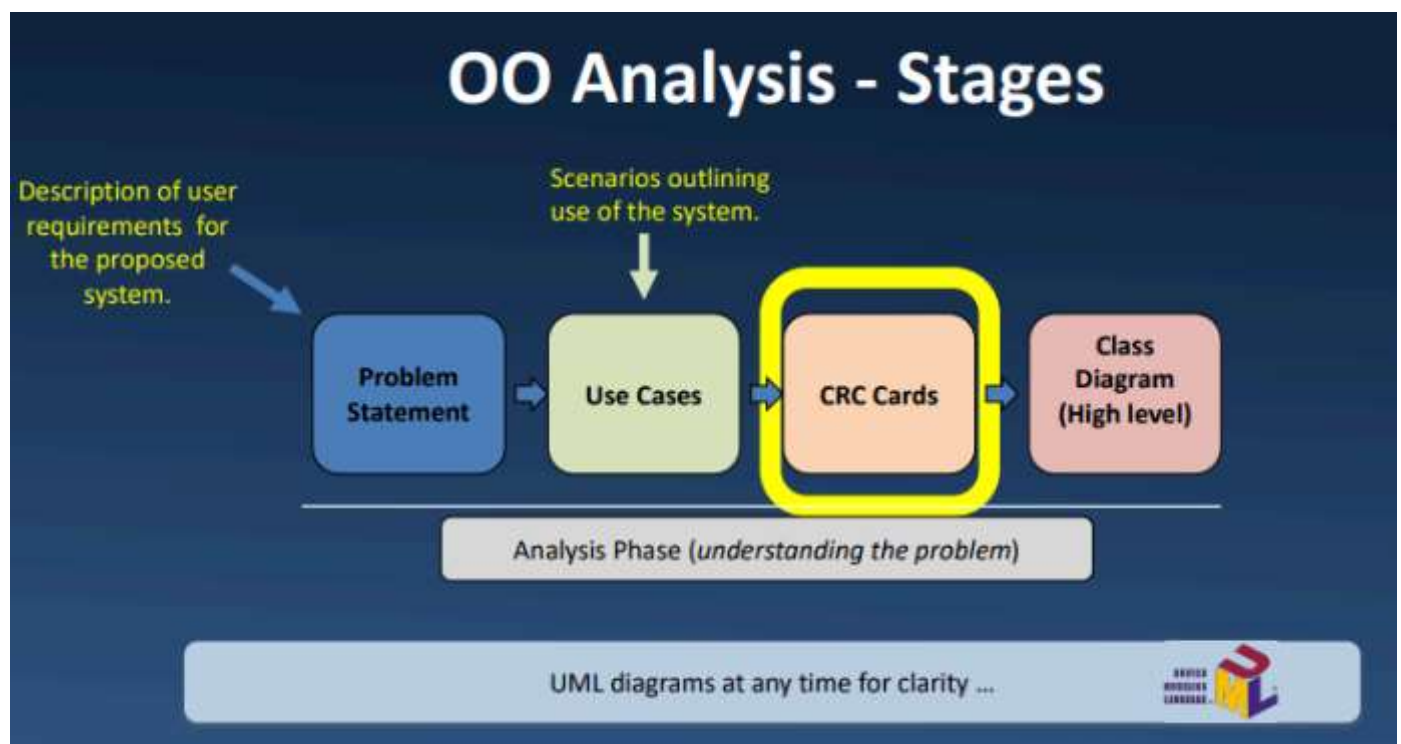
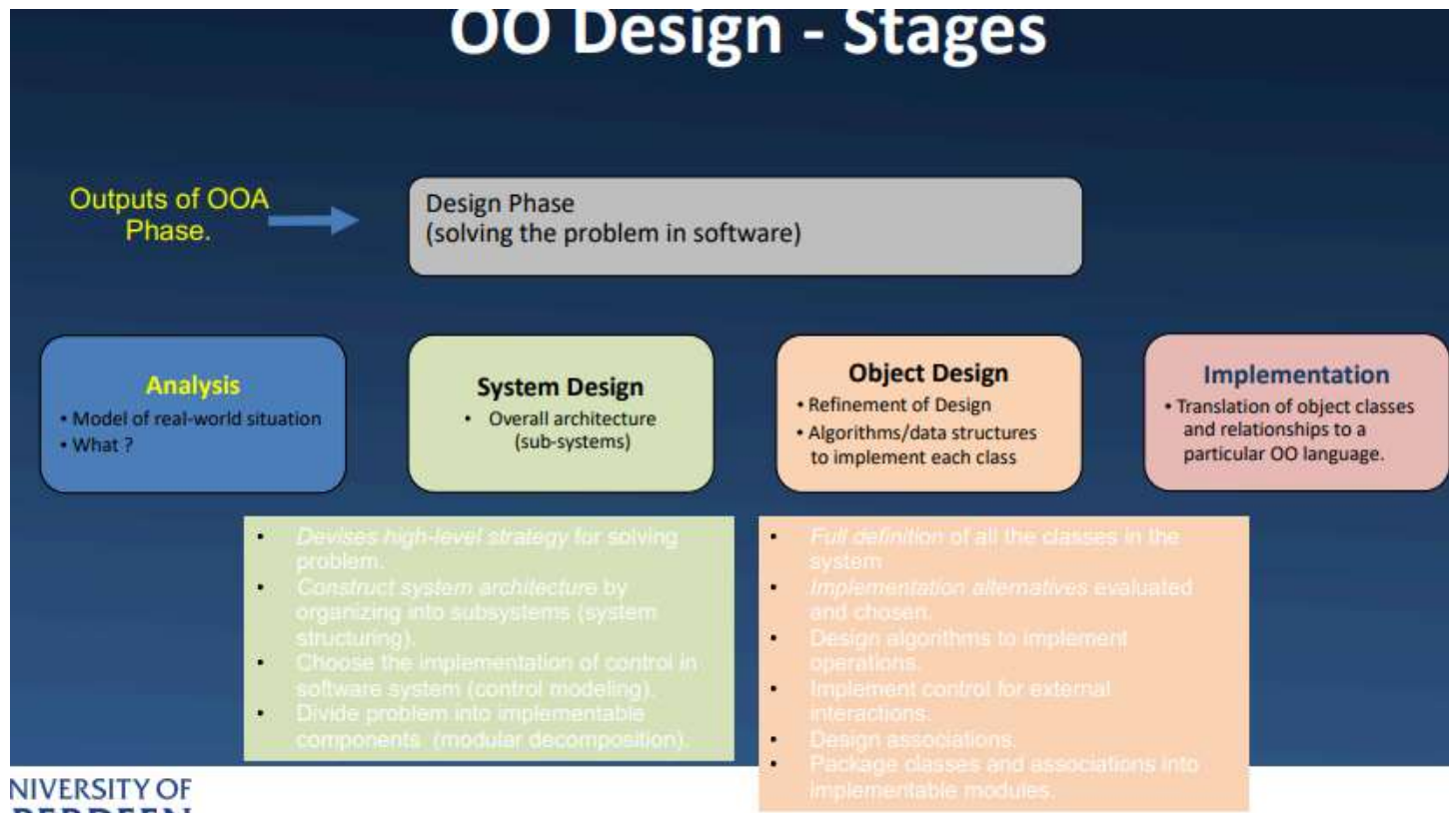


# OOA分析过程:



CRC卡:

# UML语言：

– 统一符号：Booch + OMT + 状态图

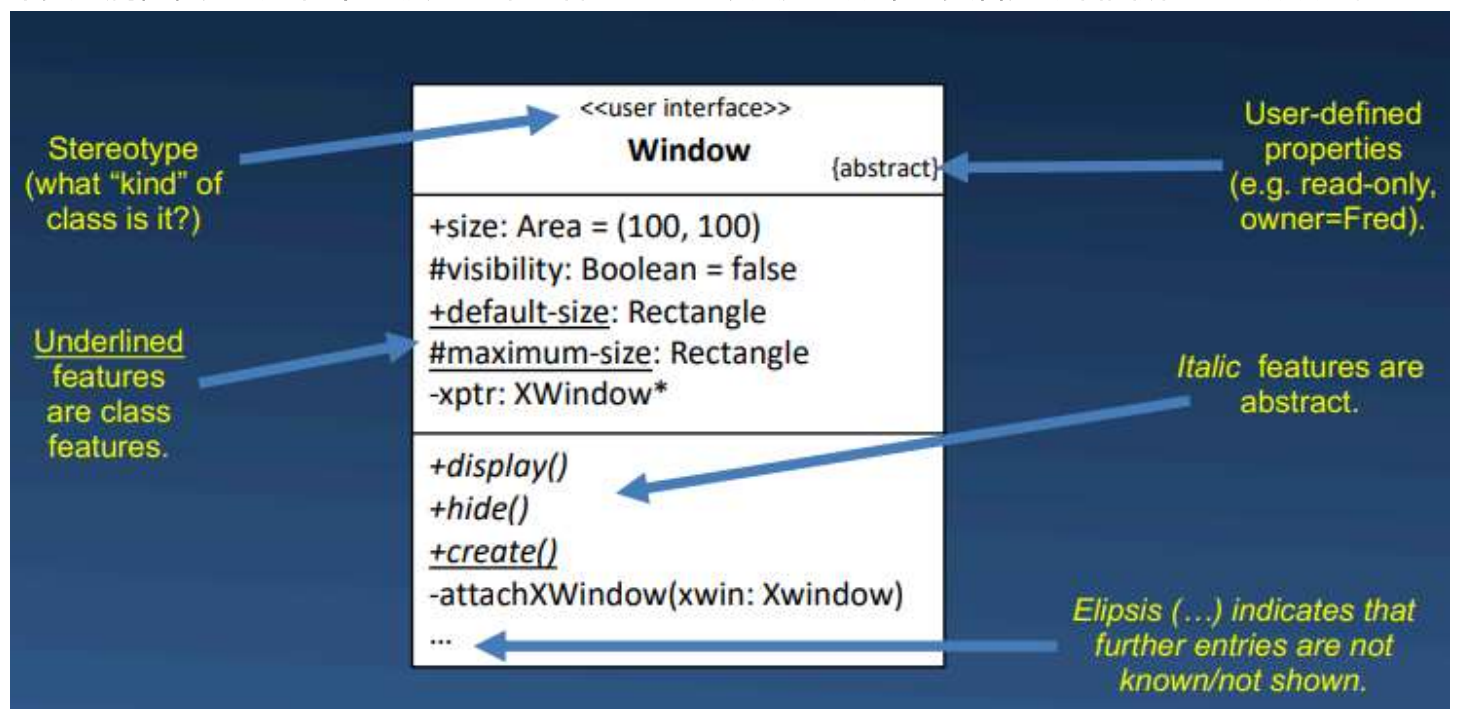
• 描述软件设计的图形符号。

• UML 不是一种方法或过程。 – 统一开发过程

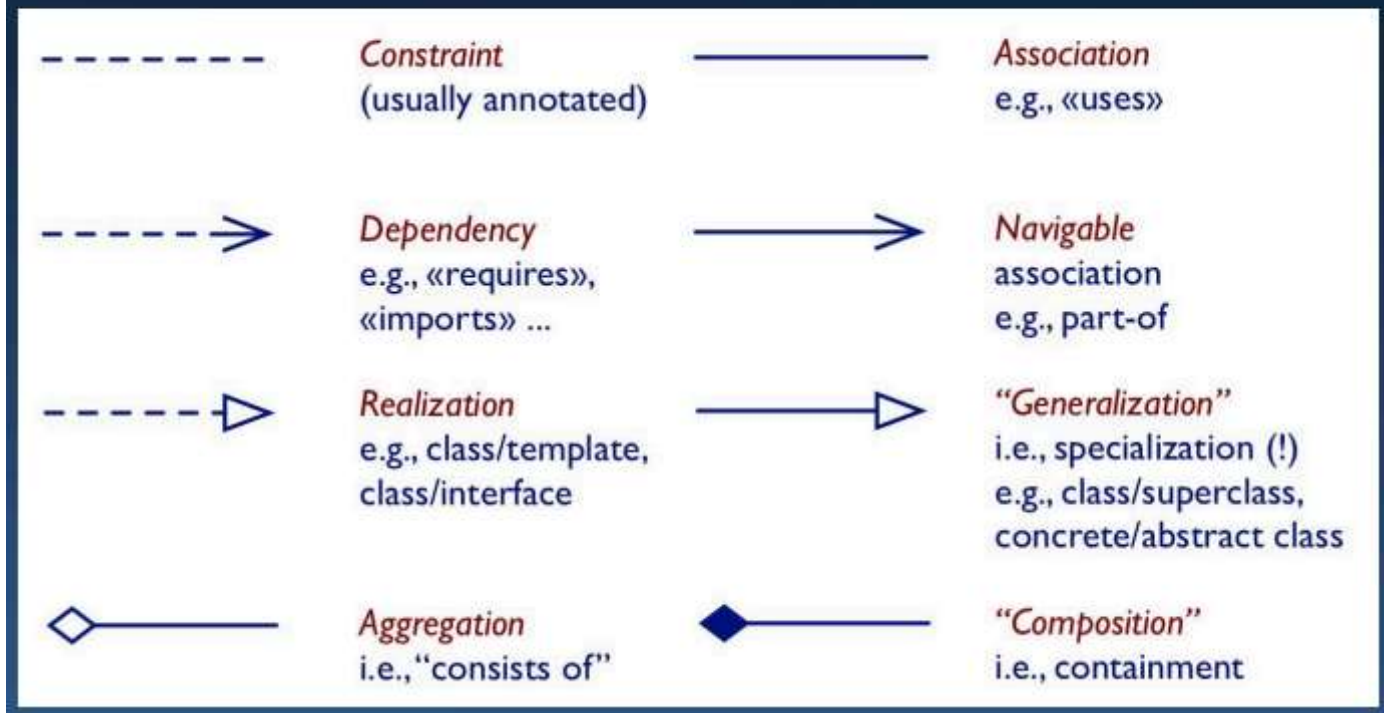
UML has rules on how to draw models of classes, associations between classes, messages sent between objects

class:

类图 – 表示要实现的类定义;– 列出每个类的名称、属性和方法;– 显示类之间的关系。• UML 允许对一个类的属性和方法进行不同级别的详细描述： – 可以只是矩形中的类名;– 或右图所示的一般形式



# UML - Lines & Arrows



Object Diagrams:

Sequence Diagram;序列图

## 编码习惯:

正确性;

不好的习惯:

- 1, 类的外面访问被保护的成员Accessing a protected member from outside the class ,
- 2, 给内置的函数赋值: Assigning to built-in function
- 3, for循环内的else 不加break
- 4, 和Java的属性风格混合
- 5, 在不属于某个class特属的方法请用静态方法

请使用显示解包using explicit unpacking

Bad Practice: Verbose and error-prone

```
elems = [4, 7, 18]

elem0 = elems[0]
elem1 = elems[1]
elem2 = elems[2]
```

Good Practice: Use unpacking

```
elems = [4, 7, 18]

elem0, elem1, elem2 = elems
```

请使用get() to return a default value from a dict:

Bad: it is verbose and inefficient because it queries the dictionary twice.

```
dictionary = {"message": "Hello, World!"}
data = ""

if "message" in dictionary:
    data = dictionary["message"]

print(data) # Hello, World!
```

When get() is called, Python checks if the specified key exists in the dict. If it does, then get() returns the value of that key. If the key does not exist, then get() returns the value specified in the second argument to get().

```
dictionary = {"message": "Hello, World!"}
data = dictionary.get("message", "")

print(data) # Hello, World!
```

- 不好的地方是它冗长且效率低下，因为它在字典中查询两次。
- 当调用get()时，Python会检查指定的键是否存在于字典中。如果存在，则get()返回该键的值。如果键不存在，则get()返回在get()的第二个参数中指定的值。

可维护性：



## Maintainability II: Not using “with” to open files

```
f = open("file.txt", "r")
content = f.read()
1 / 0 # ZeroDivisionError
# never executes, possible memory issues or file corruption
f.close()
```

```
with open("file.txt", "r") as f:
    content = f.read()
# Python still executes f.close() even though an exception occurs
1 / 0
```

- 因为它确保在不再需要文件对象时正确关闭文件。with 语句会在进入其代码块时自动获取资源，并在代码块结束时自动释放资源，即使发生异常也会如此。
- 调用者在继续执行之前必须检查返回值的类型。
- 当向函数提供无效数据、函数的前提条件未满足或在函数执行过程中发生错误时，函数不应返回任何数据。相反，函数应该引发一个异常。
- 不要滥用全局变量，尽量放到object当中

## Maintainability IV

Using the  
global  
statement  
— Bad.

```
WIDTH = 0 # global variable
HEIGHT = 0 # global variable

def area(w, h):
    global WIDTH # global statement
    global HEIGHT # global statement
    WIDTH = w
    HEIGHT = h
    return WIDTH * HEIGHT

def perimeter(w, h):
    global WIDTH # global statement
    global HEIGHT # global statement
    WIDTH = w
    HEIGHT = h
    return ((WIDTH * 2) + (HEIGHT * 2))

print("WIDTH:", WIDTH) # "WIDTH: 0"
print("HEIGHT:", HEIGHT) # "HEIGHT: 0"

print("area():", area(3, 4)) # "area(): 12"

print("WIDTH:", WIDTH) # "WIDTH: 3"
print("HEIGHT:", HEIGHT) # "HEIGHT: 4"
```

Encapsulate the global  
variables into  
objects—Good

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def circumference(self):
        return ((self.width * 2) + (self.height * 2))

r = Rectangle(3, 4)
print("area():", r.area())
```

- 不要使用单个字母去命名一个变量

可读性:

- 不要将东西和None去进行比较，用 is

## Readability II: Comparing things to None the wrong way

```
number = None

if number == None:
    print("This works, but is not the preferred PEP 8 pattern")
```

```
number = None

if number is None:
    print("PEP 8 Style Guide prefers this pattern")
```

- 区别：
- == 运算符比较两个对象的值，如果它们的值相等，则返回 True，否则返回 False。它比较的是对象的内容。
- is 运算符比较两个对象的标识，即它们是否引用内存中的同一个对象。如果两个变量引用的是同一个对象，则返回 True，否则返回 False。它比较的是对象的身份。

- 字典推导式：

式，但是用于创建字典而不是列表。字典推导式允许您根据一定的规则从一个可迭代对象中创建一个字典。

字典推导式的一般语法是：

```
{key_expression: value_expression for item in iterable}
```

python



其中，`key_expression` 是用于生成字典键的表达式，`value_expression` 是用于生成字典值的表达式，`item` 是可迭代对象中的每个元素。

下面是一个简单的示例，演示如何使用字典推导式创建一个简单的字典：

```
# 创建一个字典，将列表中的元素作为键，将其平方作为值
my_list = [1, 2, 3, 4, 5]
my_dict = {x: x**2 for x in my_list}
print(my_dict) # 输出: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

python

字典推导式也可以包含条件语句，用于筛选要包含在最终字典中的元素。例如：

```
# 创建一个字典，仅包含列表中值为偶数的元素及其平方
my_list = [1, 2, 3, 4, 5]
my_dict = {x: x**2 for x in my_list if x % 2 == 0}
print(my_dict) # 输出: {2: 4, 4: 16}
```

python

- 当一个函数需要返回不止一个值的时候，请使用元组tuples

## Readability IV: Not using named tuples when returning more than one value from a function

```
def get_name():
    return "Richard", "Xavier", "Jones"

name = get_name()

# no idea what these indexes map to!
print(name[0], name[1], name[2])
```

Easier to read:

```
from collections import namedtuple

def get_name():
    name = namedtuple("name", ["first", "middle", "last"])
    return name("Richard", "Xavier", "Jones")

name = get_name()

# much easier to read
print(name.first, name.middle, name.last)
```

- 使用py推荐的循环风格

```
l = [1,2,3]

# creating index variable
for i in range(0, len(l)):
    # using index to access list
    le = l[i]
    print(i, le)
```

Unpythonic Loop

Pythonic Loop

```
for i, le in enumerate(l):
    print(i, le)
```

"There should be one— and preferably only one —obvious way to do it."— PEP 20

```
l=[1,2,3]
for i,le in enumerate(l):
    print(i,le)
    pass

#####
for i in range(0,len(l)):
    le=l[i]
    print(i,le)
```

使用小写+下划线命名函数，而不是驼峰命名法

不要使用exec



## Security: use of `exec`

Not safe

```
s = "print(\"Hello, World!\")"
exec s
```

Safer:


```
def print_hello_world():
    print("Hello, World!")

print_hello_world()
```

- REASON

`exec` 是一个内置函数，用于执行存储在字符串或文件中的 Python 代码。它的基本语法如下：

```
exec(object[, globals[, locals]])
```

python 

其中，`object` 参数可以是字符串（包含要执行的 Python 代码）或者代码对象。`globals` 和 `locals` 是可选参数，用于指定全局命名空间和局部命名空间。

`exec` 函数会执行传入的代码，并将其作为 Python 语句或表达式执行。这个函数的使用应该谨慎，因为它可以执行任意代码，可能会导致安全问题或者难以维护的代码。通常情况下，应该尽量避免使用 `exec`，除非有特定的需求并且能够确保代码的安全性。

性能：

- Performance性能

- 不要使用列表中的键来检查键是否包含在列表中
- 而是使用集合的方式

## Performance: Using key in list to check if key is contained in list

### Not Sufficient

```
l = [1, 2, 3, 4]

# iterates over three elements in the list
if 3 in l:
    print("The number 3 is in the list.")
else:
    print("The number 3 is NOT in the list.")
```

### Sufficient

```
s = set([1, 2, 3, 4])

if 3 in s:
    print("The number 3 is in the list.")
else:
    print("The number 3 is NOT in the list.")
```

This is much more efficient, as Python can attempt to directly access the target number in the set, rather than iterate through every item in the list and compare every item to the target number.

- 这样做更为高效，因为Python可以尝试直接访问集合中的目标数字，而不是遍历列表中的每个项并将每个项与目标数字进行比较。

## 测试:

## Faults, Failures & Errors

BUG

- Software **Fault**
  - A static defect in the software.
- Software **Failure**
  - External, incorrect behaviour with respect to the requirements or other description of the expected behaviour.
- Software **Error**
  - An incorrect internal state that is the manifestation of some fault.

Fault  
Should check for  $i > 0$ , not  $\geq 0$

```
def countPositive(int_list):
    count = 0
    for i in int_list:
        if i >= 0:
            count += 1
    return count
```

TEST1  
[0, 3, -1, 6]  
Expected: 2  
Actual: 3

Error: count is 1, not 0, after first iteration.  
Failure: count is 3 at the return statement.

重构 - 在不改变其外部行为的情况下修改代码；注释、格式化、其他内部更改。

- Coverage Criteria覆盖准则

- 测试人员搜索庞大的输入空间• 试图找到能发现最多问题的最少输入。
- 覆盖准则提供了有结构、实用的方式来搜索输入空间：– 彻底搜索输入空间。– 测试之间没有太多的重叠。
- 功能覆盖 程序中的每个函数（或方法）都被调用了吗？
- 语句覆盖 程序中的每个语句都被执行了吗？
- 分支覆盖 程序中每个控制结构（比如在 if 语句中）的每个分支都执行了吗？例如，对于一个 if 语句，是真分支和假分支都被执行了吗？
- 条件覆盖 每个布尔子表达式都被评估为真和假吗？

- Testing Activities：

- 设计测试：
  - 基于准则• 设计测试值以满足覆盖准则。
  - 基于人工• 根据对程序领域的知识和测试的人类知识设计测试值。Design test values based on domain knowledge of the program and human knowledge of testing
- 测试自动化• 将测试值嵌入可执行脚本中。Embed test values into executable scripts.
- 测试执行• 在软件上运行测试并记录结果。
- 测试评估• 评估测试结果，向开发人员报告。
- Agile Methods敏捷方法：
  - 重新定义正确性，使其相对于特定的一组测试。
  - 如果软件在测试中表现正确，那么它是“正确的”。
  - 不是定义所有行为，而是演示一些行为。

#### Function Coverage

Has each function (or method) in the program been called?

#### Statement Coverage

Has each statement in the program been executed?

#### Branch Coverage

Has each branch of each control structure (such as in if statements)

been executed? For example, given an if statement, have both the true and false branches been



executed?

Condition Coverage

Has each Boolean sub-expression evaluated both to true and false?

函数覆盖率 程序中的每个函数（或方法）是否被调用？语句覆盖率 程序中的每个语句都已执行吗？分支覆盖率 是否执行了每个控制结构的每个分支（例如在 if 语句中）？例如，给定一个 if 语句，是否同时执行了 true 和 false 分支？条件覆盖率 是否将每个布尔子表达式的计算结果都计算为 true 和 false？

# 单元测试python unittest

- TestCase 类:


方法用于比较值、设置测试以及在测试完成后进行清理。

要为特定任务编写一组单元测试，创建一个 TestCase 的子类，并编写单独的方法来执行实际的测试。

## Python unittest

- Provides a common interface for **unit tests**.
  - Each unit test tests a single unit of the total amount of available code.
- **TestCase** class
  - Methods to compare values, set up tests, and clean up when they have finished.
  - To write a set of unit tests for a specific task, create a subclass of **TestCase**, and write individual methods to do the actual testing.

```
$ python main.py
.
-----
Ran 1 test in 0.000s OK
```



```
import unittest

def foo(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(foo(3), 4)

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

Code to be tested.

为单元测试提供通用接口。 – 每个单元测试测试可用代码总量的单个单元。 • TestCase 类 – 比较值、设置测试并在完成后进行清理的方法。 – 要为特定任务编写一组单元测试，请创建 TestCase 的子类，并编写单独的方法来执行实际测试



## Python单元测试

- 我们可以在一个TestCase类上有尽可能多的测试方法。
  - 每个方法的名称必须以test开头。
  - 支持多种断言内置方法。
- 测试运行器将会将每个方法作为独立的测试执行。
  - 每个测试应完全独立于其他测试。
  - setUp()、tearDown() 方法

这段话是关于Python中的单元测试的说明。它首先提到了在一个`TestCase`类中可以有多测试方法，这意味着我们可以在同一个类中编写尽可能多的测试用例。接着指出了每个测试方法的命名必须以`test`开头，这是为了让测试运行器能够自动识别它们。然后提到了可以使用多种断言内置方法来验证测试的结果。测试运行器会将每个方法都作为一个独立的测试来执行，这样确保了每个测试都能独立地执行和验证。最后提到了`setUp()`和`tearDown()`方法，它们分别用于在测试之前和之后执行一些准备和清理工作，以确保测试的环境和状态是一致的。

# 单元测试python doctest方法

轻量级测试框架。

- 在 Python Docstring 注释中嵌入简单测试（使用 >>>）。

```
new *
def square(x):
    """
    ~~~~~
    >>> square(2)
    4
    >>> square(-2)
    4
    ~~~~~
    return x*x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

# Python doctest

- Lightweight testing framework.
- Embeds simple tests (using `>>>`) in Python Docstring comments.

```
def square(x):  
    """Return the square of x.  
  
    >>> square(2)  
    4  
    >>> square(-2)  
    4  
    """  
    return x * x  
  
if __name__ == '__main__':  
    import doctest  
    doctest.testmod()
```

No doctest output  
– as all tests pass.

```
def square(x):  
    """Return the square of x.  
  
    >>> square(2)  
    4  
    >>> square(-2)  
    4  
    """  
    return x + x  
  
if __name__ == '__main__':  
    import doctest  
    doctest.testmod()
```

Fault

```
$ python main.py  
*****  
File "main.py", line 6, in __main__.square  
Failed example:  
    square(-2)  
Expected:  
    4  
Got:  
    -4  
*****  
1 items had failures:  
    1 of 2 in __main__.square  
***Test Failed*** 1 failures.
```

Coverage  
Issues with this test?

## pytest

更少的代码量

测试用例命名： – 函数以 test 开头 – 类名以 Test 开头 • 类中的方法以 test 开头

## pytest

- Alternative to `unittest`.
- Download:  
– <https://pytest.org/>
- Test cases require less coding effort.
- Test case naming:
  - Functions start with `test`
  - Class names start with `Test`
    - Methods in class start with `test_`

```
# content of test_sample.py  
def func(x):  
    return x + 1  
  
def test_answer():  
    assert func(3) == 5
```

```
$ py.test  
===== test session starts =====  
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0  
rootdir: /Users/pedwards/Desktop, inifile:  
collected 1 item  
  
test_sample.py F [100%]  
  
===== FAILURES =====  
test_answer  
  
def test_answer():  
> assert func(3) == 5  
E assert 4 == 5  
E + where 4 = func(3)  
  
test_sample.py:6: AssertionError  
===== 1 failed in 0.06 seconds =====
```