Part III

11. Iterable objects and Iterator objects

A finite collection object is said to *iterable* if it has a unique "first" element and a unique "last" element; and each element of the collection except the last has a unique "next" element.

An *iterator* object may be created from an iterable object and then used to iterate over the elements of the iterable collection object.

In the case of a sequence object C, ***C is its own iterator***.

The distinction between iterable and iterator is only important when we are creating our own custom collection classes and we want to be able to process one element of our collection at a time. We do this by defining in our class definition a special method named _next(). If your class has such a method, the built-in polymorphic function `next()` will call that method to get a following element in your collection: `next(x)` will produce the successor of `x` in your collection.

It `myiter` is an iterator, then you may use it in a `for` statement:

```
for x in myiter:
    …
```

Since a list `L` is an iterator for itself, we are able to write just `for x in L:`

We introduce these concepts at this point because some built-in functions we wish to discuss return iterators.

Note: any finite iterator object `X` may be converted to a list by `list(X)` and to a tuple by `tuple(X)`.

So now, on to more about strings.

12. Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The `upper()`, `lower()`, `isupper()`, and `islower()` String Methods

The `upper()` and `lower()` string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Characters in the string which are not letters remain unchanged.

Examples:

```
>>> spam = 'Hello world!'

>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

Note that these methods do not change the string itself but return new string values. If you want to change the original string, you have to call `upper()` or `lower()` on the string and then assign the new string to the variable that referenced the original string. This is why you must use `spam = spam.upper()` to change the string in spam instead of simply `spam.upper()`.

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. The strings 'great' and 'GREat' are not equal to each other. But in the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

When you run this program, the question is displayed, and entering a variation on great, such as GREat, will still give the output I feel great too. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?
GREat
I feel great too.
```

The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns `False`. Notice what each method call below returns.

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on those returned string values as well. Expressions that do this will look like a chain of method calls. Interactive shell example:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The `isX` String Methods

Along with `islower()` and `isupper()`, there are several string methods that have names beginning with the word `is`. These methods return a Boolean value that describes the nature of the string. Here are some common `isX` string methods:

`isalpha()`	returns `True` if the string consists only of letters and is not blank.
`isalnum()`	returns `True` if the string consists only of letters and numbers and is not blank.
`isdecimal()`	returns `True` if the string consists only of digit characters and is not blank.
`isspace()`	returns `True` if the string consists only of spaces, tabs, and new- lines and is not blank.
`istitle()`	returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '     '.isspace()
True
>>> 'This  Is  Title Case'.istitle()
True
>>> 'This Is Title Case  123'.istitle()
True
>>> 'This Is not Title   Case'.istitle()
False
>>> 'This  Is  NOT  Title  Case Either'.istitle()
False
```

The `isX` string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input.

```
while True:
    print('Enter your age:') age = input()
    if age.isdecimal():
        break
    print('Please enter an integer for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
         break
    print('Passwords can only have letters and numbers.')
```

The `startswith()` and `endswith()` methods return `True` if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return `False`.

```
>>> 'Hello   world!'.startswith('Hello')
True
>>> 'Hello   world!'.endswith('world!')
True
>>>  'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello  world!'.startswith('Hello world!')
True
>>> 'Hello   world!'.endswith('Hello  world!')
True
```

These methods are useful alternatives to the == equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called from a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list with the base string in between consecutive strings of the list.

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My',  'name',  'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string `join()` is called from is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is `'cats, rats, bats'`.

Remember that `join()` is called from a string value and is passed a list value. (It’s easy to accidentally call it the other way around.)

The `split()` method does the opposite of the `join()` method: It’s called on a string value and returns a list of strings.

Interactive shell example:

```
>>> 'My  name  is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string *'My name is Simon'* is split wherever consecutive whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the split() method to specify a different string to split upon. For example:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My  name  is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters.

```
>>> spam   =   '''Dear  Alice,
How have you  been?  I am  fine. There is a container in the fridge that
is labeled "Milk  Experiment".

Please do not drink it. Sincerely,
Bob'''

>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in
the fridge', 'that is labeled "Milk Experiment".', '', 'Please do not
drink it.', 'Sincerely,', 'Bob']
```

Passing `split()` the argument `'\n'` lets us split the multiline string stored in spam along the newlines and return a list in which each item corresponds to one line of the string.

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string.

```
>>> 'Hello'.rjust(10)
'    Hello'
>>> 'Hello'.rjust(20)
'    Hello'
>>> 'Hello   World'.rjust(20)
'    Hello World'
>>> 'Hello'.ljust(10)
'Hello    '
```

`'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length 10. `'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.
An optional second argument to rjust() and ljust() will specify a fill character other than a space character.

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text rather than justifying it to the left or right.

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

These methods are especially useful when you need to print tabular data that has the correct spacing. Open a new file editor window and enter the following code, saving it as

```
# file picnicTable.py
def printPicnic(itemsDict, leftWidth, rightWidth):
   print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
   for k, v in itemsDict.items():
      print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4,
               'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.
To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

`printPicnic()` takes in a dictionary; a `leftWidth` for the left column of a table; and a `rightWidth` for the right column.

It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call `printPicnic()` twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

Using `rjust()`, `ljust()`, and `center()` lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without whitespace characters at the beginning or end. The `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively.

```
>>> spam = '    Hello World    '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World    '
>>> spam.rstrip()
'    Hello World'
```

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

```
>>> spam  =  'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')# remove beginning and end characters of spam that
                      # are any of the characters 'a','m','p','S'
'BaconSpamEggs'
```

Passing `strip()` the argument `'ampS'` will tell it to strip occurences of `a`, `m`, `p`, and capital `S` from the ends of the string stored in spam. The order of the characters in the string passed to `strip()` does not matter: `strip('ampS')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

13.Boolean Context

Every python object can be used in a "Boolean context", that is, where a logical value is expected. In C++, integers can be used in a Boolean context, where 0 is interpreted as false and any other integer is interpreted as true.

In Python an object of any built-in type may be used in a Boolean context. Integers behave in the same way as in C++. And floats behave the same as integers. For container types like strings, lists, tuples and dictionaries, an empty container is interpreted as False and a nonempty container as True.

For example, given a list object L, instead of writing `if L != [], you` can be simply write `if L`.

One of the interesting features of Python is how the value of a Boolean expression is determined. As in C++, short-circuit evaluation is used. As soon as the result of the evaluation is known, further terms are ignored. For all Boolean operators except **`not`**, the expression returns the object that determined the result (which could then be viewed as True or False). When the **not** operator is applied to a non-boolean object, it always returns either True or False. Here are some examples.

0 **and** 2	returns	0
2 **and** 0	returns	0
0 **or** 2	returns	2
[] **and** 2	returns	[]
2 **and** []	returns	[]
(1,2) **or** (2,3)	returns	(1,2)
(1,2) **and** (2,3)	returns	(2,3)
-5 **or** 0	returns	-5
not 2	returns	False
not 0.0	returns	True
0 == False	returns	True
2 == True	returns	False
1 == True	returns	True

14. Files

Before reading or writing a file, you need to open it:

```
fileobj = open( filename, mode )
```

Here's a brief explanation of the pieces of this call:

- `fileobj` is the file object returned by `open()`
- `filename` is the string name of the file
- `mode` is a string indicating the file's type and what you want to do with it

The first letter of mode indicates the operation:

- `r` means read.
- `w` means write. If the file doesn't exist, it's created. If the file does exist, it's overwritten.
- `x` means write, but only if the file does not already exist.
- `a` means append (write after the end) if the file exists.

The second letter of mode is the file's type:

- t (or nothing) means text.
- b means binary.

After opening the file, you call functions to read or write data; these will be shown in the examples that follow. Last, you need to close the file.

Let's create a file from a Python string in one program and then read it back in the next.

Write a Text File with write()

For some reason, there aren't many limericks about special relativity. This one will just have to do for our data source:

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

The following code writes the entire poem to the file 'relativity' in one call:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
>>> fout.close()
```

The `write()` function returns the number of bytes written. It does not add any spaces or newlines, as print() does. You can also print() to a text file:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

This brings up the question: should I use `write()` or `print()`? By default, `print()` adds a space after each argument and a newline at the end. In the previous example, it appended a newline to the relativity file. To make `print()` work like write(), pass the following two arguments:
• `sep` (separator, which defaults to a space, ' ')
• `end` (end string, which defaults to a newline, '\n')

`print()` uses the defaults unless you pass something else. We'll pass empty strings to suppress all of the fussiness normally added by `print()`:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

If you have a large source string, you can also write chunks until the source is done:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...    if offset > size:
...    break
... fout.write(poem[offset:offset+chunk])
... offset += chunk
...
100
50
>>> fout.close()
```

This wrote 100 characters on the first try and the last 50 characters on the next.

If the relativity file is precious to us, let's see if using mode x really protects us from overwriting it:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

Read a Text File with read(), readline(), or readlines()

You can call read() with no arguments to slurp up the entire file at once, as shown in the example that follows. Be careful when doing this with large files; a gigabyte file willconsume a gigabyte of memory.

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

You can provide a maximum character count to limit how much read() returns at one time. Let's read 100 characters at a time and append each chunk to a poem string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...    fragment = fin.read(chunk)
...    if not fragment:
...       break
...    poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

After you've read all the way to the end, further calls to `read()` will return an empty string (`''`), which is treated as `False` in the `if not` fragment. This breaks out of the `while True` loop.

You can also read the file a line at a time by using readline(). In this next example, we'll append each line to the poem string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...        break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

For a text file, even a blank line has a length of one (the newline character), and is evaluated as `True`. When the file has been read, `readline()` (like `read()`) also returns an empty string, which is also evaluated as `False`.

The easiest way to read a text file is by using an iterator. This returns one line at a time. It's similar to the previous example, but with less code:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

All of the preceding examples eventually built the single string poem. The `readlines()` call reads a line at a time, and returns a list of one-line strings:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...    print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

We told `print()` to suppress the automatic newlines because the first four lines already had them. The last line did not, causing the interactive prompt `>>>` to occur right after the last line.

The types `bytes` and `bytearray`

In Python 3, a string object (type `str`) is a sequence of **Unicode characters**. Since a Unicode character may be represented by more than one byte, the number of characters in a str object is not necessarily the same as the number of bytes it occupies.

The types bytes and bytearray are for sequences of 8-bit integers (bytes). They are used for binary data.

Beginning with a list called `blist`, this next example creates a `bytes` variable called `the_bytes` and a `bytearray` variable called t`he_byte_array`:

```
>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
```

The representation of a bytes value begins with a b and a quote character, followed by hex sequences such as \x02 or ASCII characters, and ends with a matching quote character.

Python converts the hex sequences or ASCII characters to 8-bit integers, but shows byte values that are also valid ASCII encodings as ASCII characters.

```
>>> b'\x61'
b'a'
>>> b'\x01abc\xff'
b'\x01abc\xff'
```

Each of these would create a 256-element result, with values from 0 to 255:

```
>>> the_bytes = bytes(range(0, 256))
>>> the_byte_array = bytearray(range(0, 256))
```

When printing `bytes` or `bytearray` data, Python uses `\x`*`yy`* where the *y*s are hex digits, for non-printable bytes; and their ASCII equivalents for printable ones (plus some common escape characters, such as `\n` instead of `\x0a`). Here's the printed representation of the_bytes (manually reformatted to show 16 bytes per line):

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

This can be confusing, because they're bytes (teeny integers), not characters.

When a file is opened in binary mode, the `read()` method returns a `bytes` object. This would be the case when you were processing, say, an image file.

The str.encode() method converts a Unicode string to a bytes object and the bytes.decode() – and the bytearray.decode() method decode() convert a byte string to a unicode string (or throw an exception if it fails). With no argument, the encoding is assumed to be utf-8. However you may pass in a parameter to specify another encoding.

Write a Binary File with write()

If you include a `'b'` in the mode string, the file is opened in binary mode. We don't have a binary poem lying around, so we'll just generate the 256 byte values from 0 to 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Open the file for writing in binary mode and write all the data at once:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

Again, write() returns the number of bytes written. As with text, you can write binary data in chunks:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...    if offset > size:
...       break
...    fout.write(bdata[offset:offset+chunk])
...    offset += chunk
...
100
100
56
>>> fout.close()
```

Read a Binary File with read()

This one is simple; all you need to do is just open with 'rb':

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> type(bdata)
<class bytes>
>>> len(bdata)
256
>>> fin.close()
```

Close Files Automatically by Using with

If you forget to close a file that you've opened, it will be closed by Python after it's no longer referenced. This means that if you open a file within a function and don't close it explicitly, it will be closed automatically when the function ends. But you might have opened the file in a long-running function or the main section of the program. The file should be closed to force any remaining writes to be completed.

Python has *context managers* to clean up things such as open files. You use the form `with expression as variable`:

```
>>> with open('relativity', 'wt') as fout:
...        fout.write(poem)
```

That's it. After the block of code under the context manager (in this case, one line) completes (normally or by a raised exception), the file is closed automatically.

You are now ready to do the *StringsAndFiles* assignment