC adlı bir dilin devamı olarak oluşturuldu. Guido, diğerlerinin oldukça katkısı olmasına rağmen, Python'un ana yazarı olmaya devam ediyor.

1995'te Guido, yazılımın çeşitli sürümlerini yayınladığı Virginia, Reston'daki Ulusal Araştırma Girişimleri Kurumu'nda (CNRI, bkz. https://www.cnri.reston.va.us/) Python üzerindeki çalışmalarına devam etti.

Mayıs 2000'de, Guido ve Python çekirdek geliştirme ekibi, BeOpen PythonLabs ekibini oluşturmak için BeOpen.com'a taşındı. Aynı yılın Ekim ayında PythonLabs ekibi Digital Creations'a (şimdi Zope Corporation; bkz. https://www.zope.org/) taşındı. 2001 yılında, Python Yazılım Vakfı (PSF, bkz. https://www.python.org/psf/) kuruldu, özellikle Python ile ilgili Fikri Mülkiyete sahip olmak için oluşturulmuş kar amacı gütmeyen bir organizasyon. Zope Corporation, PSF'nin sponsor üyesidir.

Tüm Python sürümleri Açık Kaynaklıdır (Açık Kaynak Tanımı için bkz. https://opensource.org/). Tarihsel olarak, tümü olmasa da çoğu Python sürümleri de GPL uyumluydu; aşağıdaki tablo çeşitli yayınları özetlemektedir.

Yayın	Şundan türedi:	Yıl	Sahibi	GPL uyumlu mu?
0.9.0'dan 1.2'ye	n/a	1991-1995	CWI	evet
1.3 'dan 1.5.2'ye	1.2	1995-1999	CNRI	evet
1.6	1.5.2	2000	CNRI	hayır
2.0	1.6	2000	BeOpen.com	hayır
1.6.1	1.6	2001	CNRI	hayır
2.1	2.0+1.6.1	2001	PSF	hayır
2.0.1	2.0+1.6.1	2001	PSF	evet
2.1.1	2.1+2.0.1	2001	PSF	evet
2.1.2	2.1.1	2002	PSF	evet
2.1.3	2.1.2	2002	PSF	evet
2.2 ve üzeri	2.1.1	2001-Günümüz	PSF	evet

Not

GPL uyumlu olması, Python'u GPL kapsamında dağıttığımız anlamına gelmez. Tüm Python lisansları, GPL'den farklı olarak, değişikliklerinizi açık kaynak yapmadan değiştirilmiş bir sürümü dağıtmanıza izin verir. GPL

uyumlu lisanslar, Python'u GPL kapsamında yayınlanan diğer yazılımlarla birleştirmeyi mümkün kılar; diğerleri yapmaz.

Bu yayınları mümkün kılmak için Guido'nun yönetimi altında çalışan birçok gönüllüye teşekkürler.

C.2 Python'a erişmek veya başka bir şekilde kullanmak için şartlar ve koşullar

Python yazılımı ve belgeleri *PSF Lisans Anlaşması* kapsamında lisanslanmıştır.

Python 3.8.6'dan başlayarak, belgelerdeki örnekler, tarifler ve diğer kodlar, PSF Lisans Sözleşmesi ve *Zero-Clause BSD license* kapsamında çift lisanslıdır.

Python'a dahil edilen bazı yazılımlar farklı lisanslar altındadır. Lisanslar, bu lisansa giren kodla listelenir. Bu lisansların eksik listesi için bkz. *Tüzel Yazılımlar için Lisanslar ve Onaylar*.

C.2.1 PYTHON İÇİN PSF LİSANS ANLAŞMASI 3.13.0

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.13.0 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.13.0 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.13.0 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.13.0 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python
   3.13.0.

4. PSF is making Python 3.13.0 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON 3.13.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.
```

```
8. By copying, installing or otherwise using Python 3.13.0, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 PYTHON 2.0 İÇİN BEOPEN.COM LİSANS SÖZLEŞMESİ

BEOPEN PYTHON AÇIK KAYNAK LİSANS SÖZLEŞMESİ SÜRÜM 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 PYTHON 1.6.1 İÇİN CNRI LİSANS ANLAŞMASI

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 0.9.0 ARASI 1.2 PYTHON İÇİN CWI LİSANS SÖZLEŞMESİ

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 PYTHON 3.13.0 BELGELERİNDEKİ KOD İÇİN SIFIR MADDE BSD LİSANSI

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Tüzel Yazılımlar için Lisanslar ve Onaylar

Bu bölüm, Python dağıtımına dahil edilmiş üçüncü taraf yazılımlar için tamamlanmamış ancak büyüyen bir lisans ve onay listesidir.

C.3.1 Mersenne Twister'ı

`random` modülünün altyapısını oluşturan `_random` C uzantısı, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html adresinden indirilen kodu temel alır. Orijinal koddan kelimesi kelimesine yorumlar aşağıdadır:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Soketler

`socket` modülü, https://www.wide.ad.jp/ adresindeki WIDE Projesi'nden ayrı kaynak dosyalarında kodlanan `getaddrinfo()` ve `getnameinfo()` fonksiyonlarını kullanır.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asenkron soket hizmetleri

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Çerez yönetimi

`http.cookies` modülü aşağıdaki uyarıyı içerir:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Çalıştırma izleme

`trace` modülü aşağıdaki uyarıyı içerir:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode ve UUdecode fonksiyonları

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Uzaktan Yordam Çağrıları

`xmlrpc.client` modülü aşağıdaki uyarıyı içerir:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` modülü aşağıdaki uyarıyı içerir:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 kqueue seçin

`select` modülü, kqueue arayüzü için aşağıdaki uyarıyı içerir:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

`Python/pyhash.c` dosyası, Dan Bernstein'ın SipHash24 algoritmasının Marek Majkowski uygulamasını içerir. Burada aşağıdaki not yer alır:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod ve dtoa

C double'larının dizelere ve dizelerden dönüştürülmesi için dtoa ve strtod C fonksiyonlarını sağlayan `Python/dtoa.c` dosyası, şu anda https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 'den erişilebilen David M. Gay tarafından aynı adlı dosyadan türetilmiştir. 16 Mart 2009'da alınan orijinal dosya aşağıdaki telif hakkı ve lisans bildirimini içerir:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

`ctypes` modülünün altyapısını oluşturan `_ctypes` C uzantısı, `--with-system-libffi` olarak yapılandırılmadığı sürece libffi kaynaklarının dahil edildiği bir kopya kullanılarak oluşturulur:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

`zlib` uzantısı, sistemde bulunan zlib sürümü derleme için kullanılamayacak kadar eskiyse, zlib kaynaklarının dahil edildiği bir kopya kullanılarak oluşturulur:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` tarafından kullanılan hash tablosunun uygulanması cfuhash projesine dayanmaktadır:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test paketi

`test` paketindeki C14N 2.0 test paketi (`Lib/test/xmltestdata/c14n-20/`), https://www.w3.org/TR/xml-c14n2-testcases/ adresindeki W3C web sitesinden alınmıştır ve 3 maddeli BSD lisansı altında dağıtılmaktadır:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

(sonraki sayfaya devam)

(önceki sayfadan devam)

```
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

EK D

Telif Hakkı

Bütün lisans ve izin bilgileri için *Tarihçe ve Lisans* 'a göz atın.

Dizin

Alfabetik olmayan

... , **123**
(*hash*)
    yorum, 9
* (*asterisk*)
    in function calls, 32
**
    in function calls, 33
: (*colon*)
    function annotations, 34
->
    function annotations, 34
>>>, **123**
__all__, 55
__future__, **129**
__slots__, **136**

A

ad alanı, **133**
ad alanı paketi, **133**
adlandırılmış demet, **133**
anahtar işlev, **131**
anahtar kelime argümanı, **132**
anlamak, **137**
annotations
    function, 34
arama
    dizin, modül, 51
argüman, **123**
asenkron bağlam yöneticisi, **124**
asenkron jeneratör, **124**
asenkron jeneratör yineleyici, **124**
asenkron yineleyici, **124**

B

bağlam değişkeni, **126**
bağlam yöneticisi, **126**
bayt benzeri nesne, **125**
bayt kodu, **125**
BDFL, **124**
beklenebilir, **124**
belge dizisi, **127**
bitişik, **126**
BOŞTA, **130**
built-in function
    help, 93
    open, 63
bulucu, **128**

C

C-contiguous, 126
closure variable, **125**
coding
    style, 34
context, **126**
context management protocol, **126**
CPython, **126**
current context, **126**

Ç

çağırılabilir, **125**
çöp toplama, **129**

D

değişken açıklama, **138**
değişmez, **130**
değiştirilebilir, **133**
dekoratör, **127**
dipnot, **123**
dizi, **136**
dizin
    modül arama, 51
docstrings, 26, 33
documentation strings, 26, 33
dosya benzeri nesne, **128**
dosya nesnesi, **128**
dosya sistemi kodlaması ve hata işleyicisi, **128**

E

EAFP, **127**
eşyordam, **126**
eşyordam işlevi, **126**
eşzamansız yinelenebilir, **124**
etkileşimli, **130**
evrensel yeni satırlar, **138**

F

f-string, **128**
file
    object, 63
fonksiyon, **128**
fonksiyon açıklaması, **129**
for
    statement, 19
formatted string literal, 60
Fortran contiguous, 126
free threading, **128**
free variable, **128**
fstring, 60
f-string, 60
function
    annotations, 34

G

geçici API, **135**
geçici paket, **135**
genel işlev, **129**
genel tercüman kilidi, **130**
genel tip, **129**
geri çağırmak, **125**
GIL, **130**
güçlü referans, **137**

H

haritalama, **132**
help
    built-in function, 93

İ

iç içe kapsam, **134**
içe aktarıcı, **130**
içe aktarım yolu, **130**
içe aktarma, **130**
ifade (*değer döndürmez*), **137**
ifade (*değer döndürür*), **128**
ikili dosya, **124**
immortal, **130**
interpolated string literal, 60

J

jeneratör, **129**
jeneratör ifadesi, **129**
jeneratör yineleyici, **129**
json
    module, 65

K

karma tabanlı pyc, **130**
karmaşık sayı, **126**
kat bölümü, **128**
kısım, **135**
konumsal argüman, **135**

L

lambda, **132**
LBYL, **132**
liste, **132**
liste anlama, **132**

M

magic
    metot, 132
mangling
    name, 87
meta yol bulucu, **132**
metasınıf, **132**
method
    object, 83
metot, **133**
    magic, 132
    special, 137
metot kalite sıralaması, **133**
module
    json, 65
modül, **133**
    arama dizin, 51
    sys, 52
    yerleşikler, 53
modül özelliği, **133**
MRO, **133**

N

name
    mangling, 87
nitelik, **124**
nitelikli isim, **136**

O

obje, **134**
object
    file, 63
    method, 83
open
    built-in function, 63
optimized scope, **134**
ortam değişkeni
    PATH, 51, 122
    PYTHON_BASIC_REPL, 121
    PYTHON_GIL, 130
    PYTHONPATH, 51, 52
    PYTHONSTARTUP, 122

Ö

ödünç alınan referans, **125**
ördek yazma, **127**
özel metod, **137**

P

paket, **134**
parametre, **134**

parçalamak, **137**
PATH, 51, 122
PEP, **135**
Python 3000, **135**
Python Geliştirme Önerileri
    PEP 1, 135
    PEP 8, 34
    PEP 238, 128
    PEP 278, 138
    PEP 302, 132
    PEP 343, 126
    PEP 362, 124, 135
    PEP 411, 135
    PEP 420, 133, 135
    PEP 443, 129
    PEP 483, 129
    PEP 484, 34, 123, 129, 138
    PEP 492, 124, 126
    PEP 498, 128
    PEP 519, 135
    PEP 525, 124
    PEP 526, 123, 138
    PEP 585, 129
    PEP 636, 25
    PEP 683, 130
    PEP 703, 128, 130
    PEP 3107, 34
    PEP 3116, 138
    PEP 3147, 52
    PEP 3155, 136
PYTHON_BASIC_REPL, 121
PYTHON_GIL, 130
Pythonic, **135**
PYTHONPATH, 51, 52
PYTHONSTARTUP, 122
Python'un Zen'i, **139**

R

referans sayısı, **136**
REPL, **136**
RFC
    RFC 2822, 98

S

sanal makine, **139**
sanal ortam, **139**
sınıf, **125**
sınıf değişkeni, **125**
sihirli yöntem, **132**
sitecustomize, 122
soft deprecated, **137**
soyut temel sınıf, **123**
sözlük, **127**
sözlük anlama, **127**
sözlük görünümü, **127**
special
    metot, 137
statement
    for, 19
static type checker, **137**
string
    formatted literal, 60
    interpolated literal, 60
strings, documentation, 26, 33
style
    coding, 34
sürekli paketleme, **136**
sys
    modül, 52

T

tanımlayıcı, **127**
tek sevk, **137**
tercüman kapatma, **131**
tip, **138**
tip takma adı, **138**
tür ipucu, **138**

U

usercustomize, 122
uzatma modülü, **128**

Ü

üç tırnaklı dize, **138**

Y

yazı çözümleme, **137**
yazı dosyası, **137**
yeni stil sınıf, **134**
yerel kodlama, **132**
yerleşikler
    modül, 53
yıkanabilir, **130**
yinelenebilir, **131**
yineleyici, **131**
yol benzeri nesne, **135**
yol giriş kancası, **135**
yol girişi, **135**
yol girişi bulucu, **135**
yol tabanlı bulucu, **135**
yorumlanmış, **131**
yükleyici, **132**