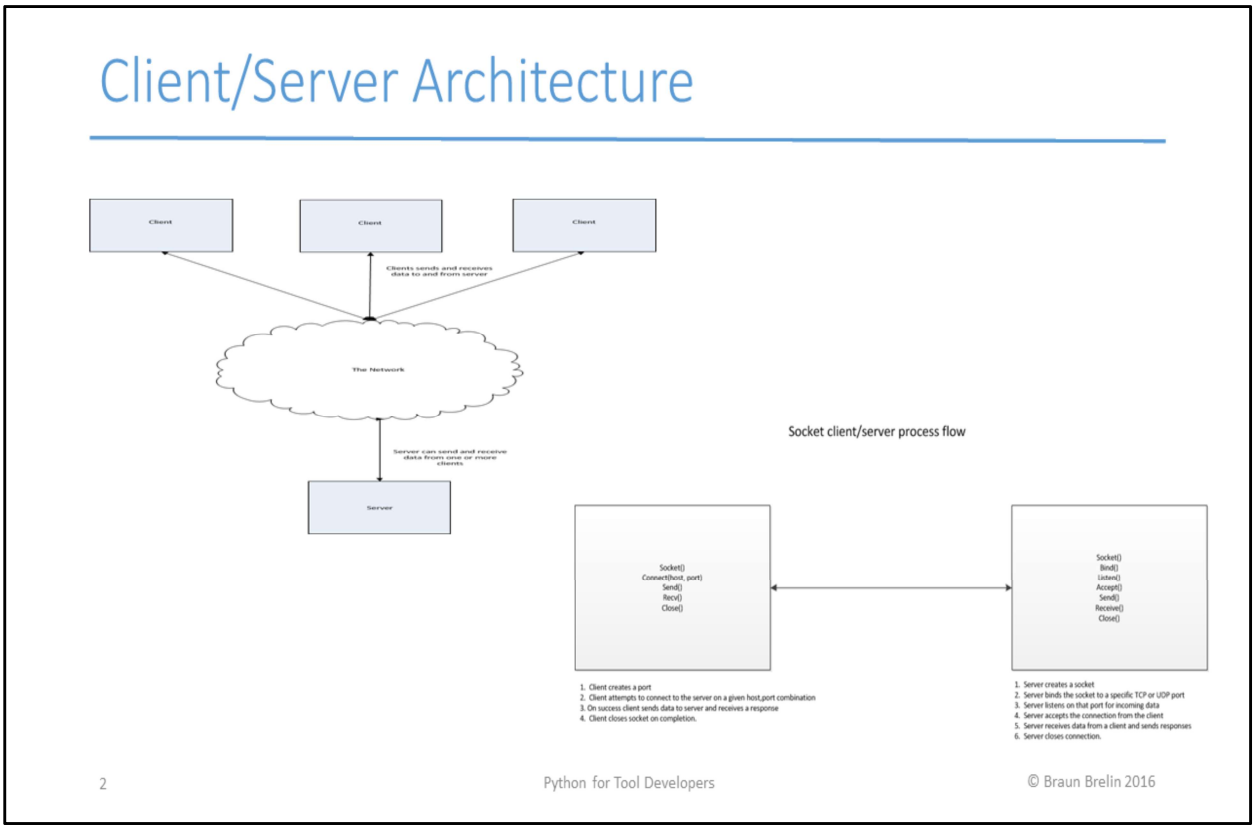
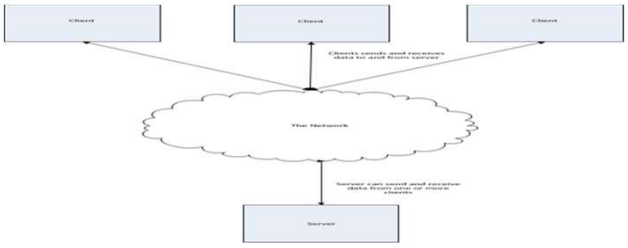
Module 2

Network Programming with Python

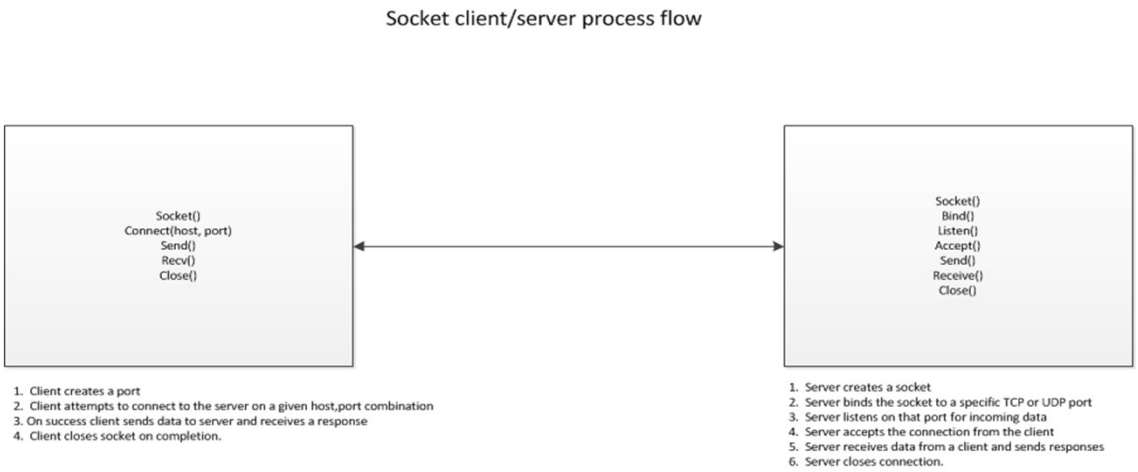
1

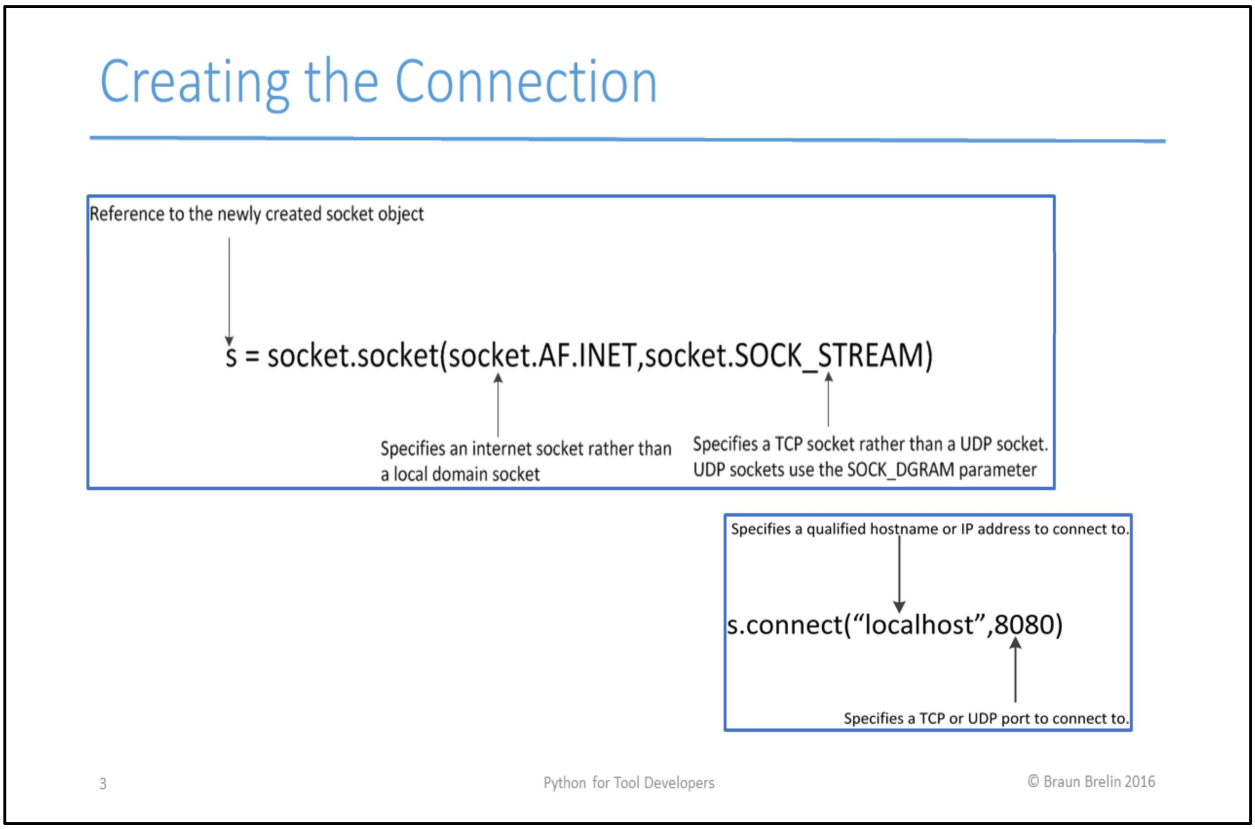


One of the most common use cases for writing applications is to allow multiple computers to be able to exchange information with each other over a network. This type of system architecture is called a *client/server* architecture. We can do this in Python by the use of the `Socket` API. The following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

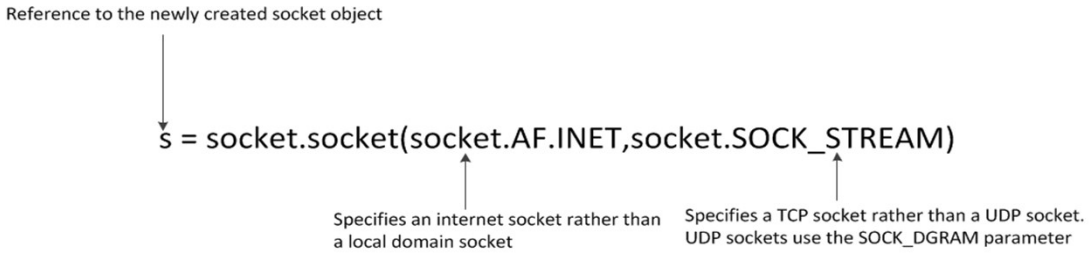


The `socket` API for Python is relatively straight-forward and follows standard pre-defined steps.

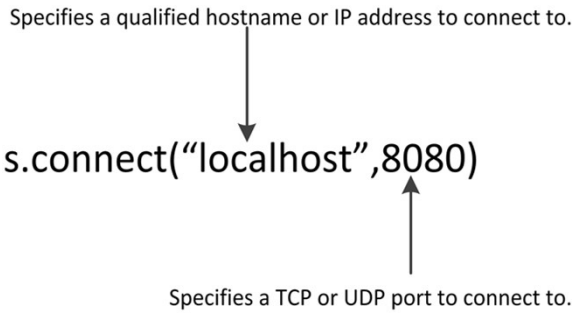




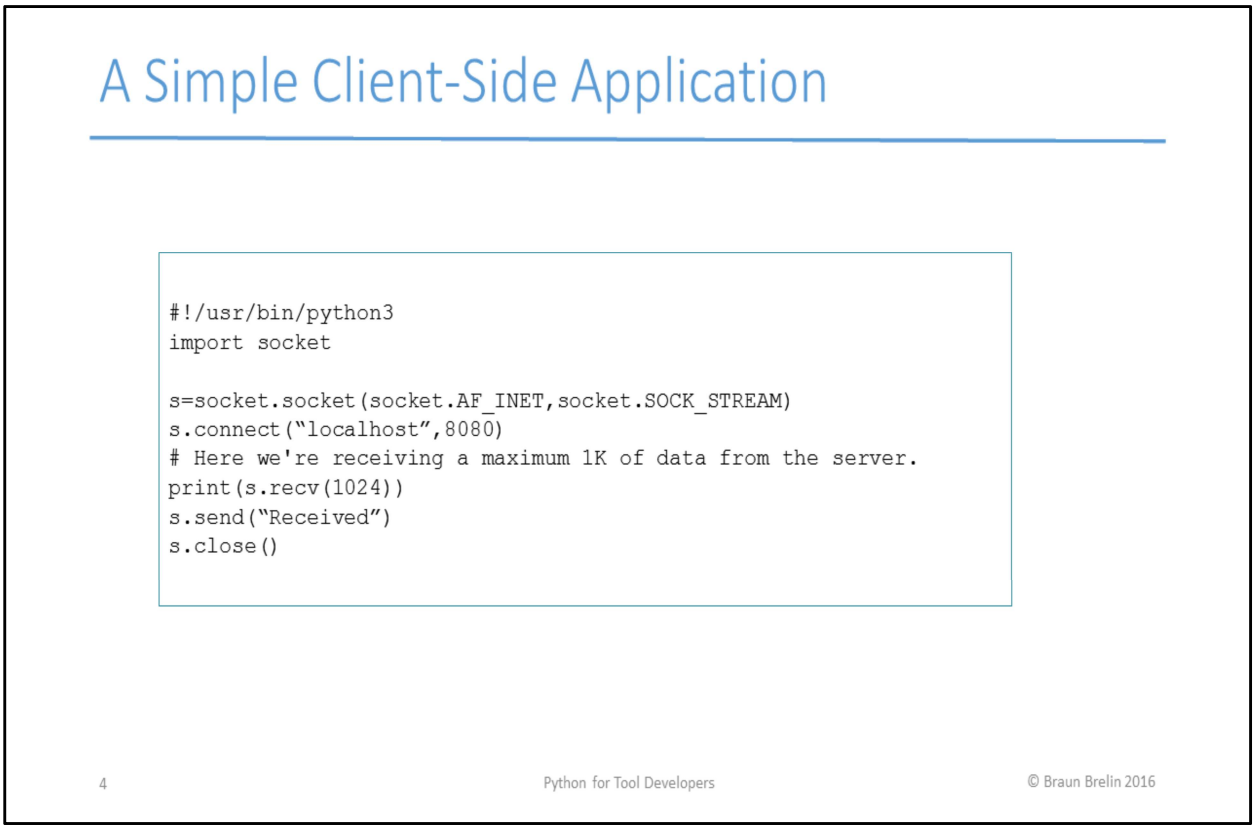
As we can see from the diagram, the process of creating a client server application follows two distinct patterns, one on the client, and the other on the server.
The client starts by creating a `socket` object using the socket library call like so:



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.

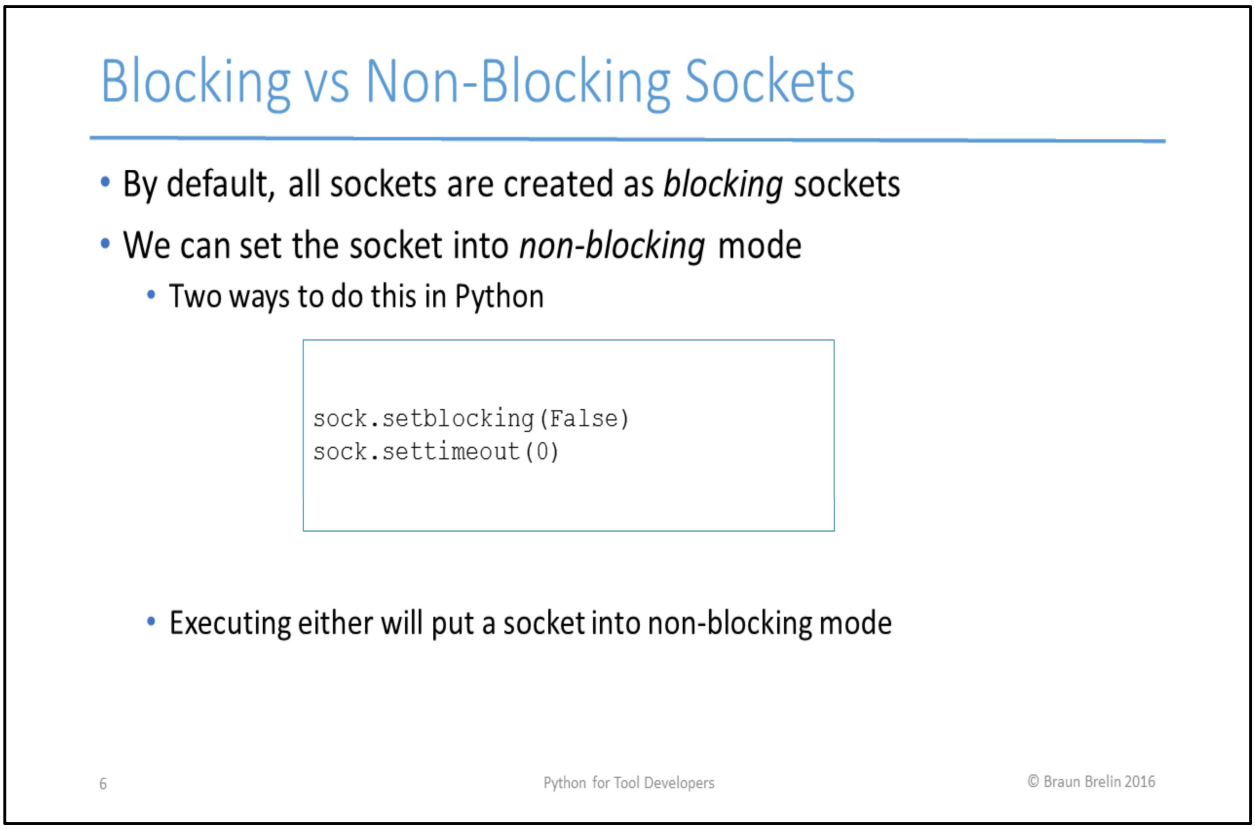
Step 2. Bind the socket to the port. We do this by using the *bind* library call. This call ties the socket object to the specific port.

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.

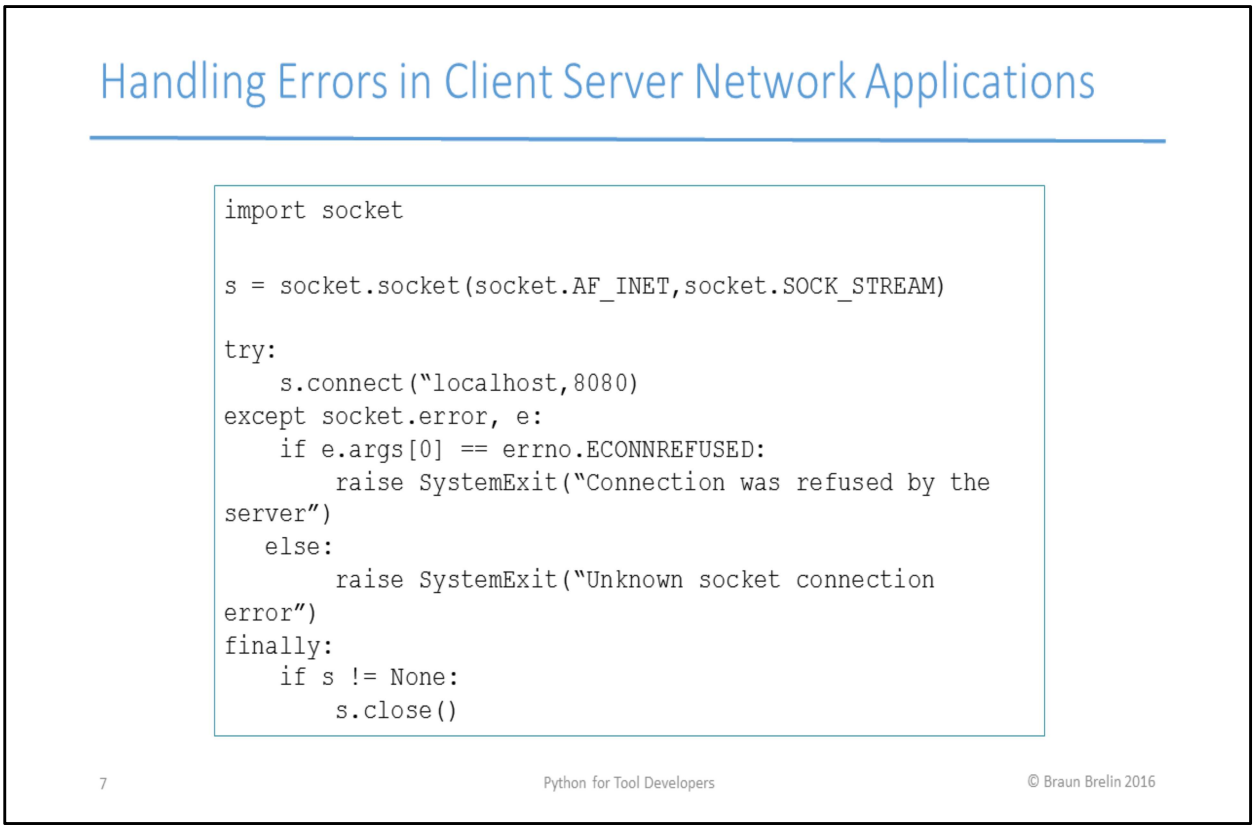
You have a client application that attempts to connect to a server. If the server is, for some reason, unavailable, it would be desirable to allow the client application to either do some other action and wait for the connect operation to succeed or cancel the connect operation altogether. However, at 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 behaviour.

This behaviour 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.



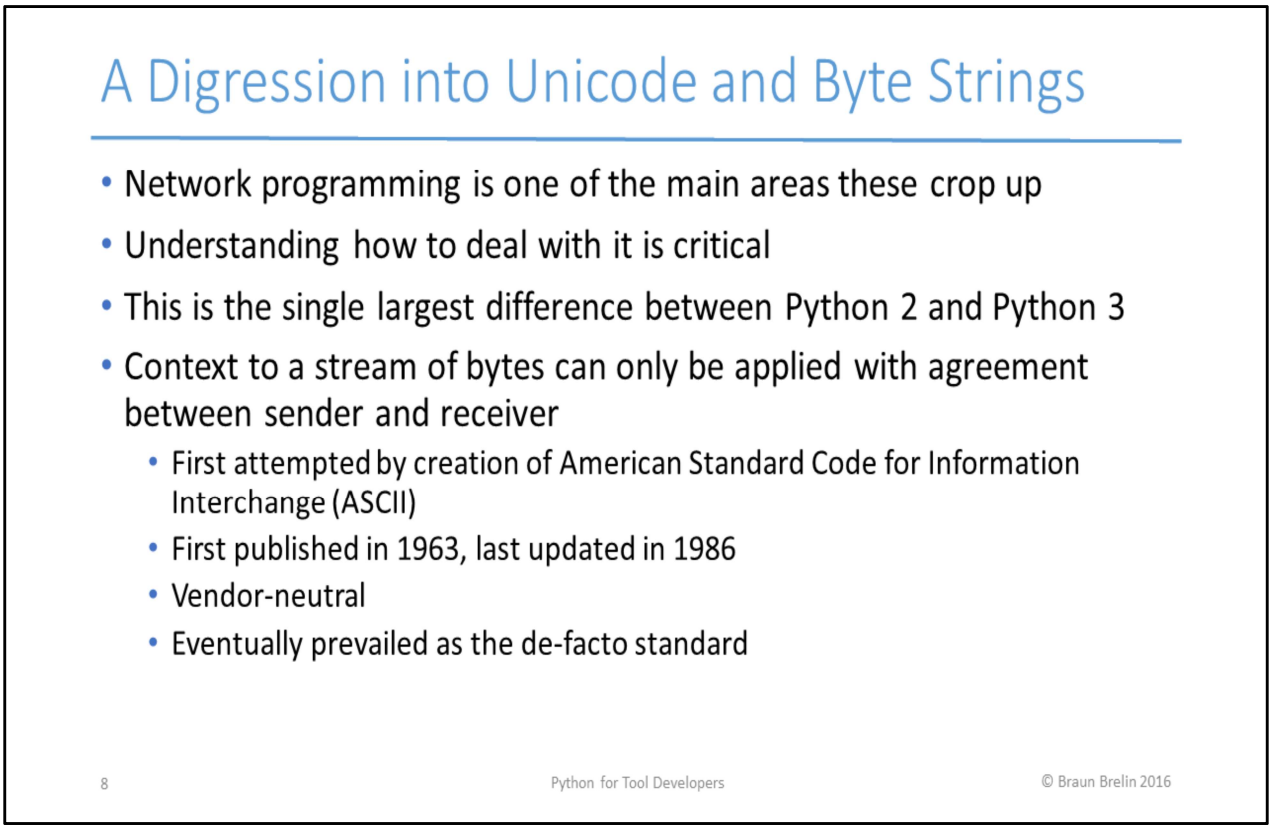
Handling errors in client server network applications

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.

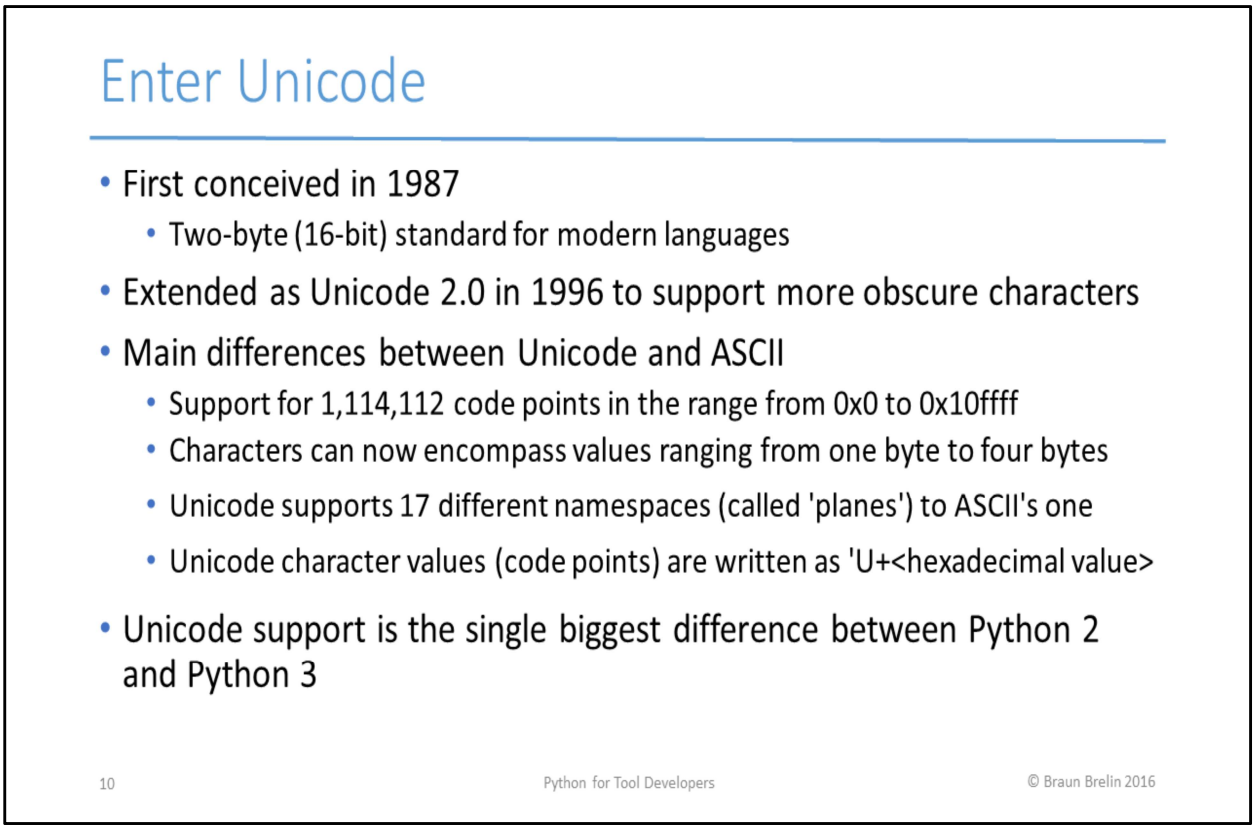
The first edition of the ASCII table was published in 1963 and last received an update in 1986. Other, vendor-proprietary, standards such as EBCDIC (created by IBM) were devised around the same time. However, ASCII, being vendor-neutral eventually prevailed as the de-facto standard for assigning meaning 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 `valuereplace`

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:

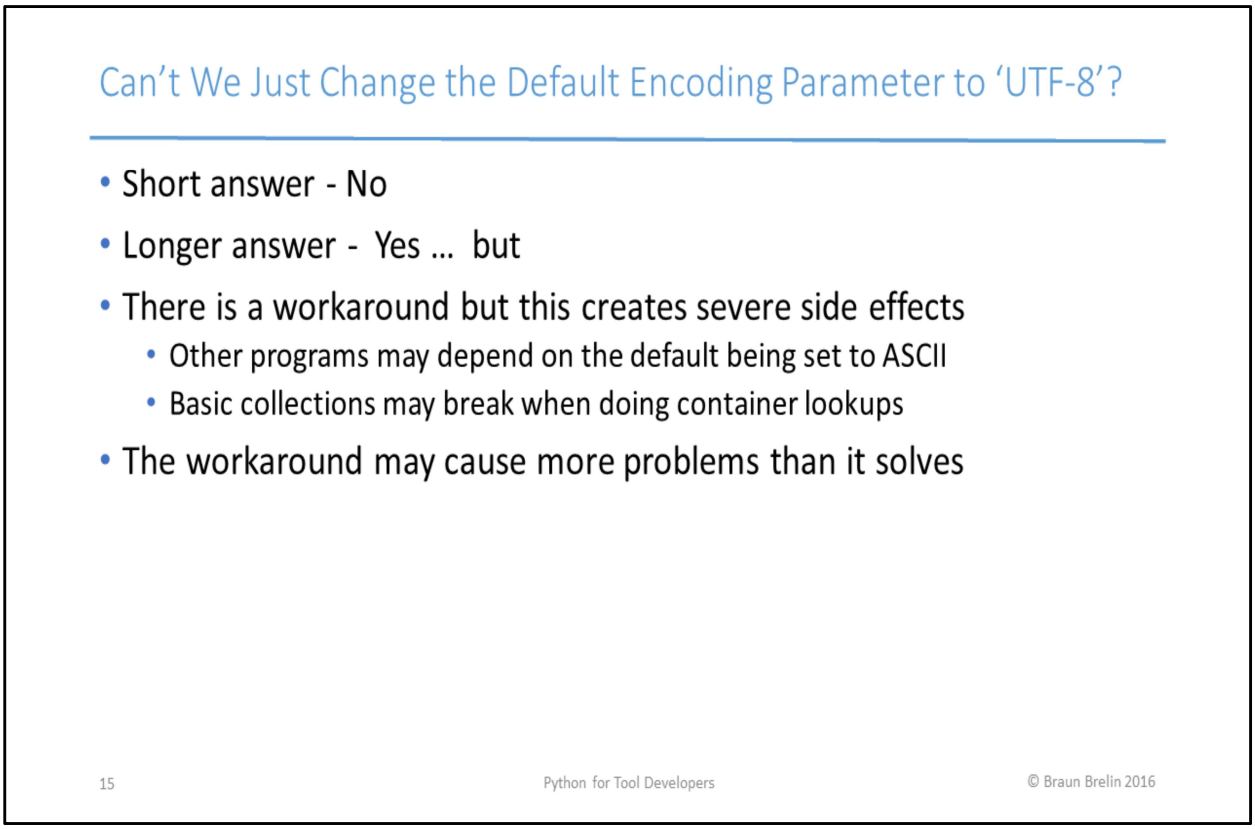
```
In [1]: import sys


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 conversion will returns the expected value.

How Does Python 3 Handle this problem?

- Python 3 completely re-does the concept of byte and Unicode strings
  - Redefined the `str` class
  - Python 2 treats `str` as a byte string
  - Python 3 treats `str` as Unicode
  - Python 3 no longer does implicit type conversion
- Chances of a runtime error are substantially reduced
- But you must explicitly encode and decode strings in your programs

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

16 Python for Tool Developers © Braun Brelin 2016

How does Python 3 handle this problem?

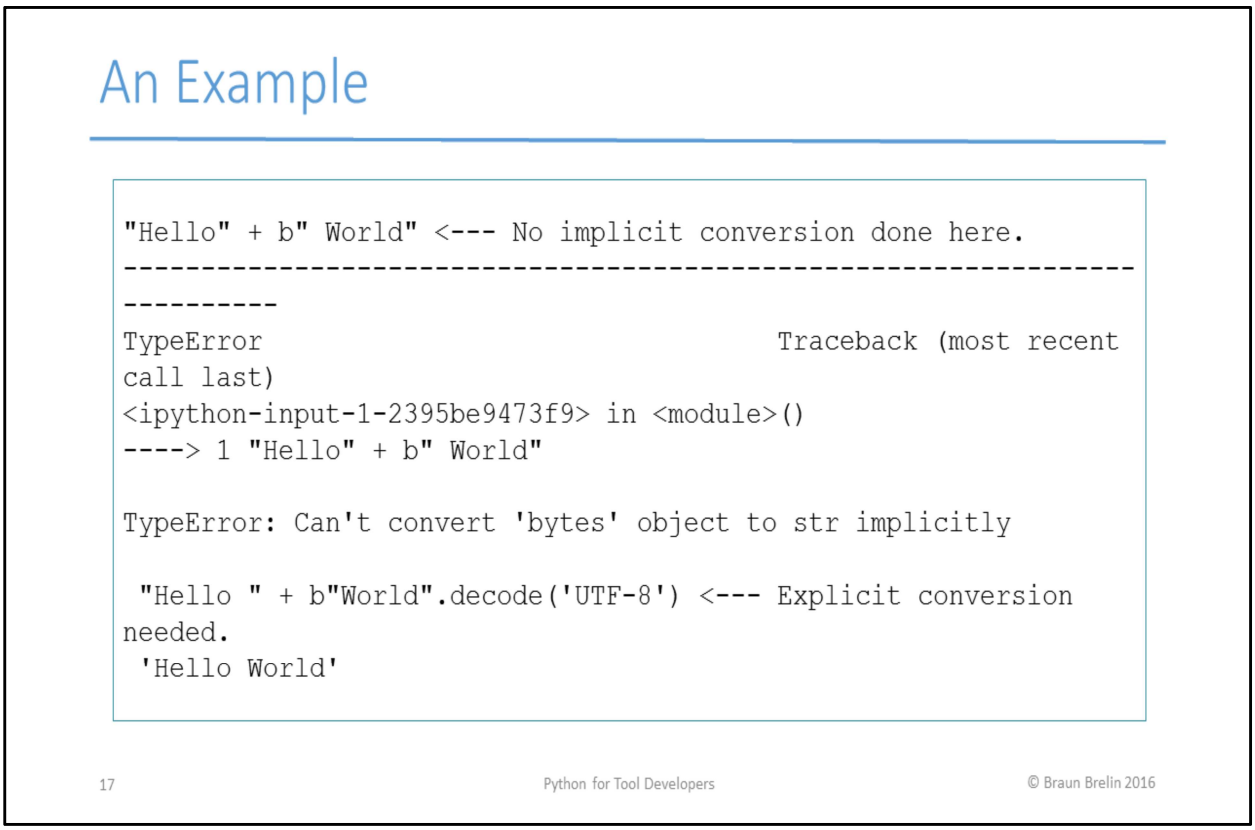
Python 3 completely re-does the concept of byte and unicode strings. Let's take a look.

```
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 Python 3 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.

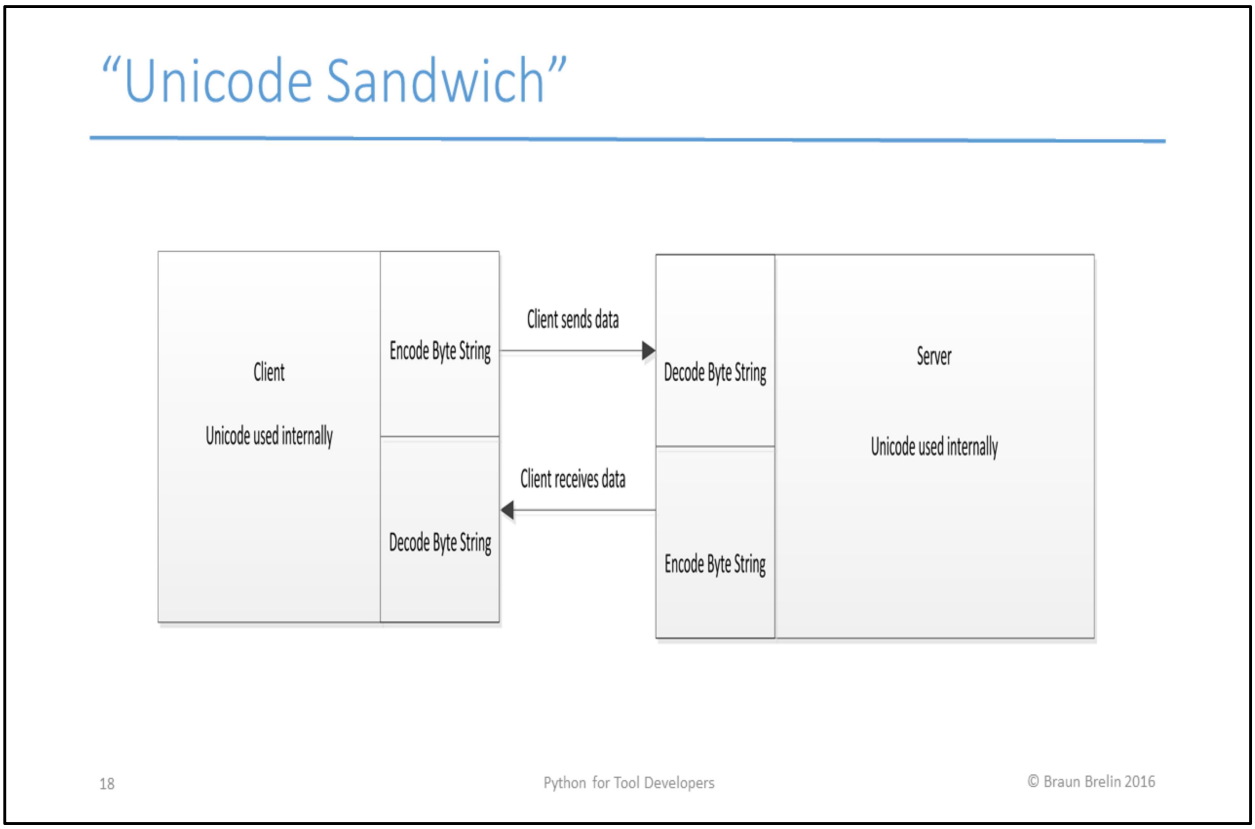


Let's see an example of this.

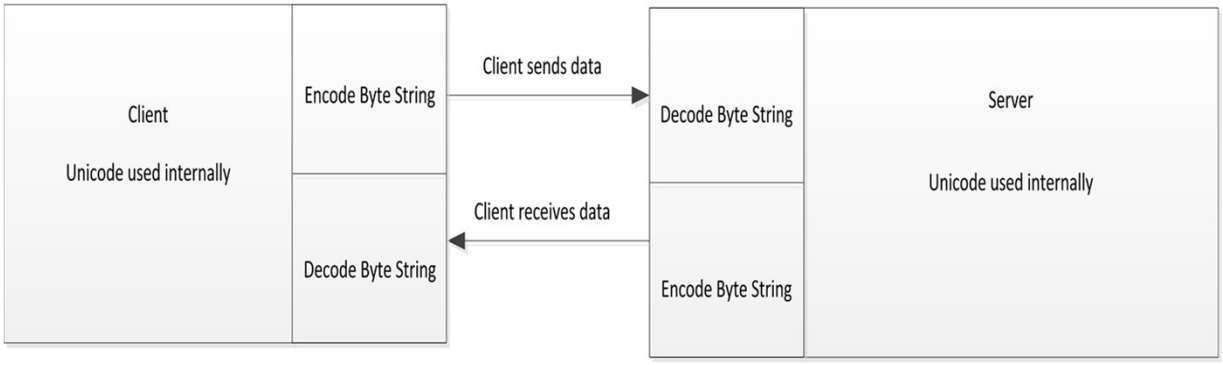
```
"Hello" + b" World" <--- No implicit conversion done here.
---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-1-2395be9473f9> in <module>()
----> 1 "Hello" + b" World"

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

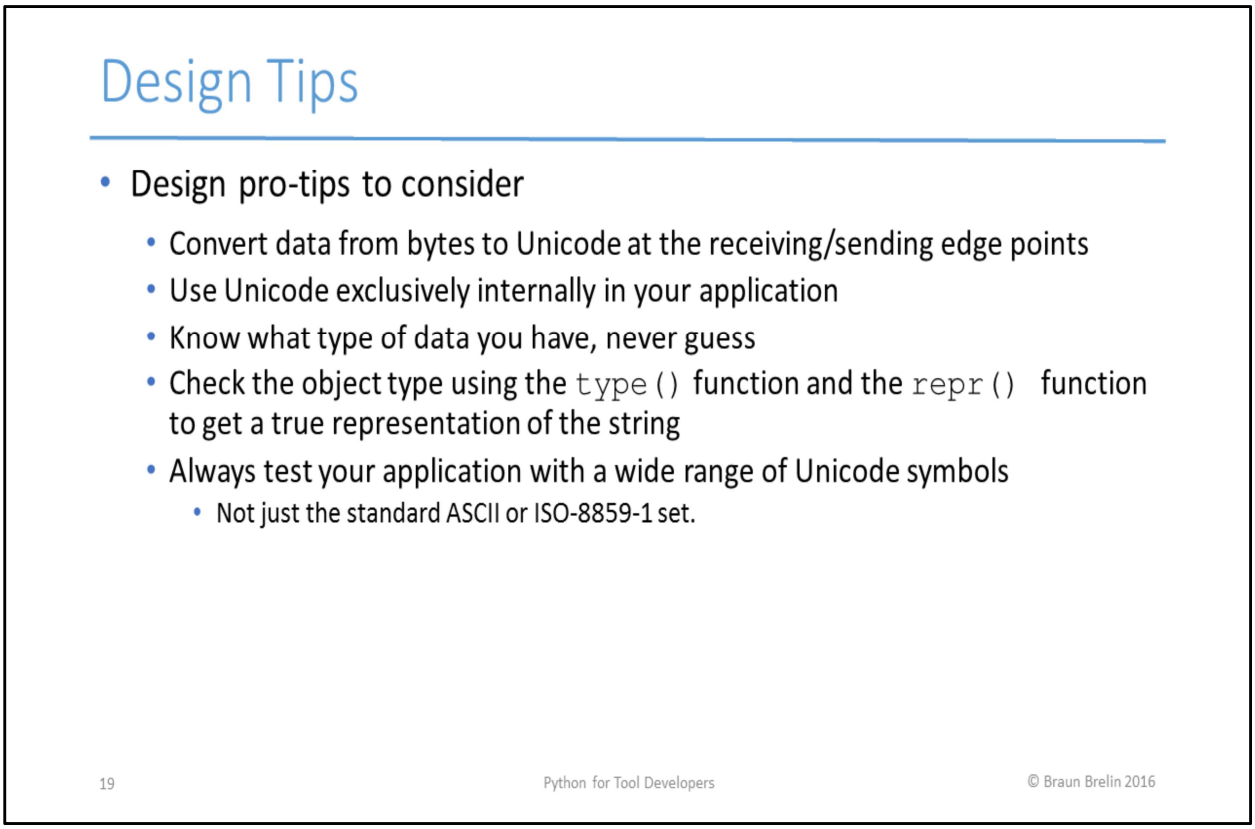


So, how do we actually use this in our applications? The best way to do this is to explicitly do the conversion to Unicode immediately upon receiving the byte string. Use Unicode strings internally, and then convert back to bytes when sending the data back. This can thought of as a "Unicode sandwich".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



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

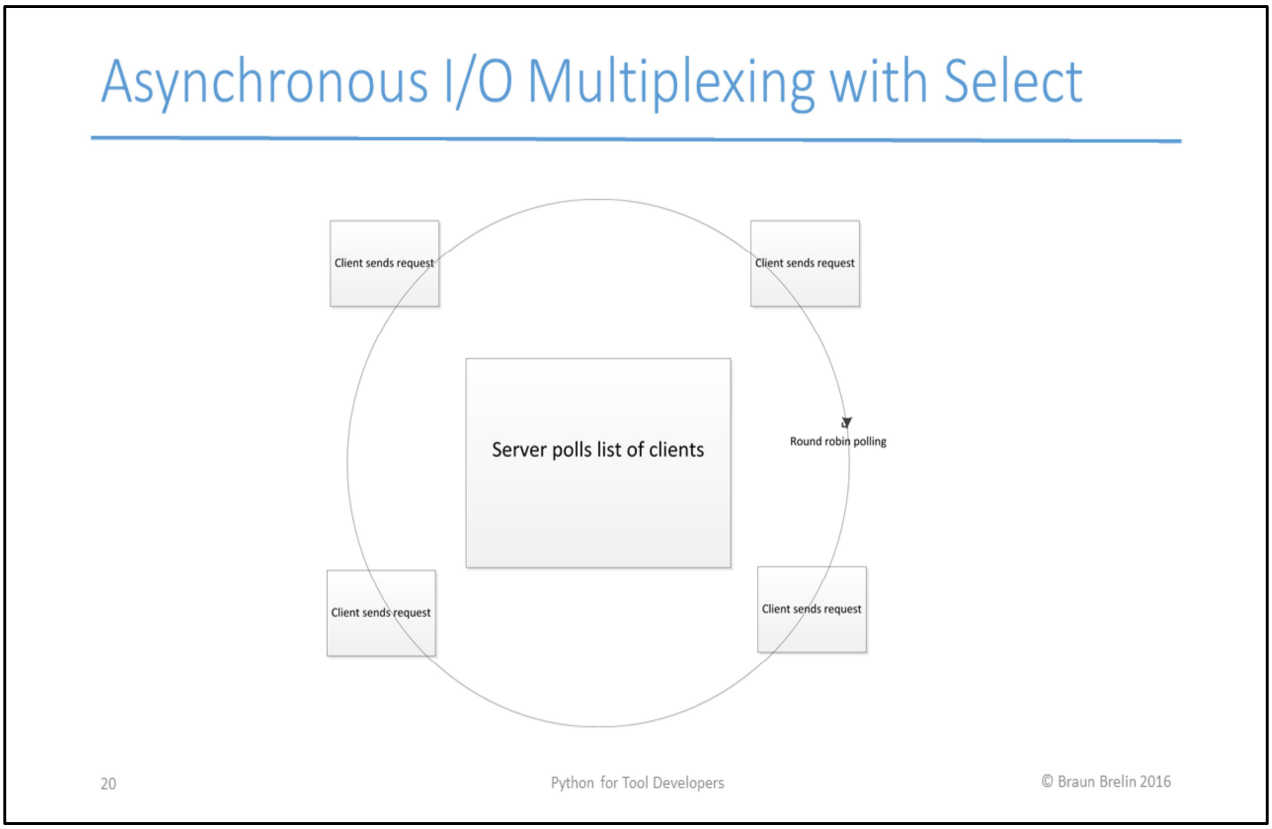
1. Everything in a computer is stored as bytes. Bytes in and of themselves have no meaning. We need some sort of convention or standard in order to assign meaning to the bytes.
2. English and the Latin alphabet set is no longer the universal medium of communication over the internet.



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.

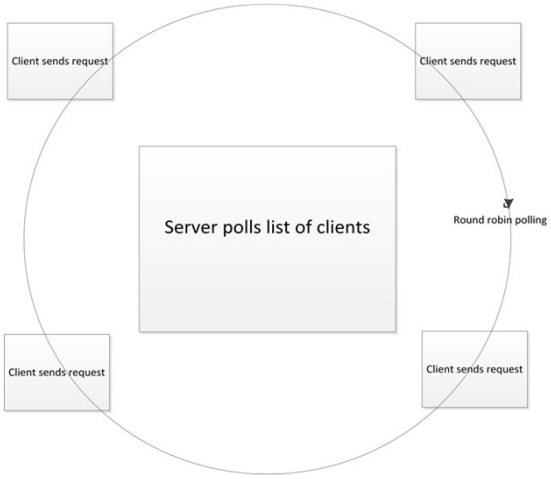
1. Convert data from bytes to Unicode at the receiving/sending edge points. Use Unicode exclusively internally in your application.
2. You need to know what type of data you have, never guess. Check the object type using the `type()` function and the `repr()` function to get a true representation of 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.

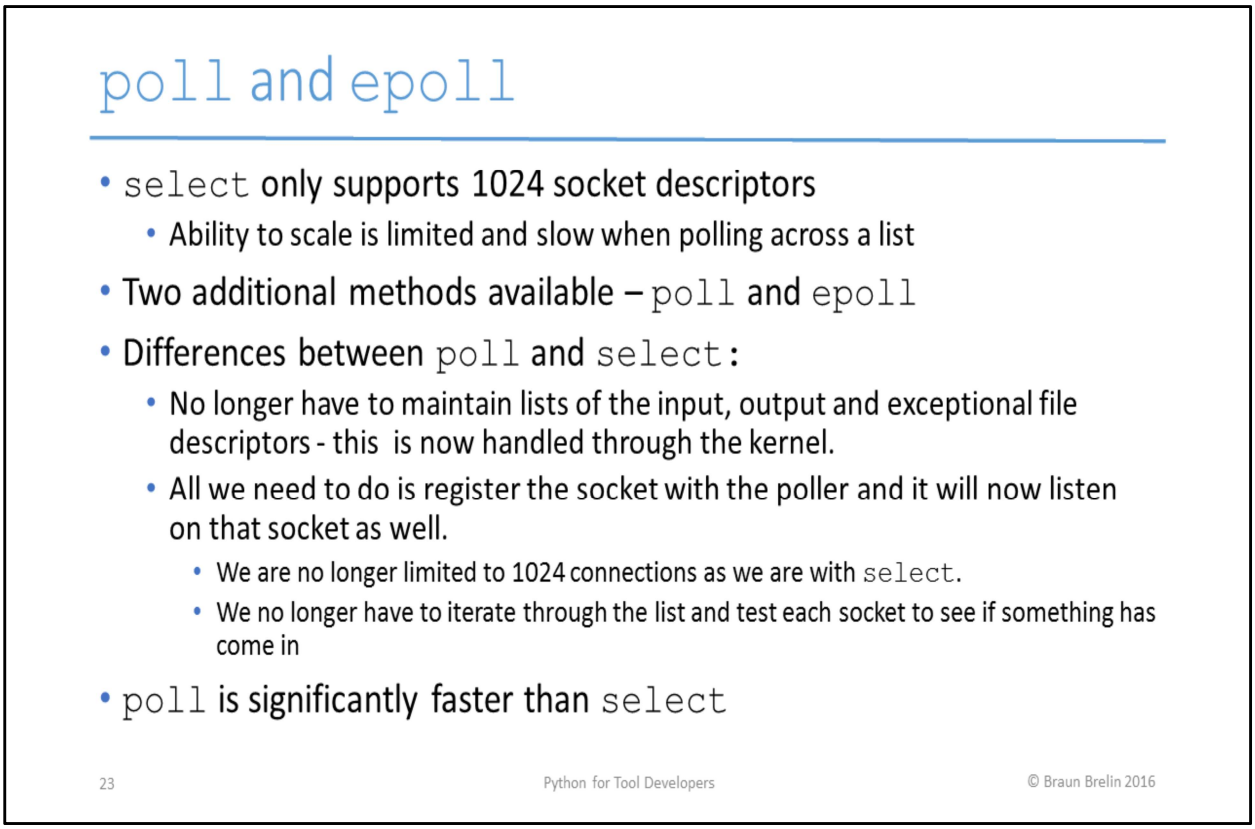
Let's take a look at some sample code that implements this.

```
#!/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:
```

Continued ……

```
# 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 a 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.
```

Continued ……

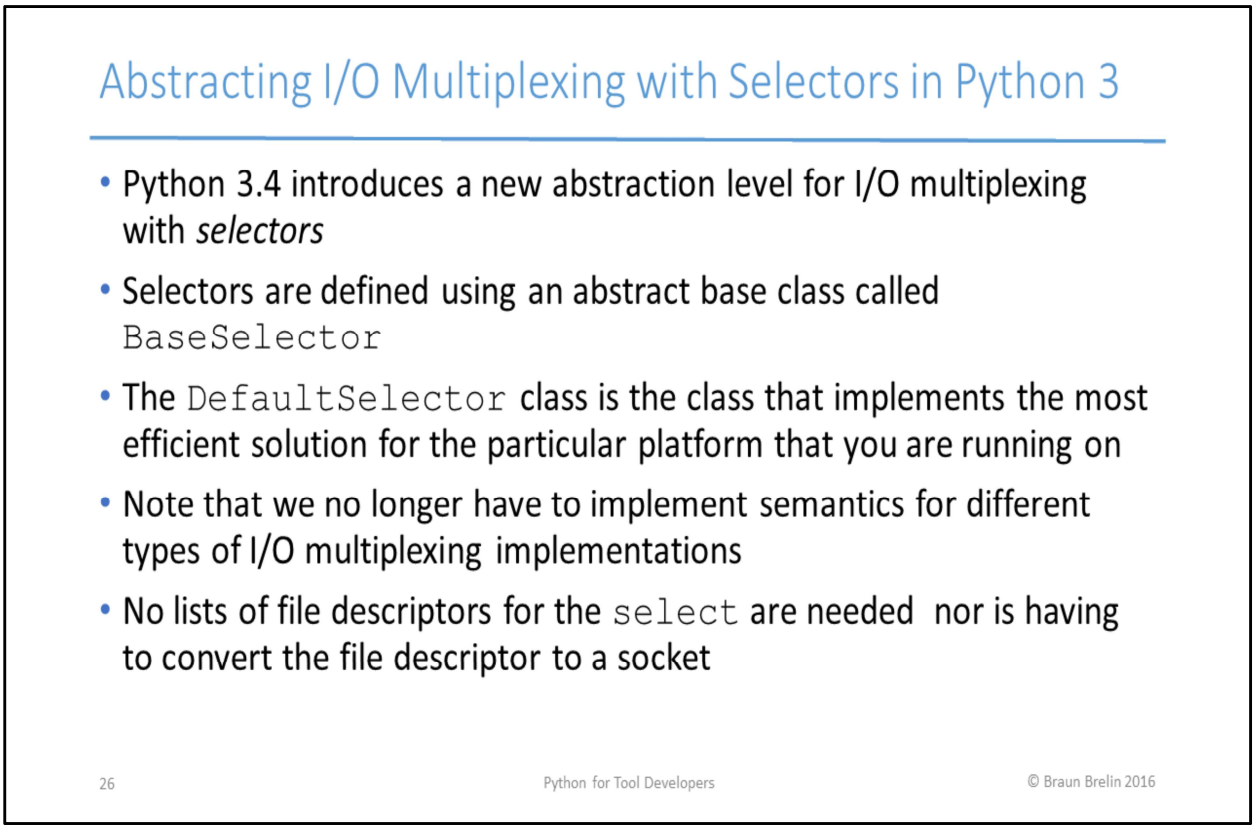
```
The poll call returns
# an event tuple which contains the file descriptor and the
READ_ONLY flag that woke it up.
# So let's first map the file descriptor to the socket.
        s = fd_to_socket[fd]

# 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()
```

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`.

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 optimal choice.

Let's take our previous examples with `select` and `poll` and re-write them 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.