University of Zurich[UZH]

Grid Computing Competence Center

Object-oriented Python, II

GC3: Grid Computing Competence Center,
University of Zurich

Sep. 28, 2012

This is called a “test case”.

Do you see anything unusual in this class definition?

```
import unittest as ut
from minmax import MinMax

class TestMinMax(ut.TestCase):

    def test_init(self):
        m = MinMax()
        self.assertEqual(m.min, None)
        self.assertEqual(m.max, None)

    def test_send42(self):
        m = MinMax()
        m.send(0)
        m.send(42)
        self.assertEqual(m.min, 0)
        self.assertEqual(m.max, 42)
```

Source code available at: http://www.gc3.uzh.ch/teaching/gc3pie2012/python/test_minmax.py

This is *not* "`(object)`"!

```
import unittest as ut
from minmax import MinMax

class TestMinMax (ut.TestCase) :

    def test_init(self):
        m = MinMax()
        self.assertEqual(m.min, None)
        self.assertEqual(m.max, None)

    def test_send42(self):
        m = MinMax()
        m.send(0)
        m.send(42)
        self.assertEqual(m.min, 0)
        self.assertEqual(m.max, 42)

if __name__ == "__main__":
  ut.main()
```

Where have these functions been defined?

```
import unittest as ut
from minmax import MinMax

class TestMinMax(ut.TestCase):

    def test_init(self):
        m = MinMax()
        self.assertEqual(m.min, None)
        self.assertEqual(m.max, None)

    def test_send42(self):
        m = MinMax()
        m.send(0)
        m.send(42)
        self.assertEqual(m.min, 0)
        self.assertEqual(m.max, 42)

if __name__ == "__main__":
    ut.main()
```

Still the program runs fine!

```
$ python test_minmax.py --verbose
test_init (__main__.TestMinMax) ... ok
test_send0 (__main__.TestMinMax) ... ok
test_send42 (__main__.TestMinMax) ... ok

----------------------------------------
Ran 3 tests in 0.000s

OK
```

This class is defined as a *descendant* of the `unittest.TestCase` class.

```
class TestMinMax (ut.TestCase) :

    def test_init(self):
        m = MinMax()
        self.assertEqual(m.min, None)
        self.assertEqual(m.max, None)
```

This means that it *inherits* all the attributes defined in the *ancestor* class.

```
class TestMinMax(ut.TestCase):

    def test_init(self):
        m = MinMax()
        self.assertEqual(m.min, None)
        self.assertEqual(m.max, None)
```

In particular, the `assertEqual` method is defined in the parent class `unittest.TestCase`.

What happens if a descendant class redefines a method already defined in an ancestor class?

What happens if a descendant class redefines a method already defined in an ancestor class?

The method in the descendant class overrides the method in the ancestor class.

What happens if a descendant class defines a `__init__` method?

What happens if a descendant class defines a `__init__` method?

The `__init__` in the descendant class overrides the method in the ancestor class. So `__init__` of the parent class(es) will not be called.

Constructor chaining

When a class is instanciated, Python only calls the first constructor it can find in the class inheritance call-chain.

If you need to call a superclass' constructor, you need to do it *explicitly*:

```
class Application(Task):
  def __init__(self, ...):
    # do Application-specific stuff here
    Task.__init__(self, ...)
    # some more Application-specific stuff
```

Calling a superclass constructor is optional, and it can happen anywhere in the `__init__` method body.

Multiple-inheritance

Python allows multiple inheritance.

Just list all the parent classes:

```
class C(A,B):
  # class definition
```

With multiple inheritance, it is your responsibility to call all the needed superclass constructors.

Python uses the C3 algorithm to determine the call precedence in an inheritance chain.

You can always query a class for its "method resolution order", via the `__mro__` attribute:

```
>>> C.__mro__
(<class 'ex.C'>, <class 'ex.A'>, <class 'ex.B'>, <type 'object'>)
```

Detour: Regular Expression objects

The `re` module in the standard library provides *regular expression searching*, allowing you to match a string against a pattern.

re.search(pattern, string)
If `pattern` is matched anywhere in `string`, return a *match object*. Otherwise, return `None`.

***match*.group(0)**
The entire string matched by `pattern` in a search operation.

Reference: http://docs.python.org/library/re.html

Exercise A: Define a `Grep` class:

- a `Grep` instance is constructed by giving a file name and a regular expression pattern, e.g.,
  `g = Grep(filename, pattern)`
- Each call to the `next()` method returns the next line in the file that matches the regular expression `pattern`.

Exercise B: Define a `GrepOnlyMatching` class, similar to `Grep` except that its `next()` method returns only the part of the line that matched the `pattern` expression.

Exercise C: Define a `GrepExactly` class, similar to `Grep` except that `pattern` is now a fixed string, and the `next()` method returns lines that *contain* it.

The "Template method" pattern

```
class Grep(object):

    def __init__(self, filename, pattern):
        self._file = open(filename, 'r')
        self._pattern = pattern

    def match(self, line):
        return re.search(self._pattern, line)

    def result(self, match, line):
        return line

    def next(self):
        line = self._file.next()
        match = self.match(line)
        while not match:
            line = self._file.next()
            match = self.match(line)
        return self.result(match, line)
```

The "Template method" pattern, I

```
class Grep(object):

  # parts omitted

  def next(self):
    line = self._file.next()
    match = self.match(line)
    while not match:
        line = self._file.next()
        match = self.match(line)
    return self.result(match, line)
```

These calls delegate the actual matching and extraction of the result from the line to instance methods.

The "Template method" pattern, II

So we need only re-define those methods in derived classes to implement a variant behavior.

```
class GrepOnlyMatching(Grep):
   def result (self, match, line):
     return match.group(0)

class GrepExactly(Grep):
   def match (self, line):
     return (self._pattern in line)
```