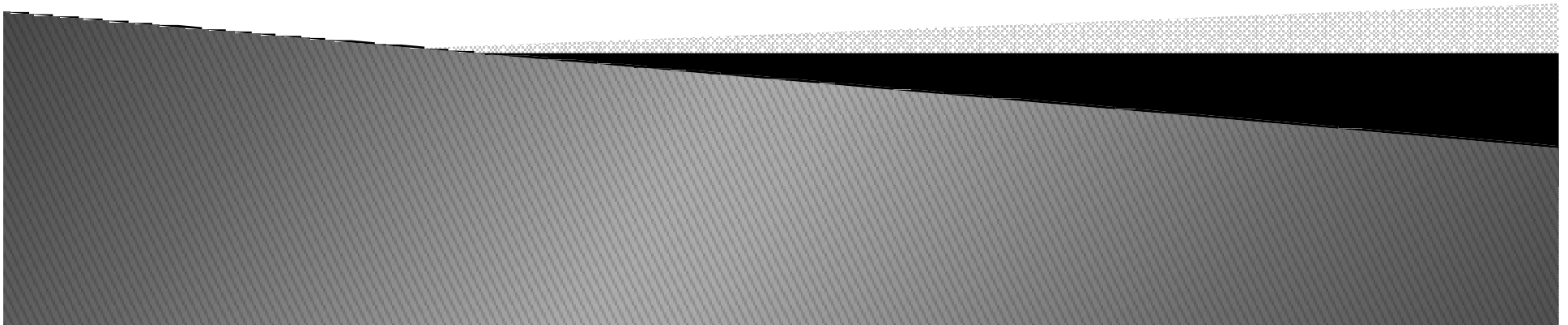


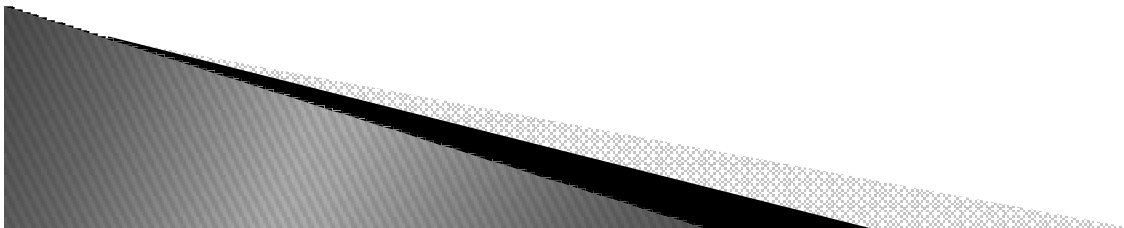
Testes Unitários

Gustavo Callou
gcallou@gmail.com



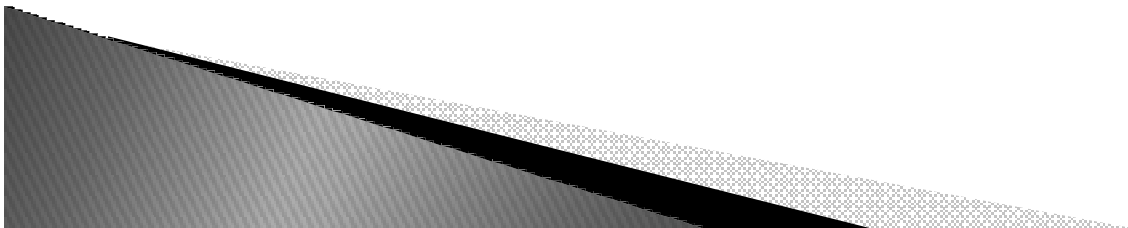
Testes Unitários

- ▶ Quando acabamos de desenvolver um software, será que acabou o serviço?
- ▶ Como saber se os requisitos foram atendidos?
- ▶ Os testes unitários podem ajudar.
- ▶ É conhecido o comportamento do código para determinadas entradas.
- ▶ Se o código se comportar conforme o esperado para determinada entrada, se admite que ele está correto.



Testes Unitários

- ▶ Duas formas fáceis de se automatizar a tarefa de testar:
 - usando o módulo doctest
 - ou unittest.
 - Esse módulos já vem com o Python por padrão.



Doctest

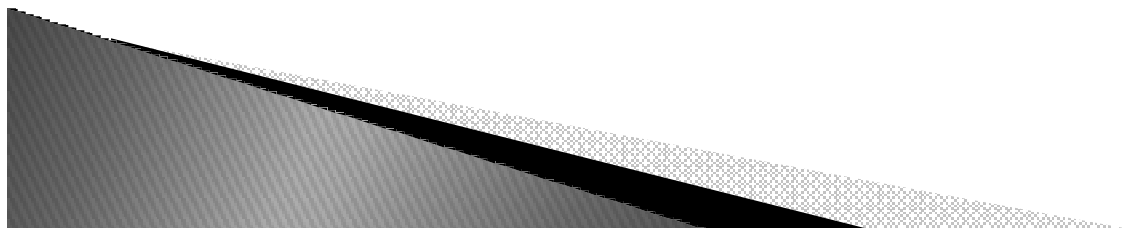
- ▶ Suponha que temos a seguinte função.

```
def fibo(n):  
    if n < 2:  
        return n  
    else:  
        return fibo(n-1) + fibo(n-2)
```

É conhecido o fibonacci de 1 é 1 e o fibonacci de 10 é 55.

Procedimento não automatizado para testar, abrir criar um programa de teste para testar esses valores.

No lugar de sempre criar um programa para testar, vamos usar o Doctest.



Doctest

- ▶ Com o doctest é possível criar testes unitários ao adicionar linhas de comentários ao seu código e depois fazer uma chamada ao doctest.
- ▶ Exemplo:

```
def fibo(n):  
    """  
    >>> fibo(0)  
    0  
    >>> fibo(1)  
    1  
    >>> fibo(10)  
    55  
    """  
    if n < 2:  
        return n  
    else:  
        return fibo(n-1) + fibo(n-2)  
  
import doctest  
doctest.testmod()
```

Doctest

- ▶ Com o doctest é possível criar testes unitários ao adicionar linhas de comentários ao seu código e depois fazer uma chamada ao doctest.

- ▶ Exemplo:

```
def fibo(n):  
    """  
    >>> fibo(0)  
    0  
    >>> fibo(1)  
    1  
    >>> fibo(10)  
    55  
    """  
    if n < 2:  
        return n  
    else:  
        return fibo(n-1) + fibo(n-2)
```

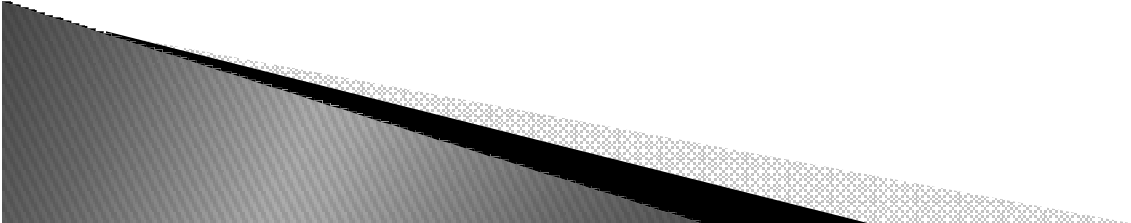
```
import doctest  
doctest.testmod()
```

Doctest

Exemplo do erro ao executar o teste:

```
$ python fibo.py
*****
File "fibo.py", line 7, in __main__.fibo
Failed example:
fibo(10)
Expected:
55
Got:
20
*****

1 items had failures:
1 of 3 in __main__.fibo
***Test Failed*** 1 failures.
```



Unittest

```
import unittest
from fibo import fibo

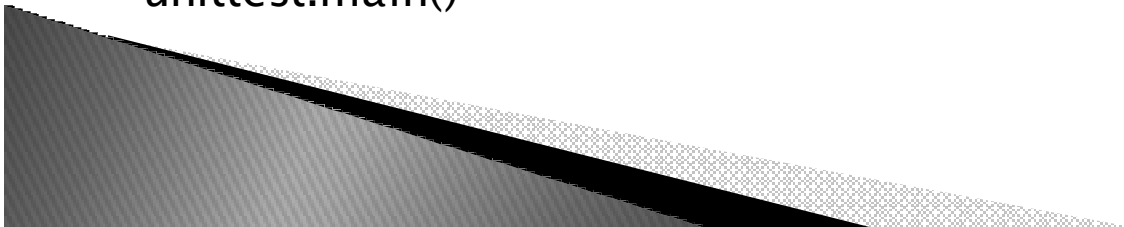
class testa_fibonacci(unittest.TestCase):
    def teste_um(self):
        self.assertEqual(fibo(0),0)

    def teste_dois(self):
        self.assertEqual(fibo(1),1)

    def teste_tres(self):
        self.assertEqual(fibo(7),13)

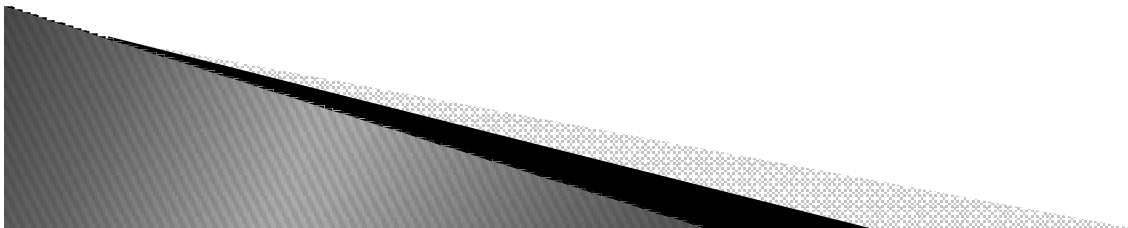
    def teste_quatro(self):
        self.assertEqual(fibo(10),55)

unittest.main()
```

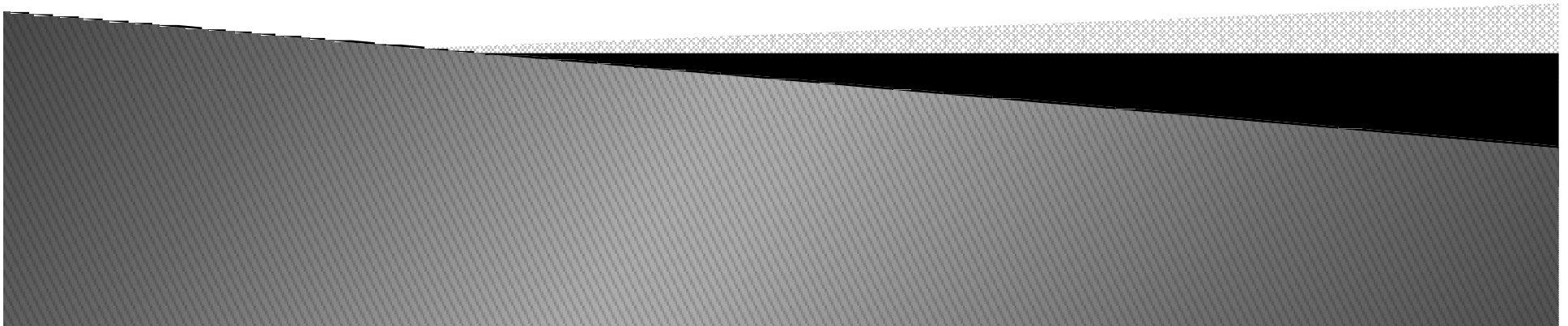


Exercício

- ▶ Definir uma classe fibo, e fazer 2 testes unitários (1 Doctest e 1 Unittest)

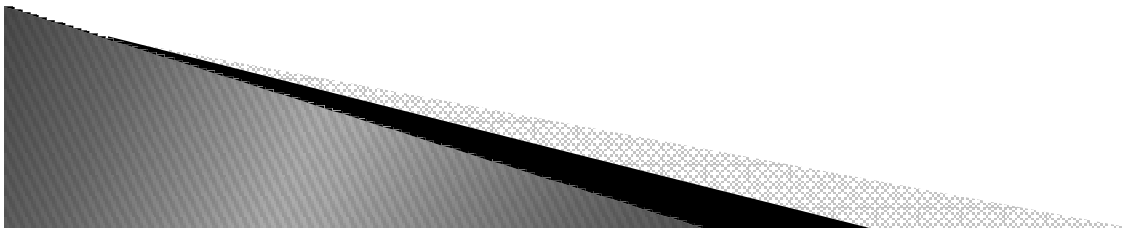


Test-Driven Development (TDD)



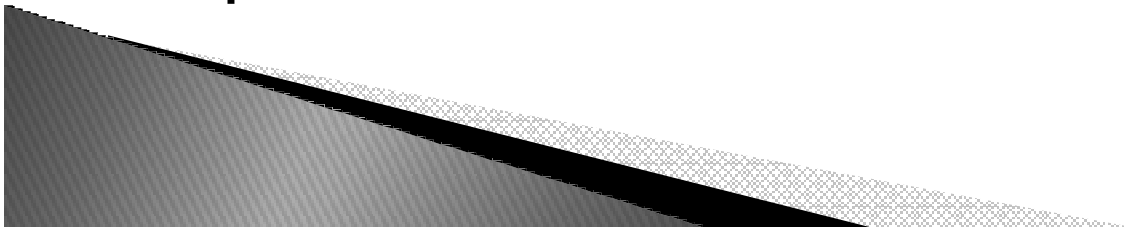
TDD – Test Driven Development

- ▶ O que é?
- ▶ -TDD (Test Driven Development) é um técnica de programação ágil.
Com TDD especificamos nosso software em detalhes no momento que vamos escrevê-lo criando testes executáveis e rodando-os de maneira que eles mesmos testem nosso software.
- ▶ Serve para diagnosticar precocemente “bugs” que podem vir a ser problemas na finalização do projeto.



TDD – Test Driven Development

- ▶ Genericamente:
- ▶ No desenvolvimento de um software → começar escrevendo seu teste
- ▶ Uma classe que vai testar o software
- ▶ Assim, deve-se testar o software e o remodelar até que os testes não falhem mais
- ▶ Então o TDD pode ser definido como uma técnica de programação onde todo o código produzido é criado em resposta a um código que falhou.



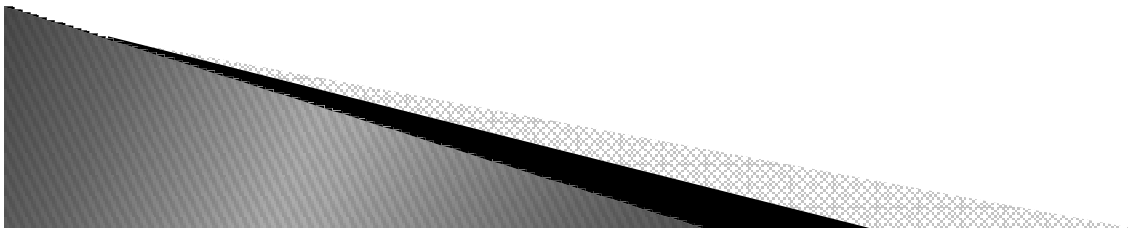
TDD – Test Driven Development

- ▶ Como funciona?
- ▶ Escrever um teste,
- ▶ rodar este teste até que algo falhe,
- ▶ escrever o código fonte mais simples possível (para passar neste teste),
- ▶ escrever o teste (melhorando para cobrir mais funções do código fonte do produto),
- ▶ testar até que algo falhe,
- ▶ reescrever o código para que ele passe no teste....
- ▶ Permanecer nesse ciclo até o software estar completo.



TDD – Test Driven Development

- ▶ ...:::Vantagens:::...
- ▶ –Simplicidade;
- ▶ –Confiança no código;
- ▶ –Documentação; e
- ▶ –Facilidade com refactorings.



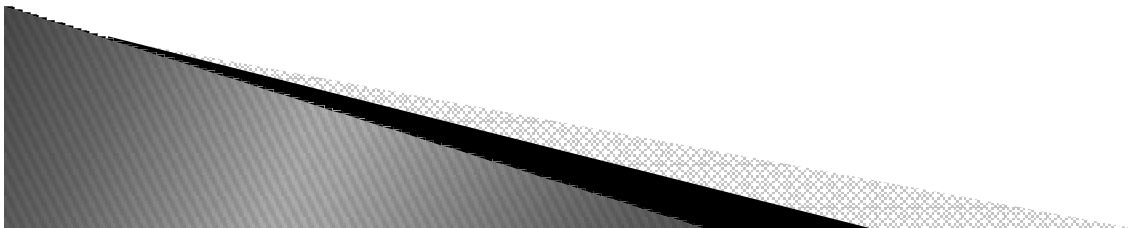
TDD – Test Driven Development

- ▶ Incentiva a simplicidade:
- ▶ Solução surge pouco a pouco → não se perde tempo com o que não será usado em seguida.
- ▶ Expressões:
 - “You Ain’t Gonna Need It ”:”Voce não vai precisar disso”
 - “Keep It Simple, Stupid “:”deixar simples,patético”
 - São recorrentes quando se está programando orientado a testes.



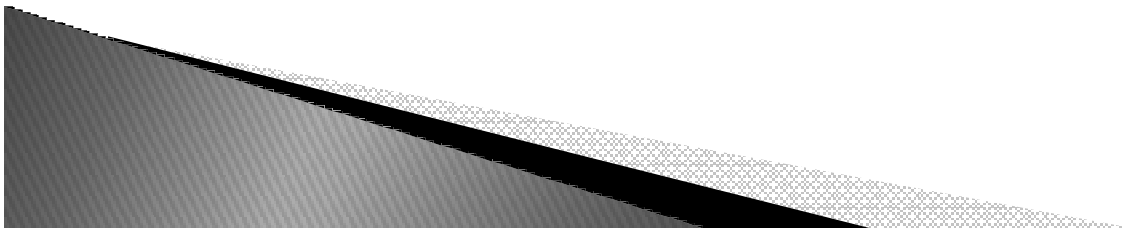
TDD – Test Driven Development

- ▶ Aumenta a confiança no código:
- ▶ Sistemas funcionam de maneira estável → testes foram utilizados durante sua criação e validam tudo o que foi criado.
- ▶ E se ainda assim algum erro surgir, um novo teste é criado para produzi-lo e garantir que depois de solucionado ele não irá se repetir.



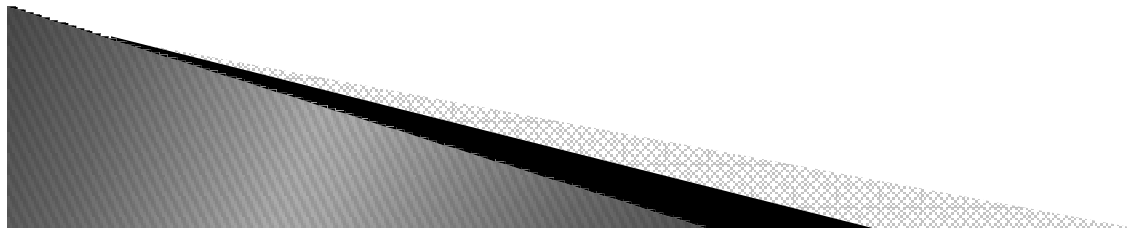
TDD – Test Driven Development

- ▶ Ajuda como documentação:
- ▶ Testes bem definidos são mais fáceis de ler que o código fonte em si
- ▶ Eles são uma fonte eficiente para entender o que o software faz.
- ▶ A documentação sempre estará atualizada com a aplicação.
- ▶ O teste aponta a funcionalidade do software.



TDD – Test Driven Development

- ▶ Facilita refactorings:
- ▶ Quanto mais testes existem no sistema, maior é a segurança para fazer refactorings.
- ▶ Um erro causado por algum refactoring dificilmente vai passar despercebido quando um ou mais testes falharem após a mudança.



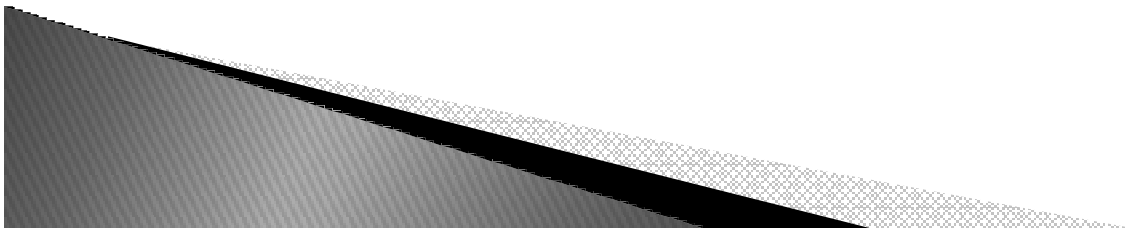
TDD – Test Driven Development

- ▶ Em TDD, um teste é um pedaço de software. A diferença entre teste e o código que está sendo produzido é que os testes têm 2 funções principais:
 - De especificação: definir uma regra que o software deve obedecer.
 - De validação: verificar que a regra é obedecida pelo software.
- ▶ Geralmente os testes são criados com algum framework do tipo xUnit (JUnit, NUnit Test::Unit etc) , mas também podem ser feitos num nível de funcionalidades (através de softwares como o FitNesse e Selenium) . Estas ferramentas servem basicamente para organizar os testes e facilitar na criação das verificações.



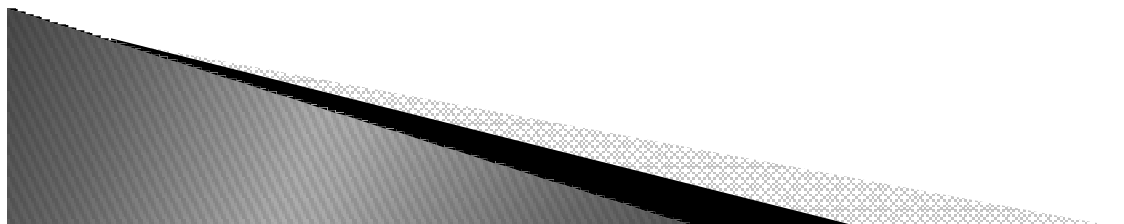
TDD – Test Driven Development

- ▶ Processo de criação de desenvolvimento orientado a testes :
- ▶ –3 passos são repetidos até que não se consiga pensar em novos testes, o que indica que a funcionalidade está pronta.



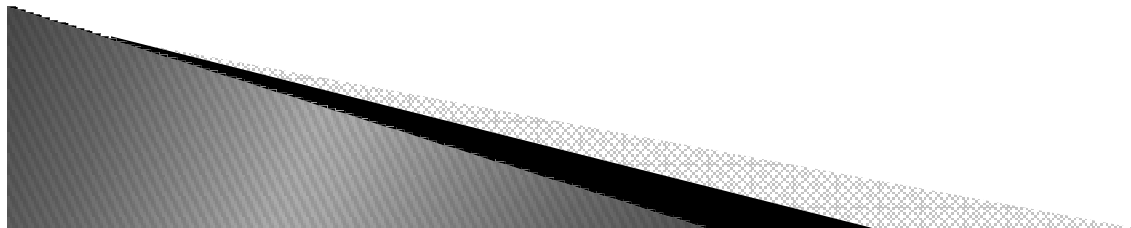
TDD – Test Driven Development

- ▶ **1 – Escrever um teste que falhe.**
 - Pensar no que o código deve fazer
 - Definir quais são as verificações que precisam ser feitas.
 - Não há um limite no número de testes, portanto quanto menos funções cada teste descrever ou verificar, melhor.
 - Não é preciso se preocupar se a classe ou método ainda não existe.
 - Pensar primeiro no teste e só depois que este estiver pronto, criar o esqueleto de código necessário para que ele compile e falhe ao rodar.



TDD – Test Driven Development

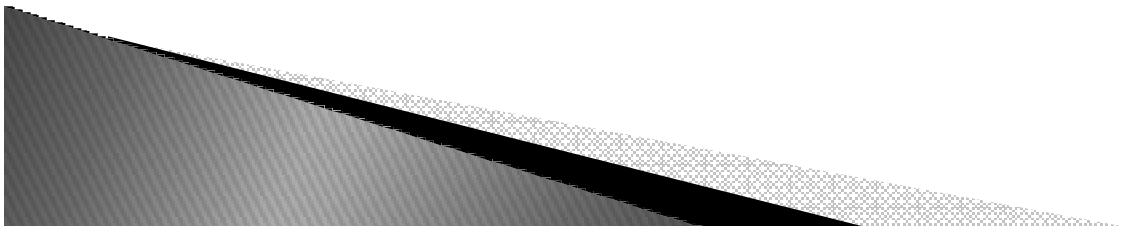
- ▶ 2– Fazer o teste passar.
 - Escrever o mínimo de código para que o teste passe.
 - Controlar o instinto natural do programador de tentar prever tudo que o código vai fazer e apenas fazer o teste passar.
 - Mesmo que tenha certeza que o código deve fazer mais coisas, fazer os testes passarem deve ser a única preocupação nesse estágio.



TDD – Test Driven Development

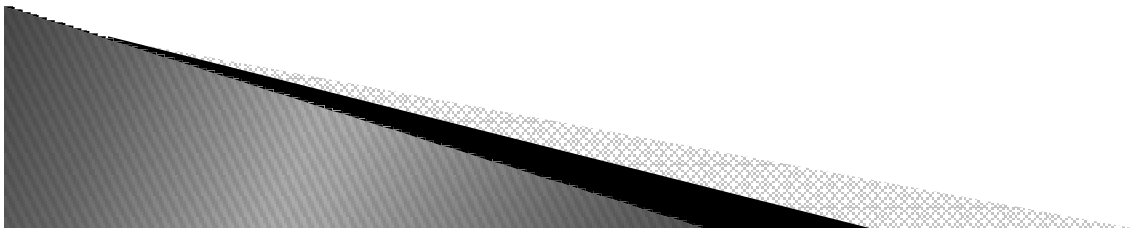
▶ 3– Refatorar .

- Se o teste passar → verificar o que pode ser melhorado no código.
- Agora é a hora de melhorar o código e remover as duplicações, lembrando que os testes devem continuar passando.



Exemplo

- ▶ Criar uma classe que diga se dada uma data, essa data se encontra em determinado padrão.



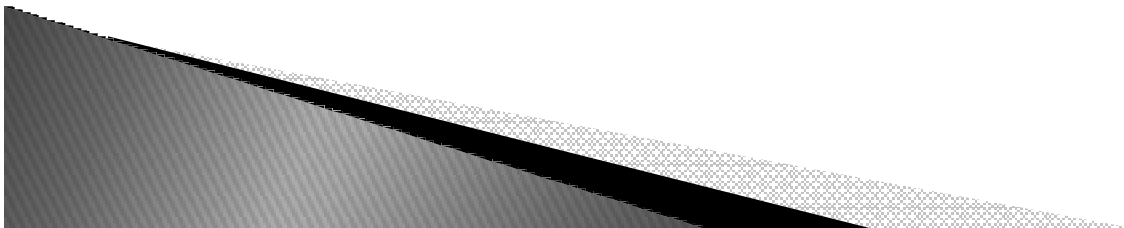
Exemplo

- ▶ Primeiro passo:
 - Criar um teste vazio que falha

```
import unittest
class FooTests(unittest.TestCase):

    def testFoo(self):
        self.failUnless(False)

def main():
    unittest.main()
if __name__ == '__main__':
    main()
```



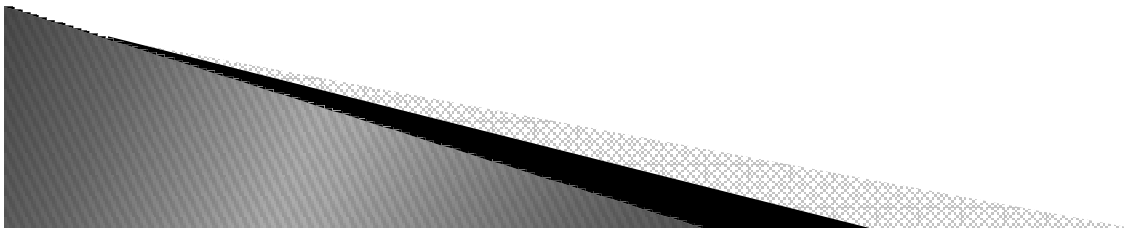
Exemplo

- ▶ Continua-se codificando o teste, mesmo sem a classe a ser testada estar implementada
 - Agora já se sabe um pouco mais sobre o problema e o método testFoo foi alterado para testMatches

```
import unittest
import datetime
from DatePattern import DatePattern

class FooTests(unittest.TestCase):
    def testMatches(self):
        p = DatePattern(2004, 9, 28)
        d = datetime.date(2004, 9, 28)
        self.failUnless(p.matches(d))

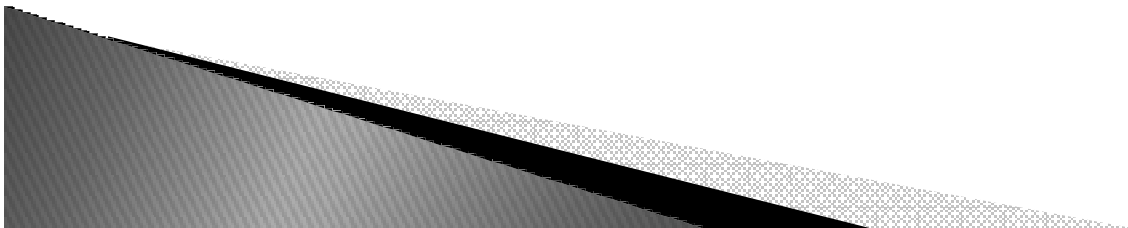
def main():
    unittest.main()
if __name__ == '__main__':
    main()
```



Exemplo

- ▶ Nesse ponto, temos que fazer o teste rodar...
 - Então vamos criar a classe a ser testada

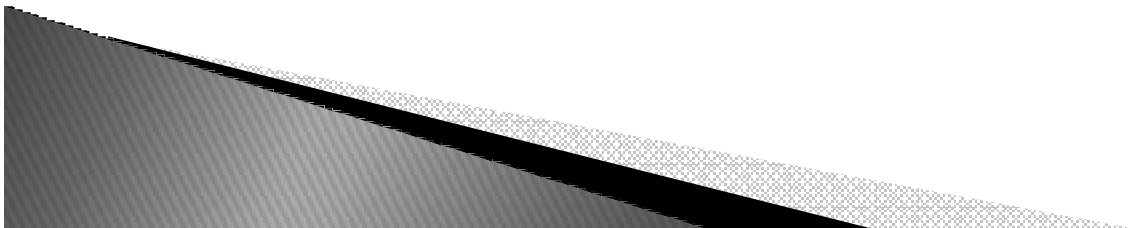
```
class DatePattern:  
    def __init__(self, year, month, day):  
        pass  
  
    def matches(self, date):  
        return True
```



Exemplo

- ▶ O código de matches na classe testada não tem valor
 - Como justificar a necessidade de uma real implementação?
 - Com outro teste !!

```
def testMatchesFalse(self):  
    p = DatePattern(2004, 9, 28)  
    d = datetime.date(2004, 9, 29)  
    self.failIf(p.matches(d))
```



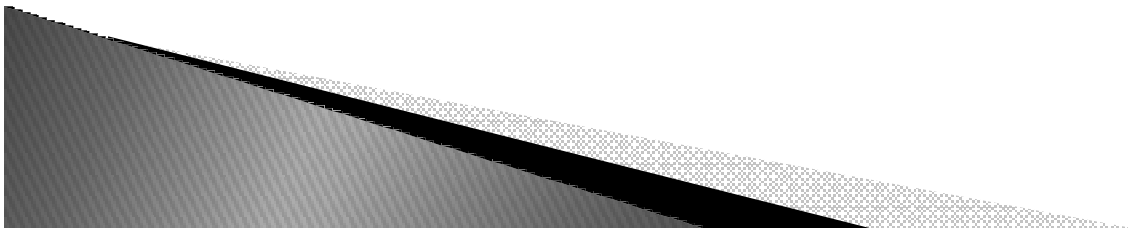
Exemplo

- ▶ Para que o teste anterior rode alteramos a função matches

```
import datetime

class DatePattern:
    def __init__(self, year, month, day):
        self.date = datetime.date(year, month, day)

    def matches(self, date):
        return self.date == date
```

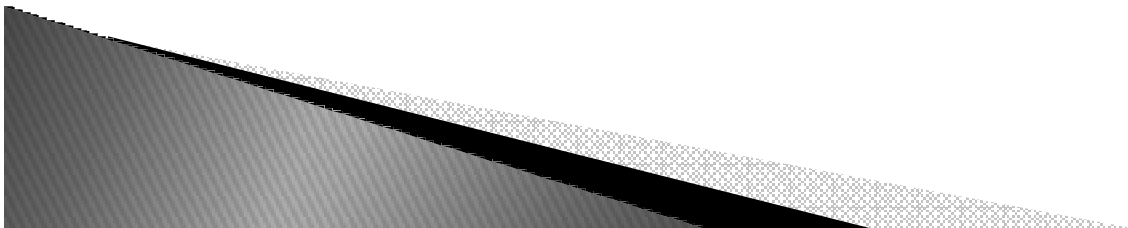


Exemplo

- ▶ Nesse ponto os dois testes passam e parece que a classe `DatePattern` é apenas um encapsulamento da classe `date`
- ▶ Qual a justificativa para a classe `DatePattern` ?

```
def testMatchesYearAsWildcard(self):  
    p = DatePattern(0, 4, 10)  
    d = datetime.date(2005, 4, 10)  
    self.failUnless(p.matches(d))
```

- ▶ Esse teste falha !!

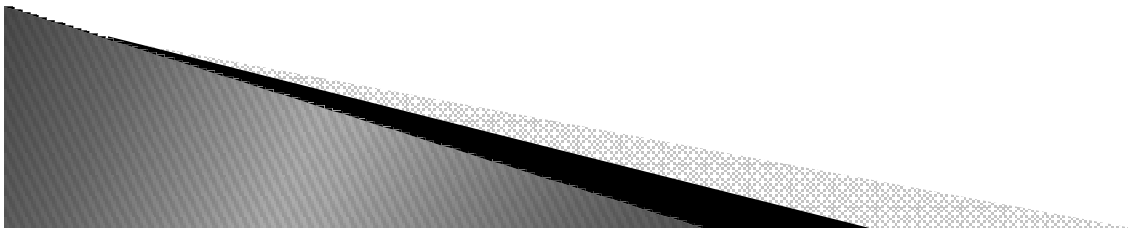


Exemplo

▶ Hora de fazer o novo teste passar

```
class DatePattern:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def matches(self, date):
        return ((self.year and self.year == date.year or True)
                and self.month == date.month
                and self.day == date.day)
```



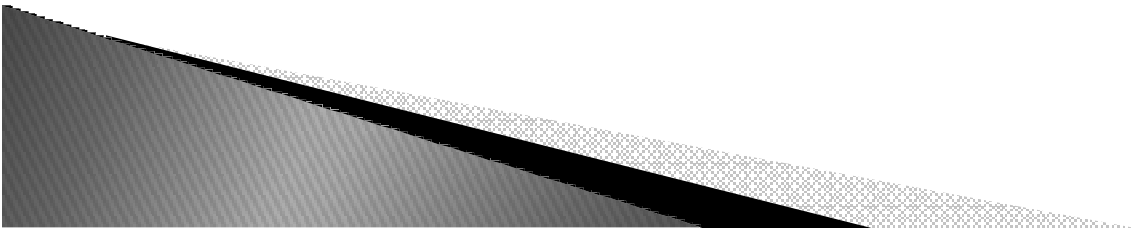
Exemplo

- ▶ Adicionando novo método para checagem do mês

```
def testMatchesYearAndMonthAsWildCards(self):  
    p = DatePattern(0, 0, 1)  
    d = datetime.date(2004, 10, 1) self.failUnless(p.matches(d))
```

- ▶ Não Passou !!
- ▶ Consertar

```
def matches(self, date):  
    return ((self.year and self.year == date.year or True)  
            and (self.month and self.month == date.month or True)  
            and self.day == date.day)
```

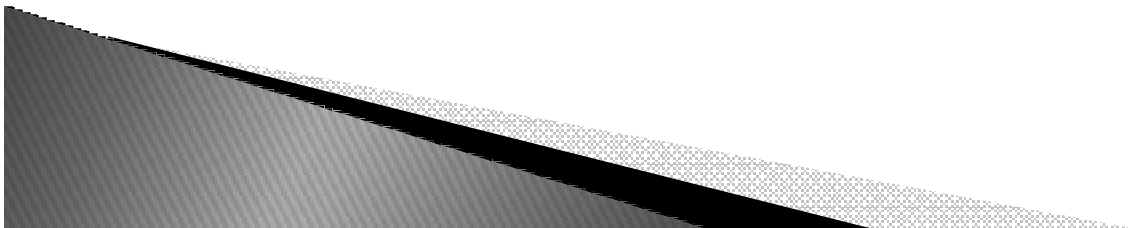


Exemplo

▶ Código do método matches após os testes

```
def matches(self, date):  
    return ((self.year and self.year == date.year or True)  
            and (self.month and self.month == date.month or True) and  
            (self.day and self.day == date.day or True)  
            and (self.weekday and self.weekday == date.weekday() or  
                True))
```

▶ Hora do Refactoring



Exemplo

- ▶ Refactoring (método matches muito mais claro agora não concorda?)

```
def matches(self, date):  
    return (self.yearMatches(date)  
            and self.monthMatches(date)  
            and self.dayMatches(date)  
            and self.weekdayMatches(date))
```

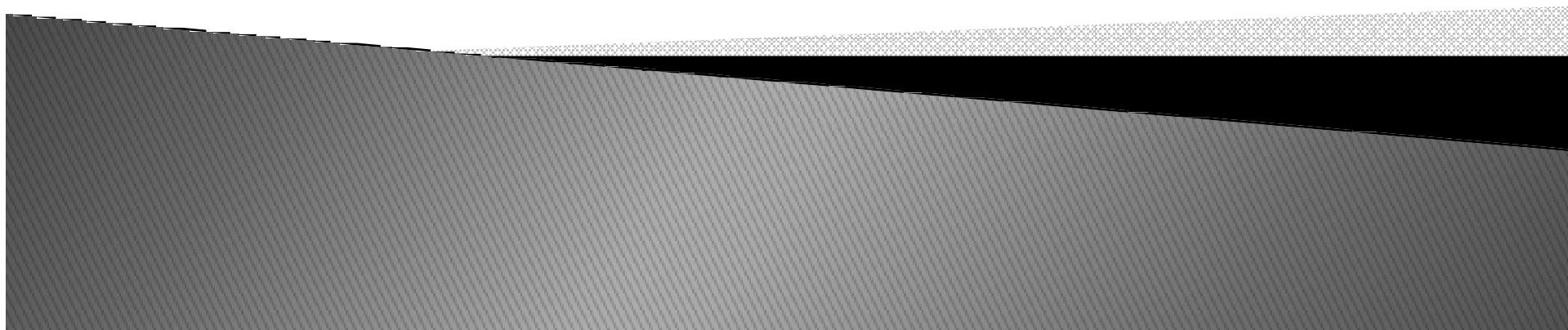
```
def yearMatches(self, date):  
    if not self.year: return True  
    return self.year == date.year
```

```
def monthMatches(self, date):  
    if not self.month: return True  
    return self.month == date.month
```

```
def dayMatches(self, date):  
    if not self.day: return True  
    return self.day == date.day
```

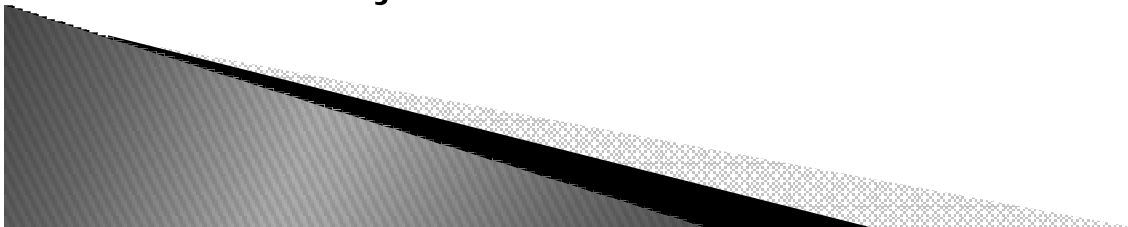
```
def weekdayMatches(self, date):  
    if not self.weekday: return True  
    return self.weekday == date.weekday()
```

Exeções



Exceções

- ▶ Exceções ocorrem quando certas situações excepcionais ocorrem em seu programa.
- ▶ Exemplo, quando você está tentando abrir um arquivo e este não existe ou quando você apaga o arquivo enquanto o programa está rodando.
- ▶ Este tipo de situação é manipulada usando exceções.



Exceções

- ▶ Try.....

- ▶ Except

- ▶ Finally

