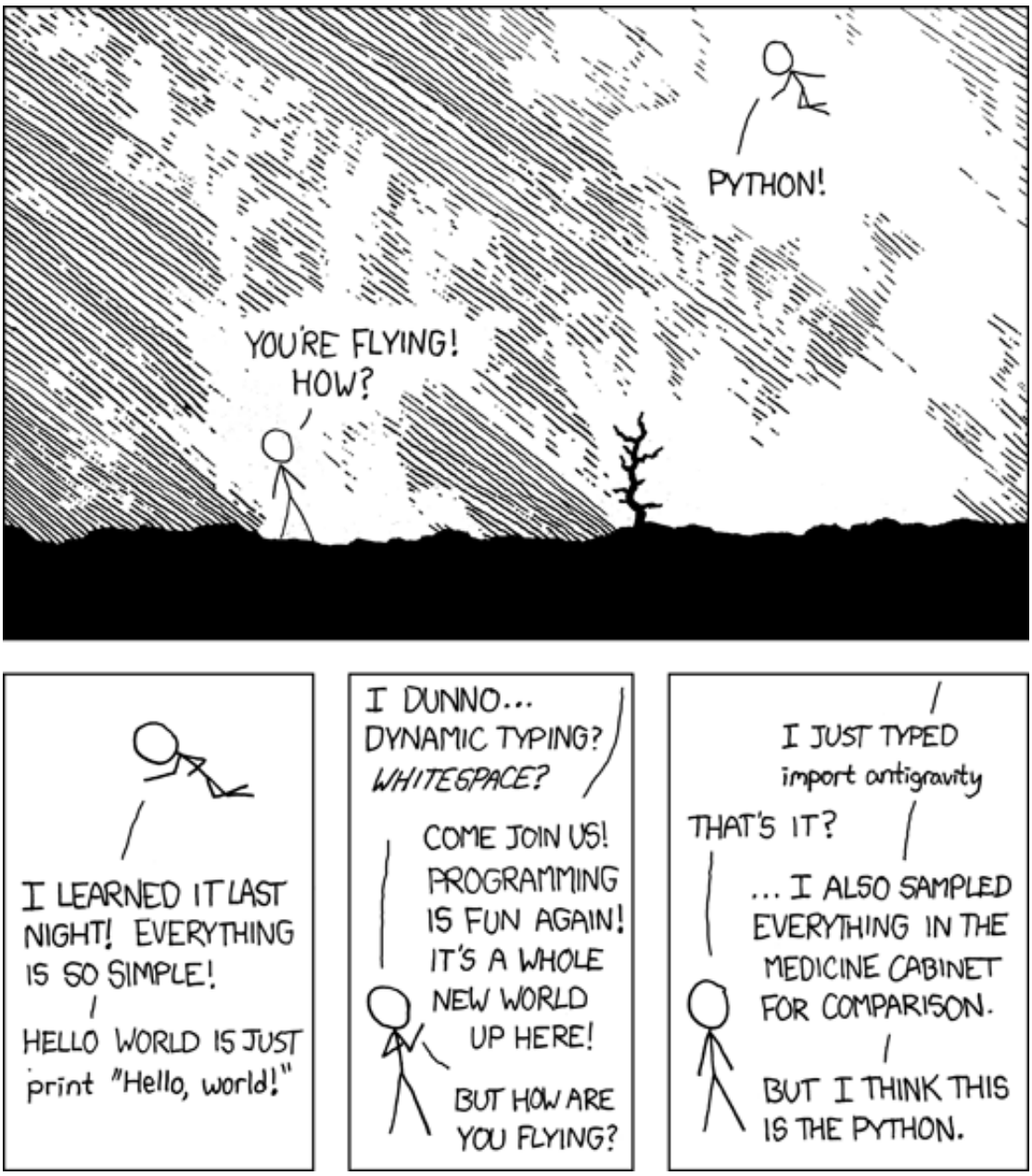


Computer Communications

Or how computers talk to each other.

Presentations

Class presentations are available for you to use. They will be unlocked in your "modules" after the presentation.

- You have to share your versions just like this one

Speaking of attribution, this course was developed from Cris Ewing's materials at (https://github.com/UWPCE-PythonCert/training.python_web)[https://github.com/UWPCE-PythonCert/training.python_web (https://github.com/UWPCE-PythonCert/training.python_web)[https://github.com/UWPCE-PythonCert/training.python_web)]

Course Materials

We now take a moment to tour the course materials and requirements.

Classroom Protocol

Questions to ask:

- What did you just say?
- Please explain what we just did again?
- How did that work?
- Why didn't that work for me?
- Is that a typo?

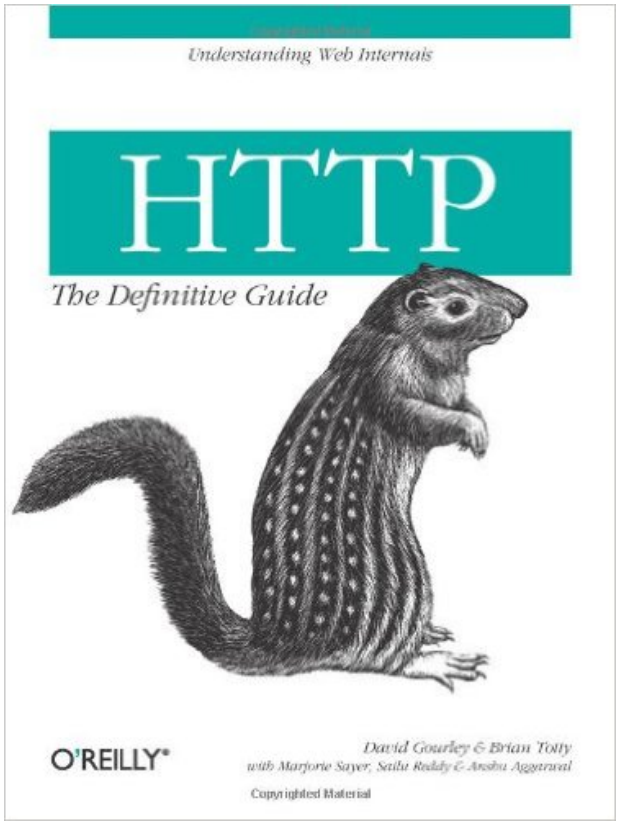
Classroom Protocol

Questions **not** to ask:

- **Hypotheticals**: What happens if I do X?
- **Research**: Can Python do Y?
- **Syllabus**: Are we going to cover Z in class?
- **Performance questions**: Is Python fast enough?
- **Unpythonic**: Why doesn't Python do it some other way?

Interesting Reading

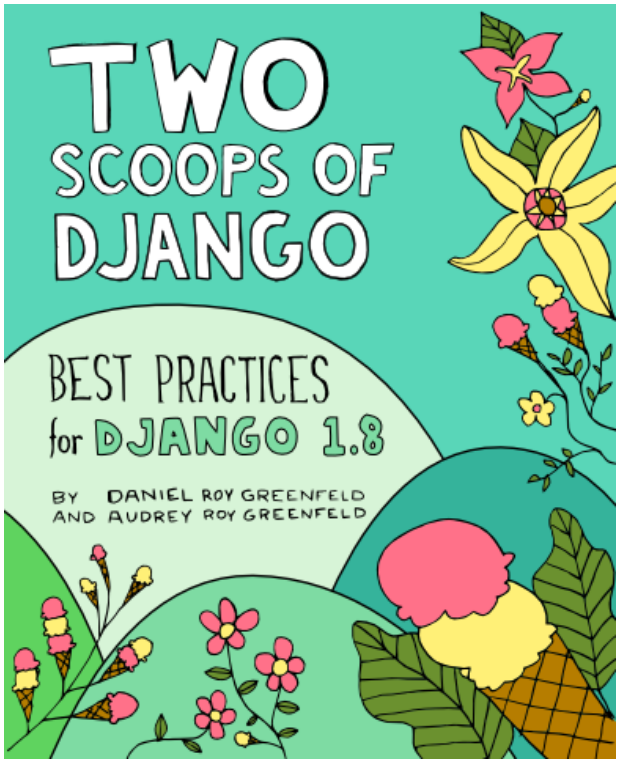
HTTP



- Outdated since the release of HTTP/2.0
- Probably still the best guide to the architecture of the web

http://www.amazon.com/HTTP-Definitive-Guide-Guides/dp/1565925092 (http://www.amazon.com/HTTP-Definitive-Guide-Guides/dp/1565925092)

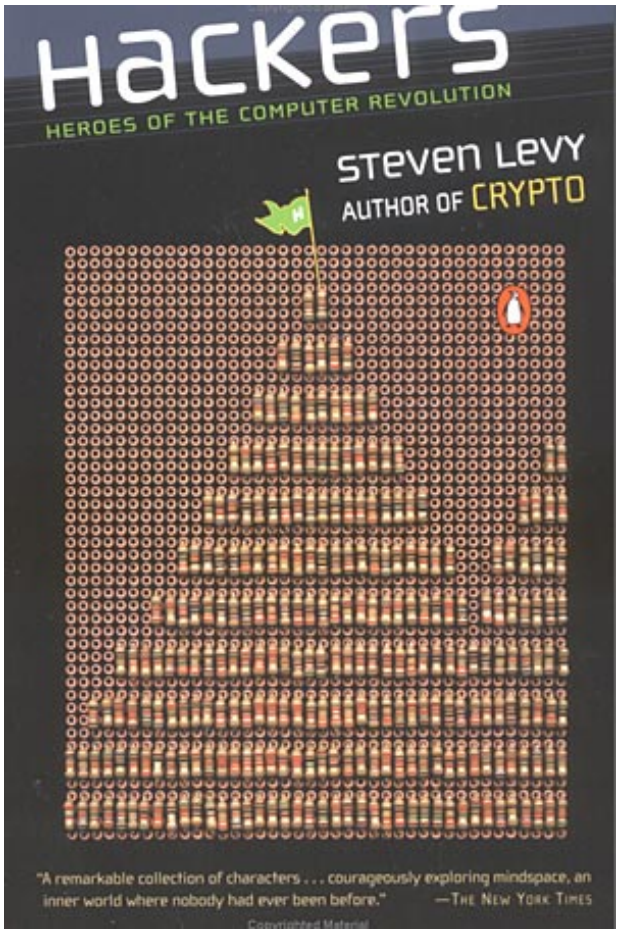
Django



- Considered one of the best series on Django development
- Frequently outdated by new releases of Django
- But Django 1.8 is a long-term-release, so this book will remain relevant

http://www.amazon.com/Two-Scoops-Django-Best-Practices/dp/0981467342 (http://www.amazon.com/Two-Scoops-Django-Best-Practices/dp/0981467342)

Computing Folk History



- Not especially relevant to this class
- Excellent folk history of computing in general

http://www.amazon.com/Hackers-Heroes-Computer-Revolution-Anniversary/dp/1449388396 (http://www.amazon.com/Hackers-Heroes-Computer-Revolution-Anniversary/dp/1449388396)

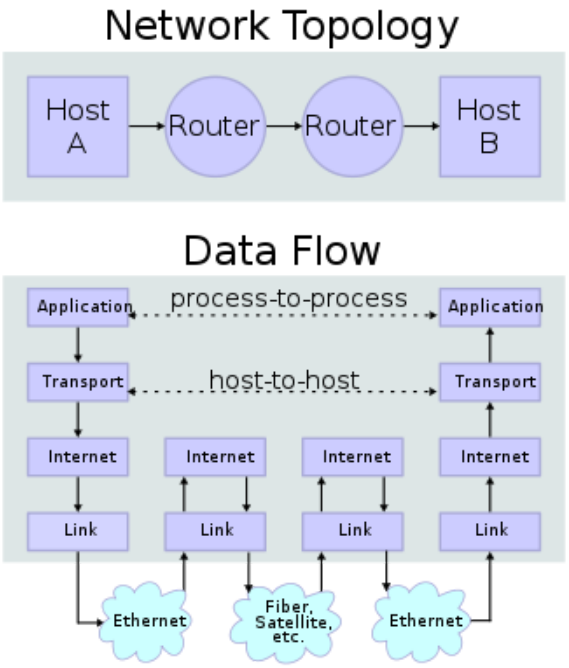
Introductions

A bit about me, your instructor, and then the following from you:

- Name
- How this course (web programming) or this series (Python) relates to your work and/or personal interests

- Your experience with web programming

TCP/IP



Processes can communicate:

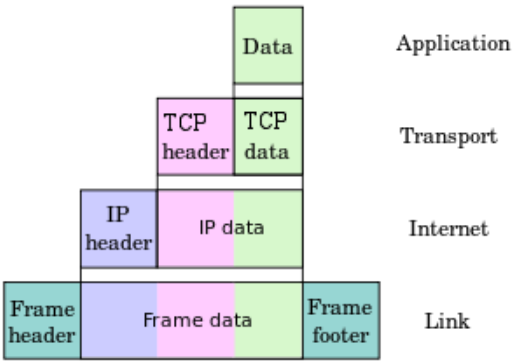
- inside one machine
- between two machines
- among many machines

Process divided into ‘layers’:

- ‘Layers’ are mostly arbitrary
- Different descriptions have different layers
- Most common is the ‘TCP/IP Stack’

https://en.wikipedia.org/wiki/Internet_protocol_suite (https://en.wikipedia.org/wiki/Internet_protocol_suite)

TCP/IP Stack : Link and Internet



Derived from: https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg (https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg)

The bottom layer is the ‘Link Layer’:

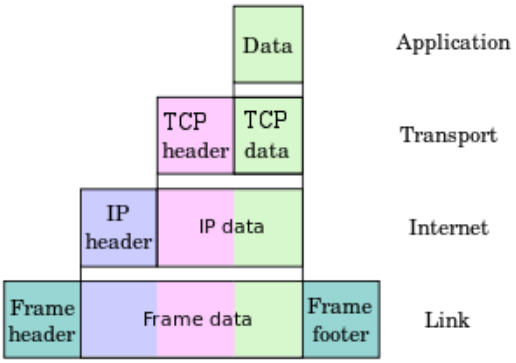
- Deals with the physical connections between machines, ‘the wire’
- Packages data for physical transport
- Executes transmission over a physical medium
  - what that medium is is arbitrary
- Implemented in the Network Interface Card(s) (NIC) in your computer

Moving up, we have the ‘Internet Layer’:

- Deals with addressing and routing
  - Where are we going and how do we get there?

- Agnostic as to physical medium (IP over Avian Carrier - IPoAC)
- Makes no promises of reliability
- Two addressing systems
  - IPv4 (current, limited '192.168.1.100')
  - IPv6 (future, 3.4 x 10^38 addresses, '2001:0db8:85a3:0042:0000:8a2e:037
  - That's 4.3 x 10^28 addresses per person alive today

TCP/IP Stack : Transport



Derived from: https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg (https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg)

Next up is the 'Transport Layer':

- Deals with transmission and reception of data
  - error correction, flow control, congestion management
- Common protocols include TCP & UDP
  - TCP: Tranmission Control Protocol
  - UDP: User Datagram Protocol
- Not all Transport Protocols are 'reliable'
  - TCP ensures that dropped packets are resent
  - UDP makes no such assurance
  - Reliability is slow and expensive

The 'Transport Layer' also establishes the concept of a port:

- IP Addresses designate a specific machine on the network
- A port provides addressing for individual applications in a single host

- 192.168.1.100:80 (the :80 part is the port)
- [2001:db8:85a3:8d3:1319:8a2e:370:7348] (:443 is the port)

This means that you don't have to worry about information intended for your web browser being accidentally read by your email client.

There are certain ports which are commonly understood to belong to given applications or protocols:

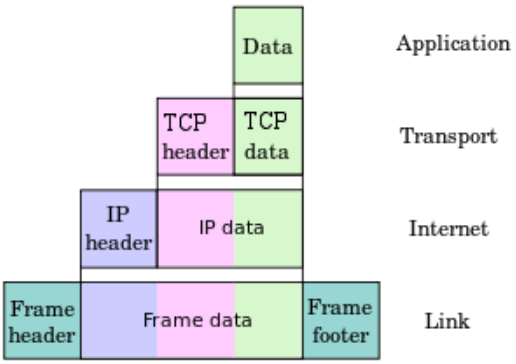
- 80/443 - HTTP/HTTPS
- 20 - FTP
- 22 - SSH
- 23 - Telnet
- 25 - SMTP
- ...

These ports are often referred to as **well-known ports** (http://en.wikipedia.org/wiki/List_of_TCP_and_

Ports are grouped into a few different classes

- Ports numbered 0 - 1023 are reserved
- Ports numbered 1024 - 65535 are open
- Ports numbered 1024 - 49151 may be registered
- Ports numbered 49152 - 65535 are called ephemeral

The TCP/IP Stack: Application



Derived from: https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg (https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg)

The topmost layer is the 'Application Layer':

- Deals directly with data produced or consumed by an application
- Reads or writes data using a set of understood, well-defined **protocols**
  - HTTP, SMTP, FTP etc.
- Does not know (or need to know) about lower layer functionality
  - The exception to this rule is **endpoint** data (or IP:Port)

this is where we live and work

Sockets

Think back for a second to what we just finished discussing, the TCP/IP stack:

- The *Internet* layer gives us an **IP Address**
- The *Transport* layer establishes the idea of a **port**.
- The *Application* layer doesn't care about what happens below...
  - Except for **endpoint data** (IP:Port)

A **socket (https://en.wikipedia.org/wiki/Network_socket)** is the software representation of that endpoint.

Opening a **socket** creates a kind of transceiver that can send and/or receive bytes at a given IP address and Port.

Sockets in Python

Python provides a standard library module which provides socket functionality. It is called **socket**.

The library is really just a very thin wrapper around the system implementation of *BSD Sockets (https://en.wikipedia.org/wiki/Berkeley_sockets)*.

Let's spend a few minutes getting to know this module.

Follow along in your own notebook!

The Python sockets library allows us to find out what port a service uses:

In [1]:
```
import socket

socket.getservbyname('ssh')
```

Out[1]:
```
22
```

You can also do a *reverse* lookup, finding what service uses a given port:

In [2]:
```
socket.getservbyport(80)
```

Out[2]:
```
'http'
```

The sockets library also provides tools for finding out information about *hosts*. For example, you can find out about the hostname and IP address of the machine you are currently using:

In [3]:
```
socket.gethostname()
```

Out[3]:
```
'EM0082'
```

In [4]:
```
socket.gethostbyname(socket.gethostname())
```

Out[4]:
```
'128.95.239.95'
```

You can also find out about machines that are located elsewhere, assuming you know their hostname. For example:

In [5]:
```
socket.gethostbyname('google.com')
```

Out[5]:
```
'216.58.193.110'
```

In [6]:
```
socket.gethostbyname('uw.edu')
```

Out[6]:
```
'128.95.155.134'
```

In [7]:
```
socket.gethostbyname('jaschilz.net')
```

Out[7]:
```
'104.28.18.112'
```

The `gethostbyname_ex` method of the `socket` library provides more information about the machines we are exploring:

In [8]:
```
socket.gethostbyname_ex('jaschilz.net')
```

Out[8]:
```
('jaschilz.net', [], ['104.28.18.112', '104.28.19.112'])
```

In [9]:
```
socket.gethostbyname_ex('google.com')
```

Out[9]:
```
('google.com', [], ['216.58.193.110'])
```

To create a socket, you use the `socket` method of the `socket` library. It takes up to three optional positional arguments (here we use none to get the default behavior):

In [10]:
```
foo = socket.socket()
foo
```

Out[10]:
```
<socket.socket fd=1140, family=AddressFamily.AF_INET, type=SocketKind.SOC
K_STREAM, proto=0>
```

Intermezzo: Representation at the REPL

Let's make sure that everyone knows why typing `foo` into the Python gives us something pretty like:

In [11]:
```
foo
```

Out[11]:
```
<socket.socket fd=1140, family=AddressFamily.AF_INET, type=SocketKind.SOC
K_STREAM, proto=0>
```

But when we instance a custom class we get something uglier like:

In [12]:
```
class MyNumberWrapper:
    def __init__(self, number):
        self.number = number

bar = MyNumberWrapper(5)

bar
```

Out[12]:
```
<__main__.MyNumberWrapper at 0x4c304b0>
```

The Python Data Model (https://docs.python.org/3/reference/datamodel.html) defines multiple "special methods (https://docs.python.org/3/reference/datamodel.html#special-method-names)" that help the interpretter accomplish "behind the scenes" work with objects.

For example, we could define `MyNumberWrapper.__add__(self, other)` to define how the interpretter should statements like `my_number_wrapper_1 + my_number_wrapper_2`.

One of these methods is `object.__repr__(self)`. This is the object method that the command line uses to create a "representation" of the object.

When we define `MyNumberWrapper2.__repr__(self)`, the REPL will be able to provide a representation of our object:

In [13]:
```
class MyNumberWrapper2(MyNumberWrapper):
    def __repr__(self):
        return "MyNumberWrapper2({number})".format(number=self.number)

baz = MyNumberWrapper2(5)

baz
```

Out[13]:
```
MyNumberWrapper2(5)
```

Back to Sockets!

A socket has some properties that are immediately important to us. THese include the family, type, and protocol of the socket:

In [14]:
```
foo.family
```

Out[14]:
```
<AddressFamily.AF_INET: 2>
```

In [15]:
```
foo.type
```

Out[15]:
```
<SocketKind.SOCK_STREAM: 1>
```

In [16]:
```
foo.proto
```

Out[16]:
```
0
```

You might notice that the values for these properties are integers. In fact, these integers are **constants** defined in the socket library.

We can dump a listing of the attributes of socket:

In [17]:
```
dir(socket)
```

Out[17]:
```
['AF_APPLETALK',
 'AF_DECnet',
 'AF_INET',
 'AF_INET6',
 'AF_IPX',
 'AF_IRDA',
 'AF_SNA',
 'AF_UNSPEC',
 'AI_ADDRCONFIG',
 'AI_ALL',
 'AI_CANONNAME',
 'AI_NUMERICHOST',
 'AI_NUMERICSERV',
 'AI_PASSIVE',
 'AI_V4MAPPED',
 'AddressFamily',
 'CAPI',
 'EAGAIN',
 'EAI_AGAIN',
```

Let's create a tool to help us navigate these constants. It will take a single argument, the shared prefix for a defined set of constants:

In [18]:
```
def get_constants(prefix):
    """filtered mapping of socket module constants to their names"""
    return {
        getattr(socket, name): name
        for name in dir(socket) if name.startswith(prefix)
    }
```

Socket Families

Think back a moment to our discussion of the *Internet* layer of the TCP/IP stack. There were a couple of different types of IP addresses:

- IPv4 (‘192.168.1.100’)
- IPv6 (‘2001:0db8:85a3:0042:0000:8a2e:0370:7334’)

The **family** of a socket corresponds to the *addressing system* it uses for connecting.

Families defined in the `socket` library are prefixed by `AF_`:

In [19]:
```
families = get_constants('AF')

families
```

Out[19]:
```
{<AddressFamily.AF_UNSPEC: 0>: 'AF_UNSPEC',
 <AddressFamily.AF_INET: 2>: 'AF_INET',
 <AddressFamily.AF_IPX: 6>: 'AF_IPX',
 <AddressFamily.AF_SNA: 11>: 'AF_SNA',
 12: 'AF_DECnet',
 <AddressFamily.AF_APPLETALK: 16>: 'AF_APPLETALK',
 <AddressFamily.AF_INET6: 23>: 'AF_INET6',
 <AddressFamily.AF_IRDA: 26>: 'AF_IRDA'}
```

Your results may vary

Of all these, the ones we care about most are `2` (IPV4) and `30` (IPV6).

When you are on a machine with an operating system that is Unix-like, you will find another generally useful socket family: `AF_UNIX` or Unix Domain Sockets (https://en.wikipedia.org/wiki/Unix_domain_socket). Sockets in this family:

- connect processes **on the same machine**
- are generally a bit slower than IPC (https://en.wikipedia.org/wiki/Inter-process_communication) connections.
- have the benefit of allowing the same API for programs that might run on one machine **or** across the network
- use an 'address' that looks like a pathname ('/tmp/foo.sock')

What is the family for the socket we created just a moment ago?

What is the *default* family for a socket created using the `socket` library?

How did you figure this out?

Socket Types

The socket *type* determines the syntax of socket communication.

Look up socket type constants with the `SOCK_` prefix:

In [20]:
```
types = get_constants('SOCK_')

types
```

Out[20]:
```
{<SocketKind.SOCK_STREAM: 1>: 'SOCK_STREAM',
 <SocketKind.SOCK_DGRAM: 2>: 'SOCK_DGRAM',
 <SocketKind.SOCK_RAW: 3>: 'SOCK_RAW',
 <SocketKind.SOCK_RDM: 4>: 'SOCK_RDM',
 <SocketKind.SOCK_SEQPACKET: 5>: 'SOCK_SEQPACKET'}
```

The most common are `1` (Steam communication(TCP)) and `2` (Datagram communication (UDP)).

What is the *default* family for a socket created using the `socket` library?

Socket Protocols

A socket also has a designated *protocol*. The constants for these are prefixed by `IPPROTO_`:

In [21]:
```
protocols = get_constants('IPPROTO_')

protocols
```

Out[21]:
```
{0: 'IPPROTO_IP',
 1: 'IPPROTO_ICMP',
 6: 'IPPROTO_TCP',
 17: 'IPPROTO_UDP',
 255: 'IPPROTO_RAW'}
```

The choice of which protocol to use for a socket is determined by the *internet layer* protocol you intend to use: `TCP`, `UDP`, `ICMP`, `IGMP`.

What is the *default* family for a socket created using the `socket` library?

Customizing Sockets

These three properties of a socket correspond to the three positional arguments you may pass to the socket constructor.

Using them allows you to create sockets with specific communications profiles:

In [22]:
```
socket.socket(socket.AF_INET,
              socket.SOCK_DGRAM,
              socket.IPPROTO_UDP)
```

Out[22]:
```
<socket.socket fd=772, family=AddressFamily.AF_INET, type=SocketKind.SOCK
_DGRAM, proto=17>
```

Break Time

So far we have:

- learned about the "layers" of the TCP/IP Stack
- discussed *families*, *types* and *protocols* in sockets
- learned how to create sockets with a specific communications profile

When we return, we'll learn how to find the communications profiles of remote sockets, how to connect to them, and how to send and receive messages.

Take a few minutes now to clear your head (do not quit your Python interpretter).

Address Information

When you are creating a socket to communicate with a remote service, the remote socket will have a specific communications profile.

The local socket you create must match that communications profile.

You ask: How can you determine the *correct* values to use?

The function `socket.getaddrinfo` provides information about available connections of a given host:

In [23]:
```
socket.getaddrinfo('google.com', 80)
```

Out[23]:
```
[(<AddressFamily.AF_INET: 2>, 0, 0, '', ('216.58.193.110', 80))]
```

This provides all you need to make a proper connection to a socket on a remote host. The value returned is a tuple of:

- socket family
- socket type
- socket protocol
- canonical name (usually empty, unless requested by flag)
- socket address (tuple of IP and Port)

Again, let's create a utility method in-place so we can see this in action:

In [24]:
```
def get_address_info(host, port):
    for response in socket.getaddrinfo(host, port):
        family, s_type, protocol, name, address = response
        print('family: {}'.format(families[family]))
        print('type: {}'.format(types[s_type]))
        print('protocol: {}'.format(protocols[protocol]))
        print('canonical name: {}'.format(name))
        print('socket address: {}'.format(address))
        print('')
```

Now, ask your own machine what possible connections are available for 'http':

In [25]:
```
get_address_info(socket.gethostname(), 'http')
```

```
family: AF_INET
type: SOCK_STREAM
protocol: IPPROTO_IP
canonical name:
socket address: ('128.95.239.95', 80)

family: AF_INET6
type: SOCK_STREAM
protocol: IPPROTO_IP
canonical name:
socket address: ('::1', 80, 0, 0)
```

Try a few other servers you know about.

Client Side

Let's put this to use and communicate with a remote server as a *client*

Construct a Socket

We've already made a socket `foo` using the generic constructor without any arguments. We can now make one using real address information from a real server online:

In [26]:
```
streams = [
    info for info in socket.getaddrinfo('jaschilz.net', 'http')
         if info[1] == socket.SOCK_STREAM
]

streams
```

Out[26]:
```
[(<AddressFamily.AF_INET: 2>,
  <SocketKind.SOCK_STREAM: 1>,
  0,
  '',
  ('104.28.18.112', 80)),
 (<AddressFamily.AF_INET: 2>,
  <SocketKind.SOCK_STREAM: 1>,
  0,
  '',
  ('104.28.19.112', 80))]
```

In [27]:
```
info = streams[0]
jaschilz_socket = socket.socket(*info[:3])
```

Connecting a Socket

Once a socket is constructed with the appropriate *family*, *type*, and *protocol*, we can connect it to the address of our remote server:

In [28]:
```
jaschilz_socket.connect(info[-1])
```

- a successful connection returns `None`
- a failed connection raises an error
- you can use the *type* of error returned to tell why the connection failed.

Sending a Message

Send a message to the server on the other end of the our connection (we'll learn in session 2 about the message we are sending):

In [29]:
```
msg = "GET / HTTP/1.1\r\n"
msg += "Host: jaschilz.net\r\n\r\n"
msg = msg.encode('utf8')
msg
```

Out[29]:
```
b'GET / HTTP/1.1\r\nHost: jaschilz.net\r\n\r\n'
```

In [30]:
```
jaschilz_socket.sendall(msg)
```

- the transmission continues until all data is sent or an error occurs
- success returns `None`
- failure to send raises an error
- the type of error can tell you why the transmission failed
- but you **cannot** know how much, if any, of your data was sent

Receiving a Reply

Whatever reply we get is received by the socket we created. We can read it back out:

In [31]:
```
response = jaschilz_socket.recv(4096)
response[:60]
```

Out[31]:
```
b'HTTP/1.1 200 OK\r\nDate: Thu, 31 Mar 2016 21:20:54 GMT\r\nConten'
```

- The sole required argument is buffer_size (an integer). It should be a power of 2 and smallish (~4096)
- It returns a byte string of buffer_size (or smaller if less data was received)
- If the response is longer than buffer size, you can call the method repeatedly. The last bunch will be less than buffer size.

Messages Are Bytes

One detail from the previous code should stand out:

```
msg = msg.encode('utf8')
msg
b'GET / HTTP/1.1\r\nHost: jaschilz.net\r\n\r\n'
```

You can **only** send bytes through a socket, **never** unicode:

In [32]:
```
jaschilz_socket.sendall('regular old py3 unicode string')
```

```
---------------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-32-03c983c1ada5> in <module>()
----> 1 jaschilz_socket.sendall('regular old py3 unicode string')

TypeError: a bytes-like object is required, not 'str'
```

Cleaning Up

When you are finished with a connection, you should always close it:

In [34]:
```
jaschilz_socket.close()
```

Putting It All Together

First, connect and send a message

In [35]:
```
# Gather info about the server's available sockets
info = socket.getaddrinfo('jaschilz.net', 'http')

# Filter out those sockets that are not TCP (SOCK_STREAM)
streams = [i for i in info if i[1] == socket.SOCK_STREAM]

# Get the first of those streams
sock_info = streams[0]

# Construct a message (mind the new-lines)
msg = "GET / HTTP/1.1\r\nHost: jaschilz.net\r\n\r\n".encode('utf8')

# Construct a socket with family, type, and protocol to match host
jschilz_socket = socket.socket(*sock_info[:3])

# Connect
jschilz_socket.connect(sock_info[-1])

# Send
jschilz_socket.sendall(msg)
```

Then, receive a reply, iterating until it is complete:

In [36]:
```
buffer_size = 4096
response = b''
done = False

while not done:
    msg_part = jschilz_socket.recv(buffer_size)
    if len(msg_part) < buffer_size:
        done = True
        jschilz_socket.close()
    response += msg_part
```

Now let's see what's inside:

In [37]:
```
len(response)
```

Out[37]:
```
7300
```

In [38]:
```
response
```

Out[38]:

b'HTTP/1.1 200 OK\r\nDate: Thu, 31 Mar 2016 21:21:22 GMT\r\nContent-Type:
text/html; charset=utf-8\r\nTransfer-Encoding: chunked\r\nConnection: kee
p-alive\r\nSet-Cookie: __cfduid=d0d269bf36107fe710fff01eeae6fe4b614594592
82; expires=Fri, 31-Mar-17 21:21:22 GMT; path=/; domain=.jaschilz.net; Ht
tpOnly\r\nContent-Language: en\r\nVary: Accept-Language,Cookie,Accept-Enc
oding\r\nServer: cloudflare-nginx\r\nCF-RAY: 28c6d9c3432a1b9d-SEA\r\n\r\n
4976\r\n<!doctype html>\n<html lang="en">\n\n\n<head>\n<meta http-equiv
="Content-type" content="text/html; charset=utf-8">\n<meta name="viewpor
t" content="width=device-width, initial-scale=1.0">\n<meta name="keyword
s" content="">\n<meta name="description" content="">\n<title>Blog | JASch
ilz.net</title>\n<script type="text/javascript">\n//<![CDATA[\ntry{if (!w
indow.CloudFlare) {var CloudFlare=[{verbose:0,p:0,byc:0,owlid:"cf",bag2:
1,mirage2:0,oracle:0,paths:{cloudflare:"/cdn-cgi/nexp/dok3v=1613a3a18
5/"},atok:"142a0ebb8dc292ba238a08accf805d19",petok:"5bb29ef82a380784f4f07
f3af3e9c0773e79ee52-1459459282-1800",zone:"jaschilz.net",rocket:"0",apps:
{"ga_key":{"ua":"UA-54041823-2","ga_bs":"2"}},sha2test:0}];!function(a,b)
{a=document.createElement("script"),b=document.getElementsByTagName("scri
pt")[0],a.async=!0,a.src="//ajax.cloudflare.com/cdn-cgi/nexp/dok3v=e98291
3d31/cloudflare.min.js",b.parentNode.insertBefore(a,b)}()}}catch(e)
{};\n//]]>\n</script>\n<link rel="shortcut icon" href="/static/media/uplo
ads/favicon.ico">\n\n\n\n<link rel="alternate" type="application/rss+xml"
title="RSS" href="/feeds/rss/">\n<link rel="alternate" type="application/
atom+xml" title="Atom" href="/feeds/atom/">\n\n\n\n\n\n\n<link rel="style
sheet" href="/static/css/base.css">\n\n\n\n\n\n\n\n\n\n\n\n\n\n<script sr
c="/static/mezzanine/js/jquery-1.7.1.min.js"></script>\n<script src="/sta
tic/js/bootstrap.js"></script>\n<script src="/static/js/bootstrap-extras.
js"></script>\n\n\n\n<!--[if lt IE 9]>\n<script src="/static/js/html5shi
v.js"></script>\n<script src="/static/js/respond.min.js"></script>\n<![en
dif]-->\n\n\n<script type="text/javascript">\n/* <![CDATA[*/\nvar _gaq =
_gaq || [];\n_gaq.push([\'_setAccount\', \'UA-54041823-2\']);\n_gaq.push
([\'_trackPageview\']);\n\n(function() {\nvar ga = document.createElement
(\'script\'); ga.type = \'text/javascript\'; ga.async = true;\nga.src =
(\'https:\' == document.location.protocol ? \'https://ssl\' : \'http://ww
w\') + \'.google-analytics.com/ga.js\';\nvar s = document.getElementsByTa
gName(\'script\')[0]; s.parentNode.insertBefore(ga, s);\n})();\n\n(functi
on(b){(function(a){"__CF"in b&&"DJS"in b.__CF?b.__CF.DJS.push(a):"addEven
tListener"in b?b.addEventListener("load",a,!1):b.attachEvent("onload",
a)})(function(){"FB"in b&&"Event"in FB&&"subscribe"in FB.Event&&(FB.Even
t.subscribe("edge.create",function(a){_gaq.push(["_trackSocial","faceboo
k","like",a])}),FB.Event.subscribe("edge.remove",function(a){_gaq.push
(["_trackSocial","facebook","unlike",a])}),FB.Event.subscribe("message.se
nd",function(a){_gaq.push(["_trackSocial","facebook","send",a])}));"twtt
r"in b&&"events"in twttr&&"bind"in twttr.events&&twttr.events.bind("twee
t",function(a){if(a){var b;if(a.target&&a.target.nodeName=="IFRAME")a:{if
(a=a.target.src){a=a.split("#")[0].match(/[^?=&]+=([^&]*)?/g);b=0;for(var
c;c=a[b];++b)if(c.indexOf("url")===0){b=unescape(c.split("=")[1]);break
a}}b=void 0}_gaq.push(["_trackSocial","twitter","tweet",b])}})})})(windo
w);\n/*]]> */\n</script>\n</head>\n\n<body id="body">\n\n\n\n\n\n\n\n\n
\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n
\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n<
div id="container">\n\n<div id="left">\n\n<img style="padding
-bottom:5px; padding-left:5px" src="/static/img/logoDescartes.png" alt="D
escartes illustration of theory of mind.">
\n\n<img cl

ass="centered" style="margin-bottom:5px" src="/static/img/home.gif">
\n\n<img class="centered" src="/static/img/essa
ys.gif" alt="Essays Header Img">\n<!--What is Mat
hematics?
\nWhat are the Real Numb
ers?
Exploring ℝ, via 0.999...≟1.
-->\n <a href
="/pages/essays/#mindAndFunction">Mind and Function
Defense agains
t Searle et al.
 \n <a href="/pages/essays/#studentGrowthAndTeach
erEvaluation">How Certain Can You Be?
Stats of student growth.
\n\n<img class="centered leftnav" src="/
static/img/projects.gif" alt="Projects Header Img">\n <a href="/pag
es/projects/#InterSIS">InterSIS
Open student information API.
\n Sigma Education
 \n RSSNext.net
Your next
unread item.
 \n Faces
An
AI driven artwork.
 \n Rover: A Py
thon MUD
 \n XPR
OO lay
er for Psychtoolbox.
 \n A Mod
ern Trivium
Foundations of learning.
 \n \n\n<a href= "/p
ages/links/"><img class="centered leftnav" src="/static/img/links.gif" al
t="Links Header Img">\n Physics &am
p; Metaphysics
 \n<!--Psychology

-->\n Blogs
 \n <a hr
ef="/pages/links/#Art">Art
 \n <a href="https://github.com/JA
Schilz">JASchilz GitHub<img src="/static/img/xternalLink.svg" alt="Li
nks Header Img">
\n \n\n <!-- End left -->\n\n
\n\n<h1>JASchilz.net</h1>\n\n <!-- End top -->\n\n
\n <!-- End right -->\n\n \n   <!-- Posts and stu
ff -->\n   \n\n<h1>Home/Log</h1>\n\n\n   \n\n\n\n   \n   \n   <h2>\n
21 Nov. 2014: The View Fr
om My Companionway Hatch\n   </h2>\n   \n   \n   \n   \n   \n
\n   \n   \n   \n   \n   \n   \n   \n   \n   \n   \n   \n
\n   \n   \n   \n\n   \n\n   \n\n   <img src="/static/media/uplo
ads/boat.jpg" height="337" width="600"> \n Downtown Se
attle over Lake Union. The view from my companionway hatch as I sit progr
amming. \n\n   \n\n   \n   \n
\n   \n   \n\n   \n   \n   \n   \n\n
\n   \n   <h2>\n   29 June 2013: <a href="/when-you-ask-a-student-
a-question/">When You Ask a Student a Question\n   </h2>\n   \n
\n   \n   \n   \n   \n   \n   \n   \n   \n   \n   \n   \n
\n   \n   \n   \n   \n   \n   \n\n   \n\n   \n\n   Incl
uded in a post from <a href="http://anglesofreflection.blogspot.com/2013/
06/the-four-most-important-words-in.html">a blog that I follow: \n
[A mentor advised me that] when you ask a student a pers
onal question, you need to be aware of your own stake in the answer. What
he meant, roughly, was that when you ask a student a personal question, y
ou need to be aware of your own reasons: are you asking because talking w
ill help the student, or for the emotional buzz of reinforcing your relat
ionship with the student, or to validate your own self-image as "the teac
her who cares." It\'s easy, he warned, to think of yourself as asking for
the student\'s benefit, when really it\'s about you, which is problemati
c: as an authority figure, you\'re in a position to demand a response, ev
en when you shouldn\'t. \n\n   \n\n   \n   <div class="blog-li'

We've just created a web browser! (kind of)

Server Side

What about the other half of the equation?

Let's build a server and see how that part works.

Construct a socket

Again, we begin by constructing a socket. Since we are constructing a server this time, we get to choose the family, type, and protocol.

Because we'll be communicating between a server process and a client process, we'll need to run one of them in another interpretter. So open a command-line Python interpretter and we will create the following server.

All code in triple-double-quotes should be typed into this other interpretter.

In [39]:
```
"""
import socket

server_socket = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM,
    socket.IPPROTO_TCP
)

server_socket
"""
```

Out[39]:
```
'\nimport socket\n\nserver_socket = socket.socket(\n    socket.AF_INET,\n
socket.SOCK_STREAM,\n    socket.IPPROTO_TCP\n)\n\nserver_socket\n'
```

Bind the socket

Our server needs to be **bound** to an address. This it the *IP address* and *port* to which clients must connect:

In [40]:
```
"""
address = ('127.0.0.1', 50000)
server_socket.bind(address)
"""
```

Out[40]:
```
"\naddress = ('127.0.0.1', 50000)\nserver_socket.bind(address)\n"
```

Terminology: In a server/client relationship, the server *binds* to an address and port. The client *connects*.

Listen for Connections

Once our socket is bound to an address, we can listen for attempted connections:

In [41]:
```
"""
server_socket.listen(1)
"""
```

Out[41]:
```
'\nserver_socket.listen(1)\n'
```

- The argument to `listen` is the *backlog*
- The *backlog* is the **maximum** number of connection requests that the socket will queue
- Once the limit is reached, the socket refuses new connections.

Accept a Connection

When a socket is listening, it can receive incoming connection requests:

In [42]:
```
"""
connection, client_address = server_socket.accept()
"""
```

Out[42]:
```
'\nconnection, client_address = server_socket.accept()\n'
```

- The call to `socket.accept()` is a *blocking* call. It will not return values until a client connects
- The `connection` returned by a call to `accept` is a **new socket**. This new socket is used to communicate with the client
- The `client_address` is a two-tuple of IP address and port for the client socket
- When a connection request is 'accepted', it is removed from the backlog queue.

  Our server is now awaiting a client connection!

Create a Client

Now we'll create a client in our notebooks.

In [43]:
```
import socket
client_socket = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM,
    socket.IPPROTO_IP)
```

Before connecting, keep on eye your your server interpreter.

In [44]:
```
client_socket.connect(('127.0.0.1', 50000))
```

Now that we have created a connection, our call to `socket.accept()` should have finally returned a connection socket.

Check the interpreter containing your server to confirm that `socket.accept()` has returned.

Talking to Yourself

Now we have a pipe between our two interpreters. We can push bytes into one end of the pipe and receive them on the other end.

Each side of this pipe is terminated by a *socket*:

- The client end of this pipe is here in our notebook and called `client_socket`.
- The server end of this pipe is in our command line interpreter and called `connection`.

Now try sending a message from the client to the server:

In [45]:
```
client_socket.sendall('Hey, can you hear me?'.encode('utf8'))
```

Now go into the server interpreter and receive the message:

In [46]:
```
"""
buffer_size=4096
connection.recv(buffer_size)
"""
```

Out[46]:
```
'\nbuffer_size=4096\nconnection.recv(buffer_size)\n'
```

And then send a reply from the server:

In [47]:
```
"""
connection.sendall("message received".encode('utf8'))
"""
```

Out[47]:
```
'\nconnection.sendall("message received".encode(\'utf8\'))\n'
```

And receive it in the client:

In [49]:
```
from_server = client_socket.recv(buffer_size)
from_server
```

Out[49]:
```
b'message received'
```

You've run your first client-server interaction!

Cleaning Up

Now that you've sent a few messages between the client and the server, let's close the connection.

On the server side:

In [51]:
```
"""
connection.close()
"""
```

Out[51]:
```
'\nconnection.close()\n'
```

On the client side:

In [52]:
```
client_socket.close()
```

And then let's also close our 'listener' on the server:

In [53]:
```
"""
server_socket.close()
"""
```

Out[53]:
```
'\nserver_socket.close()\n'
```

That's it! You're now ready to begin your homework.

In [ ]: