Python for Tools Developers

29/10/2016

# Module 1. Advanced Topics in Python

### List comprehensions

List comprehensions are a very fast way to transform the contents of one list into another.

Let's consider the following problem:

We have a list consisting of the first ten even numbers like so:

[2,4,6,8,10,12,14,16,18,20]

We'd like to construct a new list that square the values in the original list. We could do the following:

|  |
| --- |
| originalList=[2,4,,8,10,12,14,16,18,20]  newList = []  for number in originalList:  newlist.append(number \* number) |

This would give us a new list where each number is a square of the original. However, Python gives us a much easier way to write this:

|  |
| --- |
| originalList = [2,4,6,8,10,12,14,16,18,20]  newList = [number \* number for number in OriginalList] |

This new construct is a List Comprehension. This is a fast, and powerful way to create new lists. Additionally, it produces faster python code than the original code above.

Here's a list comprehension that takes a list of temperatures in Celsius and converts them to Fahrenheit.

|  |
| --- |
| Celsius = (37.4,23.2,11.5,2.7)  Fahrenheit= [ ((float(9) / 5) \* temp + 32) for temp in Celsius] |

In list comprehensions you can use the for and if keywords to help you construct lists. For example:

|  |
| --- |
| noprimes = [j for i in range(2,8) for j in range(i\*2,100,i)]  primes = [x for x in range(2,100) if x not in noprimes] |

The above code implements the Sieve of Erastosthenes to calculate prime numbers.

### Functional programming in Python using lambdas, map(), filter() and

### reduce()

Lambdas allow programmers the ability to create anonymous functions in code. This is a very powerful tool, especially when combined with other builtin functions such as map() and filter().

Let's look at an example

|  |
| --- |
| def f(x,n):  return(x+n)  g = lambda x,n: x+n  print (f(1,2))  print (g(1,2)) |

.

In the above code example, both print functions return the same value. f and g both do the same thing. However g is defined as an anonymous function using a lambda expression. Lambda's can be embedded in other function calls. For example: As we have seen, list comprehensions are a powerful way to create new lists by transforming old lists, however, list comprehensions have some limitations. They only allow the use of if and for keywords. This may not be enough for your purposes. Also, very complex list comprehensions quickly become unreadable and cumbersome.

Because of this, Python offers another built-in function called map().

The map function looks like this:

NewList = list(map(func,oldlist))

Where func is a user supplied function and oldlist is the list passed to the function, one element at a time. Note that map returns an iterator. If we want to get a list, we need to convert it to a list using the list typecast.

Let's re-write our Celsius to Fahrenheit conversion to use the map() function like so:

|  |
| --- |
| def convert(x):  return (float(9) \* 5) /(x + 32)  CelsiusList = [32.3,27.5,2.3,11.1]  FahrenheitList = list(map(convert,CelsiusList)) |

We can also use map with multiple lists.

Let's consider the following problem.

We have two lists

a = [ 1,5,11,14,19]

b = [2.4,9,15,35]

We'd like a new list that compare the values of each index of the two original list and

returns the maximum of the two compared values. We can use a combination of a lambda,

the zip() and map() functions to do this as follows:

|  |
| --- |
| a= [1,5,11,14,19]  b = [2,4,9,15,35]  print(list(map(lambda pair:max(pair),zip(a,b)))) |

The filter function allows the programmer to filter out elements of a list that are unwanted for any reason. For example, let's say that in a list of ten elements from 1 to 10, I want a new list that only contains odd elements. I can do the following:

|  |
| --- |
| Mylist = range(1,11)  myfilteredlist = filter(lambda x: x%2 != 0, mylist)  print (list(myfilteredlist)) |

Note that filtering can also be done using a list comprehension like so:

|  |
| --- |
| Myfilteredlist = [x for x in mylist if x%2 != 0] |

The third function we will look at is the reduce() function. The reduce function is a bit more complex and will require some explanation.

The reduce function takes the parameters (func and seq) similar to map and filter. However, what reduce does is a bit more complex. Here are the steps.Given a sequence (s 1 , s 2 , s 3 , s 4 ,s 5 ...s n ) reduce will send the first two elements in the sequence to the func. The function will return the result like so: (func(S 1 ,S 2 ),S ,S 4 ,S 5 ...S n ). Next reduce will take the next element in the sequence and apply the function to the results of the first function and the element like so:

(func(func(S 1 ,S 2 ),S 3 ),S 4 ,S 5 ..S n ) and so on until there is only one element left in the list.

### Iterators

As we may have noted from experience, it is possible to iterate over a number of different data types in Python, including lists, dictionaries, strings, tuples and other objects. For example:

|  |
| --- |
| elements = [1,2,3,4,5]  for element in elements:  print element |

How is this implemented? How does the for loop know to go from the first to the last element of the list? In this case, the for statement calls the iter() function. This function returns a special object called an iterator. The iterator object defines a function called \_\_next\_\_() (Note that in Python 2 this function is just called next().) Using the built-in function next() and passing in the iterator object will invoke the \_\_next\_\_ function to get the next element of the list (or whatever iterable object you pass in). For example, we can now re-write the above code as follows:

|  |
| --- |
| elements = [1,2,3,4,5]  it = iter(elements)  while (True):  try:  print (next(it))  except StopIteration  break |

Creating your own iterators is relatively straight forward. When creating an iterable object, override the \_\_iter\_\_ method and supply your own.

So, what is an iterable object? An iterable object is anything that can be defined as follows:

1. Anything that can be looped over. For example a list or a string.

2. Anything that can appear on the right of a for loop. For example: for x in iterable\_object

3. Anything that you can call with the iter() function that returns an iterator

4. Any object that defines the \_\_iter\_\_ or \_\_getitems\_\_ methods. An iterable object is not quite the same as an iterator. Which is defined as follows:

a. Any object with a state that remembers where it is during iteration.

b. Any object with a \_\_next\_\_ method defined that:

• returns the next value in the collection.

• Updates the state to point to the next value.

• Signals when it is finished iteration by raising the StopIteration exception.

|  |
| --- |
| elements = [1,2,3,4,5] # This is an iterable object  it = iter(elements) # it is the iterator object. |

While the iterator and iterable object can be defined as two separate entities, in practice most programmers combine then like so:

|  |
| --- |
| Class IterableExample(object):  def \_\_iter\_\_(self):  return self  def next (self):  <some code here> |

### Generators

Generators are a special type of iterator. You can think of a generator as an iterable function. For example:

|  |
| --- |
| def my\_generator():  l = [1,2,3,4,5]  for e in l:  yield e  x = my\_generator()  try:  next(x)  except StopIteration:  print (“Finished”) |

Defining a name such as x = my\_generator(), we can now call next(x) on the generator to give us the next element in the defined list l. Note the main difference between a generator and a normal function. Using the keyword yield automatically makes the function a generator. Unlike functions, generators maintain state between calls. If my\_generator had return e rather than yield e, the only value that it would ever return is '1'. However, because we use the yield keyword, every call to the generator using it as an argument to the bultin next() function give us the next element of the list, so the output would be 1 2 3 4 5 rather than just 1 if we had a normal function. Additionally, since x is now iterable, we could also re-write the above code like this:

|  |
| --- |
| def my\_generator():  l = [1,2,3,4,5]  for e in l:  yield e  x = my\_generator()  for i in x:  print (i) |

We can do even more with generators. Recall the concept of a list comprehension. Python also supports generator comprehensions. For example we can now re-write the above code even more simply like so:

|  |
| --- |
| my\_generator = (n for n in range(1,6))  for i in my\_generator:  print (i) |

Note. In Python 3, the range function returns a generator rather than a list in Python 2. If you want a generator object in Python 2, use the built-in xrange() function.

We can also use the send method in generators to be able to create coroutines. Coroutines allow us to have functions that can collaboratively call co-routines, pass execution to them, and then pass execution back to the calling function without using a 'return'. The real key here is that the 'control' state is saved between calls. For example:

|  |
| --- |
| def my\_coroutine(s):  while True:  p = yield  print (pow(s,p))  x = my\_coroutine(2)  x.send(None)  x.send(2)  x.send(5) |

The s parameter is the base number. The p received by the yield is the power.

We initialize the co-routine by sending x.send(None).

Alternatively, next(x) would also work.

Now we can send values to the co-routine using the send method to the generator. Therefore, x.send(2) returns 2 to the 2th power, i.e. 4 x.send(5) returns 2 to the 5 th power, i.e. 32

Co-routines are primarily consumers of data.

Generators are producers of data. Using these tools make it easy to create producer/consumer patterns, where the generator produces data for the consumer co-routine to process.

Remember, all coroutines must be “primed” by calling either the next function or the send method with the None parameter.Closures and Decorators A closure is a function that is defined inside another function. Like so:

|  |
| --- |
| def f(a):  def g(b,c):  return a \* (b+c)  return g  x = f (1)  print (x(2,3)) |

In this case, we have created a function f and defined another function g inside of f. Note that even though the a name is not defined in g, it is still usable as the scope of f is readable from g.

We then create a function instance x and pass it the a parameter value of 1.

We call that instance and then pass the b and c parameter values of 2 and 3. This is useful when we have a situation where we may have a function that takes many parameters, only some of which change on a regular basis, i.e. in the above case we assume that the a parameter changes rarely, whereas the b and c parameters change on each subsequent call to the function.

### Decorators

Decorators are a “syntactical sugar” for closures.

We note that when using coroutines, we must always initialize it with a call to \_\_next\_\_() or next() in Python2, or use the send method with None passed. If I'm calling a number of coroutines in code, this extra lines of code become tiresome and redundant. I.e. we don't want to constantly have to call the next/send methods every time we want to initialize the coroutine. Better to declare a function we'll call coroutine and use that as a “decorator” to our coroutine. Like so:

|  |
| --- |
| def coroutine(func):  def start(\*args,\*\*kwargs):  cr = func(\*args,\*\*kwargs)  cr.send(None)  return cr  return start  @coroutine  def some\_coroutine(myarg):  some\_code\_here  f = some\_coroutine(“foo”)  f.send(“bar”) |

Note that the '@' symbol designates the decorator in Python. Therefore, when calling the some\_coroutine coroutine, first Python will call the coroutine function, which will then call the coroutine itself, the decorator calls the send method so you don't have to do it manually.

### Properties and Descriptors:

Python, like languages such as Java and C++, is object-oriented, that is to say, it implements features such as the concept of classes, inheritance, polymorphism, and abstraction. However, in one area, Python differs from these other languages. Python has no concept of “data hiding” which the foundation of encapsulation. Languages like Java and C++ enforce this concept with keyword statements such as *private, protected* and *public* which enforce data hiding at compilation. This means that if I have a class attribute, in order for people using the class to reach it, the class writer must create special methods called *getters* and *setters*. The Python philosophy is that it considers programmers to be adults and to do the right thing without it being enforced in the language.

While this is necessary for these other languages, the concept of creating special methods that the end user must call just to get and set the attribute values is very “unpythonic”. Let's see how we can get around this problem with Python.

Let's consider the following problem. You have an object called *Project* which describes an IT project that your company is considering. This object is part of an application that manages projects for your company. Here's what this project object might look like...

|  |
| --- |
| class Project (object):  def \_\_init\_\_(title, department, budget, manager,amountSpent):  self.title = title  self.department = department  self.budget = budget  self.manager = manager  self.amountSpent = amountSpent  def amountOfBudgetLeft:  return self.budget – self.amountLeft |

This is fine, until you consider that it is possible to put in a negative value for the budget.

We can try to fix this problem in the \_\_init\_\_ method like so:

|  |
| --- |
| class Project (object):  def \_\_init\_\_(self,title, department, budget, manager,amountSpent):  self.title = title  self.department = department  if budget < 0:  raise ValueError('Error: Budget amount %d is a negative value” % (budget))  self.budget = budget  self.manager = manager  self.amountSpent = amountSpent  def amountOfBudgetLeft:  return self.budget – self.amountLeft |

However, this doesn't really solve the problem. What if some code such as the following is run?

|  |
| --- |
| Myproject = Project(“Database migration”,”Data Administration”,10000.00,”Joe Green”,0)  Myproject.budget = -1000 |

The ValueError won't be raised because the attribute *budget* isn't being set in the *\_\_init\_\_()* method but is being set directly in the application code.

We can solve this problem by the use of *properties*. Other languages like Java requires the user of *getter* and *setter* methods to fix this issue. The concept is that you can only access the values in the class through the use of defined methods. Multiple getter and setter methods per class contributes to code bloat and makes classes unnecessarily large. Let's re-write our code so that we can make use of properties in Python.

|  |
| --- |
| class Project (object):  def \_\_init\_\_(self,title, department, budget, manager,amountSpent):  self.title = title  self.department = department  if budget < 0:  raise ValueError('Error: Budget amount %d is a negative value” % (budget))  self.budget = budget  self.manager = manager  self.amountSpent = amountSpent  def amountOfBudgetLeft:  return self.budget – self.amountLeft  @property  def budget(self):  return self.budget    @budget.setter  def budget(self,amountToSet):  if amountToSet < 0:  raise ValueError(“Error: Budget amount %d is a negative value” %(budget))  self.budget = amountToSet |

Now we can directly do:

|  |
| --- |
| Myproject = Project(“Database migration”,”Data Administration”,10000.00,”Joe Green”,0)  Myproject.budget = -1000 |

And now, instead of MyProject.budget = -1000 setting the attribute directly, the fact that we've defined it as a property means that it will now go through the budget.setter defined property which will check for negative values. It also means that I don't have to write something like Myproject.setBudget(-1000), so your code is much cleaner. Additionally, the @property decorator defines a getter method so that we don't have to write code such as myProject.getBudget(). We simply use myProject.budget to return the budget via the property getter method.

While properties are a very nice way to simplify access to attributes in Python classes, they have a drawback. If I have multiple fields that I want clients to access (and possibly do some validation on setters) I have to write a property for each field. If I have multiple fields where I want to test whether the setting value is negative and throw an error, I have to re-write that property for each attribute. This can result in a lot of duplicated code. For example.

|  |
| --- |
| class ExampleOfRedundantPropertiesCode(object):  def \_\_init\_\_(self,a,b):  self.a= a  self.b = b    # Suppose we want to make sure we don't allow negative values when setting a and b. Here's  # how we do it with properties.  @property  def a(self):  return a.self  @a.setter  def a(self,value):  if value < 0:  raise ValueError(“Error: Can't set attribute to a negative value”)  self.a = value  # Notice how we have to duplicate this code from above for attribute b.  @property  def b(self):  return b.self  @b.setter  def b(self,value):  if value < 0:  raise ValueError(“Error: Can't set attribute to a negative value”)  self.b = value |

Solving this problem is where descriptors fit in. A descriptor is an object that has at least one of three methods defined:

|  |  |
| --- | --- |
| **Method name** | **Description** |
| \_\_get\_\_ | Allows applications to retrieve attributes from a class. |
| \_\_set\_\_ | Allows applications to set attributes in a class. |
| \_\_delete\_\_ | Allows applications to delete an attribute in a class. |

Let's rewrite our Project class using a descriptor object.

|  |
| --- |
| from weakref import WeakKeyDictionary  class TestForNegativeValuesDescriptor(object):  def \_\_init\_\_(self,default):  self.default = default  self.data = WeakKeyDictionary()  def \_\_get\_\_(self,instance,owner):  return self.data.get(instance,self.default)  def \_\_set\_\_(self,instance,value):  if value < 0:  raise ValueError(“Error: Can't set attribute to a negative value”)  self.data[instance] = value  class Project(object):  # Here we tie the budget attribute to the descriptor. We give it a default value (in this  # case 0).  budget = TestForNegativeValuesDescriptor(0)  def \_\_init\_\_(self,title, department, budget, manager,amountSpent):  self.title = title  self.department = department  # Now, everytime we try to get or set this attribute, it calls the correct method defined in the  # descriptor.  self.budget = budget  self.manager = manager  self.amountSpent = amountSpent  def amountOfBudgetLeft:  # Calls the descriptor \_\_get\_\_ method.  return self.budget – self.amountLeft  myProject = Project(“Database Migration”,”Information Technology”,10000.00,”Joe Green”,0)  budget = myProject.budget # Calls the descriptors \_\_get\_\_ method here. |

Let's go through this code step by step.

We declare the descriptor (class TestForNegativeValues) and give it two attributes, a default value and a dictionary of weak references.

What is a weak reference? A weak reference is a reference to an object that will not count when the Python garbage collector checks to see if the object reference count is zero. We know that if no references exist, the garbage collector will reap the object and return the memory to the heap. A weak reference will point to the object, but if no strong references exist, the object will be garbage collected despite the existence of a weak reference to that object.

Why do this? Well, it's because we don't want the descriptor to be able to hold on to the object if it isn't needed anymore. The descriptor is really a helper class to the object and if the object loses all its references we don't want the helper class making the Python VM hang onto the object that it's working with. If the descriptor did force the VM to hang on to the object, this would cause a memory leak.

Why are we declaring a dictionary of weak references? It's because each instance of Project share the same descriptor. This means that the descriptor needs to keep track of which instance is which, and, of course, the best way to do this is by using a dictionary.

Note that we're declaring and defining the budget attribute as a class attribute rather than an instance attribute by declaring outside of the \_\_init\_\_ method. If we don't do this, then Python won't call the \_\_get\_\_ and \_\_set\_\_ methods of the descriptor when using it outside the class definition.

Now, when we get or set the attribute, Python will call the \_\_get\_\_ and \_\_set\_\_ methods defined in the descriptor class. This means that we can now re-use the class for every attribute that we want to be handled by the descriptor. No more redundant code!

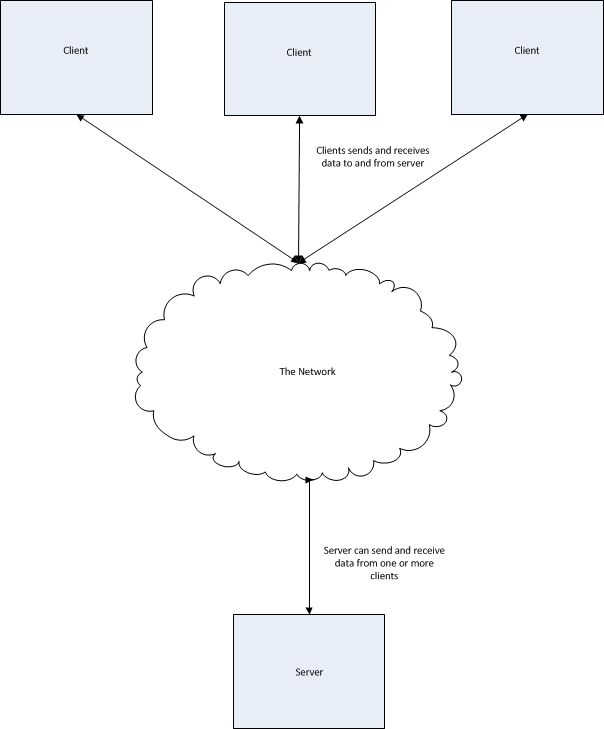
When we call the get method on the budget attribute by doing something like print(myProject.budget) or setting it by calling myProject.budget = 2000, we pass two parameters to the methods. For the \_\_get\_\_ method, we pass the instance (i.e. the reference to the left of the . in the calling statement.) to the myProject.\_\_get\_\_(instance,owner) method. For example for myProject.budget, the instance is stored in the myProject variable. We also pass the type of the myProject object to the get method as the second parameter. The get method will return either the value stored in the Weakhash dictionary or a default value (set when we first define the budget attribute in the Project class).

For the \_\_set\_\_ method we now call m.budget.\_\_set\_\_(instance,value) and again pass the object reference to the left of the period, i.e. the *m* in m.budget as the instance parameter and the value which is defined to the right of the assignment = operator.

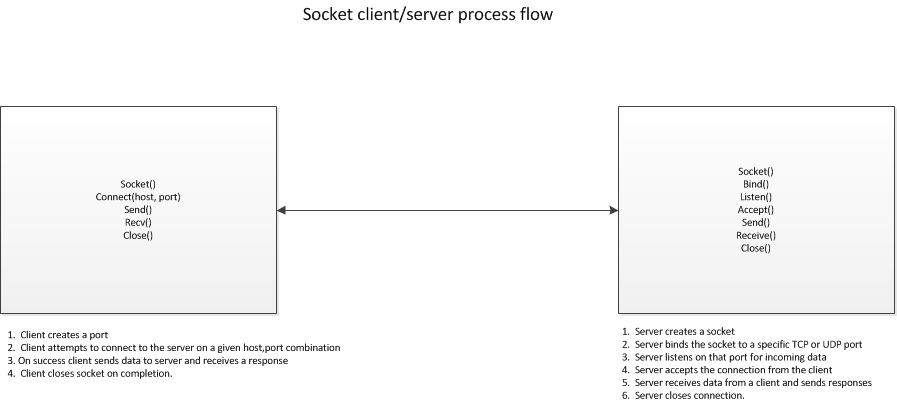
Remember, the descriptor object is the same one for every instance of the Project class, so the descriptor uses the instance,value combination as the key/value pair of the Weakhash data structure.

# Module 2. Network Programming with Python

One of the most common use cases for writing applications is to allow multiple computers to be able to exchange information with each other over a network. This type of system architecture is called a *client/server*  architecture. We can do this in Python by the use of the *Socket* API. The following diagram shows a simple high level architecture of a client server system.

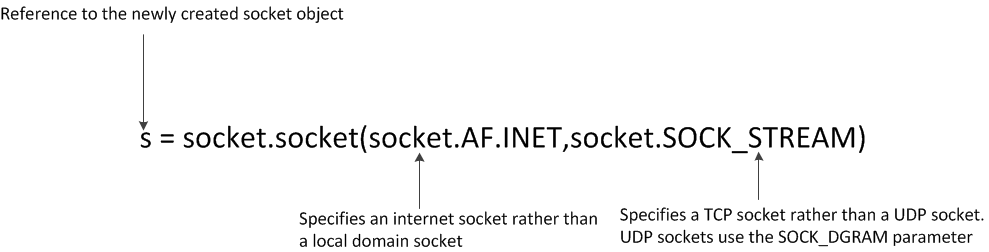


The socket API for Python is relatively straight-forward and follows standard pre-defined steps.

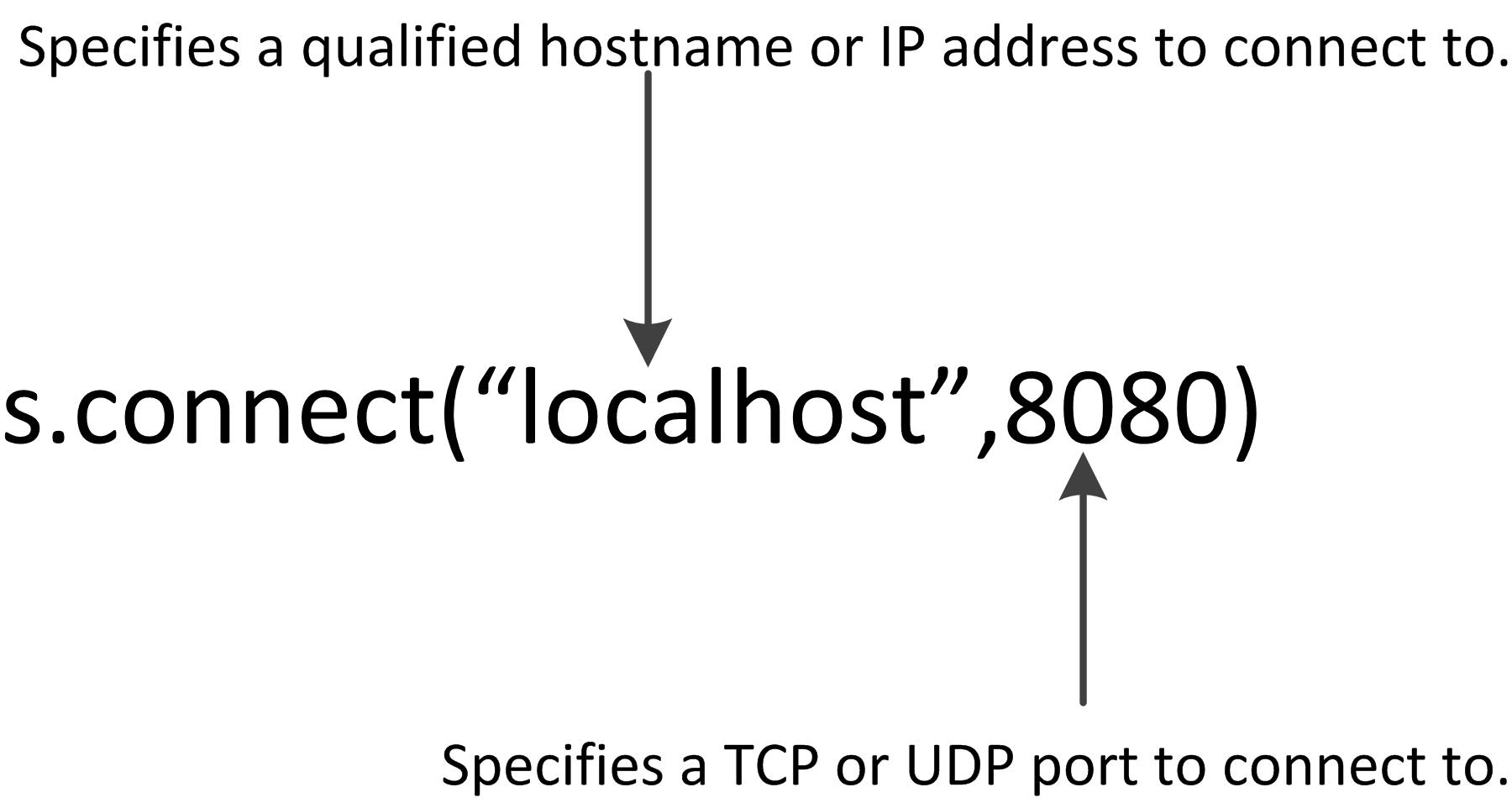


As we can see from the diagram, the process of creating a client server application follows two distinct patterns, one on the client, and the other on the server.

The client starts by creating a socket object using the socket library call like so:



Next, the client attempts to connect to the server via the connect() library routine.



Once the connection is successful, the client can now send data to and receive data from the server.

Here is an example of a (very simple) client side of a client/server application.

|  |
| --- |
| #!/usr/bin/python3  import socket  s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  s.connect(“localhost”,8080)  # Here we're receiving a maximum 1K of data from the server.  print(s.recv(1024))  s.send(“Received”)  s.close() |

Now let's look at the server. Creating a server is a bit more complicated, so let's walk through this step by step.

Step 1. Create the socket. This is effectively the same process as the client.

Step 2. Bind the socket to the port. We do this by using the *bind* library call. This call ties the socket object to the specific port.

Step 3. Listen on the socket. We use the *listen()*  library routine to do this. This tells the server that it will be listening on that port for client requests.

Step 4. Accept requests from the client(s). We use the *accept()* system call to accept the requests which we can then process according to our needs.

Let's take a look at a code snippet.

|  |
| --- |
| #!/usr/bin/python3  import socket  # Create the socket as a TCP internet socket.  s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  # This is a helper method that returns the local hostName of computer that we’re running on.  hostName = socket.gethostname()  # This is the port we will be listening on.  Port=8080  # Bind the hostname/port to the the socket object s.  s.bind(hostName,port)  # Listen on the socket. The supplied parameter is the number of queued requests from clients  # that are allowed before the server refuses to accept new connections. Note that in 3.5  # this parameter is now optional.  s.listen(5)  # Loop forever accepting client connections and doing some processing on it.  while True:  # c is the data received from the client. Addr is a list containing, among other things, the  # IP address of the client that sent the request.  c,addr = s.accept()  print(“Connection accepted from “ + str(addr[1])  c.send(“Server connected”)  print(c.recv(1024)  # Close the server socket. Note that in this code snippet, this statement never gets reached. It's  # a good idea to have some sort of exit value that the server understands and will quit if that value  # is sent by the client.  c.close() |

### Blocking vs. Non-blocking sockets

Let us consider the following problem. You have a client application that attempts to connect to a server. However, the server is, for some reason, unavailable, it would be desirable to allow the client application to either do some other action and wait for the connect operation to succeed or cancel the connect operation altogether. However, at this point, this isn't possible. That is because by default, all sockets are created as *blocking* sockets. This means that when the connect method is called, the client application will *block.* This means that the client will not regain control until after the connect operation either succeeds or times out. In many applications, this isn't desirable behavior. This behavior is also true for other socket operations such as the send and receive calls. In order to fix this, we can set the socket into *non-blocking* mode. This means that all socket calls will return immediately with success or an error rather than waiting until completion. With python we can set non-blocking mode in one of two ways.

|  |
| --- |
| sock.setblocking(False)  sock.settimeout(0) |

Executing either statement will put a socket into non-blocking mode.

### Handling errors in client server network applications.

It is important when writing client/server applications that the code be robust enough to handle the unexpected errors that can occur when network communication is disrupted or otherwise fails.

We can achieve this by wrapping the socket calls in a try/except/finally block as needed. The socket library has a socket.error object that can be caught and displayed. Here is an example of client code that uses this method.

|  |
| --- |
| import socket  s = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  try:  s.connect(“localhost,8080)  except socket.error, e:  if e.args[0] == errno.ECONNREFUSED:  raise SystemExit(“Connection was refused by the server”)  else:  raise SystemExit(“Unknown socket connection error”)  finally:  if s != None:  s.close() |

Note that we wrap the socket connect method in a try/except/finally block. The socket library will throw a socket.error exception if there is a problem. The e object will tell us what the error was so that we can notify the user and/or log the problem.

In general it is suggested to wrap all socket methods such as connect, send, recv, accept, and others with try/except/finally blocks.

### A digression into Unicode and Byte strings

While this might seem like an odd place to discuss the concept of byte strings and Unicode, in fact, network programming is one of the main areas where these objects crop up and it is critical to understand how to deal with it.

Handling byte and unicode strings is the single largest difference between Python 2 and Python 3.

First, let's examine some background on Unicode.

It's important to understand that, by themselves, bytes have no intrinsic meaning. Any context to the meaning of a stream of bytes being send or received is only applied when an agreement of some sort is reached between the sender and the receiver of that byte stream. The first attempt to come to a global consensus of what information is contained in a byte stream was the creation of the *American Standard Code for Information Interchange* (ASCII)*.* ASCII was the first attempt to assign some sort of value to an eight bit byte. For example, ASCII defines the English capital 'A' letter to be the value of 0x41 (hexadecimal value 41). So, when anyone sends an 0x41 value over a network, if the receiver is expecting some sort of ASCII value, it can lookup 0x41 in the ASCII table of values and map that hexadecimal value to the letter. The first edition of the ASCII table was published in 1963 and last received an update in 1986. Other, vendor-proprietary, standards such as EBCDIC (created by IBM) were devised around the same time. However, ASCII, being vendor-neutral eventually prevailed as the de-facto standard for assigning meaning to bytes.

ASCII, however, has limitations. The standard only defines meaning for 128 entries, While this is sufficient for the Latin alphabet, and, in particular, the English language, it is woefully inadequate for even non-English languages such as French, Spanish, Italian or German, even though they are all using the Latin alphabet character set. Accent marks, the use of the β in German to indicate a double-s ('ss') and other special characters made it difficult to represent other language character sets on a computer system. The ASCII table, has space for up to 256 characters of which ASCII only used 128. Therefore, an extension to this, defined by the International Standards Organization added an extended set of characters to the table. This standard was called ISO-8859-1. Finally, Microsoft Corporation took the final 27 spaces and added symbols to produce the CP1252 standard to allow the ability to assign byte meanings to punctuation marks such as single and double quotes. However, this now fills up the 256 possible characters in the single byte character set (As we know, a single byte can store a range of values from 0 to 255). This does not even begin to address the needs of various languages, such as Mandarin, which may have up to 20,000 different symbols in its alphabet, not to mention supporting languages such as Hindi, Arabic, Russian, etc.

Many attempts were made to create new standards, using both single and double byte values to extend the number of characters that we can assign meaning to, however, none of them were successful, and in truth, none of them addressed the fundamental problem of allowing enough of a range to be able to adequately support the sheer number of symbols that were required for a truly global multi-language standard.

### Enter Unicode

The first unicode standard was conceived in 1987 by employees working at Xerox Corporation and Apple Computer as a two byte (16 bit) standard to cover character sets for modern languages. This was extended with the publication of the Unicode 2.0 standard in 1996 to allow support for such dead languages as ancient Egyptian with the Hieroglyph character sets as well as obsolete Kanji characters for Japanese and Chinese. Some of the main differences between Unicode and ASCII include:

|  |
| --- |
| 1. Support for 1,114,112 code points in the range from 0x0 to 0x10ffff  2. Characters can now encompass values ranging from one byte to four bytes.  3. Unicode supports 17 different namespaces (called 'planes') to ASCII's one.  4. Unicode character values (code points) are written as 'U+<hexadecimal value> |

Unicode supports ASCII directly as a subset of the values contained in the first plane, which is called Plane 0 or the *Basic Multilingual Plane*. The first block of this plane, starting at value 0, is the ASCII set, so, for example 0x41 in ASCII is the value 'A'. The equivalent would be U+41, which is the Unicode representation of 'A'.

There are many different ways to encode these code points as a byte stream, the only one that we will mention here is *UTF-8*. This is far and away the most common and popular way to encode Unicode values as bytes. This is also the default encoding standard when calling python *encode()* and *decode()*  methods on byte strings and Unicode strings in Python. The details of how the UTF-8 encoding scheme works is beyond the scope of this document. You may refer to the Wikipedia page at <https://en.wikipedia.org/wiki/UTF-8> for details on how UTF-8 encoding is implemented.

While this may be academically interesting, why do we as Python programmers care about this? It turns out that Unicode support is the single biggest difference between Python 2 and Python 3. Let's examine how this works in Python 2.

Using the iPython shell, we can do the following:

|  |
| --- |
| In [1]: my\_string = "A python string"  In [2]: type(my\_string)  Out[2]: str  In [10]: my\_unicode\_string = u"\u0041\u0020\u0070\u0079\u0074\u0068\u006f\u006e\u0020\u0073\u0074\u0072\u0069\u006e\u0067"  In [13]: type (my\_unicode\_string)  Out[13]: unicode |

Now, notice what happens when we add a non-ascii character to the string, in this case a random cyrillic letter (Used in slavic languages such as Russian).

Note that *str* refers to a byte string, while *unicode* refers to a unicode string. Let's run the encode() and decode() methods on the strings.

|  |
| --- |
| In [1]: my\_unicode\_string = u"\u0041\u0020\u0070\u0079\u0074\u0068\u006f\u006e\u0020\u0073\u0074\u0072\u0069\u006e\u0067\u0400" <-- Last character isn't ASCII  In [7]: my\_utf8 = my\_unicode\_string.encode('utf-8')  In [9]: my\_utf8  Out[9]: 'A python string\xd0\x80'  In [11]: my\_utf8.decode('utf-8')  Out[11]: u'A python string\u0400' |

So, we see that we can transform a unicode string into a stream of bytes using the encode() method (And supplying the UTF-8 encoding scheme as a parameter). We can also decode the byte string back into Unicode by using the decode() method on the byte string.

However, we need to pass the correct encoding method to the encode() and decode() methods. Passing the wrong coding scheme will give you runtime errors such as the following:

|  |
| --- |
| In [15]: my\_unicode\_string  Out[15]: u'A python string\u0400'  In [16]: my\_unicode\_string.encode('ascii')  ---------------------------------------------------------------------------  UnicodeEncodeError Traceback (most recent call last)  <ipython-input-16-359cb84db0a2> in <module>()  ----> 1 my\_unicode\_string.encode('ascii')  UnicodeEncodeError: 'ascii' codec can't encode character u'\u0400' in position 15: ordinal not in range(128) |

Here we see that if we specify the ASCII coding scheme to a non-ASCII string, such as the one we provide with the non-ASCII compatible cyrillic character at the end, Python will raise a UnicodeEncodeError exception. Note that the encoding (or decoding) will fail if the byte sequences are junk or corrupted. This is a good feature because Python will fail rather than try and provide invalid output from an encode or decode method. Therefore Python won't accept invalid input as valid.

Alternatively rather than raising an exception, we can pass a second parameter to the encode and decode functions. Some values for this second parameter are as follows:

|  |  |
| --- | --- |
| **Parameter Value** | **Parameter Description** |
| strict | The default value. Will raise an exception if it can't decode the value. |
| replace | Will return a “?” for every character that can't be decoded. For example:  “A python string?” |
| xmlcharrefreplace | Will return an HTML/XML character entity reference, so /u0400 become &#1024. (0x400 = 1024 decimal). |
| Ignore | Simply ignore and don't print out the value |

And now we come to the crux of the problem with the Unicode coding scheme in Python 2. The python 2 distribution contains a file located in /usr/local/python2.x/ called site.py (This is on Unix systems, check your reference documentation for the location of this file on Microsoft Windows based systems). When Python was first being created, the designers decided to set the default encoding scheme to 'ASCII'. We can see this as follows:

|  |
| --- |
| In [1]: import sys  In [2]: sys.getdefaultencoding()  Out[2]: 'ascii' |

When Python sees some code that looks like the following:

|  |
| --- |
| In [2]: string1 = u"Hello"  string2 = "World"  In [8]: type(string2)  Out[8]: str <-- Here string2 is a byte string  In [9]: type(string2.decode())  Out[9]: unicode <-- Here string2 has been coerced to a unicode string  In [4]: print string1 + " " + string2  Hello World  In [12]: print type(string1 + " " + string2)  <type 'unicode'> |

It will try to implicitly coerce the byte string (In this case string2) to a Unicode string. Because we have seen that the default encoding scheme is 'ASCII', it will attempt to use this coding scheme to encode or decode the byte string. This works when, in fact, the byte string is accepted to be ASCII by the creator, however, this will fail if in fact, a non-ASCII character is contained in the byte string. This is the cause of the Unicode{Encode|Decode}Exception. One of the most common types of programming patterns where this is encountered is in network client/server applications.

To be fair to the designers, in the year 2000 when the language was being designed, ASCII was the 'safe' choice. However, with the explosion of applications using non-ASCII unicode characters, this is no longer true.

### Can't we just change the defaultencoding parameter to 'UTF-8'?

Short answer: No.

Longer answer: Yes, you can, but it is severely contra-indicated.

Why not?

There is a workaround which will allow Python 2 programmers to set the default encoding value, however doing this create severe side effects in which, effectively, the cure becomes worse than the disease.

Problem number 1. Other programs besides yours may depend on this value being set to 'ascii'. The site.py is loaded once for the environment. The setdefaultencoding() method is not available by default. Creating this method in your program and invoking it will affect not only your program but also any third party programs, such as libraries which may depend on that value. You may well find that the program will work on your development system,but when rolling it out into production many other things will break which makes your application unviable.

Problem Number 2. Basic collections such as dictionaries may break when doing container lookups. This is because the hash values of a key that contains only ASCII values and one that contains non-ASCII values won't be the same due to the fact that the *in* operator doesn't automatically coerce type, so the in operator will not return the expected value whereas the '==' operator which will do the implicit coversion will return the expected value.

### How does Python 3 handle this problem?

Python 3 completely re-does the concept of byte and unicode strings. Let's take a look.

|  |
| --- |
| foo = "Hello World\u0400"  print (type(foo))  <class 'str'>  bar = b"Hello World\u400"  type(bar)  builtins.bytes |

Python 3 has now completely redefined the 'str' class. Python 3 treats 'str' as Unicode whereas Python 2 treated 'str' as a byte string.

Note that specifying the 'b' prefix before a string indicates to Python 3 that this is a byte string.

Python 3 now has a specific built in class for byte strings. Additionally Python3 will no longer do implicit type conversion for you. You must now explicitly covert byte strings to unicode and vice versa using the decode and encode methods. Because of this change, the chances of getting a UnicodeEncodeException or UnicodeDecodeException runtime error are substantially reduced. The penalty, however, is that you must now explicitly encode and decode your strings in your program.

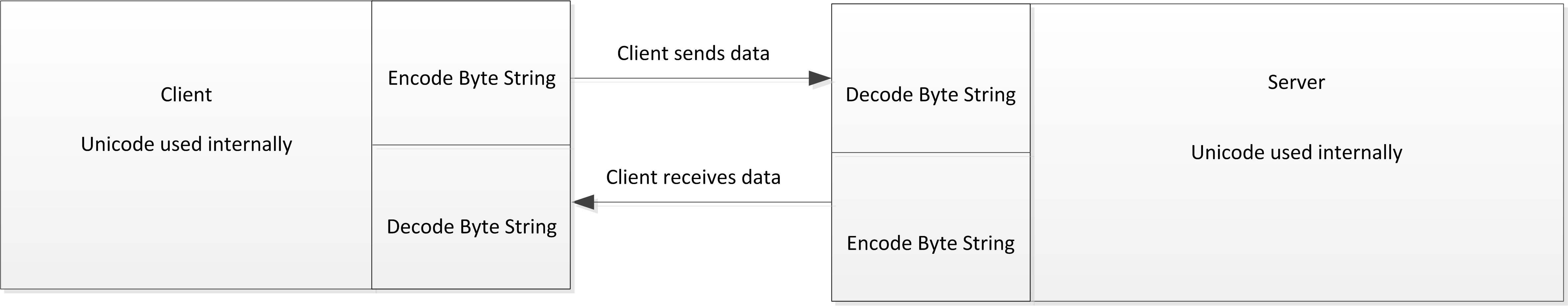
Let's see an example of this.

|  |
| --- |
| "Hello" + b" World" <--- No implicit conversion done here.  ---------------------------------------------------------------------------  TypeError Traceback (most recent call last)  <ipython-input-1-2395be9473f9> in <module>()  ----> 1 "Hello" + b" World"  TypeError: Can't convert 'bytes' object to str implicitly  "Hello " + b"World".decode('UTF-8') <--- Explicit conversion needed.  'Hello World' |

A summary of the python 2 vs python 3 differences with unicode can be compiled as follows:

|  |  |
| --- | --- |
| The str class | In Python 2 str is a byte string, in Python 3 it is a unicode string |
| The byte builtin class | Unique to Python 3 |
| Implicit type conversion | Yes in Python 2, no in Python 3 |

So, how do we actually use this in our applications? The best way to do this is to explicitly do the conversion to unicode immediately upon receiving the byte string. Use unicode strings internally, and then convert back to bytes when sending the data back. This can thought of as a “unicode sandwich”. The following diagram illustrates this.



Finally, it's important, when designing your application to keep the following ideas in mind.

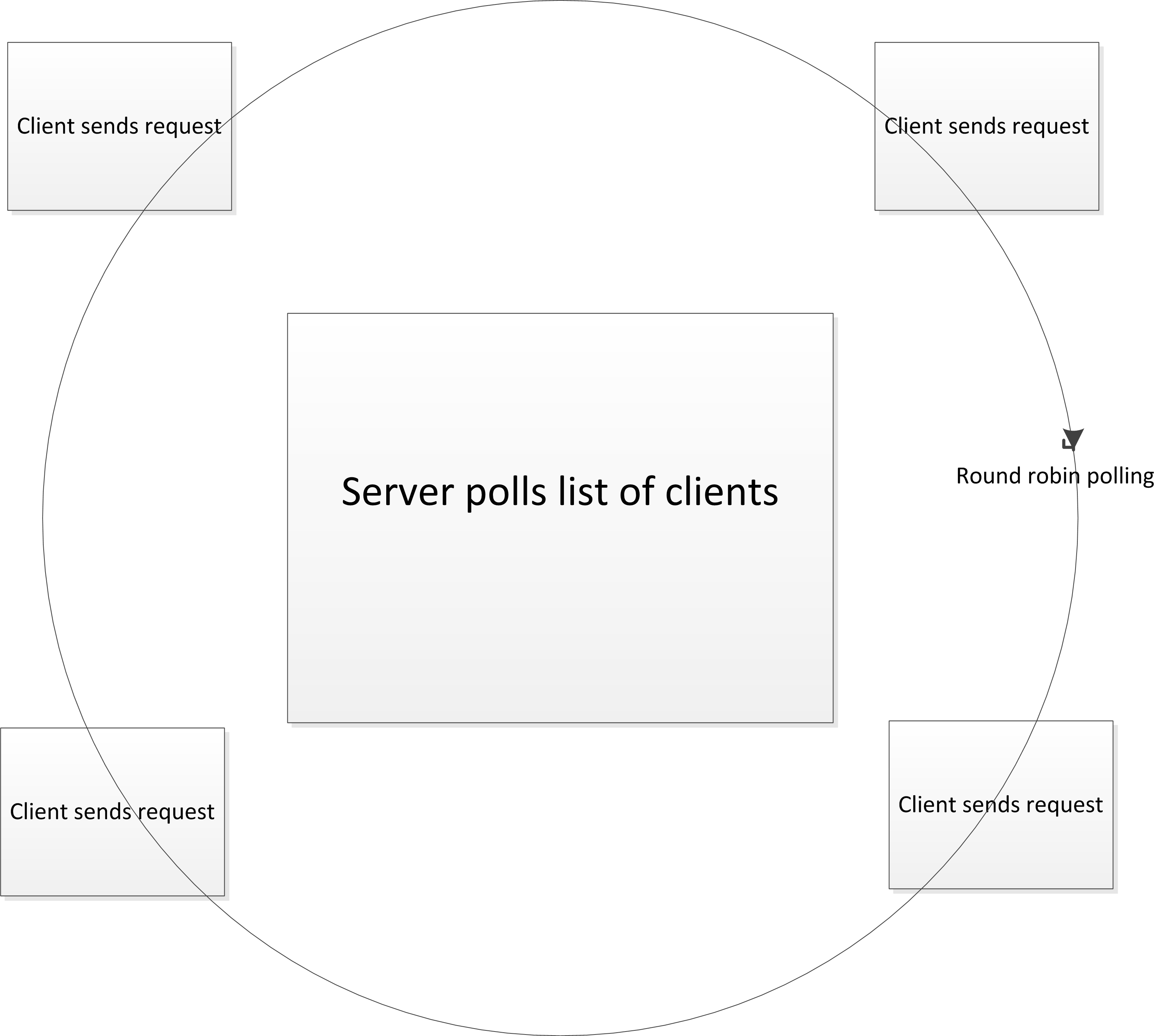
|  |
| --- |
| 1. Everything in a computer is stored as bytes. Bytes in and of themselves have no meaning. We need some sort of convention or standard in order to assign meaning to the bytes. |
| 2. English and the Latin alphabet set is no longer the universal medium of communication over the internet. |
| 3. Bytes and Unicode strings are both necessary and you must be able to keep track of both types of data. |
| 4. There is no way to look at a stream of bytes and infer what the encoding scheme. You have two choices,  A. Take a guess  B. Have someone tell you. |
| 5. Sometimes you get a bad steer. I.e. the encoding standard you choose is wrong, even if you've been told it's correct. |

There are three design pro-tips that you should consider.

|  |
| --- |
| 1. Convert data from bytes to unicode at the receiving/sending edge points. Use Unicode exclusively internally in your application. |
| 2. You need to know what type of data you have, never guess. Check the object type using the type() function and the repr() function to get a true representation of the string. |
| 3. Always test your application with a wide range of Unicode symbols, not just the standard ASCII or ISO-8859-1 set. |

### Asynchronous I/O multiplexing with select

A common use case for sockets is a single server connecting with multiple clients. There are multiple strategies to allow a server to efficiently communicate with more than one peer. One method is to *fork* a new process for every socket connection, however, this is very resource intensive and can take a prohibitive amount of time to create the new process, especially if the network load is severe. Another possibility is to create a new thread for each client connection using Python concurrency libraries. We will cover concurrency later in the course. A third way is to use the *select* library, which is based on the UNIX select system call. Select is a mechanism to allow the server to *poll* multiple clients in a round robin format waiting for a request to arrive and then handling it. Here's a conceptual diagram of how the select operation works.



Here we see that the server will poll each client to see if any new requests have come in. If so, then the select function will wake up and pass control back to the server to process the request.

Let's take a look at some sample code that implements this.

|  |
| --- |
| #!/usr/bin/python3  import socket  import select  import sys  server = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  server.setblocking(0)  server.bind((“localhost,10080))  server.listen(5)  # Here is the start of the code to implement polling with select  inputs=[server]  outputs=[]  # The select function polls on three lists, inputs, outputs and exceptional  # the inputs list has sockets and other devices to read from  # The outputs list has sockets and other devices to write to  # The exceptionals list will handle exceptions  while inputs:  # In this case, we'll send any exceptional handling error messages back to the  # client  readable,writeable,exceptional = select.select(inputs,outputs,inputs)  # Here the select has returned because a client has send some data to one of the sockets in the # # input list  for s in readable:  # It's either a new connection or a request from an existing connection. First we handle the  # case of a new connection.  if s is server:  connection,client\_address = s.accept()  # Make sure that the new connection doesn't block!  connection.setblocking(0)  # Get any data sent by the new connection and send an acknowledgement back.  data = connection.recv(1024)  sendString = “got “ + data.decode()  connection.send(sendString.encode())  # Add the new connection socket to the list of inputs that select will poll.  inputs.append(connection)  else:  # It's an existing connection, so get the data and send back an acknowledgement. However, if we # don't actually have any data, it's because the client has closed the connection on its end.  data = s.recv(1024)  if data:  sendString = “Got “ + data.decode()  s.send(sendString.encode())  else:  s.close() |

Thus we see that we can now handle connections and requests from multiple clients without having to either fork a new process or start a new thread. However, select has its weaknesses. The select system call was first proposed before the idea of having multithreaded connections serving thousands of clients was realized. With select we have to maintain a list of inputs and outputs and constantly poll them for incoming or outgoing data. This quickly becomes unmaintainable when designing an application that may serve tens or hundreds of thousands of requests from thousands of clients simultaneously. The select method will only support 1024 socket descriptors to poll on so its ability to scale is quite limited. Additionally, performance is generally much slower when polling across a list. Therefore two additional methods are available for us to use when designing an multiservice application.

These two methods are *poll* and *epoll*. Poll is a UNIX system call that is not supported on Windows versions pre-Vista, which includes Windows XP, therefore using it may negate portability. Epoll is only available on Linux systems with kernel releases of 2.544 or greater, therefore we won't be discussing it in this course specifically.

Let's re-write our select example using poll instead.

|  |
| --- |
| #!/usr/bin/python3  import socket  import select  import sys  server = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  server.setblocking(0)  TIMEOUT=1000  server.bind(("localhost",10080))  server.listen(5)  # These are the flags used by the poll system call. These are the conditions we're waiting  # on when an event happens.  READ\_ONLY = select.POLLIN | select.POLLPRI | select.POLLHUP | select.POLLERR  READ\_WRITE = READ\_ONLY | select.POLLOUT  # Create the poll object and register it to listen for the READ\_ONLY events.  poller = select.poll()  poller.register(server,READ\_ONLY)  # When poll returns an event, it returns the socket's file number. We need to map from that  # to the actual socket object, so let's create a dictionary to do that.  fd\_to\_socket = {server.fileno(): server,}  # Wait for an event to happen.  while True:  events = poller.poll(TIMEOUT)  for fd,flag in events:  # We've woken up from the poll due to some READ\_ONLY event happening. The poll call returns  # an event tuple which contains the file descriptor and the READ\_ONLY flag that woke it up.  # So let's first map the file descriptor to the socket.  s = fd\_to\_socket[fd]    # What type of event as it? If it was an input or a priority input, then let's handle that now.  if flag & (select.POLLIN | select.POLLPRI):  if s is server:  # The socket is the server socket, so it means that we have an incoming client trying to connect.  # Let's accept that, and register the new socket with the poll object so that it will now listen for  # events on that socket as well.  connection,client\_address = s.accept()  connection.setblocking(0)  fd\_to\_socket [ connection.fileno() ] = connection  poller.register(connection,READ\_ONLY)  else:  # It's not a new connection, it's a client sending us data.  data = s.recv(1024).decode()  if data:  sendString = "Got " + data  s.send(sendString.encode())  else:  # If the data is empty, then the client is closing the socket on its end so let's handle that as well.  poller.unregister(s)  s.close() |

Let's note the differences between the poll pattern and the select pattern. First of all we see that we no longer have to maintain lists of the input, output and exceptional file descriptors. This is now handled through the kernel. All we need to do is register the socket with the poller and it will now listen on that socket as well. What this means in practice is two-fold.

1. We are no longer limited to 1024 connections as we are with select.

2. We no longer have to iterate through the list and test each socket to see if something has come in. We leave that task to the kernel itself.

Poll is significantly faster than using select and is suggested if you're writing an application which uses the I/O multiplexing pattern.

### Abstracting I/O multiplexing with selectors in Python 3

Python 3.4 introduces a new abstraction level for I/O multiplexing with *selectors*. Selectors are defined using an abstract base class called *BaseSelector*. Concrete implementation of this ABC are listed in the documentation. Of particular interest is the *DefaultSelector* class. The DefaultSelector class is the class that implements the most efficient solution for the particular platform that you are running on. This means you no longer have to decide which of the implementations, such as select or poll you want to use. Simply using the DefaultSelector class will give you the most optimal choice. Let's take our previous examples with select and poll and re-write it using the selectors interface.

|  |
| --- |
| #!/usr/bin/python3.5  import socket  import selectors  def accept(s,mask):  conn,addr = s.accept()  conn.setblocking(False)  sel.register(conn,selectors.EVENT\_READ,read)  conn.send("Registered client connection!\n".encode('UTF-8'))  def read(conn,mask):  data = conn.recv(1024)  if data:  print ('Echoing',repr(data.decode('UTF-8')),'to', conn)  conn.send(data)  else:  print ('Closing connection')  sel.unregister(conn)  conn.close()    # Here we're using the default selector rather than specifically  # selecting poll, epoll, or select. Python will choose the most  # optimal implementation for us.  sel=selectors.DefaultSelector()  server = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  server.setblocking(0)  server.bind(("localhost",10080))  server.listen(5)  # Register this descriptor with the selector. We tell it that it's  # to notify us on any sort of read event and call the accept function  # when when a read event happens on this descriptor.  sel.register(server,selectors.EVENT\_READ,accept)  while True:  # Note that the select method is is an abstract method, it isn't necessarily  # related to the actual select() system call. It's just the method used for  # all implementations of the BaseSelector abstract class  # Events here is a reference to a SelectorKey class. This is a named  # tuple which consists of the following fields...  # The file object, i.e. in this case the registered socket.  # The file descriptor for this file object  # The events that we selected to wait on (The above code shows that  # We're waiting on READ events.  # A data field which, in this case, contains the callback function reference.  events = sel.select()  for key, mask in events:  # Callback is now a reference to our callback function (i.e. the accept()  # function in this case  callback = key.data  callback(key.fileobj,mask) |

Note that we no longer have to implement semantics for different types of I/O multiplexing implementations. No lists of file descriptors for the select are needed nor is having to convert the file descriptor to a socket. Simply pick a selector implementation, register the socket (or other file descriptor) and wait for events to happen.

# Module 3. Multithreading in Python

What is threading? Threading is similar to concurrent execution of programs. In concurrent execution, a process can create one or more children processes by using the POSIX system calls fork() and exec() system calls. This,however, tends to be very resource-intensive and can affect performance.

Another way to do this is to create threads. Threads are also called light weight processes. They run in the same context as process creating it, i.e. they share the same resources, although they can have their own local variables. A multithreaded application, if designed properly can be substantially faster than one that is concurrent or runs as a single process. Unfortunately for us, the reference implementation of Python, cpython, does not allow CPU based multithreading. I.e. the multicores available on standard modern Intel/AMD processors cannot be accessed from Python due to the Global Interpreter Lock (GIL) in all current versions of cpython. Note that other implementations, such as jython and IronPython, do not suffer from this problem. Even though we can't use multithreading for CPU based computation, we can still use it for I/O processing.

|  |
| --- |
| #!/usr/bin/python  import thread  import time  # Define a function for the thread  def print\_time( threadName, delay):  count = 0  while count < 5:  time.sleep(delay)  count += 1  print "%s: %s" % ( threadName, time.ctime(time.time()) )  # Create two threads as follows  try:  thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )  thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )  except:  print "Error: unable to start thread"  while 1:  pass |

Let's analyze the above code. We note that we have to import the thread library first. We then create a function that will run inside the thread, print\_time().

Each thread will run an instance of this function separately. We then create the thread with the start\_new\_thread() method, passing the actual function to run and any arguments to that function.

This is an example of the low-level thread module. However as of Python 2.4, a new library threading has been introduced and is now more commonly used as it abstracts much of the threading functionality away.

Let's re-write the first threading example using the newer threading library.

|  |
| --- |
| #!/usr/bin/python  import threading  import time  exitFlag =0  class myThread (threading.Thread):  def \_\_init\_\_(self, threadID, name, counter):  threading.Thread.\_\_init\_\_(self)  self.threadID = threadID  self.name = name  self.counter = counter  def run(self):  print "Starting " + self.name  print\_time(self.name, self.counter, 5)  print "Exiting " + self.name  def print\_time(threadName, delay, counter):  while counter:  if exitFlag:  time.sleep(delay)  print "%s: %s" % (threadName, time.ctime(time.time()))  counter -= 1  # Create new threads  thread1 = myThread(1, "Thread-1", 1)  thread2 = myThread(2, "Thread-2", 2)  # Start new Threads  thread1.start()  thread2.start()  print "Exiting Main Thread" |

Notice that the code is a bit more complex. Here we create a new class myThread which a derived class of the Thread class defined in the threading module. The first thing that the \_\_init\_\_() method does is call the superclasses init module. It also defines a run() method that will call the actual function that does the work. Calling the start method from the thread object will automatically invoke the run method inside the thread object.

Note that in the above coding examples, none of the threads actually change any variables. When a thread changes a variable, it is possible to hit a race condition. This is where multiple threads depend on the value of that variable. It is possible that one thread changes the variable value that will affect the behavior of other thread while it is executing.In order to prevent this, we use the concept of a lock. A lock is also called a mutex. In order to use a lock, we need to do the following:

create the lock object like so:

|  |
| --- |
| myLock = threading.Lock() |

Then, when changing a variable, we do the following:

|  |
| --- |
| myLock.acquire()  critical\_variable = 5  myLock.release() |

Note that in between the acquire and the release, only the thread that holds the lock can change the value of critical\_variable. Any other thread attempting to get the lock will block until the lock is released by the first thread. Note, that if the first thread doesn't release the lock due to a programmer error, all of the threads will eventually block and the program will probably hang.

# Module 4. The Python3 Collections Library

An important module in the Python libraries is the *Collections* library. This library defines a number of useful data structures for implementing python applications. Let's take a tour through this module.

### 1. The Chainmap

Chainmaps are new to Python 3. A chainmap provides the ability to take one or more (usually more) dictionaries and combine then into one large dictionary that can be searched on or iterated over.

Let's see an example of this:

|  |
| --- |
| #!/usr/bin/python3  from collections import ChainMap  # Here are three separate dictionaries that contain a car part name  # and a part number.  car\_parts ={'hood':'1P994','engine':'2X2001','front\_door':'1Y8884'}  car\_options={'air\_conditioning':'9B0003','Turbo':'1D9110','rollbar':'5Z0123'}  car\_accessories = {'Cover':'4T1413','hood\_ornament':'5N0512','seat\_cover':'7C0316'}  # Here we use the chainmap to combine the three dictionaries into one.  car\_manifest = ChainMap(car\_parts,car\_options,car\_accessories)  print (car\_manifest['engine'])  car\_manifest['Cover'] = '4T1414'  print (car\_manifest['Cover'])  print (car\_manifest.maps[0]) |

The output of this program is:

|  |
| --- |
| 2X2001  4T1414 |

Why do this instead of creating one large dictionary? There may be reasons in your application to keep the dictionaries separate for other purposes. For example, if you often do iterations over the car parts dictionary, it makes sense to keep it as small as possible for performance and resource reasons.

A couple of notes on ChainMap. The ChainMap map attribute is a list of the maps in the chain. I.e. in this case car\_manifest.maps[0] references the car\_parts dictionary. The new\_child method allows the creation of a new ChainMap containing a new map at the front of the list, followed by all of the other maps in the list. It defaults to an empty map at the start of the maps list if no map is given.

### 2. Counter

This is one of the most useful components of the Collections library. A counter is a subclass of a dictionary. Counter's maintain a key value pair where the key is the key item and the value is the number of these keys found. For example:

|  |
| --- |
| #!/usr/bin/python3  from collections import Counter  wordList = ['foo','foo','bar','bar','baz','blech','foo','foo','bar','meh','feh','feh']  wordCount = Counter(wordList)  # Print the count of key values  print (wordCount)  # Print the top two keys  print (wordCount.most\_common(2)) |

The output of this program is:

Counter({'foo': 4, 'bar': 3, 'feh': 2, 'baz': 1, 'blech': 1, 'meh': 1})

[('foo', 4), ('bar', 3)]

As we can see, the wordCount variable contains a dictionary which has the elements of the wordList list as the keys and the number of times it finds each key in the list as it's value. We can also call methods such as most\_common(n) which gives back the n most common elements in the list. There are many use cases for using the Counter dictionary such as finding the most common word or words in a set (such as a book).

### 3. The defaultdict

If we have a dictionary *d* and I try and get or set a key value pair where the key doesn't exist, Python will raise a KeyErrorException. This may not be the behavior you desire. One way around this is to declare a default value for the key. For example, if we have an empty dictionary d and I want to print a default value for the key 'foo' so that it doesn't throw an exception, We can do the following : d.setdefault['foo',0]. Now, everytime we try to print d['foo'], if the 'foo' key hasn't been set for d, then it will print a 0. However, this method, for a number of reasons that we will see in the next section, is somewhat cumbersome. For one thing, we'd have to call the setdefault() method for every key in the dictionary. For large dictionaries this becomes unsuitable. However, the collections library allows us to have a sub class of dictionary called a default dictionary. This allows us to set the value of any undefined keys just once. For example:

|  |
| --- |
| #!/usr/bin/python3  from collections import defaultdict  d = {}  # Here we use the setdefault method from the python dictionary to  # set the default value for the key 'foo' in dictionary d to 0.  # However, the problem here is that we need to do this for every key  # that we expect to be able to insert if we don't want to get a  # KeyErrorException thrown when accessing d['foo'] if it hasn't  # already got a value.  d.setdefault('foo',0)  print (d['foo'])  # Better way. Make d a defaultdict. Note that we have to pass into  # the constructor a callable object. You can look in the Python  # documentation for built in functions to find the callable built in  # objects (or create and pass in your own callable object). Passing  # an int object automatically means that the defaultdict will set a value  # of 0 for any undefined key/value pairs.  d = defaultdict(int)  # Now, everytime we try to print out a key that doesn't exist in the  # dictionary, we get a 0 printed instead of an exception thrown.  print (d['foo']) |

So the output from this program is 0 as d['foo'] was not set before being used.

### 4. The deque

A deque (pronounced 'deck') is a short hand term for a double ended queue. In a double ended queue, items can be added to or removed from either end of the queue. The main advantage of a deque is that the time it takes to insert or delete values from either end of the queue is significantly less than using a standard python list. Here's an example of this:

|  |
| --- |
| In [1]: from collections import deque  In [2]: s=range(10000) <-- Setup a list with 10,000 elements  In [3]: d=deque(s) <-- Create a deque using the list as input  In [4]: s\_append,s\_pop= s.append, s.pop  In [5]: d\_append, d\_pop = d.append, d.pop  In [6]: %timeit s\_pop(); s\_append(None)  10000000 loops, best of 3: 113 ns per loop <-- Timings using a standard python list  In [7]: %timeit d\_pop();d\_append(None)  10000000 loops, best of 3: 88.3 ns per loop <-- Timings using a deque |

Note that there is a performance improvement of greater than 20 per cent for using deques rather than lists when creating data structures where you consistently add or delete elements from either end. Another singular advantage of deques over lists is the ability to *rotate* them in either direction. For example:

|  |
| --- |
| In [1]: from collections import deque  In [2]: dq = deque([1,2,3,4,5])  In [3]: dq.rotate(2)  In [4]: print (dq)  deque([4, 5, 1, 2, 3])  In [5]: dq.rotate(-3)  In [6]: print (dq)  deque([2, 3, 4, 5, 1]) |

Note that providing a positive number to the rotate() method rotates the deque to the right whereas a negative number rotates the deque to the left.

Additionally dequeues are *thread-safe* which means that they can be safely used in multithreaded applications.

### 5. The named tuple.

Tuples are one of the most widely used data structures available in Python. While tuples are useful for many applications, they do have one drawback. In the case of tuples with many elements, the only way to access each element is through a numerical index. This can lead to code that is difficult to read and maintain and can lead to programmatic errors in extreme cases. The collections library in Python offers a class called a *named tuple*. Simply put, this is a standard tuple in which each element can now be accessed by field name rather than just by index. Let's see an example of this.

|  |
| --- |
| #!/usr/bin/python3  from collections import namedtuple  # Here we create a standard tuple of personnel information that contains  # the following fields, first name, last name, Employee ID and Address  standardtuple = ('Braun','Brelin','12345','1234 Main Street')  # With a standard tuple, if I want to access the first and last name, I need to  # specify it with the index value of the tuple, i.e. first name is index  # position 0, last name is index position 1.  print (standardtuple[0] + " "+ standardtuple[1])  # Let's see how to do the same thing with a named tuple.  # We use the named tuple factory to create a named tuple type called 'Person'  # The first argument is the named tuple's name, the second argument is the  # list of fields  Person = namedtuple('Person','firstname lastname ID Address')  # Now we can use this named tuple type to create Persons. Such as:  Braun = Person('Braun','Brelin','12345','1234 Main Street')  # Note here that we can now access elements by field name, rather than  # index position  print (Braun.firstname + " " + Braun.lastname)  # We can create a person object by using the \_make method and passing some  # sort of iterable data structure such as a list.  Bob=Person('Bob','Bird','12346','2345 Main Street')  print (Bob.firstname + " " + Bob.lastname) |

Note that not only can we create the named tuple directly, but we can also call the \_make() method and pass it an iterable data structure such as a list into it and create the tuple that way. There are a number of other methods contained in the Python documentation which lists some of the other methods available for a named tuple.

There are a number of other containers available in the collections library, however, the ones discussed above are the most useful and commonly used data structures in the Python environment.

# Module 5. Writing idiomatic Python.

Note. This chapter was inspired by a talk given by Raymond Hettinger, a Python core developer, at the 2013 Pycon conference. The talk can be found [here](https://www.youtube.com/watch?v=OSGv2VnC0go) on Youtube. It is highly recommended that students watch this video in order to increase their knowledge of Python.

This purpose of this chapter is to give students some guidance on the 'proper' way to write Python code. Unlike languages such as Perl, whose design philosophy is 'there is more than one way to do things', Python assumes that there is a proper 'Pythonistic' way to write your code. Herein are some tips students should know when writing Python applications.

### 1. Iterating over a list of numbers

A common way to iterate over a list is as follows:

|  |
| --- |
| for i in [1,2,3,4,5]:  do\_some\_statements\_here |

Alternatively, in Python 2, you could do the following:

|  |
| --- |
| for in in range(1,6):  do\_some\_statements\_here |

However, in Python 2 the second statement can be problematic. If I do a range between 1 and one million, Python 2 will allocate a list for a million entries. This can take up to 32 Megabytes on a 64 bit machine, with a corresponding waste of resources. Python 2 offers the xrange() function, which returns an iterator rather the actual list. In this instance, xrange will only allocate memory as necessary rather than all at once. Python 3 replaces the range function with the xrange function (but still calls it range).

### 2. Iterating over a collection.

Given a collection such as the following:

|  |
| --- |
| colors = ['red','green','blue','yellow','purple'] |

How would we iterate over this? People who come from the C/C++ world might do something like this:

|  |
| --- |
| for i in range(len(colors)):  print (colors[i]) |

The better way to iterate over a collection would be to do this:

|  |
| --- |
| for color in colors:  print (colors) |

Note that the for loop in Python is really a *foreach* loop. Although you can emulate anumeric for a la C or Java, it's not the 'Pythonic' way to do things. Also, using the timeit option in the Python REPL shows that iterating over a list using the second method is significantly faster.

Similarly we can also do a reverse iteration through a collection. The non-pythonic way to do this might be a follows:

|  |
| --- |
| colors = [“red”,”green”,”blue”,”yellow”,”purple”]  for i in range(len(colors)-1, -1, -1):  print (colors[i]) |

A better approach would be to do the following:

|  |
| --- |
| for color in reversed(colors):  print (color) |

Again, this is not only more readable, it execute faster as well.

Another common technique is to loop over the index and the collection like so:

|  |
| --- |
| colors = ['red','green','blue','yellow','purple']  for i in range(len(colors)):  print (i, “ = “ , colors[i]) |

A better, more pythonic way to do this:

|  |
| --- |
| For i,color in enumerate(colors):  print (i , “ = “,color) |

The enumerate builtin function returns an enumerate object, which can be converted into an iterable such as a list of tuples of which the first element in the tuple is the iterable index position and the second element is the actual iterable element.

### Looping over multiple collections.

When looping over multiple conditions, there are a number of ways to do this in Python. Here we present three options ranging from the least to the most efficient.

|  |
| --- |
| colors = ['red', 'green','blue','yellow','purple']  names = ['Braun','Guido','Bob','Sue']  # Option 1. Least efficient  n = min(len(names),len(colors)) # Find the smaller length of the two lists.  for i in range(n):  print (names[i],” = “, colors[i])  # Option 2, Less inefficient  for name,color in zip(colors,names):  print (name, “ = “ , color)  # Option 3, Most efficient  for name,color in izip(colors,names):  print (name, “ = “, color) |

Let's examine the three options. Option 1 is the non-pythonic way to do it. It's again emulating a numeric for loop and looping over the smallest list and mapping it to the larger list.

Option 2 uses the zip() builtin function. Zip takes the first element of the first list and the first element of the second list and builds a third list containing, as each element, a tuple of the first and second list elements). It then repeats this for every element until it reaches the end of the shorter of the two lists. While this is better than option 1, zip still has a problem. It creates, in memory, a third collection which is larger than the two source collections combined. For large ccollections, this can mean that you suffer CPU cache misses which can cause significant performance slowdowns. Note that this only applies to Python 2. The Python 3 version replaces zip with izip.

Option 3 is the optimal pattern. Instead of using zip, it uses izip, which creates an iterator object rather preallocating memory to the target container itself. Again, in Python3, izip is no longer used, instead, the standard builtin zip function is replaced by izip. The advantage here is that only the memory actually needed in order to hold the container information is allocated at run-time.

### Sorting container elements with Python

The easiest way to loop over an unordered container in sorted order is the following:

|  |
| --- |
| colors = ['red','green','blue','yellow','purple']  for color in sorted(colors):  print (color)  # Sorts in reverse order.  for color in sorted(colors, reverse = True):  print (color) |

There are times, however, when you want to create a custom comparator in order to perform container sorts. The traditional way to do this was to create a comparator function and pass this to the sorted function like this:

|  |
| --- |
| colors = ['red','blue','green','yellow','purple']  def compare\_length(c1,c2):  if len(c1) < len(c2):  return -1  if len(c1) > len(c2):  return 1  return 0  print sorted(colors,cmp=compare\_length) |

This is not only not idiomatic, it is slow. For any container, the number of calls to the compare\_length function is (n \* logn). In other words for an container with a million elements, the log base 2 of 1000000 is 20. Therefore 20 million calls are made to the compare\_length() function.

The better way to do this is to use a key function like so:

|  |
| --- |
| colors = ['red','blue','green','yellow','purple']  print sorted(colors,key=len) |

A key function is one that takes one parameter (usually the container element) and returns a value to base the sorting on.

In this scenario, the key function *len()* only gets called once for each element. So, it is significantly faster than a custom comparator. Remember that len is a builtin function in Python that returns the length of an item such as a string, a tuple or a list. In Python 3 the custom comparator component has been removed from the sorted builtin function.

### Using the iter builtin function to iterate over non-iterable objects

A common pattern in Python is to iterate over something until a *sentinel* value is reached. Here is an example of this:

|  |
| --- |
| buffer = []  while True:  block = f.read(32) <-- f is defined earlier as the return value from an open()  if block = ' ':  break  buffer.append(block) |

In this code snippet, we iterate over a file object and read data from it until we come to the end of the file. Here is a better way to do it.

|  |
| --- |
| from functools import partial  buffer= []  for block in iter(partial(f.read,32), '')  buffer.append(block) |

Let's analyze this code. The iter() builtin function behaves differently depending on how many parameters are passed to it. If only one parameter is passed, then that parameter must be an iterable object. That is, it must have a \_\_next\_\_() and a \_\_hasNext\_\_() method defined for that container. However, if two parameters are passed to it, the first parameter must be callable. In other words, the passed object must implement the \_\_call() method. The second parameter is the sentinel value, so that when this value is returned from the call, the iter method will stop. The partial function is imported from the functools library. It creates a callable object that when called, will call the f.read() method. Note that this code is substantially shorter than the first method. It is also faster.

### Defining multiple exits from loops using the else clause

Let us take a look at a common pattern in programming.

|  |
| --- |
| colors = ['red','green','blue','yellow','purple']  foundRed == False  for color in colors:  if color == 'red':  foundRed = True  break  if not foundRed:  print (“There is no color red in the list”) |

In this case, we notice that there are two possible exits from the loop. [I](mailto:I@n)n case one, we find an element called 'red' and immediately exit from the loop. In case two, we don't find a 'red' element and exit from the loop normally. We would then like to do some operation or operations depending on whether we exited the loop normally or via the break statement. The old way of doing this is to set some sort of a flag value, in this case *foundRed*. We set the foundRed flag to true if we found a 'red' element and then break. Python, however, gives us a much cleaner way to do this:

|  |
| --- |
| colors = ['red','green','blue','yellow','purple']  for color in colors:  if color == 'red':  break  else:  print (“There is no color red in the list”) |

Notice the else clause attached to the for loop. The else clause will only run if the loop as exited normally. It will not run if we broke out of the loop for any reason.

### Looping over dictionary keys

There are two major ways to loop over a dictionary.

|  |
| --- |
| d = {'foo':'bar','fee':'fi','fo':'fum'}  for k in d:  print (d[k])  for k in d.keys():  print (d[k])  del d[k] |

What's the difference between the two? In the second example, d.keys() creates a new list of the keys which you can then use to mutate (i.e. change) the dictionary. Attempting to change the dictionary using the first example will throw an exception. Note also that the foreach behavior when iterating through a dictionary is to only return the keys by default. Attempting to create a tuple of key,value in the for loop will also throw an exeception.

Let's see how to get both the keys and values from a dictionary when looping over it.

|  |
| --- |
| # First way  for k in d:  print (k, “=”,d[k])  # Second way  for k,v in d.items():  print (k, “=”,v) |

Which way is better? Clearly the second way is faster, in the first example the dictionary hash has to be recalculated for every item in the dictionary (The d[k] operation). In the second way, d.items() creates an iterator which is only created one. We can then iterate over the object. Note that in Python2, the items() method created a new list, which, although better than the first way, still created a list in memory. To emulate the iterator in python 2, use d.iteritems(). Python 3 creates iterators by default.

### Constructing a dictionary from a pair of lists.

What is the best way to construct a dictionary from a pair of lists where one list contains the keys and the other one contains the values?

|  |
| --- |
| keys = ['fee','fi','fo','fum']  values = ['blood','of','an','Englishman']  d = dict(zip(keys,values)) |

Here, we see that we can create a dictionary using the zip function which returns an iterator of tuples. Note that this is highly efficient because the iterator only has to allocate space for one tuple which is re-used for each operation rather than allocating space for a new tuple each time. In Python 2, you would use the izip builtin function.

### Using a dictionary to count the number of elements.

Let's look at a number of ways to perform this function. Often times we will be given a list of potentially repeatable values and we will need to count the number of times that we receive any arbitrary value. For example if we have a list of values:

|  |
| --- |
| mynumbers = [1,2,4,2,3,6,4,3,3,3,7,8,6,9,9,0,0,1] |

How do we count the number of “1's”, “2's”, etc..

Let's look at three ways to do this:

|  |
| --- |
| # First way  d = {}  for number in mynumbers:  if number not in d:  d[number] = 0  d[number] +=1  # Second way  d={}  for number in numbers:  d[number]= d.get(number,0) +1  # Third way  d = defaultdict(int)  for number in mynumbers:  d[number] +=1 |

The first method for counting items in a dictionary is the standard, default way of doing it. Very simply, the program checks to see if the key exists, if it doesn't create a key with a value of 0 and then increments it. The whole point being that we want to avoid raising a KeyErrorException and halting program execution. The second way effectively does the same thing as the first method, however, here we use the *get()*  method to either retrieve the value if it exists or a zero if it doesn't. Get will then create the key with a value of zero in the dictionary. The last method is to use the defaultdict class which is described in the previous module. Note that there may be some use cases where we will have to convert the defaultdict back into a regular dict with the *dict()*  built in function.

### Grouping with dictionaries.

A common programming pattern is to take a list of values and put them into a dictionary by some characteristic of the value, e.g. the first character of the string value or its length.

Here is the naive, first implementation of this pattern.

|  |
| --- |
| names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']  d = {}  for name in names:  key = len(name)  if key not in d:  d[key] = []  d[key].append(name)  {3: ['Bob', 'Sue', 'Ben'], 4: ['Dave', 'Mark', 'Rory'], 5: ['Braun']} |

In this first implementation, we create an empty dictionary *d*. We iterate through the names list and make the length of each element of the list a key for the dictionary. If the key/value pair doesn't exist, create the key and initialize the value as an empty list. Once that's done, we append the list element as a new element of the value list. Note that in order to change the grouping criteria here, we only need to change one line of code, in this case, the key assignment.

Let's see a nicer way of implementing this pattern.

|  |
| --- |
| names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']  d = {}  for name in names:  key = len(name)  d.setdefault(key,[]).append(name) |

Here we use the *setdefault()* method to add the key value pair. If the key doesn't exist, the setdefault method will create an empty list as the value and add the key/value pair to the dictionary. Once that's done, we then append the new name element to the value list.

Let's see the up-to-date modern way using defaultdict to implement grouping by dictionary.

|  |
| --- |
| names = ['Braun','Bob','Sue','Dave','Ben','Mark','Rory']  d=defaultdict(list)  for name in names:  key = len(name)  d[key].append(name) |

As we saw in our last example, we can use a defaultdict from the collections library to implement these patterns. We pass in a list parameter to the defaultdict factory to tell it that for this defaultdict, if the key/value pair doesn't exist create an empty list as the value and insert the key/value pair into the defaultdict.

### Simultaneous state updates with tuple packing and unpacking.

Consider the following code to calculate the fibonacci sequence.

|  |
| --- |
| def fibonacci(n):  x = 0  y = 0  for i in range(n):  print x  t = y  y = x + y  x = t |

What's wrong with this code fragment? For one things, the state of both x and y can become inconsistent. Here we store the value of y into a temporary variable t, then we update y's state. Now, however, y has it's new state and x still has the old state. In other words, the state changes are not atomic. Additionally if the ordering of state changes is misapplied, then the code will run incorrectly and cause faults in the application. What's a better way to write this?

|  |
| --- |
| def fibonacci(n):  x,y=0,1  for i in range(n):  x,y = y, x + y |

Now we note that instead of having to break down the transformation into details, we simply say that we want to update x and y according to the relevant equations. Additionally, state changes are now atomic.

Finally, some tips on writing better Python code.

1. Don't put too much code on one line.

2. Don't break atoms of thought into subatomic particles.

While these rules sound conflicting, they're really not. Write your python code logic as if you're writing it in english. As we saw with the fibonacci example, don't break down the code into individual steps, this becomes difficult to code and maintain, additionally, you are no longer thinking in “higher order” terms but rather are bogged down in details. Additionally, don't try and pack multiple thoughts into one line of code as that also causes maintainability problems.

### To PEP 8 and beyond.

Python Enhancement Proposals (PEP) are documents that are written by developers to describe potential enhancements to the Python language. Some of these are accepted by the core developers for inclusion into new releases of the language. One of the most well known PEP's is PEP 8. PEP 8 is the style guide for writing Python. While detailed analysis of this document is best left to the student, let's see how we can use PEP 8 to clean up some code and even go beyond the style guide.

Let's consider the following program.

|  |
| --- |
| import jnettool.tools.elements.NetworkElement, \  jnettool.tools.Routing, \  jnettool.tools.RouteInspector  ne=jnettool.tools.elements.NetworkElement( '171.0.2.45' )  try:  routing\_table=ne.getRoutingTable() # fetch routing table  except jnettool.tools.elements.MissingVar:  # Record table fault  logging.exception('''No routing table found ''')  # Undo partial changes  ne.cleanup('''rollback''')  else:  num\_routes-routing\_table.getSize() # determine table size  for RToffset in range( num\_routes ) :  route=routing\_table.getRouteByIndex(RToffset)  name=route.getName() # route name  ipaddr = route.getIPAddr() # ip address  print ("%15s -> %s" % \*name,ipaddr) # format nicely  finally:  ns.cleanup( '''commit''' ) #lock in changes  ns.disconnect() |

This program connects to a network routing device, iterates over its routing table and prints out the route name and IP address as a report. Note that the library was written in Java and we're using Python to connect to it.

How can we use PEP 8 to make this code better? Let's look at the next version of this with PEP 8 applied.

|  |
| --- |
| #!/usr/bin/python3  # Put the import statements on separate lines for readability  # Also, it's nice to alphabetize them if you have a lot  # of imports.  import jnettool.tools.elements.NetworkElement  import jnettool.tools.Routing  import jnettool.tools.RouteInspector  # Put spaces around the '=' sign.  # Also, delete the space between the parameter and the parenthesis.  ne = jnettool.tools.elements.NetworkElement('171.0.2.45')  # Tighten up the code by removing the spacing between the try/except/else  # /finally blocks. Also, get rid of useless comments in the code.  # Get rid of the ''' double quotes. Just use the single quotes.  try:  routing\_table = ne.getRoutingTable()  except jnettool.tools.elements.MissingVar:  logging.exception('No routing table found')  ne.cleanup('''rollback''')  else:  num\_routes-routing\_table.getSize()  for RToffset in range(num\_routes) :  route=routing\_table.getRouteByIndex(RToffset)  name=route.getName()  ipaddr = route.getIPAddr()  # Put a space between function arguments.  print ("%15s -> %s" % \*name, ipaddr)  finally:  ns.cleanup('commit')  ns.disconnect() |

This code now conforms to PEP 8 standards. It's much clearer and easier to read. However, this code, while PEP8 conformant, isn't really Pythonic. Let's now take a look at the next program.

|  |
| --- |
| #!/usr/bin/python3  from nettools import NetworkElement  with NetworkElement('171.0.2.45') as ne:  for route in ne.routing\_table:  print ("%15S -> %s" % (route.name, route.ipaddr) |

This code does exactly the same thing as the first two programs, however note how much shorter it is. It does have one drawback, however. This code as it stands doesn't work. It doesn't work because the API library is written in Java and the Python bindings as written doesn't allow for Pythonic code. For example, it requires that the imports use dotted notation for packages. While this may be the correct way to do things in Java, in Python it's use is contra-indicated. Packages are used to make sure that namespace collisions don't happen. If you don't have a very large code base, then it is pointless to create packages and sub-packages. Also note that we're now abstracting away the setup and teardown code from the business logic of the program. The business logic is to print out a routing table with two columns, a route name and a route IP address. The setup and teardown code should be moved aside into its own module.

How do we then make this application more Pythonic?

Let's examine the options.

1. Any time we have reoccurring set up and tear down code, create a context manager so that we can use a with statement in Python to abstract away this code from the business logic.

2. Take multiple packages and combine them into a single module for ease of readability and maintenance.

3. Where we see code that uses a method to get the size of a list, loops over a range of elements and calls a function to get something by index, we do the following:

A. Instead of calling a getSize() method, use the builtin *len()*  function in Python.

B. Instead of calling a getValueByIndex() method, use the '[]' operator in Python.

C. In Python, anything where you can access an element by the [] operator and get it's

length via the len() method, is known as a *sequence*. One of the properties of a

sequence is that it is *iterable.*  This means that we can loop directly over the sequence

in Python.

D. Anytime you see getter and setter methods in Python, replace them with properties or

descriptors.

E. Create custom exceptions to give clear and understandable error messages back to the

clients.

F. To generalize on point *C*, use the builtin methods in Python, such as \_\_len\_\_ and

\_\_getitem\_\_ rather than custom methods to perform the same work.

G. Create your own \_\_repr\_\_ methods for debugging purposes.

Let's now look at an example of code that implements these options.

|  |
| --- |
| #!/usr/bin/python3  class NetworkElementError(Exeception):  pass  class NetworkElement(object):  def \_\_init\_\_(self,ipaddr):  self.ipaddr = ipaddr  self.oldne = jnettool.tools.elements.NetworkElement(ipaddr)  @property  def routing\_table(self):  try:  return RoutingTable(self.oldne.getRoutingTable())  except jnettool.tools.element.MissingVar:  raise NetworkElementError('No routing table found')  def \_\_enter\_\_(self):  return (self)  def \_\_exit\_\_(self,exctype, excinst, exctb):  if exctype == NetworkElementError:  logging.exception('No routing table found')  self.oldne.cleanup('rollback')  else:  self.oldne.cleanup('commit')  self.oldne.disconnect()  def \_\_repr\_\_(self):  return ('%s(%r)' % (self.\_\_class\_\_.\_\_name\_\_, self.ipaddr)  class RoutingTable(object):    def\_\_init\_\_(self,oldrt):  self.oldrt = oldrt  def \_\_len\_\_(self):  return self.oldrt.getSize()  def \_\_getitem\_\_(self, index):  if index >= len(self):  raise IndexError  return Route(self,oldrt.getRouteByIndex(index))  class Route(object):    def \_\_init\_\_(self, old\_route):  self.old\_route = old\_route  @property  def name(self):  return self.old\_route.getName()  @property  def ipaddr(self):  return (self.old\_route.getIPAddr() |

This code is an implementation of the *Adapter* design pattern. The idea here is that we wrap the API in a new class that allows us to write the re-usable, pythonic code shown in example 3 above.

We start with the creation of a NetworkElementError class which derives from the Base Exception class. We do this mainly because we can now call our exception something useful.

We next create our Network Element class. The *\_\_enter\_\_()* and *\_\_exit\_\_()* methods are used to create a context manager. This is the set up and tear down code that is run when the with statement is executed. Note that the \_\_exit\_\_ routine now defines the tear down logic. If an exception is passed then we do the logging and rollback, otherwise we do the commit. Finally, we disconnect from the router. Note that this code is now reusable. Anytime we write Python code that opens the router, we merely have to use the with statement. We no longer have to re-write the setup/teardown routines in our business logic. We then define the Routing Table. Note that since we've defined the \_\_getitem\_\_ and \_\_len\_\_ methods ourselves, we can now directly iterate over the ne.routingtable iterable. Also note that each call to \_\_getitem\_\_ returns a route object. Finally for each route object, instead of using getter methods for the name and the IP address, we can define properties for the attributes so that we can simply instead call route.name and route.ipaddr directly, even though under the covers we're actually calling the getter methods.

We've now created an adapter class that can be re-used everywhere it's needed. We've separated the business logic from the infrastructure logic so that the code is far cleaner, shorter, more readable and more maintainable.

# Module 6. Python bindings to C libraries

While Python is a wonderful language for writing applications, it is certainly slower than natively compiled code such as the C Programming language. There are times when we'll want to call functions that are written in C for both performance and accesibility reasons. We'll look at a python library called *ctypes* which we will use to call libraries and functions written in C.

This methods uses Python's *Foreign Function Interface* library to allow us to directly call functions from a C shared library (In Windows, this is called a dynamic link library).

### Using Ctypes

Let's look at an example. Here we have created a C shared library called *libmymath.so*. This library contains a number of computationally expensive mathematics formulas, including the fibonacci sequence calculation. Let's take a look at a Python program that does two things.

1. It calculates the fibonacci sequence on it's own in Python.

2. It calls the libmymath.so file, written in C to do the same calculation.

It also benchmarks the two functions using the *timeit* module.

First, let's look at the code.

|  |
| --- |
| #!/usr/bin/python3.5  import timeit  from ctypes import \*  def fib(n):  if n == 0:  return 0  elif n == 1:  return 1  else:  return(fib(n-1) + fib(n-2))  if \_\_name\_\_ == "\_\_main\_\_":  l = cdll.LoadLibrary("./libmymath.so")  for i in range(10):  t = timeit.Timer(lambda: fib(c\_int(i).value))  print ('Pure python %.2f usec/pass' % (1000000 \* t.timeit(number=100000)/100000))  t1 = timeit.Timer(lambda: l.fib(i))  print ('Ctypes python %.2f usec/pass' % (1000000 \* t1.timeit(number=100000)/100000)) |

Here we see that we've declared a function *fib*, which calculates the sum of the first *n* numbers of the sequence in Python. We're also going to use ctypes to call the libmymath.so *fib* function to do the same thing. Let's examine the output of this program.

|  |
| --- |
| Value of n = 0  Pure python 0.49 usec/pass  Value of n = 0  Ctypes python 0.50 usec/pass  Value of n = 1  Pure python 0.51 usec/pass  Value of n = 1  Ctypes python 0.50 usec/pass  Value of n = 2  Pure python 0.92 usec/pass  Value of n = 2  Ctypes python 0.52 usec/pass  Value of n = 3  Pure python 1.34 usec/pass  Value of n = 3  Ctypes python 0.53 usec/pass  Value of n = 4  Pure python 2.07 usec/pass  Value of n = 4  Ctypes python 0.55 usec/pass  Value of n = 5  Pure python 3.23 usec/pass  Value of n = 5  Ctypes python 0.58 usec/pass  Value of n = 6  Pure python 5.20 usec/pass  Value of n = 6  Ctypes python 0.64 usec/pass  Value of n = 7  Pure python 8.64 usec/pass  Value of n = 7  Ctypes python 0.72 usec/pass  Value of n = 8  Pure python 13.95 usec/pass  Value of n = 8  Ctypes python 0.84 usec/pass  Value of n = 9  Pure python 26.26 usec/pass  Value of n = 9  Ctypes python 1.03 usec/pass |

Note that for small values for n, the performance of the pure python and the ctypes call is quite similar, however, as n gets larger, and the computation gets more expensive, calling the c function produces quite dramatic performance increases.

If we examine the program, we see that we call a method *LoadLibrary()* from the cdll class which is part of the ctypes module. We pass in the path to the desired shared object as the argument to the LoadLibrary method. We can then call functions on that library as we would call a Python method, by using the '.' operator.

Ctypes defines a number of intrinsic types that map to Python objects. Here is a small sampling of them.

|  |  |  |
| --- | --- | --- |
| **Ctypes type** | **C type** | **Python type** |
| c\_bool | \_Bool | bool |
| c\_byte | char | One character byte object |
| c\_short | short | int |
| c\_int | int | int |
| c\_long | long | int |
| c\_float | float | float |
| c\_double | Double | float |
| c\_char\_p | char \* (NULL terminated) | bytes object or None |
| c\_void\_p | void \* | int or None |

This is an incomplete list. For the full table of C to Python intrinsic mappings refer to the Python.org documentation on ctypes.

We can also pass pointers via ctypes. Consider the following C function *divide()* which we have defined in our mymath library as taking three parameters (int a, int b and float \* remainder) and returning an integer result and a floating point remainder using the *fmod()* function from the C math library. Here's how we would pass the floating point value into the C function.

|  |
| --- |
| #!/usr/bin/python3.5  import timeit  from ctypes import \*  if \_\_name\_\_ == "\_\_main\_\_":  l = cdll.LoadLibrary("./libmymath.so")  div = l.divide  div.argtypes = [c\_int, c\_int,POINTER(c\_float)]  x = c\_int(3)  y = c\_int(10)  remainder =c\_float(0)  result = div(x.value,y.value,byref(remainder))  print ("result = %d remainder = %f" % (c\_int(result).value,remainder.value)) |

Let's look at this program. We note that we have an attribute to the div object (Which is a function pointer to the divide function in the C library) called *argtypes*. We can use this attribute to specify what arguments will be passed to the divide function. Note that the third one is specified as a pointer to a float (The POINTER(c\_float) statement). We declare a variable called remainder as a floating point number and then pass it by reference in to the divide function (The byref(remainder) statement).

# Module 7. Speeding up your code.

Let's look now at how we can use some different tools to analyze your Python code with an eye towards improving performance. This is usually called *profiling* or *coverage analysis.* In this module, we'll discuss some methods to examine your code to find out where it is spending most of its time. Once we know that, we can get some ideas of where we can refactor things to speed up its performance.

Let's take a look at a simple python program:

|  |
| --- |
| #!/usr/bin/python3  import time  import cProfile  def calcsum():  calcsum=0  for i in range(100000):  calcsum += i  def main() :  calcsum()  if \_\_name\_\_ == "\_\_main\_\_":  cProfile.run('main()') |

Here we have a program that calculates the sum of all integers up to 100000. We'd like to know how much time that the calcsum() function takes to do this. Let's examine a couple of ways to do this.

The first approach is to simply time the function by using the time module in Python. Let's do this and take a look at our re-written program.

|  |
| --- |
| #!/usr/bin/python3  import time  def calcsum1():  start=0  finish=0  calcsum = 0  start = time.time()  for i in range(10000):  calcsum += 1  finish = time.time()  return finish - start  def main():  timetorun = calcsum1()  timetorun = timetorun \* 100000  print("%.2f" % (timetorun))  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Note here that we've taken our calcsum1 method and timed the relevant code, in this case the loop iteration over 10000 integers and summing them up manually. We start the timer with the *start = time.time()* and end with the *finish=time.time()*. It is then a trivial operation to subtract the start time from the finish time and get the resultant delta. We can even use this method to compare two different ways of doing the summation algorithm.

|  |
| --- |
| #!/usr/bin/python3  import time  def calcsum1():  start=0  finish=0  calcsum = 0  start = time.time()  for i in range(10000):  calcsum += 1  finish = time.time()  return finish - start  def calcsum():  start=0  finish=0  start = time.time()  calc = sum(range(10000))  finish = time.time()  return finish - start  def main():  timetorun = calcsum1()  timetorun = timetorun \* 100000  print("%.2f" % (timetorun))  timetorun = calcsum()  timetorun = timetorun \* 100000  print("%.2f" % (timetorun))  if \_\_name\_\_ == "\_\_main\_\_":  main() |

The output of this program is:

|  |
| --- |
| 64.75  18.33 |

This output clearly shows that the builtin sum function is far superior in terms of performance for doing any sort of summation.

This methodology, however, has serious flaws. For one thing, the Python garbage collection algorithm is also running, which may very well affect how the algorithm performs. Also, we only call the function and time it once. It's is far preferable to run the function numerous times and take an average of the times. This better reflects a real world scenario where other external events may affect the runtime environment of the program.

In order to fix this, we turn to another Python module designed specifically to get around these issues, the *timeit* module. Let's take a look at some code that showcases the timeit module.

|  |
| --- |
| #!/usr/bin/python3  import timeit  def fib(n):  if n < 2:  return n  else:  return fib(n-2) + fib(n-1)  def main():  i = 5  t = timeit.Timer(setup = 'from \_\_main\_\_ import fib', stmt = 'fib(5)')  print ('Value of n = %.d\nPure python %.2f usec/pass' % (i,t.timeit(number=100000)))  outputs = t.repeat(number = 1000000, repeat = 3)  for time\_value in outputs:  print ('Value of n = %.d\nPure python %.2f usec/pass' % (i,time\_value))  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Here we see timeit being used in two different ways. First, we set up the timeit module by calling the *Timer* method. This method takes a setup parameter which will set up the timing. The setup in this case is that to import the fib function and make it available to our timeit instance.. The stmt argument will indicate to the timeit module what code will be timed. Some things to note.

The timeit argument by default turns off the Python Virtual Machine garbage collection. This can result in more accurate timings. Also, if the number of times the code is run isn't specified, timeit will attempt to determine the number based on what code is being run.

It is also possible to run the timeit module from the command line. Here's an example:

|  |
| --- |
| python -m timeit -n 1000000 -r 100 '10/2.4' |

The -m option tells Python to load the timeit module, the -n option tells Python to run the timed statement one million times and repeat this for one hundred times. Finally, the last line is the Python statement to run, in this case 10 divided by 2.4.

The output of this statement looks like this:

|  |
| --- |
| 1000000 loops, best of 100: 0.0559 usec per loop |

While timeit is useful for timing specific functions, what if we try and time a function like *calculate\_stuff()* defined below?

|  |
| --- |
| def calculate\_stuff(n):  calcsum = 0  for i in range(10):  calcsum += i  if calcsum < 0:  calcsum = abs(calcsum) |

Using timeit will tell us how long it took to run calculate\_stuff, but we see from the definition that there are a number of sub-functions that get called, including the builtin *range()* and *abs()* functions. We would have to wrap those functions with timeit calls as well in order to get information about how long those individual functions took to run. Therefore, what we really need is a tool that will not only time a function or method, but also all of the functions that are called from the parent as well as any other executing code.

Python offers a tool called *cProfile.* This tool, when run, generate output about each function and sub-function in the program. Let's look at a very simple example.

|  |
| --- |
| #!/usr/bin/python2.7  import time  import cProfile  def calcsum():  calcsum=0  for i in range(100000):  calcsum += i  def main() :  calcsum()  if \_\_name\_\_ == "\_\_main\_\_":  cProfile.run('main()') |

Note here that we import the cProfile and call cProfile.run with the main function as the argument. cProfile will generate a table of data that looks like this:

|  |
| --- |
| 5 function calls in 0.006 seconds  Ordered by: standard name  ncalls tottime percall cumtime percall filename:lineno(function)  1 0.000 0.000 0.006 0.006 <string>:1(<module>)  1 0.000 0.000 0.006 0.006 profile4.py:12(main)  1 0.004 0.004 0.006 0.006 profile4.py:6(calcsum)  1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}  1 0.002 0.002 0.002 0.002 {range} |

The output gives us the following:

<x> function calls in <y> seconds. A rather self-explanatory line saying that the profiler ran, in this case, five different functions in .006 seconds.

Ordered by means that the output was sorted by the name of the function.

The columns are described as:

|  |  |
| --- | --- |
| **Column Name** | **Description** |
| Ncalls | How many times that particular function was called. |
| Tottime | How long it took to execute each function *not including* sub functions |
| PerCall | The total time divided by the number of calls |
| CumTime | How long it took to execute each function *including* sub functions |
| PerCall | The cumulative time divided by the number of calls. |
| Function name | The name of the function being profiled. |

The run method of the cprofiler also allows for a second argument which is a file name that the statistical information can be stored to. This is quite useful when manipulating the output with a module called *pstats()*. Additionally, saving the profiler output to a file will also allow us to visualize the data with tools such as *snakeviz* and *runsnakerun*.

Another way to run cProfile is from the command line. Instead of calling it programmatically, we can do something like this:

|  |
| --- |
| python -m cProfile -o <profile output file> -s <sort order of output> <script\_name> |

Note that the -s flag only works if you do not specify an output file to which to send the data.

Running the pstats module on the output allows us to sort the output in various ways as well as controlling the output of the statistical information. The Python documentation has more information about the pstats module.