Python Design Patterns Cheat Sheet

by sercand via cheatography.com/32560/cs/10035/

Singleton

```
def Singleton(cls):
  _instances = {}
  def getinstance():
    if cls not in _instances:
      _instances[cls] = cls()
    return _instances[cls]
  return getinstance
@Singleton
class Counter(object):
  def __init(self):
    if not hasattr(self ,’val’):
self.val = 0
  def get(self): return self.val
  def incr(self): self.val += 1
```

Factory

```
@Singleton
class PlayerFactory(object):
  def new(self, name, type):
    if type == ’peasant’:
      return Peasant(name)
    elif type == ’warior’:
      return Warior(name)
    return None
```

Proxy

```
class CounterProxy(object):
    def __init__(self, type):
        self.type = type
    def incr(self):
        if self.type == 'W':
            Counter().incr()
    def get(self):
        Counter().get()
```

Producer

```
class Producer(Thread):
  def __init__(self, queue, cvs,
maxsize=5):
    Thread.__init__(self)
    self.queue = queue
    self.maxsize = maxsize
```

Producer (cont)

```
    self.notfull = cvs[0]
    self.notempty = cvs[1]
    self.terminate = False
    self.counter = 0
  def run(self):
    while not self.terminate:
      sleep(0.1 * randint(0, 10))
# sleep randomly
      self.notfull.acquire()
      while len(self.queue) >=
self.maxsize: # full
        print "full queue,
waiting"
        self.notfull.wait()
        if self.terminate:
          break
      self.notfull.release()
      self.counter += 1
      self.queue.append(self.counte
r)
      self.notempty.acquire()
      self.notempty.notify() #
notify consumer, a new item
      self.notempty.release()
  def quit(self):
    self.terminate = True
    self.notfull.acquire()
    self.notfull.notify()
    self.notfull.release()
```

TCP Server/Client

```
#Client
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
s.connect(('127.0.0.1', 50007))
s.send('Hello')
data = s.recv(1024)
print 'Received', repr(data)
s.close()
#Server
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
s.bind(('', 50007))
```

TCP Server/Client (cont)

```
s.listen(1)
while True:
    conn, addr = s.accept()
    print 'Connected by', addr
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.send(data)
    conn.close()
```

Observer

```
class Subject(object):
    _observers = []
    def register(self, obs):
        self._observers.append(obs)
    def unregister(self, obs):
        self._observers.remove(obs)
    def notify(self):
        for o in self._observers:
            o.update(self)
    def state(self): pass
class Observer(object):
    def update(self, subj): pass
class Clock(Subject):
    def __init__(self):
        self.value = 0
    def state(self):
        return self.value
    def tick(self):
        self.value += 1
        self.notify()
class Person(Observer):
    def update(self, obj):
        print "heyo", obj.state()
a = Person()
b = Clock()
b.register(a)
b.tick()
```

facade

```
class AuthFacade(object):
    def __init__(self ,method):
        if method == 'passwd':
            #
        elif method =='oauth':
            #
        elif method =='otp':
            #
        else:
                throw ...
    def auth(self,identity,data):
        #check authentication based
on setting
```

Consumer

```
class Consumer(Thread):
  def __init__(self, queue, cvs):
    Thread.__init__(self)
    self.queue = queue
    self.notfull = cvs[0]
    self.notempty = cvs[1]
    self.terminate = False
  def run(self):
    while not self.terminate:
      sleep(0.1 * randint(0, 10)) #
sleep randomly
      self.notempty.acquire()
      while len(self.queue) == 0: #
empty
        print "empty queue,
waiting"
        self.notempty.wait()
        if self.terminate:
          break
      self.notempty.release()
      item = self.queue[0]
      del self.queue[0]
      print "consumed ", item
      self.notfull.acquire()
      self.notfull.notify() #
notify consumer, a new item
      self.notfull.release()
  def quit(self):
```

Consumer (cont)

```
    self.terminate = True
    self.notempty.acquire()
    self.notempty.notify()
    self.notempty.release()
```

patterns

A complex library, tool or system; consisting of many functions and/or classes; probably poorly designed is tried to be accessed. It is hard to read and understand. There are many dependencies distributed in the source, needs many housekeeping tasks and stages to access. Any change in the system require changes in the whole source code.	Facade:Define a class implementing all details of the library/system and providing a simple uniform interface. Access the library through this interface.
You need to access a hard to duplicate, limited, probably old class definition. You donot have a chance to improve it or change the interface. Or, you want to have restricted access to methods (authorization). Or, you want smarter access like caching.	Proxy: Write a class interface implementing functionalities missing in the original interface.

patterns (cont)

Objects depending on eachothers states need to be informed when the other encountered a change. Event handling systems.	Observer: Maintain a registry of observing objects in Subject object. When an event occurs, notify the observers.
Create objects from a set of classes. Implementation depends on class definitions. Introduction of new classes and other changes need recompilation. Class constructors exposed.	Factory: Use interface functions/objects to encapsulate class names and constructors. Use a interface functions, methods to provide you instances. Rest is handled by polymorphism.

UDP Server-Client (not design pattern)

```
#Client
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 20001))
s.sendto("hello", ('localhost',
20000))
(data, addr) = s.recvfrom(1024)
print 'server responded: ', data
s.close()
#Server
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 20000))
while True:
    (data, addr) =
s.recvfrom(1024)
    s.sendto("output", addr)
```