



良好的编程实践

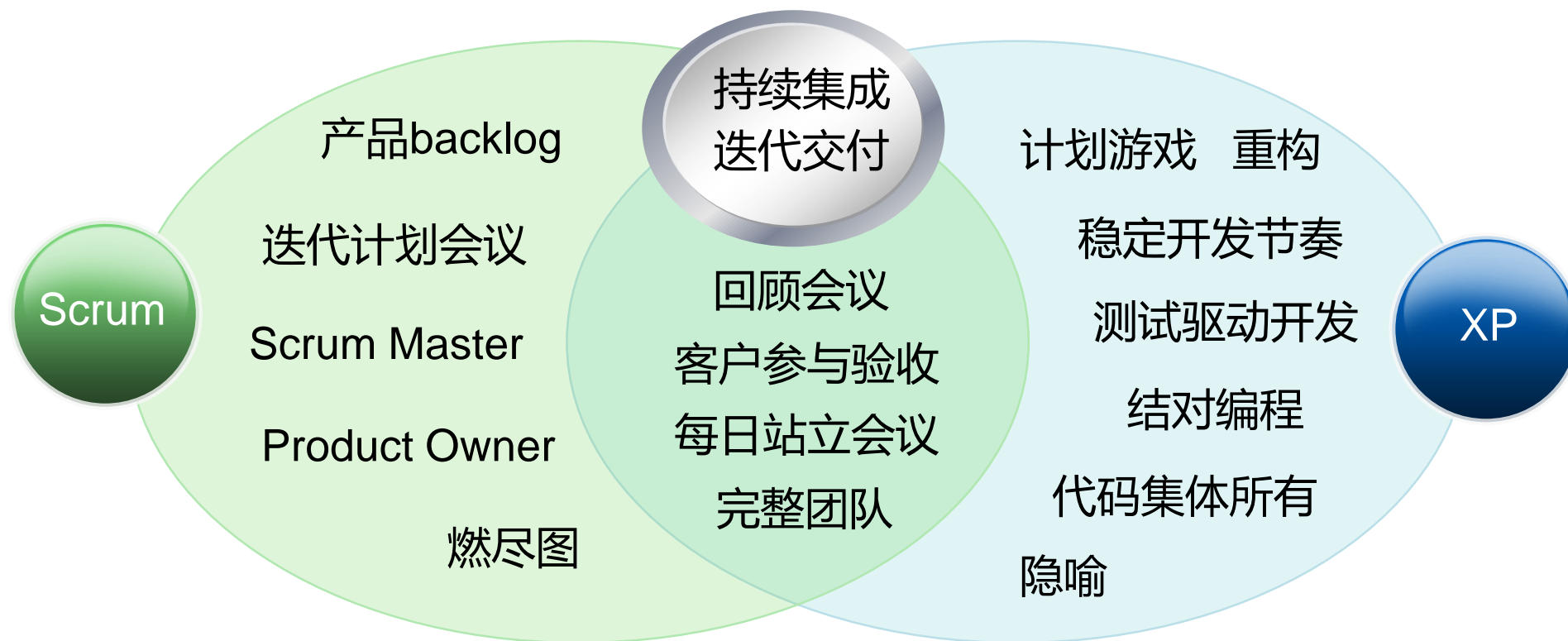
清华大学软件学院 刘强





1	极限编程 XP
2	软件编程规范
3	代码评审技术
4	结对编程实践

回顾：敏捷开发方法

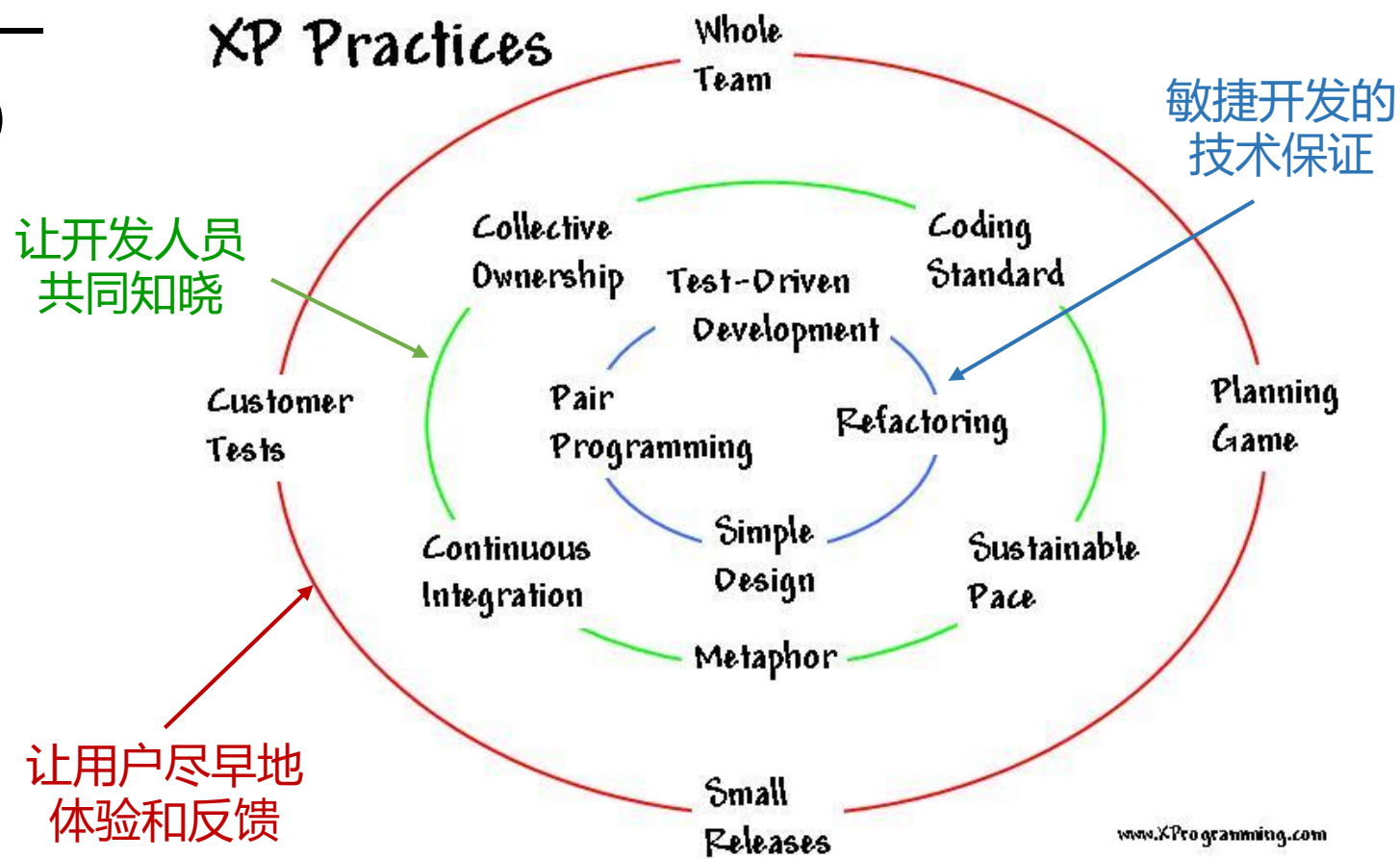


Scrum偏重项目管理

XP偏重编程实践

极限编程实践

极限编程 —— 敏捷开发方法之一
(eXtreme Programming, XP)



开发过程

完整的团队 (Whole Team)

- 客户应该是能够和团队一起工作的团队成员，最好是能够在同一个房间中工作
- 如果确实无法和客户工作在一起，你会怎么办？

计划游戏 (Play Game)

- 将整个项目规划为若干用户故事，与客户一起确定每个故事的重要程度和优先级，规划每个开发周期（一般建议是2周）要实现哪些故事。

小型发布 (Small Release)

- 每两周发布一个小型版本让用户尽早体验和收集反馈，尽早地发现和解决问题

验收测试 (Customer Tests)

- 验收测试可以使用脚本语言实现自动化，也可以人工进行，验证系统符合客户规定行为

计划游戏

敏捷估算扑克本质上是扑克牌，它基于Delphi估算原理，可以快速地估算出需要的数字。



- 估算扑克牌上的数字：有些牌是自然数排列，有些是斐波纳契数，有些则是不连续自然数。
- 具体选用哪种扑克，要根据被估算的内容的跨度大小而定，如果估算值跨度在10倍以内，那么采用顺序自然数比较好，如果数值跨度较大，达到10倍以上，那么采用斐波纳契数比较好。
- 一般而言，估算软件开发工时的话，自然数可能更好一些，毕竟数值都不大，跨度也不会很夸张。

计划游戏

分牌：每名参与估算的成员分得相同花色的一组牌，两张Joker不参与估算。



敏捷扑克和普通游戏扑克一样，也有54张牌，也拥有4种花色（每种各13张）和两张Joker。

敏捷扑克的每种花色均是一组13张牌组成的估算扑克牌，牌正面上印刷有供估算用的数字与符号，数字分别是1/2、1~10和20以及符号“！”（代表一些未知情况，如无法提供准确估算值等）。

- 一副估算扑克可供四人使用，如果参与的人员多于4人，可使用多副扑克。
- 一般推荐4-8人参与估算，人太少会使估算结果偏差很大，而太多会拉长估算时间，降低估算效率。

计划游戏

讲解订单故事：产品负责人从Backlog中选择一个条目，为大家详细讲解该条目；团队成员进行讨论并提问，产品负责人逐一解答大家的问题。



在讲解过程中，千万不要指定该条目的负责人或有明显倾向的人来做这个条目，这样会大大降低团队成员的积极性，甚至会扰乱估算秩序与结果。

这个步骤是团队和产品负责人之间的交互环节，帮助团队和产品负责人共同加深对条目的理解；产品负责人也会根据大家的反馈，及时修改和完善条目。



计划游戏

估算：当团队成员确认已经对该条目完全了解且无任何重大问题后，大家开始对该条目进行估算，同时选出代表自己估算值的纸牌，但不可立即亮牌。在估算过程中，为避免干扰估算结果，团队成员之间不可以互相商讨。当所有成员选牌完毕，大家可以同时亮牌。



计划游戏



VS



争论与讨论：对比每张牌估算值之间的大小，若估算值差距明显，代表大家对该条目的价值没有获得共识，团队需要对该条目价值评估结果进行讨论。

共识：对该条目重新进行估算，直到团队的评估数值达成一致。一般情况下，最多三轮就可以得出一个比较统一的意见；如果三轮之后依然没有得到统一的意见，那么Scrum Master应当立即中断该条目的估算，取平均值或其他大家能接受的值作为估算结果。



团队协作

编码规范 (Code Standards)

- 建议采用统一的编码规范，这样开发人员编写出来的代码才能易于阅读和理解。

集体所有权 (Collective Ownership)

- 实现代码的版本管理，保证大家都能看到和共享这些代码。

可持续的开发速度 (Sustainable Pace)

- XP有一个40小时工作的规则，在版本发布前的一个星期是该规则的唯一例外。如果发布目标就在眼前并且能够一蹴而就，则允许加班。

持续集成 (Continuous Integration)

- 团队开发成员频繁地集成他们的工作，每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽早地发现集成错误。

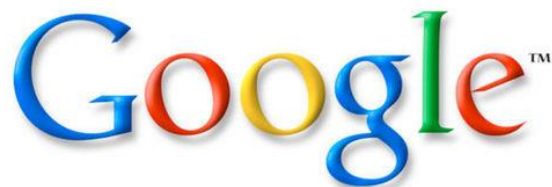
软件编码规范是与特定语言相关的描写如何编写代码的规则集合。

现实

- 软件全生命周期的 70% 成本是维护
- 软件在其生命周期中很少由原编写人员进行维护

目的

- 提高编码质量，避免不必要的程序错误
- 增强程序代码的可读性、可重用性和可移植性



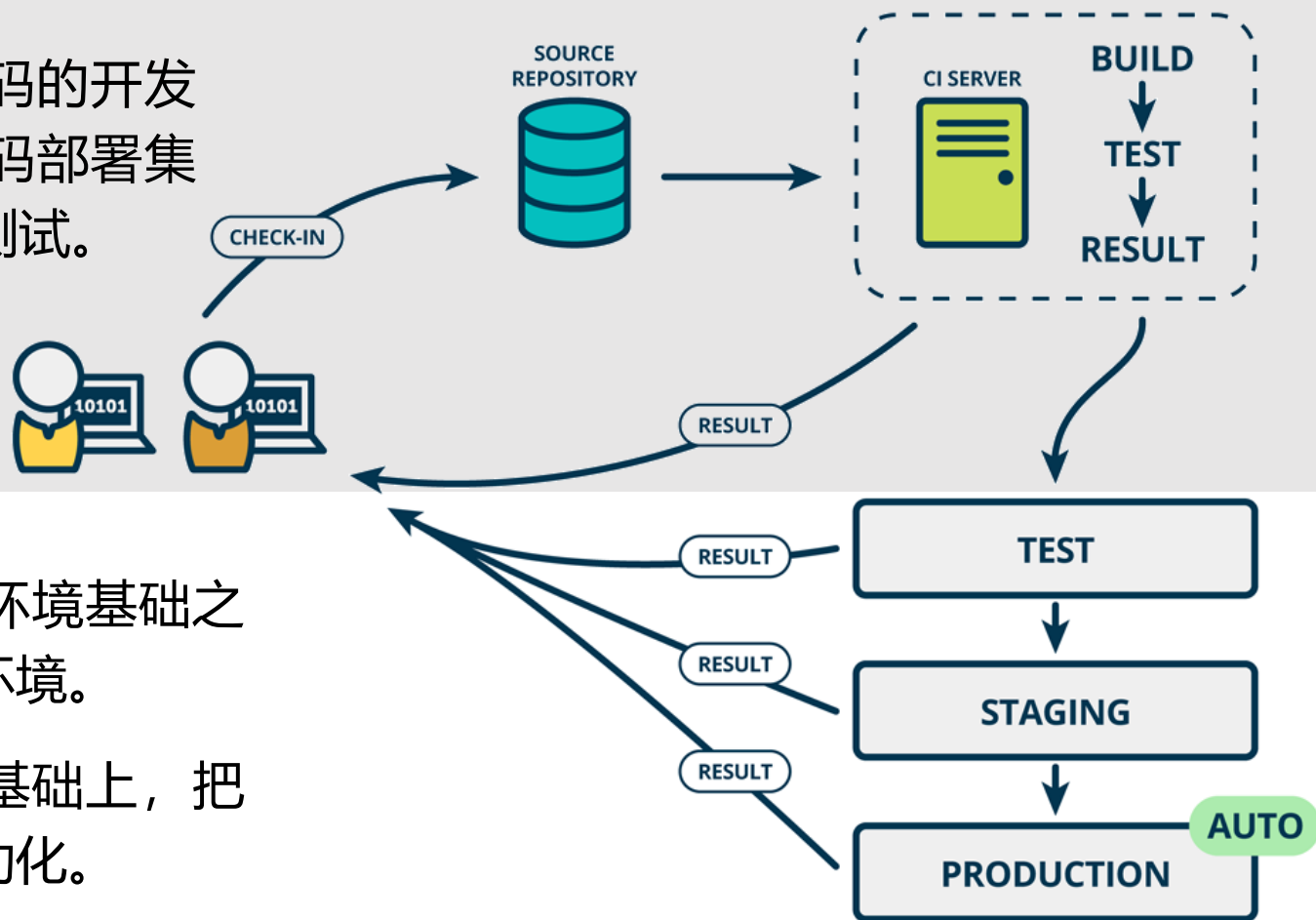
<https://github.com/google/styleguide>

Our [C++ Style Guide](#), [Objective-C Style Guide](#), [Java Style Guide](#), [Python Style Guide](#), [Shell Style Guide](#), [HTML/CSS Style Guide](#), [JavaScript Style Guide](#), [AngularJS Style Guide](#), [Common Lisp Style Guide](#), and [Vimscript Style Guide](#) are now available. We have also released [cpplint](#), a tool to assist with style guide compliance, and [google-c-style.el](#), an Emacs settings file for Google style.

If your project requires that you create a new XML document format, our [XML Document Format Style Guide](#) may be helpful. In addition to actual style rules, it also contains advice on designing your own vs. adapting an existing format, on XML instance document formatting, and on elements vs. attributes.

持续集成与交付

持续集成是指开发者在代码的开发过程中，可以频繁地将代码部署集成到主干，并进行自动化测试。



持续交付是在持续集成的环境基础之上，将代码部署到预生产环境。

持续部署是在持续交付的基础上，把部署到生产环境的过程自动化。

技术保证

简单设计 (Simple Design)

- XP团队使他们的设计尽可能的简单、有表达力
- 仅关注于计划在本次迭代中要完成的用户故事，而不会考虑那些未来的用户故事
- 团队更愿意在迭代中不断地变迁系统设计，使其对正在实现的用户故事保持在最优状态

结对编程 (Pair Programming)

- 代码都是由结对的程序员使用同一台机器共同完成的

重构 (Refactoring)

- “简单设计” 可能导致没有考虑未来的变化，从而在添加新需求时代码越来越糟。
- 极限编程通过重构来保证代码不再腐化，并且通过单元测试来保证重构的正确性。

测试驱动开发 (Test-Driven Development)

- 在编写代码之前先写相应的测试用例，编写的代码要通过测试，并不断进行重构优化。

简单设计

敏捷方法认为设计非常重要，因此应该是一个持续的事务。设计只考虑当前的需求定义，先尝试使用能够工作的最简单设计，然后随着现实的不断显现来更改它。

- 第一次迭代搭建基本的系统框架
- 以后的迭代过程是在反馈和编程的基础上做交互式设计，减少了设计的投机性
- 重构对设计进行优化

对简单设计的需求并不是说所有设计都很小，也不表示它们是无足轻重的；它们只不过需要尽可能地简单，但是仍能工作。

重构

重构（Refactoring）是对软件内部结构的一种调整，其目的是在不改变软件功能和外部行为的前提下，提高其可理解性、可扩展性和可重用性。



软件的可测试性

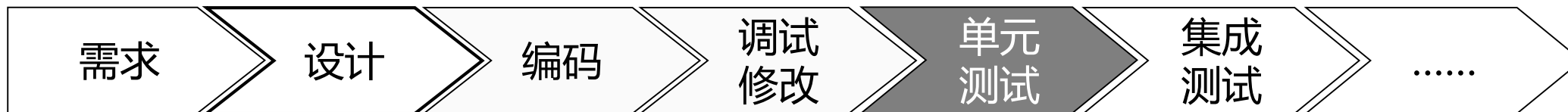
软件可测试性是指一个计算机程序能够被测试的容易程度。

- 可测试性好的软件：强内聚、弱耦合、接口明确、意图明晰
- 不可测的软件：过强的耦合、混乱的逻辑



传统的开发方法

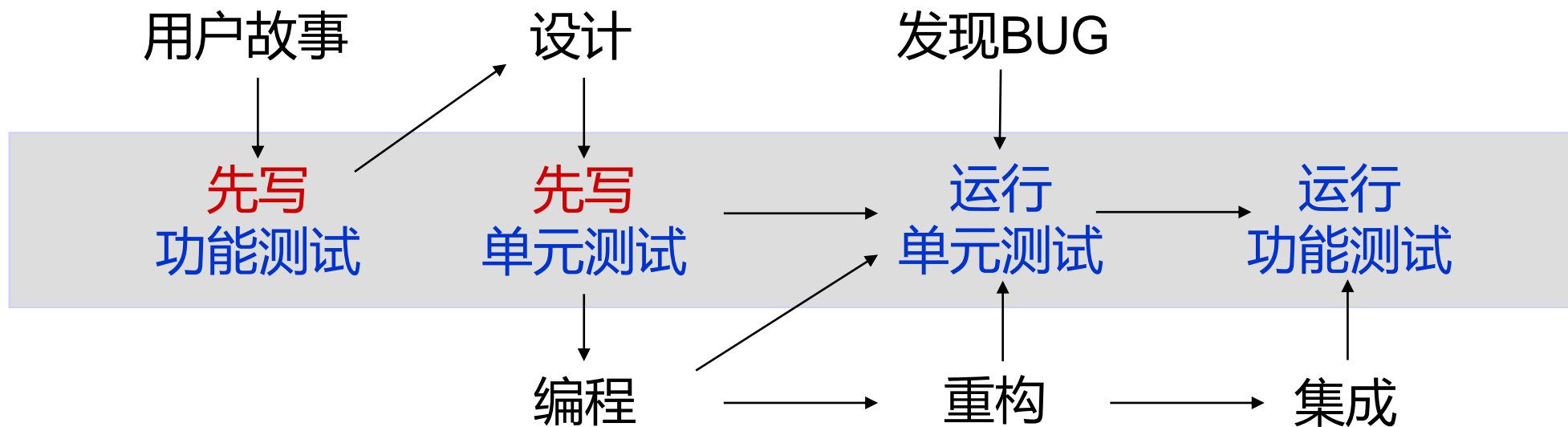
软件开发过程：先设计，再实现，最后测试



迫于时间压力，程序员往往会简化或者不执行单元测试

测试驱动开发

测试驱动开发是以测试作为编程中心，要求在编写任何代码之前，首先编写定义代码功能的测试用例，编写的代码要通过测试，并不断进行重构优化。



测试驱动开发过程

- 对要解决的问题建模，编写测试代码
- 编写解决问题代码，并使其通过测试
- 思考意外情况，编写额外的测试
- 解决意外情况，使测试通过
- 重复上述流程，直到想不到意外



测试驱动开发

- 先写测试可以帮助开发者集中精力于编写真正需要的代码
- 测试实际上展示了代码是如何工作的，在一定程度上测试代码变成了实现代码的使用文档，部分体现了“代码即文档”的思想
- 编写测试有助于开发者发现代码中可以抽象出来的API，从而将测试变成了设计过程的一部分
- 随着测试驱动开发的深入，我们会发现测试代码逐渐演变为对系统行为的定义描述

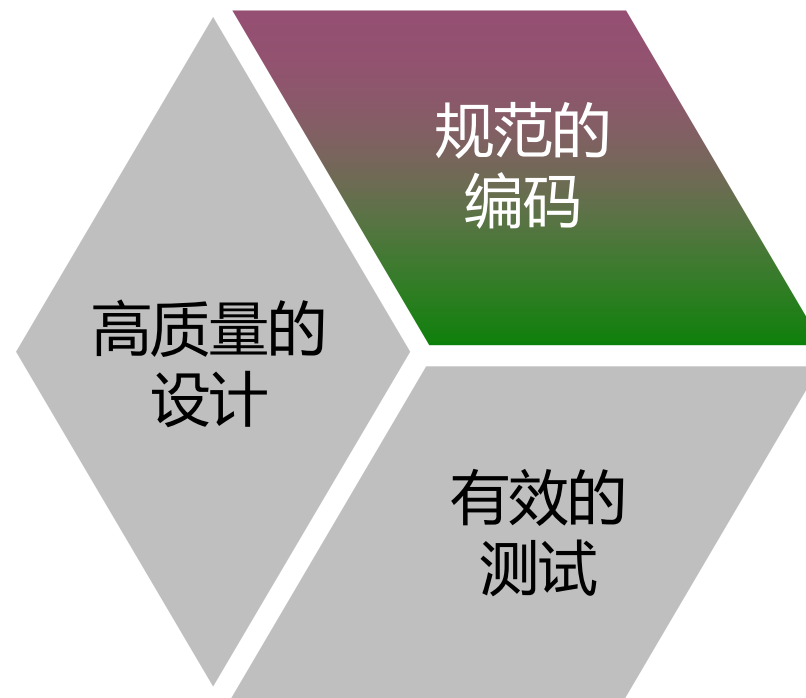


1	极限编程 XP
2	软件编程规范
3	代码评审技术
4	结对编程实践

软件编程规范



高质量软件
开发之道



Python编程规范：程序模板

```
# !/usr/bin/env python3
# -*- coding: utf-8 -*-
#
__author__ = 'YourName'
```

被直接执行的文件建议增加此行，被导入的模块请忽略
文件的基本头部

```
import sys
```

模块的导入总应该放在文件顶部

```
def main(argv):
    """Do something..."""
    return 0
```


主功能建议创建main函数来执行
函数、类等应遵循一定的注释规范

```
if __name__ == '__main__':
    exit(main(sys.argv[1:]))
```

即使打算被直接执行的文件也应是可导入的，
这样当模块被导入时主程序就不会被执行


Python编程规范：注释

- 形式1：由 # 开头的“真正的”注释，说明选择当前实现的原因以及这种实现的原理和难点；



```
def gold_divide(n):  
    # 黄金分割点比例为 $(\sqrt{5}-1)/2 \approx 0.618034$   
  
    return n * 0.618 # 直接取0.618以加速
```

- 形式2：文档字符串，说明如何使用包、模块、类、函数（方法），甚至包括使用示例和单元测试。



```
def gold_divide(n):  
    """ Get gold divide value of n.  
  
    Args:  
        n: input number  
  
    Returns:  
        A float value  
    """  
    return n * 0.618
```

```
def main(argv):
```

```
    """
```

函数main, 返回0表示操作成功, 返回-1表示失败

```
    """
```

```
    try:
```

```
        # 以二进制只读方式打开文件
```

```
        with open('cat.pic', 'rb') as fin: # fin是输入文件
```

```
            # 将文件指针移动到文件尾
```

```
            if fin.seek(0, 2) != 400 * 400 * 3: # 判断文件尾位置是否等于400*400*3
```

```
                print("输入文件 cat.pic 不符合格式要求") # 如果不等, 显示错误信息
```

```
                return -1
```

```
            fin.seek(0) # 将文件指针移动到文件开头
```

```
            try:
```

```
                # 以二进制写方式打开文件
```

```
                with open('cat2.pic', 'wb') as fout: # fout是输出文件
```

```
                    for i in range(400): # 从图像的第1行到第400行循环
```

```
                        for j in range(400): # 从每行的第1列到第400列循环
```

```
                            # 从输入文件中读入每像素的RGB值
```

```
                            b = ord(fin.read(1))
```

```
                            g = ord(fin.read(1))
```

```
                            r = ord(fin.read(1))
```

```
                            # 按照公式  $Y=0.299R+0.587G+0.114B$  计算灰度值
```

```
                            y = (299 * r + 587 * g + 114 * b) / 1000
```

```
                            fout.write(chr(y)) # 将计算出来的灰度值写到输出文件中去
```

```
            except IOError as e:
```

```
                print("打开文件 cat2.pic 时错误") # 如果打开失败则显示错误信息
```

```
                return -1
```

```
except IOError as e:
```

```
    print("打开文件 cat.pic 时错误") # 如果打开失败则显示错误信息
```

```
    return -1
```

```
return 0 # 返回0表示正确处理完毕
```

注释仅仅是语句的重复解释, 没有任何价值



你明白这段代码的作用吗?

学会只编写够用的注释，过犹不及，重视质量而不是数量。

- 好的注释解释为什么，而不是怎么样
- 不要在注释中重复描述代码
- 当自己在编写密密麻麻的注释来解释代码时，需要停下来看是否存在更大的问题
- 想一想在注释中写什么，不要不动脑筋就输入
- 写完注释之后要在代码的上下文中回顾一下，它们是否包含正确的信息？
- 当修改代码时，维护代码周围的所有注释

```

def main(argv):
    """
    主函数, 返回0表示成功
    """
    try:
        with open('cat.pic', 'rb') as fin:
            if fin.seek(0, 2) != 400 * 400 * 3: # 判断文件长度是否符合格式要求
                print("输入文件 cat.pic 不符合格式要求")
                return -1
            fin.seek(0)
            try:
                with open('cat2.pic', 'wb') as fout:
                    # 下面是图像转换的算法实现。彩色图像到灰度图像的转换主要利用
                    # RGB色彩空间到YUV色彩空间的变换公式来取得灰度值Y, 公式是
                    #  $Y = 0.299R + 0.587G + 0.114B$ 
                    for i in range(400):
                        for j in range(400):
                            b = ord(fin.read(1))
                            g = ord(fin.read(1))
                            r = ord(fin.read(1))
                            y = (299 * r + 587 * g + 114 * b) / 1000
                            fout.write(chr(y))
            except IOError as e:
                print("打开文件 cat2.pic 时错误")
                return -1
    except IOError as e:
        print("打开文件 cat.pic 时错误")
        return -1
    return 0

```



Python编程规范：注释

```
# -*- encoding: utf-8 -*-

"""
斐波那契数列
"""

def fibonacci(position):
    """ 获得斐波那契数列指定位置上的数。

    斐波那契数列是形如1,1,2,3,5,8,11,...的数列，通项公式为 $F_0=F_1=1, F_n=F_{n-1}+F_{n-2}, n \geq 2$ 。

    Args:
        position: 需要获得的指定位置

    Returns:
        一个数，即 $F_{position}$ 
    """
    if position < 2:
        return 1

    previousButOne = 1
    previous = 1
    result = 2
    for n in range(2, position):
        previousButOne = previous
        previous = result
        result = previous + previousButOne
    return result
```

pydoc

fibonacci

斐波那契数列

Functions

fibonacci(position)

获得斐波那契数列指定位置上的数。

斐波那契数列是形如1,1,2,3,5,8,11,...的数列，通项公式为 $F_0=F_1=1, F_n=F_{n-1}+F_{n-2}, n \geq 2$ 。

Args:

position: 需要获得的指定位置

Returns:

一个数，即 $F_{position}$

Python编程规范：命名

好的名字一目了然，不需要读者去猜，甚至不需要注释。



- Python库的命名约定有点混乱，因此很难使之变得完全一致，不过还是有公认的命名规范。
- 新的模块和包（包括第三方的框架）必须符合这些标准，但对已有的库存在不同风格的，保持内部的一致性的首选的。

Python编程规范：命名

不要编写需要外部文档支持的代码，这样的代码是脆弱的，要确保你的代码本身读起来就很清晰。

编写自文档化的代码

- 唯一能完整并正确地描述代码的文档是代码本身
- 编写可以阅读的代码，其本身简单易懂



Python编程规范：命名



```
def fval(i):  
    ret = 2  
    n1 = 1  
    n2 = 1  
    i2 = i - 3  
    while i2 >= 0:  
        n1 = n2  
        n2 = ret  
        ret = n2 + n1  
        i2 -= 1  
    return 1 if i < 2 else ret
```



Python编程规范：命名



```
def fval(i):
    ret = 2
    n1 = 1
    n2 = 1
    i2 = i - 3
    while i2 >= 0:
        n1 = n2
        n2 = ret
        ret = n2 + n1
        i2 -= 1
    return 1 if i < 2 else ret
```



```
def fibonacci(position):
    if position < 2:
        return 1

    previous_but_one = 1
    previous = 1
    result = 2
    for n in range(2, position):
        previous_but_one = previous
        previous = result
        result = previous + previous_but_one
    return result
```

Python编程规范：语句

```
print("Hello, world!");
```

不要在行尾加分号

```
for i in range(n):  
    result += i
```

不要使用制表符（tab）进行缩进，千万不要混用tab与空格，应使用4个空格进行缩进

```
class book shelf:  
    pass
```

类名应为驼峰风格且首字母大写，这里应该是BookShelf

```
bookShelf = BookShelf()
```

变量名应为下划线风格，这里应该是book_shelf

```
f = open('output.txt', 'w')
```

使用文件时应注意显式地调用close()，或使用with：

```
if n > 5: print(n)  
else: n = 6
```

每行只写一条语句

```
with open('output.txt', 'w') as f:  
    pass
```

import 语句应遵循的原则：

- import 次序：先 import Python 内置模块，再 import 第三方模块，最后 import 自己开发的项目中的其它模块；这几种模块中用空行分隔开来。
- 一条 import 语句 import 一个模块。
- 当从模块中 import 多个对象且超过一行时，使用如下断行法（py2.5以上版本）：

```
from module import (obj1, obj2, obj3, obj4, obj5, obj6)
```

- 不要使用 from module import *, 除非是 import 常量定义模块或其它你确保不会出现命名空间冲突的模块。

Python编程规范：语句

Python还有很多灵活的用法，使用时应注意删繁就简。

- Python的列表推导简洁方便，但仅应在简单的情况下使用

✓ `[s['name'] for s in students if s['age'] > 18]`

✗ `[(x, y) for x in range(m) for y in range(n) if x ** 2 + y ** 2 >= 4]`

- Python的条件表达式是if语句的简短语法规则，但同样仅应在简单的情况下使用

✓ `return s['name'] if s['age'] > 18 else s['nickname']`

✗ `return s['name'] if s['age'] > 18 else s['nickname'] if s['age'] > 14 else 'anonymous'`

- Python还有一些很“酷”的特性，但奇技淫巧的代码会难以开发、调试、维护



1	极限编程 XP
2	软件编程规范
3	代码评审技术
4	结对编程实践

编写优美的程序

任何一个傻瓜都能写出计算机可以理解的程序，
只有写出人类容易理解的程序才是优秀的程序员。

—— Martin Fowler



高效程序员不应该浪费很多时间用于程序调试，
他们应该一开始就不要把故障引入。

—— Edsger Dijkstra



编写优美的程序

优美的代码可理解性高，修改成本低。不过优美的代码是很难一次写出的！

- 对代码的逻辑层次要有感觉
- 善于抽取算法代码
- 提取工具代码
- 提高抽象技巧



代码评审技术

代码检查 (Code Review) 是一种用来确认方案设计和代码实现的质量保证机制，用于在开发过程中改进代码质量。

代码检查的作用

- 检查设计的合理性
- 互为 Backup
- 分享知识、设计、技术
- 增加代码可读性
- 处理代码中的“地雷区”



缺陷检查表

编程规范

- 按照具体编程语言的编码规范进行检查，包括命名规则、程序注释、缩进排版、声明与初始化、语句格式等。

面向对象设计

- 类的设计和抽象是否合适
- 是否符合面向接口编程的思想
- 是否使用合适的设计模式

性能方面

- 在出现海量数据时，队列、表、文件在传输、上载等方面是否会出现问题，是否控制如分配的内存块大小、队列长度等
- 对 Hashtable、Vector 等集合类数据结构的选择和设置是否合适
- 有无滥用 String 对象的现象
- 是否采用通用的线程池、对象池等高速缓存技术以提高性能
- 类的接口是否定义良好，如参数类型等应避免内部转换

性能方面（续）

- 是否采用内存或硬盘缓冲机制以提高效率？
- 并发访问时的应对策略
- I/O 方面是否使用了合适的类或采用良好的方法以提高性能（如减少序列化、使用 buffer 类封装流等）
- 同步方法的使用是否得当，是否过度使用？
- 递归方法中的迭代次数是否合适（应保证在合理的栈空间范围内）
- 如果调用了阻塞方法，是否考虑了保证性能的措施
- 避免过度优化，对性能要求高的代码是否使用profile工具

资源释放处理

- 分配的内存是否释放，尤其在错误处理路径上（如 C/C++）
- 错误发生时是否所有对象被释放，如数据库连接、Socket、文件等
- 是否同一个对象被释放多次（如 C/C++）
- 代码是否保存准确的对象引用计数

缺陷检查表

程序流程

- 循环结束条件是否准确
- 是否避免了死循环的产生
- 对循环的处理是否合适，应考虑到性能方面的影响

线程安全

- 代码中所有的全局变量是否是线程安全的
- 需要被多个线程访问的对象是否线程安全，检查有无通过同步方法保护
- 同步对象上的锁是否按相同的顺序获得和释放以避免死锁，注意错误处理代码
- 是否存在可能的死锁或是竞争，当用到多个锁时，避免出现类似情况：线程A获得锁1，然后锁2，线程B获得锁2，然后锁1
- 在保证线程安全的同时，注意避免过度使用同步，导致性能降低

数据库处理

- 数据库设计或SQL语句是否便于移植（注意与性能会存在冲突）
- 数据库资源是否正常关闭和释放

数据库处理（续）

- 数据库访问模块是否正确封装，便于管理和提高性能
- 是否采用合适的事务隔离级别
- 是否采用存储过程以提高性能
- 是否采用 PreparedStatement 以提高性能

通讯方面

- Socket 通讯是否存在长期阻塞问题
- 发送接收的数据流是否采用缓冲机制
- Socket 超时处理和异常处理
- 数据传输的流量控制问题

JAVA对象处理

- 对象生命周期的处理，是否对象引用已失效可设置 null 并被回收
- 在对象传值和传参方面有无问题，对象的 clone 方法使用是否过度
- 是否大量经常地创建临时对象
- 是否尽量使用局部对象（堆栈对象）
- 在只需要对象引用的地方是否创建了新的对象实例

缺陷检查表

异常处理

- 每次当方法返回时是否正确处理了异常，如最简单的处理是记录日志到日志文件中
- 是否对数据的值和范围是否合法进行校验，包括使用断言
- 在出错路径上是否所有的资源和内存都已经释放
- 所有抛出的异常是否都得到正确的处理，特别是对子方法抛出的异常，在整个调用栈中必须能够被捕捉并处理
- 当调用导致错误发生时，方法的调用者应该得到一个通知
- 不要忘了对错误处理部分的代码进行测试，很多代码在正常情况下执行良好，而一旦出错整个系统就崩溃了？

方法（函数）

- 方法的参数是否都做了校验
- 数组类结构是否做了边界校验

方法（续）

- 变量在使用前是否做了初始化
- 返回堆对象的引用，不要返回栈对象的引用
- 方法的 API 是否被良好定义，即是否尽量面向接口编程，以便于维护和重构

安全方面

- 对命令行执行的代码，需要详细检查命令行参数
- WEB 类程序检查是否对访问参数进行合法性验证
- 重要信息的保存是否选用合适的加密算法
- 通讯时考虑是否选用安全的通讯方式

其他

- 日志是否正常输出和控制
- 配置信息如何获得，是否有硬编码

代码静态分析工具



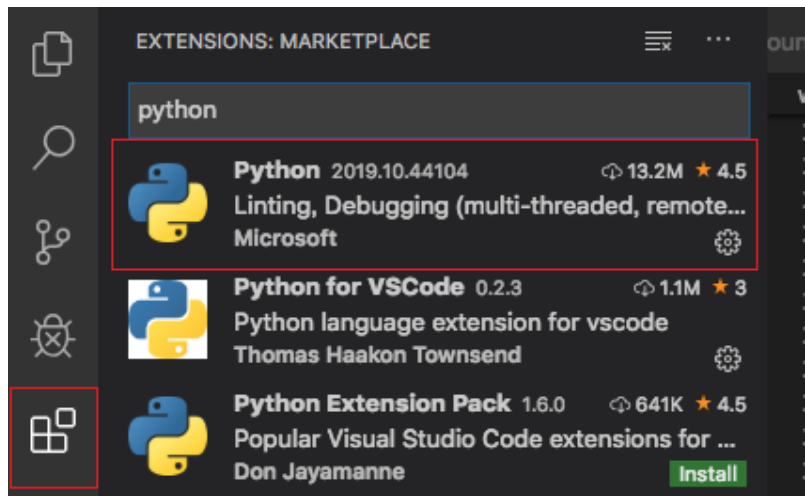
- Google 风格指南: <https://github.com/google/eslint-config-google>
- Airbnb 风格指南: <https://github.com/airbnb/javascript>
- ESLint 工具: <https://eslint.org/>



- 官方风格指南: <https://www.python.org/dev/peps/pep-0008/>
 - 工具: pylint、flake8、pep8等
- 参见: <https://code.visualstudio.com/docs/python/linting>

VS Code 代码风格检查与格式化工具——以Python为例

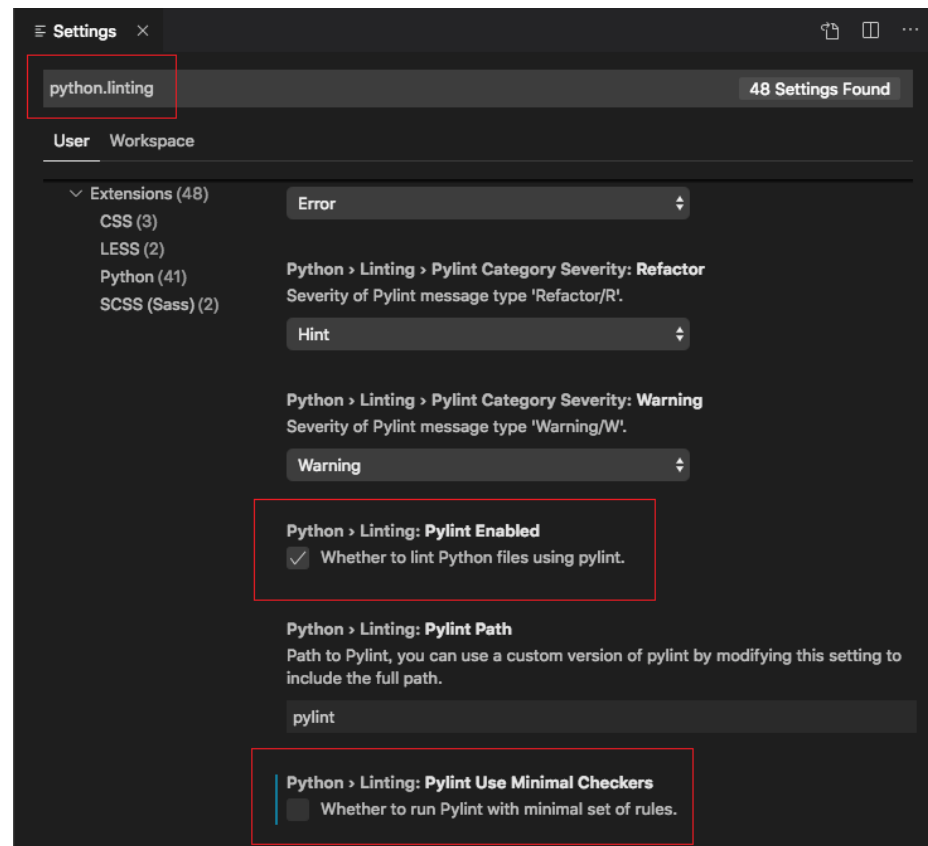
1、安装VS Code Python扩展



注意：

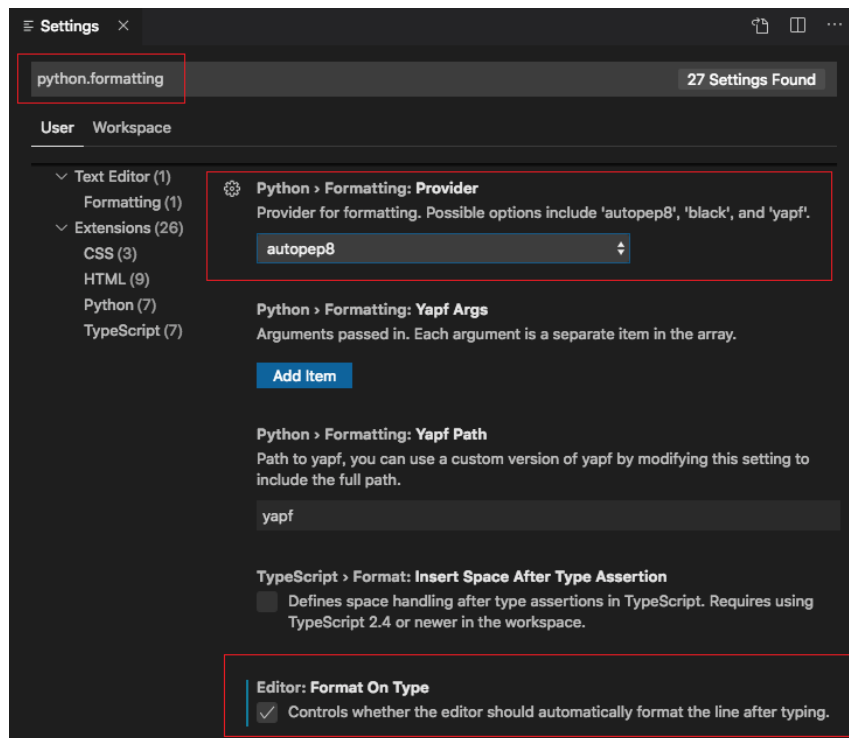
- 1、如果你没有安装相应的Lint工具或Formatter工具，当编辑python文件时，VS Code会在右下角弹框提示你，点击“install”安装工具后才能生效。也可自行pip安装。
- 2、如果你希望Lint工具严格按照代码规范检查，请将Lint工具对应的“Use Minimal Checkers”一项置为False

2、打开设置面板，将你想要的Lint工具设置为 “Enabled”



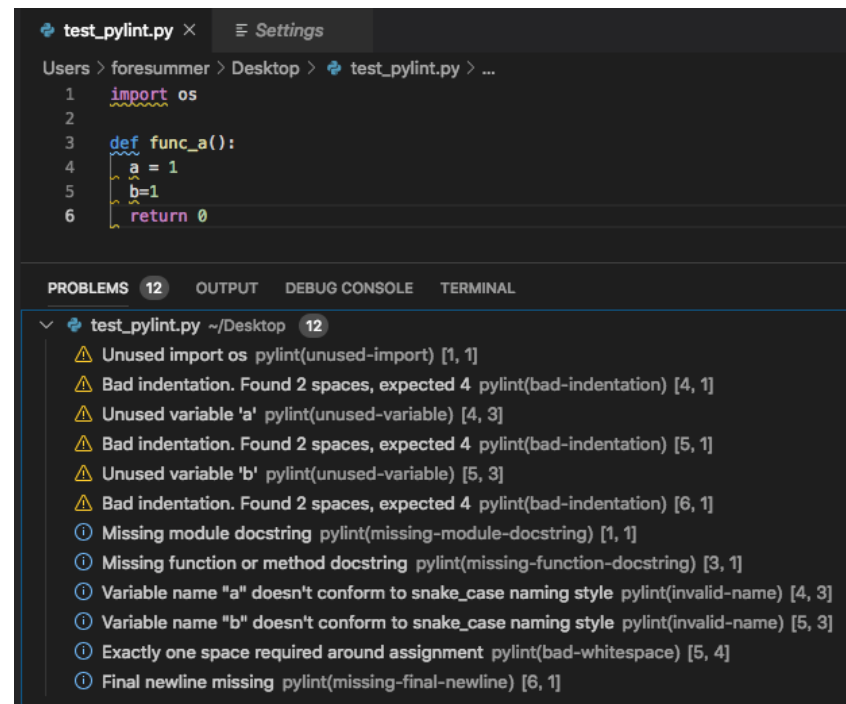
VS Code 代码风格检查与格式化工具——以Python为例

3、在设置面板上设置格式化工具



仅以autopep8为例，其他工具也可以使用

4、Enjoy it!



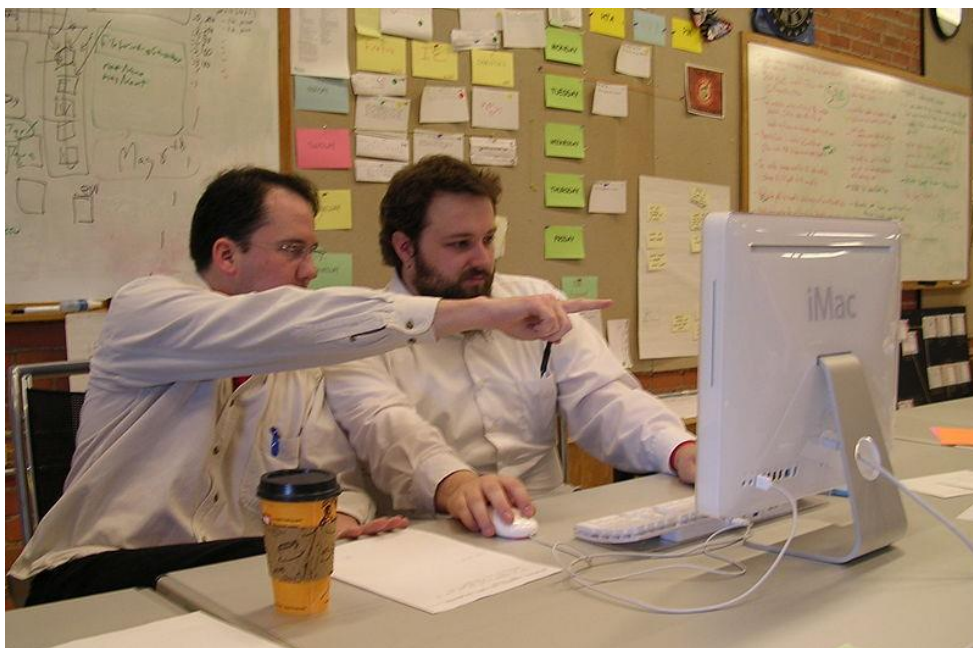
使用快捷键调用格式化工具：Ctrl + Shift + I（Linux）、Shift + Alt + F（Windows系统）或 ⌘(Shift) + ⌥(Option) + F（Mac系统）。



1	极限编程 XP
2	软件编程规范
3	代码评审技术
4	结对编程实践

结对编程

结对编程是由两名程序员在同一台电脑上结对编写解决同一问题的代码。



结对编程



结对编程

驾驶员：负责用键盘编写程序

领航员：起到领航、提醒的作用

两个人轮流驾驶，角色互换



- 结对编程不仅意味着编程活动，也包括分析、设计、测试等全程活动。
- 结对编程有助于按时完成项目，并且保证高质量的代码。

结对编程



Where is your partner?

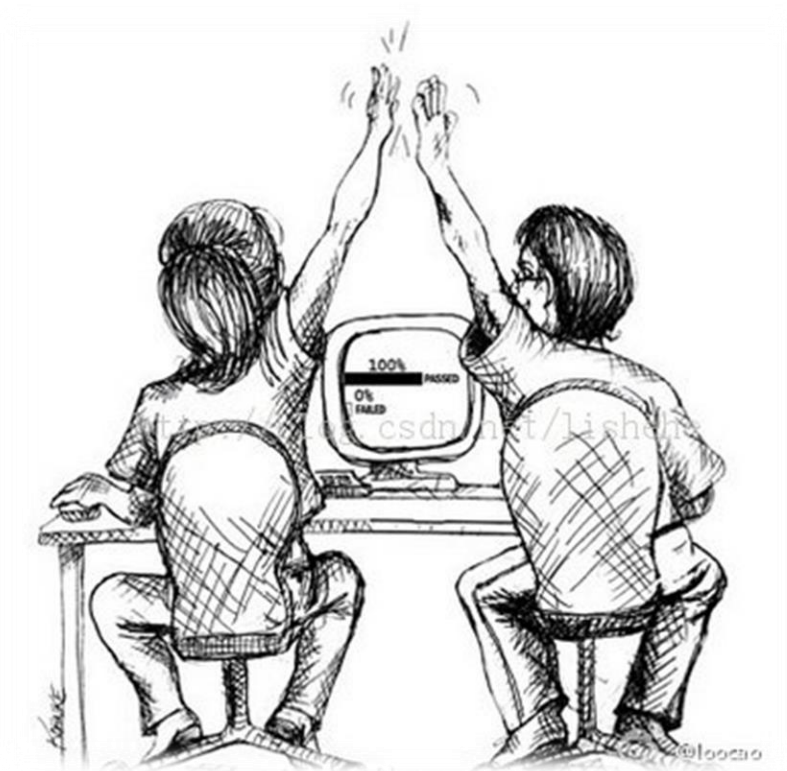
- 结对编程的基础是会话和讨论，不是一个人自得其乐。
- 口渴指数是核实伙伴交流程度的一个考核标准。

结对编程



- 驾驶员不要连续工作超过1个小时，每1个小时休息10分钟
- 给驾驶员一点时间去发现和找到错误，别让伙伴觉得你很烦
- 常用的资料和规范（可以打印出来）以及书籍等应该放在手边
- 结对开始之前要协调沟通，彼此互相通告希望对方关注些什么，自己喜欢做什么
- 主动参与，任何一个任务都是共同的责任，只有“我们的代码”
- 坚持代码标准和流程规范
- 注意听取伙伴的意见

结对编程



- 结对编程是一个相互学习、相互磨合的渐进过程，实施时需要团队成员克服个性冲突和习惯差异。
- 结对编程应该自由选择和灵活运用，它不应是强制性的，也不要教条地运行，最好由两位程序员自己决定合适的方式。

结对编程

下面的一些情况不适合结对编程：

- 处于探索阶段的项目
- 后期维护的技术含量不高
- 验证测试需要运行很长时间
- 团队的人员要在多个项目中工作
- 领航的用处不大

另外，也不是所有的人都适合结对编程。





谢谢大家!

THANKS

