Lecture 1.1

Defining Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime and can be modified further after creation.

The simplest form of class definition looks like this:

class ClassName:

```
<statement-1>
.
.
.
<statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

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.

The_init_() Method

The __init__ method is similar to **constructors** in C++ and Java. Constructors are used to initialize the object's state. The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created. Like methods, a constructor also contains collection of statements(i.e. instructions) that are executed at time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name


    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Tommy')
p.say_hi()
```

```
Hello, my name is Tommy
```

Understanding the code

In the above example, a person name Tommy is created. While creating a person, "Tommy" is passed as an argument, this argument will be passed to the __init__ method to initialize the object. The keyword self represents the instance of a class and binds the attributes with the given arguments. Similarly, many objects of Person class can be created by passing different names as arguments.

Instantiating Classes and Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
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".

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable x.

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)
```

The self-Parameter

The self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. The reason why we use self is that Python does not use the '@' syntax to refer to instance attributes. In Python, we have methods that make the instance to be passed automatically, but not received automatically.

Example:

```
class food():

     # init method or constructor
     def __init__(self, fruit, color):
          self.fruit = fruit
          self.color = color

     def show(self):
          print("fruit is", self.fruit)
          print("color is", self.color )

     apple = food("apple", "red")
     grapes = food("grapes", "green")

     apple.show()
     grapes.show()
```

Output:

```
Fruit is apple
color is red
Fruit is grapes
color is green
```

Python Class self-Constructor

self is also used to refer to a variable field within the class. Let us take an example and see how it works:

```
class Person:

     # name made in constructor
     def __init__(self, John):
          self.name = John
```

```
        def get_person_name(self):
             return self.name
```

In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

Is self a Keyword?

self is used in different places and often thought to be a keyword. But unlike in C++, self is not a keyword in Python.

Delete Python object

A class implements the special method __del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed –

#!/usr/bin/python

```
class Point:
   def __init__( self, x=0, y=0):
      self.x = x       self.y = y    def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"
pt1 = Point()
pt2 = pt1 pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

Output

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
```

```
Point destroyed
```

Instance Variables:

Declared inside the constructor method of class (the __init__ method). They are tied to the particular object instance of the class; hence the contents of an instance variable are completely independent from one object instance to the other.

Let's start with a short and easy-to digest example:

```
class Car:
    wheels = 4  # <- Class variable
     def __init__(self, name):
        self.name = name    # <- Instance variable
```

Above is the basic, no-frills *Car* class defined. Each instance of it will have class variable *wheels* along with the instance variable *name*. Let's instantiate the *class* to access the variables.

```
>>> jag = Car('jaguar')
>>> fer = Car('ferrari')>>> jag.name, fer.name
('jaguar', 'ferrari')>>> jag.wheels, fer.wheels
(4, 4)>>> Car.wheels
4
```

Accessing the instance variable *name* is pretty straight forward. However there is a little more flexibility when it comes to access the class variable. As above, we can access *wheels* via the object instance or the *Class* itself.

Also trying to access the *name* through the class will result in an *AttributeError* since instance variables are object specific and are created when __init__ constructor is invoked. This is the central distinction between the class and instance variables.

```
>>> Car.name
AttributeError: type object 'Car' has no attribute 'name'
```

Now, let's assume for time being that our *Jaguar* car has 3 wheels.

To represent that in our code, we can modify the *wheels* variable.

```
>>> Car.wheels = 3
```

What we did above will make all cars with 3 wheels since we have modified a class variable, which will apply to all instances of Car class.

```
>>> jag.wheels, fer.wheels
(3, 3)
```

Therefore, modifying a class variable on the class namespace affects all the instances of the class. Let's roll back the change and modify the wheels variable using the *jag* object.

```
>>> Car.wheels = 4
>>> jag.wheels = 3
```

This will give us the output we desire.

```
>>> jag.wheels, fer.wheels, Car.wheels
(3, 4, 4)
```

Although we got the result we wanted, but what happened behind the scenes is a new *wheels* variable that has been added to *jag* object and this new variable shadows the class variable with same name, overriding and hiding it. We can access both the *wheels* variable as below.

```
>>> jag.wheels, jag.__class__.wheels
(3, 4)
```

Hence we can conclude that `jag.wheels = 3` created a new instance variable with the same name as the class variable (*wheels).*

Lecture 1.2
A Fibonacci Iterator

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator:

Example: Return an iterator from a tuple, and print each value

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator:

We can also use a for loop to iterate through an iterable object:

Example: Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

Example: A Fibonacci Iterator

```
class Fib:                                              ①
    def __init__(self, max):                            ②
        self.max = max

    def __iter__(self):                                 ③
        self.a = 0
        self.b = 1
        return self

    def __next__(self):                                 ④
        fib = self.a
        if fib > self.max:
            raise StopIteration                         ⑤
        self.a, self.b = self.b, self.a + self.b
        return fib                                      ⑥
```

1) To build an iterator from scratch, Fib needs to be a class, not a function.
2) "Calling" Fib(max) is really creating an instance of this class and calling its __init__() method with max. The __init__() method saves the maximum value as an instance variable so other methods can refer to it later.
3) The __iter__() method is called whenever someone calls iter(fib). After performing beginning-of-iteration initialization, the __iter__() method can return any object that implements a __next__() method. In this case, __iter__() simply returns self, since this class implements its own __next__() method.
4) The __next__() method is called whenever someone calls next() on an iterator of an instance of a class.
5) When the __next__() method raises a StopIteration exception, this signals to the caller that the iteration is exhausted. Unlike most exceptions, this is not an error; it is a normal condition that just means that the iterator has no more values to generate. If the caller is a for loop, it will notice this StopIteration exception and gracefully exit the loop. This little bit of magic is the key to using iterators in for loops.
6) To spit out the next value, an iterator's __next__() method simply returns the value. Do not use yield here; that is a bit of syntactic sugar that only applies when you're using generators. Here you are creating your own iterator from scratch; use return instead.

Let us see how to call this iterator:

```
>>> from fibonacci2 import Fib
>>> for n in Fib(1000):
...     print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

There is a bit of magic involved in for loops. Here is what happens:

- The for loop calls Fib(1000), as shown. This returns an instance of the Fib class. Call this fib_inst.
- Secretly, and quite cleverly, the for loop calls iter(fib_inst), which returns an iterator object. Call this fib_iter. In this case, fib_iter == fib_inst, because the __iter__() method returns self, but the for loop doesn't know (or care) about that.
- To "loop through" the iterator, the for loop calls next(fib_iter), which calls the __next__() method on the fib_iter object, which does the next-Fibonacci-number calculations and returns a value. The for loop takes this value and assigns it to n, then executes the body of the for loop for that value of n.
- How does the for loop know when to stop? When next(fib_iter) raises a StopIteration exception, the for loop will swallow the exception and gracefully exit.

A Plural Rule Iterator

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename,
encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self

    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
```

```
        return funcs

rules = LazyRules()
```

So this is a class that implements __iter__() and __next__(), so it can be used as an iterator. Then, you instantiate the class and assign it to rules. This happens just once, on import.

Let us take the class one bite at a time.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename,
                                 encoding='utf-8')    ①
        self.cache = []                               ②
```

1) When we instantiate the LazyRules class, open the pattern file but do not read anything from it.
2) After opening the patterns file, initialize the cache. You'll use this cache later (in the __next__() method) as you read lines from the pattern file.

Before we continue, let's take a closer look at rules_filename. It's not defined within the __iter__() method. In fact, it's not defined within *any* method. It is defined at the class level. It is a *class variable*, and although you can access it just like an instance variable (self.rules_filename), it is shared across all instances of the LazyRules class.

```
>>> import plural6
>>> r1 = plural6.LazyRules()
>>> r2 = plural6.LazyRules()
>>> r1.rules_filename ①
'plural6-rules.txt'
>>> r2.rules_filename
'plural6-rules.txt'
>>> r2.rules_filename = 'r2-override.txt' ②
>>> r2.rules_filename
'r2-override.txt'
>>> r1.rules_filename
'plural6-rules.txt'
>>> r2.__class__.rules_filename ③
'plural6-rules.txt'
```

```
>>> r2.__class__.rules_filename = 'papayawhip.txt' ④
>>> r1.rules_filename
'papayawhip.txt'
>>> r2.rules_filename ⑤
'r2-overridetxt'
```

1) Each instance of the class inherits the rules_filename attribute with the value defined by the class.
2) Changing the attribute's value in one instance does not affect other instances…
3) …nor does it change the class attribute. You can access the class attribute (as opposed to an individual instance's attribute) by using the special __class__ attribute to access the class itself.
4) If you change the class attribute, all instances that are still inheriting that value (like r1 here) will be affected.
5) Instances that have overridden that attribute (like r2 here) will not be affected.

```
def __iter__(self):          ①
    self.cache_index = 0
    return self              ②
```

1) The __iter__() method will be called every time someone — say, a for loop — calls iter(rules).
2) The one thing that every __iter__() method must do is return an iterator. In this case, it returns self, which signals that this class defines a __next__() method which will take care of returning values throughout the iteration.

```
def __next__(self):                                    ①
    .
    .
    .
    pattern, search, replace = line.split(None, 3)
    funcs = build_match_and_apply_functions(           ②
        pattern, search, replace)
    self.cache.append(funcs)                           ③
    return funcs
```

1) The __next__() method gets called whenever someone — say, a for loop — calls next(rules). This method will only make sense if we start at the end and work backwards. So let's do that.
2) The last part of this function should look familiar, at least. The build_match_and_apply_functions() function hasn't changed; it's the same as it ever was.

3) The only difference is that, before returning the match and apply functions (which are stored in the tuple funcs), we're going to save them in self.cache.

```
def __next__(self):
    .
    .
    .
    line = self.pattern_file.readline()   ①
    if not line:                          ②
        self.pattern_file.close()
        raise StopIteration               ③
    .
    .
    .
```

1) A bit of advanced file trickery here. The readline() method (note: singular, not the plural readlines()) reads exactly one line from an open file. Specifically, the next line. (File objects are iterators too! It's iterators all the way down…)
2) If there was a line for readline() to read, line will not be an empty string. Even if the file contained a blank line, line would end up as the one-character string '\n' (a carriage return). If line is really an empty string, that means there are no more lines to read from the file.
3) When we reach the end of the file, we should close the file and raise the magic StopIteration exception. Remember, we got to this point because we needed a match and apply function for the next rule. The next rule comes from the next line of the file… but there is no next line! Therefore, we have no value to return. The iteration is over.

```
def __next__(self):
    self.cache_index += 1
    if len(self.cache) >= self.cache_index:
        return self.cache[self.cache_index - 1]      ①

    if self.pattern_file.closed:
        raise StopIteration                          ②
    .
    .
    .
```

1) self.cache will be a list of the functions we need to match and apply individual rules. (At least that should sound familiar!) self.cache_index keeps track of which cached item we should return next. If we haven't exhausted the cache yet (i.e. if the length of self.cache is greater than self.cache_index), then we have a cache hit! Hooray! We can return the match and apply functions from the cache instead of building them from scratch.

2) On the other hand, if we don't get a hit from the cache, and the file object has been closed (which could happen, further down the method, as you saw in the previous code snippet), then there's nothing more we can do. If the file is closed, it means we have exhausted it — we have already read through every line from the pattern file, and we've already built and cached the match and apply functions for each pattern. The file is exhausted; the cache is exhausted; I am exhausted. Wait, what? Hang in there, we are almost done.

Putting it all together, here is what happens when:

- When the module is imported, it creates a single instance of the LazyRules class, called rules, which opens the pattern file but does not read from it.
- When asked for the first match and apply function, it checks its cache but finds the cache is empty. So, it reads a single line from the pattern file, builds the match and apply functions from those patterns, and caches them.
- Let's say, for the sake of argument, that the very first rule matched. If so, no further match and apply functions are built, and no further lines are read from the pattern file.
- Furthermore, for the sake of argument, suppose that the caller calls the plural() function *again* to pluralize a different word. The for loop in the plural() function will call iter(rules), which will reset the cache index but will not reset the open file object.
- The first time through, the for loop will ask for a value from rules, which will invoke its __next__() method. This time, however, the cache is primed with a single pair of match and apply functions, corresponding to the patterns in the first line of the pattern file. Since they were built and cached in the course of pluralizing the previous word, they're retrieved from the cache. The cache index increments, and the open file is never touched.
- Let's say, for the sake of argument, that the first rule does *not* match this time around. So the for loop comes around again and asks for another value from rules. This invokes the __next__() method a second time. This time, the cache is exhausted — it only contained one item, and we're asking for a second — so the __next__() method continues. It reads another line from the open file, builds match and apply functions out of the patterns, and caches them.
- This read-build-and-cache process will continue as long as the rules being read from the pattern file don't match the word we're trying to pluralize. If we do find a matching rule before the end of the file, we simply use it and stop, with the file still open. The file pointer will stay wherever we stopped reading, waiting for the next readline() command. In the meantime, the cache now has more items in it, and if we start all over again trying to pluralize a new word, each of those items in the cache will be tried before reading the next line from the pattern file.

Lecture 2.1

Advanced Iterators

Just as regular expressions put strings on steroids, the itertools module puts iterators on steroids. But first, let us see a classic puzzle.

```
HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246
H = 5
A = 1
W = 0
I = 9
D = 8
O = 3
S = 6
T = 2
E = 4
```

Puzzles like this are called cryptarithms or alphametics. The letters spell out actual words, but if you replace each letter with a digit from 0–9, it also “spells” an arithmetic equation. The trick is to figure out which letter maps to each digit. All the occurrences of each letter must map to the same digit, no digit can be repeated, and no “word” can start with the digit 0.

The most well-known alphametic puzzle is SEND + MORE = MONEY.

We will investigate an incredible Python program originally written by Raymond Hettinger. This program solves alphametic puzzles in just 14 lines of code.

```
import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
```

```
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters,
                                                 guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)
```

You can run the program from the command line. On Linux, it would look like this.

```
you@localhost:~/examples$ python3 alphametics.py "HAWAII + IDAHO
+ IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you@localhost:~/examples$ python3 alphametics.py "I + LOVE + YOU
== DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you@localhost:~/examples$ python3 alphametics.py "SEND + MORE ==
MONEY"
```

```
SEND + MORE == MONEY
9567 + 1085 == 10652
```

Finding all occurrences of a Pattern

The first thing this alphametics solver does is find all the letters (A–Z) in the puzzle.

```
>>> import re
>>> re.findall('[0-9]+', '16 2-by-4s in rows of 8')  ①
['16', '2', '4', '8']
>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY')     ②
['SEND', 'MORE', 'MONEY']
```

1. The re module is Python's implementation of regular expressions. It has a nifty function called findall() which takes a regular expression pattern and a string, and finds all occurrences of the pattern within the string. In this case, the pattern matches sequences of numbers. The findall() function returns a list of all the substrings that matched the pattern.
2. Here the regular expression pattern matches sequences of letters. Again, the return value is a list, and each item in the list is a string that matched the regular expression pattern.

Here is another example that will stretch your brain a little.

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's
sick.")
[' sixth s', " sheikh's s", " sheep's s"]
```

This is the hardest tongue twister in the English language.

Surprised? The regular expression looks for a space, an s, and then the shortest possible series of any character (.*?), then a space, then another s. Well, looking at that input string, we see five matches:

```
1. The sixth sick sheikh's sixth sheep's sick.
2. The sixth sick sheikh's sixth sheep's sick.
3. The sixth sick sheikh's sixth sheep's sick.
4. The sixth sick sheikh's sixth sheep's sick.
5. The sixth sick sheikh's sixth sheep's sick.
```

But the re.findall() function only returned three matches. Specifically, it returned the first, the third, and the fifth. Why is that? Because it does not return overlapping matches. The first match overlaps with the second, so the first is returned and the second is skipped. Then the third overlaps with the fourth, so the third is returned and the fourth is skipped. Finally, the fifth is returned. Three matches, not five.

This has nothing to do with the alphametics solver;

Lecture 2.2

Finding the Unique items in a sequence.

Sets make it trivial to find the unique items in a sequence.

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth',
"sheep's", 'sick']
>>> set(a_list)                          ①
{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'
>>> set(a_string)                        ②
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ''.join(words)                       ③
'SENDMOREMONEY'
>>> set(''.join(words))                  ④
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

1. Given a list of several strings, the set() function will return a set of unique strings from the list. This makes sense if you think of it like a for loop. Take the first item from the list, put it in the set. Second. Third. Fourth. Fifth — wait, that's in the set already, so it only gets listed once, because Python sets don't allow duplicates. Sixth. Seventh — again, a duplicate, so it only gets listed once. The end result? All the unique items in the original list, without any duplicates. The original list doesn't even need to be sorted first.
2. The same technique works with strings, since a string is just a sequence of characters.
3. Given a list of strings, ".join(a_list) concatenates all the strings together into one.
4. So, given a list of strings, this line of code returns all the unique characters across all the strings, with no duplicates.

The alphametics solver uses this technique to build a set of all the unique characters in the puzzle.

```
unique_characters = set(''.join(words))
```

This list is later used to assign digits to characters as the solver iterates through the possible solutions.

Lecture 3.1

Making Assertions

Like many programming languages, Python has an assert statement. Here is how it works.

```
>>> assert 1 + 1 == 2                                          ①
>>> assert 1 + 1 == 3                                          ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 + 2 == 5, "Only for very large values of 2"       ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Only for very large values of 2
```

1. The assert statement is followed by any valid Python expression. In this case, the expression 1 + 1 == 2 evaluates to True, so the assert statement does nothing.
2. However, if the Python expression evaluates to False, the assert statement will raise an AssertionError.
3. You can also include a human-readable message that is printed if the AssertionError is raised.

Therefore, this line of code:

```
assert len(unique_characters) <= 10, 'Too many letters'
```

…is equivalent to this:

```
if len(unique_characters) > 10:
    raise AssertionError('Too many letters')
```

The alphametics solver uses this exact assert statement to bail out early if the puzzle contains more than ten unique letters. Since each letter is assigned a unique digit, and there are only ten digits, a puzzle with more than ten unique letters can not possibly have a solution.

Generator Expressions

A generator expression is like a generator function without the function.

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

```
>>> gen = (ord(c) for c in unique_characters)   ①
>>> gen                                         ②
<generator object <genexpr> at 0x00BADC10>
>>> next(gen)                                   ③
69
>>> next(gen)
68
>>> tuple(ord(c) for c in unique_characters)    ④
(69, 68, 77, 79, 78, 83, 82, 89)
```

1. A generator expression is like an anonymous function that yields values. The expression itself looks like a list comprehension, but it's wrapped in parentheses instead of square brackets.
2. The generator expression returns… an iterator.
3. Calling next(gen) returns the next value from the iterator.
4. If you like, you can iterate through all the possible values and return a tuple, list, or set, by passing the generator expression to tuple(), list(), or set(). In these cases, you don't need an extra set of parentheses — just pass the "bare" expression ord(c) for c in unique_characters to the tuple() function, and Python figures out that it's a generator expression.

Using a generator expression instead of a list comprehension can save both cpu and ram. If you're building an list just to throw it away (e.g. passing it to tuple() or set()), use a generator expression instead!

Here is another way to accomplish the same thing, using a generator function:

```
def ord_map(a_string):
    for c in a_string:
        yield ord(c)

gen = ord_map(unique_characters)
```

The generator expression is more compact but functionally equivalent.

Lecture 3.2
Calculating Permutations.

First, what are permutations? Permutations are a mathematical concept. (There are several definitions, depending on what kind of math you are doing. Here we are talking about combinatorics.)

The idea is that you take a list of things (could be numbers, could be letters, could be dancing bears) and find all the possible ways to split them up into smaller lists. All the smaller lists have the same size, which can be as small as 1 and as large as the total number of items. Oh, and nothing can be repeated. Mathematicians say things like "let's find the permutations of 3 different items taken 2 at a time," which means you have a sequence of 3 items, and you want to find all the possible ordered pairs.

```
>>> import itertools                                    ①
>>> perms = itertools.permutations([1, 2, 3], 2)       ②
>>> next(perms)                                         ③
(1, 2)
>>> next(perms)
(1, 3)
>>> next(perms)
(2, 1)                                                  ④
>>> next(perms)
(2, 3)
>>> next(perms)
(3, 1)
>>> next(perms)
(3, 2)
>>> next(perms)                                         ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

1. The itertools module has all kinds of fun stuff in it, including a permutations() function that does all the hard work of finding permutations.
2. The permutations() function takes a sequence (here a list of three integers) and a number, which is the number of items you want in each smaller group. The function returns an iterator, which you can use in a for loop or any old place that iterates. Here we will step through the iterator manually to show all the values.
3. The first permutation of [1, 2, 3] taken 2 at a time is (1, 2).
4. Note that permutations are ordered: (2, 1) is different than (1, 2).

5. That is it! Those are all the permutations of [1, 2, 3] taken 2 at a time. Pairs like (1, 1) and (2, 2) never show up, because they contain repeats so they aren't valid permutations. When there are no more permutations, the iterator raises a StopIteration exception.

The itertools module has all kinds of fun stuff.
The permutations() function doesn't have to take a list. It can take any sequence — even a string.

```
>>> import itertools
>>> perms = itertools.permutations('ABC', 3)  ①
>>> next(perms)
('A', 'B', 'C')                               ②
>>> next(perms)
('A', 'C', 'B')
>>> next(perms)
('B', 'A', 'C')
>>> next(perms)
('B', 'C', 'A')
>>> next(perms)
('C', 'A', 'B')
>>> next(perms)
('C', 'B', 'A')
>>> next(perms)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(itertools.permutations('ABC', 3))    ③
[('A', 'B', 'C'), ('A', 'C', 'B'),
 ('B', 'A', 'C'), ('B', 'C', 'A'),
 ('C', 'A', 'B'), ('C', 'B', 'A')]
```

1. A string is just a sequence of characters. For the purposes of finding permutations, the string 'ABC' is equivalent to the list ['A', 'B', 'C'].
2. The first permutation of the 3 items ['A', 'B', 'C'], taken 3 at a time, is ('A', 'B', 'C'). There are five other permutations — the same three characters in every conceivable order.
3. Since the permutations() function always returns an iterator, an easy way to debug permutations is to pass that iterator to the built-in list() function to see all the permutations immediately.

Other Fun stuff in the itertools Module

```
>>> import itertools
>>> list(itertools.product('ABC', '123'))   ①
[('A', '1'), ('A', '2'), ('A', '3'),
 ('B', '1'), ('B', '2'), ('B', '3'),
```

```
 ('C', '1'), ('C', '2'), ('C', '3')]
>>> list(itertools.combinations('ABC', 2))   ②
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

1. The itertools.product() function returns an iterator containing the Cartesian product of two sequences.
2. The itertools.combinations() function returns an iterator containing all the possible combinations of the given sequence of the given length. This is like the itertools.permutations() function, except combinations don't include items that are duplicates of other items in a different order. So itertools.permutations('ABC', 2) will return both ('A', 'B') and ('B', 'A') (among others), but itertools.combinations('ABC', 2) will not return ('B', 'A') because it is a duplicate of ('A', 'B') in a different order.

```
>>> names = list(open('examples/favorite-people.txt', encoding='utf-8'))   ①
>>> names
['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']
>>> names = [name.rstrip() for name in names]                              ②
>>> names
['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']
>>> names = sorted(names)                                                  ③
>>> names
['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']
>>> names = sorted(names, key=len)                                         ④
>>> names
['Alex', 'Anne', 'Dora', 'John', 'Mike',
'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

1. This idiom returns a list of the lines in a text file.
2. Unfortunately (for this example), the list(open(filename)) idiom also includes the carriage returns at the end of each line. This list comprehension uses the rstrip() string method to strip trailing whitespace from each line. (Strings also have an lstrip() method to strip leading whitespace, and a strip() method which strips both.)
3. The sorted() function takes a list and returns it sorted. By default, it sorts alphabetically.
4. But the sorted() function can also take a function as the key parameter, and it sorts by that key. In this case, the sort function is len(), so it sorts by len(each item). Shorter names come first, then longer, then longest.

What does this have to do with the itertools module?

```
…continuing from the previous interactive shell…
>>> import itertools
>>> groups = itertools.groupby(names, len)   ①
>>> groups
```

```
<itertools.groupby object at 0x00BB20C0>
>>> list(groups)
[(4, <itertools._grouper object at 0x00BA8BF0>),
 (5, <itertools._grouper object at 0x00BB4050>),
 (6, <itertools._grouper object at 0x00BB4030>)]
>>> groups = itertools.groupby(names, len)    ②
>>> for name_length, name_iter in groups:     ③
...     print('Names with {0:d} letters:'.format(name_length))
...     for name in name_iter:
...         print(name)
...
Names with 4 letters:
Alex
Anne
Dora
John
Mike
Names with 5 letters:
Chris
Ethan
Sarah
Names with 6 letters:
Lizzie
Wesley
```

1. The itertools.groupby() function takes a sequence and a key function, and returns an iterator that generates pairs. Each pair contains the result of key_function(each item) and another iterator containing all the items that shared that key result.
2. Calling the list() function “exhausted” the iterator, i.e. you’ve already generated every item in the iterator to make the list. There’s no “reset” button on an iterator; you can’t just start over once you’ve exhausted it. If you want to loop through it again (say, in the upcoming for loop), you need to call itertools.groupby() again to create a new iterator.
3. In this example, given a list of names already sorted by length, itertools.groupby(names, len) will put all the 4-letter names in one iterator, all the 5-letter names in another iterator, and so on. The groupby() function is completely generic; it could group strings by first letter, numbers by their number of factors, or any other key function you can think of.

The itertools.groupby() function only works if the input sequence is already sorted by the grouping function. In the example above, you grouped a list of names by the len() function. That only worked because the input list was already sorted by length.

Are you watching closely?

```
>>> list(range(0, 3))
[0, 1, 2]
```

```
>>> list(range(10, 13))
[10, 11, 12]
>>> list(itertools.chain(range(0, 3), range(10, 13)))          ①
[0, 1, 2, 10, 11, 12]
>>> list(zip(range(0, 3), range(10, 13)))                      ②
[(0, 10), (1, 11), (2, 12)]
>>> list(zip(range(0, 3), range(10, 14)))                      ③
[(0, 10), (1, 11), (2, 12)]
>>> list(itertools.zip_longest(range(0, 3), range(10, 14)))    ④
[(0, 10), (1, 11), (2, 12), (None, 13)]
```

1. The itertools.chain() function takes two iterators and returns an iterator that contains all the items from the first iterator, followed by all the items from the second iterator. (Actually, it can take any number of iterators, and it chains them all in the order they were passed to the function.)
2. The zip() function does something prosaic that turns out to be extremely useful: it takes any number of sequences and returns an iterator which returns tuples of the first items of each sequence, then the second items of each, then the third, and so on.
3. The zip() function stops at the end of the shortest sequence. range(10, 14) has 4 items (10, 11, 12, and 13), but range(0, 3) only has 3, so the zip() function returns an iterator of 3 items.
4. On the other hand, the itertools.zip_longest() function stops at the end of the longest sequence, inserting None values for items past the end of the shorter sequences.

OK, that was all very interesting, but how does it relate to the alphametics solver? Here is how:

```
>>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
>>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')
>>> tuple(zip(characters, guess))  ①
(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
 ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))
>>> dict(zip(characters, guess))   ②
{'E': '0', 'D': '3', 'M': '2', 'O': '4',
 'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

1. Given a list of letters and a list of digits (each represented here as 1-character strings), the zip function will create a pairing of letters and digits, in order.
2. Why is that cool? Because that data structure happens to be exactly the right structure to pass to the dict() function to create a dictionary that uses letters as keys and their associated digits as values. (This isn't the only way to do it, of course. You could use a dictionary comprehension to create the dictionary directly.) Although the printed representation of the dictionary lists the pairs in a different order (dictionaries have no "order" per se), you can see that each letter is associated with the digit, based on the ordering of the original characters and guess sequences.

The alphametics solver uses this technique to create a dictionary that maps letters in the puzzle to digits in the solution, for each possible solution.

```
characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
...
for guess in itertools.permutations(digits, len(characters)):
    ...
    equation = puzzle.translate(dict(zip(characters, guess)))
```

But what is this translate() method? Ah, now you are getting to the really fun part.

Lecture 4.1

A new kind of string Manipulation

Python strings have many methods. You learned about some of those methods in the Strings chapter: lower(), count(), and format(). Now I want to introduce you to a powerful but little-known string manipulation technique: the translate() method.

```
>>> translation_table = {ord('A'): ord('O')}  ①
>>> translation_table                         ②
{65: 79}
>>> 'MARK'.translate(translation_table)       ③
'MORK'
```

1. String translation starts with a translation table, which is just a dictionary that maps one character to another. Actually, “character” is incorrect — the translation table really maps one byte to another.
2. Remember, bytes in Python 3 are integers. The ord() function returns the ascii value of a character, which, in the case of A–Z, is always a byte from 65 to 90.
3. The translate() method on a string takes a translation table and runs the string through it. That is, it replaces all occurrences of the keys of the translation table with the corresponding values. In this case, “translating” MARK to MORK.

What does this have to do with solving alphametic puzzles? As it turns out, everything.

```
>>> characters = tuple(ord(c) for c in 'SMEDONRY')        ①
>>> characters
(83, 77, 69, 68, 79, 78, 82, 89)
>>> guess = tuple(ord(c) for c in '91570682')             ②
>>> guess
(57, 49, 53, 55, 48, 54, 56, 50)
>>> translation_table = dict(zip(characters, guess))      ③
>>> translation_table
{68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}
>>> 'SEND + MORE == MONEY'.translate(translation_table)   ④
'9567 + 1085 == 10652'
```

1. Using a generator expression, we quickly compute the byte values for each character in a string. characters is an example of the value of sorted_characters in the alphametics.solve() function.
2. Using another generator expression, we quickly compute the byte values for each digit in this string. The result, guess, is of the form returned by the itertools.permutations() function in the alphametics.solve() function.
3. This translation table is generated by zipping characters and guess together and building a dictionary from the resulting sequence of pairs. This is exactly what the alphametics.solve() function does inside the for loop.
4. Finally, we pass this translation table to the translate() method of the original puzzle string. This converts each letter in the string to the corresponding digit (based on the letters in characters and the digits in guess). The result is a valid Python expression, as a string.

That is pretty impressive. But what can you do with a string that happens to be a valid Python expression?

Lecture 4.2

Evaluating Arbitrary Strings as Python Expressions.

This is the final piece of the puzzle (or rather, the final piece of the puzzle solver). After all that fancy string manipulation, we're left with a string like '9567 + 1085 == 10652'. But that is a string, and what good is a string? Enter eval(), the universal Python evaluation tool.

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

But wait, there is more! The eval() function isn't limited to boolean expressions. It can handle any Python expression and returns any datatype.

```
>>> eval('"A" + "B"')
'AB'
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> eval('"AAAAA".count("A")')
5
>>> eval('["*"] * 5')
['*', '*', '*', '*', '*']
```

But wait, that is not all!

```
>>> x = 5
>>> eval("x * 5")          ①
25
>>> eval("pow(x, 2)")      ②
25
>>> import math
>>> eval("math.sqrt(x)")   ③
2.2360679774997898
```

1. The expression that eval() takes can reference global variables defined outside the eval(). If called within a function, it can reference local variables too.
2. And functions.
3. And modules.

Hey, wait a minute…

```
>>> import subprocess
>>> eval("subprocess.getoutput('ls ~')")                    ①
'Desktop         Library         Pictures \
 Documents       Movies          Public   \
 Music           Sites'
>>> eval("subprocess.getoutput('rm /some/random/file')")    ②
```

1. The subprocess module allows you to run arbitrary shell commands and get the result as a Python string.
2. Arbitrary shell commands can have permanent consequences.

It's even worse than that, because there's a global __import__() function that takes a module name as a string, imports the module, and returns a reference to it. Combined with the power of eval(), you can construct a single expression that will wipe out all your files:

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')")   ①
```

1. Now imagine the output of 'rm -rf ~'. Actually there wouldn't be any output, but you wouldn't have any files left either.

```
eval() is EVIL
```

Well, the evil part is evaluating arbitrary expressions from untrusted sources. You should only use eval() on trusted input. Of course, the trick is figuring out what's "trusted." But here is something I know for certain: you should NOT take this alphametics solver and put it on the internet as a fun little web service. Do not make the mistake of thinking, "Gosh, the function does a lot of string manipulation before getting a string to evaluate; I can't imagine how someone could exploit that." Someone WILL figure out how to sneak nasty executable code past all that string manipulation (stranger things have happened), and then you can kiss your server goodbye.

But surely there is some way to evaluate expressions safely? To put eval() in a sandbox where it can't access or harm the outside world? Well, yes and no.

```
>>> x = 5
>>> eval("x * 5", {}, {})                  ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
>>> eval("x * 5", {"x": x}, {})            ②
25
>>> import math
>>> eval("math.sqrt(x)", {"x": x}, {})     ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'math' is not defined
```

1. The second and third parameters passed to the eval() function act as the global and local namespaces for evaluating the expression. In this case, they are both empty, which means that when the string "x * 5" is evaluated, there is no reference to x in either the global or local namespace, so eval() throws an exception.
2. You can selectively include specific values in the global namespace by listing them individually. Then those — and only those — variables will be available during evaluation.
3. Even though you just imported the math module, you didn't include it in the namespace passed to the eval() function, so the evaluation failed.

Gee, that was easy. Lemme make an alphametics web service now!

```
>>> eval("pow(5, 2)", {}, {})                          ①
25
>>> eval("__import__('math').sqrt(5)", {}, {})         ②
2.2360679774997898
```

1. Even though you've passed empty dictionaries for the global and local namespaces, all of Python's built-in functions are still available during evaluation. So pow(5, 2) works, because 5 and 2 are literals, and pow() is a built-in function.
2. Unfortunately (and if you don't see why it's unfortunate, read on), the __import__() function is also a built-in function, so it works too.
3. Yeah, that means you can still do nasty things, even if you explicitly set the global and local namespaces to empty dictionaries when calling eval():

```
>>> eval("__import__('subprocess').getoutput('rm
/some/random/file')", {}, {})
```

Oops. I am glad I didn't make that alphametics web service. Is there any way to use eval() safely? Well, yes and no.

```
>>> eval("__import__('math').sqrt(5)",
...      {"__builtins__":None}, {})                    ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",
...      {"__builtins__":None}, {})                    ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

1. To evaluate untrusted expressions safely, you need to define a global namespace dictionary that maps "__builtins__" to None, the Python null value. Internally, the “built-in” functions are contained within a pseudo-module called "__builtins__". This pseudo-module (i.e. the set of built-in functions) is made available to evaluated expressions unless you explicitly override it.
2. Be sure you’ve overridden __builtins__. Not __builtin__, __built-ins__, or some other variation that will work just fine but expose you to catastrophic risks.
3. So eval() is safe now? Well, yes and no.

```
>>> eval("2 ** 2147483647",
...      {"__builtins__":None}, {})          ①
```

1. Even without access to __builtins__, you can still launch a denial-of-service attack. For example, trying to raise 2 to the 2147483647th power will spike your server’s cpu utilization to 100% for quite some time. (If you’re trying this in the interactive shell, press Ctrl-C a few times to break out of it.) Technically this expression will return a value eventually, but in the meantime your server will be doing a whole lot of nothing.

In the end, it is possible to safely evaluate untrusted Python expressions, for some definition of “safe” that turns out not to be terribly useful in real life. It’s fine if you’re just playing around, and it’s fine if you only ever pass it trusted input. But anything else is just asking for trouble.

Lecture 5.1

Reading from Text Files

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory.

Python has several functions for creating, reading, updating, and deleting files.

File operation takes place in the following order.

- Open a file
- Read or write (perform operation)
- Close the file

The open Function:

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details-

- ***file_name***: The file_name argument is a string value that contains the name of the file that you want to access.
- ***access_mode***: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
- ***buffering***: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

Here is a list of the different modes of opening a file-

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.

rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes

Once a file is opened and you have one file object, you can get various information related to that file.

Here is a list of all the attributes related to a file object-

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.

file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Example

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.close()
```

This produces the following result-

```
Name of the file: foo.txt
Closed or not : False
Opening mode : wb
```

The close() Method

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close();
```

Example

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
# Close opened file
fo.close()
```

This produces the following result-

```
Name of the file: foo.txt
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
```

```
    f = open("test.txt", encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

Closing the file automatically

The best way to close a file is by using the with statement. This ensures that the file is closed when the block inside the with statement is exited. We don't need to explicitly call the close() method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
    # perform file operations
```

The read() Method

The read() method reads a string from an open file. It is important to note that Python strings can have binary data apart from the text data.

Syntax

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

Example

Let us take a file foo.txt, which we created above.

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)
# Close opened file
fo.close()
```

This produces the following result-

```
Read String is : Python is
```

Lecture 5.2
Writing to text files.

The write() Method

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string-

Syntax

```
fileObject.write(string);
```

Here, passed parameter is the content to be written into the opened file.

Example

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")
# Close opend file
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content-

```
Python is a great language.
Yeah its great!!
```

File Positions

The tell() method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file foo.txt, which we created above.

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)
# Check current position
position = fo.tell()
print ("Current file position : ", position)
# Reposition pointer at the beginning once again
position = fo.seek(0, 0)
str = fo.read(10)
print ("Again read String is : ", str)
# Close opened file
fo.close()
```

This produces the following result-

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```

Lecture 6.1

Binary Files

Not all files contain text. Some of them contain pictures.

```
>>> an_image = open('examples/beauregard.jpg', mode='rb')          ①
>>> an_image.mode                                                  ②
'rb'
>>> an_image.name                                                  ③
'examples/beauregard.jpg'
>>> an_image.encoding                                              ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

1. Opening a file in binary mode is simple but subtle. The only difference from opening it in text mode is that the mode parameter contains a 'b' character.
2. The stream object you get from opening a file in binary mode has many of the same attributes, including mode, which reflects the mode parameter you passed into the open() function.
3. Binary stream objects also have a name attribute, just like text stream objects.
4. Here is one difference, though: a binary stream object has no encoding attribute. That makes sense, right? You are reading (or writing) bytes, not strings, so there is no conversion for Python to do. What you get out of a binary file is exactly what you put into it; no conversion necessary.

Did I mention you are reading bytes? Yes you are.

```
# continued from the previous example
>>> an_image.tell()
0
>>> data = an_image.read(3)  ①
>>> data
b'\xff\xd8\xff'
>>> type(data)               ②
<class 'bytes'>
```

```
>>> an_image.tell()          ③
3
>>> an_image.seek(0)
0
>>> data = an_image.read()
>>> len(data)
3150
```

1. Like text files, you can read binary files a little bit at a time. But there is a crucial difference…
2. …you are reading bytes, not strings. Since you opened the file in binary mode, the read() method takes the number of bytes to read, not the number of characters.
3. That means that there's never an unexpected mismatch between the number you passed into the read() method and the position index you get out of the tell() method. The read() method reads bytes, and the seek() and tell() methods track the number of bytes read. For binary files, they will always agree.

Stream Objects From Non-File Sources

Imagine you are writing a library, and one of your library functions is going to read some data from a file. The function could simply take a filename as a string, go open the file for reading, read it, and close it before exiting. But you should not do that. Instead, your api should take an arbitrary stream object.

In the simplest case, a stream object is anything with a read() method which takes an optional size parameter and returns a string. When called with no size parameter, the read() method should read everything there is to read from the input source and return all the data as a single value. When called with a size parameter, it reads that much from the input source and returns that much data. When called again, it picks up where it left off and returns the next chunk of data.

That sounds exactly like the stream object you get from opening a real file. The difference is that you are not limiting yourself to real files. The input source that is being "read" could be anything: a web page, a string in memory, even the output of another program. As long as your functions take a stream object and simply call the object's read() method, you can handle any input source that acts like a file, without specific code to handle each kind of input.

```
>>> a_string = 'PapayaWhip is the new black.'
>>> import io                                    ①
>>> a_file = io.StringIO(a_string)               ②
```

```
>>> a_file.read()                                    ③
'PapayaWhip is the new black.'
>>> a_file.read()                                    ④
''
>>> a_file.seek(0)                                   ⑤
0
>>> a_file.read(10)                                  ⑥
'PapayaWhip'
>>> a_file.tell()
10
>>> a_file.seek(18)
18
>>> a_file.read()
'new black.'
```

1. The io module defines the StringIO class that you can use to treat a string in memory as a file.
2. To create a stream object out of a string, create an instance of the io.StringIO() class and pass it the string you want to use as your "file" data. Now you have a stream object, and you can do all sorts of stream-like things with it.
3. Calling the read() method "reads" the entire "file," which in the case of a StringIO object simply returns the original string.
4. Just like a real file, calling the read() method again returns an empty string.
5. You can explicitly seek to the beginning of the string, just like seeking through a real file, by using the seek() method of the StringIO object.
6. You can also read the string in chunks, by passing a size parameter to the read() method.

io.StringIO lets you treat a string as a text file. There's also a io.BytesIO class, which lets you treat a byte array as a binary file.

Handling Compressed Files

The Python standard library contains modules that support reading and writing compressed files. There are a number of different compression schemes; the two most popular on non-Windows systems are gzip and bzip2. (You may have also encountered PKZIP archives and GNU Tar archives. Python has modules for those, too.)

The gzip module lets you create a stream object for reading or writing a gzip-compressed file. The stream object it gives you supports the read() method (if you opened it for reading) or the write() method (if you opened it for writing). That means you can use the

methods you have already learned for regular files to directly read or write a gzip-compressed file, without creating a temporary file to store the decompressed data.

As a bonus, it supports the with statement too, so you can let Python automatically close your gzip-compressed file when you're done with it.

```
you@localhost:~$ python3
>>> import gzip
>>> with gzip.open('out.log.gz', mode='wb') as z_file:          ①
...   z_file.write('A nine mile walk is no joke, especially in the
                   rain.'.encode('utf-8'))
...
>>> exit()

you@localhost:~$ ls -l out.log.gz                               ②
-rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
you@localhost:~$ gunzip out.log.gz                              ③
you@localhost:~$ cat out.log                                    ④
A nine mile walk is no joke, especially in the rain.
```

1. You should always open gzipped files in binary mode. (Note the 'b' character in the mode argument.)
2. I constructed this example on Linux. If you're not familiar with the command line, this command is showing the "long listing" of the gzip-compressed file you just created in the Python Shell. This listing shows that the file exists (good), and that it is 79 bytes long. That's actually larger than the string you started with! The gzip file format includes a fixed-length header that contains some metadata about the file, so it's inefficient for extremely small files.
3. The gunzip command (pronounced "gee-unzip") decompresses the file and stores the contents in a new file named the same as the compressed file but without the .gz file extension.
4. The cat command displays the contents of a file. This file contains the string you originally wrote directly to the compressed file out.log.gz from within the Python Shell.

Did you get this error?

```
>>> with gzip.open('out.log.gz', mode='wb') as z_file:
...         z_file.write('A nine mile walk is no joke, especially in
the rain.'.encode('utf-8'))
```

```
...
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'GzipFile' object has no attribute '__exit__'
```

If so, you are probably using Python 3.0. You should really upgrade to Python 3.1.

Python 3.0 had a gzip module, but it did not support using a gzipped-file object as a context manager. Python 3.1 added the ability to use gzipped-file objects in a with statement.

Lecture 6.2
Standard Input, Output, and Error.

Command-line gurus are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated stdout and stderr) are pipes that are built into every unix-like system, including Mac OS X and Linux. When you call the print() function, the thing you're printing is sent to the stdout pipe. When your program crashes and prints out a traceback, it goes to the stderr pipe. By default, both of these pipes are just connected to the terminal window where you are working; when your program prints something, you see the output in your terminal window, and when a program crashes, you see the traceback in your terminal window too. In the graphical Python Shell, the stdout and stderr pipes default to your "Interactive Window".

```
>>> for i in range(3):
...     print('PapayaWhip')                    ①
PapayaWhip
PapayaWhip
PapayaWhip
>>> import sys
>>> for i in range(3):
...     l = sys.stdout.write('is the')         ②
is theis theis the
>>> for i in range(3):
...     l = sys.stderr.write('new black')      ③
new blacknew blacknew black
```

1. The print() function, in a loop. Nothing surprising here.
2. stdout is defined in the sys module, and it is a stream object. Calling its write() function will print out whatever string you give it, then return the length of the output. In fact, this is what the print function really does; it adds a carriage return to the end of the string you're printing, and calls sys.stdout.write.
3. In the simplest case, sys.stdout and sys.stderr send their output to the same place: the Python ide (if you're in one), or the terminal (if you're running Python from the command line). Like standard output, standard error does not add carriage returns for you. If you want carriage returns, you'll need to write carriage return characters.

sys.stdout and sys.stderr are stream objects, but they are write-only. Attempting to call their read() method will always raise an IOError.

```
>>> import sys
>>> sys.stdout.read()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: not readable
```

Redirecting Standard Output

sys.stdout and sys.stderr are stream objects, albeit ones that only support writing. But they are not constants; they're variables. That means you can assign them a new value — any other stream object — to redirect their output.

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old


print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file,
RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Check this out:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

Did you get this error?

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
  File "stdout.py", line 15
```

```
    with open('out.log', mode='w', encoding='utf-8') as a_file,
RedirectStdoutTo(a_file):
SyntaxError: invalid syntax
```

If so, you're probably using Python 3.0. You should really upgrade to Python 3.1.

Python 3.0 supported the with statement, but each statement can only use one context manager. Python 3.1 allows you to chain multiple context managers in a single with statement.

Let us take the last part first.

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file,
RedirectStdoutTo(a_file):
    print('B')
print('C')
```

That is a complicated with statement. Let me rewrite it as something more recognizable.

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

As the rewrite shows, you actually have two with statements, one nested within the scope of the other. The "outer" with statement should be familiar by now: it opens a utf-8-encoded text file named out.log for writing and assigns the stream object to a variable named a_file. But that's not the only thing odd here.

```
with RedirectStdoutTo(a_file):
```

Where is the as clause? The with statement does not actually require one. Just like you can call a function and ignore its return value, you can have a with statement that does not assign them with context to a variable. In this case, you are only interested in the side effects of the RedirectStdoutTo context.

What are those side effects? Look inside the RedirectStdoutTo class. This class is a custom context manager. Any class can be a context manager by defining two special methods: __enter__() and __exit__().

```
class RedirectStdoutTo:
    def __init__(self, out_new):     ①
        self.out_new = out_new

    def __enter__(self):             ②
        self.out_old = sys.stdout
        sys.stdout = self.out_new
```

```
    def __exit__(self, *args):          ③
        sys.stdout = self.out_old
```

1. The __init__() method is called immediately after an instance is created. It takes one parameter, the stream object that you want to use as standard output for the life of the context. This method just saves the stream object in an instance variable so other methods can use it later.
2. The __enter__() method is a special class method; Python calls it when entering a context (i.e. at the beginning of the with statement). This method saves the current value of sys.stdout in self.out_old, then redirects standard output by assigning self.out_new to sys.stdout.
3. The __exit__() method is another special class method; Python calls it when exiting the context (i.e. at the end of the with statement). This method restores standard output to its original value by assigning the saved self.out_old value to sys.stdout.

Putting it all together:

```
print('A')                                                    ①
with open('out.log', mode='w', encoding='utf-8') as a_file,
RedirectStdoutTo(a_file):                                     ②
    print('B')                                                ③
print('C')                                                    ④
```

1. This will print to the ide “Interactive Window” (or the terminal, if running the script from the command line).
2. This with statement takes a comma-separated list of contexts. The comma-separated list acts like a series of nested with blocks. The first context listed is the “outer” block; the last one listed is the “inner” block. The first context opens a file; the second context redirects sys.stdout to the stream object that was created in the first context.
3. Because this print() function is executed with the context created by the with statement, it will not print to the screen; it will write to the file out.log.
4. The with code block is over. Python has told each context manager to do whatever it is they do upon exiting a context. The context managers form a last-in-first-out stack. Upon exiting, the second context changed sys.stdout back to its original value, then the first context closed the file named out.log. Since standard output has been restored to its original value, calling the print() function will once again print to the screen.

Redirecting standard error works exactly the same way, using sys.stderr instead of sys.stdout.

The write() Method

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string-

Syntax

```
fileObject.write(string);
```

Here, passed parameter is the content to be written into the opened file.

Example

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")
# Close opend file
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content-

```
Python is a great language.
Yeah its great!!
```

File Positions

The tell() method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file foo.txt, which we created above.

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
```

```
print ("Read String is : ", str)
# Check current position
position = fo.tell()
print ("Current file position : ", position)
# Reposition pointer at the beginning once again
position = fo.seek(0, 0)
str = fo.read(10)
print ("Again read String is : ", str)
# Close opened file
fo.close()
```

This produces the following result-

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```