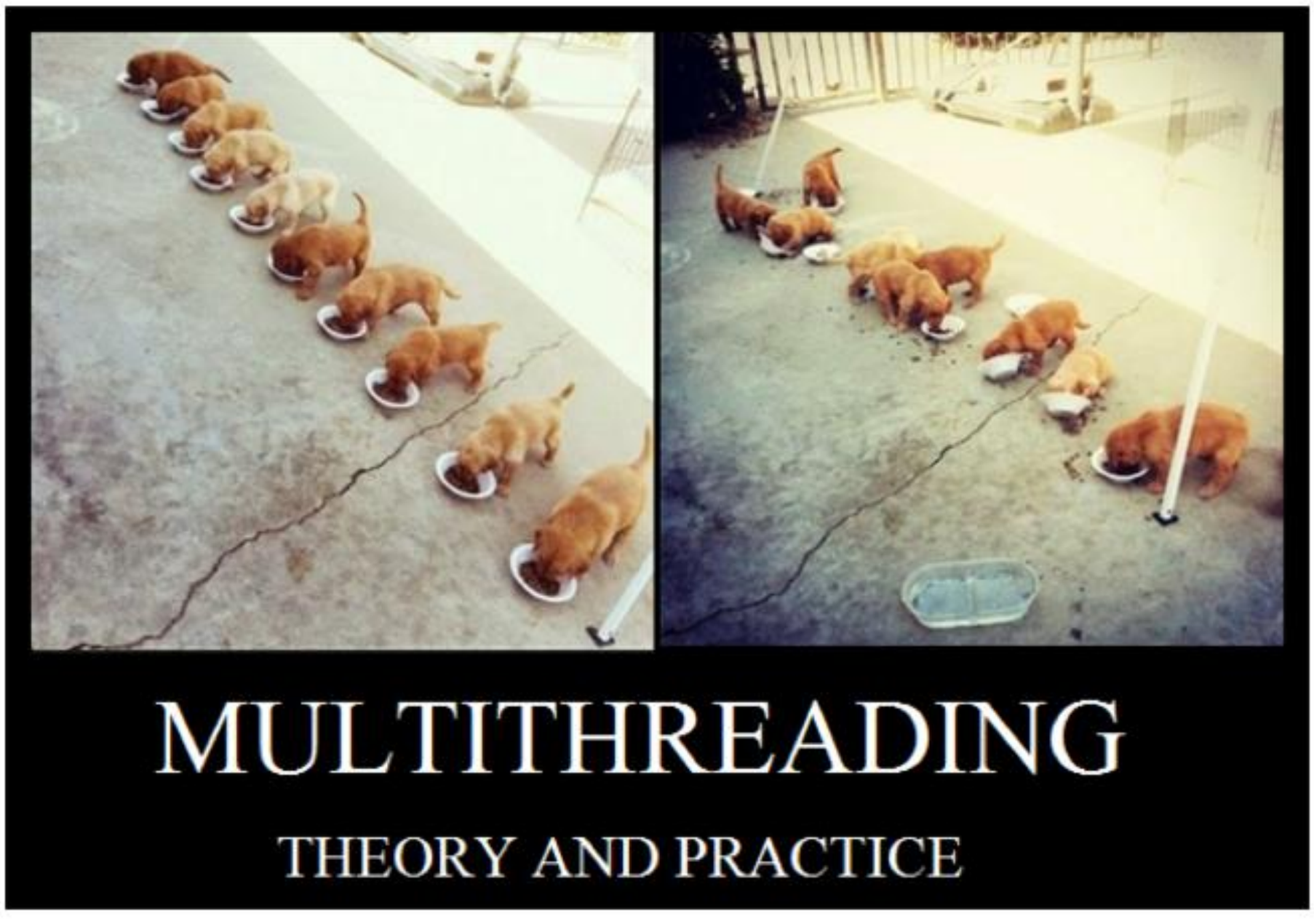
Python: Threads e Socket

Prof. Dr. Dilermando Piva Jr.

Threading

... Criando suas Threads...

Threading



Thread - Básico

- **threading** - módulo responsável por criar e controlar threads

- Comunicação facilitada entre nós de um mesmo processo

- Não precisa troca de contexto

- Compartilham o tempo de CPU igual ao de um processo

- Dependendo da tarefa, comportam-se muito melhor que processos.

- Processamento em paralelo é uma das áreas mais pesquisadas como saída para o limite tecnológico.

Thread - Básico

- Para implementar uma *thread* utilizando o *threading module*, basta extender a classe *Thread*
- Sobrescrever o método *__init__* com os parâmetros adicionais da *thread*
- Sobrescrever o método *run* implementando a lógica que a *thread* deverá executar ao iniciar

thread1.py

```
1   import threading, time
2
3   class MyThread(threading.Thread):
4       def __init__(self, times, delay):
5           threading.Thread.__init__(self)
6           self.counter = times
7           self.delay = delay
8
9       def run(self):
10          while self.counter:
11              print("{} - Iterating: {}".format(self.getName(), self.counter))
12              time.sleep(self.delay)
13              self.counter -= 1
```

Criando objetos thread

- Usa-se o módulo **threading** o qual contem o objeto Thread

- Pode-se estender o objeto **Thread** e implementar o método **run**

```
import threading

var = 1

class MinhaThread ( threading.Thread ):
    def run ( self ):
        global var
        print 'Thread %d' % var
        var += 1

for x in xrange ( 20 ):
    MinhaThread().start()
```

Criando objetos thread

- Para definir a execução de um método em thread pode-se usar o conceito de uso pythônico.

- Usa-se diretamente o construtor indicando um target.

- Basta usar o decorator.

```
def minha_func(...):
    codigo
thr = Thread(target=minha_func, args=args, kwargs=kwargs)
thr.start()
```

Mas o que é decorator ??

- **Decorator** é um artifício desenvolvido para surprir algumas dificuldades com relação ao controle de chamada de funções.

- Podem receber parâmetros.

```
def deco(func):
    ...
    return ...

def func(...):
    return ...

soma = test_decorator(soma)
```

```
def deco(func):
    ...
    return ...

@deco
def func(...):
    return ...
```

Decorator - exemplo

```
def test_decorator(func):
    def nova_soma(*args):
        return reduce(func, args)
    return nova_soma

@test_decorator
def soma(a, b): return a + b

print soma(10, 20, 40)
```

http://wiki.python.org/moin/PythonDecoratorLibrary

Decorator

- Criação de métodos estáticos
- staticmethod

```
>>> class ObjTest:
...     @staticmethod
...     def s_test():
...             print "funciona!!!"
...
>>> ObjTest.s_test()
"funciona!!!"
>>>
```

Thread Pythônica

- Método que roda em background
- Basta usar o Decorator

```
def threaded(func):
    def proxy(*args,**kwargs):
        thr = Thread(target=func, args=args, kwargs=kwargs)
        thr.start()
        return thr
    return proxy

@threaded
def minha_funcao():
    codigo
```

Controlando Threads

- Eventos são utilizados para controlar e sincronizar Threads.

- Existem outras estruturas que facilitam o uso de threads como Lock, RLock e Semáforos (Ver documentação !)

```
class NovaThread(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.__stop_thread_event = Event()
    def stop(self):
        self.__stop_thread_event.set()
    def run(self):
        i = 0
        while not self.__stop_thread_event.isSet():
            print "%d executing" % i
            i+=1
            time.sleep(0.1)
```

Finalizando objetos thread

- Não existe nenhum comando que finalize diretamente uma thread.
- Basta terminar o método run do objeto Thread
- No caso de execução de alguma função em thread, a execução é finalizada quando a função termina.
- Para finalizar eficientemente e elegantemente uma thread, usamos objetos Event que respondem rapidamente a thread o que está ocorrendo

Finalizando objetos thread

```
class NovaThread(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.__stop_thread_event = Event()
def stop(self):
    self.__stop_thread_event.set()
def run(self):
    i = 0
    while not self.__stop_thread_event.isSet():
        print "%d executing" % i
        i+=1
        time.sleep(0.1)
```

Threads - Filas (Queue)

- **FIFO** - First In First Out!
- Queue é uma classe de listas sincronizadas para comunicar threads.
- Quando cheia, lança a exceção Full e quando vazia lança a exceção Empty
- Métodos importantes:
- **Put** - injeta dados na fila
- **Get** - retira dados da fila

Threads - Filas (Queue)

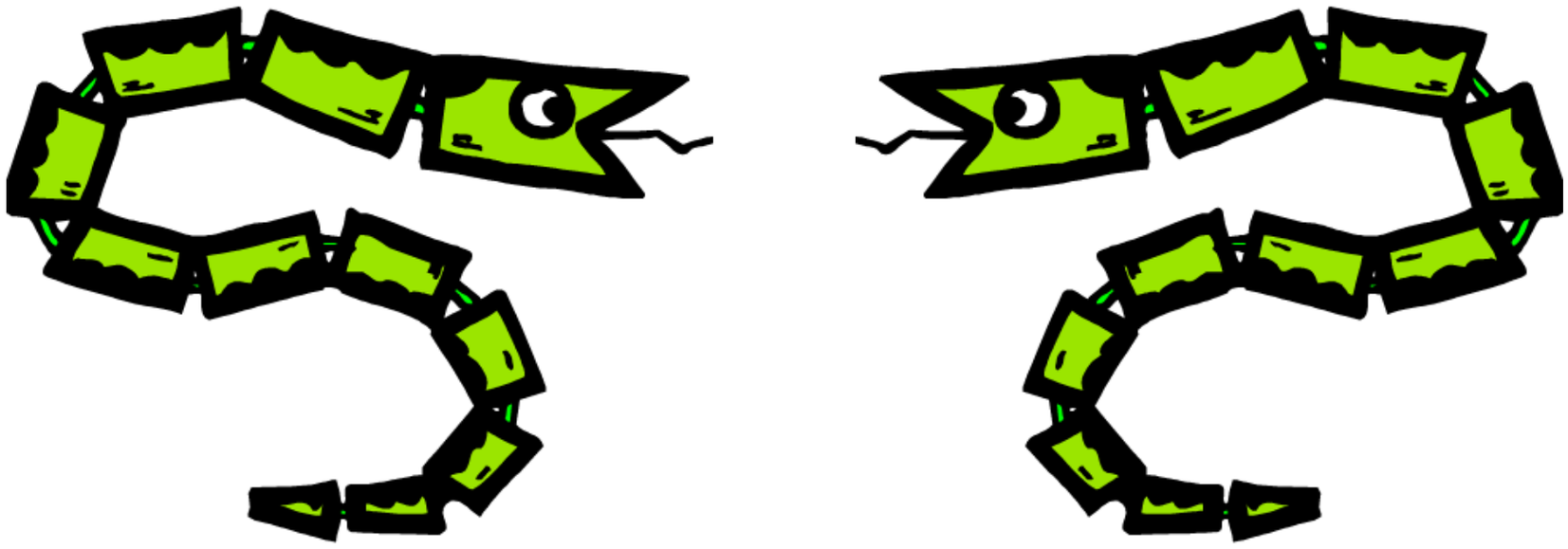
```
@threaded
def func():
    while 1:
        item = q.get()
        operacao(item)
        q.task_done()

q = Queue()
for i in range(5):
    func()

for item in source():
    q.put(item)
q.join() # Bloqueia a fifo até que tudo seja processado
# Não é possível escrever na fifo neste momento!
```

Socket

Comunicando Programas...



Socket

- Conexão virtual entre processos
- Um dos mais populares meios de comunicação de processos em rede
- Segue a idéia da arquitetura cliente-servidor

- Módulo **socket**

- Permite o uso de vários protocolos, os mais usados são:
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram Protocol)

Socket Básico

- **Cliente**

- Se conecta a um servidor que aguarda conexões

- **Servidor**

- Libera conexões para serem usadas (bind)

- Aceita ou rejeita conexões

```
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM
```

```
udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

Socket Servidor

- Cria o socket
- **TCP**
  - Precisa explicitamente aceitar conexões, o cliente tem que se conectar com o servidor
- **UDP**
  - O Cliente não precisa se conectar ao servidor
- Dá um bind no socket

```
udp_server_sock.bind((host, port))
```

```
tcp_server_socket.bind((host, port))
```

- Espera conexões

```
tcp_server_socket.listen(<número de possíveis conexões>)
```

Socket Servidor

- Aceita conexões

```
(clientsocket, address) = tcp_server_socket.accept()
```

- Servidor UDP

```
import socket
HOST, PORT = 172.0.0.1, 20000

udp_server_sock = socket(AF_INET,SOCK_DGRAM)
udp_server_sock.bind(addr)

while 1:
    data,addr = udp_server_sock.recvfrom(buf)
    if not data: break
    print "\nReceived message '", data,"'"
udp_server_sock.close()
```

Socket Servidor

- Servidor TCP

```
import socket
HOST, PORT = 172.0.0.1, 20000

tcp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_server_socket.bind((HOST, PORT))
tcp_server_socket.listen(1)
conn, addr = tcp_server_socket.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    print "\nReceived message '", data,"'"
conn.close()
```

Socket Cliente

- Cria o socket
- No TCP tem que se conectar ao servidor

```
tcp_server_socket.connect((host, port))
```

- Cliente UDP

```
import socket
HOST, PORT = 172.0.0.1, 20000
udp_client_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
# Send messages
while (1):
    data = "TEST"
    if not data: break
    else:
        if(udp_client_socket.sendto(data,(HOST,PORT))):
            print "Sending message"
udp_client_socket.close()
```

Socket Cliente

- Cliente TCP

```
import socket
HOST, PORT = 172.0.0.1, 20000

tcp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_client_socket.connect((HOST, PORT))
tcp_client_socket.send('Hello, world')
data = tcp_client_socket.recv(1024)
tcp_client_socket.close()
print 'Received', repr(data)
```

Socket

- Podemos deparar com o problema de alocação de porta durante o desenvolvimento de módulo que usem certas portas

- **NEM TODAS PORTAS ESTÃO LIVRES PARA USO!!**
  - Pode-se usar livremente portas acima de 1024

- Para podermos usar a mesma porta durante o tempo todo sem problemas, usar a seguinte configuração:

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Logging

- Python facilita o uso de logs.
- Existe o módulo chamado **logging** pronto para o uso!

```
import logging

logging.basicConfig(leavel=logging.DEBUG,
                    format='%(asctime)s "\
                    '%(levelname)s %(message)s',
                    filename='/tmp/myapp.log',
                    filemode='w')

logging.debug('Isto eh uma msg de debug')
logging.info('Isto eh uma informacao')
logging.warning('Isto eh um aviso!')
```