8. Hatalar ve İstisnalar

Şimdiye kadar hata mesajları belirtilenden daha fazla olmadı, ancak örnekleri denediyseniz muhtemelen bazılarını gördünüz. (En az) ayırt edilebilir iki tür hata vardır: *sözdizimi hataları* ve *istisnalar* .

8.1. Sözdizimi Hataları

Ayrıştırma hataları olarak da bilinen sözdizimi hataları, hala Python öğrenirken aldığınız en yaygın şikayet türüdür:

```
>>>
```

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

Ayrıştırıcı, rahatsız edici satırı tekrarlar ve hatanın tespit edildiği satırdaki en erken noktayı gösteren küçük bir 'ok' görüntüler. Hata, oktan *önceki* jetondan kaynaklanır (veya en azından algılanır) : örnekte, önce `print()` bir iki nokta üst üste (`':'`) bulunmadığından hata işlevde algılanır . Dosya adı ve satır numarası yazdırılır, böylece girdinin bir komut dosyasından gelmesi durumunda nereye bakılacağını bilirsiniz.

8.2. İstisnalar

Bir ifade veya ifade sözdizimsel olarak doğru olsa bile, yürütme girişiminde bulunulduğunda hataya neden olabilir. Yürütme sırasında algılanan hatalara *istisna* denir ve koşulsuz olarak ölümcül değildir: yakında Python programlarında bunların nasıl ele alınacağını öğreneceksiniz. Bununla birlikte, çoğu özel durum programlar tarafından işlenmez ve burada gösterildiği gibi hata iletilerine neden olur:

```
>>>
```

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Hata mesajının son satırı ne olduğunu gösterir. İstisnalar farklı türde gelir ve tipi mesajın parçası olarak yazdırılır: örnekteki türleridir `ZeroDivisionError`, `NameError`ve `TypeError`. İstisna türü olarak yazdırılan dize, gerçekleşen yerleşik istisnanın adıdır. Bu, tüm yerleşik istisnalar için geçerlidir, ancak kullanıcı tanımlı istisnalar için doğru olması gerekmez (yararlı bir kural olmasına rağmen). Standart istisna adları yerleşik tanımlayıcılardır (ayrılmış anahtar kelimeler değil).

Çizginin geri kalanı, istisna türüne ve neye neden olduğuna bağlı olarak ayrıntı sağlar.

Hata iletisinin önceki bölümü, kural dışı durumun gerçekleştiği bağlamı yığın izleme biçiminde gösterir. Genel olarak kaynak satırlarını listeleyen bir yığın geri izleme içerir; ancak, standart girişten okunan satırları göstermez.

Yerleşik İstisnalar yerleşik istisnaları ve anlamlarını listeler.

8.3. İstisnaları Ele Alma

Seçilen istisnaları ele alan programlar yazmak mümkündür. Geçerli bir tamsayı girilene kadar kullanıcıdan giriş yapmasını isteyen, ancak kullanıcının programı (`Control-c`işletim sisteminin desteklediği ya da desteklediği her şeyi kullanarak) kesmesine izin veren aşağıdaki örneğe bakın ; kullanıcı tarafından oluşturulan bir kesinti `KeyboardInterrupt`istisnayı yükselterek bildirilir .

```
>>>
```

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

`try`Açıklamada, şunlar çalışır.

- İlk olarak *try cümlesi* (`try`ve `except`anahtar kelimeler arasındaki deyimler) yürütülür.

- Herhangi bir istisna oluşmazsa, *hariç yan tümcesi* atlanır ve `try`ifadenin yürütülmesi tamamlanır.
- Try yan tümcesinin yürütülmesi sırasında bir istisna oluşursa, yan tümce atlanır. Daha sonra, türü `except`anahtar kelimeden sonra adlandırılan istisna ile eşleşirse , hariç yan tümcesi yürütülür ve sonra deyimden sonra yürütme devam eder `try`.
- Dışlama yantümcesinde belirtilen kural dışı durumla eşleşmeyen bir kural dışı durum oluşursa, dış `try`deyimlere aktarılır ; hiçbir işleyici bulunamazsa, *işlenmeyen bir istisnadır* ve yürütme yukarıda gösterildiği gibi bir mesajla durur.

Bir `try`ifadenin, farklı istisnalar için işleyicileri belirtmek üzere yan tümcesi hariç birden fazla olabilir. En fazla bir işleyici çalıştırılır. İşleyiciler, aynı `try`ifadenin diğer işleyicilerinde değil, yalnızca ilgili try yan tümcesinde oluşan istisnaları işler . Dışlama yantümcesi, birden çok istisnayı parantez içine alınmış bir demet olarak adlandırabilir, örneğin:

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

Bir cümledeki sınıf `except`, aynı sınıf veya bunun bir temel sınıfı ise istisna ile uyumludur (ancak bunun tersi değil - türetilmiş bir sınıfı listeleyen bir fıkra, bir temel sınıfla uyumlu değilse). Örneğin, aşağıdaki kod bu sırayla B, C, D yazdıracaktır:

```
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")
```

Eğer hariç cümle tersine çevrilmişse (ilk olarak), B, B, B - basılır, cümle hariç ilk eşleşme tetiklenir.`except B`

Son cümle hariç, joker karakter olarak hizmet vermek için istisna ad (lar) ı hariç tutabilir. Gerçek bir programlama hatasını bu şekilde maskelemek kolay olduğu için bunu çok dikkatli kullanın! Ayrıca bir hata mesajı yazdırmak ve daha sonra istisnayı yeniden yükseltmek için de kullanılabilir (arayanın istisnayı da işlemesine izin verilir):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

`try`... `except`ifadesi opsiyonel sahip *başka maddesini* mevcut, maddeleri dışındaki tüm uymalı,. Try deyimi bir istisna oluşturmazsa çalıştırılması gereken kod için kullanışlıdır. Örneğin:

```
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()
```

Cümlenin kullanımı, cümleye `else`ek kod eklemekten daha iyidir `try`çünkü yanlışlıkla `try`... `except`ifadesi tarafından korunan kod tarafından yükseltilmeyen bir istisnayı yakalamaktan kaçınır .

Bir kural dışı durum oluştuğunda, kural dışı durumun *argümanı* olarak da bilinen ilişkilendirilmiş bir değeri olabilir . Argümanın varlığı ve türü istisna türüne bağlıdır.

Hariç yan tümcesi istisna adından sonra bir değişken belirtebilir. Değişken, içinde depolanan argümanların bulunduğu bir istisna örneğine bağlıdır `instance.args`. Kolaylık olması açısından, istisna örneği `__str__()`,

bağımsız değişkenlerin başvurmak zorunda kalmadan doğrudan yazdırılabilmesini sağlar `.args`. Ayrıca, bir istisnayı yükseltmeden önce başlatabilir ve istediği nitelikleri ekleyebilirsiniz.

```
>>>
```

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     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
```

Bir istisnanın bağımsız değişkenleri varsa, bunlar işlenmeyen istisnalar için iletinin son parçası ('ayrıntı') olarak yazdırılır.

İstisna işleyicileri, yalnızca try yan tümcesinde hemen ortaya çıkarsa istisnaları değil, aynı zamanda try yan tümcesinde (dolaylı olarak bile) çağrılan işlevlerin içinde de ortaya çıkarlar. 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. İstisnaları Artırma

`raise`Deyimi programcının kasıtlı olarak belirtilen bir istisna oluşturmasını sağlar. Örneğin:

```
>>>
```

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

Tek argüman `raise`dile getirilecek istisnayı gösterir. Bu bir istisna örneği veya bir istisna sınıfı (türetilmiş bir sınıf `Exception`) olmalıdır. Bir istisna sınıfı iletilirse, yapıcısını argüman olmadan çağırmak dolaylı olarak somutlaştırılır:

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

Bir istisnanın oluşup oluşmadığını belirlemeniz gerekiyorsa ancak bununla ilgilenmek istemiyorsanız, `raise`ifadenin daha basit bir biçimi istisnayı yeniden yükseltmenize izin verir:

```
>>>
```

```
>>> 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>
NameError: HiThere
```

8.5. Kullanıcı Tanımlı İstisnalar

Programlar yeni bir istisna sınıfı oluşturarak kendi istisnalarını adlandırabilir (Python sınıfları hakkında daha fazla bilgi için Sınıflar bölümüne bakın). İstisnalar tipik olarak `Exception`doğrudan veya dolaylı olarak sınıftan türetilmelidir .

Başka herhangi bir sınıfın yapabileceği her şeyi yapan istisna sınıfları tanımlanabilir, ancak genellikle basit tutulur, genellikle yalnızca hata hakkındaki bilgilerin istisna için işleyiciler tarafından çıkarılmasına izin veren bir dizi özellik sunar. Birkaç farklı hata oluşturabilen bir modül oluştururken, yaygın bir uygulama, bu modül tarafından tanımlanan istisnalar için bir temel sınıf ve farklı hata durumları için özel istisna sınıfları oluşturmak için alt sınıf oluşturmaktır:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
```

```
class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error
occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition
that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is
not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Çoğu istisna, standart istisnaların adlarına benzer şekilde, "Hata" ile biten adlarla tanımlanır.

Birçok standart modül, tanımladıkları işlevlerde oluşabilecek hataları bildirmek için kendi istisnalarını tanımlar. Sınıflar hakkında daha fazla bilgi Sınıflar bölümünde sunulmaktadır .

8.6. Temizleme Eylemlerini Tanımlama

`try`İfadesi her koşulda idam edilmelidir temizlik işlemleri belirten seçimlik bir bloğu vardır. Örneğin:

```
>>>
```

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
```

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

Bir `finally`cümle mevcutsa, `finally` cümle, `try` ifade tamamlanmadan önce son görev olarak yürütülür . `finally`Tümcesi olsun veya olmasın çalışan `try`deyimi bir istisna oluşturur. Aşağıdaki noktalar, bir istisna oluştuğunda daha karmaşık vakaları tartışır:

- Cümlenin yürütülmesi sırasında bir istisna oluşursa `try` , istisna bir `except` cümle tarafından işlenebilir . Kural dışı durum bir `except` yan tümce tarafından ele alınmazsa, kural `finally` yürütüldükten sonra kural dışı durum yeniden kaldırılır .
- Bir `except` veya `else`cümlesinin yürütülmesi sırasında bir istisna oluşabilir . Yine, `finally`fıkra yürütüldükten sonra istisna yeniden ortaya çıkar .
- Eğer `try`deyimi bir ulaştığında `break`, `continue`ya `return`deyimi, `finally`fıkra hemen önce çalıştırır `break`, `continue`veya `return` ifadenin çalıştırılması.
- Bir ederse `finally`yantümce içeren `return` bildiri, döndürülen değer dan biri olacak `finally`Clause en `return`açıklamada, değil gelen değer `try`Clause en `return` açıklamada.

Örneğin:

```
>>>
```

```
>>> 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>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Gördüğünüz gibi, `finally`madde herhangi bir olayda yürütülür. `TypeError`İki dizeyi bölerek kaldırdı tarafından ele edilmez `except`maddesi ve sonrasında bu nedenle istisna tekrar `finally` bloğu icra edilmiştir.

Gerçek dünya uygulamalarında, bu `finally`madde, kaynak kullanımının başarılı olup olmadığına bakılmaksızın dış kaynakları (dosyalar veya ağ bağlantıları gibi) serbest bırakmak için yararlıdır.

8.7. Önceden Tanımlanmış Temizleme İşlemleri

Bazı nesneler, nesneyi kullanma işleminin başarılı olup olmadığına bakılmaksızın, nesneye artık ihtiyaç duyulmadığında 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 bittikten sonra dosyayı belirsiz bir süre açık bırakmasıdır. Bu basit komut dosyalarında bir sorun değildir, ancak daha büyük uygulamalar için bir sorun olabilir. `with`Deyimi dosyaları gibi nesneleri her zaman derhal ve doğru şekilde temizlenir sağlayan bir şekilde kullanılmasını sağlar.

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

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