Errors, Exceptions,

Iterators, Generators

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>>
>>> 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 a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: 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.

https://docs.python.org/3/tutorial/errors.html

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 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
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred.

Handling Exceptions¶

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

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

The `try` statement works as follows.

- First, the try clause (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.
- 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 a message as shown above.

A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

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

```
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
```

The `try` … `except` statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
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()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` … `except` statement.

Raising Exceptions¶

The `raise` statement allows the programmer to force a specified exception to occur. For example:

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

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

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

Defining Clean-up Actions¶

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

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

Iterators, Iterables

An ITERABLE is:

- anything that can be looped over (i.e. you can loop over a string or file)
- anything that can appear on the right-side of a for-loop: `for x in iterable: ...`
- anything you can call with `iter()` have it return an ITERATOR: `iter(obj)`
- an object that defines `__iter__` that returns a fresh ITERATOR, or it may have a `__getitem__` method suitable for indexed lookup.

```
s = 'cat'
```

```
for si in s:
    print si
```

```
c
a
t
```

```
s.__getitem__
```

```
<method-wrapper '__getitem__' of str object at 0x10
```

```
n = [1,4,16,32]
```

```
for ni in n:
  print ni,
```

```
1 4 16 32
```

```
iter(n)
```

```
<listiterator at 0x10deebcd0>
```

```
n.__iter__
```

```
<method-wrapper '__iter__' of list object at 0x10d7
```

```
s.__iter__
```

```
---------------------------------------------------
------------
AttributeError                            Traceback
nt call last)
```

```
for k in {"x": 1, "y": 2}:
    print k
```

```
y
x
```

```
for line in open("/Users/beerli/a.txt"):
    print line
```

```
The quick fox jumps of the lazy dog

The red fox is running away
```

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

Many functions consume iterables

An ITERATOR is:

- an object with state that remembers where it is during iteration
- an object with a `__next__` method (Python 3; `next`) before that:
  - returns the next value in the iteration
  - updates the state to point at the next value
  - signals when it is done by raising `StopIteration`
- an object that is self-iterable (meaning that it has an `__iter__` method that returns self).

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
niter = iter(n)
```

```
for ni in niter:
    print ni
```

```
1
4
16
32
```

Generators

facilitate the construction of iterators

```
def zrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

http://anandology.com/python-practice-book/iterators.html

```
z = zrange(3)
```

```
z
```

```
<generator object zrange at 0x10def4500>
```

```
z.next()
```

```
0
```

```
z.next()
```

```
1
```

```
z.next()
```

```
2
```

```
z.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-34-6c49e4c11a56> in <module>()
----> 1 z.next()

StopIteration:
```

So a generator is also an iterator. You don't have to worry about the iterator protocol.

The word "generator" is confusingly used to mean both the function that generates and what it generates.

We will use the word "generator" to mean the generated object and "generator function" to mean the function that generates it.

Can you think about how it is working internally?

When a generator function is called, it returns a generator object without even beginning execution of the function. When *next* method is called for the first time, the function starts executing until it reaches `yield` statement. The yielded value is returned by the `next` call.

The following example demonstrates the interplay between `yield` and call to `next` method on generator object.

http://anandology.com/python-practice-book/iterators.html

```
>>> def foo():
...     print "begin"
...     for i in range(3):
...         print "before yield", i
...         yield i
...         print "after yield", i
...     print "end"
...
>>> f = foo()
>>> f.next()
begin
before yield 0
0
>>> f.next()
after yield 0
before yield 1
1
>>> f.next()
after yield 1
before yield 2
2
>>> f.next()
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

http://anandology.com/python-practice-book/iterators.html

```
def integers():
    """Infinite sequence of integers."""
    i = 0
    while True:
        yield i
        i = i + 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(seq.next())
    except StopIteration:
        pass
    return result

print take(5, squares())
```

```
[0, 1, 4, 9, 16]
```

```
print take(6,integers())
```

```
[0, 1, 2, 3, 4, 5]
```

```
def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(seq.next())
    except StopIteration:
        pass
    return result
```

```
def fibonacchi():
    last = 1
    secondlast = 0
    yield 1
    for i in integers():
        yield (secondlast + last)
        secondlast, last = last, secondlast + last
```

```
print take(20,fibonacchi())
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

5.3. Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

We can use the generator expressions as arguments to various functions that consume iterators.

```
>>> sum((x*x for x in range(10)))
285
```

When there is only one argument to the calling function, the parenthesis around generator expression can be omitted.

http://anandology.com/python-practice-book/iterators.html

Another example:
Lets say we want to find first 10 (or any n) pythagorian triplets.
A triplet `(x, y, z)` is called pythagorian triplet if `x*x + y*y == z*z`.

It is easy to solve this problem if we know till what value of *z* to test for.
But we want to find first n pythagorian triplets.

```
pyt = ((x, y, z) for z in integers() for y in xrange(1, z) for x in range(1, y) if x*x + y*y == z*z)
```

```
take(10, pyt)
```

```
[(3, 4, 5),
 (6, 8, 10),
 (5, 12, 13),
 (9, 12, 15),
 (8, 15, 17),
 (12, 16, 20),
 (15, 20, 25),
 (7, 24, 25),
 (10, 24, 26),
 (20, 21, 29)]
```

http://anandology.com/python-practice-book/iterators.html

5.3.1. Example: Reading multiple files

Lets say we want to write a program that takes a list of filenames as arguments and prints contents of all those files, like `cat` command in unix.

The traditional way to implement it is:

```
def cat(filenames):
    for f in filenames:
        for line in open(f):
            print line,
```

Now, lets say we want to print only the line which has a particular substring, like `grep` command in unix.

```
def grep(pattern, filenames):
    for f in filenames:
        for line in open(f):
            if pattern in line:
                print line,
```

Both these programs have lot of code in common. It is hard to move the common part to a function. But with generators makes it possible to do it.

http://anandology.com/python-practice-book/iterators.html

```python
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in lines)

def printlines(lines):
    for line in lines:
        print line,
```

```python
def main(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

The code is much simpler now with each function doing one small thing. We can move all these functions into a separate module and reuse it in other programs.

http://anandology.com/python-practice-book/iterators.html

5.4. Itertools

The itertools module in the standard library provides lot of intersting tools to work with iterators.

Lets look at some of the interesting functions.

chain – chains multiple iterators together.

```
>>> it1 = iter([1, 2, 3])
>>> it2 = iter([4, 5, 6])
>>> itertools.chain(it1, it2)
[1, 2, 3, 4, 5, 6]
```

izip – iterable version of zip

```
>>> for x, y in itertools.izip(["a", "b", "c"], [1, 2, 3]):
...     print x, y
...
a 1
b 2
c 3
```

http://anandology.com/python-practice-book/iterators.html

```
import itertools
```

```
dir(itertools)
```

```
['__doc__',
 '__file__',
 '__name__',
 '__package__',
 'chain',
 'combinations',
 'combinations_with_replacement',
 'compress',
 'count',
 'cycle',
 'dropwhile',
 'groupby',
 'ifilter',
 'ifilterfalse',
 'imap',
 'islice',
```

to scroll output; double click to hide

```
 'izip_longest',
 'permutations',
 'product',
 'repeat',
 'starmap',
 'takewhile',
 'tee']
```

```
help(itertools)
```

```
Help on module itertools:

NAME
    itertools - Functional tools for creating and using iterators.
```

logy.com/python-practice-book/iterators.html

Problem 2: Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.

Problem 3: Write a function `findfiles` that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.

Problem 4: Write a function to compute the number of python files (.py extension) in a specified directory recursively.

Problem 5: Write a function to compute the total number of lines of code in all python files in the specified directory recursively.

Problem 6: Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

Problem 7: Write a program `split.py`, that takes an integer `n` and a filename as command line arguments and splits the file into multiple small files with each having `n` lines.

http://anandology.com/python-practice-book/iterators.html

http://anandology.com/python-practice-book/iterators.html