John Romero Programming Proverbs

- 6. “As soon as you see a bug, you fix it. Do not continue on. If you don’t fix your bugs your new code will be built on a buggy codebase and ensure an unstable foundation.”

- John Romero, “The Early Days of Id Software - John Romero @ WeAreDevelopers Conference 2017”

If your ioquake is not allowing Pylego to disableAI

- then you need to download and install this tarball

```
$ ssh mcgreg.comp.glam.ac.uk
<enter your linux password>
$ cd $HOME/Sandpit
$ wget http://floppsie.comp.glam.ac.uk/download/c/ioquake-20161025.tar.gz
$ rm -rf ioquake-latest
$ tar zxf ioquake-20161025.tar.gz
$ exit
# your command line is back on the client
$ cd $HOME/Sandpit/ioquake-latest3/ioquake
$ ./compilequake
```

- please only do the above if you are experiencing problems getting pylego to connect

python-bot/bot-legoman/botfiles/bots/botlib.py

- enumerated type of rpc meta types:

- ```
  CHAR, WEIGHT, AI, BASIC, GOAL, COMMANDS = range(1, 7)
  ```

- CHAR
  - the packets contain characteristic information

- WEIGHT
  - not yet implemented, but it will upload weightings for weapons

- AI
  - codes are only just being implemented they consist of new commands necessary to enable Python to connect to ioquake
  ```

python-bot/bot-legoman/botfiles/bots/botlib.py

- the following meta types correspond to the AI layered architecture in ioquake (they should be extended when time permits)

- BASIC
  - basic actions which we can tell the bot to do (jump, crouch, fire)

- GOAL
  - interface to the goal logic of the AI engine.

- COMMANDS
  - give bot commands (as a team leader)

python-bot/bot-legoman/botfiles/bots/botlib.py

```
# AI codes
SKILL, CONT = range(1, 3)
# BASIC codes
JUMP, CROUCH, FIRE = range(1, 4)
```

class bot: __init__

```
def __init__ (self, server, port):
    global s

    self.init_chars()
    s = socket(AF_INET, SOCK_STREAM)
    print "bot trying to connect to the server",
    while True:
        try:
            s.connect((server, port))
            break
        except:
            print ".",
            sys.stdout.flush()
            time.sleep(1)
    print "bot connected"
```

class bot: init_chars (self)

- initialises all the characteristics which can be altered in the Python bot
- no data is sent to the ioquake3 server at this point
- the `botlib.py` is told the characteristic name, type, code, max, min values

Python bot

- recall our bot code can be simplified to

Python bot

```
id = botlib.bot("localhost", 7000)
print "hello world, python is alive in Quake 3"
id.defaults()
print "bot is now active!"
id.disableAI()
while True:
    print "trying to crouch"
    id.crouch()
    print "in crouch position"
    time.sleep(1)
    id.jump()
    print "in jump position"
    time.sleep(1)
    id.fire()
    print "fire"
    time.sleep(1)
```

class bot: defaults

- **ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py**

```
    def defaults(self):
        if self.skill() == 1:
            self.c_name.set("Pylego")
            self.c_gender.set("male")
            self.c_attack_skill.set(0.9)
            self.c_weaponweights.set("bots/Easy_w.c")
            self.c_aim_skill.set(0.95)
...
```

class bot: skill

- **ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py**

```
def skill (self):
    """ this must be the next method called after __init__ """
    print "sending skill"
    return callBI(AI, SKILL)
```

callBI

- **ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py**

```
#
#  callBI - makes a call and returns an byte integer.
#

def callBI (c1, c2):
    global s

    p = makeHeader(struct.pack("B", c1), struct.pack("B", c2))
    for i in p:
        print ord(i),
    print
    s.send(p)
    return getReturnByte(s)
```

- the `struct.pack("B",c1)` packs a Python integer `c1` into a byte

makeHeader

ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py

```
#
#  makeHeader - creates the rpc header
#

def makeHeader (meta, data = ""):
    s = meta + data
    return prependLength(s)
```

makeHeader

ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py

```
#
#  prependLength - places the length byte at the start of the packet
#

def prependLength (s):
    print "prepending length of", len(s)
    return struct.pack("B", len(s)) + s
```

getReturnByte

ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py

```
#
#  getReturnByte - defensively receives the return byte
#

def getReturnByte (s):
    p = ""
    while True:
        p = s.recv(1)
        if len(p) == 1:
            print "received packet of length", ord(p[0])
            break
    if ord(p[0]) == 2:
        while True:
            p = s.recv(1)
            if len(p) == 1:
                break
        printf("byte returned has value %d\n", struct.unpack("B", p)[0])
        return struct.unpack("B", p)[0]
    else:
        printf("expecting length of 2 received %d\n", ord(p[0]))
    return 0
```

Protocol

- first byte contains the packet length
- so the above code expects two bytes
  - a length byte of value, 2
  - the second byte is the data byte which is returned as an integer

setchar - low level function to send a characteristic to the ioquake3 server

ioquake-latest/python-bot/bot-legoman/botfiles/bots/botlib.py

```
#
#  setchar - send value using code, type, value to the server
#

def setchar (code, type, value):
    global s

    print "sending characteristic", code, type, value
    c = struct.pack("B", CHAR)
    d = struct.pack("B", code)
    if type=="int":
        s.send(makeHeader(c, d+struct.pack("i", value)))
    elif type=="float":
        s.send(makeHeader(c, d+struct.pack("f", value)))
    elif type=="string":
        s.send(makeHeader(c, d+struct.pack(("%ds" % (len(value)+1)), value)))
    if not getReturnBoolean(s):
        print "failed to set characteristic", code, "type", type, "value", value
```

ioquake3 server must match these rpc requests

- examine the file `$HOME/Sandpit/ioquake-latest/ioquake3/code/botlib/be_ai_py.c` and in particular start by understanding the function `initPy`
- then examine `testFor` and `waitForCont`
- notice that both are very similar
- notice that `execFunction` is called to handle the rpc
  - a return packet is sent back provided that the `execFunction` populated the return buffer
- understand that the rpc server is a state machine
  - `py->state` is set to `pyInit`, `pyOut`, `pyIn`

testFor

ioquake-latest/ioquake3/code/botlib/be_ai_py.c

```
int testFor (py_bot_t *py)
{
  if (py->state == pyInit) {
    py->in = 0;
    py->out = 0;
    py->used = 0;
    py->state = pyIn;
  }
  if (py->state == pyIn) {
    if (getPacket(py, qtrue))
      py->state = pyInit;
    else
      return qfalse;
    execFunction(py);
  }
```

testFor

ioquake-latest/ioquake3/code/botlib/be_ai_py.c

```
  if (py->state == pyOut) {
    if (putPacket(py, qtrue)) {
      py->out = 0;
      py->used = 0;
      py->state = pyInit;
    } else
      return qfalse;
  }
  return qtrue;
}
```

Tutorial

- read these notes carefully and open up emacs and read the actual code being discussed

- take a pen and paper and now create your own notes detailing how a Python call to the method `ourself` is executed
  - your notes should include both client and server activity