Python

Обо всем понемногу

Python

Singleton. Almostly

Singleton. Again

- Основная цель – работать с одним и тем же объектом. Но что под этим подразумевается?

- В стандартном понимании синглтона подразумевается работа с одним и тем же **экземпляром** объекта.
- А если экземпляры разные, но **поведение и состояние** у них у всех одинаковое?

Borg

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state

>>> b1 = Borg()
>>> b2 = Borg()
>>> b1.foo="123"
>>> b2.foo
'123‘
>>> b1 is b2
False
>>> b1.foo is b2.foo
True
```

Borg

```
class Config(object):
    _we_are_one = {}

    def __init__(self):
        self.__dict__ = self._we_are_one

    def set_myvalue(self, val):
        self._myvalue = val

    def get_myvalue(self):
        return getattr(self, '_myvalue', None)

    myvalue = property(get_myvalue, set_myvalue)

c = Config()
print(c.myvalue) # prints None
c.myvalue = 5
print(c.myvalue) # prints 5
c2 = Config()
print(c2.myvalue) #prints 5
```

Borg. Contras

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state

class Child(Borg):
    pass

>>> b1 = Borg()
>>> b1.foo=2
>>> c1 = Child()
>>> c1.foo
2
```

Python

PyUnit

PyUnit

- Фреймворк для тестирования
- По сути – написанная на Python версия JUnit

TestCase

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
      def runTest(self):
            widget = Widget("The widget")
            assert widget.size() == (50,50), 'incorrect default size‘
```

Все классы тестов должны наследоваться от unittest.TestCase.

Запуск:

>>> unittest.main()

Или

>>> testCase = DefaultWidgetSizeTestCase()

>>> testCase.run()

Инициализация в тестах

```
import unittest

        class SimpleWidgetTestCase(unittest.TestCase):
            def setUp(self):
                self.widget = Widget("The widget")

        class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
            def runTest(self):
                assert self.widget.size() == (50,50), 'incorrect default size'

        class WidgetResizeTestCase(SimpleWidgetTestCase):
            def runTest(self):
                self.widget.resize(100,150)
                assert self.widget.size() == (100,150), 'wrong size after resize'
```

Завершение теста

```
import unittest

        class SimpleWidgetTestCase(unittest.TestCase):
            def setUp(self):
                self.widget = Widget("The widget")
            def tearDown(self):
                self.widget.dispose()
                self.widget = None
```

Тест с несколькими методами тестирования

```
import unittest

        class WidgetTestCase(unittest.TestCase):
            def setUp(self):
                self.widget = Widget("The widget")
            def tearDown(self):
                self.widget.dispose()
                self.widget = None
            def testDefaultSize(self):
                assert self.widget.size() == (50,50), 'incorrect default size'
            def testResize(self):
                self.widget.resize(100,150)
                assert self.widget.size() == (100,150), \
                       'wrong size after resize‘
>>> defaultSizeTestCase = WidgetTestCase("testDefaultSize")
>>> resizeTestCase = WidgetTestCase("testResize")
```

TestSuite

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase("testDefaultSize"))
widgetTestSuite.addTest(WidgetTestCase("testResize"))
```

Или

```
suite = unittest.makeSuite(WidgetTestCase,'test')
```

Запуск:

```
>>> runner = unittest.TextTestRunner()
>>> runner.run(widgetTestSuite)
```

Python

Threads… GIL… OMG… WT#

«Сложные» вычисления

```
def count(n):
  while n > 0:
    n -= 1

def test1():
      start = time.time()
      count(10000000)
      count(10000000)
      print(time.time()-start)

>>> test1()
6.762113094329834
```

А что если?..

- Попробуем распараллелить, т.е. создадим два потока, каждый из которых будет считать свой «кусок»

- Библиотека: threading
- Создание задания для потока:

```
>>> t1 = threading.Thread(target=count,args=(10000000,))
```

- target – функция для выполнения
- args – ее аргументы

Работа с потоками

```
t1 = threading.Thread(target=count,args=(10000000,))
t1.start()
t1.join()
```

start – запуск потока.

join – заставляет текущий поток «приостановить» свое выполнение до момента пока не закончится поток t1.

Протестируем

```
>>> import threading
>>> def test2():
      start = time.time()
      t1 = threading.Thread(target=count,args=(10000000,))
      t1.start()
      t2 = threading.Thread(target=count,args=(10000000,))
      t2.start()
      t1.join(); t2.join()
      print(time.time()-start)
>>> test2()
7.239532947540283
```

Итого:

- Один поток: `6.762113094329834 сек`
- Два потока: `7.239532947540283 сек`

- Это плохо и странно...
- Почему так происходит? Из-за GIL

GIL

- Global Interpreter Lock — GIL
- Нельзя использовать несколько процессоров
- В один момент исполняется только ОДИН поток (тот который обладает блокировкой на GIL)
- Но все потоки имеют доступ к общей памяти и могут ее изменять.

- В итоге два потока все равно работают последовательно, плюс тратится время на переключение.

Тогда зачем?

- Зачем же тогда использовать потоки в Python?

```
def ping(addr):
        ret = subprocess.call("ping -n 1 %s | FIND /I \"TTL\"" % addr,
              shell=True)
        if ret == 0:
                print("%s: is alive" % addr)
        else:
                print("%s: did not respond" % addr)
```

```
>>> def test1():
        start = time.time()
        for i in range(12):
                ping("192.168.211."+str(i))
        print(time.time()-start)

>>> test1()
192.168.211.0: did not respond
192.168.211.1: is alive
192.168.211.2: is alive
192.168.211.3: did not respond
192.168.211.4: did not respond
192.168.211.5: is alive
192.168.211.6: did not respond
192.168.211.7: did not respond
192.168.211.8: did not respond
192.168.211.9: did not respond
192.168.211.10: did not respond
192.168.211.11: did not respond
14.905341863632202
```

```
>>> def test2():
        start = time.time()
        t = []
        for i in range(12):
                t.append(threading.Thread(target=ping, args=("192.168.211."+str(i), )))
                t[i].start()
        for i in range(12):
                t[i].join()
        print(time.time()-start)

>>> test2()
192.168.211.5: is alive192.168.211.1: is alive
192.168.211.2: is alive192.168.211.4: is alive
192.168.211.6: is alive
192.168.211.7: did not respond192.168.211.3: did not respond192.168.211.11: did not
respond192.168.211.10: did not respond192.168.211.9: did not respond192.168.211.8: did
not respond192.168.211.0: did not respond
3.37402606010437
```

- Дело в том, что Python не переключается на потоки, которые заблокированы на ввод-вывод!
- Соответственно полезно использовать потоки в ситуация похожих на эту.
- Если вам необходимо использовать несколько процессоров, то используйте библиотеку multiprocessing

Python

Про хранилища данных

Pickle или cPickle

```
import pickle

>>>with open('filename', 'wb') as f:
    var = {1 : 'a' , 2 : 'b'}
    pickle.dump(var, f)

>>> with open('filename', 'rb') as f:
     entry = pickle.load(f)

>>> entry
{1: 'a', 2: 'b'}
```

JSON

```
import json
>>> with open('filename', 'w') as f:
    var = {1 : 'a' , 2 : 'b'}
    json.dump(var, f)

>>> with open('filename', 'r') as f:
     entry = json.load(f)
>>> entry
{'2': 'b', '1': 'a'}
```

DB API 2.0

- Существует большое количество библиотек для работы с различными базами данных. Почти все из них поддерживают стандарт DB API 2.0, который унифицирует способ общения с базой данных.

- Мы для примера рассмотрим работу с базой данных SQLite

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Create table
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')

# Insert a row of data
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")

# Save (commit) the changes
conn.commit()

# We can also close the cursor if we are done with it
c.close()
```

Правильная работа с передачей переменных в запросы

```
# Never do this -- insecure!
symbol = 'IBM'
c.execute("select * from stocks where symbol = '%s'" % symbol)

# Do this instead
t = ('IBM',)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ]:
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

Получение данных

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...     print(row)
...
('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSOFT', 1000, 72.0)
```