Journeyman's Python

Now we're cooking with grease.

Topics to cover:

- Data structures in Python - what are they, how they work, how to use them.
- File I/O - storing data on disk, and retrieving it.
- Generators & Iterators - Bit weird, but extremely useful.
- Networking
  - Sockets
    - IPv4/v6
    - TCP/UDP
  - Socket functions
- Classes
- Exception Handling
- Special Functions - lambda, with

Python Data structures

- What is a data structure?
  - A logical means of organizing data.
  - Almost every programming language makes use of them to efficiently store and use data.
- How does Python use them?
  - You can create your own with classes. But right now, we're focusing on the built-in Python Data Structures.

Python Built-in Data Structures

- Lists
  - We’ve dealt with these already, as lists are a fundamental aspect of Python, but understanding how they actually work is important.
  - A list is an array (in Python’s help pages, it’s referred to as a “mutable sequence”) which is stored in a specific order. A list created as [‘a’ , ‘b’ , ‘c’] will always be [‘a’ , ‘b’ , ‘c’] unless you change it.

Python Built-in Data Structures (Cont)

- Dictionaries
  - Data structures made up of *"key: value"* pairs. The Key can be whatever you want (mostly, there are some exceptions) and is how you will reference the value.
  - Dictionary key values are hashed and stored in order based on that hash. If you create a dictionary as {1:'a' , 2:'b' , 3:'c'}, it may print correctly, or it may print as {2:'b' , 1:'a' , 3:'c'}, but the key:value will always be properly matched.

Files in Python

- File I/O is a fundamental aspect of programming, it’s needed to store data on disk, and to read in configurations and settings.
- File objects in Python are implemented with the C stdio package, meaning that the Python file commands map almost directly to C commands for the same tasks.
- In Linux, Python’s open() maps to C’s fopen()
- In Windows, Python’s open() goes through the WinAPI CreateFile function.

Opening a File

- open(name[, mode[, buffering]])
  - same thing as the file(), but is preferable according to Python documentation
- name - filename
- mode - how you’re opening the file.
  - Common Options: w (write) , r (read) , a (append)
  - you can add a ‘+’ to your option, which allows read/write to be done with the same object
- buffering - if used, determines the file buffer size (usually unnecessary)

File Methods

- These are the various functions which can be used on a file object.
- file.write(string) - prints a string to a file, there is no return
- file.read([bufsize]) - read up to “bufsize” number of bytes from the file. If run without the buffer size option, read the entire file
- file.readline([bufsize]) - Read one line from the file (keeps the newline)
- file.close() - close the file and destroy the file object. Python will do this automatically, but it’s still good practice when you’re done with a file

Generators

- A generator is a function which doesn’t return immediately. Instead, it “yields” a return value, while continuing to run a loop of code until complete. Essentially, what this means is that it gives back data without ending its own execution. This can be very handy in certain situations, though tricky to master.

Iterators

- An Iterator is a method of going over through a data array one item at a time.
- We’ve actually covered the most common iteration method, “for”. There are others, but they aren’t especially important now.

Networking

Take a break, clear your mind. Get some chips

Networking

- Computers without networking are basically just big calculators.
- Computers with networking are also basically just big calculators, but they let us watch cat videos online.

Essentials of Networking

IPv4

- 32 bits (4,294,967,296 possible IPs)
- Old, but used everywhere
- Covered in RFC 791
- No built in security, trivial checksum

IPv6

- 128 bits (340,282,366,920,938,463,463,374,607,431,768,211,456 possible IPs)
- Not used everywhere, for a variety of reasons
- Covered in RFC 2460
- Built in security, less trivial checksum

Essentials of Networking

TCP

- Connection-oriented
- “Reliable” (Sort of)
- Sequence & Acknowledgement #’s
- Governed by RFC 793
- One-to-one communication

UDP

- Connectionless
- “Unreliable” (Not exactly)
- No seq/ack #’s
- Governed by RFC 768
- One-to-one or one-to-many communication

Essentials of Networking (Useful Ports)

- 20/21 - FTP
- 22 - ssh
- 23 - telnet
- 25 - SMTP
- 53 - DNS
- 69 - TFTP
- 80 - HTTP
- 135 - RPC
- 137-139 - NetBIOS
- 179 - BGP
- 1080 - SOCKS
- 6665-6669 - IRC

Lesser known Protocols

- HTCPCP - Hyper Text Coffee Pot Control Protocol
- Notable error: “I’m a teapot”
- governed by rfc 7168

- IPoAC (IP over Avian Carriers)
- A communication protocol using carrier pigeons
- governed by rfc 1149

Sockets

- What is a socket?
  - The Berkeley Software Distribution (BSD) socket is the primary means of communication between computers. It’s a virtual interface between the network stack and the application.

Sockets

- What is Binding?
  - Associating a socket with a specific port, this is necessary for servers which need to always be available from the same port so that they can be reached. It is not usually necessary in clients (though it is acceptable)

Python Sockets

- import socket
    - the socket module, discussed in previous videos, is the library for all socket operations, and defines for Python the means of accessing BSD sockets for network communication. Any networking program will have this import.

Python Sockets

- sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
  - sock - the variable name for the socket object which will be returned
  - socket.socket() - the constructor function for socket objects
  - socket.AF_INET - IPv4
    - socket.AF_INET6 - IPv6
  - socket.SOCK_STREAM - TCP
    - socket.SOCK_DGRAM -UDP

Python Sockets

- socket.bind((HOST , PORT))
  - No variable, you can (and typically should) error check with this function, but it doesn’t return any objects, so there’s no need to store anything
  - socket.bind() - associates a socket object with a specific interface and port
  - HOST - the IP/interface with which to associate
  - PORT - the logical address on an interface with which to associate

Python Sockets

- socket.listen(listenqueue)
  - socket.listen() - sets the socket object in blocking mode waiting for a client to attempt connnection
  - listenqueue - determines the number of connections to permit while waiting for acceptance. Older systems need a lower number, but modern computers can pretty much listen for as many as necessary

Python Sockets

- conn, addr = socket.accept()
  - conn - a new socket object, this one is for a specific client connection, whereas the “sock” object was for our server to listen
  - addr - address information about the client (IP, port, etc)
  - socket.accept() - this is where a connection is fully negotiated, and data can begin travelling back and forth

Python Sockets

- socket.send(string)
  - socket.send() - sends a buffer to the specified target and returns the number of bytes sent. This return value is a useful error check, make sure you send as much as you think you do.
  - string - this can be a hard-coded string, or it can be a variable you define and populate elsewhere

Python Sockets

- socket.recv(bufsize[, flags])
    - socket.recv() - receives data from a connection.
    - bufsize - the amount of data to accept
    - flags - we’re not going to worry about this now. It defaults to 0, and that’s the only value most people will ever use.

Python Sockets

- socket.close()
  - The last socket function to use, this terminates the connection, and frees up the memory. The interpreter will do this automatically, but again; good habits are good habits.

Exceptions and Classes

I can see the light at the end of the tunnel

Exceptions

- An Exception is what happens when your code causes an error at execution. (I.E. If you attempt to access a file which doesn’t exist).
- There are two sides to working with Exceptions: Handling them, and creating them.

Handling Exceptions

```
try:
        some code
except (some error):
        some other code
finally:
        some code you really need to run
```

Creating Exceptions

- To fully understand Creating Exceptions, you have to understand Classes. So, we're going to go on to Classes, and then come back and use that to create a custom Exception.

Classes

- Classes are an essential part of Python’s Object Oriented nature. They allow the user to create objects with specific attributes and functions (called methods) quickly and easily. They are somewhat analogous to C’s Structs, but a bit easier to understand.

Defining a new Class

- The first-order parent class is “object”. Every other class descends from it.
- You create a new class like this:

```
class MyClass(object):
    <Code Goes Here>
```

Attributes

- Attributes are data fields associated with a class.

class MyClass(object):
variable = ‘’ ⇐ This is a string attribute

Methods

- Methods are like Attributes, but instead of data fields, they're functions.

```
class MyClass(object):
        def printone(self): ⇐ "self" is the first arg in any method
                print "1"
                return
```

Inheriting

- A class can inherit from other classes (That's one of the main principles of Object Oriented Programming)

```
class MyClass(object)
class MyOtherClass(MyClass)
```

Putting them together!

- Here’s a bit of a twist on things: We can combine the information we just learned to create our own class which is also…. (drumroll), an Exception!

Creating An Exception

```
class CustomError(Exception):
        def __init__(self, error):
                super(MyError, self).__init__(error)
                self.errormessage = error
        def __str__(self):
                return "Error: %s" % self.msg
```

Some Extras

Lambda, command line, and list comprehension

Lambda

- Lambda is a way of creating “anonymous functions”. That is, you can use it to create functions which are not bound to a name.

```
lambda x: x + 1
```

Command-line arguments

- Taking arguments from the command line is a staple of “real” programming, and is a lot cleaner than print statements for every argument.

```
    import sys
    sys.argv
```

List Comprehension

- A quicker way to do something to every item in a list.

```
list = [1,2,3]
newlist = [i**2 for i in list]
```

ACTIVITIES

And now for something completely different.