Reading information from files is a common and important operation in Python. In
this lecture we will discuss various options for reading files, and characterize
them based on simplicity and efficiency (mostly with regards to efficiency in
the use of space/memory, which is important when reading very large files).

In a nutshell, you should avoid the use of the .read() and .readlines() methods
and instead iterate over files (using a standard for loop or a for loop in a
comprehension).

The last section defines the parse_line function (available in the goody module)
and how to use it, making reading files that contain multi-type records (fields
of values) easy. One parameter binds to a tuple/list of function objects, each
specifying how to convert the field in its position (a substring of each line)
into a value of the appropriate type.

Simple and Space Efficient Code

To start, let's suppose we are reading a file where each line in the file is
just a string of text. The simplest and most efficient way to read such a file
is iterating over an "open" (file) object.

```
  for line in open(file_name):  # where file_name is a string naming a file
      process(line)             # where process is a function or some code
block
```

Typically, we need to strip off the newline character(s) at the end of each line
as it is read from the file, before it is processed further. We can strip off
this information by calling .rstrip('\n'). The code would become

```
for line in open(file_name):
    process(line.rstrip('\n'))
```

or

```
for line in open(file_name):
    line = line.rstrip('\n')
    process(line)
```

  Technically the os module binds the name linesep to the character(s) forming
  a newline on that operating system. For PCs os.linesep is '\r\n'; on Macs it
  is '\n'. So, we could call .rstrip(os.linesep) on the line. But when we read
  a line from a text file in Python, it converts any newline character(s) to

just the single character '\n', regardless of the operating system. This makes it easier to write Python code that works on all operating systems.

Note that if we call .rstrip() with no arguments, all white-space characters (including the newline character(s), spaces, and tabs) are stripped. We can use this simpler alternative if we don't meaningfully process whitespace at the ends of lines, but if we need to preserve it, we must call rstrip('\n').

Recall that there are NO MUTATOR methods on strings: so, when we call the line.rstrip(...) method, it does not mutate the object associated with line, but instead it produces a new string that has the same contents except with the requested character(s) stripped off its right end; we pass this new string as an argument to the process function, or rebind line to that new string.

So, the left code is correctly by the right code is INCORRECT

```
  for line in open(file_name):
      line = line.rstrip('\n')
      process(line)
```

```
for line in open(file_name):
    line.rstrip('\n') # does NOTHING!
    process(line)
```

The for loops in the code fragments above are all space efficient, because at any time Python stores only one line of the file in memory (although the file itself is likely cached/stored in a memory buffer). Note that if we wanted to create a list of lines read from the file (where the newline character(s) are removed from each string), we can write the following simple comprehension.

```
line_list = [line.rstrip('\n') for line in open(file_name)]
```

Note that in this example, line_list occupies about the same amount of space as the file: all lines are stored in memory (one per list entry) at the same time.
If we can process lines independently of each other (process each one without know the previous or subsequent lines) we should NOT write code to store a list of all the lines, because it is not space efficient.

we can use open as a context manager in a with statement, which handles file exceptions and automatically closes the file when the context manager finishes (which is often useful/important, but not always). With the open context manager, we would write the above code fragments as

```
with open(file_name) as open_file:
    for line in open_file:
        process(line.rstrip('\n'))
```

or

```
with open(file_name) as open_file:
    line_list = [line.rstrip('\n') for line in open(open_file)]
```

Using context managers does not change the space efficiency of the file reading.

Back To Top

The .readlines() and .read() methods

We can apply the .readlines() method to an open file: it produces a list of all
the lines in the file, where each line still ends in the newline character '\n'.

So, if open_file refers to the following open file,

```
Line 1
Line 2
Line 3
```

calling open_file.readlines() returns the list

```
['Line 1\n', 'Line 2\n', Line 3\n']
```

If we wanted to call .readlines() and process every string in the file (without
the '\n' characters at the end) we would write

```
for line in open(file_name).readlines():
    process(line.rstrip('\n'))
```

Notice that this code is longer than the loop given in the previous section, and is less space efficient, because it first computes a list of all the lines
in the file, storing it in memory at one time, and then it iterates over the strings in that list; the loop in the previous section stores in memory only one line at a time of the file, while it process that line.

If we wanted to create a list of lines without the '\n' characters at the end,
we could write

```
line_list = [line.rstrip('\n') for line in open(file_name).readlines()]
```

which is similar to but also more complicated than the code in the previous section.

Finally, if we wanted to compute a list of lines WITH the '\n' characters at the end, then calling

```
line_list = open(file_name).readlines()
```

which is simpler than the comprehension code below

```
line_list = [line for line in open(file_name)]
```

and does occupy the equivalent amount of storage. So this is one of the few places (and not really common) where calling the .readlines() method is useful.

We can apply the .read() method to an open file: it produces one giant string that contains all the lines in the file, each ended by the newline character '\n'.

So, if open_file refers to the following open file,

```
Line 1
Line 2
Line 3
```

calling open_file.read() returns the string

```
'Line 1\nLine 2\nLine 3\n'
```

We can split this string into a list of strings by calling the .split method.

```
line_list = open(file_name).read().split('\n')
```

and this code is simpler than what we have seen before, which is equivalent to

```
line_list = [line.rstrip('\n') for line in open(file_name)]
```

The .read() code above is a bit less space efficient than the comprehension, because it stores both the entire file string and a list of all the lines in the file at the same time; the comprehension stores the list of all the lines in the file, but not a string whose contents is the entire file itself.

Likewise, if we wanted to process every string in the file (without the '\n' characters at the ends) we can write

```
for line in open(file_name).read().split('\n'):
    process(line)
```

The for loop code is more complicated than the loop in the previous section, and
it uses space much less efficiently: this code stores the entire file (and a list of lines in the file) in memory at one time; the loop in the previous section stores in memory only one line at a time of the file. Not that because
of the call to split after read, there is no need for a call to split inside the
loop.

Bottom Line:

There is little to be gained by reading files by calling the .readline() and the
.read() method. Iterate directly over the "open" file with a standard for loop
or a for loop in a comprehension.

Back To Top

Reading Files and Parsing

Some text files contain lines that store other types or mixed-types of information. Suppose that we wanted to read a text file that stored strings representing numbers (one number per line). We can easily rewrite our original
code to the following, calling the int conversion function on each rstripped line.

```
for line in open(file_name):
    process( int(line.rstrip('\n')) )
```

Here we are assuming process takes an integer value as an argument.

In some files each line is a "record": a fixed number of fields of values, with
possibly different types, separated by some special character (often a space or
punctuation character like a comma or colon). To process each record in a file,
we must

```
  (1) read its line
  (2) separate its fields of values (still each value is a string)
  (3) call a conversion function for each string to get its value
```

The goody module contains the parse_lines function that easily supports reading records from files
(similarly to how lines are read from "open" files).

```
def parse_lines(open_file, sep, conversions):
    for line in open_file:
        yield [conv(item) for conv,item in
                  zip(conversions,line.strip('\n').split(sep))]
```

Here sep is a special character used to separate the fields in the record; conversions is a tuple (or list) of function objects: they are applied in sequence to the string values of the separated fields. When we iterate over a call to parse_lines (simlar to iterating over a call to "open"), the index variable is bound to a list of the values of the fields in the record.

For example, the following file contains fields of a name (str) followed by two
test scores (ints) all separated by commas.

```
Bob Smith,75,80
Mary Jones,85,90
```

We could read this file and print out the names of each student and their average test score by

```
for fields in parse_lines( open(file_name), ',' , (str,int,int) ):
```

```
  print(fields[0], (fields[1]+fields[2])/2)
```

Here fields is repeatedly bound to a 3-list containing a name (str) followed by two test scores (ints). fields is first bound to ['Bob Smith', 75, 80] and then to ['Mary Jones', 85, 90].

---

1) Note that if we specified conversions as (str,int) it would return the 2-lists ['Bob Smith', 75] followed by ['Mary Jones', 85] (because looping over
a zip stops when one of its arguments runs out of values: here each line contains more field values than conversion functions). Accessing fields[2] in the code above would raise an IndexError exception.

2) Likewise (because looping over a zip stops when one of its arguments runs out of values), if the a line contains a name and 3 integer values, only the name and first two integers would be returned in the 3-list: the line

```
Paul White,80,75,85
```

returns only the 3-list ['Paul White', 80, 75]

So parse_lines would not raise any exceptions in the code above.

We could define a more complicated parse_lines function that checked and immediately raised an exception if the number of separated field values in a record was not equal to the length of the tuple of conversion functions.

A simpler way to write such code is to use multiple index variables and unpacking (as we do when we write: for k,v in adict.items()).

```
for name,test1,test2 in parse_lines(open(file_name),',',(str,int,int)):
  print(name, (test1+test2)/2)
```

With this for loop, the first error noted above would also raise an exception because there would not be three values to unpack into name, test1, and test2;
the second error would again go unnoticed.

Finally, note that besides using the standard conversion function(s) like str and int, we can define our own conversion function(s). For example, suppose that
each record in the file specified a string name, some number of
int quiz results separated by colons, and an int final exam, with these three fields (name, quizzes, final) separated by commas. Such a file might look like

```
Bob Smith,75:80,90
Mary Jones,85:90:77,85
```

Here Bob took two quizzes but Mary took three. We could define

```
def quiz_list(scores):
    return [int(q) for q in scores.split(':')]
```

and then write

```
for name,quizzes,final in
parse_lines(open(file_name),',',(str,quiz_list,int)):
  print(name, sum(quizzes)/len(quizzes), final)
```

which would print

```
Bob Smith 77.5 90
Mary Jones 84.0 85
```

Note that 77.5 is (75+90)/2 and 84 is (85+90+77)/3. If we instead wrote

```
for fields in parse_lines(open(file_name),',',(str,quiz_list,int)):
  print(fields)
```

it would print the 3-lists

```
['Bob Smith', [75, 80], 90]
['Mary Jones', [85, 90, 77], 85]
```

Of course, we can also use lambdas instead of named functions; below we have substituted a lambda for the quiz_list function.

```
for fields in parse_lines(open(file_name),',',
                          (str,
                           lambda scores : [int(q) for q in
scores.split(':')],
                           int)):
  print(fields)
```

Back To Top

Problems

1) Suppose that we want to process the lower case version of every word on every line (where the words on a line are separated by colons) in a file named
file.txt. Which of the following code fragments correctly does so? For those that don't, explain why they fail. For example, if the file contained the three
lines:

```
  See spot
  See spot run
  Run spot run
```

it should process the following words in the following order:
'see', 'spot', 'see', 'spot', 'run', 'run', 'spot', 'run'.

```
for line in open('file.txt'):
    for word in line.rstrip().lower().split():
        process(word)
```

```
for line in open('file.txt'):
```

```
    for word in line.rstrip().split().lower():
        process(word)

for line in open('file.txt'):
    for word in line.lower().rstrip().split():
        process(word)

for line in open('file.txt'):
    for word in line.lower().split().rstrip():
        process(word)

for line in open('file.txt'):
    for word in line.split().rstrip().lower():
        process(word)

for line in open('file.txt'):
    for word in line.split().lower().rstrip():
        process(word)

for line in open('file.txt').read().lower().split('\n'):
    process(word)

for line in open('file.txt').read().split('\n').lower():
    process(word)
```

Overview

This lecture reviews basic material that you should know about defining and using classes, although it also presents some new (hopefully easy to understand)
material that you may have not seen. Primarily this lecture discusses how to use the namespaces in class objects and their instance objects to store data as
well as functions/methods.

Defining Class

When we define a class in a Python, we are binding the class name to an object
representing the entire class. We call the class object to create a new object
that is an instance of the class: Python constructs an empty instance object and then calls the ___init__ method defined in the class, (passing the new/empty
instance object to the self parameter) to initialize the state of the instance
object. Recall that all names in Python refer/are-bound to objects; so defining

```
class C:
    ...
```

creates the name C and binds it to an object representing the class C.

What names are attributes defined in a class object's namespace? I'm not talking about the instance objects that will be constructed from the class C, but the names in the namespace of the class object itself. Mostly, a class defines names that are bound to its methods (__init__, etc.), but later in this
lecture we will discuss names representing class variables as well as class methods.

When we want to construct an object that is an instance of a class, we refer to
a class object (typically by a name bound to it) followed by () and possibly arguments in these parentheses. Python does three things when constructing new
instance objects:

(1) It calls a special function that creates an object that is a new instance of
    the class. Note that this object automatically has an dictionary associated
     with it, with the name __dict__, but that dictionary starts empty.

(2) It calls the __init__ method for the class, passing the object created in
    (1) to the first/self parameter of __init__ , and following this with all

```
    the other argument values in the parentheses used in the call to
construct
    the state of this instance. Typically it assigns values to self/instance
    variables, storing the name/binding in __dict__ for the self object.

(3) A reference to the object that was created in (1) and initialized in (2)
is
    returned as the result of constructing the object: most likely this
    reference will be bound to a name: e.g., c = C(...) which means c refers
to
    a newly constructed/initialized object of class C.

So if we call c = C(1,2) Python calls C.__init__(reference to new object,1,2)
and binds c to refer to the newly constructed object somehow initialized by
1 and 2.

Note that we can define other names that can bind to/share the same class
object. For example:

class C:
    def __init__(self):
        print('instance of C object initialized')
        self.m = 'starting' # initialize an instance/attribute name

D = C   # C and D share the same class object
x = C() # Use C to construct an instance of a class C object
y = D() # Use D to construct an instance of a class C object

print(C,D,x,y)
print(type(x), type(y), type(C), type(type(x)))

Running this script produces the following: the first two lines from calling
__init__ twice, the next two from calling the two print statements above

  instance of C object initialized
  instance of C object initialized
    <__main__.C object at 0x027B0E10> <__main__.C object at 0x02889C50>


Finally, if we print(x.m) or print(y.m) it prints 'starting'

We can use the type function to determine the type of any object. The objects
x
and y refer to are instances of the class C, defined in the main script. C
(and
all classes that we define) are instances of a special class called 'type').
So
x is an instance of C, and C is an instance of 'type'.
```

Back To Top

Manipulating an object's namespace (and __init__):

All objects have namespaces, which are dictionaries in which names defined in that object are bound-to/associated-with values. Typically we write something like:

```
  self.name = value
```

in the __init__ or any other method, to defined a name in the dictionary of the
object self refers to, and bind that name (or update an exsting binding) to refer to value.

Now we illustrate a cruder way to add names to the namespace of an object. This way is not recommended now, but it furthers our understanding of objects and their namespaces in Python, and we will find concrete uses for this understanding later. Given class C defined above, we can write

```
c = C()
print(c.__dict__)
c.x = 1
c.y = 'two'
c.__dict__['z'] = 3
print(c.__dict__)
print(c.m, c.x, c.y, c.z)
```

Running this script produces

```
  instance of C object initialized
  {'m': 'starting'}
  {'x': 1, 'y': 'two', 'm': 'starting', 'z': 3}
  starting 1 two 3
```

So, we have used two different forms for adding three names to the namespace/dictionary for the object that c refers to (which initially stores the name m, initialized by self.m = 'starting'). Adding names updates the object's __dict__. Writing object.name = value is equivalent to writing object.__dict__['name'] = value.

The object c has a __dict__ attribute that stores its namespace bindings. Each identifier for which we define an attribute for the object appears as a string keyword in __dict__.

Generally we don't initialize the namespace of an object this way; instead we use the automatically-called __init__ method and its self parameter to do the initialization. But really, the same thing is happening in __init__ below as was shown above.

```
class C:
    def __init__(self,init_x,init_y):
        print('instance of C object initialized')
```

```
        self.x = init_x               # value depends on the argument
matching init_x
        self.y = init_y               # value depends on the argument
matching init_y
        self.z = 3            # always the same object: 3

c = C(1,'two')
print(c.__dict__)
```

Running this script produces the same results.

```
  instance of C object initialized
  {'z': 3, 'y': 'two', 'x': 1}
```

The purpose of the __init__ method is to create all the attribute names needed
by the object's methods and initializes them appropriately. Typically, once __init__ is called, no new names are added to the object's namespace (although
Python allows additions, as illustrated in the prior section (c.x = 1) and some
useful examples are illustrated later in this lecture as well). Every object constructed is likely to need exactly the same names: all the names used in the methods defined in the class; methods that process instance objects of the class. The __init__ method, which is automatically called by Python when an object is constructed, is a convenient place to localize the creation and initialization of all these attribute names.

So, for every assignment statement

```
  self.name = value
```

Python puts an entry into the object's namespace (self.__dict_) with the key 'name' (keys are strings) associated with value. We can do this in the __init__
method or after the object is constructed: both ways are shown above. When self.name appears in an expression (e.g., a = self.name), Python substitutes the
expression self.__dict__['name'] for the right hand side of the =, to retrieve
the value of that name from the self object's namespace/dictionary.

If we try to access a non-existent attribute name (c.mumble in the class C above, which is translated into c.__dict__['mumble']), Python raises an exception. print(c.mumble) prints the exception as

```
  AttributeError: 'C' object has no attribute 'mumble'
```

Note that some names defined in __init__ (z above) always receive the same initialization, so we don't need a parameter to initialize them. But often names need to be initialized to different values when different objects are constructed, so typically we add just enough parameters to the __init__ method
to allow us to specify how those names with different values should be initialized.

Interlude: assert in initialization

Sometimes __init__ will ensure that a parameter is matched to an argument that
stores a legal and reasonable value for it; if not, Python will raise an exception to indicate that the object being constructed cannot be properly initialized. Sometimes it raises an exception explicitly, using an if statement
that tests for an illegal value. Sometimes it uses an assert statement for this
purpose. Remember that the form of assert is:

```
  assert boolean-test, string
```

which is equivalent to the slightly more verbose

```
  if not boolean-test:
      raise AssertionError(string)
```

I suggest that the string argument to AssertionError should always contain 4 parts:

(1) The name of the class (if the problem occurs in the method of a class) or
the name of the module (if the problem occurs in a function in a module)

(2) The name of the method/function that detects the problem (here __init__)

(3) A short description of the problem...

(4) ...including the values of the argument(s) that is causing the problem

For example, if class C included an __init__ method that required x's argument
to be a positive integer, we could write __init__ as follows. We typically check
all the arguments FIRST in this method, before binding any self/instance names.

```
  def __init__(self,x):
      assert type(x) is int and x > 0, 'C.__init__: x('+str(x)+') must be an int and positive'
      ...
```

or

```
  def __init__(self,x):
      assert type(x) is int and x > 0, 'C.__init__: x({v}) must be an int and positive'.format(v=x)
      ...
```

If so, calling C(-1) would result in the error message

```
   C.__init__: x(-1) must be an int and positive
```

and calling C('abc') would result in the error message

```
   C.__init__: x(abc) must be an int and positive
```

Such a message provides useful information to whoever is writing/debugging the
program. In a well-written program, someone just using the program (possible not a programer) should not have to read/interpret such a message.

Question: if we wrote the assert as just x > 0, what exception would be raised
by calling C('abc') and why? Why is the given assertion "better", even though it
is less efficient to check?

We could be even more descriptive and write

```
  def __init__(self,x):
      assert type(x) is int, 'C.__init__: x('{v}') must be an
int'.format(v=x)
      assert x > 0, 'C.__init__: x({v}) must be positive'.format(v=x)
```

Note that int refers to the int class (not an instance of an int class: e.g., not 1), so writing "type(x) is int" is checking whether the type of the object
x refers to is the same as the object int refers to: meaning x is bound to an object constructed from the int class.

Once an object is constructed and initialized, typically we use it by calling methods (or possibly passing it to another function/method that calls its methods). We call an object's method using the syntax
object.method(arguments);
recall the Fundamental Equation of Object-Oriented Programming.

```
    o.m(...) is translated into type(o).m(o,...)
```

This means, the method m in the class specifying the type of object o is called,
and the object o used to call the method is passed to the method as the first argument (matching the self parameter). For example, 'a;b;c'.split(';') is translated into type('a;b;c').split('a;b;c',';') which is equivalent to str.split('a;b;c',';'). It calls the split method defined in the str class with
the arguments 'a;b;c' (matching the self parameter) and ';' (matching the second parameter).

Before finishing the discussion of objects and their dictionaries, recall that
C refers to a class object. As an object, it also has a __dict__ that stores its
namespace. Here is some code that shows what names are defined in the C object.

```
for (k,v) in C.__dict__.items():
    print(k,'->',v)
```

And here is what it prints.

```
__weakref__ ->
__dict__ ->
__doc__ -> None
__init__ ->
__qualname__ -> C
__module__ -> __main__
```

Note that __init__ is the only function that we defined, and it is there (on the 4th line). Because I ran this code as a script (the main module), its __module__ variable is bound to '__main__', which you should know something about, because you should have written (and understand) code like

```
  if __module__ == '__main__':
      ...
```

If this module were imported, the __module__ key would be associated with the file/module name it was written in, when imported. Only the module corresponding to the script that started execution has its __module__ key bound
to '__main__'. We can test this name to run the script only when its module is
run, not when it is imported by another module. The if accomplishes this behavior.

If I had defined C with a docstring it would appear as an attribute of __doc__.
Don't worry about __weakref__, the internal __dict__ or __qualname__.

Back To Top

Different kinds of names: definition and use

Let's discuss four different kinds of names in relation to classes. We will call all these names variables.

  (1) local variables: defined and used inside functions/methods to help with the computation; parameter variables are considered local variables too.

  (2) instance variables of an object: typically defined inside __init__ and used inside class methods (we saw other ways to define them above too). These are referred to as self.name.

  (3) class variables: typically defined in classes (at same level as methods,
      not inside methods) and typically used in methods; we use a class variable for information COMMON to all objects of a class (rather than putting the same local variable in each object constructed from the

class). Methods are actually class variables, bound to function objects.

All class variables are defined in the class object and they are found by the Fundamental Equation of Object-Oriented Programming through instances of that class. That is, if class C defines an attribute a (method or class variable) and x refers to an object constructed from class C, then x.a will find attribut a in class C, but only if it is not stored directly in x (in x's __dict__). For class variables, that is typically what we want.

(4) global variables: typically defined in modules (outside of functions and classes) and used inside functions and/or class methods; we typically avoid using them (don't use global variables), and when we do use them, we do so in a cautious and limited manner.

You should know how to use all these kinds of variables (and their semantics).
Use local variables and instance variables as needed (most function/methods have the former, and most classes define the later in __init__ and use them in methods). Class variables are sometimes useful to solve certain kinds of problems where common information is stored among all the instances, by storing them just once in their common class object. Global variables are fine to use in scripts, but are often frowned upon when declared in modules that are imported (although they too have their uses there, but in more advanced settings).

The following script uses each kind of variable, appropriately named. Ensure that you understand how each use of these variables works. The use of the command named 'global' (see the two lines with #comments) is explained in more detail below.

```
global_var = 0

class C:

    class_var = 0

    def __init__(self,init_instance_var):
        self.instance_var = init_instance_var

    def bump(self,name):
        print(name,'bumped')
        #global_var = 100    # comment out this line or the next
        global global_var    # comment out this line or the previous
        global_var += 1
        C.class_var += 1     # self.class_var get translated to C.class_var
        self.instance_var += 1
```

```
    def report(self,var_name):
        print('instance referred to by', var_name,
              ': global_var =', global_var,
              '/class_var =', self.class_var,     # can write as
self._class_var
              '/instance_var=', self.instance_var)

x=C(10)
x.report('x')
x.bump('x')
x.report('x')
print()

prints
  instance referred to by x : global_var = 0 /class_var = 0 /instance_var= 10
  x bumped
  instance referred to by x : global_var = 1 /class_var = 1 /instance_var= 11

print('y = x')
y = x
y.bump('y')
x.report('x')
y.report('y')
print()

prints
  y = x
  y bumped
  instance referred to by x : global_var = 2 /class_var = 2 /instance_var= 12
  instance referred to by y : global_var = 2 /class_var = 2 /instance_var= 12

print('y = C(20)')
y=C(20)
y.bump('y')
y.report('y')
print()

prints
  y = C(20)
  y bumped
  instance referred to by y : global_var = 3 /class_var = 3 /instance_var= 21

C.report(x,'x')       # same as x.report('x') by the Fundamental Equation of
OOP
type(x).report(x,'x') # ditto: the meaning of the Fundamental Equation of OOP
print()

prints
  instance referred to by x : global_var = 3 /class_var = 3 /instance_var= 12
  instance referred to by x : global_var = 3 /class_var = 3 /instance_var= 12


print(C.class_var, x.class_var)  # discussed below
print(x.instance_var)

prints
  3 3
```

```
  12
```

So, the global variable is changing every time, as is the class variable, because there is just one of each. But each object thta is an instance of C has its own instance variable, which changes only when bump is called on that instance.

If we instead commented as follows

```
        global_var = 100     # comment out this line or the next
        #global global_var   # comment out this line or the previous
```

running the script would have the following result: By removing the statement global global_var, then the statement global_var = 100 actually defines a local
variable in the bump method -despite its name- so its increment does not affect
the true global_var, which stays at zero.

Recall, if a variable defined in a function/method has not been declared global,
it is created as a local variable inside the function/procedure.

Note that one can REFER to the value of a global_var inside methods of class C
WITHOUT a global declaration (see the report method), but if a method wants to
CHANGE global_var it must declare it global (then all reference and changes are
to the real global variable). With no global global_var, the assignment global_var = 100 creates a new name local to the bump method and always binds it to 100.

```
  instance referred to by x : global_var = 0 /class_var = 0 /instance_var= 10
  x bumped
  instance referred to by x : global_var = 0 /class_var = 1 /instance_var= 11

  y = x
  y bumped
  instance referred to by x : global_var = 0 /class_var = 2 /instance_var= 12
  instance referred to by y : global_var = 0 /class_var = 2 /instance_var= 12

  y = C(20)
  y bumped
  instance referred to by y : global_var = 0 /class_var = 3 /instance_var= 21
  instance referred to by x : global_var = 0 /class_var = 3 /instance_var= 12

  1 1
  11
```

Finally, it is clear what C.class_var and x.instance_var refer to, but what about x.class_var? As shown above this prints 1 just as C.class_var does. This
meaning is a result of the Fundamental Equation of Object Oriented Programming
(but applied to variable attributes, not method attributes).

Technically, when specifying o.attr, any access to an attribute name in object o
(whether a variable or method) Python first tries to find attr in the object o;
if it is not defined in o's namespace/__dict__ Python uses the FEOOP to try to
find it by checking type(o).attr, which attempts to find attr in type(o)'s namespace/__dict__. So when trying to find x.class_var if fails to find class_var in o's namespace/__dict__, so it tries type(o).class_var or C.class_var and finds the class_var attribute in C's namespace/__dict__.

When we study inheritance, we will learn more about how Python searches for all attributes by generalizing this rule: if it is not in an instance, then it
tries in the class that instance was constructed from, and if not in that class, it tries in its super classes, and if not in its superclass....

But for now remember: to find an attribute, look first in the namespace of the
object; if it isn't there, then look in the namespace of the class thta the object was constructed from.

Back To Top

Strange Python (but now understandable)

---

1) Defining/using a method for a class, AFTER the class has been declared:

2) Defining a method for an instance (but not the whole class) after the instance has been constructed:

We will now discuss one more interesting thing a dynamic language like Python can do (but languages like Java and C++ cannot). We can change the meaning of a class as a program is running. Let's go back to a very simple class C, that stores one instance variable, but has no methods that change it. The report method prints the value of this instance variable.

```
class C:
    def __init__(self,init_instance_var):
        self.instance_var = init_instance_var

    def report(self,var_name):
        print('instance referred to by', var_name,
              '/instance_var=', self.instance_var)
```

Now look at the following code. It defines x to refer to an object constructed
from the class C, which defines only a report method (and then it calls that method to report). Next it defines the bump function with a first parameter

named self: its body increments the instance_var in self's namespace dictionary. We call bump with x and it updates x's instance_var (as seen by the
report).

Then we do something strange. We add the bump function into the namespace of C's class object with the name cbump (that is just the same as writing C.__dict__['cbump'] = bump, creating the cbump method). We could have used just
bump, but instead used a slightly different name. Then, we call x.cbump('x') which by the Fundamental Equation of OOP is the same as calling
C.cbump(x,'x')
and because we just made the cbump attribute of the object representing class C
refer to the bump function, its meaning is to call the same function object that bump refers to.

```
x=C(10)
x.report('x') # By FEOOP, exactly the same as calling C.report(x,'x')
print()
```

```
prints
  instance referred to by x /instance_var= 10
```

```
def bump(self,name):
    print(name,'bumped')
    self.instance_var += 1

bump(x,'x')
x.report('x')
print()
```

```
prints
  x bumped
  instance referred to by x /instance_var= 11
```

```
C.cbump = bump; # put bump in the namespace of C, with the name cbump
x.cbump('x')    # By FEOOP, exactly the same as calling C.cbump(x,'x')
x.report('x')
print()
```

```
prints
  x bumped
  instance referred to by x /instance_var= 12
```

```
y=C(20)
y.cbump('y')
y.report('y')
```

```
prints
  y bumped
  instance referred to by y /instance_var= 21
```

So, even after the class C has been defined, we can still add a method to it namespace and then can call it using any object that has already been (or will

be) constructed from the class C. That is, we can change the meaning of a class
C, and all the objects constructed from the class will respond to the change through the FEOOP.

Recall that the del command gets rid of an association in a dict. If we wrote del C.cbump, then if we tried to call that method by x.cbump('x') the result would be that Python raises

```
  AttributeError: 'C' object has no attribute 'cbump'
```

So we can both ADD and REMOVE names from a class object's namespace!

Note that we could have defined/written bump as follows (note use of p not self
as the first parameter of bump)

```
def  bump(p,name):
    print(name,'bumped')
    p.instance_var += 1
```

and nothing changes, so long as every occurrence of self is changed to p inside
the bump function (as is done). We could likewise write

```
def report(p,var):
    ...
```

inside the C class itself. The parameter name self is just the standard name used in Python code; but there is nothing magical about this name, and we can substitute whatever name we want for the first parameter. This parameter will be matched with the object used to call the method. Although any name is allowed, I strongly recommend always following Python's convention and using the name self. Eclipse always supplies the name self automatically as the first
parameter to any methods we define.

Defining a method for an instance (but not the whole class) after the
  instance has been constructed:

In fact, Python also allows us to add a reference to the bump method to a single
instance of an object constructed from class C, not the class itself. Therefore
unlike the example above, bump is callable only on that one object it was added to, not on all the other instances of that class. We can also use this technique to add a method to an object that is different than the one defined in the object's class.

Start with the class C as defined above, defining just __init__ and report. Then

```
def bump(self,name):
    print(name,'bumped')
    self.instance_var += 1
```

```
x = C(0)
y = C(100)

x.bump = bump;

x.bump(x,'x')
x.bump(x,'x')
x.report('x')

y.bump(y,'y')     # fails because there is no bump attribute in the object y
y.report('y')     #    refers to, either directly in y or in y's class C
```

Note that calling x.bump(..) finds the bump method in x's namespace, without needing to translate it using the Fundamental Equation of Object-Oriented Programming. Without the translation, we explicitly need to pass the x argument
which becomes bump's self parameter.

Generally when looking up the attribute x.attr (whether attr is a variable or method name), Python first looks in the namespace of the object x refers to, and if it doesn't find attr, it next looks in the namespace of the object type(x) refers to.

When run, this script produces:

```
x bumped
x bumped
instance referred to by  x /instance_var= 2
Traceback (most recent call last):
  File "C:\Users\USERNAME\Desktop\python32\test\script.py", line 21, in
    y.bump(y,'y')     # fails because there is no bump attribute in the
object y
AttributeError: 'C' object has no attribute 'bump'
```

The error message mentions C because after failing to find the bump attribute in the object y refers to, it looks in the object C refers to; when it fails there too, the raised exception reports the error.

So, we can add methods to

(a) classes: methods that all the instances of that class can refer to, via FEEOP

(b) an instance of the class, such that only that instance can call the method
(and their self parameter must also be passed as an explicit argument), since FEOOP is not used.

Combining both worlds using delegation

Suppose that we wanted to be able to call methods attached to objects, but do so using the standard object.method(...) syntax, as we did in the first part of
this section. In the second part of this section, we had to duplicate the

object, e.g., calling x.bump(x,'x') instead of just x.bump('x'). We will see how to return to this simpler behavior using the concept of delegation in Python.

We start by defining C as follows, adding the bump function below

```
class C:
    def __init__(self,init_instance_var):
        self.instance_var = init_instance_var

    def report(self,var):
        print('instance referred to by', var,
              '/instance_var=', self.instance_var)

    def bump(self,name):
        try:
            self.object_bump(self,name)
        except AttributeError:
            print('could not bump',name) # or just pass to handle the
exception;
                                         # or omit try/except altogther
```

In this definition of C, there is a bump method defined in the class C, for all
instances constructed from this class to execute, findable by FEEOP.

When bump is called here, it tries to call a method named object_bump on the instance it was supplied, passing the object itself to object_bump (doing explicitly what FEOOP does implicitly/automatically). If that instance defines
an object_bump function, it is executed; if not, Python raises an attribute exception, which at present prints a message, but if replaced by pass would just silently fail. Of course, we could also remove the entire try/except so an
attribute failure would raise an exception and stop execution.

Note that in the call x.bump(...) Python uses the Fundamental Equation of OOP to translate this call into C.bump(x,'x'), which calls the equivalent of x.object_bump(x,'x').

In the world of programming, this is called delegation (which we will see more
of): the bump method delegates to the object_bump method (if present) to get its work done.

Here is a script, using this class. It attaches different object_bump methods to the instance x refers to, and the instance y refers to, but not to the instance z refers to (nor to the class C). It calls this object_bump method not
directly, but through delegation by calling bump in the C class.

```
x = C(10)
y = C(20)
z = C(30)

def bump1(self,name):
    print('bump1',name)
```

```
    self.instance_var += 1

def bump2(self,name):
    print('bump2',name)
    self.instance_var += 2

x.object_bump = bump1
y.object_bump = bump2
# No binding of z.object_bump

x.report('x')
x.bump('x')
x.report('x')
print()

prints
  instance referred to by x /instance_var= 10
  bump1 x
  instance referred to by x /instance_var= 11

y.report('y')
y.bump('y')
y.report('y')
print()

prints
  instance referred to by y /instance_var= 20
  bump2 y
  instance referred to by y /instance_var= 22

z.report('z')
z.bump('z')
z.report('z')

prints
  instance referred to by z /instance_var= 30
  could not bump z
  instance referred to by z /instance_var= 30
```

Back To Top

Redefinition of Function Names

Note that we can redefine a function or class. For example, we can write

```
def f(): return 0
def f(): return 1
print(f())
```

Calling f() would return 1. Eclipse gets upset about this, and marks the second definition as an error (duplicate signature), but there is nothing technically wrong with this code (although the first definition is useless, and there may be a mistake in the spelling of one of these functions). Python will run the script. We can also write the following script, which Eclipse

won't complain about, and runs the same.

```
def f(): return 0
def g(): return 1
f = g
print(f())
```

Calling f() returns 1. Again, def just makes a name refer to a function object;
if, as in the case of the two definitions of the f name above, the name already
refers to an object, the binding of f is just changed to refer to the function
object g refers to.

Conceptually, it is no different than writing x = 1 and then x = 2 (changing what x refers to from the int object 1 to the int object 2).

We can do the same thing for classes, as we saw with the names C and D in the first example in these notes.

In summary, def f or class C just define a name and binds it to a function/class
object. We can call the function or the class's constructor. We can rebind that
name later to any other object. We can even write

```
def f(): return 0
f = 0
```

Now f is bound to an int instance object, not a function object.

Back To Top

Accessor/Mutator Methods (or query/command methods) and Instance Variables...single/double underscore prefix

---

If calling o.method(...) returns information about an o's state but does not change o's state, it is called an accessor (or query). If o.method(...) changes
o's state and returns None (all functions/methods must return some value), it is called a mutator (or command). Some method calls do both: they change o's state but also return a non-None value.

A design question arises. Suppose that we know that an object o of a class C has an instance variable name iv: should we directly refer to o.iv? Should we use its value by writing o.iv or change it by writing o.iv = ...? The high

road says, no: a class should hide the actual instance variables from the clients/users of the class; they should provide methods to manipulate the objects under control of the class. What instance variables we need to implement a class might change over time, but the methods that define the behavior of objects created from that class should stay the same and always work correctly with whatever instance variables we are using.

Python is a bit at odds with this philosophy. Some languages (Java/C++) have a
mechanism whereby instance variables can be tagged PRIVATE, so that they are accessible only from methods defined in the class itself; these languages do not allow new instance variables nor methods to be added dynamically to objects
as Python does (as we showed above).

Python's philosophy is a bit more open to accessing instance variables outside
of the class methods. But there is danger in doing so, and beginners often use
this convenience and end up taking longer to get their code to work correctly,
and also make it harder to understand and change the code, because they are accessing information that is not guaranteed to be there in future changes to the code.

In fact, Python does have a weaker form of tagging PRIVATE names, which we can
use for names that should not be referred to outside the methods in the object's class. Below we explain the meaning of instance variable names that begin with one or two underscores (but don't have two trailing underscores, so
are unlike __init__).

Single Underscore:

When a programmer uses a single underscore in a name in a class (for data or a
method), he/she is indicating that the name should NOT BE ACCESSED outside of
the methods in the class. But, there is nothing in the Python interpreter that
stops anyone from accessing that name. We can use this convention for private data and helper methods.

```
class C:
    def __init__(self):
        self._mv = 1

    def _f(self):
        return self._mv == 1

x =  C()
print(x._mv, x._f())
```

When run, this script produces: 1 True

Still, if a class is written with names prefixed by a single underscore, it indicates that objects constructed from that class should not access those names outside of the methods defined inside the class.

Double Underscore:

If a Python name in a class begins with two underscores, it can be referred to
by that name in the class, but not easily outside the class: but it can still be
referred to outside the class, but with a "mangled" name that includes the name
of the class. If a class C defines a name __mv then the name outside the class
can be referred to as _C__mv. This is called a "mangled" name.

So, if we changed the code in the class C above by writing _mv as __mv and _f as
__f, and tried to execute

```
x =  C()
print(x.__mv, x.__f())
```

Python would complain by raising an AttributeError exception for the first value in the print statement

```
  AttributeError: 'C' object has no attribute '__mv'
```

It x.__mv was remove, Python would complain about x.__f(), indicating the object has no attribute '__f'.

But given class C defines __mv, and __f we could execute the following code

```
x =  C()
print(x._C__mv, x._C__f())
```

by writing the mangled names. When we run this script it again produces: 1 True

In fact, if we printed the dictionary for x, it would show '_C__mv' as a key, which is the true name of these functions outside of the module.

print(x.__dict__) would print: {'_C__mv': 1}

So, Python does contain two conventions for hiding names: the first (one underscore) is purely suggestive; the second (two underscores) actually makes it harder to refer to such names outside of a class.

But neither truly prohibits accessing the information by referring to the name.

When we discuss operator overloading (later this week) and inheritance (later in the quarter) we will learn more about controlling access to names defined in objects and classes.

Back To Top

Defining classes in unconventional places

We normally define classes in modules, and often a module does nothing but define just one class (although some define multiple, related classes). Other modules define lots of functions. All such modules are called library modules (not scripts) because they typically don't run code themselves, but we import them to gain access to the names (of classes and functions) that they define. (If they do run code, it is inside the code if __module__ == '__main__': and often the code there allows us to test the class or module).

We can declare a class in a function and call the function to return an object
constructed from the class (and even use the returned object if we know its defined instance variables or methods).

```
def f(x):
    class C:
        def __init__(self,x):  self.val = x
        def double  (self)  :  return 2*self.val
    return C(x)

y = f(1)
print(y, y.val, y.double())
```

When run, this script prints the following.

```
<__main__.f..C object at 0x02829170> 1 2
```

The first value indicates (reading the first word after the left angle-bracket,
from back to front) that class C is defined local to function f, which is in the script we ran (named by Python to be __main__).

We can also declare a class in a class, and call some method in the class to return an object constructed from the inner class (and even use the object if we know its defined instance variables or methods).

```
class C:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    class Cinner:
        def __init__(self,x):  self.val = x
        def double  (self)  :  return 2*self.val
    def identity(self)   : return (self.x, self.y)
    def x_construct(self): return C.Cinner(self.x)
    def y_construct(self): return C.Cinner(self.y)

z = C(1,2)
```

```
a = z.x_construct()
b = z.y_construct()
print( z, a, b,sep='\n')
print(z.identity(), a.double(), b.double())

When run this script prints:
  <__main__.C object at 0x02A36310>
  <__main__.C.Cinner object at 0x02A36290>
  <__main__.C.Cinner object at 0x02A36E70>
  (1, 2) 2 4

z.x_construct() returns a reference to an instance of class C.Cinner, that
was
initialized by the x parameter in the __init__ method. When we call
a.double(),
the double method defined in Cinner() returns twice the value it was
initialized with.

In fact, instead of the x_construct and y_construct functions, we could
define
a more general construct function that takes either 'x' or 'y as arguments
and
constructs an object for self.x or self.y. The first way to do this uses the
parameter (matching 'x' or 'y') as a key to access __dict__

    def construct(self,which): return C.Cinner(self.__dict__[which])

The second way to do this uses the eval function, whose argument is either
'self.x' or 'self.y' depdending on the value of which.

    def construct(self,which): return C.Cinner(eval('self.'+which))
```

Back To Top

Defining/Using Static Methods in Classes

```
A method defined in a class is considered "static" if it does not have a self
parameter. Sometimes it is useful to write methods in a class that have this
property.

For example, suppose that we wanted to declare a Point2d class for storing
the
x,y coordinates for Points in 2-d space. The __init__ method would take two
such
arguments. But suppose that we also wanted to create Point2d objects by
specifying polar coordinates (i.e, using a distance and angle in radians). We
could write such a class (with this static method) as follows.

import math

Class Point2d:
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

```
    @staticmethod
    def from_polar(dist,angle):
        return Point2d( dist*math.cos(angle), dist*math.sin(angle) )

    ...more code
```

This method is preceded by @staticmethod (a decorator: we will discuss decorators later). It is meant to be called from outside the class, to create Point2d objects from polar coordinates. We can write calls like

```
  a = Point2d(0., 1.)
  b = Point2d.from_polar(1.0, math.pi/4)
```

Notice that we call Point2d.from_polar outside of the class by using the class
name Point2d and the static method name from_polar defined in that class. It has no self parameter.

Likewise, suppose that we wanted to write a helper function for computing the distance between two Point2d objects as a static method in this class. We could
write it as

```
    @staticmethod
    def _distance(x1,y1,x2,y2):
        return math.sqrt( (x1-x2)**2 + (y1-y2)**2 )

    def dist(self,another):
        return Point2d._distance(self.x, self.y, another.x, another.y)
```

Here this helper function is meant to be called only by the dist method in this class, so we write its name with a leading underscore. Note that again we
call it using Point2d. Because of FEOOP, we could also call this helper as self._distance(self.x, self.y, another.x, another.y) because type(self) is Point2d. Because it is static, the call does not put self as the first argument.

Finally, we could also write this helper function as a global function defined
outside Point2d, in the module that Point2d is defined in. But it is better to
minimize any kinds of global names; so, it is better to define this name inside
the class. In this way it won't conflict with any other name the importing module has defined.

Back To Top

Problems

---

1) What does the following script print?

```
class C:
    def __init__(self):
        print('C object created')
D = C
def C():
  print('C function called')

x = C()
y = D()
```

2) What does the following script print? Draw a picture of the object x refers
to using the graphical form for representing objects we learned during Week #1.

```
class C:
    def __init__(self,a):
        self.a = a

x = C(1)
C.__init__(x,2)
print(x.a)
```

3) Write a class C whose __init__ method has a self parameter, followed by low
and high. __init__ should store these values in self's dictionary using the same names, but do so only if low is strictly less than high (otherwise it should raise the AssertionError exception with an appropriate string.

4) Explain why each of the following code fragments does what it does: two execute (with different results) and one raises an exception.

```
g = 0
def f():
  print(g)

f()
g += 1
f()
```

```
g = 0
def f():
  print(g)
  global g
  g += 1

f()
g += 1
f()
```

```
g = 0
def f():
  print(g)
  g += 1

f()
g += 1
f()
```

5) Write a class C that uses a class variable to keep track of how many objects
are created from C (remember that each object creation calls __init__)

That is, for a class C

```
a = C(...)
b = C(...)
print(C.instance_count) prints 2
c = C(...)
print(C.instance_count) prints 3
```

6) What would the following script print; explain why. Also, explain why the call self.object_bump(name) in bump is not self.object_bump(self,name) as it was in the notes.

```
class C:
    def __init__(self,init_instance_var):
        self.instance_var = init_instance_var

    def report(self,var):
        print('instance referred to by', var,
              '/instance_var=', self.instance_var)

    def bump(self,name):
        try:
            self.object_bump(name)
        except AttributeError:
            print('could not bump',name) # or just pass to handle the
exception

x = C(10)
y = C(20)

def bump(self,name):
    print('bumped',name)
    self.instance_var += 1

C.object_bump = bump

x.report('x')
x.bump('x')
x.report('x')

y.report('y')
y.bump('y')
y.report('y')
```

7) The function print is bound to a function object. What is printed by the following script (and why)?

```
print = 1
print(print)
```

Iterators are one of the most useful (and used) features in Python. Besides being used explicitly in loops (including for loops in comprehensions), constructors often include a parameter that is iterable, whose values are used
to initialize the state of the objects they are building: e.g., we write set(alist) to create a set with all the values from alist (obviously removing duplicates in the process). Also, calls to functions/classes like sorted/reversed are used in iterators: e.g., for i in sorted(aset): ... takes an iterable as an argument and returns a list.

In the next three lectures we will explore iterators in more detail. The first
focuses on the underlying mechanics of iterators; the second focuses on examples
of iterators written in/for classes; the third introduces a new kind of function, called a generator (in computer science it is also known as a coroutine) that returns a value, but when called again remembers where it "left
off" to return another value: this feature makes writing all sorts of iterators
as functions (not classes) much easier.

Throughout these lectures we will also discuss iterator efficiency issues (both
time and space): because iterators are used so often, their running times can dominate a program's running time; they often use little space because they produce their values one at a time, rather than storing them all in a data structure which is iterated over (although with constructors and comprehensions
we can easily create such a data structure from a iterator if we need to).

In this lecture we will learn about the __iter__ and __next__ methods (which a
class must implement to be iterated over), how a for loop for iteration is equivalent to a while loop (which needs a try/except block to catch the StopIteration exception that is raised by "exhausted" iterators), and how sharing and mutation affects iterators: sometimes they can cause problems, and
we will learn how to mitigate such problems, although often at the cost of using more space.

Back To Top

Semantics of the For Loop

For a class to be iterable (i.e., used in a for loop) it must implement the iterator protocol, which means the __iter__ and __next__ methods. A protocol is
just a group of methods that work together to accomplish something useful. We have already seen the __enter__ and __exit__ methods that work together to

implement the context manager protocol. Included in the iterator protocol is the StopIteration class, which is an exception raised by __next__ to signal there are no more values to iterate over (terminating the for loop). The builtins module defines the StopIteration class.

Before writing some classes that act as iterators, we will explore the semantics
of Python's for loop, by showing how to build it from an equivalent -but more primitive- while loop. We can think of Python translating the former into the latter; the while loop is more primitive than the for loop, so we can see more
details of how for loop works by analyzing the while loop. In this process we will better understand for loops, and also see how to write loops that process
iterators in a more intricate way than the straightforward (but simple and very
useful) way that for loops proccess them.

---
else Interlude

Before discussing this translation, we should first understand how the else keyword is used in for/while loops. Since I'm not sure how familiar you are with these statements, I'll start at the beginning (but move quickly).

1) The break statement in Python (just the keyword break as a statement) can be
put in any for/while loop; semantically, when executed it terminates the loop.
I often write loops like

```
  while True:
     statements
     if test:
          break
     statements
```

In fact, when the if test: break statement appears first in the the loop, as shown below,

```
  while True:
     if test:
         break
     statements
```

we can write an equivalent while loop that incorporates the test directly

```
  while not (test):    # not is high precedence, so I put () around test
      statements
```

Some programmers/educators banish using breaks in loops, but I think that edict
is too extreme. I could give a long lecture on how programmers (and educators who teach programming) feel about break statements. Instead I'll just say the
following.

a) When I teach indefinite looping I teach the while True/break form first. It decouples deciding to loop from how the loop terminates. When students need to write a loop, they write an infinite loop first, and later decide where to test the "termination" condition, and what this condition should be; this test can be stated in the "positive" form: terminate when the test is True.

b) If that test is the first statement in the loop (as shown above; sometimes we need to re-arrange code to get equivalent code with that test first) then I will SOMETIMES convert it into a while loop whose test is a "continuation" condition; the test must be stated in the "negative" form: terminate when the test is False (equivalent to continue when the test is True).

c) I think it is much easier to think about termination (stop when some specific state occurs) than about continuation (continue in lots of other states). Think about terminating when a value reaches 0 versus continuing when the value is any positive number.

d) Students sometimes go crazy and write too many different conditional breaks inside loops: there is no limit. Programmers need to work hard to reduce the number of breaks to simplify their code, but sometimes the simplest to understand code has one or a few breaks. In fact, we can also have conditional breaks in for loops, because they can terminate that kind of loop too (since for loops, as we'll see below, are translated into while loops): and sample use might be searching over a range of values with a for loop, but terminating when a special value is reached.

e) Often difficulties with breaks get resolved if we take the loop code we are writing and put it in a function, replacing muiltiple breaks by multiple returns. Of course some programmers/educators don't like to write functions with multiple returns.

Here is a cannonical example of a while loop that is more easily understood when written as a while True loop with a conditional break. This is a "sentinel" loop, which sums the values read until a sentinel (-1) is read. It is a "middle exit" loop, because termination is computed in the middle of the body of the loop.

```
sum = 0
while True:
    value = prompt.for_int('Enter test score (-1 to terminate)')
    if value == -1:
        break
    sum += value
print(Sum =',sum)  # we know here the loop exited, so value == -1 is True
```

To use a while test loop (and no conditional break), we would need to write

```
sum = 0
value = prompt.for_int('Enter test score (-1 to terminate)')
while value != -1:
    sum += value
    value = prompt.for_int('Enter test score (-1 to terminate)')
print(Sum =',sum)  # we know here the loop exited, so value != -1 is False
```

What I object to in the code above is the duplicate prompt; in other code there
might be more even statements duplicated.

2) The actual syntax of a for/while loops are

```
for index(es) in iterable:              while boolean-expression:
    block-body                              block-body
[else:                                  [else:
    block-else]                             block-else]
```

Semantically, when each loop terminates (it may terminate "normally" or by executing a break inside the first block), if the else keyword is present and the loop terminated normally, then Python executes block-else. So else means: execute block-else if the loop didn't execute a break to terminate. In the case of the for loop, it means the iterator stopped; in the case of the while loop it means the boolean-expression finally evaluated to False.

As a simple example that illustrates the use of else, we could write the following code (notice the lese is indented at the level of the for, not the if)

```
for i in range(100):
    if special_property(i):
        print(i,'is the first with the special property')
        break
else:
     print('No value had the special property')
```

I don't find myself writing else in loops much, but that might be because I am new to a language that allows such a feature. As I continue to use Python, I'll come to a more concrete conclusion about the usefulness of this language feature.

OK, now back to the main event: iterators. Python translates for loops like

```
  for index(es) in iterable:  # indexes for unpacked assignment
      block-body
  [else:
      block-else]
```

into the following code.

```
  _hidden = iter(iterable)             # ultimately calls iterable.__iter__()
  try:
      while True:
```

```
          index(es) = next(_hidden)    # ultimatelyl calls _hidden.__next__()
          block-body
  except StopIteration:
      pass                             # A place-holder in case [] discarded;
      [block-else]                     #    the except block cannot be empty
```

Note that like Python's len function, its iter and next functions translate into method calls of __iter__ and __next__ on their argument: e.g.,

```
def iter(i):
    return i.__iter__()

def next(i):
    return i.__next__()
```

Study this equivalence carefully. Note that _hidden is a special name used by Python to translate the for loop (it is not really "_hidden"; I'm just trying to indicate it is a hidden name); it is initialized by calling iter(iterable) but we cannot use this name in our code; it can be used only in the Python translation of the for loop. Then Python executes a while True loop INSIDE a try/except statement. Each iteration of the loop binds the next value(s) to be
iterated over to index(es) and then executes block-body. This
rebinding/block-body execution continues until either

  (1) next raises the StopIteration exception, which is handled by causing the
      loop to terminate (the except is OUTSIDE the loop, so handling the
      exception causes Python to exit the loop, terminating it) and executing
      block-else (and if there is not block-else, just executing pass)

  (2) block-body executes a break statement, which causes the loop to terminate.
      Because there was no exception in the body of the try, the try/except
      terminates without having to handle an exception, and therefore does not
      execute block-else.

As a concrete example, the simple for loop

```
for i in range(1,6):        # for the values 1-5
    print(i)
    #break  # uncomment to see what break does
else:
    print('executing else')
```

is translated into

```
_hidden = iter(range(1,6))
try:
    while True:
        i = next(_hidden)
        print(i)
        #break  # uncomment to see what break does
except StopIteration:
    pass
    print('executing else')
```

Try executing both scripts in Eclipse, including uncommenting each break statement to observe the effect of executing this statement: it terminates the
loop and doesn't print 'executing else'

Now that we understand the use of the iter/next functions, we can write more interesting loops (while loops) that process iterables. For example, suppose that we wanted to write a function that returned a list of the absolute values
of the differences between each adjacent pair of values in an iterable: e.g., for the 5-list [5, 2, 8, 3, 5] it would return the 4-list [3, 6, 5, 2], where 3 = abs(5-2), 6 = abs(2-8), 5 = abs(8-3), and 2 = abs(3-5): there will always be one fewer values in the returned list because in a list of N values their are N-1 adjacent pairs to subtract.

The following simple code works for arguments that can be sliced (like lists)

```
def abs_dif(alist):
    return [abs(a-b) for a,b in zip(alist,alist[1:])]
```

But it is not easy to write this function efficiently using a for loop operating on an arbitrary iterable (since arbitrary iterables don't support slicing). We could turn the iterable into a list but that would waste space.

```
def abs_dif(iterable):
    alist = list(iterable)
    return [abs(a-b) for a,b in zip(alist,alist[1:])]
```

We can write abs_dif without ever turning the iterable into a list as follows,
using the kind of code we explored above.

```
def abs_dif(iterable):
    answer = []
    i  = iter(iterable)
    v2 = next(i)
    try:
        while True:
            v1, v2 = v2, next(i)        # first time, v1 is 1st value, v2 is
2nd
            answer.append(abs(v1-v2))
    except StopIteration:
        pass
    return answer

print(abs_dif(range(1,10,2)), abs_dif([5, 2, 8, 3, 5]))
```

Now, I wouldn't call this function simple/beautiful, but it is pretty easy to write and understand. I don't think this function can be written with a for loop or comprehension (and not waste space), and I'm unsure if there are any other Python features that we could use to write it more simply and efficiently.
But maybe I just haven't yet seen the right Python idiom.

Back To Top

Classes implementing the iterator protocol: how range really works in Python

---

In this section we will first write a very simple Countdown class and then a more complex prange class (pseudo-range) that acts like the real range class from Python's builtins module. Then we will generalize it by overloading some simple operators for it (as the real range class does). I'm not sure how range
is really implemented, but this implementation seems straight-forward.

I would like the following iterator behavior for Countdown objects. The loop

```
for i in Countdown(10):
    print(str(i)+', ',end='')
print ('blastoff')
```

Should print: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, blastoff

Here is the class implementing these semantics.

```
class Countdown:
    def __init__(self,last):
        self.last = last       # self.last never changes

    # __iter__ must return an object on which __next__ can be called; it returns
    # self, which is an object of the Countdown class, which defines __next__
    # Later we will see a problem with returning self: when multiple Countdown
    # objects must be iterated over simultaneously.

    def __iter__(self):
        self.n = self.last     # self.n changes in __next__
        return self

    def __next__(self):
        if self.n < 0:
            raise StopIteration
        else:
            answer = self.n    # or, self.n -= 1
            self.n -= 1                #     return self.n+1
            return answer
```

In this class, when __iter__ is called it (re)sets self.n (the value __next__ will return first) to self.last (which is set in __init__ and never changes for
a constructed object). The __iter__ method has a requirement that it must return an object that defines a __next__ method. Here it returns self, which is
an object constructed from Countdown, which defines __next__ (right below

```
__iter__).

When __next__ is called it checks whether self.n has been decremented past 0,
and if is has, throws StopIteration; otherwise it returns the current value
of self.n but before doing so it decrements self.n by 1 (by saving it,
decrementing it, then returning the saved value).

Note that if we substituted Countdown(-1) in the loop above, its body would
be
executed 0 times and nothing would be printed before "blastoff".

Also, the following code counts-down twice; in each for loop __iter__ is
called,
which initializes self.n to 10 before calling __next__ multiple times.

cd = Countdown(10)
for i in  cd:
    print(str(i)+', ',end='')
print ('blastoff')

for i in  cd:
    print(str(i)+', ',end='')
print ('blastoff')

It should print:
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, blastoff
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, blastoff

------------------------------
Quick interlude: I originally said Python defines iter like len:

def iter(i):
    return i.__iter__()

But the truth is closer to the following (not completely true yet, but
correct
until we discuss inheritance).

def iter(i):
    x = i.__iter__()
    if '__next__' not in x.__dict__ and '__next__' not in type(x).__dict__:
        raise TypeError('iter() returned non-iterator of type
'+type_as_str(x))
    return x

If neither the object returned by i.__iter__() or the class (type) it was
constructed from define a __next__  method, then Python raises an exception
immediately: not waiting for a call of __next__ on the iterator to raise a
NameError exception.
------------------------------

prange: initialization (and the meaning of its arguments)

Now let's aim higher and write a prange class that operates like the range
class. Recall there are 3 possible parameters for range: start, stop, and
step.
```

Start has a default value of 0 and step has a default value of 1, but we cannot
write the following parameter structure for the __init__ in prange.

```
    def  __init__(self,start=0,stop,step=1):
        ....
```

because prange(5) would be an illegal call: it would bind start to 5 but have no value to bind for stop; but in this case start should be 0 and stop should be bound to 5. More generallly prange(5) should bind the first/only argument to
stop; prange(1,5) should bind the first argument to start and the second to stop; and prange(1,5,2) should bind the first argument to start and the second
to stop, and the third to step. How can we write such arguments for such an __init__ method?

Here is the start of the prange class, using *args in __init__ to solve this parameter problem. The __init__ methods raises exceptions like range: there must
be 1-3 int arguments and the value of step cannot be 0; the __repr__ method returns a string just like range: it always shows start and stop values, but shows the step value only if it is not 1.

To run this code, I need to import the type_as_str function from goody.

```
class prange:
    def __init__(self,*args):
        for a in args:
            if not type(a) is int:
                raise TypeError('\''+type_as_str(a)+'\' object cannot be interpreted as an integer')

        self.start, self.step = 0,1   # defaults
        if len(args) == 1:
            self.stop = args[0]
        elif len(args) == 2:
            self.start, self.stop = args[0], args[1]
        elif len(args) == 3:
            self.start, self.stop, self.step = args[0], args[1], args[2]
            if self.step == 0:
                raise ValueError('3rd argument must not be 0')
        else:
            raise TypeError('range expected at most 3 arguments, got '+str(len(args)))

    def __repr__(self):
        return 'prange('+str(self.start)+','+str(self.stop)+('' if self.step==1 else ','+str(self.step))+')'
```

---

prange: implementing the iteration protocol

Now let's add the main functionality: the iterator protocol methods. They are similar to but generalize what we wrote in the Countdown class.

```
    def __iter__(self):
        self.n = self.start # first value to return
        return self         # must return object on which __next__ is
callable

    def __next__(self):
        if self.step > 0 and self.n >= self.stop or\
           self.step < 0 and self.n <= self.stop:
            raise StopIteration
        answer = self.n
        self.n += self.step
        return answer
```

In this class, when __iter__ is called it (re)sets self.n (the value __next__ will return) to self.start (which never changes). The __iter__ method has a requirement that it must return an object that defines a __next__ method. Here
it returns self, which is an object constructed from prange, which defines __next__ (right below __iter__).

When __next__ is called it checks whether self.n has reached or exceeded self.stop (different tests, depending on whether self.step is positive or negative: self.step cannot be 0) and throws StopIteration; otherwise it returns
the current value of self.n but before returning it increments self.n by self.step (by saving it, incrementing it, returning the saved value). We could
avoid the temporary name answer by writing the following code, but it seems clumsy to me.

```
        if self.step > 0 and self.n >= self.stop or\
           self.step < 0 and self.n <= self.stop:
               raise StopIteration
        self.n += self.step
        return self.n - self.step
```

We discussed similar code in Countdown.

Try running various loops or comprehensions using prange objects to ensure that
this code perform like Python's built-in ranges.

---

prange: overloading operators

Now we move prange closer to Python's real range class by writing methods that
implement __len__, __getitem__, __contains__, and __reversed__. All of these methods use some fancy mathematics to compute their results, so I won't discuss
here how these method work in detail (but I encourage you to examine them and calculate examples). Note that __reversed__ just returns a new prange object, but with different start/stop values and a step that is the opposite.

To run this code, I need to import math (to use the math's ceiling function:

```
returns an integer >= its float/int argument; ceiling(3.9) is 4).

    def __len__(self):
        if self.step > 0 and self.start >= self.stop or \
           self.step < 0 and self.start <= self.stop:
            return 0
        else:
            return math.ceil((self.stop-self.start)/self.step)

    def __getitem__(self,n):
        if n < 0 or n >= len(self) :  # yes, could be n >= self.__len__()
            raise IndexError(str(self)+['+str(n)+'] index out of range')
        return self.start+n*self.step

    def __contains__(self,n):
        if self.step > 0:
            return self.start<=n< self.stop and (n-self.start)%self.step == 0
        else:
            return self.stop< n <= self.start and abs(n-
self.start)%abs(self.step) == 0

    def __reversed__(self):
        if self.step > 0:
            return prange(self.start+(len(self)-1)*self.step,self.start-1,-
self.step)
        else:
            return prange(self.start+(len(self)-1)*self.step,self.start+1,-
self.step)


------------------------------

prange: an alternative implementation (poor use of space and potentially
time)

For my final topic in this section, I am going to rewrite an alterntaive
implementation of this class: one that uses __init__ to generate and store
the
complete list of values that are in a range. I will then discuss the
complexity
of the code and some  time/space tradeoffs.

In this code, the __init__ method computes and remembers start, stop, and
step
(but only for use in __repr__). The code at the end of __init__ is a while
loop
that explicitly iterates through all the values in the range, storing each
value
in a list. (try to reverse the test and replace True by the continuation
condition for this loop).

Once we have a list with all these values, all the other methods are much
simpler to implement (they typically just delegate to the list methods to get
their job done) and require no complicated mathematics to write correctly.
The
__iter__ method just delegates to construct an iterator for the list (defined
in the list class); The __next__ method is not defined in this class, because
```

```
the object returned by __iter__ is a list iterator, which has its own
__next__
method defined in the list class. len, [], in, and reversed all work by
delegating to the list.

from goody import type_as_str
import math
class prange:
    def __init__(self,*args):
        for a in args:
            if not type(a) is int:
                raise TypeError('\''+type_as_str(a)+'\' object cannot be
interpreted as an integer')

        self.start, self.step = 0,1   # defaults
        if len(args) == 1:
            self.stop = args[0]
        elif len(args) == 2:
            self.start, self.stop = args[0], args[1]
        elif len(args) == 3:
            self.start, self.stop, self.step = args[0], args[1], args[2]
            if self.step == 0:
                raise ValueError('3rd argument must not be 0')
        else:
            raise TypeError('range expected at most 3 arguments, got
'+str(len(args)))

        self.listof = []
        self.n = self.start
        while True:
            if self.step > 0 and self.n >= self.stop or \
               self.step < 0 and self.n <= self.stop:
                break
            self.listof.append(self.n)
            self.n += self.step

    def __repr__(self):
        return 'prange('+str(self.start)+','+str(self.stop)+','+('' if
self.step==1 else str(self.step))+')'

    def __iter__(self):
        return iter(self.listof)

    # no need to define __next__: __iter__ returns iter(list) and list
defines __next__

    def __len__(self):
        return len(self.listof)

    def __getitem__(self,n):
        return self.listof[n]

    def __contains__(self,n):
        return n in self.listof

    def __reversed__(self):
        return reversed(self.listof)
```

So what are the tradeoffs between these two implementations? The list implementation of the prange class is much simpler (except for the loop in __init__ to turn a the range into a real list is 8 extra lines: 23 vs 15); it requires no complex mathematics (but while such mathematics is hard for us, it
is trivial for the computer). This class can require a huge amount of memory for storing a large range (e.g., prange(1,1000000000), while the original pranges stores only 4 instance variables. Also, __init__ takes a lot of time to construct such a list (and we might not even iterate over all the values in
this prange). Finally, the __contains__ method here takes an amount of time proportional to the range size (we will study why soon), to search for the value in the list; whereas the original implementation computes this value just
by some quick arithmetic, independent of the size of the range.

As the quarter goes on, we will study efficiency more formally, but I thought this was a good first time to bring up the issue, dealing with alternative implementations for the prange class (and its iterator). Again, the main point
here is that the original prange COMPUTED the values in the range, while the list version EXPLICITLY stored all the values in a list.

Generally, new Python students often create extra data structures that are not
needed. For example, they read an entire open file into a list and then iterate
over that list, when they could just iterate over the open file: if the file is
huge, it might not fit in memory as a list; but iterating over an open file just needs enough space to store one line at a time.

Although I've promoted "write simple code" in this course, we should start thinking about some of the time/space issues of using Python. Often using Python "efficiently" doesn't make the code more complicated.

Back To Top

Sharing iterators and Mutating objects that are being iterated over

---

Examine the following code

```
l = [0, 1,2,3,4,5,6,7,8,9]
x = iter(l)
y = iter(l)

print(next(x))
print(next(y))

print(next(x))
```

```
print(next(y))

print(next(x))
print(next(y))
```

It defines a list l, and constructs two iterator objects for the list: the first bound to x, the second bound to y. Each call of next(x) refers to one iterator object and each call of next(y) refers to another. Each call advances
the state of one iterator object. So identical values come out of each iterator: the code prints 0, 0, 1, 1, 2, 2 (two 0s, 1s, and 2s)

Now change the code to

```
l = [0,1,2,3,4,5,6,7,8,9]
x = iter(l)
y = x
```

Now x and y are bound to the same iterator object. Each call of next is on the
same OBJECT (whether next(x) or next(y)), which advances the state of that one
iterator object. So the print statements above print 0, 1, 2, 3, 4, and 5.

Finally, what do you think the following code will produce? The big question is,
does Python iterate over the values in l when the iterator is CREATED, or does
it iterate over the values of l at the time the iterator is USED (which might have been mutated since the iterator was created).

```
l = [0,1,2,3,4,5,6,7,8,9]
x = iter(l)
l.append(10)

try:
    while True:
       print(next(x),end=' ')
except StopIteration:
    pass
```

The answer is that it, iterates over the values of l at the time the iterator is USED. In this case it prints the integers 0 - 10.

What if we change the loop to

```
try:
    while True:
        v = next(x)
        print(v,end=' ')
        if v == 4:
            l[4:7] = ('a','b','c')
except StopIteration:
    pass
```

It prints: 0 1 2 3 4 b c 8 9, 10. Although 'a' is added to position 4, the value

in that position of the list has already been iterated over (returned), so when
__next__ is called again it prints the value in the next position, 'b'. So any
mutation we make to a list while it is being iterated over can affect the
results of the iteration.

Now look at the following code; it causes an infinite loop (printing higher and
higher values) because for every iteration of the loop, a new value is added to
the list, increasing the number of values to iterate over.

```
i = 0
l = [i]
for x in l:
    print(x)
    i += 1
    l.append(i)
```

So, the iterator object for a list is keeping track of what index it is on, but
the list it indexes (from which to get these values) is also growing (its len is
increasing). We can avoid all these problems by iterating over a COPY of the
list. There are many ways to create copies of objects (see the copy module, for
example) but the easiest way to make a copy of a list l is by using the idiom
list(l) or even l[:]. If we replace for x in l: by for x in l[:]: in the
example above, only the original list's value (just one 0) is printed. Of
course doing so make a copy of the list (using more space).

It would be easy to write a variant of a list class whose __iter__ method always
makes a copy of the list to iterate over, so any changes subsequently made to
the list will not change the result of the iteration. Of course, a drawback of
that approach is that it requires extra space to "remember what the list
contained when the iterator started". When we study lots of iterators in the
next lecture, we will see examples of these kinds of classes. Of course, besides
the extra space it takes extra time to make a copy, but sometimes it is
worth it (to avoid the problems shown above).

prange Class

```
import math
class prange:

    def __init__(self,*args):
        try:
            self.start, self.step = 0,1
            if len(args) == 1:
                self.stop = int(args[0])
```

```
            elif len(args) == 2:
                self.start, self.stop = int(args[0]), int(args[1])
            elif len(args) == 3:
                self.start, self.stop, self.step = int(args[0]),
int(args[1]), int(args[2])
                if self.step == 0:
                    raise ValueError('3rd argument must not be 0')
            else:
                raise TypeError('range expected 1-3 arguments, got
'+str(len(args)))
        except ValueError:
            raise TypeError('some argument cannot be interpreted as
integer:'+str(args))
        except TypeError:
            raise

    def __repr__(self):
        return 'prange('+str(self.start)+','+str(self.stop)+','+('' if
self.step==1 else str(self.step))+')'

    def __iter__(self):
        self.n = self.start
        return self # must return an object on which __next__ can be called

    def __next__(self):
        if self.step > 0 and self.n >= self.stop or \
           self.step < 0 and self.n <= self.stop:
            raise StopIteration
        save = self.n
        self.n += self.step
        return save

    def __len__(self):
        if self.step > 0 and self.start >= self.stop or \
           self.step < 0 and self.start <= self.stop:
            return 0
        else:
            return math.ceil((self.stop-self.start)/self.step)

    def __getitem__(self,n):
        if n >= len(self) :  # yes, could be n >= self.__len__()
            raise IndexError('range('+str(self)+') index('+str(n)+') out of
range')
        return self.start+n*self.step

    def __contains__(self,n):
        if self.step > 0:
            return self.start<= n < self.stop and (n-self.start)%self.step ==
0
        else:
            return self.stop< n <=self.start and abs(n-
self.start)%abs(self.step) == 0

    def __reversed__(self):
        if self.step > 0:
            return prange(self.start+(len(self)-1)*self.step,self.start-1,-
self.step)
```

```
        else:
            return prange(self.start+(len(self)-1)*self.step,self.start+1,-
self.step)
```