软件测试方法与技术

软件测试方法：控制流

内容大纲

- 概述
- 图论基础
- 流程图结构以及表示
- PYTHON中的条件和判定
- 语句覆盖
- 判定覆盖
- 条件覆盖
- 修正条件判定覆盖
- 路径覆盖

1. 概述

- 对于同一个问题需求，存在不同的实现方法。不同的实现方法，可能引入的缺陷是完全不一样的。
- 以斐波那契（Fibonacci）数列为例子来描述其实现的不同。公元前13世纪意大利数学家斐波那契的《算盘书》描述著名的兔子问题：假定一对大兔子每月能生一对小兔子（一雄一雌），且每对新生的小兔子经过一个月可以长成一对大兔子，如果不发生死亡，且每次均生下一雌一雄，问由一对刚出生的小兔开始，一年后共有多少对大兔子？
  - 依据该特征，每个月的兔子数量实际上是如下的数列：
  - 0，1，1，2，3，5，8，13，21，34，55，89，144，233，…
- 该数列就是著名的斐波那契数列，从第三项开始每一项都是前两项之和。写成为一般形式为：$f_1=0$，$f_2=1$，…，$f_n=f_{n-1}+f_{n-2}$。

1. 概述

- 斐波那契数列至少存在三种不同的实现方法：（1） 朴素递归，(2) 自底向上动态规划，(3) 数学归纳法。
- 1）朴素递归：自顶向下求解问题，导致了大量的重复求解。以求第5个斐波那契数列为例，其求解过程如图1所示，若将第n位斐波那契数列记为Fib(n)在求解，在求解Fib(5)时，需要同时计算Fib(3)和Fib(4)，而在计算Fib(4)时，又得重新计算Fib(3)。

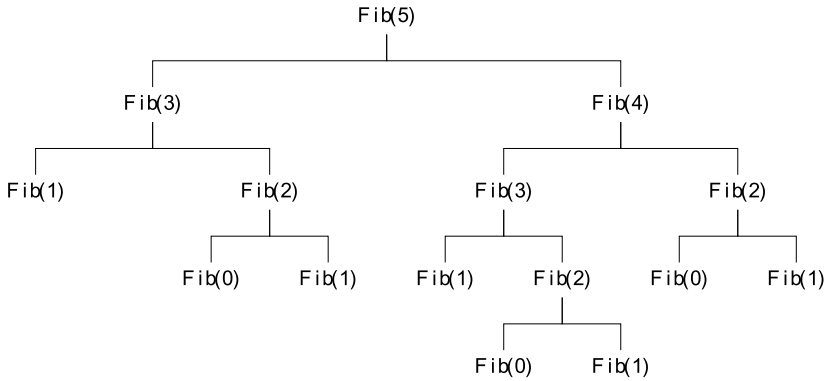


图1 斐波那契数列的朴素递归求解过程

1. 概述

- 详细代码见程序1，其核心内容为直接递归调用。

程序1 斐波那契的朴素递归的实现程序

```
#程序1
#fibonacci1.py
def fibonacci1(i):
    if (i<0):
        raise;
    if (i==0):
        return 0;
    if (i==1):
        return 1;
    else:
        return fibonacci1(i-1)+fibonacci1(i-2);
print  fibonacci1(12);
```

1. 考虑下述“理由”：$Fibonacci$函数，$F(n)$是$O(n)$。

   基本情况（$n \leq 2$）：$F(1) = 1, F(2) = 2$。

   归纳步骤（$n > 2$）：假设当$n' < n$时，结论正确。考虑n，$F(n) = F(n-2) + F(n-1)$。通过归纳，$F(n-2)$是$O(n-2)$且$F(n-1)$是$O(n-1)$。之后，根据加法法则，$F(n)$是$O\big((n-2) + (n-1)\big)$。因此，$F(n)$的运行时间是$O(n)$。

   该“理由”错在哪里？

1. 概述

- 2）递推法：先求出Fib(0)和Fib(1)，然后根据Fib(2)=Fib(0)+Fib(1)求出Fib(2)，以此类推，求出所需要的斐波那契数列。该算法的核心内容是一个循环体，采用正向递推的动态规划，在实现过程中并利用一个链表保存前面已经求出的结果，所以其所有的中间结果只要计算一次，如图2所示。

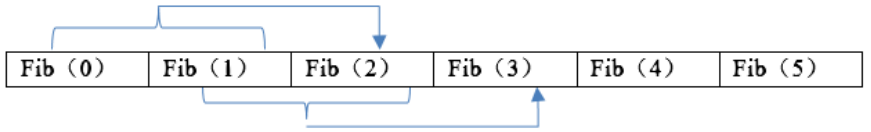


图2 动态规划法

1. 概述

- 动态规划的详细代码如下：

程序2 斐波那契的动态规划的实现程序

```
#程序2
#fibonacci2.py
def fibonacci2(i):
    fibArray=[];
    if (i<0):
        raise;
    if(i==0):
        return 0;
    if (i==1):
        return 1;
    else:
        fibArray.append(1);
        fibArray.append(1);
        for j in range(2,i):                #python下标从0开始
            fibArray.append(fibArray[j-1]+fibArray[j-2]);
        return fibArray[len(fibArray)-1];
print  fibonacci2(12);
```

1. 概述

- 3）利用数学归纳法求解。其基本原理，利用 $f_1=0$，$f_2=1$，…，$f_n=f_{n-1}+f_{n-2}$ 之间数学关系，构建其数学的递推关系。

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}=\begin{pmatrix} F_n+F_{n-1} & F_{n-1}+F_{n-2} \\ F_{n-1}+F_{n-2} & F_{n-1} \end{pmatrix}=\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^1$$

- 由此可以利用矩阵关系来表达递推关系。

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}=\begin{pmatrix} F_n+F_{n-1} & F_n \\ F_{n-1}+F_{n-2} & F_{n-1} \end{pmatrix}$$

$$=\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1$$

$$=\begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2$$

$$=\cdots$$

$$=\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

1. 概述

- 由了上述数学关系，将求斐波那契的数列转换成为矩阵的幂运算。而对于幂运算有可以采用快速幂方式进行优化。如程序3所示。其中power(a,n)用来实现矩阵的幂运算。

程序3 斐波那契的数学归纳法的实现程序

```
#程序3
#fibonacci3.py
import numpy as np;
a0=[[1,1],[1,0]];
def power1(a,n):
      b=[[1,0],[0,1]];                    #单位矩阵
      while (n>0):                        #快速幂算法求矩阵幂
            if( n & 1):b=np.dot(b,a);     #numpy.dot 矩阵乘法
            a=np.dot(a,a);
            n>>=1;                        # 左移
      return b;
def fibonacci3(i):
      if (i<0):
            raise;
      if (i==0):
            return 0;
      if (i==1):
            return 1;
      else:
            temp=power1(a0,i);
            return temp[1][0];
print  fibonacci3(12);
```

1. 概述

- 显然对于上面的同一个求解问题，三个程序的基本原理、实现的技术要求都完全不一样。
  - 朴素递归的核心内容是通过递归所产生的状态树，而递归程序测试的核心要点是递归是否正确，递归的终止条件是否正确。
  - 动态规划方法通过一个循环语句，实现正向的传递。其测试好的核心是循环的终止条件，动态规划的传递方程是否正确。
  - 带有快速幂的数学归纳法的是所有算法中效率最高，其测试的重心将是求快矩阵速幂先关条件的测试，其测试重点将是快速幂的基本特征测试。
- 基于控制流的软件测试充分利用被测单元内部的控制信息，允许测试人员对程序内部逻辑结构及有关信息来设计和选择测试用例，对程序的逻辑路径进行测试。其考虑的是测试用例对程序内部逻辑的覆盖程度，其基本目标是覆盖程序中的每一条路径。但是由于程序中一般含有循环，所以路径的数目极大，以程序4为例简要说明。

1. 概述

程序4 带有三路分支的循环程序

在这段程序中，在循环语句中，存在两个判断，共形成了三个分支，同时循环本身在存在一个循环终止的条件判断，如图3所示。在这个图中，循环变量的递增并没有独立的语句，为了表达的清晰，将其单独画出。

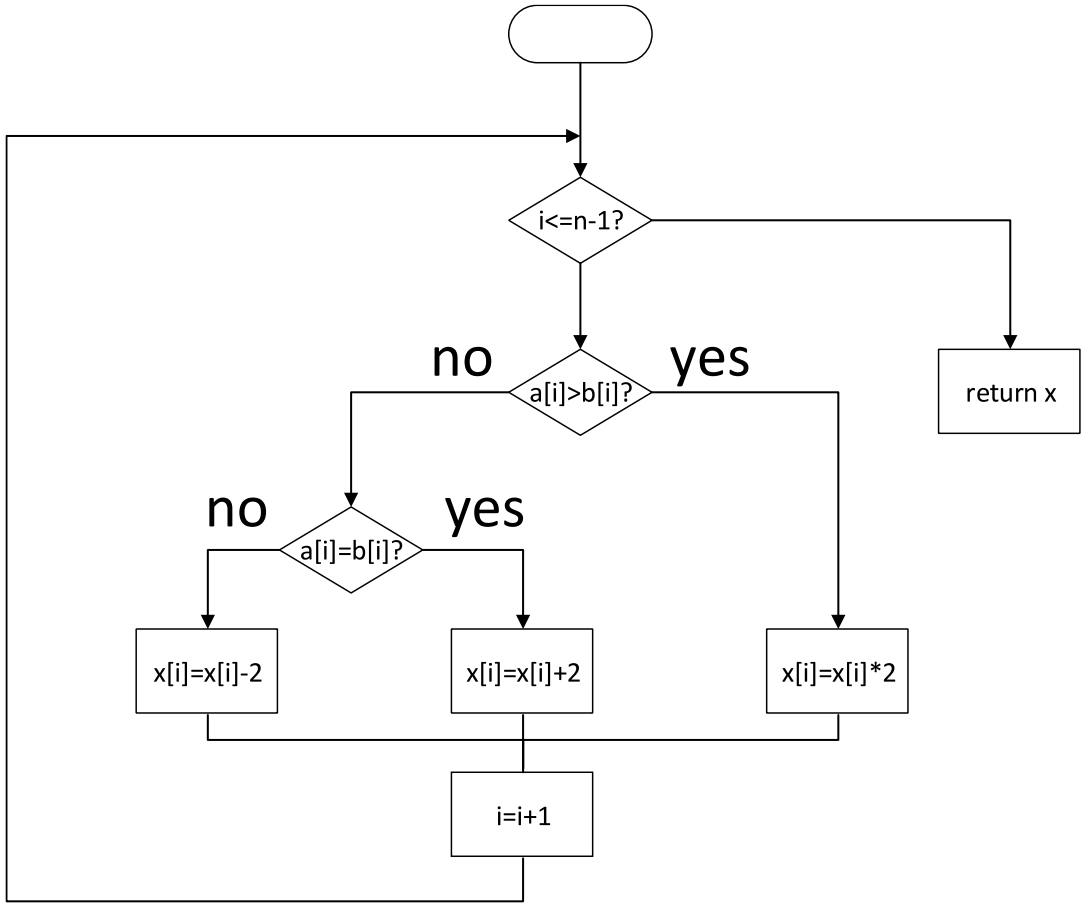


图3 带有3路分支的循环流程图

1. 概述

- 将程序中，存在实际含义的语句分别加以编号，单独的else由于依附于if语句，是if语句的一个分支，所以不单独编号。当循环次数为1时，由于a[0]和b[0]的值不同，存在三种不同路径。考虑循环的直接终止条件，共有4种不同的路径。如表1所示。

表1 循环次数为1次时的路径

编号		路径
1	a[0]>b[0]	(1)、(2)、(3)、(8)
2	a[0]=b[0]	(1)、(2)、(4)、(5)、(8)
3	a[0]<b[0]	(1)、(2)、(4)、(7)、(8)
4	循环结束	x (1)、（8）

- 可以表达为3种分支和1种循环结束路径所构成： $4=3^1+1$
- 当循环次数为2时，第1次、第2次均可能存在三种不同的路径选择。那么产生的路径分别由第1次判断条件和第2次判断条件所共同决定，如表2所示。

1. 概述

表2 循环次数为2次的路径分析

编号	两次不同的条件	路径
1	a[0]>b[0]，a[1]>b[1]	(1)、(2)、(3)、(1)、(2)、(3)、(8)
2	a[0]>b[0]，a[1]=b[1]	(1)、(2)、(3)、(1)、(2)、(4)、(5)、(8)
3	a[0]>b[0]，a[1]<b[1]	(1)、(2)、(3)、(1)、(2)、(4)、(7)、(8)
4	a[0]=b[0]，a[1]>b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(3)、(8)
5	a[0]=b[0]，a[1]=b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(4)、(5)、(8)
6	a[0]=b[0]，a[1]<b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(4)、(7)、(8)
7	a[0]>b[0]，a[1]>b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(3)、(8)
8	a[0]>b[0]，a[1]=b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(4)、(5)、(8)
9	a[0]>b[0]，a[1]<b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(4)、(7)、(8)
10	循环结束	x (1)、（7）

- 其路径分别为9条分支路径和1个循环结束分支所构成：$10=3^2+1$
- 由以上的分析可以知道，如果由n次循环所构成的路径总数为L：$L=3^n+1$
- 假设n=30，那么其所产生的路径总数为：$L=3^{30}+1=205891132094650$
- 假设机器每秒钟能够执行1000条路径测试，1年转换成为以秒为单位：$1年=365\times24\times3600秒=31536000秒$
- 那么机器完成所有路径的测试，需要的时间（以年为单位）达到：$205891132094650\div1000\div31536000=6529$ （年）

2. 图论基础

- 1736年瑞士数学家欧拉发表了图论的第一篇著名论文“哥尼斯堡七桥问题”，如图4所示。哥尼斯堡城有一条横贯全城的普雷格尔河，城的各部分用七桥联结，每逢节假日，有些城市居民进行环城周游，于是便产生了能否“从某地出发，通过每桥恰好一次，在走遍了七桥后又返回到原处”的问题。

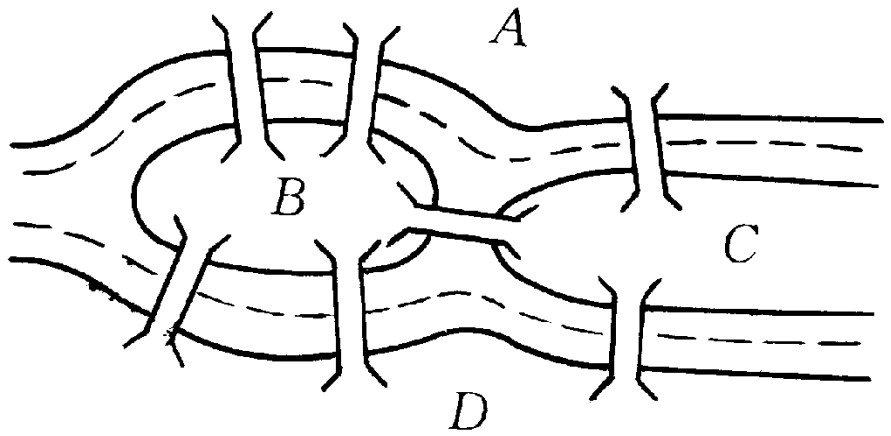


图4 哥尼斯堡七桥问题

2. 图论基础

- 哥尼斯堡城的四块陆地部分以A，B，C，和D标记。欧拉把陆地用结点表示，分别标记为A、B、C和D，同时用连接结点的边来表示对应的桥，如图5所示，这些节点和边构成了一个图。
- 图由结点和连接两个结点间的连线组成。一个图可以用三元组<V(G), E(G), Ψ(G)>表示，其中：
  - 1)V(G):非空结点的集合；
  - 2)E(G):边的集合；
  - 3)Ψ(G):从边集合E到结点无序偶(有序偶)上的函数。

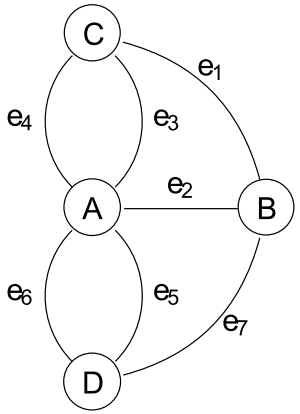


图5 哥尼斯堡七桥问题的对应的图

2. 图论基础

- 在图5中的哥尼斯堡城图，由于只要求每一座桥均被走1遍，但是对于方向并没有定义。从A走向B，与B走向A的效果是完全相同的。可以表示为：
  - 1)V(G)={A，B，C，D}
  - 2)E(G)={e1，e2，e3，e4，e5，e6，e7}
  - 3)Ψ(G)={ ψ(e1)=(A,B)，ψ(e2)=(A,B)，ψ(e3)=(A,C)，ψ(e4)=(B,C)，ψ(e5)=(B,D)，ψ(e6)=(B,D)，ψ(e7)=(C,D)}
- 关于图的相关定义为：
  - 无向边： 若边el与结点无序偶（vi,vj ）相关联。
  - 有向边： 若边el与结点有序偶<vi,vj>相关联。其中vi为el起始结点，vj为el的终止结点。
  - 无向图： 若图中所有的边都是无向边，则该图称为无向图。
  - 有向图： 若图中所有的边都是有向边，则该图称为有向图。

2. 图论基础

- 图6是一个无向图的示例。在该图中，V（G1）={A,B,C,D,E}，E(G1)={e1,e2,e3,e4,e5,e6,e7}，所有的边都是无向边，和一条边相关联的是无序偶对。例如e5关联了顶点D和E，对于e5来说，从D到E边，E到D效果是完全相同的。

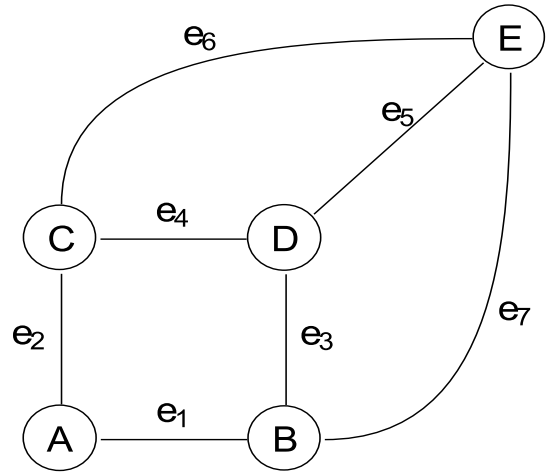


图6 无向图示例G1

2. 图论基础

- 图7是一个有向图示例，V（G2）={A,B,C,D,E}，E(G2)={e1,e2,e3,e4,e5,e6,e7}，在这个图中所有的边都是有向边，这些边具有明显的方向性。例如e3是关联的A和D，A为e3起点，D是e3的终点，A到D和D到A是不同。例如在边表示交通时间，从山底向山顶所需要的时间和从山顶向山底所需要的时间根本不同，必须区分其方向，讨论才具有实际意义。

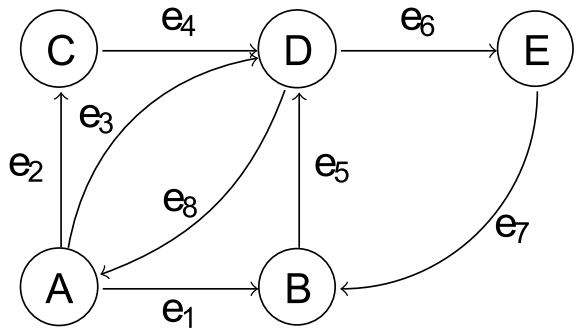


图7 有向图示例G2

2. 图论基础

- 邻接点：与一条有向边 / 无向边关联的两个结点。
- 结点的度：与结点v (v ∈V)关联的边数，称作该结点的度数，记为D(v)。
- 有向图的出度：在有向图中的结点v，以v为始点的边的条数称为结点v的出度，记为Dout(v)；
- 有向图的入度：在有向图中的结点v，以v为终点的边的条数称为结点v的入度，记为Din(v)。
- 有向图的度：有向图中某一个结点的度为结点的出度与入度之和，标记为D(v)= Dout(v)+ Din(v)。
- 图的通路：给定图G = <V，E，ψ>，设v0，v1， …， vn ∈V，e1，e2， …，en ∈E，其中ei是关联于结点vi-1，vi的边，交替序列v0e1v1 e2v2…envn 称为联结v0到vn的通路。其中v0为起点，而vn是通路的终点。

2. 图论基础

- 图的回路：若一条通路的起点和终点，则称该通路为回路。
- 在图6中，边e6关联两个结点分别结点C和结点E，结点C和结点E称为边e6的邻接结点。与结点A关联的边分别为e1和e2，其度为2。而结点B、C、D、E都有3个边和其关联，所以这几个结点的度都是3。序列Ae2Ce4De6E为结点A到E的一条通路，同理Ae1Be4De6E也是一条结点A到E的一条通路。序列Ae2Ce3De5Be1A所构成回路中，其起点和终点都是结点A，所以该序列是一条回路。在无向图中，通路和回路都没有方向的概念。回路Ae2Ce3De5Be1A和回路Ae1Be5De3Ce2A是完全相同的。
- 在图7中，e4的邻接结点为C和D，e4和e8的邻接结点都是A和D，但是e4和e8的方向是不一样的。结点C只和边e4和e2相关联，并且分别作为边e4的起点和边e2的起点，结点C的出度Dout(C)=1，入度Din(C)=1，度D(C)= Dout(C)+ Din(C)=2。序列Ae3De6E构成结点A到结点E一条通路。序列Be5De6E e7B构成了一条通路，其起点和终点都是结点B。

3. 流程图结构以及表示

- 控制流图（可简称流图）是对程序控制流进行简化后得到的，可以更加清晰地表示程序控制流结构。
- 控制流图中包括两种图形符号：结点和控制流线。
  - （1）结点可代表一个或多个语句、一个处理序列和一个条件判定。
  - （2）控制流线由带箭头的弧或线表示，可称为边，它代表程序中的控制流。
- 在流程图表示时，区分起始终止结点、循序执行结点、判断结点、输入输出结点。

图8 包含细节的控制流节点

3. 流程图结构以及表示

- 其中典型的流程图包括了顺序、分支、循环三种类型。
- 顺序结构：对于程序（函数）的所有输入，依次执行程序中每一条语句，这种结构称为顺序结构。
- 分支结构：分支结构包括两路分支结构和多路分支，两路分支一般采用IF语句来实现，一般可以表示为图9。若需要分析其中的语句细节，可以在判断结点(condition1结点)上表示判定内部的详细信息。一般而言，condition1的计算结果为逻辑值，存在真和假两种结果。依据不同的计算结果，分别执行statement2或者statement3语句或者语句组合。图9中，结点A表示判断语句，结点B和结点C可能是空语句、简单语句、也可能是由其他简单语句构成复合语句，结点D是语句B和语句C执行以后的汇聚点。结点A由IF判断语句所构成，但是判断结果为真时，执行一条路径，如果判断结果为假时，执行另外一条路径。

3. 流程图结构以及表示

- 多个if组合形成了多路分支。图10表示由多个if语句构成的多路分支结构。

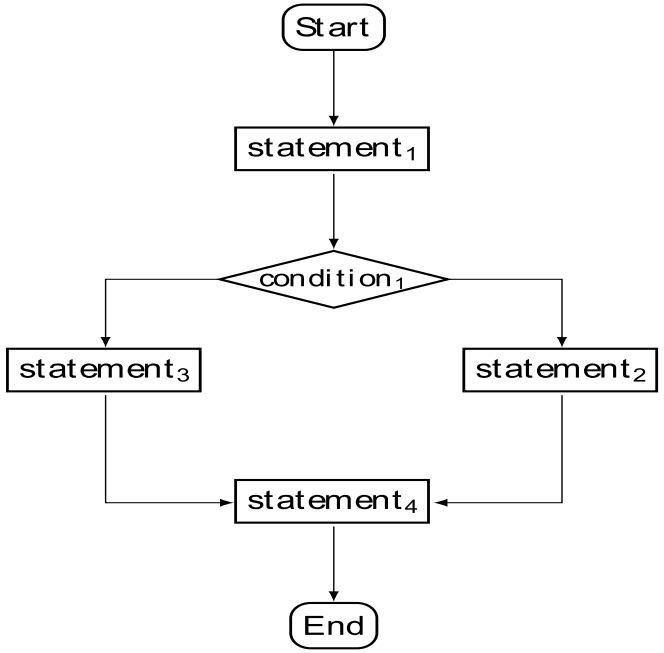


图9 具有两路分支结构

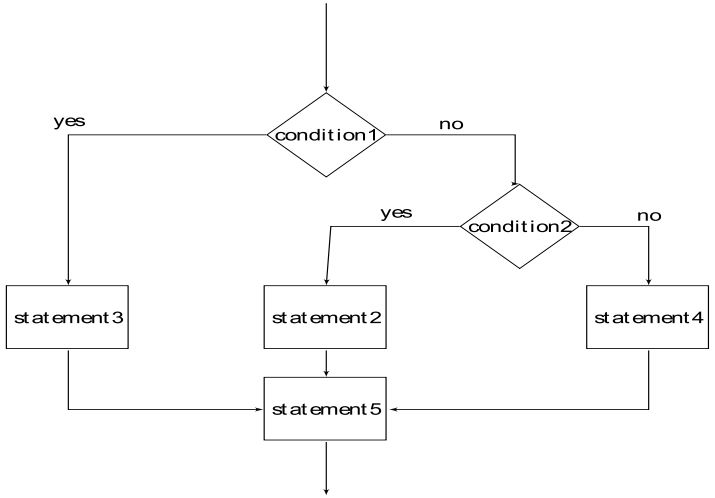


图10 单判断多路分支的结构

3. 流程图结构以及表示

- 多路分支结构在大部分编程语言中采用case语句来表示，在有些语言中， 图11表示类似case语句所构成的分支结构。多路分支有两个判断组合而构成，其效果是condition1和condition2两个条件组合而形成复合分支是一样的。这两种结构只是表示上的差别，在流程功能上并没有区别。

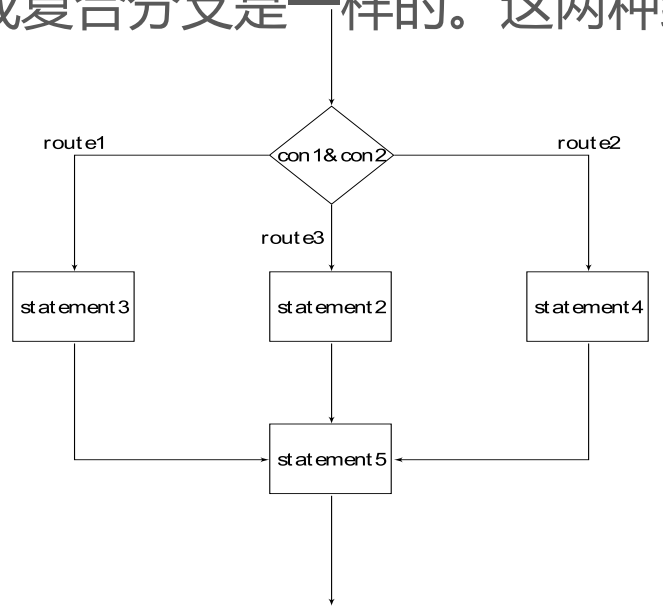


图11 组合判断多路分支的结构

3. 流程图结构以及表示

- 控制流结构。如果一个系列处理要重复执行，往往采用循环结构。例如对链表的遍历、数据的累加、执行次数的控制等。依据循环终止条件的判断所在位置，可以分成while结构和until结构。先判断后执行循环体称为while结构，在一般编程语言中都存在for循环语句也是先判断后执行，和while的差异是循环变量的所在位置，在这里统一归纳为while结构，如图12左边所示。先执行一次循环体，然后判断执行条件，称为until结构，如图12右边所示。

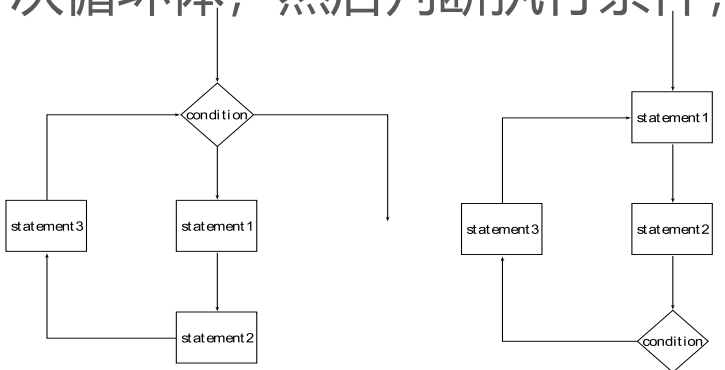


图12 循环结构

4. PYTHON中的条件和判定

- 4.1 条件与布尔值认定
  - 条件：不包含布尔操作符的布尔表达式。布尔表达式是运算结果为True或者False。
  - Python常见的布尔表达式包括大于（>）、小于（<）、等于(==)、不等于(!=)、是(is)、包含(in)等。Python对象包含了id、type、value三个要素，其中==表示两个标量值是否相同，is 表示两个对象是否为同一个对象。当一个对象有多个引用的时候，并且引且有不同的名称，称这个对象有别名(aliase)。
  - 图13在IDLE中表示两者之间的差异。两个变量a和b的值都是1.0，他们的值是一样的，但是他们是不同的对象，从a和b具有的id也可以说明。因此a==b其结果为True 而 a is b 结果是False。

4. PYTHON中的条件和判定

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37)
[MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=1.0;
>>> b=1.0;
>>> a==b;
True
>>> id(a);
41014648
>>> id(b);
30815936
>>> a is b;
False
```

图13 条件中值相同判断

4. PYTHON中的条件和判定

- 要判断两个变量是否为同一个对象，应采用 is 来执行判断两个变量是否都是指向同一个对象的引用。当其中的一个变量值改变时，系统才会创建另一个对象，然后将变量作为引用指向新的对象，如图14 所示。
- 开始时，变量a赋值2，同时变量b作为引用指向a。a和b都具有相同的值和id，显然a is b返回值为True。当将a的值修改为3以后，a和b无法共享相同的值，所以系统为其创建一个新的对象，而b仍然指向原来的对象。这是 a is b 返回的是False。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=2;
>>> b=a;
>>> id(a);
31503964
>>> id(b);
31503964
>>> a  is  b;
True
>>> a=3;
>>> b
2
>>> id(a);
31503952
>>> id(b);
31503964
>>> a  is  b;
False
>>>
```

图14 条件中对象相同判断

4. PYTHON中的条件和判定

- 如果有别名的对象是可变类型的，那么对一个别名个别元素的修改就会影响到另一个，但是他们任然是同一个对象。图15中，变量a指向列表[1,2,3]，是一个指向一个列表的引用，同时b指向同一个列表[1,2,3]。变量a和变量b都是一个变量，当修改变量b所指向对象的第1个元素时，将其修改为5，列表变更为[1,5,3]。变量a 和变量b还是指向同一个列表的对象，所以a的第2个元素值也随之修改。当把变量a指向一个新的列表[4,5,6]时，变量a和变量b不再指向同一个列表。

4. PYTHON中的条件和判定

- 除了上述讨论的情况外，包括函数在内任何返回值为True的都可以认为是一个条件。

  Python 对于布尔值有一些特别的规定，主要包括：

  - （1）任何非零数值或者非空对象都是True。
  - （2）数值零、空对象以及特殊对象None都是False。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=[1,2,3];
>>> b=a;
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> b[1]=5;
>>> a
[1, 5, 3]
>>> b
[1, 5, 3]
>>> a is b
True
>>> a=[4,5,6];
>>> a
[4, 5, 6]
>>> b
[1, 5, 3]
>>> a is b
False
>>>
```

图15 可变对象类型是否相同的判定

4. PYTHON中的条件和判定

- 4.2 判定与短路计算
  - 判定：由零个或者多个条件语句通过逻辑运算符组成，如果同一条件在判定中出现多次，则认为是不同的条件。逻辑运算符包括逻辑与(and)、逻辑或(or)、逻辑非(not)三种。在逻辑与（and） 和逻辑或（or） 构成的判定表达式中，其结果返回不是简单的True 或者False，而是对象。不是运算符左边的对象就是右边的对象。
  - 在逻辑与运算(and)运算中，Python从左到右计算每一个条件，并且停留在第一个为False的对象上。图16给出了具体的例子，在第1个表示式中，2表示True，而运算符是逻辑与(and)无法直接确认该判定值，而3也是表示True，这是已经可以确定判定的值，直接将该值作为表达式的值。在0 and 3判定中，0表示False，其和任何值的与都是False，可以直接确定表达式的值，于是0作为整个判定的值。[] and 3也是同样的情况。

4. PYTHON中的条件和判定

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 and 3;
3
>>> 3 and 2;
2
>>> 0 and 3;
0
>>> [] and 3;
[]
>>> 3 and [];
[]
>>> [] and {};
[]
>>>
```

图16 逻辑与判定的计算样例

4. PYTHON中的条件和判定

- 在由逻辑或or 所构成的判定中，Python 从左到有操作对象，然后返回第一个为真的操作对象。在图17中的2 or 3的判定表达式中，2是一个非0值，被认定为True，直接范围该值作为判定的表达式，在or 右边的表达式也不在执行。而在[] or 2判定中，[]为False无法直接确定判定的值，所以继续计算在or 右边的表达式，并将该值作为判定的值。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 or 3
2
>>> 3 or 2
3
>>> [] or 2
2
>>> 2 or []
2
>>> [] or {}
{}
>>> 0 or []
[]
>>>
```

图17 逻辑或判定的计算样例

4. PYTHON中的条件和判定

- 无论是and 还是 or 运算，Python在第一个能够确定其表达式的条件上停止运算，并将该条件的值作为整个判定的最终值。对于剩余的部分，不再继续进行计算，这个现象称为短路计算。
- 例如代码段:
  - if func1() and func2(): ……
  - 当func1()的返回值为True时，Python将不再执行函数func2()。为了保证两个函数都能够得到正确的执行，必须在判定语句之前调用它们。建议在执行and语句之前先调用它们，然后根据其结果来执行判定。代码段可以修改为
  - temp1=func1();
  - temp2=func2();
  - if func1() and func2();
- 在设计测试时，除了通用的语句、判定、条件、路径覆盖以外，必须考虑到和特定语言相关的规则作为设计的重要补充。

5. 语句覆盖

- 5.1 语句覆盖定义及其测试
  - 如果一个测试用例套，使得被测试的文件（类或者函数）中所有的语句至少被执行到一次，那么这种测试覆盖准则被称为语句覆盖。
  - 语句的覆盖程度可以采用语句覆盖率来表示。
  - 语句覆盖率，是指测试用例套所覆盖的可执行语句和可执行语句总数的比例来表示。

$$Cov_{statement} = \frac{St_{executed}}{St_{total}}$$

  - 其中:
  - $St_{executed}$ ：表示被执行到语句;
  - St_{total} ：所有的可以执行的语句。

5. 语句覆盖

- 语句覆盖就是度量被测代码中可执行语句的被执行程度。“可执行语句”，不包括代码注释、空行等，只统计能够执行的代码被执行了多少。Coverage是一个用于统计Python代码覆盖率的工具，支持HTML报告生成，最新版本支持对分支覆盖率进行统计。官方站点：http://nedbatchelder.com/code/coverage/，目前最新版本为3.7.1。
- 其在线安装的命令为：
  - easy_install coverage
- 在Windows系列下，离线安装的命令为：
  - coverage-3.7.1.win32-py2.7.exe
- 其基本命令为：
  - coverage <command> [options] [args]

5. 语句覆盖

- 最常用的命令：
  - 1)收集覆盖信息：coverage run
  - 2)查看覆盖概要信息：coverage report
  - 3)生成html格式的详细报告：coverage html
  - 4)组合多个覆盖信息：coverage combine
- 若原来的Python程序运行命令为：
  - myprog first second
  - 其中myprog为Python程序，first和second为程序命令行参数。
- 则收集覆盖信息命令为：
  - coverage run myprog first second。

5. 语句覆盖

- 在图18中一个coverage覆盖率统计结果的样例，其基本程序调用另一个文件中的sumEven函数。生成该报告的命令为：coverage run statementcoveragedemo2.py。然后生成html文件：coverage html statement- coveragedemo2.py
- coverage统计结果为3个可执行语句，从报告可以看出，无论是#开始的单行注释，以及’’’ 的多行注释都没被包含在代码行中间。在这个代码段中，调用另一个文件的函数，coverage 作为一个语句进行统计。

图18 注释和空格不作为可执行语句的统计

5. 语句覆盖

- 如果需要对sumEven函数一起统计，可以利用run 的-p参数自动生成多文件的覆盖信息，再利用combine命令组合被调用函数的统计结果，生成的统计结果如图19所示。
- 在上图的统计结果中，和if 对应的else 语句并没有作为可执行语句，因为其始终是和if语句相伴而形成，仅仅作为另外一个分支，并且是一个确定的分支，并没有产生新的语义和路径。

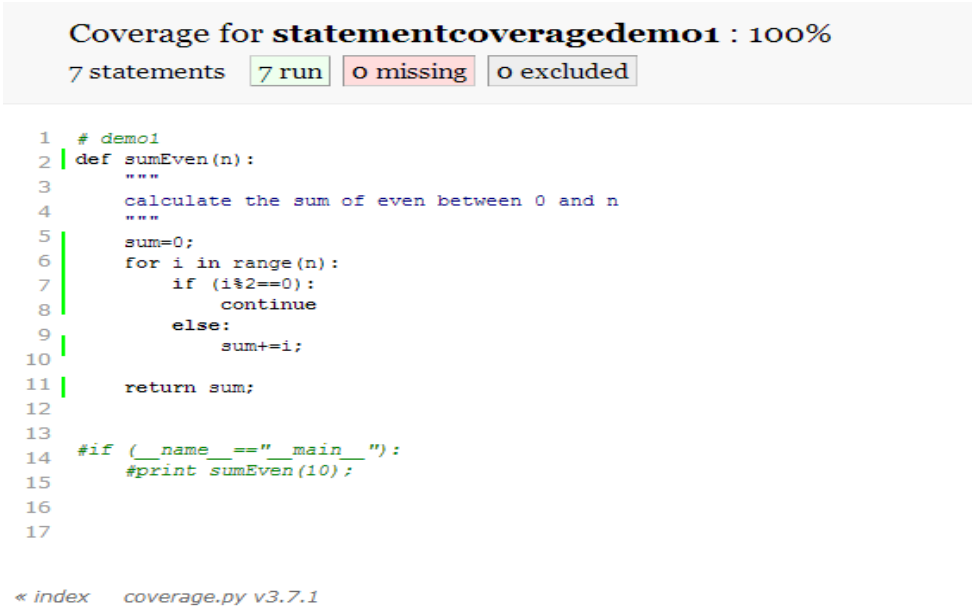


图19 和if 对应的else并不作为统计结果

5. 语句覆盖

- 类似的情况，当else出现在While 和For循环中时，统计结果类似，如图20所示。

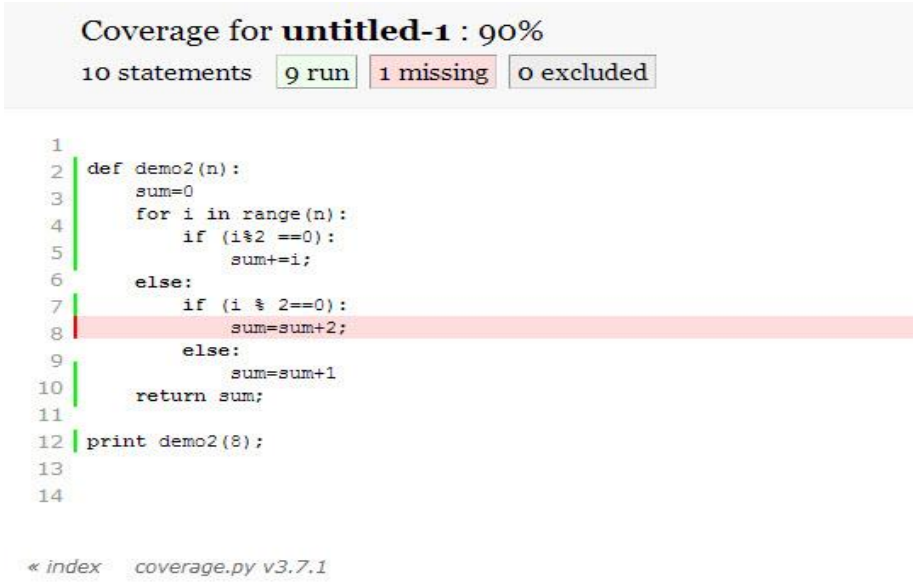


图20 循环语句中的else语句

5. 语句覆盖

- 接下来，以程序5为例来阐述语句覆盖的基本原理。

程序5 一个简单的语句覆盖演示函数

```
#state1.py
def  demo1(a,b,x):
      if (a>1 and b==0):
            x=x/a;
      if(x>1 or a==2):
            x=x+1;
      return  x;
```

- 程序5对应的控制流图如图21所示。控制流图存在两个判定，而每一个判定都存在两个分支，在判定1中的分支为(2)和(3)，分支(2)不执行任何语句直接跳转到判定2中，分支(3) 执行了x=x/a语句。第2个判定的情况类似。

5. 语句覆盖

- 为了满足语句覆盖的基本要求，只要让测试沿着(1)、(3)、(5)往下执行，就能满足语句覆盖的要求。
- 在测试程序中，若令输入a=2,b=0,x=3，函数demo1的输出用变量r表示，此时r应该为2，其对应的测试程序如程序6所示。

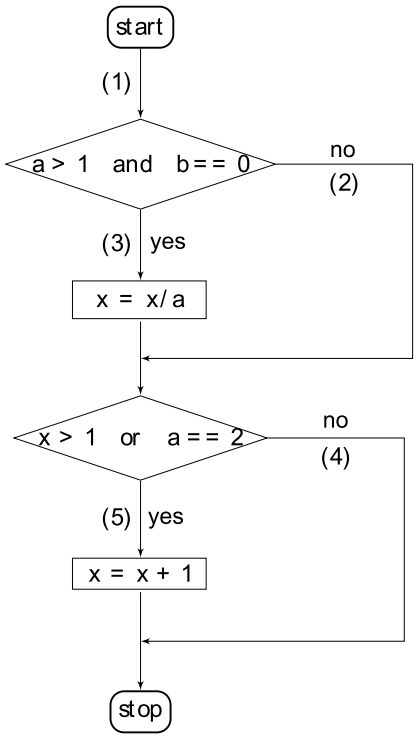


图21 程序5对应的控制流图

程序6 demo1对应的测试程序1

```
#test_state1.py
import  pytest;
import state1;
@pytest.mark.parametrize("a,b,x,r", [
      (2,0,3,2),
])
def test_demo1(a,b,x,r):
      assert state1.demo1(a,b,x)==r;
```

5. 语句覆盖

- 若是输入为a=3,b=1,x=0，输出r为0，其对应的测试用例如程序7所示，那么其执行的路径为(1)、(2)、(4)，显然语句x=x/a以及语句x=x+1都没有被覆盖到。语句覆盖率为：4/6=66.7%

程序7 demo1对应的测试程序2

```
#test_state1.py
import  pytest;
import state1;
@pytest.mark.parametrize("a,b,x,r", [
        (3,1,0,0),
])
def test_demo1(a,b,x,r):
      assert state1.demo1(a,b,x)==r;
```

5. 语句覆盖

- 5.2 语句覆盖的优缺点
  - 语句覆盖优点包括：
    - （1）检查所有语句。
    - （2）结构简单的代码的测试效果较好。
    - （3）容易实现自动测试。
    - （4）代码覆盖率高。
  - 但语句覆盖无法发现很多逻辑判断相关的问题。例如在程序5中，倘若将if (a>1 and b==0)中的逻辑与and 误写成逻辑或or，如程序8所示。

程序8 demo1一个包含缺陷的变体

```
# state2.py
def  demo2(a,b,x):
      if (a>1 or b==0):        #错误的语句，应该为if (a>1 and b==0)
            x=x/a;
      if(x>1 or a==2):
            x=x+1;
      return  x;
```

5. 语句覆盖

- 构建如程序9所示的测试，但是并不能发现该问题。

程序9 程序8 对应的测试用例

```
#test_state2.py
import  pytest;
import state1;
@pytest.mark.parametrize("a,b,x,r", [
        (2,1,0,1),
])
def test_demo2(a,b,x,r):
        assert state2.demo2(a,b,x)==r;
```

程序10 隐含循环次数错误的程序

```
#state3.py
def demo1(a,k):
        num=0;
        n=5;
        for i in range(1,n,1):
                if (a[i]==k):
                        num=num+1;
        return num;
```

- 同样在循环语句中，也存在尽管测试用例实现了语句覆盖，但是仍无法发现其中隐含的错误。例如，求列表a中从第1个到第n个元素中，其值等于k的个数，有个实现如程序10所示。

5. 语句覆盖

- 在Python中，range(1,n,1)函数的范围为1到n-1，不包括n。正确的循环判断应该range(1,n+1,1)或者range(0,n,1)。设计了一个语句覆盖的测试程序，如程序11所示，其包含的测试用例实现了语句全覆盖，但是无法发现该错误。

程序11 实现程序10全语句覆盖的测试程序

```
#test_state3.py
import  pytest;
import state2;
@pytest.mark.parametrize("a,k,r", [
        ([0,3,2,3,4,5],3,2),
])
def test_demo1(a,k,r):
       assert state3.demo1(a,k)==r;
```

5. 语句覆盖

- 5.3 语句覆盖与死代码
  - 所谓死代码（Dead Code）就是无论设计多少个测试用例，执行流程都不能到达的代码。
  - 在Python中遇到return语句以后直接返回，而不论后面是否存在其他的代码。 在程序12中，print 语句直接放在return 语句后面，print语句将永远无法执行到。

程序12 return 语句后的死代码

```
def f(self):
    return 0
    print 'dead code'
f()
```

5. 语句覆盖

- 这种缺陷比较直观，一般的静态扫描工具都能够直接发现，甚至在一些IDE都提供了内置死代码缺陷检测功能。图22 给出Wing IDE的关于简单死代码的警告性信息：

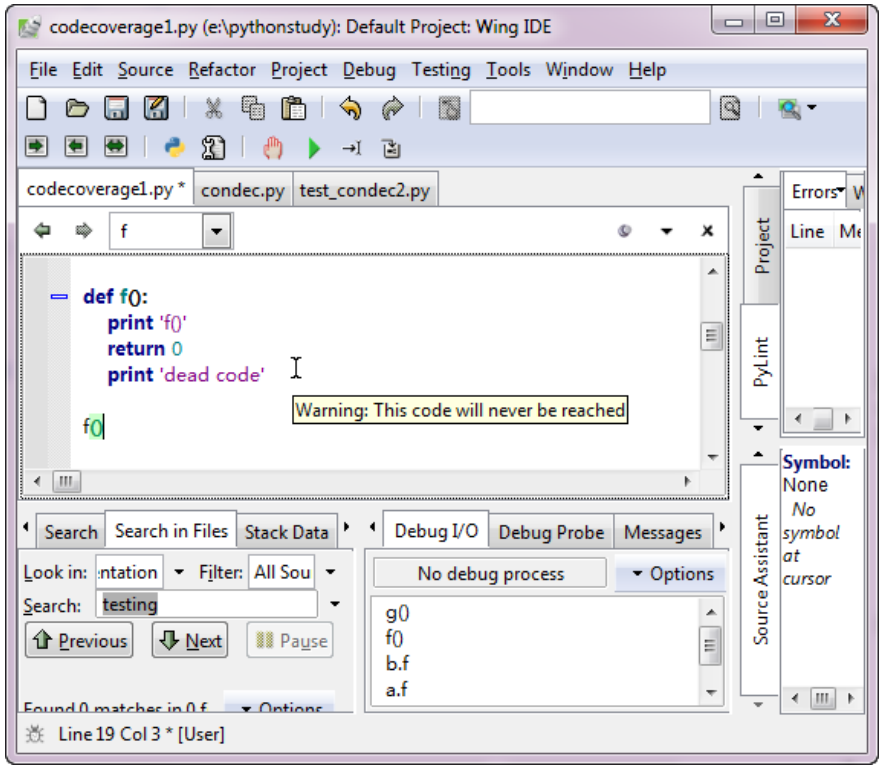


图22 Wing IDE 给出了简单死代码的提醒

5. 语句覆盖

- 程序13中，return 语句位于if 语句体中间，而后面的print 语句是与if 语句处于同一个层次。但是由于not a 在这代码段中是一个永真的表达式，实际上后面的print 语句是无论如何都不可能执行到。但是由于其通过一个表达式来判定，仅从静态的语法结构上比较难发现类似的死代码。一般需要通过语句覆盖才能发现类似的缺陷。

程序13 相对比较隐蔽的死代码

```
def  g():
        print 'g()'
        a=[];
        if not a:
                return 0
        print 'dead code'
```

5. 语句覆盖

- 另外两个在用于控制循环的语句：break语句和continue 语句，也比较容易造成死代码。break语句用于跳出其所在的for或者while循环，break语句用来终止循环语句，即循环条件没有False条件或者序列还没被完全递归完，也会停止执行循环语句。在嵌套循环中，break语句将停止执行最深层的循环，并开始执行下一行代码。continue 语句用来跳过当前循环的剩余语句，然后继续进行下一轮循环。一般情况下，break或者continue 语句都是使用在if 语句体之中。和break 或者continuet同一缩进层次，并且在它们之后的语句都是永远不可达的。

5. 语句覆盖

- 程序14中，右边为和需求规格说明书符合的代码，从1开始一直到累加到100，并输出每次累加结果。若累加和大等于50，也停止累加。print 语句处于while模块中和if 语句并列的层次。由于编程人员出现错误，将print 语句缩进到if 语句块中间，并且处于break语句的后面，成为死代码。这个程序中，print 语句仅仅用于使得问题更加清楚而特意设置的，只要直接运行该函数就能发现该问题。

程序14 break 语句所构成的死代码

```
# Error function including dead code
def k( ):
    x=1;
    print 'k()'
    sum=0
    while x<100:
        sum+=x
        if (sum>=50):
            break
            print sum
        x+=1
k( )
```

```
#Correct function
def k1( ):
    x=1;
    print 'k1()'
    sum=0;
    while x<100:
        sum+=x
        if (sum>=50):
            break
        print sum
        x+=1
k1( )
```

5. 语句覆盖

- 程序15右边用于输出1到10之间的奇数，当x为偶数时跳过输出语句直接进行下一次循环，当x为奇数时执行print 语句，将x的值输出。而在左边代码中，由于出现错误将print语句放在if 语句块中间，并且紧接在continue语句。在x为奇数时，直接跳过整个if 语句体，而当x为偶数时，由于遇到了continue语句，直接跳过print 语句执行下一次循环。无论在哪一种情况，无法执行到print语句。。

程序15 continue语句所构成的死代码

```
# Error function including dead code
def l( ):
    x=0;
    print 'l()'
    while x<10 :
        x+=1;
        if (x % 2==0):
            continue
            print x;
l( )
```

```
#Correct function code
def l1( ):
    x=0;
    print 'l1()'
    while x<10 :
        x+=1;
        if (x % 2==0):
            continue
        print x;
l1( )
```

6. 判定覆盖

- 6.1 判定覆盖
  - 判定覆盖(Decision Coverage)就是设计若干个测试用例，使得程序中每个判断的取真分支和取假分支至少经历一次。
  - 判定覆盖率Dec定义如下： $\mathrm{Dec}=\frac{Br_{executed}}{Br_{total}}$
  - 其中：
  - $Br_{executed}$：表示已经被测试用例覆盖过的判定输出分支。
  - Br_{total} ：表示所有的判定输出分支总数。

6. 判定覆盖

- 6.2 两路分支覆盖
  - 以程序5为例，在这个例子中，存在两个判定，第1个判定结果会经过路径(2)和路径(3)两种情况，而第2个判定结果会经过路径(4)和路径(5)两种情况。判定覆盖，要求把这些所有的路径全部覆盖。根据这个要求，产生满足判定覆盖的测试用例，如表 3 所示。

表3 满足判定覆盖的用例及其覆盖情况

编号	取值情况				判定1	判定2	覆盖
	a	b	x	r			
1	2	0	3	2	Yes(True)	Yes(True)	1、3、5
2	3	1	0	0	No(False)	No(False)	1、2、4

6. 判定覆盖

- 显然，这两个用例已经覆盖了两个判定的两个结果，这两个测试用例，利用pytest框架表示的测试如程序17所示。

程序17 程序5的满足分支/判定覆盖的测试代码

```
#test_state1.py
import  pytest;
import state1;
@pytest.mark.parametrize("a,b,x,r", [
        (2,0,3,2),
        (3,1,0,0),])
def  test_demo1(a,b,x,r):
      assert state1.demo1(a,b,x)==r;
```

6. 判定覆盖

- 6.3 多路分支覆盖
  - 使用if-elif-else，或者使用嵌套的if-else语句实现都能实现多路分支覆盖。
  - 例如，为了根据学生成绩的不同范围，执行不同的处理，可以采用if-elif-else实现。为了演示的方便，仅仅返回一个学生成绩的等级符号，如程序18所示。

程序18 学生等级划分程序

```
#mutipleif.py
def  myRank(score):
    if(score>=90) and (score<=100):
        return "A"
    elif(score>=80 and score<90):
        return "B";
    elif(score>=60 and score<80):
        return "C";
    else:
        return "D";
```

6. 判定覆盖

- 该程序共有4路分支，每一个分支处理不同分数段的成绩，若要产生满足分支覆盖准则的测试，必须在4个分数段均需遍历。构造4个处于不同分数段的分数：55,65,89,90作为测试的输入。

程序19 学生等级划分程序的测试

```
#test_myRank1.py
import  pytest
from app.mutipleif import myRank
@pytest.mark.parametrize("x,r", [
        (55,"D"),
        (65,"C"),
        (89,"B"),
        (90,"A"),])
def  test_demo1(x,r):
        assert myRank(x)==r;
```

6. 判定覆盖

- 在大多程序设计语言中， 除了利用if语句实现多路分支处理以外，还往往提供switch-case语句实现多路分支，当变量等于不同的特定值时，选择不同的执行路径。Python语言不提供switch-case语句，可以使用字典实现多路分支功能。
- 程序20根据不同的运算符号执行不同的运算功能，这是一个典型的多路分支/判定程序。其核心功能通过定义字典calculation来实现的。

程序20 简易计算器

```
#decision.py
def add(a,b):
    return a + b
def multi(a,b):
    return a* b
def sub(a,b):
    return a - b
def div(a,b):
    return a/ b  #b is non-zero
def calc(type,x,y):
    calculation  = {'+':lambda:add(x,y),
                    '*':lambda:multi(x,y),
                    '-':lambda:sub(x,y),
                    '/':lambda:div(x,y)}
    return calculation[type]()
```

6. 判定覆盖

- 为了满足这个程序的多路分支测试，必须将+,-,*,/ 四种运算符都覆盖到，产生的测试如程序21所示。

程序21 简易计算器的测试用例

```
#test_myRank1.py
import  pytest
from app.decision import calc
@pytest.mark.parametrize("op,a,b,r", [
        ("+",6,3,9),
        ("-",6,3,3),
        ("*",6,3,18),
        ("/",6,3,2),])
def test_demo1(op,a,b,r):
        assert calc(op,a,b)==r;
```

6. 判定覆盖

- 6.4 不可达分支
  - 有些分支，无论设计多少个测试用例，均无法使得该分支的语句体得到执行，那么该分支称为不可达分支。
  - 不可达分支，是由于前置路径的相关性导致一些判定的取值始终为True或者False。通常存在下面几种情况。
    - 1）值依赖不可达分支：如果一个不可达路径是由判定引用了一个已经赋值的变量而导致该路径不可达，成为值依赖不可达分支。最简单的一种形式是一个变量被赋予一个常数，该常数会导致判定值的为一个特定值。

6. 判定覆盖

- 程序22中具有一个简单的分支，但是其判定结果是永远为False，print 语句处在一个不可到达的分支中。

程序22 值依赖可达分支样例

```
def  g():
        print 'g()'
        if []:
                print 'hello'
g()
```

程序23 判定依赖不可达分支样例

```
def decdemo(x,y):
        k=5
        if (x<y):
                a=k+x
                if (x<y+1):
                        b=a+1
                else:
                        b=2*a
         else:
                b=k+y
         return  b;
```

- 2）判定依赖不可达分支。若前面的某一个特定判定导致本判定的取值为一个特定值，称为判定依赖不可达分支。
- 在程序23中，当第1个判定值为True时，x的值始终小于y，第2个判定的值也只有1种情况，其必然为True。在这种情况下，第2个判定取值无法达到False，也就是说b=2*a这条语句，始终是无法执行到的。

6. 判定覆盖

- 6.5 异常处理多分支覆盖
  - 在程序的执行过程中，必然会出现各种各样的异常情况。所谓异常是指改变程序中控制流程的事件。
  - 异常发生的时间一般不受软件所控制，异常可能会发生，也可能不会发生。异常发生时，如果没有合适的异常处理机制，将会导致控制流程中断，程序处于完全不受控状态。
  - 常用的方法提供异常处理，用户根据可以预见的异常发生情况，编写合适的异常处理模块，处理发生的异常。在异常发生时，程序中止当前正在执行的代码块，在这个代码块中所有后继代码将不再执行，程序逻辑进入异常处理器，依据用户的异常处理模块执行相应的动作。一个代码段中，可能会同时包含多个异常情况，当发生不同的异常，程序将进入不同的异常处理模块。

6. 判定覆盖

- 在Python中，异常会根据错误自动地被触发，也能够由代码触发或者截获。异常是Python对象，表示一个错误。当Python程序发生异常时需要捕获处理它，否则程序会终止执行。图23给出了一个异常的例子。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def add(a,b):
                        c=a+b
                        print c

>>> add(2,4)
6
>>>
>>> add('abc','efg')
abcefg
>>>
>>> add('a',2)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    add('a',2)
  File "<pyshell#11>", line 2, in add
    c=a+b
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

图23 一个加运算函数遇到的异常

6. 判定覆盖

- 在这个例子中，利用交互环境给出了一个加运算的函数add的例子。当调用具有合法加运算的对象时，其输出了加运算的结果。例如add(2,4)的结果为6，而add(‘abc’,’efg’)的结果为abcefg。然而当调用add('a',2)时，在Python语言中，’a’和2属于字符型和整形，两者之间不能执行运算。系统抛出异常，并给出了调用栈的调用关系，指明该异常的类型为TypeError，在异常语句后面后面的print 语句将不再被执行。

6. 判定覆盖

- Python系统已经内置很多标准异常，表4是部分标准Python异常，用户也可以定义特殊的异常。

表4 Python 常见的标准异常

编号	异常	含义
1	Exception	常规错误的基类
2	ZeroDivisionError	除(或取模)零 (所有数据类型)
3	EOFError	没有内建输入,到达EOF 标记
4	IOError	输入/输出操作失败
5	NameError	尝试访问一个没有申明的变量
6	KeyError	请求一个不存在的字典关键字
7	ValueError	传给函数的参数类型不正确，比如给int()函数传入字符串形

6. 判定覆盖

- 在Python中，捕捉异常使用try/except语句。try/except语句用来检测try语句块中的错误，从而让except语句捕获异常信息并处理。一般的异常处理框架如图24 所示。
- 异常处理的一般流程如下：
  - 若在语句块1的执行时发生了异常，程序跳出语句块1，执行第一个符合触发异常except的语句块。例如在语句块中，触发了expname2异常，那么其执行语句块3。然后执行finally中的语句块6，程序的流程将到try语句的后继语句。
  - 若在语句块1的执行时发生了异常，没有符合的except，异常将直接执行finally中的语句块6，程序的流程将到try语句的后继语句。
  - 若语句块1的执行未发生异常，如果存在else语句，那么程序执行else的语句块。然后执行finally中的语句块6，程序的流程将到try语句的后继语句。

```
try:
<语句块1>                                    #可能引发异常的代码块
except <expname1>:
<语句块2>                                    #如果在try部份触发了expname1异常
except < expname2>，<数据>:
<语句块3>                                    #如果引发了expname2异常，获得附加的数据
except < expname3, expname4>，<数据>:
<语句块4>                                    #如果引发了列表中的异常 (expname3, expname4) ，#并获得附加的数据
else:
<语句块5>                                    #如果没有异常发生，执行语句4
finally:
<语句块6>                                    总会执行的语句块
```

图24 异常处理框架

6. 判定覆盖

- 从上述流程上看，异常处理的流程在实际含义上类似于多路分支，图25给出了一个异常的实际例子。

异常	类似的多路分支含义

def dowork():
func1()
 	
try：
dowork( )
except expname1:
expthandle1()
except expname2:
expthandle2()
	def dowork ()：
	if (func1()==expname1):
	return
	expname1;
	elseif
	(func1()==exceptname2):
	return
	exceptname2;
	
	def mainfunc()
	if(dowork()==expname1):
	
	expthandle1()
	else if(dowork()==expname2)
	
	expthandle2()

图25 异常处理在语义上和if 分支类似

6. 判定覆盖

- 类似分支覆盖，在异常处理代码中，可以使用异常覆盖准则来设计测试。
- 异常覆盖准则：设计的测试用例集，使得代码中所有的异常处理块必须被执行过一次。
- 若将图23中运算放在一个异常处理中，将其代码加运算发在try 代码块中，当出现异性错误TypeError异常时，输出’ ERROR ‘的提示信息，如程序24所示。

程序24 具有异常处理的加运算程序

```
def add(a,b):
    try:
        c=a+b
    except:
        msg=  'ERROR'
    else:
        msg=  str(c)
    finally:
        return  str(msg)
```

6. 判定覆盖

- 这个程序包含了错误时，输出信息为”ERROR”，以便于上层调研函数继续能够处理。为了测试这个程序，必须构造两个测试用例，其中一个为具有合法加运算的两个变量，另一个是可能出现的非法变量，如程序25 所示。

程序25 具有测试加运算的异常处理的两个用例

```
#test_exceptadd.py
import  pytest
from app.addexcept import add;
@pytest.mark.parametrize("a,b,r", [
        (2,3,'5'),
        ('a',2,'ERROR'),
])
def test_add(a,b,r):
        assert add(a,b)==r;
```

6. 判定覆盖

- 若在一个代码段中，包含了多个异常，那么必须覆盖所有的异常。如果需要执行两个数的除法运算，那么可能出现两种异常情况：异常情况1，传入该函数的不是合法的数值。异常2，除法中的分母为0，如程序26所示。在实际应用中，应该是先判断分母是否为零，然后执行运算，在这里仅仅是为了说明异常处理的演示。

程序26 具有异常处理的除法运算

```
def divdemo(a,b):
    try:
        avalue=int(a)
        bvalue=int(b)
        cvalue=avalue/bvalue
    except ValueError:
        result="NoNumber"
    except ZeroDivisionError:
        result="DeviedByZero"
    else:
        result=str(cvalue)
    finally:
        return result
```

6. 判定覆盖

- 显然，对于这个程序，为了满足异常覆盖的要求，需构造3个测试用例。其中一个测试是正确运算的数据，第二个用例不是合法的数值，第三个用例分母为零的情况，如程序27所示。

程序27 除法的程序的三个测试用例

```
#test_multipleexcept.py
import  pytest
from app.multipleexcept import divdemo;
@pytest.mark.parametrize("a,b,r", [
        (6,3,'2'),
        ('a',4,'NoNumber'),
        (4,0,'DeviedByZero'),
])
def test_demo1(a,b,r):
      assert divdemo(a,b)==r;
```

6. 判定覆盖

- 6.6 复合判定覆盖
  - 若由多个判定嵌套而形成，称为嵌套型分支；若由多个分支串接起来的，称为连锁型分支。更加复杂的是嵌套型分支和连锁型分支的组合。
  - 所谓嵌套型分支，是指在一个判定/分支中的一条分支中间，又存在判定/分支的情况。在嵌套型分支中，其最小的测试用例数等于其判定输出分支的最小测试用例数和，若不存在分支情况，那么其用例数记为1。若一个判定点的测试用例数记为 TC_d，子分支n的测试用例数记为 TC_{sdn}，那么测试用例数按如下公式计算：

$$\begin{cases} TC_d = 1 \\ TC_d = TC_{sd1} + TC_{sd2} + \cdots + TC_{sdn} \end{cases}$$

6. 判定覆盖

- 显然这是一个递归的过程。图26中的(a) 是一个嵌套分支的具体例子。在判定P1上，存在三个分支，其中判定P2有3个分支，判定P3有2个分支。那么判定点P1开始计算的满足分支覆盖的测试用例数，分别由其3个判定输出分支的测试用例数所构成。其总测试用例数为：TCp1=3+1+2=6条。以p1为分析点，这6条路径如表5所示。

表5 嵌套型分支产生的测试用例

编号	输出分支	路径
1	p	p-p-s-p-p
2		p-p-s-p-p
3		p-p-s-p-p
4	s	p-s-p
5	p	p-p-s-p-p
6		p-p-s-p-p

6. 判定覆盖

- 图26中的(b)是一个连锁型分支的具体例子，由三个判定串接而形成。第1个判定具有3个输出分支，第2个判定具有2个输出分支，而第3个判定具有4个输出分支。在连锁型分支中，其分支覆盖的路径，其最小测试用例数由串接的判定中输出分支中最多的一个判定所决定。在串接的代码中，测试用例执行的一条路径将经过所有的判定点，每次产生的新路径都经过一个判定点的不同输出，除非该判定点的所有输出分支都已经被覆盖过。在图26(b)中，p5判定点的输出分支数最多，所以该代码段的测试用例数共4条。从p1开始，从左到右选择每个判定点不同的输出分支，如果所有的分支点都已经被覆盖过，那么从左到右重新选择输出分支。最终产生的测试用例如表6所示。

6. 判定覆盖

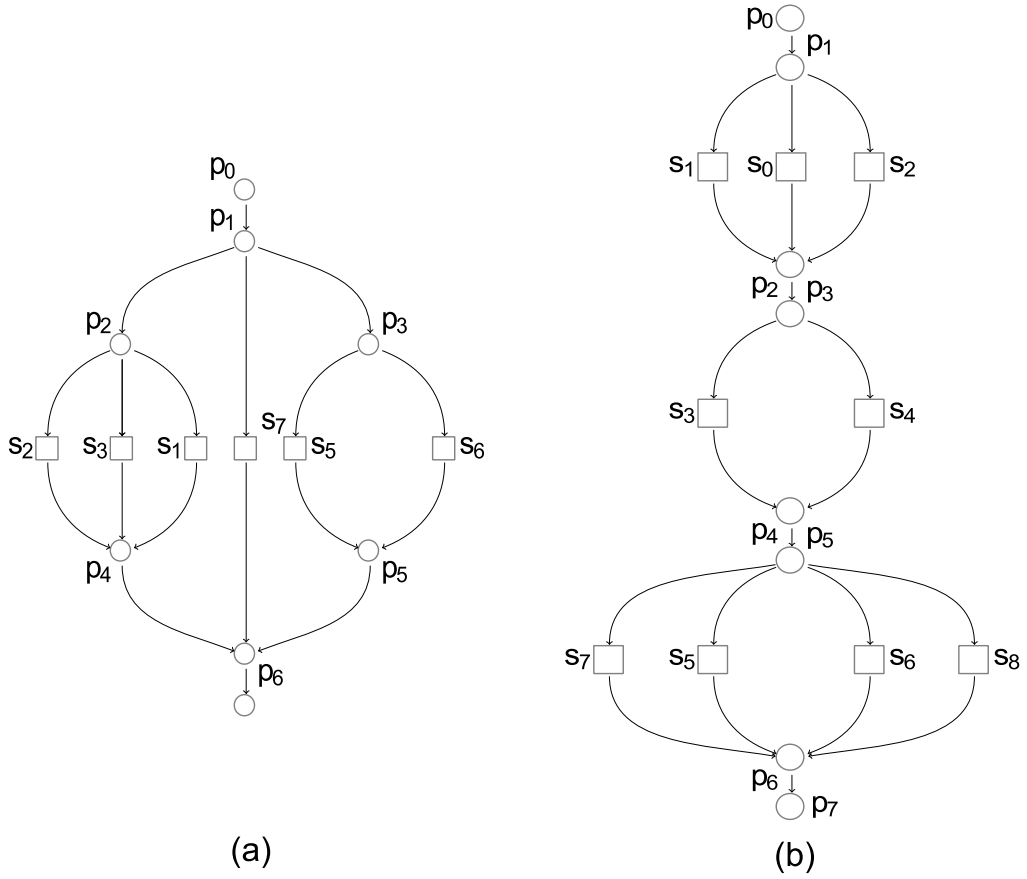


图26 嵌套型和连锁型分支

表6 连锁性分支所产生的测试用例

编号	路径
1	p-p-s-p-p-s-p-p-s-p-p
2	p-p-s-p-p-s-p-p-s-p-p
3	p-p-s-p-p-s-p-p-s-p-p
4	p-p-s-p-p-s-p-p-s-p-p

6. 判定覆盖

- 在连锁性分支覆盖中，由于只考虑每一个判定点的输出分支情况。而实际上，对于每一个判定点而言，由不同分支路径进入到该判定点，会对其后继产生影响。而前面的讨论过程中，并没有考虑这种影响。引入分支交换覆盖准则：对于每一个判定点，其每一个输入分支和输出分支的组合必须被测试用例覆盖一次。在分支交换覆盖准则中，对于判定点的每一个输入分支，都必须对应所有的输出分支。在图27 中，通过语句s0的输入分支，必须和所有的输出分支{s0′ , s1′ , ……,s′ m-1, sm′ }构成输入输出对：

  {(s0′ ,s0′),(s0′ ,s1′),……, (sn′ ,s′ m-1), sn′ ,s′ m)}

  若对于其他的输入分支，和必须和所有的输出分支构成输入和输出关系。若一个判定点具有n个输入分支路径，m个输出分支路径，那么满足分支交换覆盖的测试用例数为n×m。

6. 判定覆盖

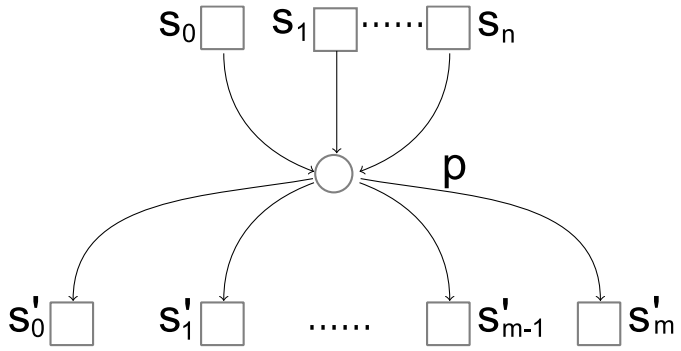


图27 一个具有n个输入分支，m个输出分支的判定点

- 在图26(b)中，满足分支交换覆盖的测试用例数量为3×2×4=24条，如表7所示。

表7 满足分支交换覆盖的测试用例

编号	路径
1	p-p-s-p-p-s-p-p-s-p-p
2	p-p-s-p-p-s-p-p-s-p-p
3	p-p-s-p-p-s-p-p-s-p-p
4	p-p-s-p-p-s-p-p-s-p-p
5	p-p-s-p-p-s-p-p-s-p-p
6	p-p-s-p-p-s-p-p-s-p-p
7	p-p-s-p-p-s-p-p-s-p-p
8	p-p-s-p-p-s-p-p-s-p-p
9	p-p-s-p-p-s-p-p-s-p-p
10	p-p-s-p-p-s-p-p-s-p-p
11	p-p-s-p-p-s-p-p-s-p-p
12	p-p-s-p-p-s-p-p-s-p-p
13	p-p-s-p-p-s-p-p-s-p-p
14	p-p-s-p-p-s-p-p-s-p-p
15	p-p-s-p-p-s-p-p-s-p-p
16	p-p-s-p-p-s-p-p-s-p-p
17	p-p-s-p-p-s-p-p-s-p-p
18	p-p-s-p-p-s-p-p-s-p-p
19	p-p-s-p-p-s-p-p-s-p-p
20	p-p-s-p-p-s-p-p-s-p-p
21	p-p-s-p-p-s-p-p-s-p-p
22	p-p-s-p-p-s-p-p-s-p-p
23	p-p-s-p-p-s-p-p-s-p-p
24	p-p-s-p-p-s-p-p-s-p-p

7. 条件覆盖

- 7.1 简单条件覆盖
  - 一个简单判定是由一个简单条件语句所构成，那么一个判定就是一个条件判断语句。而在实际的应用中，一个判定往往是由多个条件组合而成。组合两个条件的逻辑运算符是：逻辑与、逻辑或。其逻辑与真值表如表8所示。

表8 逻辑与的真值表

条件表达式1	条件表达式2	逻辑与
True	False	False
False	True	False
False	False	False
True	True	True

7. 条件覆盖

- 当逻辑与的结果为True时，可以推断出条件1和条件2的值均为True。在逻辑与的结果为False时无法判断条件1和条件2的值是True还是False。例如在python语言中的一个判定：a>1 and b==0。就是由a>1 和b==0两个条件通过逻辑与运算所构成，当a<=1 或者b!=0 都有可能使得结果为False。在python交互环境IDLE中，很容易验证，如图28所示。
- 逻辑或真值表如表9所示。

表9 逻辑或的真值表

条件1	条件2	逻辑或
True	False	True
False	True	True
True	True	True
False	False	False

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=0
>>> b=1
>>> a>1 and b==0
False
>>> a=3
>>> b=1
>>> a>1 and b==0
False
>>> a=0
>>> b=0
>>> a>1 and b==0
False
>>>
```

图28 IDLE中逻辑与验证

7. 条件覆盖

- 在逻辑或中，只有判定结果为False时能够确定条件1和条件2的表达式均为False，在结果为True时，无法判定哪个值为True。例如在python语言中的一个判定：a>1 or b==0。就是由a>1 和b==0两个条件通过逻辑或运算所构成，当a>1 或者b==0 都有可能使得结果为True。在python交互环境IDLE中，很容易验证，如在图29所示。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=2
>>> b=1
>>> a>1 or b==0
True
>>> a=0
>>> b=0
>>> a>1 or b==0
True
>>> a=2
>>> b=0
>>> a>1 or b==0
True
>>> a=0
>>> b=1
>>> a>1 or b==0
False
>>>
```

图29 IDLE中逻辑或验证

7. 条件覆盖

- 从上面的分析看出，分支覆盖并不能反映出判定中不同条件的情况，为了区分不同条件情况，引入条件覆盖(Condition Coverage)：构造一组测试用例，使得每一判定语句中每个逻辑条件的可能值至少满足一次。
- 程序28存在两个判定，判定1由x>4以及z<8两个条件由逻辑与运算连接而成，第2个判定由x==4以及y>6两个条件通过逻辑或运算连接而成。

程序28 条件覆盖的程序

```
def work(x,y,z):
    k=0;
    j=0;
    if (x>2 and z<8):          判定1
        k=x*y;
        j=k+z;
    if (x==4 and y>6):         判定2
        j=x*y+5;
    j=j%4;
    return j;
```

7. 条件覆盖

- 如果用T1表示x>2结果为真，F1表示x>2结果为假。用T2表示z<8结果为真，F2表示z<8结果为假。T3表示x==4结果为真，F3表示x==4结果为假。用T4表示y>6结果为真，F4表示y>6结果为假。设计测试用例，确保这8个条件值均出现一次就满足条件覆盖准则。
- 程序29 给出了和程序28 对应的满足条件覆盖的测试程序。

程序29和程序28 对应的满足条件覆盖的测试程序

```
#test_switch1.py
import  pytest
from app.condition import calc
@pytest.mark.parametrize("x,y,z,r", [
        (3,8,7,3),
        (1,5,9,0),
        (4,5,8,0),
])
def test_demo1(x,y,z,r):
        assert calc(x,y,z)==r;
```

7. 条件覆盖

- 在上述测试数据中， 测试用例(3,5,7,2)实现了T1T2F3T4的覆盖，而(1,5,9,0)实现了F1F2F3F4的覆盖。检查条件覆盖情况，T3并没有被覆盖，需要补充覆盖T3的测试用例。这个测试用例具有(4，-，-，-)的形式，也就是x=4，其他变量的取值不做限定，可以根据其他信息作为启发式进行补充，现在取(4,5,8,1)作为该测试用例。
- 进一步观察可以发现判定点1中的x>2和判定点2中的x==2，存在使其条件结果重叠的情况，在设计测试用例时，可以用同样数据来覆盖更多的条件。即取x=4，那么x>2以及x==4均为True，当x=1时，那么x>2以及x==4均为False。对程序29所给出的测试用例进一步的精简，如程序30所示。在这个测试程序中，第1个测试用例(4,5,7,1)实现了T1T2T3T4的覆盖，而第2个测试用例实现了F1F2F3F4的覆盖。

7. 条件覆盖

程序30 满足条件覆盖的另一测试程序

```
#test_switch1.py
import  pytest
from app.condition import calc
@pytest.mark.parametrize("x,y,z,r", [
        (4,8,7,1),
        (1,5,9,0),
])
def test_demo1(x,y,z,r):
        assert calc(x,y,z)==r;
```

7. 条件覆盖

- 7.2 条件判定覆盖
  - 条件覆盖关注的是一个判定中间的每一个条件的真假情况，而判定覆盖关注的是整个判定的最终结果。依据逻辑与、逻辑或的两个运算真值表，不同的组合可能产生同样的输出，满足条件覆盖不一定满足判定/分支覆盖准则。
  - 程序31包含了两路的分支，并且其判定点也仅仅由两个条件通过逻辑运算与连接而形成。

程序31 包含简单与运算的判定

```
#import math
#E:\2014\bookpub\software_testing\python\pytestdemo6\app\condec.py
def condec(a,b):
        x=12;
        if (a==0 and b>2):
                x=x/a;
        else:
                x=x+b;
        x=x+1;
        return x;
```

7. 条件覆盖

- 和前面类似，判定点中的两个条件分别存在结果为True和False两种情况，只要构建的测试用例集分别使得两个条件分别取得结果为True和False，就满足条件覆盖准则。测试程序32所包含的测试数据就可以满足上述条件。

程序32 带有逻辑或判定中满足条件覆盖的测试

```
#test_condec1.py
import  pytest
from app.condec1 import condec;
@pytest.mark.parametrize("a,b,r", [
        (3,4,17),
        (0,1,14),
])
def test_demo1(a,b,r):
        assert condec(a,b)==r;
```

7. 条件覆盖

- 在这个测试程序包含了两个测试用例，分别为(3,4,17)和(0,1,14)。其中(3,4,17)覆盖了T_1F_2，而(0,1,14)覆盖了F_1T_2。判定所有的条件都已经被覆盖，但是判定的结果都是False，该判定在两个测试用例下都执行了判定结果为False的输出分支，如图30中的虚线所示，即(1)-(2)-(5)-(6)。显然这两个测试并没有使得判定为True的分支被覆盖到，不能满足判定/分支覆盖准则。由此可以知道，满足了条件覆盖准则并不能保证满足判定/分支覆盖准则，两者之间不具备包含关系。如有语句处于未被覆盖的分支上，那么该语句也不能够被执行，在图30中x=x/a这条语句没有被测试所覆盖到，事实上在a=0时恰恰在该语句存在被零除的错误，但是该错误未能被发现。

7. 条件覆盖

- 在利用条件覆盖测试逻辑或所构成的判定也有类似的情况。程序33给出了一个简单的仅仅包含了逻辑或运算判定的示例程序，该程序包含了两路的分支，程序判定点由两个条件通过逻辑运算或连接而形成。

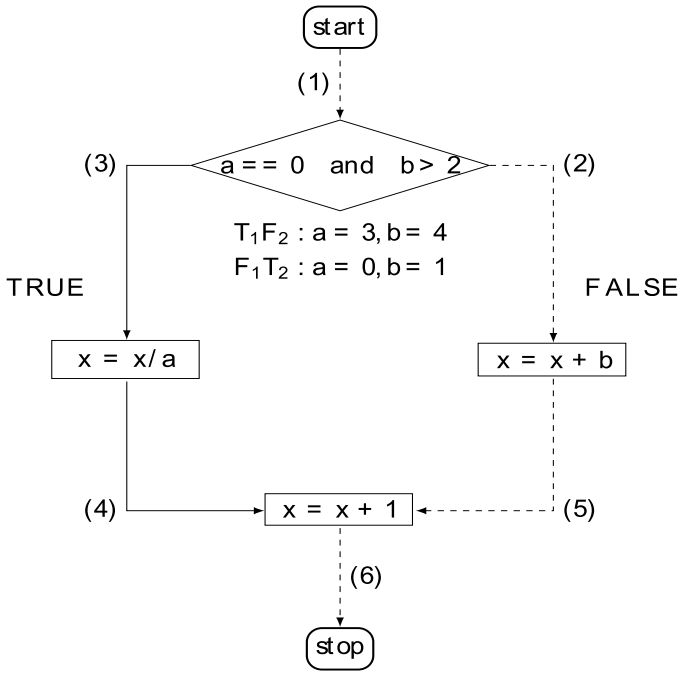


图30 逻辑与运算下条件和分支的差异

程序33 包含简单逻辑或运算的程序

```
#import math
#E:\2014\bookpub\software_testing\python\pytestdemo7\app\condec2.p
ydef condec(a,b):
        x=12;
        if (a>2 or b!=0):
                x=x+a;
        else:
                x=x/b;
        x=x+1;
        return x;
```

7. 条件覆盖

- 和前面类似，判定点中的两个条件分别存在结果为True和False两种情况，只要构建的测试用例集分别使得两个条件分别取得结果为True和False，就满足条件覆盖准则。程序34包含的测试数据就可以满足上述条件。

程序34 带有逻辑或判定中满足条件覆盖的测试

```
#test_switch1.py
import  pytest
from app.condec import condec;

@pytest.mark.parametrize("a,b,r", [
        (3,0,16),
        (1,1,14),
])
def test_demo1(a,b,r):
        assert condec(a,b)==r;
```

7. 条件覆盖

- 这个测试程序包含了两个测试用例，分别为(3,0,16)和(1,1,14)。其中(3,0,16)覆盖了T1F2，而(1,1,14)覆盖了F1T2。判定所有的条件都已经被覆盖，但是判定的结果都是True，该判定在两个测试用例下都执行了判定结果为True的输出分支，如图31中的虚线所示，即(1)-(2)-(4)-(6)。显然这两个测试并没有使得判定为False的分支被覆盖到，不能满足判定/分支覆盖准则。由此可以知道，满足了条件覆盖准则并不能保证满足判定/分支覆盖准则，两者之间不具备包含关系。如有语句处于未被覆盖的分支上，那么该语句也不能够被执行。例如，语句x=x/b没有被覆盖到。实际上，在b=0时恰恰在该语句存在被零除的错误，但是该错误未能被发现。

7. 条件覆盖

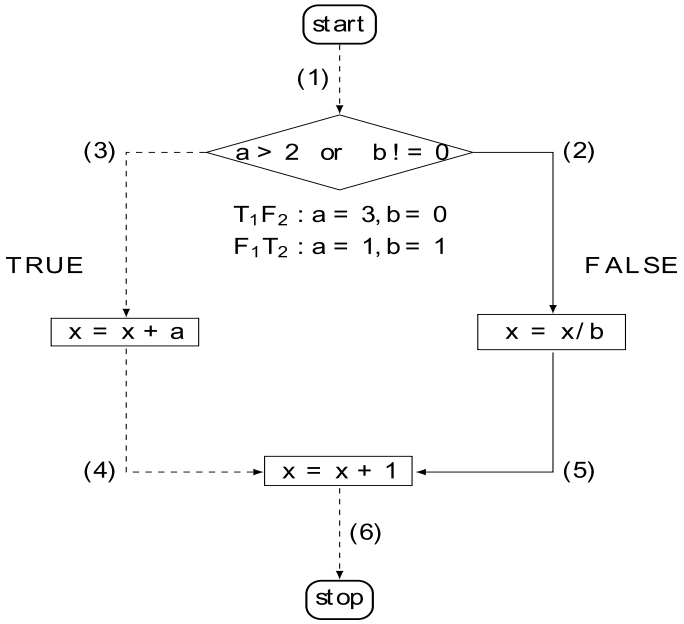


图31逻辑与运算下条件和分支的差异

- 无论是逻辑与还是逻辑或所构成的判定，都存在满足条件但是不满足判定覆盖的情况，同样处于未被覆盖分支上的语句也无法被覆盖到。因此，条件覆盖准则也并不能保证语句覆盖。因此，提出了条件判定覆盖(Condition Decision Coverage)：
  - 设计足够的测试用例，使得判定中每个条件的所有可能(真/假)至少出现一次，并且每个判定本身的判定结果(真/假)也至少出现一次。

7. 条件覆盖

- 对于程序31，测试程序32给出测试用例集，如表10第1行和第2行所示，测试用例1和测试用例2仅仅覆盖了判定结果为FALSE的分支，所以在此基础上，必须补充其判定结果为True的测试数据。对于变量a只能取值0，而变量b取值任意大于2的数值都可以，可以根据启发式信息做出抉择，这里将其取值为4，其产生的判定结果为True。显然无论在条件层次上还是在判定层次上均满足覆盖准则。

表10 程序31中补充用例满足条件判定覆盖

编号	测试输入	覆盖的条件(逻辑与)		覆盖的判定
		a==0	b>2	
1	(3,4)	False	True	False
2	(0,1)	True	False	False
3	(0,4)	True	True	True

7. 条件覆盖

- 对于程序33，测试程序34给出测试用例集，如表10第1行和第2行所示，测试用例1和测试用例2仅仅覆盖了判定结果为True的分支，所以在此基础上，必须补充其判定结果为False的测试数据。对于变量a任意小于2的数值都可以，可以根据启发式信息做出抉择，这里将其取值为1，变量b取0，其产生的判定结果为True。显然无论在条件层次上还是在判定层次上均满足覆盖准则。

表11 程序31中补充用例满足条件判定覆盖

编号	测试输入	覆盖的条件(逻辑或)		覆盖的判定
		a>2	b!=0	
1	(3, 0)	True	False	True
2	(1, 1)	False	True	True
3	(1,0)	False	False	False

7. 条件覆盖

- 而实际上只由一个运算符构成的判定，无论与运算还是或运算，一个简单的构造满足条件判定准则的输入的方法是让两个条件的结果同时为True或者同时为False，其判定也必定包含有True和False的结果。此时满足条件判定准则。当有多个条件构成，可以采用递归方法做进一步的分解，确定每一个输入参数的取值。
- 条件判定覆盖可以发现但是无法保证发现所有的缺陷：
  - （1）逻辑变量错误。
  - （2）逻辑运算括弧错误。
  - （3）关系操作符错误。
  - （4）算术表达式错误。
  - （5）遗漏逻辑操作符。
  - （6）多余逻辑操作符。
  - （7）不正确逻辑操作符。

7. 条件覆盖

- 7.3 条件组合覆盖
  - 在条件组合中，将所有逻辑变量的取值为True和False两种情况全部组合一次。显然若一个判定具有n个独立的条件，那么最后的组合数为 2^n 。显然随着n的增大，测试用例数将以指数方式增长。
  - 例如判定R=（a>5 and b<10 or c=20 and d>30），abd的取值在原则上由其他启发式信息决定，例如边界值、等价类等。假设变量a小于5时取值4，大于5时取值6。b小于10时，取值9，大于10时取值11。c不等于20取值15，当d小于30时取值29，大于30时取值31。产生的测试用例，如表12所示。

7. 条件覆盖

- 在上述讨论过程中，4个条件是完全独立的，一个条件的取值结果不会对另一个条件结果产生任何的影响。如果不同条件之间的关系不是完全的独立的，那么在组合过程中有些用例就不可能出现。例如判定R=a<5 and a<10，and运算法前后两个条件并不完全独立。当a<5结果为True时，a>10的条件只有一种取值False。
- 组合条件优点是：测试全面。
- 其缺点是：测试用例数量大。当出现问题时定位比较困难。

表12 条件组合覆盖示例

编号	测试用例				覆盖条件
	a	b	c	d	
1	4	9	20	31	T_1 T_2 T_3 T_4
2	4	9	20	29	T_1 T_2 T_3 F_4
3	4	9	15	31	T_1 T_2 F_3 T_4
4	4	9	15	29	T_1 T_2 F_3 F_4
5	4	11	20	31	T_1 F_2 T_3 T_4
6	4	11	20	29	T_1 F_2 T_3 F_4
7	4	11	15	31	T_1 F_2 F_3 T_4
8	4	11	15	29	T_1 F_2 F_3 F_4
9	6	9	20	31	F_1 T_2 T_3 T_4
10	6	9	20	29	F_1 T_2 T_3 F_4
11	6	9	15	31	F_1 T_2 F_3 T_4
12	6	9	15	29	F_1 T_2 F_3 F_4
13	6	11	20	31	F_1 F_2 T_3 T_4
14	6	11	20	29	F_1 F_2 T_3 F_4
15	6	11	15	31	F_1 F_2 F_3 T_4
16	6	11	15	29	F_1 F_2 F_3 F_4

8. 修正条件判定覆盖

- 8.1 修正条件判定覆盖的定义
  - Python语言中，由于存在短路计算现象，导致很多缺陷无法发现。
    - 这个代码段执行的结果为：
      >>>
      this is A!
      OK!
      >>>
      该程序段中的判定：
      a() or b() and not c()
      实际上等效于：
      a() or (b() and not c())

程序35 一个具有短路效应的判定

```
def a():
    print 'this is A!'
    return 1
def b():
    print 'this is B!'
    return 1
def c():
    print 'this is C!'
    return 1
if a() or b() and not c():
    print 'OK!'
```

  - 为了描述判定的输入和输出的关系，特别是为了避免产生计算短路现象，业界提出了修正的条件判定覆盖。这个度量最早是波音公司创建的，并被用于航空软件标准RCTA/DO-178B中。航空电子标准RTCA/DO-178B的A级认证要求程序的每一行代码都要进行MC/DC覆盖测试。

8. 修正条件判定覆盖

- 修正条件判定覆盖MC/DC（Modeified Condition/Decision Coverage）方法要求程序中的每一个条件必须产生所有可能的输出结果至少一次，并且每一个判定中的每一个条件必须能够独立影响一个判定的输出，即在其他条件不变的前提下仅改变这个条件的值，而使判定结果改变。修正的条件判定覆盖，包含三个含义：
  - 1）判定的每个条件所有取值至少出现一次。
  - 2）每个判定的所有可能结果至少出现一次。
  - 3）每个条件都能独立地影响判定的结果。
- MC/DC发现的主要软件问题包括：
  - （1）ORF：Operator Reference Faults，例如“与”被误写成“或”。
  - （2）VNF：Variable Negation Faults，一个变量被误写成了它的否定。
  - （3）ENF：Expression Negation Faults，一个表达式被误写成了它的否定。

8. 修正条件判定覆盖

- 一般而言，一个判定是由逻辑运算符（主要是与运算、或运算）连接而构成。为了分析复杂的判定的MC/DC的设计方法，先观察两个最简单的运算规律。以 R = A and B 为例进行讨论。列出条件A和条件B的取值，如表13所示。

表13 A and B 的MC/DC覆盖

测试用例编号	A	B	R=A and B
1	True	True	True
2	False	True	False
3	True	False	False

- 在测试用例1和测试用例2中，在B为True的前提下，结果R的值随着A的变化而变化，即A独立地影响判定结果。而在测试用例1和测试用例3中，A为True，R的值随着B的变化而变化，即B独立地影响了结果R。显然表13满足MC/DC的覆盖要求。

8. 修正条件判定覆盖

- 同理，可以列出R=A or B满足MC/DC覆盖的测试用例，如表14所示。

表14 A or B的MC/DC覆盖

测试用例编号	A	B	R=A and B
1	False	False	False
2	False	True	True
3	True	False	True

- 在测试用例1和测试用例2中，在A为False的前提下，结果R的值随着B的变化而变化，即B独立地影响判定结果。而在测试用例1和测试用例3中，B为False，R的值随着A的变化而变化，即A独立地影响了结果R。显然表14满足MC/DC的覆盖要求。

8. 修正条件判定覆盖

- 若一个判定具有n（n>=2）个逻辑变量（条件），则满足MC/DC覆盖要求的测试用例至少需要n+1个测试用例。
- 证明：利用归纳法。
- 当n=2时，根据表13和表14，显然至少3个测试用例才能满足MC/DC覆盖条件。
- 假设当n=j时，至少需要j+1个测试用例，当n=k时，至少需要k+1个测试用例。
- 现在证明当n=j+k时，至少需要j+k+1测试用例才能满足MC/DC覆盖。
- 先假设通过and 运算符连接两个子判定：

$$R = D_1 \quad \text{and} \quad D_2$$

- 其中D1具有j个原子逻辑变量（条件），D2具有k个原子逻辑变量（条件）。其中D1中满足MC/DC的测试用例集表示为{S11,S12,…,S1j+1}，而D2中满足MC/DC的测试用例集表示成为{S21,S22,…,S2k+1}。

8. 修正条件判定覆盖

- 显然在{S11,S12,…,S1j+1}中至少包含了一个使得D1为True的测试用例，不妨假设为S11。同样{S21,S22,…,S2k+1}至少包含了一个使得D2为True的测试用例，不妨假设为S21。现在，利用{S11,S12,…,S1j+1}和{S21,S22,…,S2k+1}构建是的R满足MC/DC的的测试用例集。
- 为了使得D1的变量能够独立影响结果R，那么D2的结果必须为True。选择D2表示的测试用例为S21，并且将其与{S11,S12,…,S1j+1}组合构成新的测试用例集：
- TS1={S11 S21,S12 S21,…,S1j+1 S21}
- 测试用例集TS1必然满足使得D1的每一个条件独立影响R的结果。
- 同理，为了使得D2中原有逻辑变量能够独立影响结果R，那么D1的结果必须为True，选择D1的测试用例S11，并且将其与{S21,S22,…,S2k+1}组合构成新的用例：
- TS2={ S11S21, S11S22,…, S11S2k+1}
- 测试用例集TS2必然满足使得D2的每一个条件独立影响R的结果。

8. 修正条件判定覆盖

- 考虑D1和D2中的所有逻辑变量，并且使其满足MC/DC覆盖的测试用例集：
- TC=TC1∪TC2
- ={S11 S21,S12 S21,…,S1j+1 S21, S11S22,…, S11S2k+1}
- 根据前面假设，TS1和TS1都是最小测试用例集，无法最精简。并且TS1和TS1只有一个共同测试用例S11S21，所以TC的元素最小个数为：
- (j+1)+(k+1)-1=(j+k)+1
- 同理，可以证明当：
- R=D1 or D2
- 其满足MC/DC的最小测试用例个数也是(j+k)+1。
- 证毕。
- 目前常用的MC/DC的设计方法包括了唯一原因法、屏蔽法、二叉树法等。

8. 修正条件判定覆盖

- 8.2 唯一原因法生成MC/DC测试用例
  - 唯一原因法是一种较早的MC/DC测试用例设计方法，通过条件的真值表比对找出在其他条件不变的前提下影响输出的条件。包括了三个基本步骤：
    - （1）列出条件的全组合。
    - （2）找出独立影响对。
    - （3）找出最小测试用例集。
  - 以R=A and (B or C) 为例说明唯一原因法的原理，第1步列出A、B、C三个条件所有组合并计算输出值，如表15 中的左边4列所示。

表15 A and (B or C)输入和输出的关系

测试用例编号	A	B	C	R	独立影响		
					A	B	C
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

8. 修正条件判定覆盖

- 在表15 中，比对不同的行，找出独立影响输出的测试用例。例如在测试用例1和测试用例5中，输入分别为TTT和FTT，输出为T和F，如表中阴影部分所示。显然在B和C固定为TT的前提下，A独立影响了输出结果R。在表格的右边的第1行A所对应的位置填上5，而在第5行A所对应的位置填上1。其含义是测试用例1和测试用例5可以使得条件A独立影响判定输出R。同理，测试用例2和测试用例4，在A和C固定为TF的前提下，可以使得条件B独立地影响判定输出R。以此类推，完成右边的表格。
- 在上述的基础上，将每一个条件能够独立影响的输出的测试用例对重新整理，形成表 16 所示。

表16 A and (B or C)独立影响判定的条件测试用例对

条件	测试用例对1	测试用例对2	测试用例对3
A	5, 1	6, 2	3, 7
B	2, 4	--	--
C	3, 4	--	--

8. 修正条件判定覆盖

- 根据表16找出最小的测试用例集。对于条件B和条件C都只有一个测试用例对能够满足独立影响条件。所以这两个测试用例对都必须作为测试用例。由于在这两个用例对中间，测试用例4是重复的，所有只要使用1次即可。可以将测试用例TS={2,3,4}添加到最后测试集中。条件A可以在三个候选测试用例中选择。测试用例对2和测试用例3和已经选择测测试用例TS={2,3,4}都有一个用例时重复的，而测试用例对1和已经选择TS没有任何的重复。所以在测试用例对2或者测试用例对3中选择。若选择的是测试用例对2，那么形成的测试用例集TS={2,3,4,6}，如表17所示。

表17 A and (B or C) 满足MC/DC的测试用例

测试用例编号	A	B	C	R
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
6	F	T	F	F

8. 修正条件判定覆盖

- 8.3 屏蔽法生成MC/DC测试用例
  - 大部分的多目条件判定都认为是由简单的逻辑与或者逻辑或连接而形成，非运算是单目运算。根据表13和表14的分析，先隐藏一个运算对结果的影响。对于R= A and (B or C)。为了确认A能够对于R存在对立影响，那么B or C的结果必须为True。此时条件B和条件C有一个为True即可。选择B=True，C=False，那么可以形成两个测试用例，如表18所示。

表18 A and (B or C) 中A对于R的独立影响

测试用例编号	A	B	C	R
1	T	T	F	T
2	F	T	F	F

8. 修正条件判定覆盖

- 同样，为了讨论条件B对结果的影响，A只能选择True，而C只能选择False。形成两个测试用例，如表19所示。

表19 A and (B or C) 中B对于R的独立影响

测试用例编号	A	B	C	R
3	T	T	F	T
4	T	F	F	F

表20 A and (B or C) 中C对于R的独立影响

测试用例编号	A	B	C	R
5	T	F	T	T
6	T	F	F	F

- 同样，为了讨论条件C对结果的影响，A只能选择True，而B只能选择False。形成两个测试用例，如表20所示。
- 在上述测试用1到测试用例6中，测试用例1和测试用例3是重复的，测试用例4和测试用例6是重复的，重新整理得到最后的测试用例集如表21所示。

表21 完成的测试用例

测试用例编号	A	B	C	R
1	T	T	F	T
2	F	T	F	F
3	T	F	F	F
4	T	F	T	T

8. 修正条件判定覆盖

- 8.4 二叉树法生成MC/DC测试用例
  - 一个判定表达式包含有and、or、（、）四个字符。在没有括号的情况下，and的优先级高于or，相邻的两个相同的运算符优先级从左到右。如果有括号，那么括号内运算优先。构建判定二叉树的方法，和执行中缀表达式的算法类似。构建两个堆栈，分别为运算符堆栈和逻辑变量堆栈。从左到右扫描判定表达式，如果是变量表达式则压入变量堆栈，若为运算符，则将其和运算符堆栈的栈顶运算符比较，若栈顶运算符的优先级高，则将栈顶的运算符弹出，从变量堆栈弹出两个变量，分别作为该运算符的左右孩子，加入变量栈。若为左括号则将其入栈，若是右括号，则不断执行动作（从运算符堆栈中弹出运算符，同时从变量运算符弹出两个变量，分别作为该运算符的左右孩子，加入到变量栈），一直到遇到左括号。若扫描完毕，运算符堆栈不为空，值持续执行动作（从运算符堆栈中弹出运算符，同时从变量运算符弹出两个变量，分别作为该运算符的左右孩子，加入到变量栈）。若运算符堆栈为空，则其变量栈中保存就是判定二叉树。

8. 修正条件判定覆盖

- 判定R=(A and B) and (C or (D and E)) 对应的判定二叉树如图32所示。

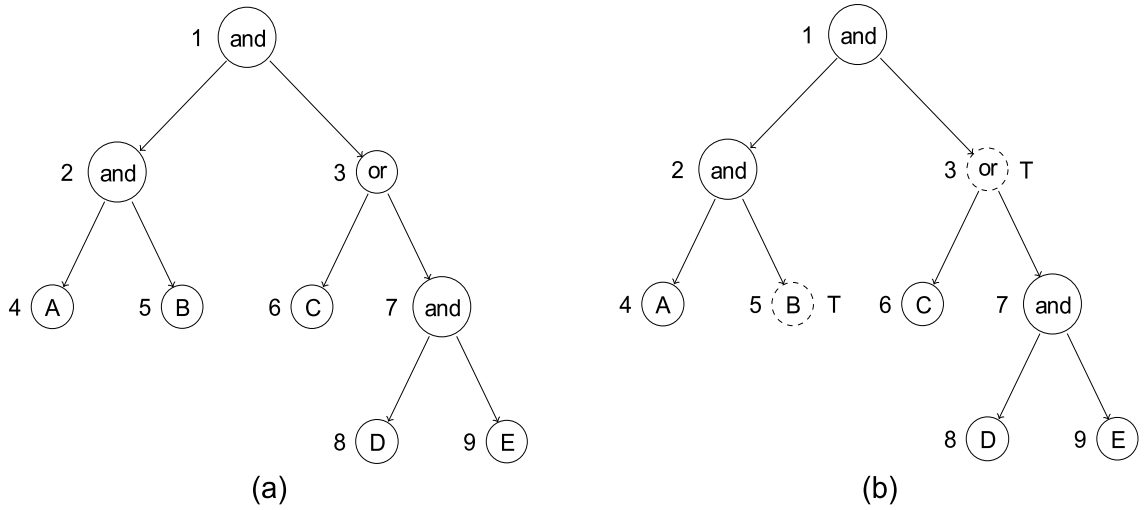


图32 判定二叉树及其向上标注

- 现规定根节点的层数为1，根节点的左右孩子的层数为2。对于一个逻辑变量，变量到根节点所经过节点构成的路径，称为变量的影响路径。
- 在图32（a）中的判定二叉树，变量A的影响路径由节点4,2,1所构成。而变量D的影响路径由节点8,7,3,1所构成。

8. 修正条件判定覆盖

- 二叉判定树的方法构建满足MC/DC覆盖的测试用例集，选定独立影响判定结果的变量，除了该变量到树根节点所经过的独立影响路径外，通过两轮标记确定各个逻辑变量的值。对于一个逻辑变量，具体包括以下几个步骤：
  - （1）选定变量。选定独立影响判定结果的变量，也就是判定二叉树的一个叶子节点。
  - （2）向上标注。从该节点开始向上访问每一个节点，根据其父节点的运算符确定其兄弟节点的逻辑值。若父节点为and运算符，则其兄弟节点标记为True。若其父节点or 运算符，则其兄弟节点标记为False运算符。如此反复，一直到父节点为根节点。
  - （3）向下标注。在判定树的第2层，从不在独立影响路径的节点开始，向下访问一个节点，标记节点的逻辑值。若该节点的标记值为True，而节点的运算符为and，则其两个孩子节点均标记为True。若该节点标记值为False，而节点运算符为and，则两个孩子节点任意选择一个为False，另外一个值可以为True或者False。若该节点的标记值为False，而节点的运算符为or，则其两个孩子节点均标记为False。若该节点的标记值为True，而节点的运算符为or，则其一个孩子标记为True，另外一个孩子的值可以为True或者False。
  - （4）构造用例。对于选定的变量的两个取值True和False，和其他所有叶子节点通过上述2）和3）步骤确定的值构成两个测试用例。

8. 修正条件判定覆盖

- 对于每一个逻辑变量，都采用上述方法确定其测试用例集。
- 以其中的逻辑变量A为例，来阐述上述的过程。
- 向上标注。变量A所在的节点编号为4，其父节点为节点2，包含了运算符and，其兄弟节点为节点5。根据第2条规则，将B的值设为True。接下来，考虑第2层的节点，节点2位于影响路径上，其父节点的内容为and，所以可以将其兄弟节点设置为True。由于节点2的父节点为根节点，自此，完成向上的标记过程，如图32（b）中的的虚框所示。
- 向下标注。从第2层开始，搜索影响路径以外的所有没有标注的节点。节点2处于影响路径上，而节点的两个孩子分别为节点6和结点7，其由于节点自身的运算符为or，而节点标注的结果为True，所以对于节点6和结点7至少需要一个标注为True，现在选择将两个节点均标注为True，如图33(a)所示。接下来看，节点7的左右孩子都没有标注，其本身已经标注为True，所以其左右孩子均标准为True。在这个过程中，标注的结果并不唯一。例如节点3已经被标注为T，而其运算符内容为or，左右两个孩子如果节点标注为True，而节点7标注为False，就会得到另一测试用例集。

8. 修正条件判定覆盖

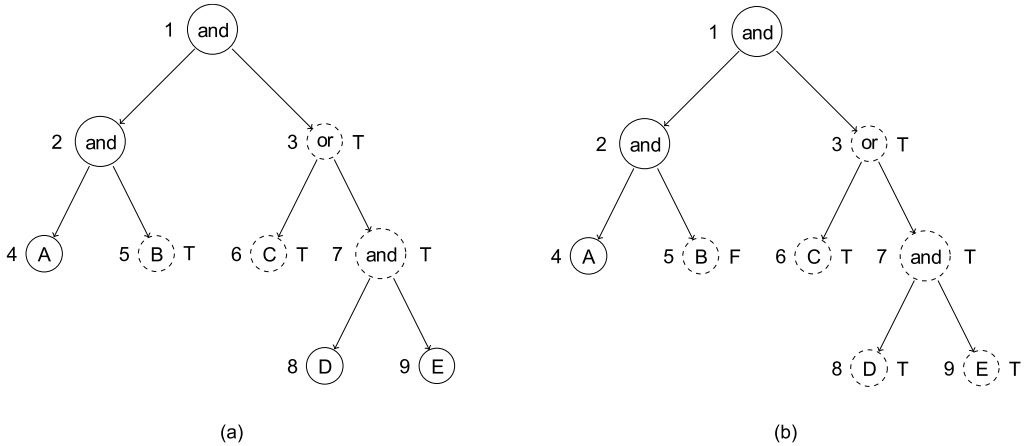


图33 二叉判定树的向下标注

- 通过向上标注和向下标注，完成所有除影响路径以外的所有节点的标注。将完成标注的所有叶子节点组合，形成逻辑变量A的独立影响判定R=(A and B) and (C or (D and E))的测试用例集，如表22所示。

表22 二叉树法形成对变量A的独立影响测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	F	T	T	T	T	F

8. 修正条件判定覆盖

- 同理可以确定B、C、D、E变量的独立影响测试用例集，如表23所示，其中带底纹的格子表示独立影响判定结果。

表23 二叉树法形成对变量B、C、D、E的独立影响测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	T	F	T	T	T	F
3	T	T	T	F	T	T
4	T	T	F	F	T	F
5	T	T	F	T	T	T
6	T	T	F	F	T	F
7	T	T	F	T	T	T
8	T	T	F	T	F	F

表24 二叉树法构造的满足MC/DC覆盖测测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	F	T	T	T	T	F
3	T	F	T	T	T	F
4	T	T	T	F	T	T
5	T	T	F	F	T	F
6	T	T	F	T	T	T
7	T	T	F	T	F	F

- 在利用二叉树法产生的这些测试用例集中，中间存在重复的测试用例，例如表22中的测试用例1和表23中的测试用例1、表23中的测试用例4和测试用例6、测试用例5和测试用例7，删除这些重复的测试用例，使其仅保留1个非重复的测试用例，最后形成完整的满足MC/DC覆盖的测试用例集，如表24所示。

8. 修正条件判定覆盖

- 8.5 MC/DC的进一步讨论
  - 逻辑变量的独立性问题
    - 一个逻辑变量值和另外一个变量没有任何的约束关系。但在实际应用中，不同逻辑变量之间的取值存在一定的约束关系。例如程序36中的判定：A and B 并不独立。当A为True时，B的值必定为False，而当A为False时，B的值可能会True和False两种取值。A and B的值相对C是独立的，可做一个变量替换A1=A and B，那么判定可以看成为A1 or C，这时A1和C是完全独立的。

程序36 判定之间独立性示例

```
def fun(a,b):
        A=5<a
        B=a<10
        C=b>10
        if (A and B) or C:
                print "True"
        else:
                print "False"
```

8. 修正条件判定覆盖

- 8.5 MC/DC的进一步讨论
  - if 语句构成的多路分支和多条件判定的转换
    - 由多个if语句构成，若每一个语句都只在同一个结果（True或者False）分支中，那么其等效于一个多条件判定。在图34中，左边是有if语句构成多路分支伪代码，其对应的判定如右边所示。在这种情况下，右边MC/DC覆盖和左边的分支覆盖是等效的。

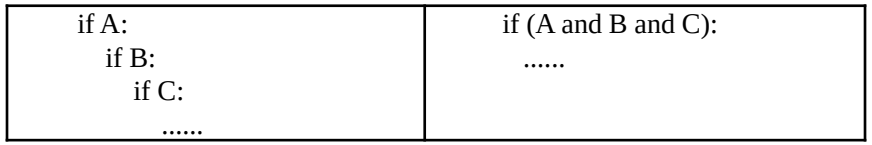


图34 多路分支和多条件判定

8. 修正条件判定覆盖

- 8.5 MC/DC的进一步讨论
  - 间接的条件嵌套问题
    - 在实际的代码中，可能存将一个复杂的逻辑表达式赋值给一个逻辑变量，然后再将该变量作为判定中的逻辑变量。如果仅仅考虑判定中的变量的直接MC/DC覆盖，将会遗漏一些发现缺陷的机会。
    - 若有如下伪代码段：

```
A=A1 and B1
B=C1 or  D1
if (A and B):
    ……
```

    - 对于判定R=A and B，若仅考虑本判定自身。仅需要三个用例即可，如表25中的A列、B列、R列所示。但是对于A或者B的产生，并没有要求满足MC/DC覆盖。例如当A为F时，可能是由A1和B1的三种情况所构成，而B为True也可能有两种情况构成。在这几种情况下只要选取一种情况即可。当A1B1C1D1取值分别为{TTTF、FTTT、TTFF}构成的测试用例集就满足了R=A and B的MC/DC的覆盖要求。

8. 修正条件判定覆盖

- 8.5 MC/DC的进一步讨论

表25 条件嵌套

编号	A	B	A	C	D1	B	R
1	T	T	T	T	F	T	T
				F	T		
2	F	T	F	T	T	T	F
	T	F		F	T		
	F	F					
3	T	T	T	F	F	F	F

- 实际上，这段伪代码的实际效果等效于：
- R=（A1 and B1）and (C1 or D1)
- 对于具有4个逻辑变量的判定表达式，若为了满足MC/DC覆盖最少的测试用例数为5个。显然当A1B1C1D1取值分别为{TTTF、FTTT、TTFF}无法满足MC/DC的覆盖要求。为了避免造成的缺陷，在设计测试用例时，将A和B的表达式替换到R的表达式中，然后根据MC/DC准则进行测试用例的设计。

9. 路径覆盖

- 9.1 程序和控制流图表示
  - 在控制流图中，有两个重要的组件：节点和有向弧（边）。
  - 节点：用○表示控制流图的一个节点，它表示一个或者多个无分支的语句。为了讨论的方便，以及不同节点之间所构成路径的差异，在○标记不同数字。
  - 边（弧）：用有方向的弧线表示节点之间执行的逻辑次序，称为有向边（弧）。
  - 一个控制流图必定具有如下特征：
    - （1）具有一条入口边，该边没有起始节点，该边指向的节点称为入口节点。
    - （2）具有一条出口边，该边没有终止节点，该边起始的结点称为出口节点。
    - （3）所有的节点都至少有一条入边、至少有一条出边。
    - （4）图中的任意一个节点，都在一条从入口边到出口边的路径上。
    - （5）图中的任意一条边，都在一条从入口边到出口边的路径上。

9. 路径覆盖

- 路径是由节点构成的有序序列，该序列的初始节点为入口节点，而终止节点为出口节点。并且序列中两个相邻节点之间必定存在一个有向边。
- 和程序流程图不同，控制流图仅仅关注逻辑走向，而不关注中间的每一条语句的具体含义。在顺序结构中，尽管可能包含了多条语句，但是在控制流图仅仅用一个节点表示，如图35所示。

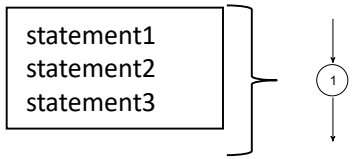


图35 顺序结构的控制流图

9. 路径覆盖

- 图36给出了if以及if-else构成的控制流图示意图。在左边的if分支结构中，如果是仅有if分支，statement1是入口节点，当判定计算结果为True是执行一个语句体statement2（可能是一条语句，也可能是多个顺序执行语句），接着执行statement3。而当判定结果为Flase时，直接跳过语句statement2执行statement3。其可能的执行路径，用控制流图的节点表示为：
  - 路径1：1-2-3-4
  - 路径2：1-2-4
- 在右边的if-else结构中，在decesion1的结果为True或者False时，分别执行语句体statement3和语句体statement4。其可能的执行路径，用控制流图的节点表示为：
  - 路径1:1-2-3-5
  - 路径2:1-2-4-5
  - 除了2路分支以外，有if-elseif构成多路分支，其情况类似，不在重复。

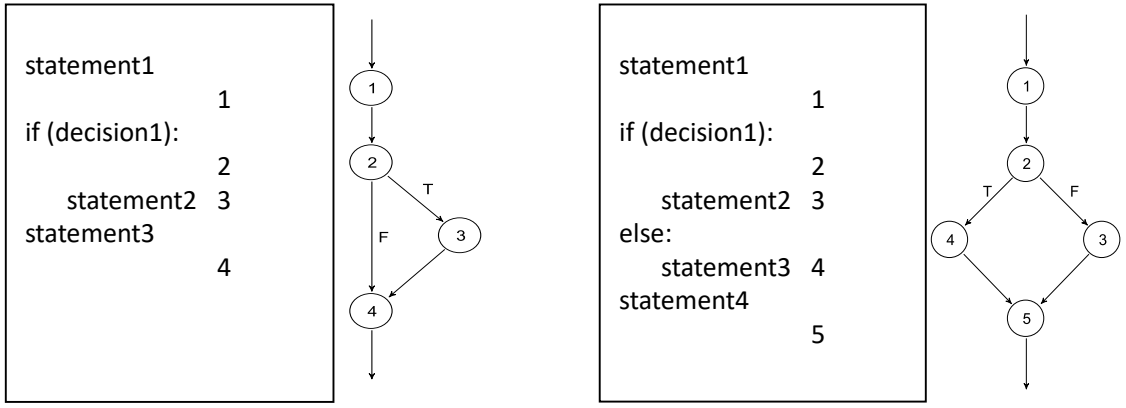


图36 if 和ifelse 构成的控制流图

9. 路径覆盖

- 除了if语句构成的路径以外，循环语句也是复杂控制流的重要组成要素。在Python的循环有两种形式：while循环和For循环。在Python没有repeat语句，实际上repeat语句是直接用while语句实现。在while循环中，存在显式的循环变量的变更，图37右边的incstatement语句用于变更循环变量。在for循环中，循环控制通过访问可迭代对象来实现的，并没有独立的循环变量变更语句。一个简单的例子求1-10之间的累加，利用while和for 两种不同的实现方式。在程序37中右边的是用while语句实现，显示地定义了循环变量i，并且在循环体中，显式通过i+=1实现循环变量的变化。而右边循环变量的定义以及改变均包含在for语句内部。

程序37 while和 for实现的累加运算的等效性

```
def sum1():
    sum1=0
    i=1
    while(i<11):
        sum1=sum1+i
        i+=1
print sum1
```

```
def sum():
    sum=0
    for i in range(1,11):
        sum=sum+i
print sum
```

9. 路径覆盖

- 为了讨论的方便，增加逻辑的循环变量语句虚拟节点，如图37中间的伪代码所示。在考虑了虚拟节点以后，while循环变量和for循环的控制流程图就一致了，如图37右边所示。在这个控制流图中，入口节点到出口节点之间存在很多条路径：
  - 路径1:1-2-5
  - 路径2:1-2-3-4-2-5
  - 路径3:1-2-3-4-2-3-4-5
  - 路径4: 1-2-3-4-2-3-4-2-3-4-5
  - ……

```
statement1
                    1
while(condition1):

                    2
    statement2      3
    incstatement    4
statement3          5
```

```
statement1
1
for iterating_var in sequence:
                    2
    statement2          3
                        4
statement3
5
```

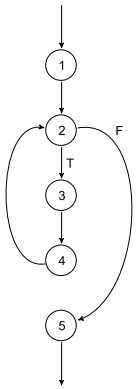


图37 while循环和for循环构成的控制流图

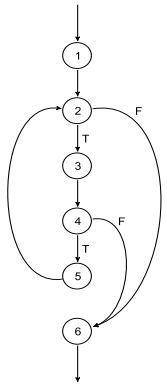
9. 路径覆盖

- 其路径的条数是由循环变量所控制。
- 在Python循环体中，有两个重要影响程序控制流走向的语句，分别为break和continue语句。break语句用来终止循环语句，即循环条件没有False条件或者序列还没被完全递归完，也会停止执行循环语句。在while和for循环中，都可以使用break语句。在使用嵌套循环，break语句将停止执行最深层的循环，并开始执行下一行代码。continue 语句跳出本次循环，而break跳出整个循环。continue 语句用来告诉Python跳过当前循环的剩余语句，然后继续进行下一轮循环。同样在while和for循环中，都可以使用continue语句。通常情况下，break 语句和continue语句都结合一个判定使用。在图38给出了带有break和continue语句的控制流图。由于break和continue语句本身只控制流程方向，语句自身不包含任何的业务逻辑，break和continue相当于控制流图的边，而不是节点，在控制流图的语句标注中，不将他们单独标注。另外在包含continue语句的循环中，一般要求循环变量的控制语句在continue语句的前面，否则遇到continue语句，后面的循环变量将无法变更，循环将变成死循环。

9. 路径覆盖

- 在带有break语句的程序中，可能存在的路径：
  - 路径1:1-2-6
  - 路径2:1-2-3-4-5-2-6
  - 路径3:1-2-3-4-6
  - 路径4:1-2-3-4-5-2-3-4-5-6
  - 路径5:1-2-3-4-2-3-4-6
  - ……
- 在带有continue语句的程序中，可能存在路径：
  - 路径1:1-2-6
  - 路径2:1-2-3-4-5-2-6
  - 路径3:1-2-3-4-2-6
  - 路径4:1-2-3-4-5-2-3-4-5-2-6
  - ……

```
statement1                    1
while(cond1):                 2
    statement2                3
    if (not cond2):           4
        break
    incstatement              5
statement3          6
```




```
statement1                    1
while(cond1):                 2
    incstatement              3
    if (not cond2):           4
        continue
    statement2                5
statement3          6
```

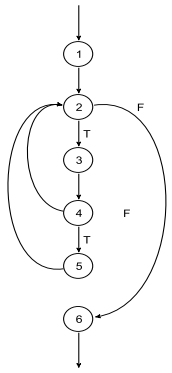


图38 循环中break和continue构成的控制流图

9. 路径覆盖

- 无论是带有break还是continue语句，在循环内部存在的分支将引起路径的指数级增长。
- 在Python中，无论是for循环还是while循环都可以带有else子句。else 中的语句会在循环正常执行完（即不是通过 break 跳出而中断的）的情况下执行。显然，带有else字句的循环体内部，一般都带有break语句，否则else中的语句体变成了必然要执行的语句体，失去了else语句的含义。
- 图39给出了一个带有else的循环结构的控制流图，当循环体正常结束时，流程将转移到结点6，执行statement3语句体，然后执行statement4语句体。当在循环体内部执行到break语句时，循环将终止，流程将跳转到结点7，直接执行语句体statement4。

9. 路径覆盖

```
statement1
                        1
while(cond1):           2
    statement2          3
    if (not cond2):     4
        break
    incstatement        5
else:
  statement3      6
statement4        7
```

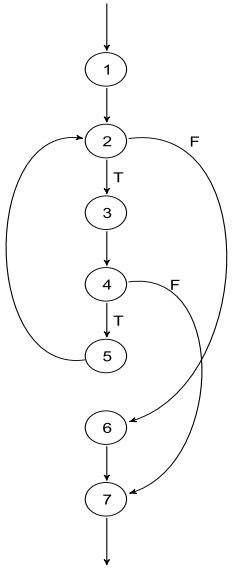


图39 带else的循环结构的控制流图

程序38 带有else的循环例子

```
#
def evenOdd():
    for num in range(100,120):
        for i in range(2,num):
            if num%i ==0:
                j=num/i
                print '%d=%d *%d' % (num,i,j)
                break
        else:
            print num,' is a prime number'

evenOdd()
```

- 程序38给出了一个带有else循环结构的例子，该例子在100到119之间的每一个数字判断其奇偶性，如果是奇数给出提示信息，如果是偶数，则给出其一个因式分解表达式，该表达式中，第1个因子的值为最小。

9. 路径覆盖

- 该程序，由内外两个循环构成，外循环控制每一个被判断的数，内循环判断其奇偶性。执行结果如下：
  - 00=2 *50
  - 101 is a prime number
  - 102=2 *51
  - 103 is a prime number
  - 104=2 *52
  - 105=3 *35
  - 106=2 *53
  - 107 is a prime number
  - 108=2 *54
  - 109 is a prime number
  - 110=2 *55
  - 111=3 *37
  - 112=2 *56
  - 113 is a prime number
  - 114=2 *57
  - 115=5 *23
  - 116=2 *58
  - 117=3 *39
  - 118=2 *59
  - 119=7 *17

9. 路径覆盖

- 9.2 独立路径和圈复杂度
  - 在一个实际的应用程序中，实现路径的全覆盖是不现实的，折中采用独立路径（基本路径）覆盖来替代全路径覆盖。
  - 一条路径是独立路径，那么其应满足：
    - 1）是一条从入口节点到出口节点的路径；
    - 2）该路径至少包含一条其它基本路径没有包含的边。
  - 从上述定义可以看出，独立路径是相对已经选择的路径的集合而言的。对于图38中的路径集合：
    - 路径1:1-2-6
    - 路径2:1-2-3-4-2-6
    - 路径3:1-2-3-4-5-2-6
    - 路径4:1-2-3-4-5-2-3-4-5-2-6

9. 路径覆盖

- 相对于已经选择的路径1而言，路径2是独立路径，因为其包含新的边2-3、3-4、4-2，这些都是路径1所不包含的。若已经选定了路径1和路径2，则路径3也是独立路径，因为其包含了新的节点5，边4-5、5-2都是新增加的边。若选定了路径1、路径2、路径3，则路径4不是独立路径，因为路径4上所有的边都是在选定的路径之中。显然对于独立路径多少是和路径的选择顺序是有关联的。若先选择路径3，那么路径1和路径2都不再是独立路径。
- 一个控制流图中，基本路径的最大数量可以用圈复杂度来表示。圈复杂度由Thomas J.McCabe提出，用于度量程序的复杂性，可以通过控制流图计算出程序的圈复杂度。圈复杂度可以通过以下几个不同的公式进行计算。假设一个控制流程图G，其圈复杂度用V(G)表示，而E表示控制流图的边的数量，N表示控制流图中节点的数量，P为控制流图中连通图的数量。根据前面的分析，控制流图是一个连通图，所以P的值始终等于1。

9. 路径覆盖

- （1）圈复杂度V(G)在数值上等于控制流图有效边(不含入口边和出口边)的数量E减去节点的数量N，加上2倍的连通图个数P，即：V(G)=E-N+2P
- 在图35表示的顺序结构，除了入口边和出口边以外，并无其他边，所以E=0。节点数量N=1，P=1。其复杂度为：V(G)=0-1+2×1=1
- 在顺序结构中无论包含有多少条语句，其复杂度始终为1。
- 在图36表示单纯的if或者if-else结构中，E=4，N=4，P=1，其复杂度为：V(G)=4-4+2×1=2
- 在图37表示的简单循环结构中，E=5，N=5，P=1，其复杂度为：V(G)=5-5+2×1=2
- 图 38给出具有break或者continue语句的循环结构中，E=7，N=6，P=1，其复杂度为：V(G)=7-6+2×1=3
- 图39中，具有else的循环结构中，E=8，N=7，P=1，其复杂度为：V(G)=8-7+2×1=3
- 实际上，从计算结果上，具有break或者continue语句的循环结构和具有else的循环结构在复杂度上是相同的。实际上，else语句的不同，是由循环体内部的if判断中的break所引起的。

9. 路径覆盖

- （2）圈复杂度等于控制流图将平面划分区域的数量 V(G)=R,R为区域数量
- 所谓区域，是有控制流图的边所构成的封闭的一个区域，控制流图的外部也存在认为是一个区域。在顺序结构中，控制流图内部并没有构成任何的区域，仅存在控制外部的一个区域，所以其复杂度V(G)=R=1。图40 给出了基本循环、包含Continue语句循环构成区域示意图。
  - 左边的基本循环结构(a)中，构成两个区域，其中第1个由2-3-4-2节点包围二构成，另一个是在控制流图的外围，故左边的基本循环的圈复杂度为V(G)=R=2。
  - 在右边的带有continue语句的循环结构(b)中，存在三个区域，一个是2-3-4-2节点所包围，第2个是有2-3-4-5-2所包围，第3个是控制流图的外围。故右边边的基本循环的圈复杂度为V(G)=R=3。

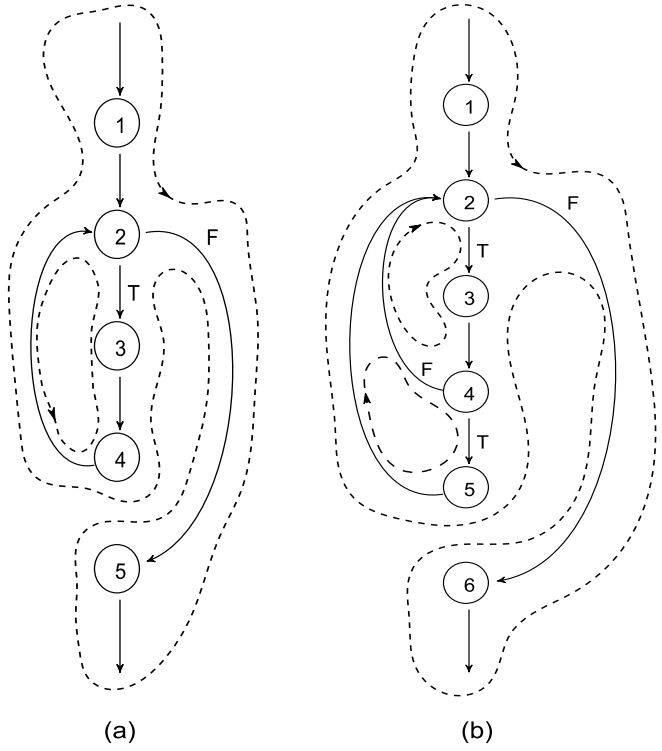


图40 基本循环和包含Continue语句循环的区域

9. 路径覆盖

- 图41给出具有break语句以及具有else语句的循环结构形成的区域，左边为break语句的区域，其三个区域和具有continue语句的结构类似，比较清楚。图 3 39中给出的控制流程图存在交叉线条的情况。连接节点4和结点7之间的连线、连接节点2和结点6的两线，这两条连线似乎交叉。但实际上，可以通过重新绘制连接节点4和结点7之间的联系，避免交叉，如图中(b)所示。

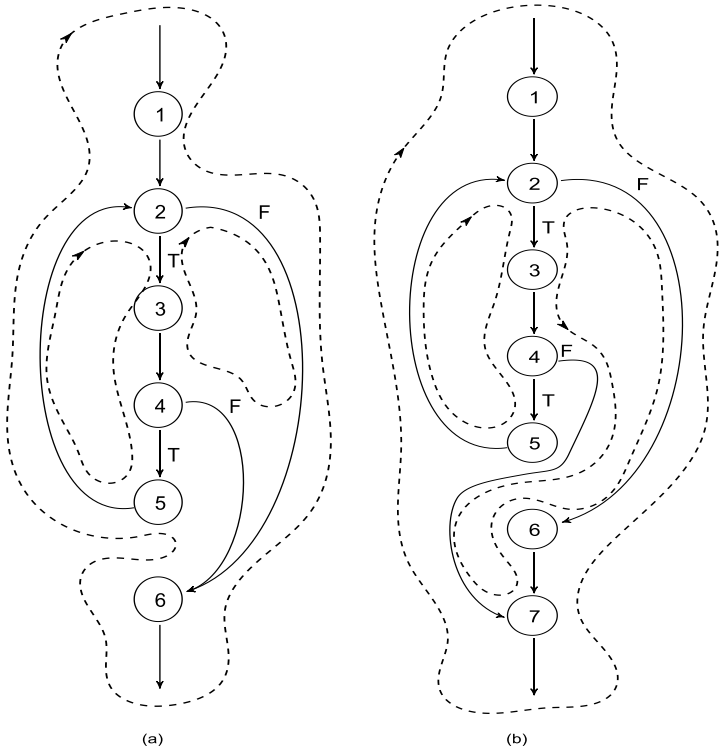


图41 具有break和else结构的循环区域

9. 路径覆盖

- （3）圈复杂度等于控制流图的中判定节点的数量加1
- V(G)=C+1
- 其中：
- V(G)：控制流图的圈复杂度。
- C：判定节点数量，这里的判定节点假定为简单判定，就是不包含有逻辑运算符。
- 显然在如图35所表示的顺序结构中，不存在判定节点，其圈复杂度V(G)=1，和前面的计算结果一致。
- 图36的分支结果，仅仅包含了一个判定节点2，其圈复杂度V(G)=2，和前面的计算结果一致。图37的简单循环中，仅仅包含了一个判定节点2，其圈复杂度V(G)=2，和前面的计算结果一致。图38和图39，中间都存在2个判定节点，其圈复杂度V(G)=2，和前面的计算结果一致。

9. 路径覆盖

- 若一个判定节点存在多于两个的输出分支，此时必须进行分支的拆分，使其等效的控制流图的每一个判定节点都仅包含2个输出分支。图42中，根据前面方法1，可以计算得到：
- V(G)=E-N+2C=12-9+2=5
- 根据区域容易看出，在节点1和结点6之间存在3个区域，在节点6和节点9之间存在一个区域，外部一个区域，显然：
- V(G)=3+1+1=5
- 显然前面两种方法计算得出的结果是一样的。若不进行判定的转换，则仅有节点1和结点6为判定结果，但是判定节点1处在4路输出，将该判定节点转化为等价的2路分支输出，如图中右边所示，则判定节点包含了节点1、节点11、节点111、节点6。因此得到的全复杂度为：
- V(G)=P+1=5

9. 路径覆盖

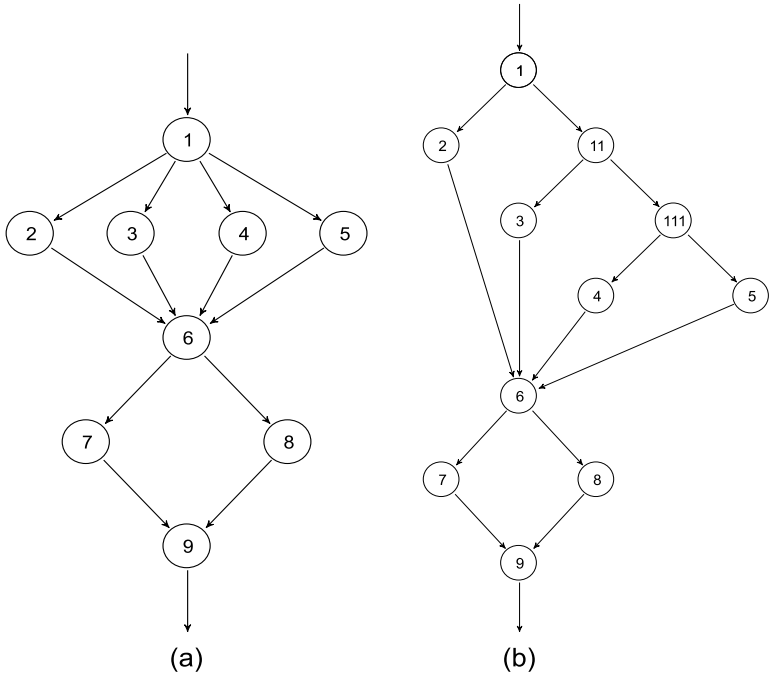


图42 具有多输出分支的判定转换求圈复杂度

9. 路径覆盖

- 另外一种情况，即使输出的路径是只有两个输出，但是包含了组合条件而形成的判定，也需要将判定改写成为等效的简单判定。
- 例如：
  - if　(A>=MIN and A <= MAX):
  - 　　dosomething()
  - else:
  - 　　dosthelse()
  - 这个判定中间由and运算符连接了两个条件，其实其等效于：
  - if (A<MIN):
  - 　　dosthelse()
  - elseif (A>MAX):
  - 　　dosthelse()
  - else：
  - 　　dosomething()

9. 路径覆盖

- 9.3 基本路径覆盖
  - 执行基本路径测试通常需要包括以下4个步骤：
    - （1）依据程序的基本结构，分析其路径个数，画出程序的控制流图。
    - （2）根据控制流图，计算出程序的圈复杂度。
    - （3）根据控制流图，构造程序的独立路径集合，其上限为圈复杂度。
    - （4）根据（3）中的独立路径，设计测试用例的输入数据和预期输出。
  - 有个数据列表，求出其数值在min和max之间的数据平均值。其Python 代码如程序39所示。试用基本路径法设计测试用例。

9. 路径覆盖

程序39 求列表平均值程序

```
#average.py
def average(values,min,max):
    i=0;
    validNum=0;                                          ①
    sum=0;
    while(i<len(values)    and \                         ②
          values[i]!=-999):                              ③
        if(values[i]>=min and \                          ④
           values[i]<=max):                              ⑤
            validNum+=1
            sum+=values[i];                              ⑥
        i=i+1;                                           ⑦
    if (validNum>0):                                     ⑧
        mean=sum/validNum                                ⑨
    else:
        mean=-999                                        ⑩
return mean                                              ⑪
```

9. 路径覆盖

- 1）画出控制流程图
  - 在这个程序中，其主要功能是一个while循环和两个if判定构成。其中while的循环条件和第1个if 语句中判定都是复合判定，需要将这两个条件进行分拆。在代码中，通过换行符\将两个条件写在两行上，分别标注为两个行。else字句始终和if相对应，只表示分支方向，无需单独标注。多个没有分支的初始化语句，标注为1。依据上述编码，画出控制流图，如图43所示。

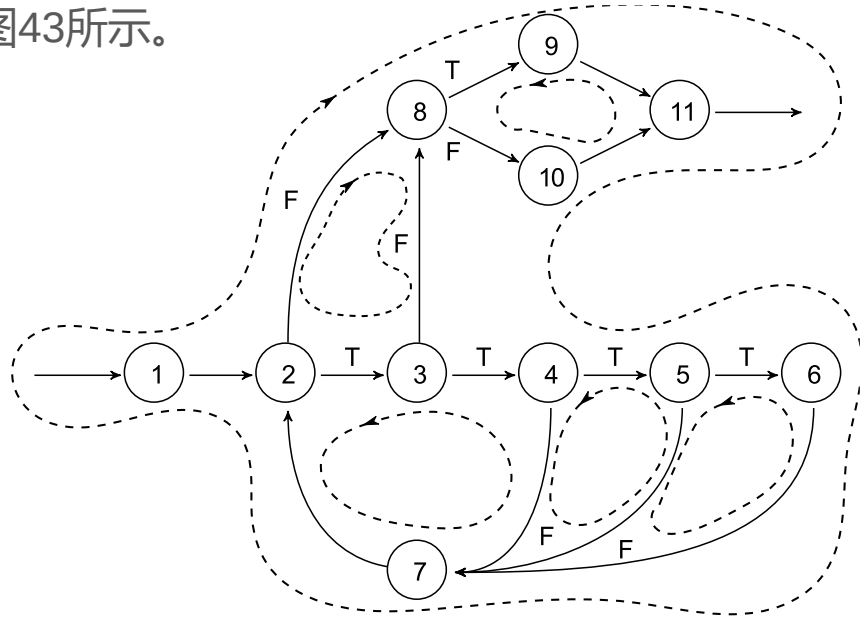


图43 平均值问题的控制流图

9. 路径覆盖

- 2）计算圈复杂度
  - 根据控制流图，很容易计算出其圈复杂度。在图中，形成区域都已经用虚线表示出来了。显然：V(G)=R=6
  - 用判定节点数加以验证，在图中判定节点为节点2、节点3、节点4、节点5、节点8共5个，所以：
  - V(G)=C+1=5=1=6
  - 均值问题的圈复杂度为6。
- 3）查找基本路径序列
  - 查找基本路径的方法，一般从入口边开始一直到出口边。在本题中，查找以节点1为开始、节点11为结束的路径。一般而言，从最短的路径开始，将其加入到基本路径序列中。因为不同次序会影响基本路径序列的构成，所以不用集合表示多个基本路径构成测试集合。
  - 在控制流图中，只是表示可能的流程走向，有些路径是实际上不可行的。

9. 路径覆盖

- 例如路径：
- 1-2-8-9-11
- 当values列表为空时，i<len(values)不成立，节点2的结果为F。此时validNum的值始终为0，节点8中的validNum>0始终为F，流程永远不会达到节点9。所以路径1-2-3-8-9-11这种路径是实际上不可行的，但是简单地从控制流图，无法发现这种不可行的路径。
- 选择第1条最短的独立路径为：
- 1-2-8-10-11
- 在第1条路径中，考虑节点3，新增加2-3以及3-8两条边，形成第2条独立路径：
- 1-2-3-8-10-11
- 在第2条独立路径的基础上，考虑节点4，新增加边：3-4、4-7、7-2，形成第3条独立路径：
- 1-2-3-4-7-2-8-10-11
- 同时，考虑节点5，形成独立路径：
- 1-2-3-4-5-7-2-8-10-11

9. 路径覆盖

- 考虑节点6时，显然validNum>0是始终成立的，节点8的结果永远为True，所以最短的独立路径为：
- 1-2-3-4-5-6-7-2-8-9-11
- 该独立路径仅仅考虑只有一个元素。但在一般情况下，需要考虑多个数据元素。在这个独立路径中，[2-3-4-5-6-7]是可以重复的。
- 另外，在列表中，-999作为计数结束的另一个重要标准，可增加一个以-999作为结束标志的测试用例。
- 自此，控制流图中的所有边都已经被覆盖。综上分析，形成的测试路径为：
- 路径1：1-2-8-10-11
- 路径2：1-2-3-8-10-11
- 路径3：1-2-3-4-7-2-8-10-11
- 路径4：1-2-3-4-5-7-2-8-10-11
- 路径5：1-[2-3-4-5-6-7]-2-8-9-11
- 路径6：1-[2-3-4-5-6-7]-2-3-8-9-11

9. 路径覆盖

- 4）构造测试用例
  - 确定路径1的测试用例，由于在这个路径上i初始值为0，关键所在是节点2，输入的列表values为空列表，此时min和max的值不影响输出结果，期望的输出结果为-999。
  - 路径2是在节点2结果为True的情况，考虑values[i]!=-999不成立，而validNum>0不成立的情况。综合3点判定的情况，要求列表的有且仅有1个元素，该元素的值必须为-999，此时min和max的值不影响输出结果。期望的输出为-999。
  - 路径3，列表仅允许有一个元素，且该元素的值小于最小值min，使得节点8的结果为False，期望的输出为-999。
  - 路径4和路径3类似，列表仅允许有一个元素，且该元素的值大于最大值max，使得节点8的结果为False，期望的输出为-999。
  - 路径5表示非空列表，且列表至少一个值在min和max之间，且不包含有-999。路径6表示列表有多个元素，至少有一个值在min和max之间，且包含有-999。

9. 路径覆盖

- 假定min的值为0，max的值为100，在min和max之间的值为70。最终形成的的测试如程序40所示。

程序40 基本路径测试程序

```
mport  pytest
from app.average import average

@pytest.mark.parametrize("vlist,min,max,result", [
    ([],0,100,-999),
    ([-999],0,100,-999),
    ([-5],0,100,-999),
    ([102],0,100,-999),
    ([75,85,80],0,100,80),
    ([70,90,80,-999,88],0,100,80),
])
def test_average(vlist,min,max,result):
    assert average(vlist,min,max)==result;
```

- 从该例子可以看出，路径测试和前面测试不同，其关注的整体的逻辑关系，特别前后不同判定或者条件之间的相互影响。