Fundamentals of Testing

What is the scope of software testing?

How does testing fit in?

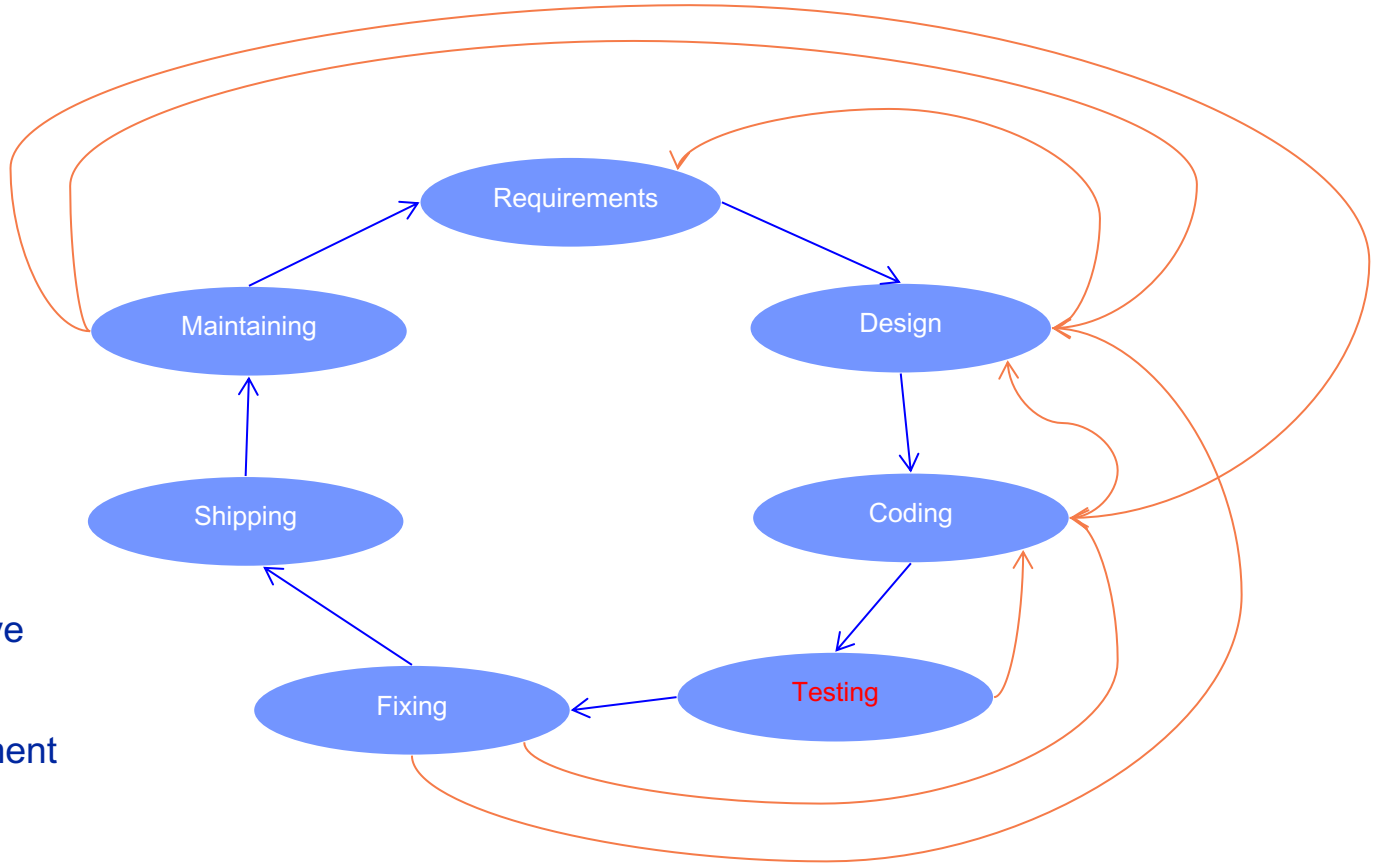
Development lifecycle: (very?) Classic view of testing (**testing the produced code post-compilation**)

- This type of testing has quite a complex interplay with other lifecycle activities – as can be seen.

- Traditionally the role of the software testers or the software test team / department.

- A complex and expensive activity in its own right.

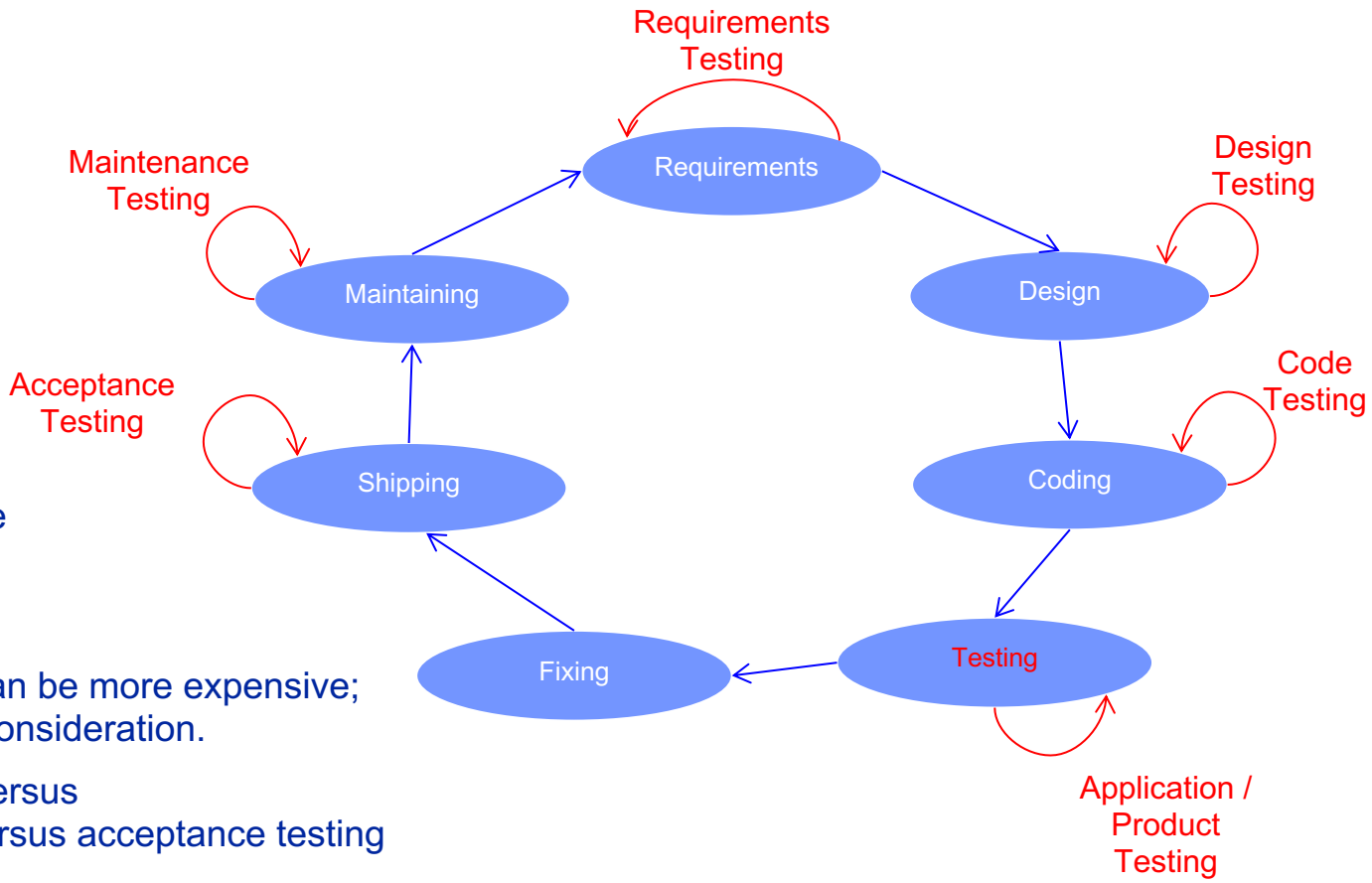
- Agile software development is different again.



- But this is just one aspect of the software testing domain.

How does testing fit in?

Development lifecycle: Broader scope of testing domain (**testing throughout the lifecycle**)



- Testing is everywhere

- Testing is expensive

- Not testing enough can be more expensive; there is an economic consideration.

- Note on integration versus system/qualification versus acceptance testing

- This module is concerned with all of these different types of testing.

Why is testing necessary?

Why testing is important?

After spending $130 million on it's problematical health insurance exchange, one of the 14 U.S. states that opted to create their own health insurance exchange (rather than utilize a federal-government-provided exchange) hired a new contractor in April 2014 to redo the site. Among the many problems with the initial site since it went live in October 2013, reportedly **hundreds of enrollees receiving enrollment information with names and birth dates of other enrollees**. It was estimated the **revamping would cost another $60 million**. Additionally, the primary contractor and subcontractor for the site are embroiled in a lawsuit with one another and at last report were in arbitration. Eventually it was reported that **the prime contractor agreed to pay $45 million to the government to avoid a lawsuit**.

7 out of 10 IT projects fail and the reasons for failure can often be attributed to poor software testing.

Software testing can represent c.40% of development costs within a project.

There are significant cost, time and quality benefits to getting software testing right.

Unfortunately, organisations often struggle to attain the desired balance between test coverage and time to delivery.

Certain test activities are one of the final activities.

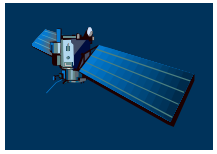
Because testing is one of the last tasks prior to customer delivery, well intentioned and appropriately planned testing activities are often subject to significant time pressure.

In this instance – it is not good to be last.

Software is a Skin that Surrounds Our Civilization

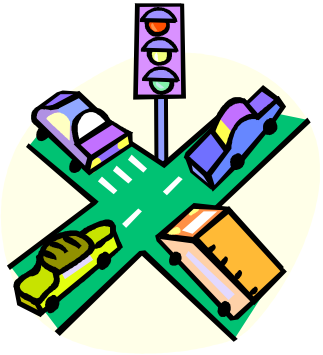


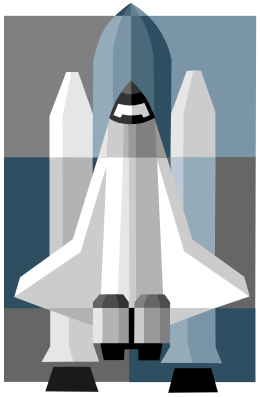




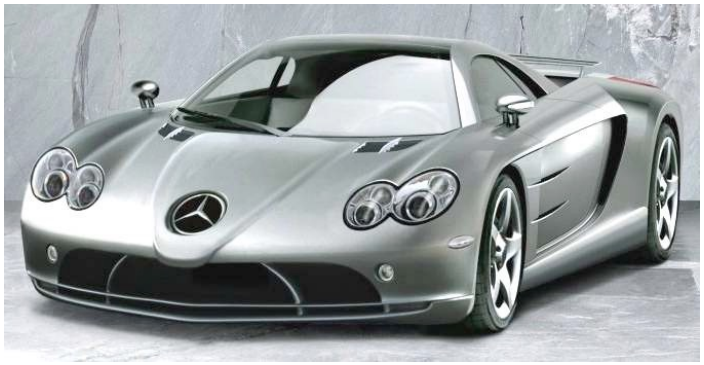






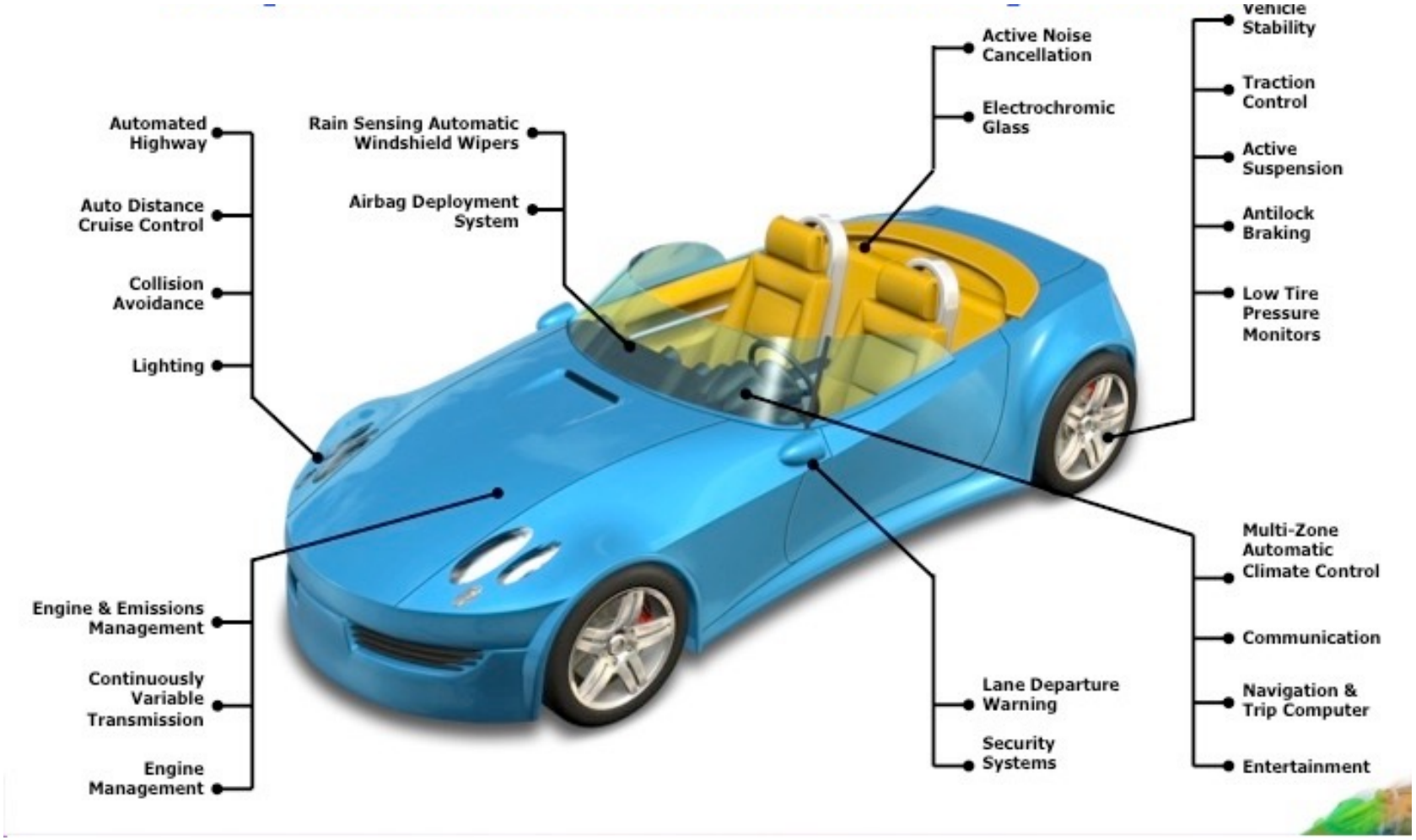
Trivia Quiz: Lines of code



>20 million

6.5 million

Think about what is really inside a car



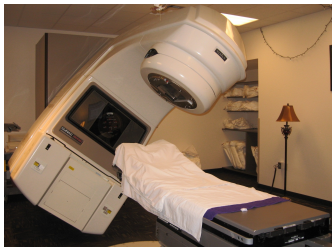
Failures in Production Software

- NASA's Mars lander, September 1999, crashed due to a unit integration fault—over $50 million US !
- Huge losses due to web application failures
  - Financial services : $6.5 million per hour
  - Credit card sales applications : $2.4 million per hour
- In Dec 2006, amazon.com's BOGO offer turned into a double discount
- 2007 : Symantec says that most security vulnerabilities are due to faulty software
- Stronger testing could solve most of these problems

World-wide monetary loss due to poor software is staggering

Types of failure

Examples of famous (or infamous) failures:



- Therac-25 (1985-1987)
- London Ambulance System (1992)
- Denver baggage handling system (1990's)
- Taurus (1993)
- Ariane 5 (1996)
- E-mail buffer overflow (1998)
- USS Yorktown (1998)
- Mars Climate Orbiter (September 23rd, 1999)

Why Does Testing Matter?

- NIST report, "The Economic Impacts of Inadequate Infrastructure for Software Testing"
  - Inadequate software testing costs the US alone between $22 and $59 billion annually
  - Better approaches could cut this amount in half
- Major failures: Ariane 5 explosion, Mars Polar Lander, Intel's Pentium FDIV bug
- Insufficient testing of safety-critical software can cost lives:
  - THERAC-25 radiation machine: 3 dead
- We want our programs to be reliable
  - Testing is how, in most cases, we find out if they are


Ariane 5: exception-handling bug : forced self destruct on maiden flight (64-bit to 16-bit conversion: about 370 million $ lost)


Mars Polar Lander crash site?

Northeast Blackout of 2003

- 508 generating units and 256 power plants shut down
- Affected 10 million people in Ontario, Canada and 40 million people in 8 US states
- Financial losses of $6 Billion USD
- The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system

More recent software blunders!

Amazon 1p sales bonanza Christmas Glitch 2014.

Knight Capital Group $440M loss, 2012

Nissan Motor company, 3.53M vehicles recalls (multi-sensor system malfunction).

2016 Hitomi Satellite, $273M loss

Heartbleed OpenSSL crypto library vulnerability, 2014.

Smaller scale – defects not publicised

- Software companies have to maintain their software, which is costly.
- Potential defects are reported – as issues - from users / clients
- Issues are prioritised, examined and resolved.
- Some issues are the result of defects.
- Triage is required.

Causes of software defects

A human being can make an error (mistake), which produces a defect (fault, bug) in the code, in software or a system, or in a document

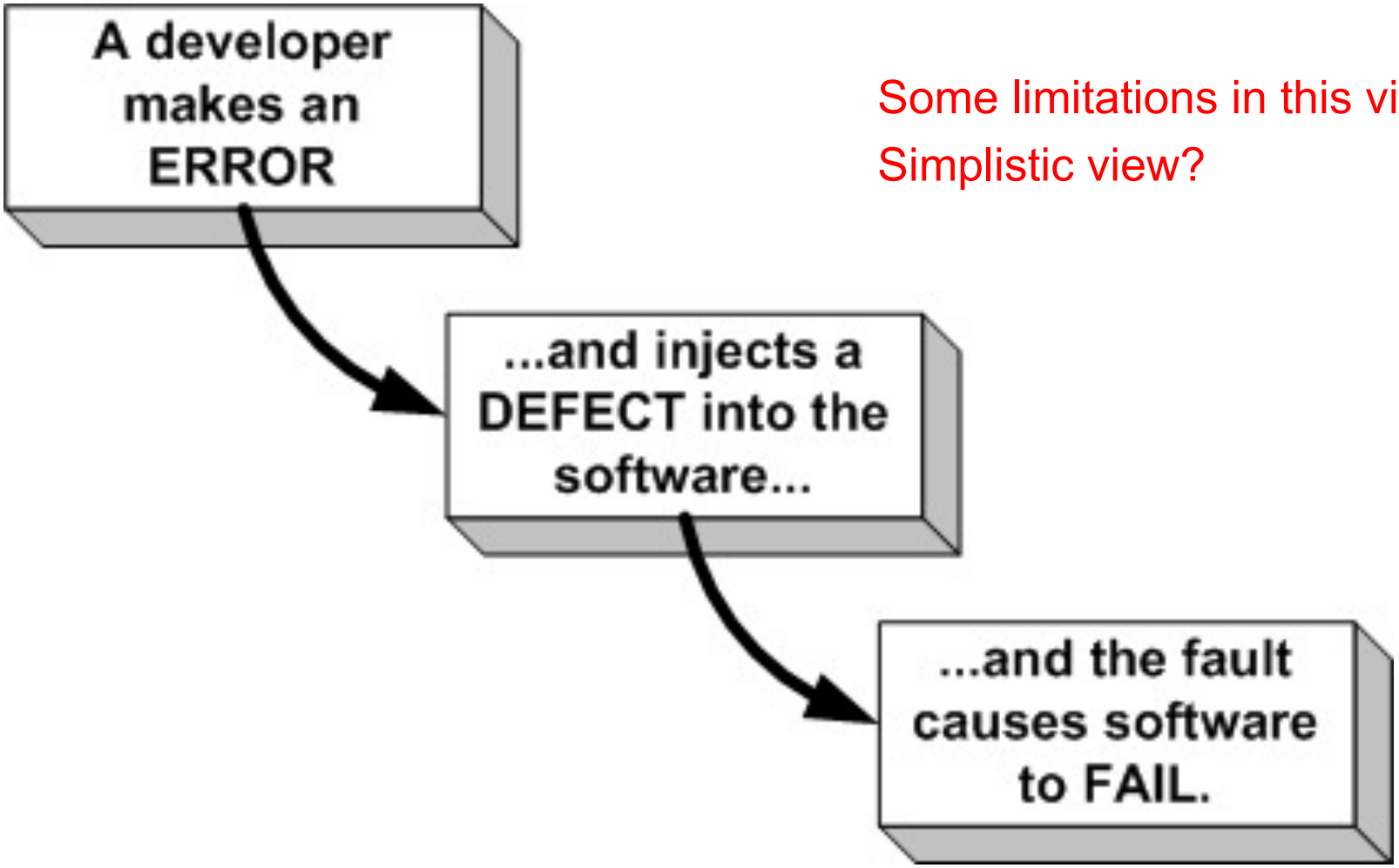
If the code is executed, the system will fail, causing a failure

Defects in software, systems or documents may result in failures, but not all defects do so

Failures can be caused by environmental conditions as well:

- radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing hardware conditions.

Errors, defects and failures



Some limitations in this view?

Simplistic view?

A failure is…

"A failure is present if a warrantable (user) expectation is not fulfilled adequately."

Examples of failures include: products that are too hard to use, or too slow, even if they still fulfill the functional requirements. i.e. when a user experiences a problem.

A failure is: a deviation of the software from its expected delivery or service

A failure occurs when software does the 'wrong' thing

Most of the time software does the right thing

Software defects cause software failures when the program is executed with a set of inputs that expose the defect.

Terminology…

Sometimes a failure will be called a “problem”, “incident”
“issue”

A defect is…

A manifestation of human error in software

Defects may be caused by requirements, design or coding errors

They are discovered either by inspecting code or by inferring their existence from software failures.

Terminology…

Sometimes a defect will be called a “bug” or a “fault”

An error is…

A human action producing an incorrect result

When programmers make errors they introduce faults to program code

Errors are not

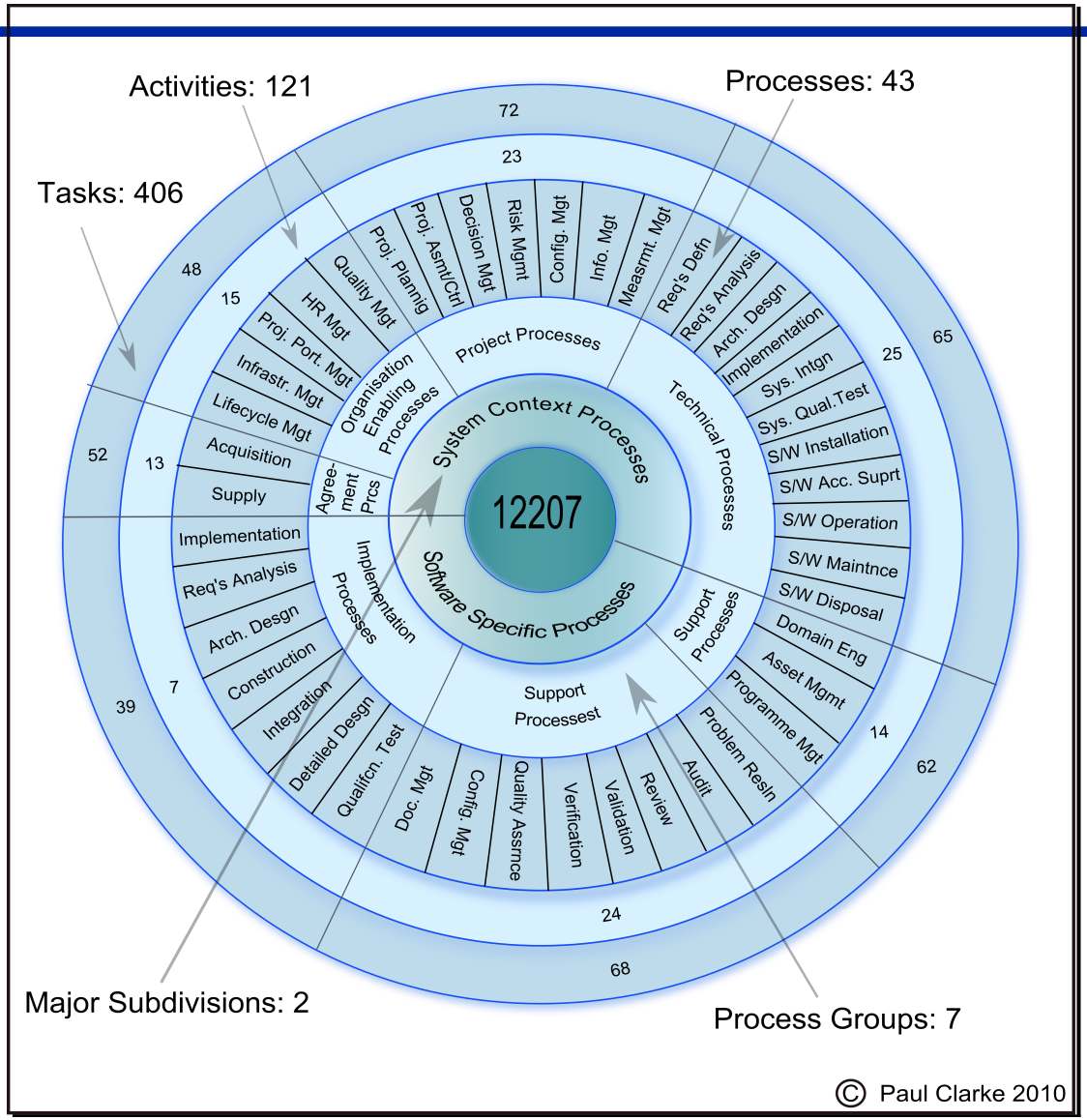
- Errors are not just accidents or mistakes
- Errors are not cured by "just being more careful"
- Errors are not necessarily an act of incompetence

Errors are inevitable in a complex activity.

Fundamentals continued…

- Software complexity
- Role of testing in development, maintenance and operations
- Testing and…..

How complex is software development?



Role of testing in development, maintenance and operations

Rigorous testing of systems and documentation can help to:

- reduce the risk of problems occurring in an operational environment
- and contribute to quality

Software testing may also be required:

- to meet contractual or legal requirements,
- or industry-specific standards.

Testing and confidence

If we buy a software package

- we may believe it works, but...
- a test gives us the confidence that it will work

When we buy a car, cooker, off-the-peg suit

- we assume they work, but do they work for me?
- we still test them

When we buy a kitchen, haircut, bespoke suit

- we were involved in the requirements
- and we test them.

Testing and contractual requirements

When we buy custom-built software, a contract will usually state

- the requirements for the software
- the price of the software
- the delivery schedule and acceptance process

We don't pay the supplier until we have received and acceptance tested the software

Acceptance tests help to determine whether the supplier has met the requirements.

Testing and other requirements

Software requirements may be imposed:

- laws governing our industry must be obeyed
- industry regulations must be adhered to
- our customers may insist we are compliant (e.g. Year 2000, EMU etc)
- industry standards (e.g. safety-critical software)

When we write, change or buy software, we have to provide evidence that we comply

Testing provides most of that evidence.

Testing and quality

- Testing can measure quality of a product and indirectly, improve its quality.
- Testing measures quality
  - if we assess the rigour and number of tests and if we count the number of faults found...
  - we can make an objective assessment of the quality of the system under test
- Testing improves quality
  - tests aim to identify defects
  - if defects are found, we can improve the quality of the software.

Fundamentals continued…

- How much testing is enough?

How much testing is enough?

A perplexing question

There's no upper limit

If we don't test enough, we get blamed for faults in production

The tester can't 'win'.

Think of managing a deficit or a surplus.

How much testing is enough?

There are an infinite number of tests we could apply and software is never perfect

So how much testing is enough?

Objective coverage measures can be used:

- standards may impose a level of testing
- test design techniques give an objective target

But all too often, time is the limiting factor

- so we may have to rely on a consensus view to ensure we do at least the most important tests.

Complete Testing is not practically feasible (1)

Consider a simple little program that consists of a loop in which there are 5 possible execution paths from the start of the loop to the end of the loop.

If the maximum number of iterations of the loop is 20 then there is: $5^{20} + 5^{19} + 5^{18} + \ldots + 5^{1}$ different execution traces. That's about 100 quadrillion different execution traces.

If each test is done manually and takes 5 minutes to test each trace then the full test takes 1 billion years.

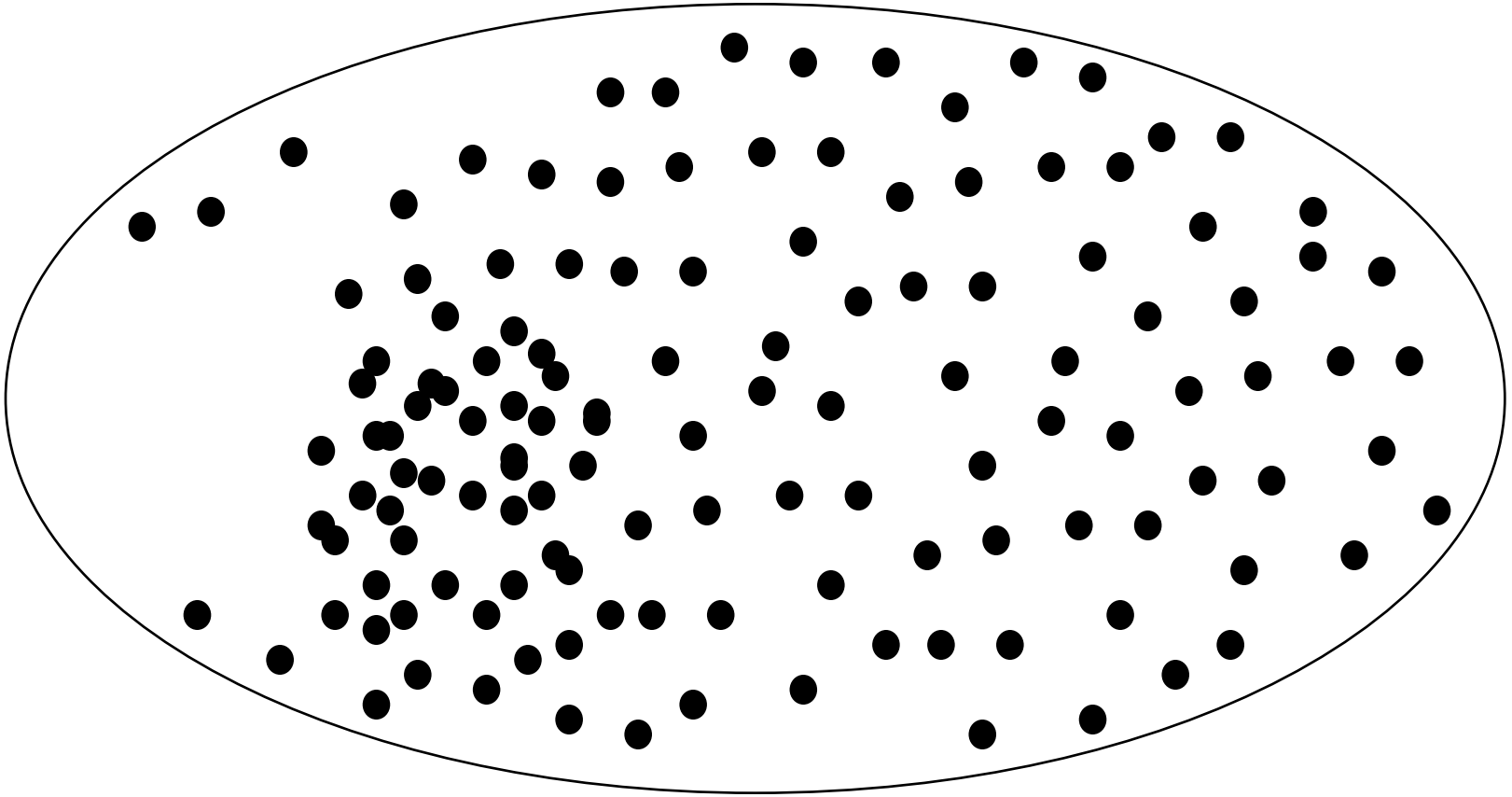
If automated so that each test takes 5 microseconds, then it would take 19 years!

Complete Testing is not practically feasible (2)

Complete testing is nearly impossible because of the following reasons:

- The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid inputs and invalid inputs.
- The program may have a large number of states.
- The design issues may be too complex to completely test (especially for larger systems)
- It may not be possible to create all possible execution environments of the system

The bugs that lurk in our systems



Fundamentals continued…

- Where are the bugs?

Where are the bugs?

Of course, if we knew that, we could fix them and go home!

Experience tells us

- bugs are sociable! - they tend to cluster
- some parts of the system will be relatively bug-free (bought-in or unchanged code)
- bug fixing and maintenance are error-prone – and some changes cause unintended faults.

Testing as a fishing net

Size of the net reflects size of the test

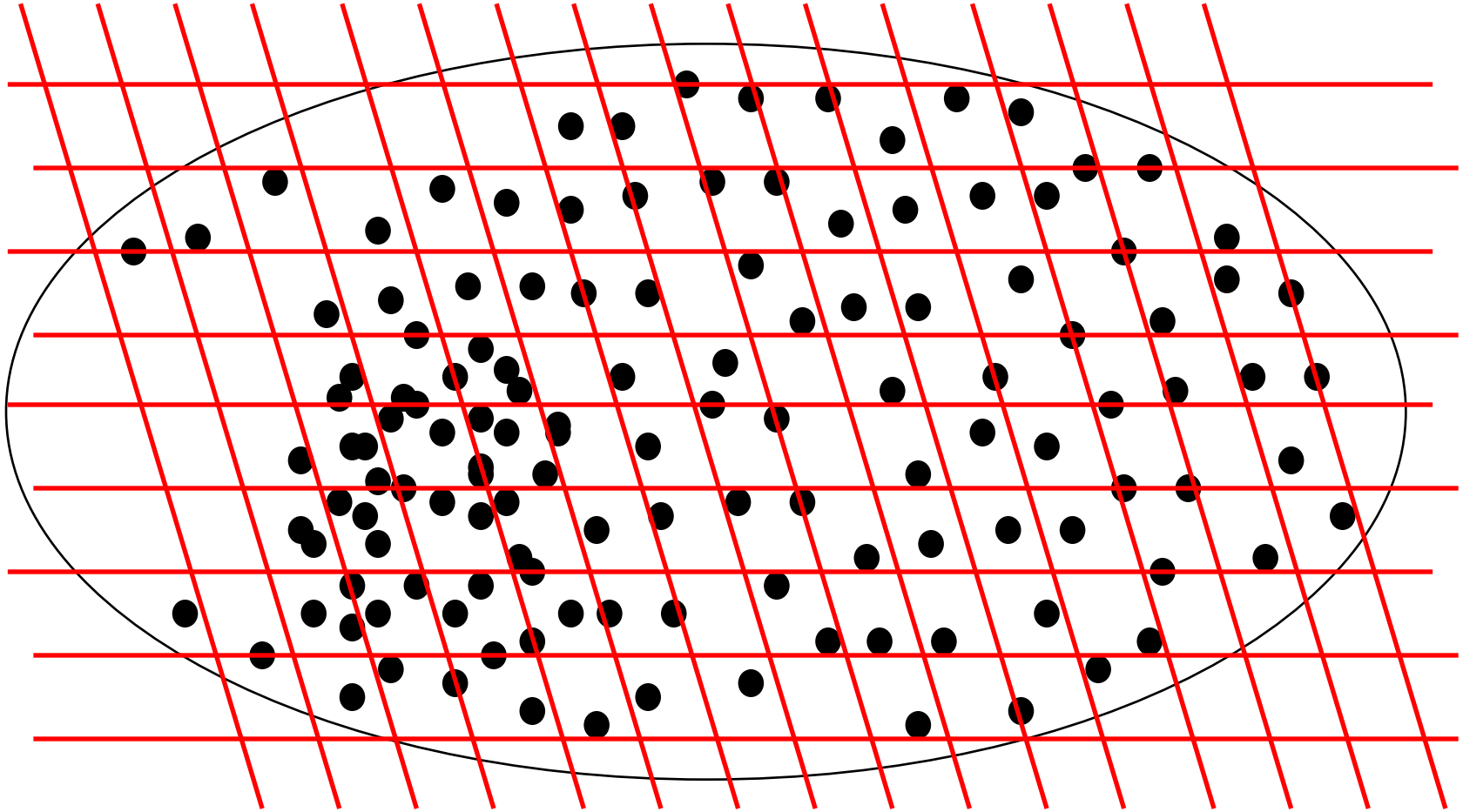
Density of the mesh reflects test depth or thoroughness

Large nets, large mesh to give overall confidence

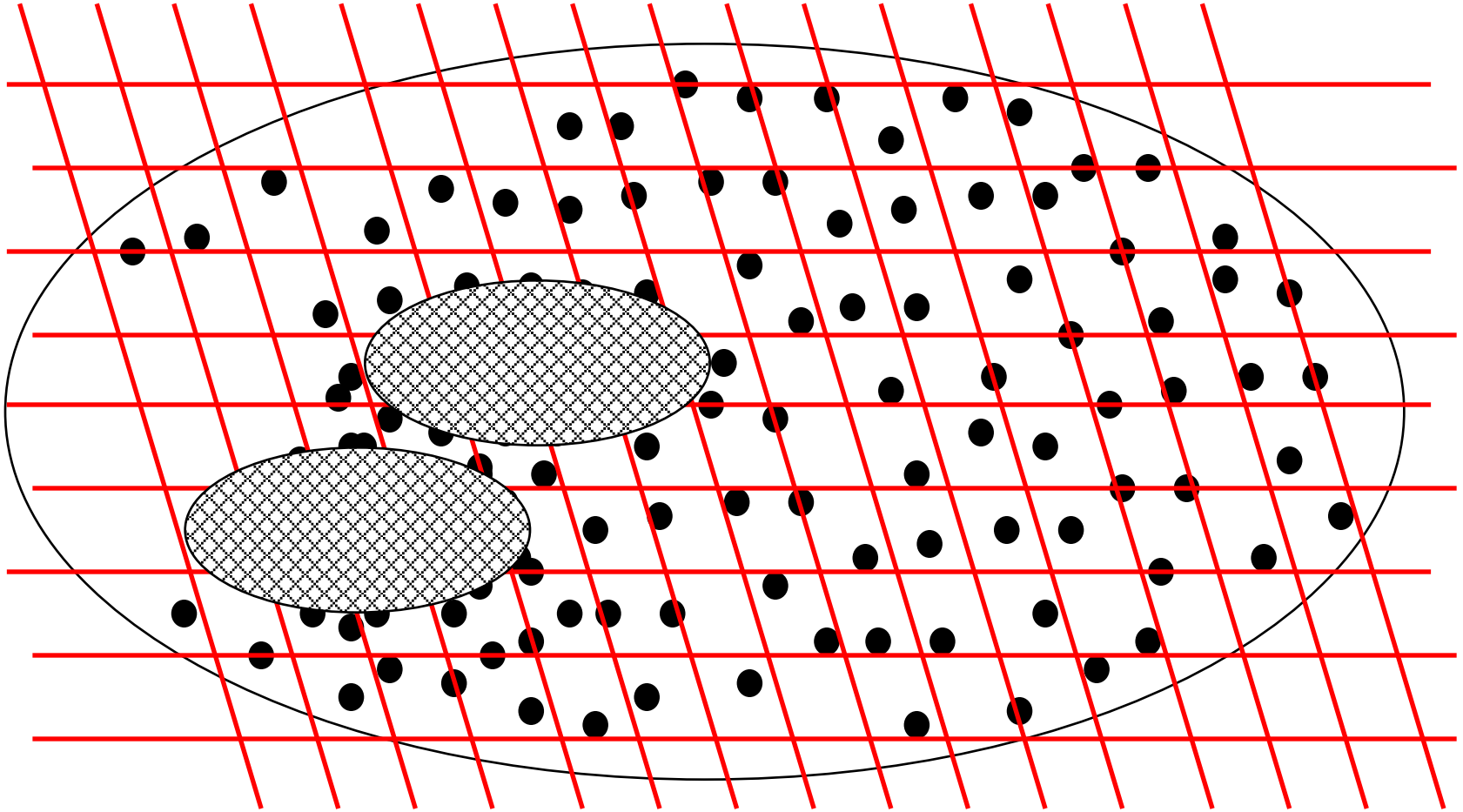
Small nets, fine mesh to find bugs

- use in areas that are critical to make sure the bugs aren't there
- use in areas where we expect lots of bugs.

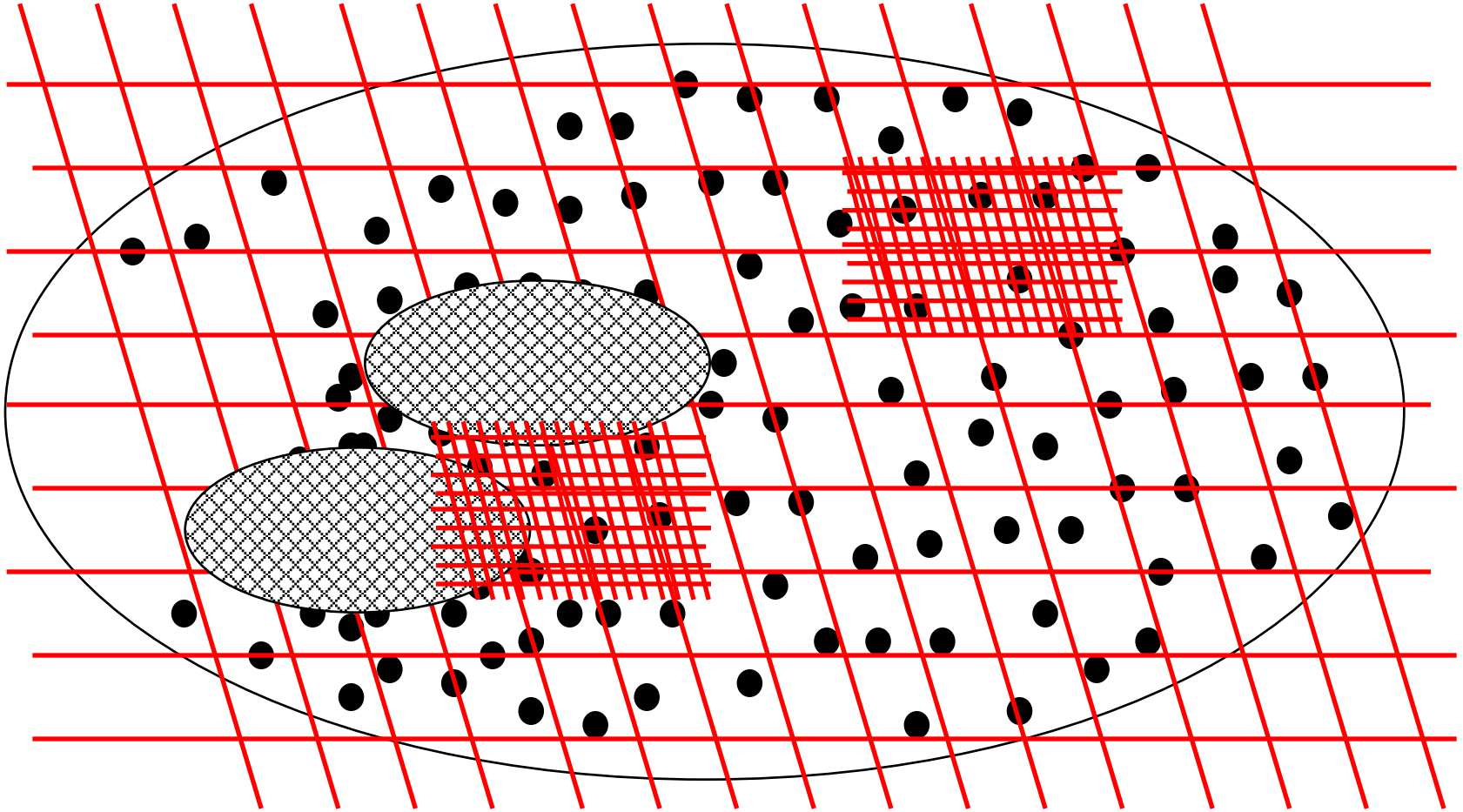
Coverage of business-oriented tests



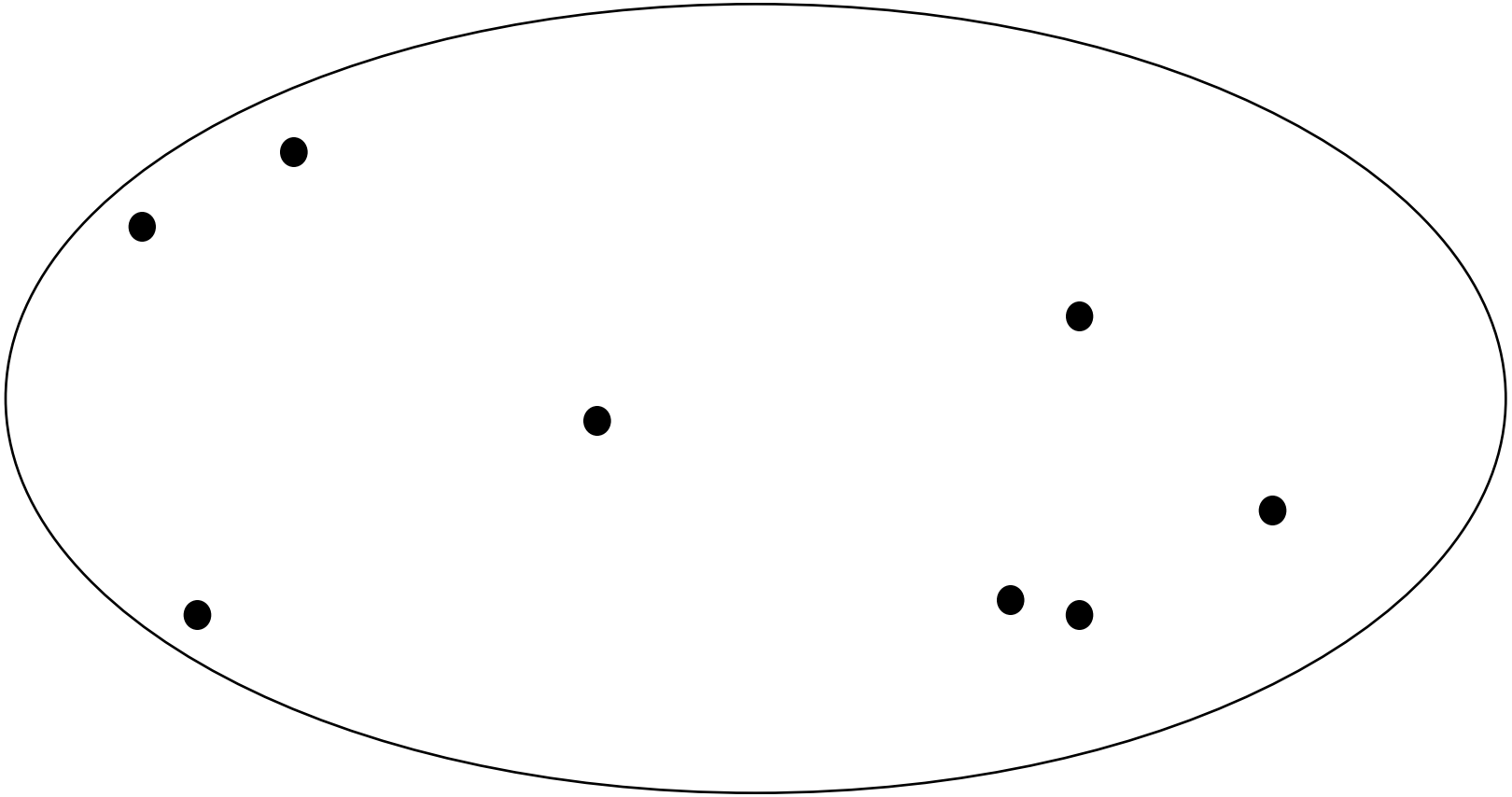
Rigorous tests of the riskiest parts of the system



More tests of the business critical parts of the system



What about the bugs we don’t find?



What about the bugs we don’t find?

- If not in the business critical parts of the system - would the users care?
- If not in the system critical parts of the system - should be low impact
- If they are in the critical parts of the system
  - should be few and far between
  - should be pushed to the domain of obscure usage.

Fundamentals continued…

- What is testing?

What is testing?

Testing

The systematic exploration of a component/system with the main aim
of finding and reporting defects.

Testing rigorously examines the component/system behaviour and reports defects found.

Tests are repeated to ensure that defect corrections have been effective.

Debugging

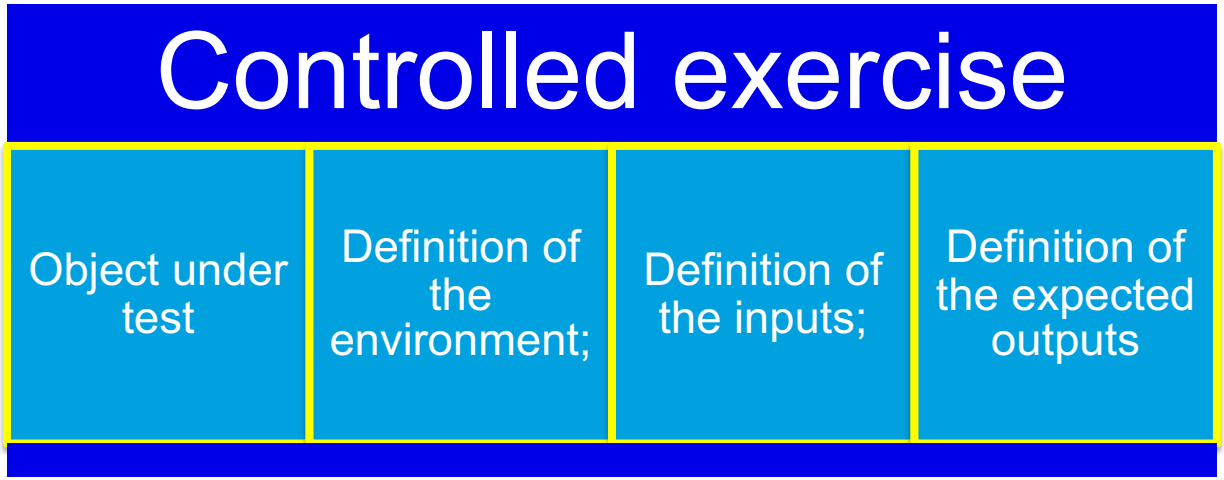
A process undertaken by developers to identify the cause of defects in code and undertake corrections.

Debugging is done first to ensure that the component or system is at a level
to enable rigorous testing.

Debugging can be used to understand the root cause of observed failures.

What is testing?

- What is a Test?



- Thus once a Test is run, you'll get
  - An actual result / output
  - A decision on its success

Testing isn't just running tests

planning and control

choosing test conditions

designing test cases

preparing expected results

checking actual results against expected results

evaluating completion criteria

reporting on the testing process and system under test

and closing the test phase.

Test Objectives

There can be different test objectives:

- finding defects
- gaining confidence about the level of quality and providing information
- preventing defects

In acceptance testing, the main objective may be to confirm that the system works as expected, to gain confidence that it has met the requirements.

Fundamentals continued…

- Static and Dynamic Testing

Static and Dynamic Testing

Static Testing

Testing without executing the program

- This include software inspections and some forms of analyses
- Very effective at finding certain kinds of problems – especially “potential” faults, that is, problems that could lead to faults when the program is modified

Dynamic Testing

Testing by executing the program with real inputs

- Includes various techniques but requires the coding stage to be completed.

Dynamic and Static Testing

Testing also includes reviewing of documents (including source code).

Both dynamic testing and static testing can be used as a means for achieving similar objectives, and will provide information in order to improve both the system

Reviews of documents (e.g. requirements) also help to prevent defects appearing in the code.

Static testing in the lifecycle

Static tests are tests that do not involve executing software

Reviews, walkthroughs, inspections of (primarily) documentation

Requirements

Designs

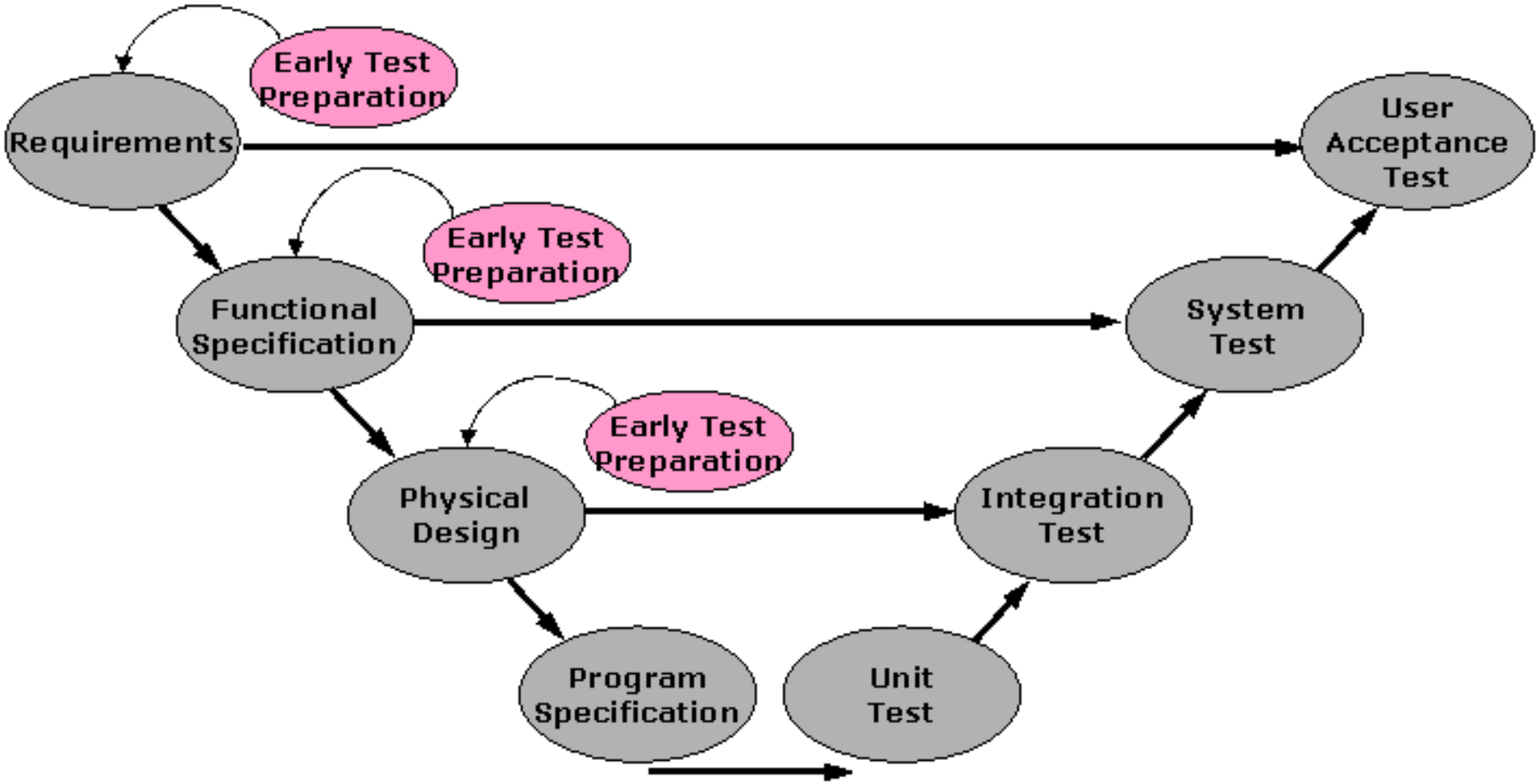
Code

Test plans.

Dynamic testing in the lifecycle

- Program (unit, component)
- Integration or Component Integration testing
- System testing
  - non-functional (e.g. performance, security, backup and recovery, stress)
  - functional where functionality as a whole is evaluated
- User acceptance testing.

Early test case preparation



Fundamentals continued…

- Types of Testing

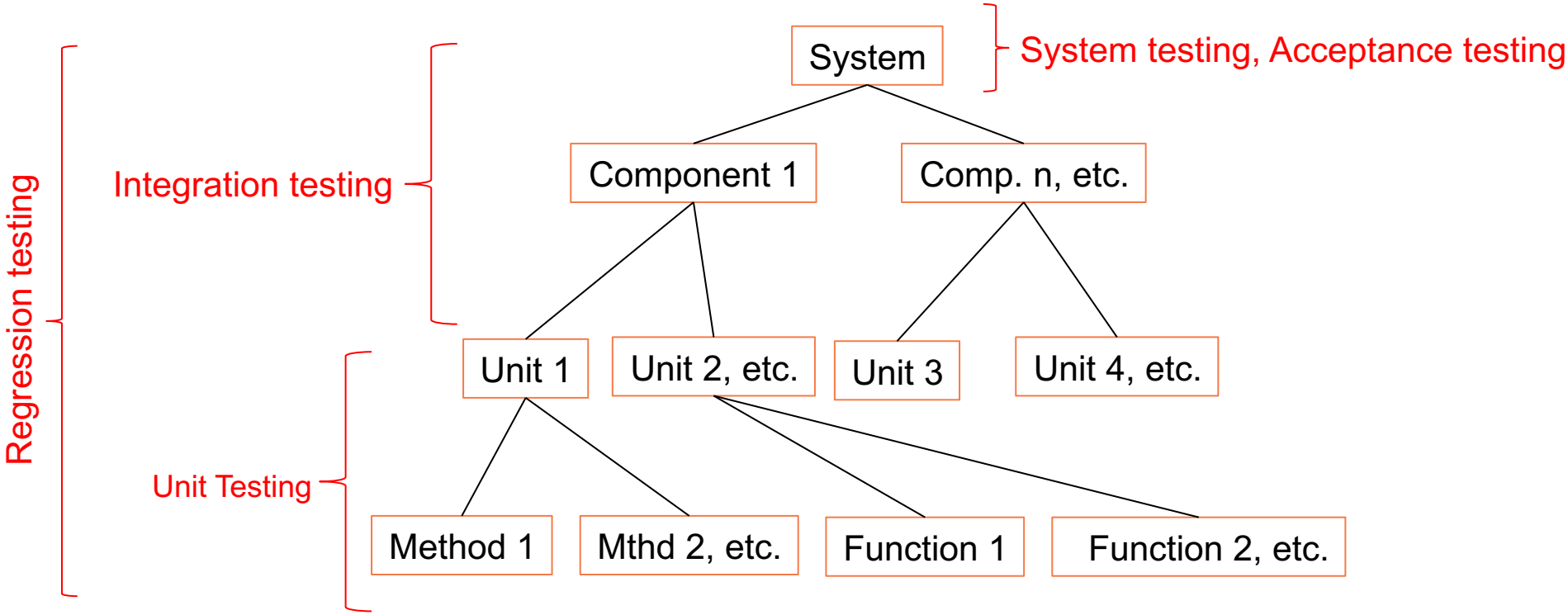
Types of Testing

In some cases the main objective of testing may be to assess the quality of the software (with no intention of fixing defects), to give information to stakeholders

Regression/Maintenance testing often includes testing that no new errors have been introduced during development of the changes.

During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Types of Dynamic Testing



Each Type Of Testing

Each type of testing addresses a specific concern:

Unit Testing: addresses the quality of individual units (e.g. methods, functions)

Integration Testing: checks if individual components (which are a combination of two or more individual units) operate together

System Testing: checks if the system as a whole delivers the desired functionality.

Other types of testing also exist that address other concerns, e.g. acceptance testing allows users or a designated user representative to establish if the system is acceptable to the users.

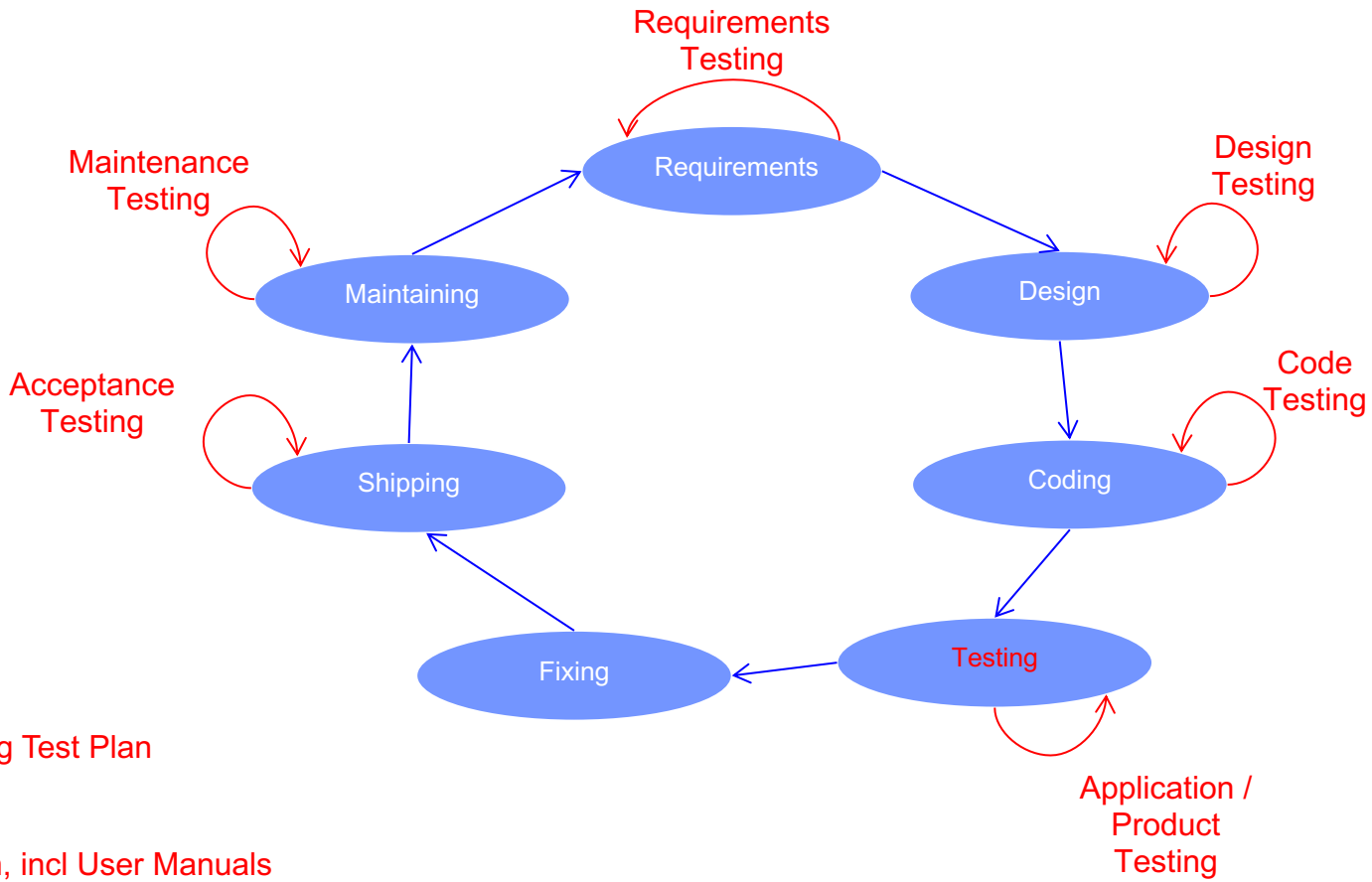
Types of Static Testing

Mostly taking the form of:

Reviewing

Inspecting

Software Lifecycle



Other testing:

Testing plans, including Test Plan

Testing test cases

Testing documentation, incl User Manuals

Remember: Debugging and Testing

Debugging and testing are different

Testing can show failures that are caused by defects

Debugging is the programmer activity that identifies the cause of a defect, repairs the code and checks that the defect has been fixed correctly

Subsequent confirmation testing by a tester ensures that the fix does indeed resolve the failure

The responsibility for each activity is very different, i.e. testers test and developers debug.

Fundamentals continued…

- End of Week 2 Materials

Unit Testing Using PyUnit

An Introduction

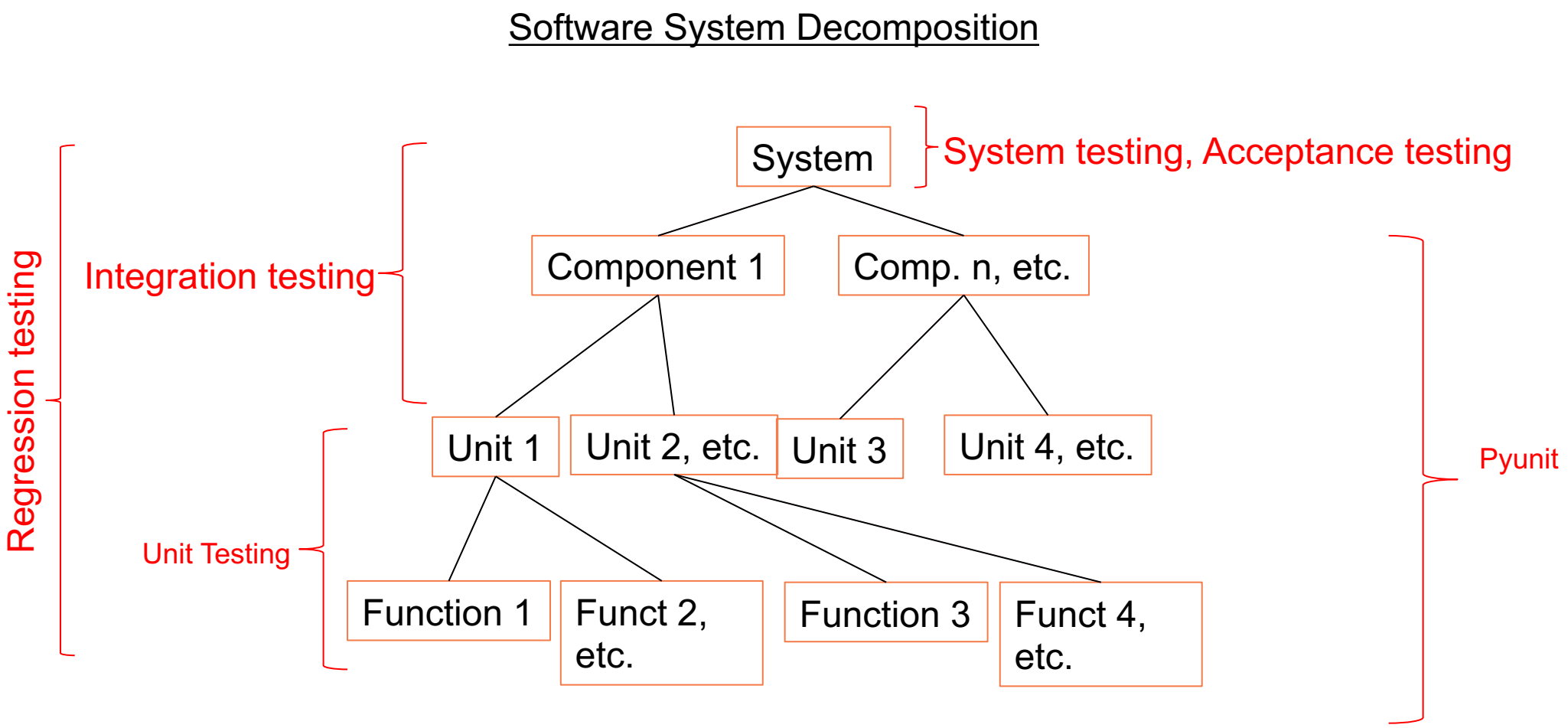
What is PyUnit?

- PyUnit is a unit testing framework for Python
- PyUnit is a Python language version of JUnit
- Plays a crucial role in test-driven development
- Advocates the idea of "first testing then coding“ which is aligned with a test driven development (TDD) philosophy
  - Set up the test data for a piece of code
  - Test the test set up
  - Then write the code
  - Then unit test the code using the test set up
- Unit testing can increase programme productivity
  - If implemented correctly, less time will be required for debugging / resolving defects

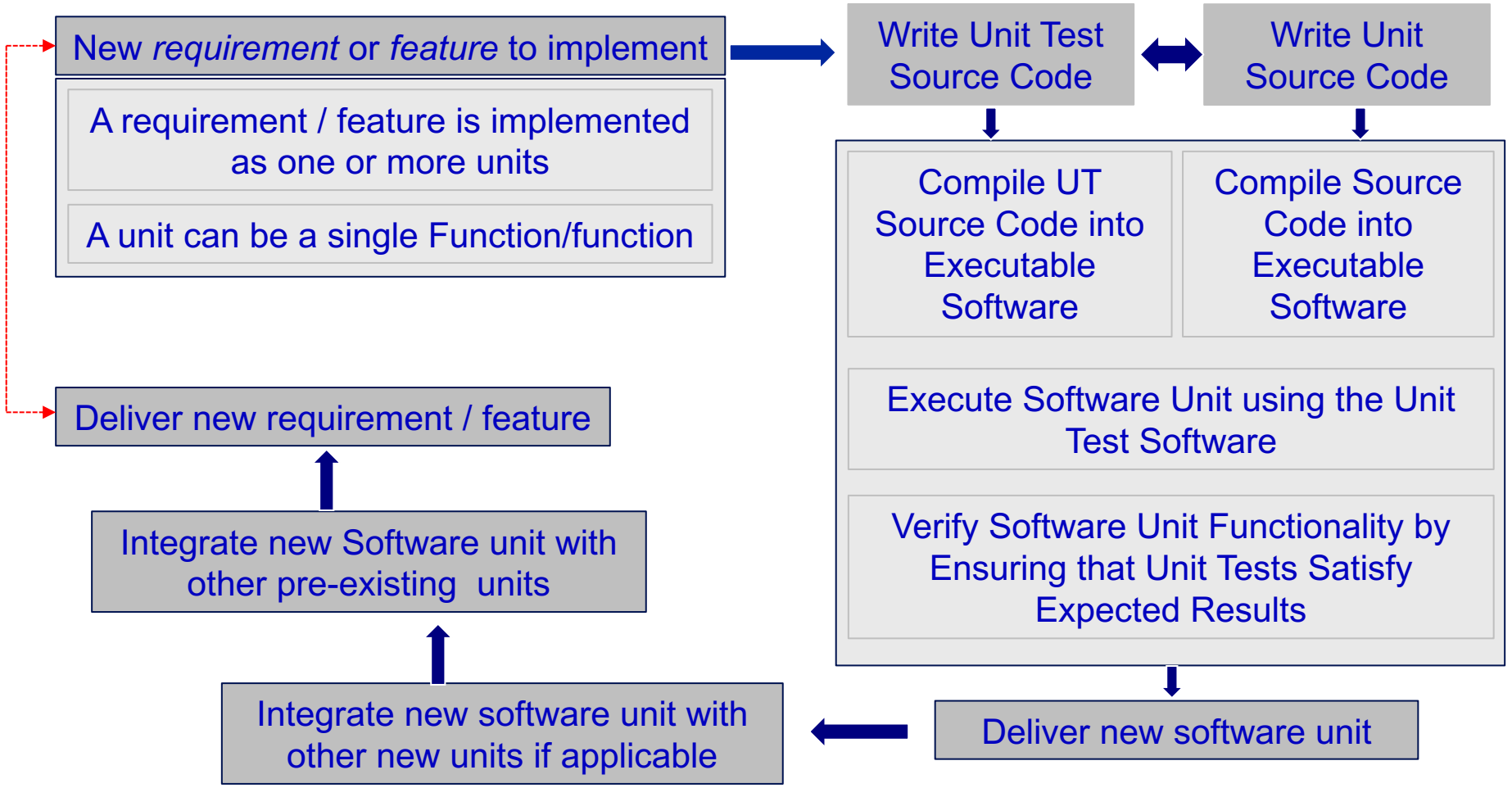
Why use unit tests?

- Unit tests help developers to verify that the logic / behaviour of a piece of code/software is correct.
- Re-performing unit tests (automatically) can identify software regressions introduced by changes in the code.
- Unit tests are therefore very useful for regression testing
- Unit testing frameworks can be harnessed for integration and system level testing also.
- May not be sensible to write unit tests for all code.

Where unit testing fits in…



Sequencing of unit testing - generalised



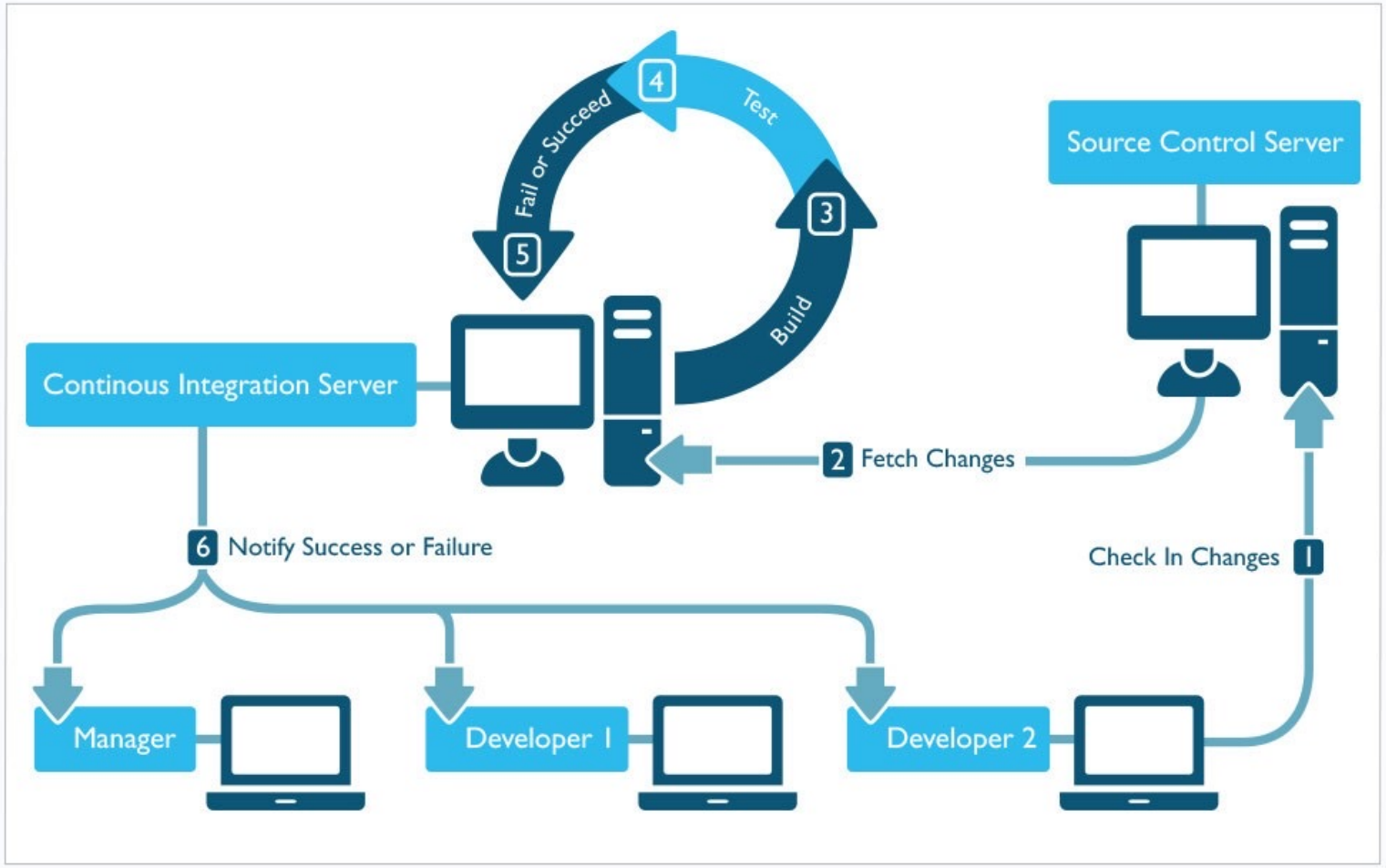
What happens when unit tests fail?

- For an individual programmer:
  - Their compilation will fail until the unit test passes
- For a group programming together:
  - The group build/compile will fail unless all unit tests pass
  - Manifestly more embarrassing for the individual who breaks the build!
- Major implications for Continuous Integration systems that are a feature of most modern software development

Continuous Integration (CI) systems

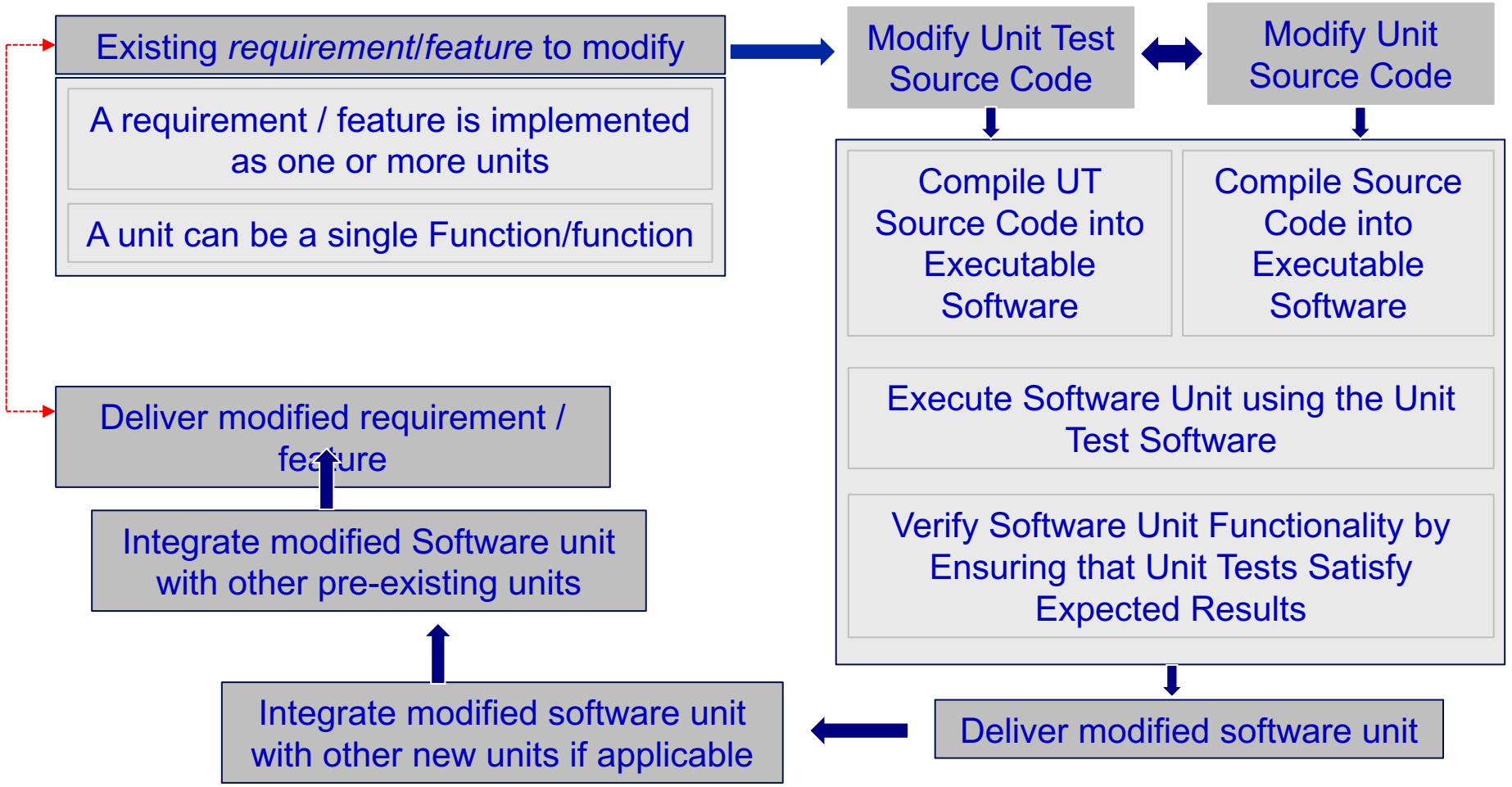
- Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day.
- Each check-in is then verified by an automated build, allowing teams to detect problems early.
- By integrating regularly, you can detect errors quickly, and locate them more easily.
- Automated builds are inherently associated with unit testing.

Continuous Integration (CI) systems



- Figure source: https://insights.sei.cmu.edu

Sequencing of unit testing (2)



What is PyUnit?

- PyUnit is an open source framework
- Used for writing and running unit tests (and test suites)
- Part of the python standard library (*import unittest*)
- Provides assertions for testing expected results
  - E.g. *assertEqual()* to check if an actual value corresponds to the expected value
- Provides test runners for executing tests
  - E.g. a Function called *TestRunner* that can invoke, execute and report on unit test case results (can be used to run suites of tests)

PyUnit – automation and regression

- PyUnit runs automatically (once the tests are written!)
- PyUnit checks results automatically – and flags any errors
- No errors means that all the unit tests have passed as expected
- PyUnit tests can be organised into suites of test cases
- Test suites can be executed any time and much more quickly than manual testing
- This has positive impacts for regression testing and associated software release pace.

What is a PyUnit test case?

- A piece of code that checks (ensures?) that another piece of code (e.g. a function / method) works as expected
- PyUnit is a testing framework that enables us to quickly develop, execute and re-execute unit test cases for python code

What is a PyUnit test case?

- A unit test case is characterised by
  - a known input, and
  - an expected output.
- Both inputs and expected outputs are identified **before** the test is executed.

How many unit test cases?

- It is suggested that there should be at least two unit test cases for each requirement
  - one positive test, and
  - one negative test.
- Where a requirement has sub-requirements, each sub-requirement should also have at least two test cases: one positive and one negative.

What are positive / negative test cases? (1)

- No commonly accepted general definition for these terms!
- Positive test cases – test the behaviour of the code using data that is expect to be received in a correct operational mode
  - E.G. The year entered is valid, i.e. “2017”
- Negative test cases are – test the behaviour of the code when incorrect data is entered (i.e. how does the code handle instances of incorrect data)
  - E.G. The year entered is invalid, i.e. “1822”, or “2050”, or “asde”, or “-dfg”
- It is important to adopt both perspectives.

What are positive / negative test cases? (2)

- An alternative definition:
  - Positive testing is the testing for something that should happen, does happen.
  - Negative testing is the testing for something that should not happen, does not happen

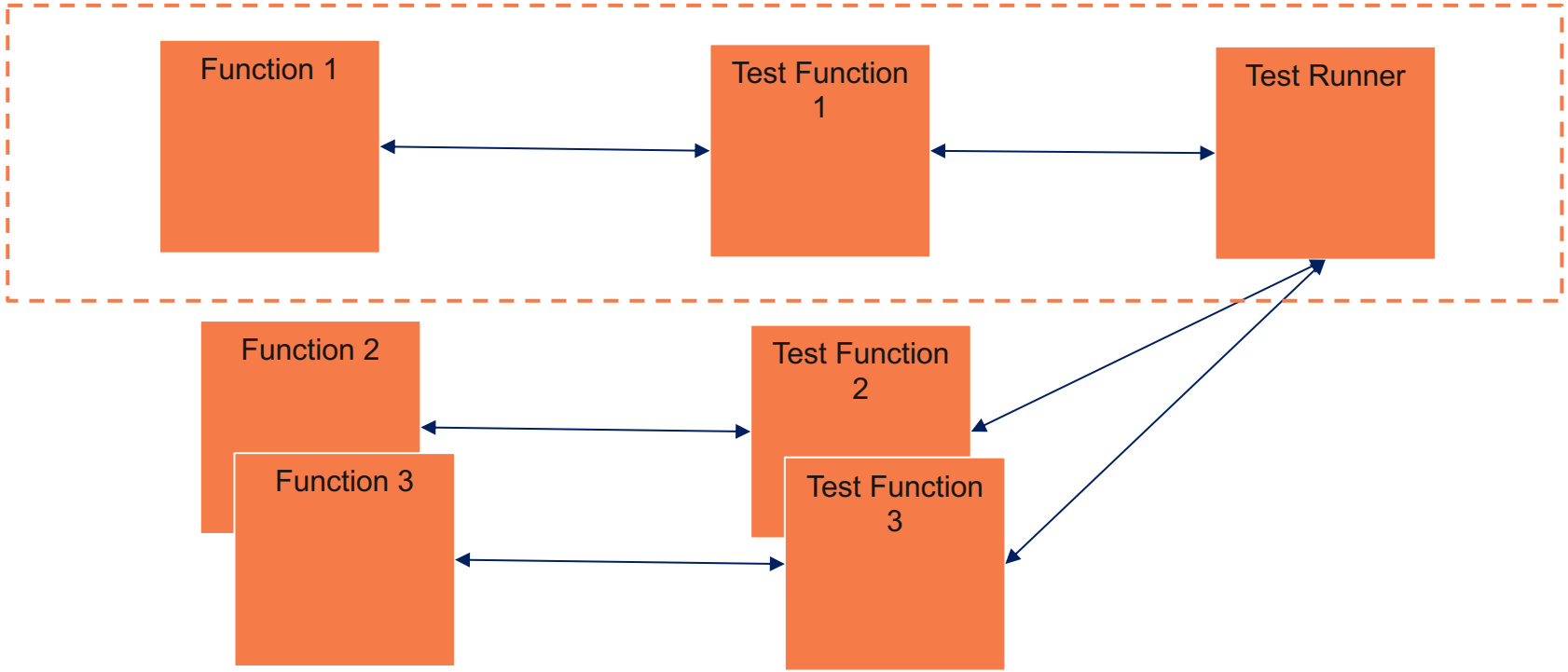
- Whatever definition is adopted, a close examination of the requirement / user story / feature is necessary to design both positive and negative test cases.
- When completed, all tests should be complete and they should all pass

Unit test tips

- Provide meaningful / accurate messages in assert statements:
  - It may be (more) obvious what the issue is, meaning that it will be easier to identify and fix.
  - It won’t always be the original developer (or any developer) who will notice/manage reported unit test failures
- But only add messages to the assert statement if they represent an improvement from the standard error message that will be issued in the case of a unit test failure.

3 different .py files…

- A *TestSuite* is used by convention for *running* suites of tests.



PyUnit Assertions

- There are many ways to use PyUnit assertions
- A list of PyUnit assertion usage is available here: https://docs.python.org/2/library/unittest.html#assert-methods
- *assertEqual()* and *assertNotEqual()* unit test assert methods are perhaps the two most basic / well known

PyUnit Assertions

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertAlmostEqual(a, b)	a is considered equal to b in view of desired precision
assertNotAlmostEqual(a, b)	a is not considered equal to b in view of desired precision

Where to locate unit tests?

- No universally agreed response to this question.
- From a code management perspective, there are some benefits to separating the unit test code from the actual code – but the location of the unit test code is a subject of some contention.
- Some advocate that the unit tests should reside alongside the code, perhaps within a sub directory.
- Others advocate that the unit tests should be elsewhere in the directory hierarchy, separated from the code.

Why the location debate?

- Some advocate shipping unit test code to production systems (and in some cases this is desirable)
- Others advocate that unit test code should never be shipped to production systems.
- And there is some debate in terms of sequencing, with some advocating that unit tests should only be executed once there are no underlying build issues (others advocate that this is not necessary or desirable). Isn't a basic compile error (e.g. arising from a syntax error) different to a unit test failure?
- So what are the benefits of these different approaches?

Unit Testing Using PyUnit

Code Examples

Some coding examples

1. Sample code to add two numbers and some associated unit test code (using assertEqual() and assertNotEqual())
2. Sample code with a broken unit test – to examine the type of output we receive
3. Sample code to check if a number is a prime number and associated unit test code (using assertTrue() and assertFalse())
4. Sample code to run the tests in (1) and (2) above as a test suite

We're using **python 3.7** and the associated IDLE IDE.

(1) Python code: add.py

```
def add(x,y):
        return x+y
```

Obviously, from a practical perspective, do not need to write code to test that python can add two numbers reliably

This sample code is created for the purpose of demonstrating unit testing in action

This file will be stored as add.py and in a subdirectory called ‘code’

(1) Python code: test_add.py

```
import unittest
from code.add import add

class AddTestCase(unittest.TestCase):

    def test_one(self):
        self.assertEqual(add(1,1),2)

    def test_two(self):
        self.assertEqual(add(1,2),3)

    def test_three(self):
        self.assertNotEqual(add(1,3),5)

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

Unittest is the python package for pyunit

Import ‘add’ from the add package in the subdirectory ‘code’

Create a new test case class to test ‘add’

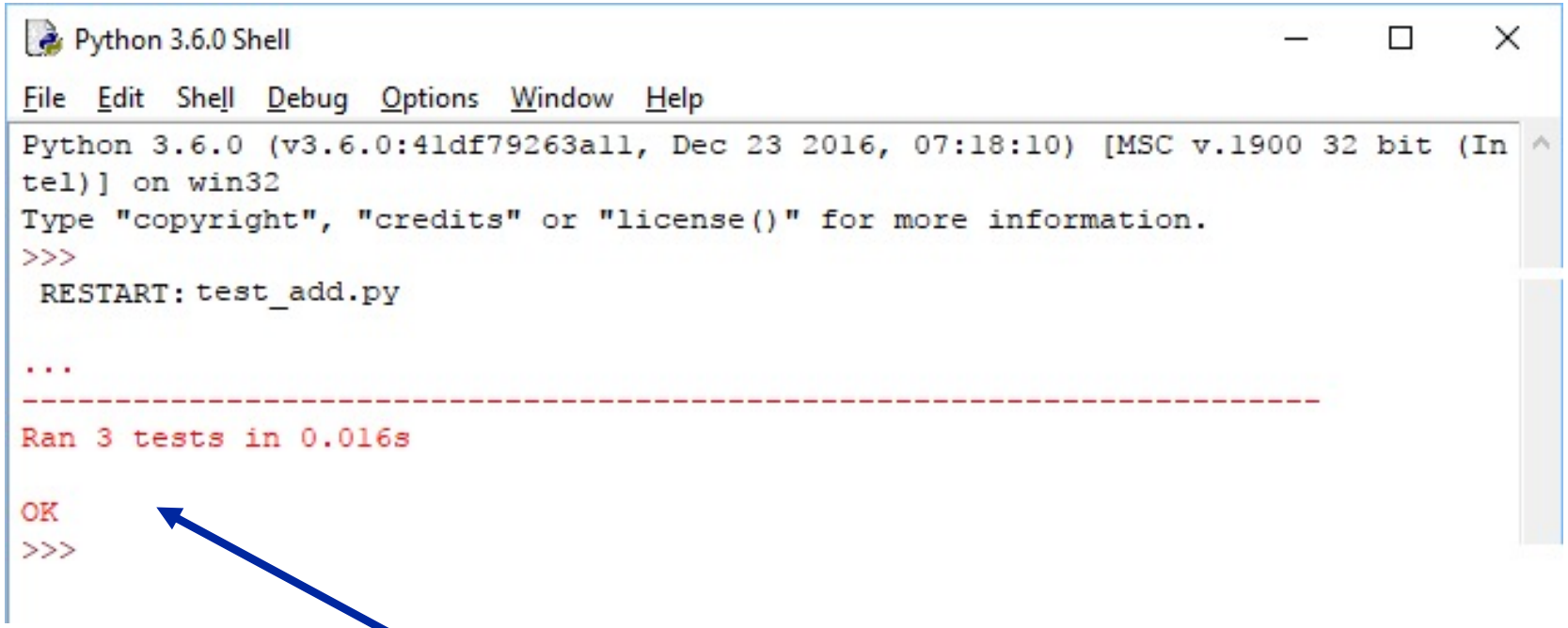
Positive test case: Check if the return value from add(1,1) is 2

Positive test case: Check if the return value from add(1,2) is 3

Negative test case: Check if the return value from add(1,2) is not equal to 5

Allows us to run our tests as if it was the main line of the program (i.e. the place the program starts)

(1) What happens when we run this code?



All three tests ran, everything is “ok”

In IDLE, you can use the F5 shortcut key to automatically run the code.

(2) When a test fails… (because of a faulty test case)

```
import unittest
from code.add import add

class AddTestCase(unittest.TestCase):

    def test_one(self):
        self.assertNotEqual(add(1,1),2)

    def test_two(self):
        self.assertEqual(add(1,2),3)

    def test_three(self):
        self.assertNotEqual(add(1,3),5)

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

Unittest is the python package for pyunit

Import ‘add’ from the add package in the subdirectory ‘code’

Create a new test case class to test ‘add’

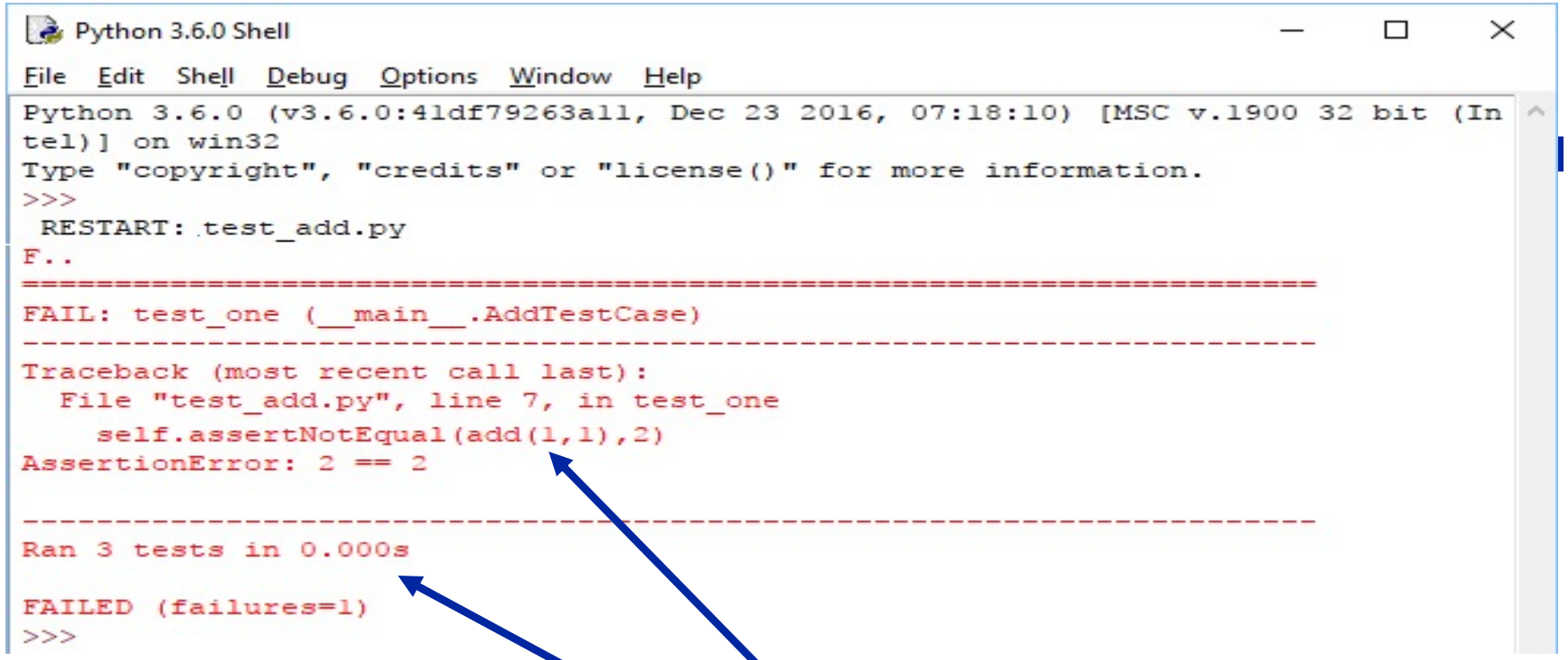
Broken test case that expects the return value from add(1,1) not to equal 2

Positive test case: Check if the return value from add(1,2) is 3

Negative test case: Check if the return value from add(1,2) is not equal to 5

Allows us to run our tests as if it was the main line of the program (i.e. the place the program starts)

(2) What happens when we run this broken test case?



In IDLE, you can use the F5 shortcut key to automatically run the code.

All three tests ran, there has been one “Failure”, in line 7

(3) Python code: is_prime.py

```
def is_prime(number):
#funtion goes here
    for element in range(2,number):
        if number % element == 0:
            print(number)
            print("NOT PRIME")
            return False
    print(number)
    print("PRIME")
    return True

x=int(input("enter:"))
is_prime(x)
```

(3) Python code: corresponding unit test code

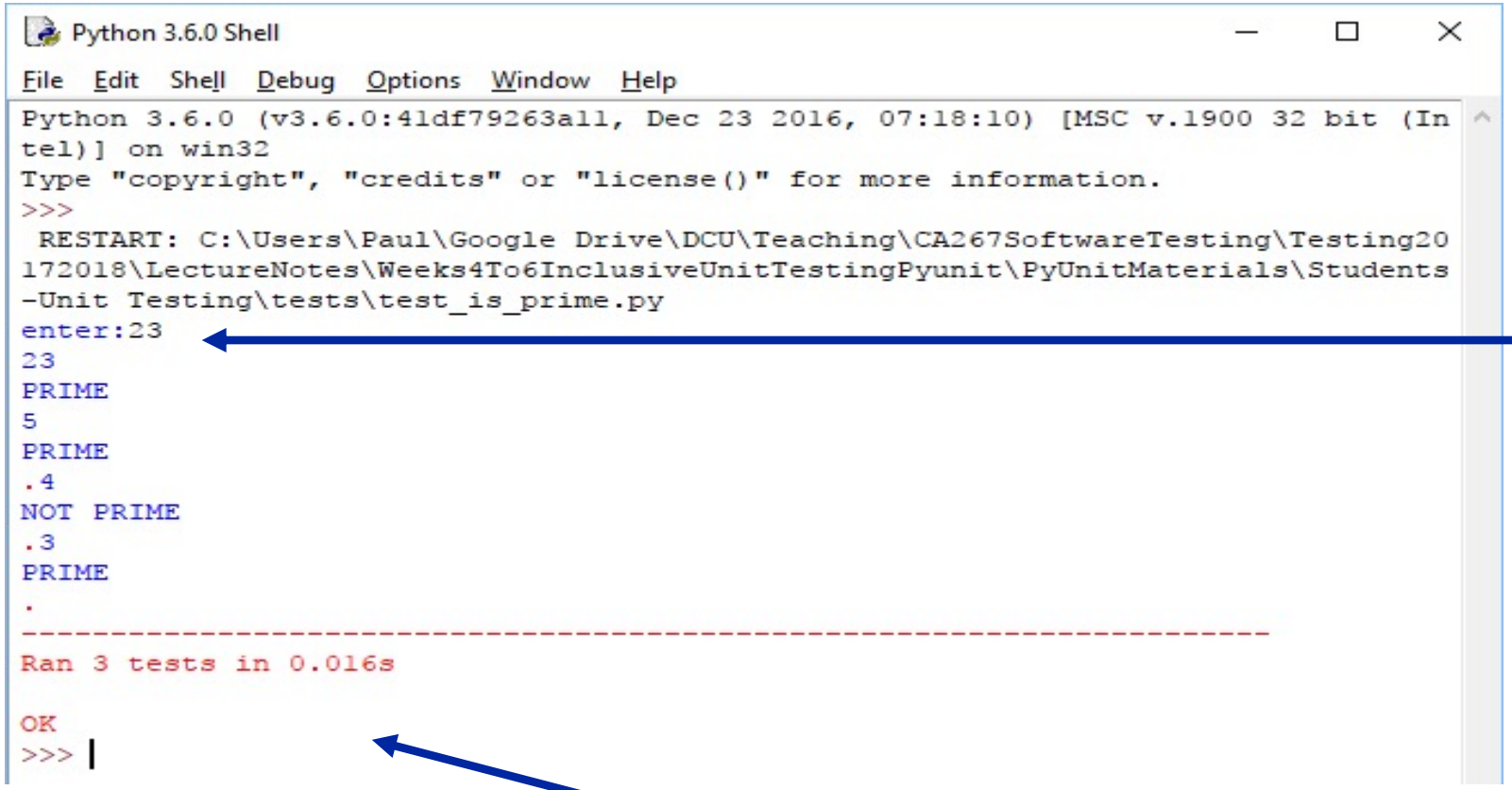
```
import unittest
from code.primes import is_prime

class PrimesTestCase(unittest.TestCase):

    def test_is_five_prime(self):
        self.assertTrue(is_prime(5))

    def test_is_four_prime(self):
        self.assertFalse(is_prime(4))

    def test_is_three_prime(self):
        self.assertTrue(is_prime(3))


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

Check that the return value from is_prime() is true for 5

Check that the return value from is_prime() is false for 4

Check that the return value from is_prime() is true for 3

(3) What happens when we run this code?



Manually entered “23” at the command line. Note that this is not part of the pyunit tests.

All three tests ran, everything is “OK”

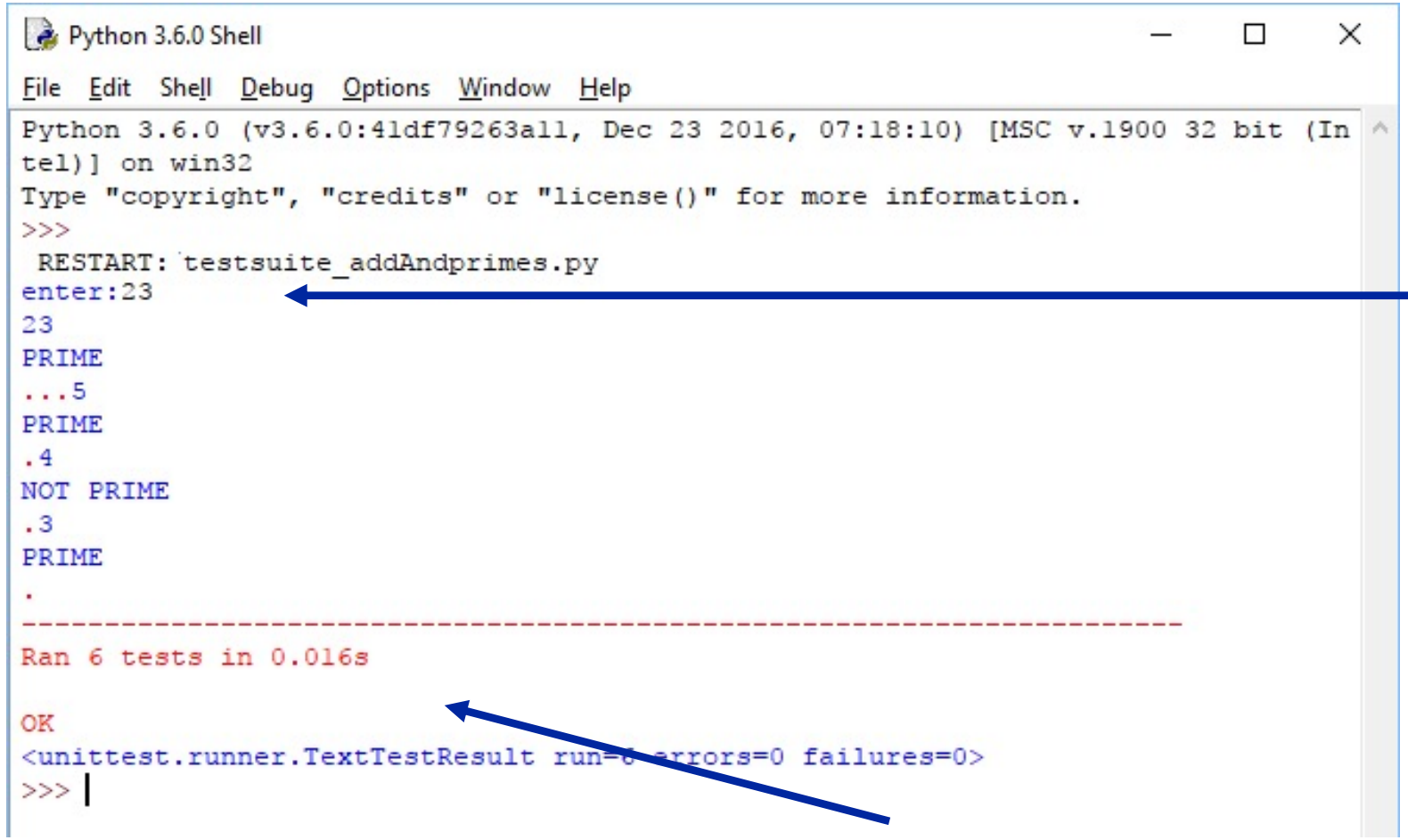
(4) Python test suite code: add and primes

```
import unittest

from test_add import AddTestCase
from test_is_prime import PrimesTestCase


def my_suite():
    suite = unittest.TestSuite() #instance of Testsuite
    suite.addTest(unittest.makeSuite(AddTestCase))# addTest is a keyword
    suite.addTest(unittest.makeSuite(PrimesTestCase))
    runner = unittest.TextTestRunner()
    print(runner.run(suite)) # run the suite and print out the results

my_suite()
```

Import ‘AddTestCase’ and ‘’PrimesTestCase’

Create my test suite

(4) What happens when we run this code?



Manually entered “23” at the command line. Note that this is not part of the pyunit tests.

Note: both sets of test cases and source files are correct prior to running this test suite.

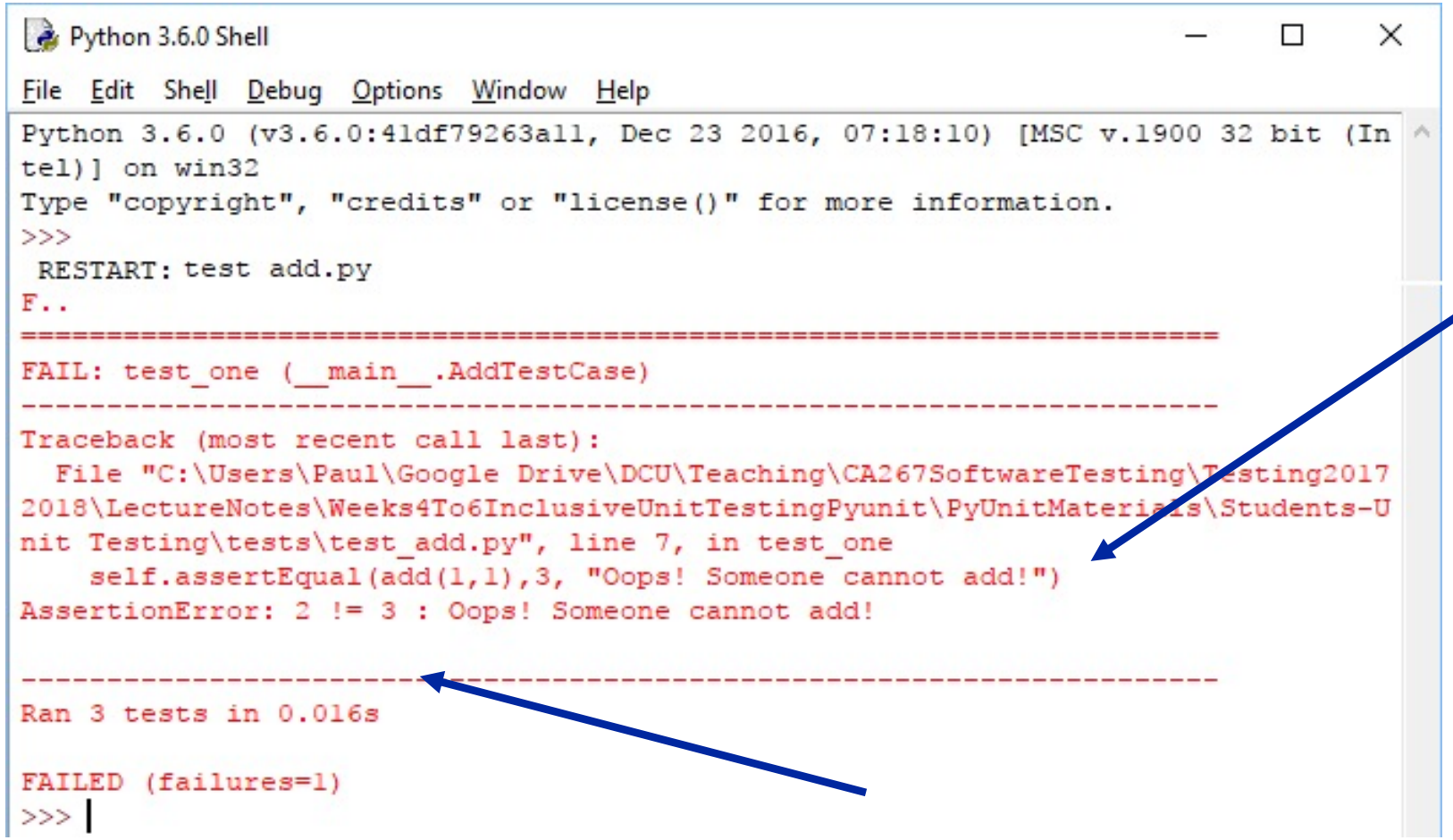
All 6 tests ran, everything is “OK”

Example: Adding a custom message to a unit test

```
import unittest
from code.add import add

class AddTestCase(unittest.TestCase):

    def test_one(self):
        self.assertEqual(add(1,1),3, "Oops! Someone cannot add!")

    def test_two(self):
        self.assertEqual(add(1,2),3)

    def test_three(self):
        self.assertNotEqual(add(1,3),5)

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

Custom message. In practice, a more informative / professional message should be added!

What happens when we run this code?



Custom message appears in output.

All 3 tests ran, but one failed.

PyUnit for large scale automated testing?

We have looked at some very small-scale examples constructed just for the purpose of learning to use pyunit at an introductory level.

These same principles can be scaled to allow great numbers of test cases to be executed automatically using pyunit.

This is a powerful way to continually perform tests, especially when used in conjunction with continuous integration

PyUnit for integration and system testing?

Since the unit test framework can invoke any code and in any sequence, it can be considered to be quite effective for integration testing (and possibly some system testing also).

We can therefore orchestrate entire automated integration testing sweeps that can run continuously

Together with unit testing, this approach is important in enabling continuous software engineering

But to truly continuous software engineer, other things are also needed (consider GUI testing, automated deployment, monitoring and control, etc…)

Pair Programming

An Introduction

What is *Pair* Programming?

- Effectively: two people performing one programming task concurrently, while seated together at one computer.
- A definition from the Agile Alliance:
  - Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair). The programmer at the keyboard is usually called the "driver", the other, also actively involved in the programming task but focusing more on overall direction is the "navigator"; it is expected that the programmers swap roles every few minutes or so.

Benefits of pair programming

- Two people working on problem solving and two minds can be better than one single mind.
- Instant testing and reviewing as a second person is overseeing the work, asking questions and making suggestions for improvements.
- Defect removal (a study suggests 15% quality improvement, but that pair programming is 15% more costly!)
- Shown to be effective in programming educational environments.

Benefits of pair programming

- Less obvious benefits:
  - When implemented correctly, potentially increased peer pressure to perform (e.g. surfing the net and checking email is not so easy when someone is sitting alongside you and checking your work)
  - Increased socialising among the team can increase overall team performance
  - Knowledge sharing and training
  - Reduced overall coordination effort (1/2 as many programming tasks at any time)

Reported issues with pair programming

- both programmers must be actively engaging with the task throughout a paired session, otherwise no benefit can be expected
- at least the driver, and possibly both programmers, are expected to keep up a running commentary; i.e. "programming out loud" - if the driver is silent, the navigator should intervene
- pair programming cannot be fruitfully forced upon people, especially if relationship issues, including the most mundane (such as personal hygiene), are getting in the way; solve these first!

Reported issues with pair programming

- Grades of noise:
  - Some programmers have a propensity for chitchat, focus must be on the programming task otherwise it is not pair *programming,* it is pair *chitchatting*.
  - Some programmers talk too loud and distract other pair programmers, it is important to moderate the volume.
  - One of the pair do not engage, freeloading.
  - No noise at all is just as concerning.

Pair programming and testing

- When used simultaneously with test-driven development, a flavour sometimes termed *ping-pong* programming promotes more frequent switching of roles
- One programmer writes a failing unit test, then passes the keyboard to the other who writes the corresponding code, then goes on to a new test.
- This variant can be used purely for educational purposes.

Pair programming

- Further info available online, e.g. https://www.agilealliance.org

Useful Pair Programming video links

- An introduction to student based pair programming:
  https://www.youtube.com/watch?v=rG_U12uqRhE

- Pair programming in the main:
  https://www.youtube.com/watch?v=u_eZ-ae2FY8

Introduction to Test Design Techniques

Equivalence Class Partitioning

Boundary Value Analysis

Test design techniques

Purpose of test design techniques is to identify test conditions and test cases.

Software testing is complicated, we need to have robust design techniques to enable effective, yet economic/practical testing.

Why use techniques?

Exhaustive testing of all program paths is impractical

Exhaustive testing of all inputs is also impractical

Even if we could, most tests would represent duplicates (that may bring no additional value)

So, we need to select tests which are

- effective at finding faults
- efficient.

Category of Test Design Technique

Test Technique Category	Common features
Specification based Black-Box	**Deriving test cases directly from a specification or model of a system/proposed system.** Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components. Test cases can then be derived systematically.
Structure based White-box	**Deriving test cases directly from the code written (or design) to implement a system**. Information about how the software is constructed is used to derive test cases. The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage.
Experience based	**Deriving test cases from the tester's experience of similar systems and general experiences of testing.** Knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment. They will have knowledge about likely defects and their distribution.

Specification based or black box techniques

- Equivalence partitioning.
- Boundary value analysis.
- Decision table testing.
- State transition testing.
- Use case testing.

- ISO/IEC/IEEE 29119-4:2015 Software and systems engineering — Software testing — Part 4: Test techniques

Equivalence Partitioning

Equivalence partitioning

Can be applied at all test levels.

Is usually recommended as one of the first test techniques to use.

Is based on **dividing a set of test conditions into groups** that can be considered the same, that is, the **system will treat them equivalently**.

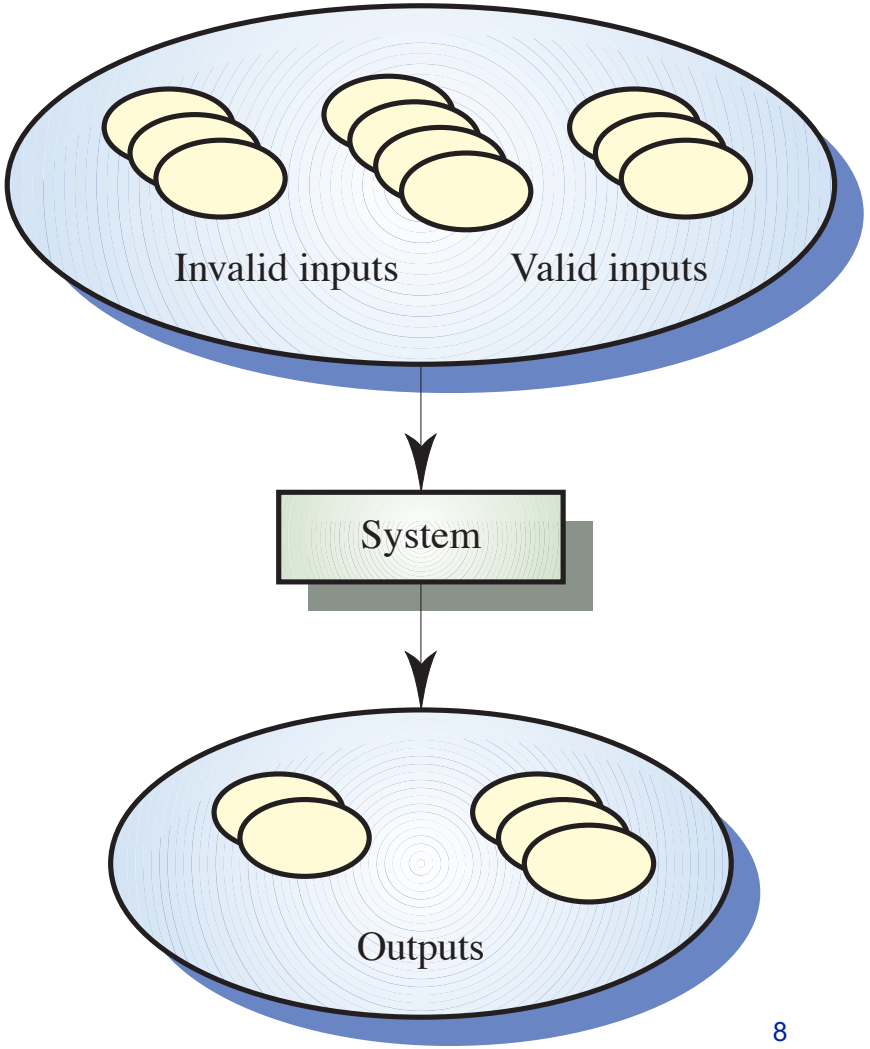
When we have our partitions, we can choose one condition from the partition to test as **we assume that all the conditions in the partition will work the same**.

The result of testing a single value from an equivalence partition is **considered representative of the complete partition**.

This is a very simplified assumption and it is usually advisable to test more than one condition from a particular partition.

Equivalence Partitioning

Equivalence partitioning is the process of methodically reducing the large (or perhaps infinite) set of possible test cases into a small, but equally effective, set of test cases.



Valid and Invalid

Equivalence partitions may also be treated in terms of **valid** partitions and **non-valid** partitions.

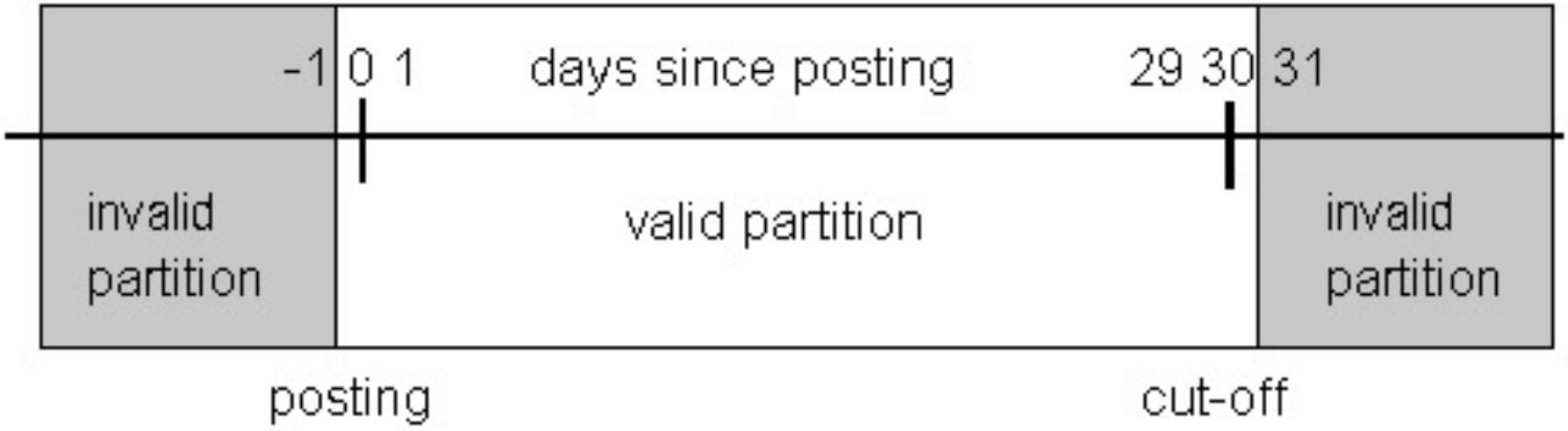
The particular specification may only mention information that can derive the valid partitions but it is important, as a tester, to be able to identify non-valid partitions as well.

Invalid partition values may still be entered by the user but they are not expected input by the system.

In this case the software should be able to correctly handle values from the invalid partition by displaying the appropriate error message.

Equivalence partitioning example

Requirement:
Contestants have 30 days from date of posting form to return the entry form.



Identifying equivalence partitions

Validity Rule	*Input Requirement*	*Valid Equivalence Class(es)*	*Invalid Equivalence Classes*
Range	Between 0.0 and 99.9	Between 0.0 and 99.9	<0.0 >99.9
Count	Must be 1, 11, 43, 82, 955	1, 11, 43, 82, 955	any number not in the valid list e.g. 2, 37, -56
Set	Must be BUS, CAR or TAXI	BUS CAR TAXI	e.g. XXX, TRAIN, PLANE
Must	Must begin with a letter	Begins with a letter (a-z, A-Z)	Any non alpha character or null e.g. 6, +, >, ~

Identifying the partitions for ranges

Requirement:

- "…must be between 23 and 30"
- Does between "include" the values 23 and 30?

Implies:

- The range 23 to 30 is valid
- Less than 23 and greater than 30 are invalid
- But does this mean <=22?
- Or does it mean <=22.9999999?

Need to consider the precision of the domain under consideration

e.g. In finance, precision could be very important.

Selecting tests from partitions

When you have identified the partitions, which values should you take?

E.g. in a partition of 1-1000, which value would you choose for a test?

It doesn't matter!

To achieve EP coverage, all that is required is to select ANY value in the range

If we believe the software treats every value in a partition in the same way, the test values in that partition are equivalent, according to our definition.

Test Case Strategy

Once the set of equivalence classes has been identified, here is how to derive test cases:

- Assign a unique identifier to each equivalence class.
- Until all **valid** equivalence classes have been covered by at least one test case, write a new test case covering as many of the valid equivalence classes as possible.
- Until all **invalid** equivalence classes have been covered, write a test case that covers one, and only one, of the uncovered invalid equivalence classes.
- For each test case, annotate it with the equivalence class identifiers that it covers.

Valid and Invalid Classes: When in doubt...

If there is any reason to believe that elements in an equivalence class are not handled in an identical manner by the implementation software, split the equivalence class into further classes.

Equivalence Class Partitioning

Consider creating an equivalence partition that handles the *default*, *empty*, *blank*, *null*, *zero*, or *none* conditions.

Default: no value supplied, and some value is assumed to be used instead.

Empty: value exists, but has no contents.

e.g. Empty string ""

Blank: value exists, and has content.

e.g. String containing a space character " "

Null: value does not exist or is not allocated.

E.g. object that has not been created.

Zero: numeric value

None: when selecting from a list, make no selection.

Equivalence Classes Partitioning - Problems

- Specification doesn't always define expected output for invalid test-cases.
- Strongly typed languages eliminate the need for the consideration of some invalid inputs.
- Brute-force of defining a test case for every combination of the inputs' ECs.
  - Provides good coverage, but...
  - …impractical when number of inputs and associated classes is large

Boundary Value Analysis

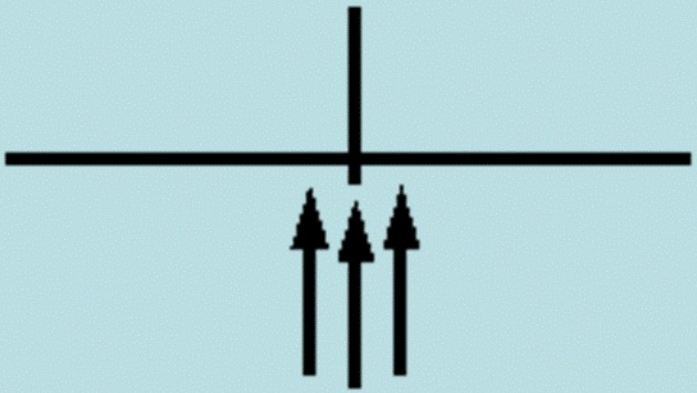
Boundary value analysis

Experience shows more faults at boundaries of equivalence partitions where partitions are continuous

So test cases just above, just below and on the boundary tend to find faults

Don't forget test cases for output partitions: just above, on, just below boundaries.

BS5925-2 boundary value analysis



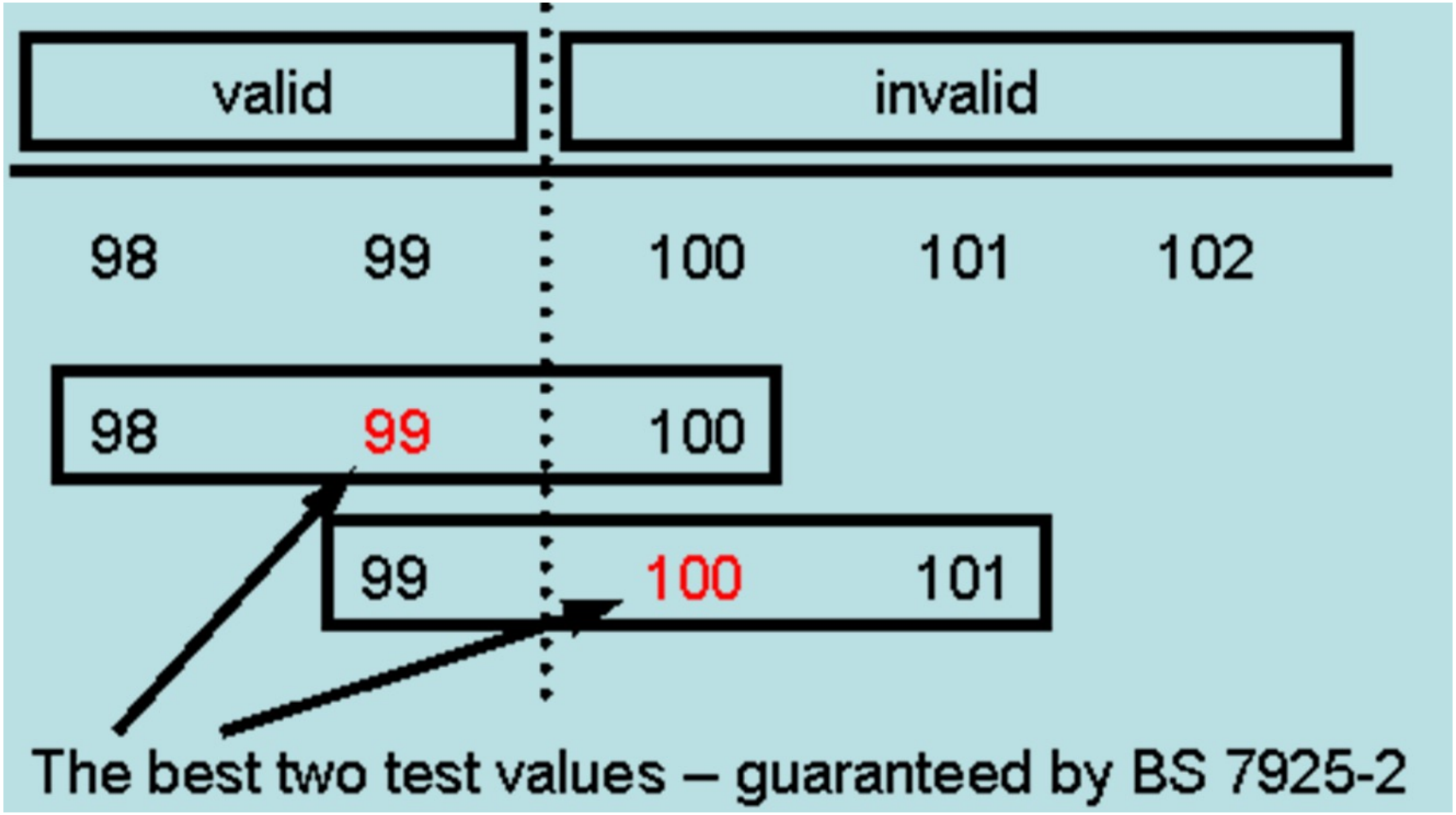
1. Identify the boundary value
2. Select test values:
   - JUST ABOVE AND
   - JUST BELOW

⇒ THREE test values per boundary

Choosing the boundary can be subjective

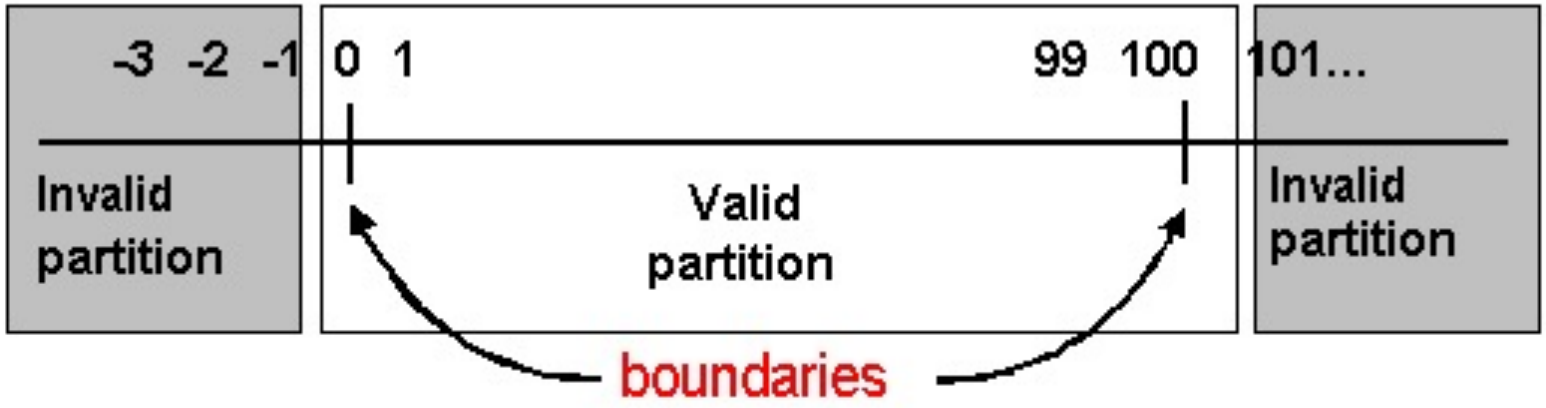
- Can be subjective e.g.
  - X < 100 (choose boundary = 100?)
  - X <= 99 (chose boundary = 99?)
- The same requirement, but you could have different test values depending on the boundary:
  - 99, 100, 101 or 98, 99, 100
- Which is best?
  - Because testers can be “one out”, the standard suggests three boundary values (and not two) to guarantee you get the BEST two test cases.

Choosing the boundary can be subjective 2



Boundary value analysis example

Requirement:
A value is valid in the range 0 to 100



Equivalence partitioning test cases: -5, 56, 101

Boundary value analysis: -1, 0, 1, 99, 100, 101

Boundary values subsumes equivalence partitions for ranges

- BV subsumes EP where the partitions are ranges of values
- If you have covered all boundary values you automatically get EP coverage
- You might have chosen different EP values, but since all values in a partition ARE equivalent any test value – even extreme ones - will do it.

Questions

Debugging

Debugging != Testing

Source code debugging is often undertaken in an IDE (for the purpose of our work, we will use the IDLE debugger).

You may want to have a laptop to work through the debugger and see how it works.

Note: debugging is not testing, it is a tool used by developers to investigate suspected defects.

- Debugging is done by developers, it is not testing.
- Unit testing is implemented by developers, it is testing!
- Confused? Let's take a closer look.

Debugging & Defects

If you enter the wrong code, the computer won't give you the right program.

Computer programs do what you tell them to, but what you tell the program to do may not be the same as what you *wanted* the program to do.

This gives rise to a particular type of defect – a defect introduced by a programmer wherein the programmer's expectation of their (own) code is not consistent with the execution of the corresponding program.

Debuggers are tools used by developers to investigate this particular type of defect.

It is *sometimes* the quickest way for a developer to examine and resolve the defect.

Debugging

It can be difficult to figure out how your code can be causing a defect.

Lines of code get executed quickly, values in variables change frequently.

The debugger is a program that enables developers to *step through* their code one line at a time in the same order that they are executed.

The debugger also identifies the values stored in variables at each step.

To speed up the debugging process, it is not necessary to step through each and every line – we can jump to specific code of interest to our investigation (which can be very efficient in terms of identifying the root cause of defects).

Getting set up in IDLE

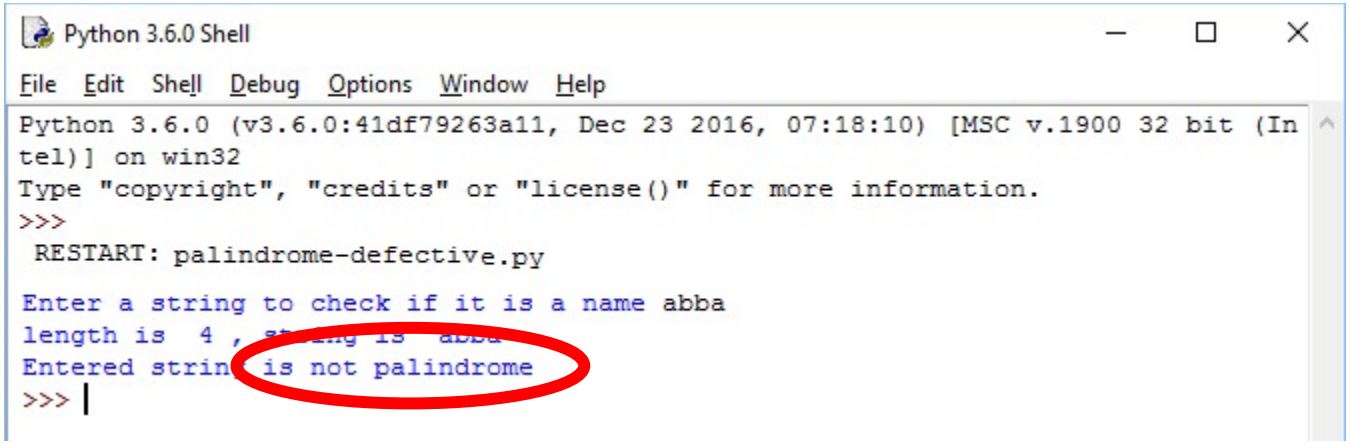
Open the sample defective source code provided in Loop:
palindrome-defective.py

Execute the programme in IDLE (either press F5 or Menu option "Run", sub option "Run Module")

Test the execution of the programme by entering a valid palindrome, e.g. "abba"

Observe that the program does not function as expected, it reports that the string "abba" is not a palindrome (which is incorrect).

Observe that the program does not function as expected



Our options for investigating this issue in the code?

There are essentially two options:

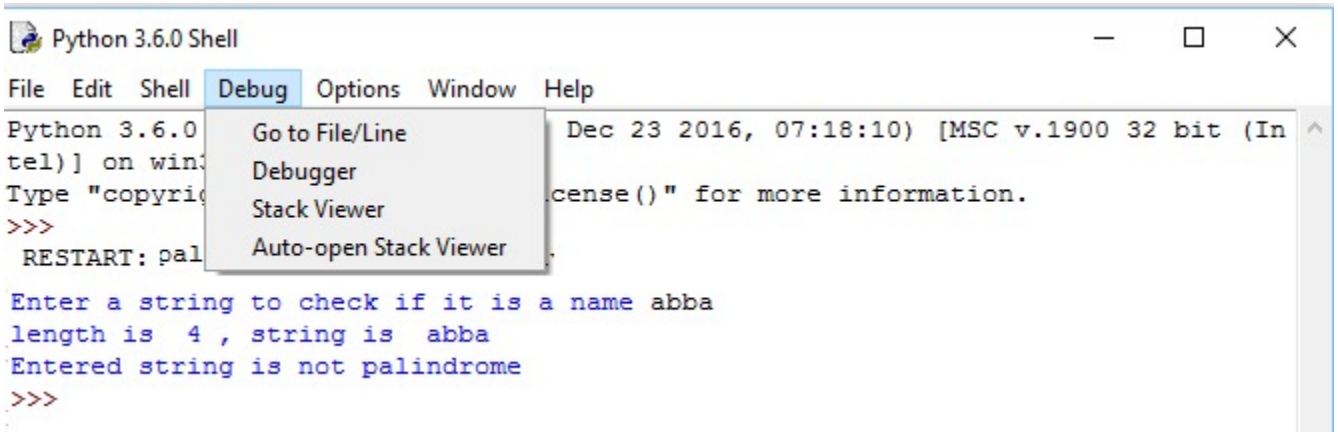
1. Visually review the code
2. Start introducing print statements to try to understand what is happening
3. Use the debugger to try to identify the issue

Limitations of (1) & (2): when there is a lot of code, it can be time consuming getting to review visually.

Benefit of (3): The debugger can *step* straight to the code of interest and *observe the program variables and control flow*.

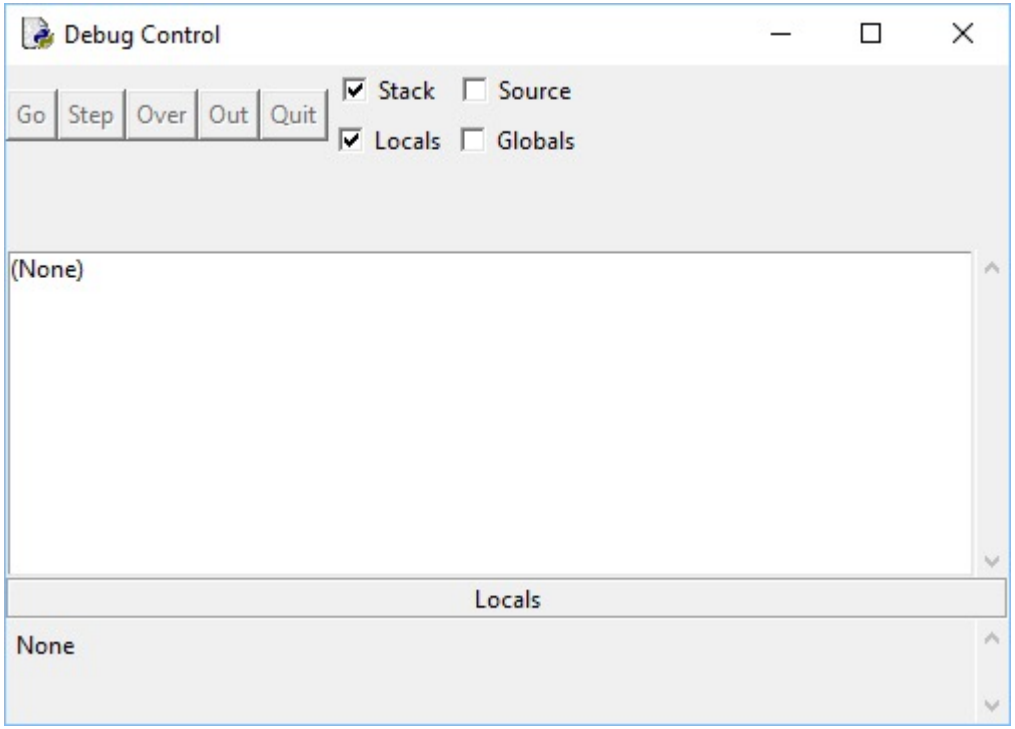
Accessing the IDLE Debugger (1)

1. In the IDLE Python Shell window (i.e. the window where the code is executed), select menu option “Debug”, sub-option “Debugger”



Accessing the IDLE Debugger (2)

1. The IDLE Debugger is now launched and visible via the Debugger window.



Debugging the Code (1)

1. Return to the source code window: *palindrome-defective.py* and execute the programme again.

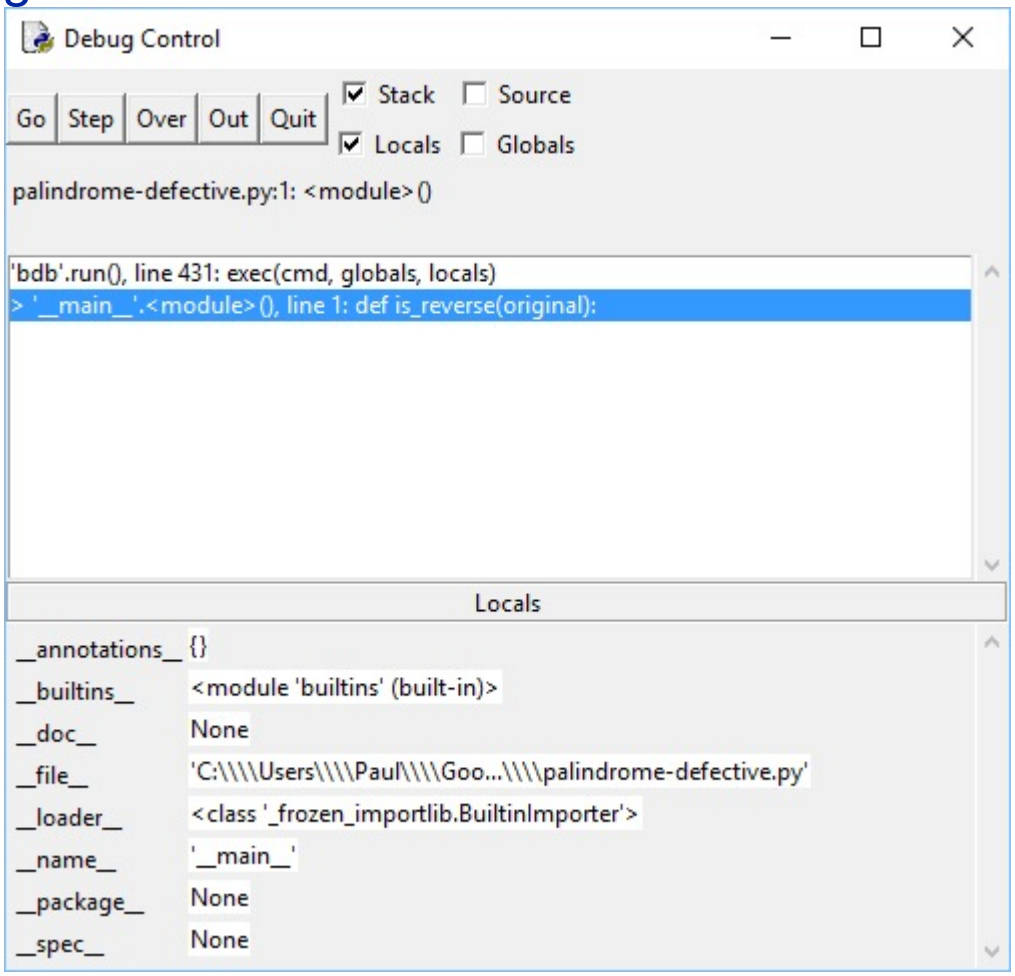
palindrome-defective.py

File Edit Format Run Options Window Help

```
def is_reverse(original):
    reverse=''
    length = len(original)

    for j in range(length - 2, -1, -1):
        reverse = reverse + original[j]
    return reverse


def is_palindrome():
    original=''

    i = 0
    j = 0

    original = input('Enter a string to check if it is a name ')
    length = len(original)

    print('length is ', length, ', string is ', original)

    reverse = is_reverse(original)

    if original == reverse:
        print('Entered string is a palindrome')
    else:
        print('Entered string is not palindrome')

is_palindrome()
```

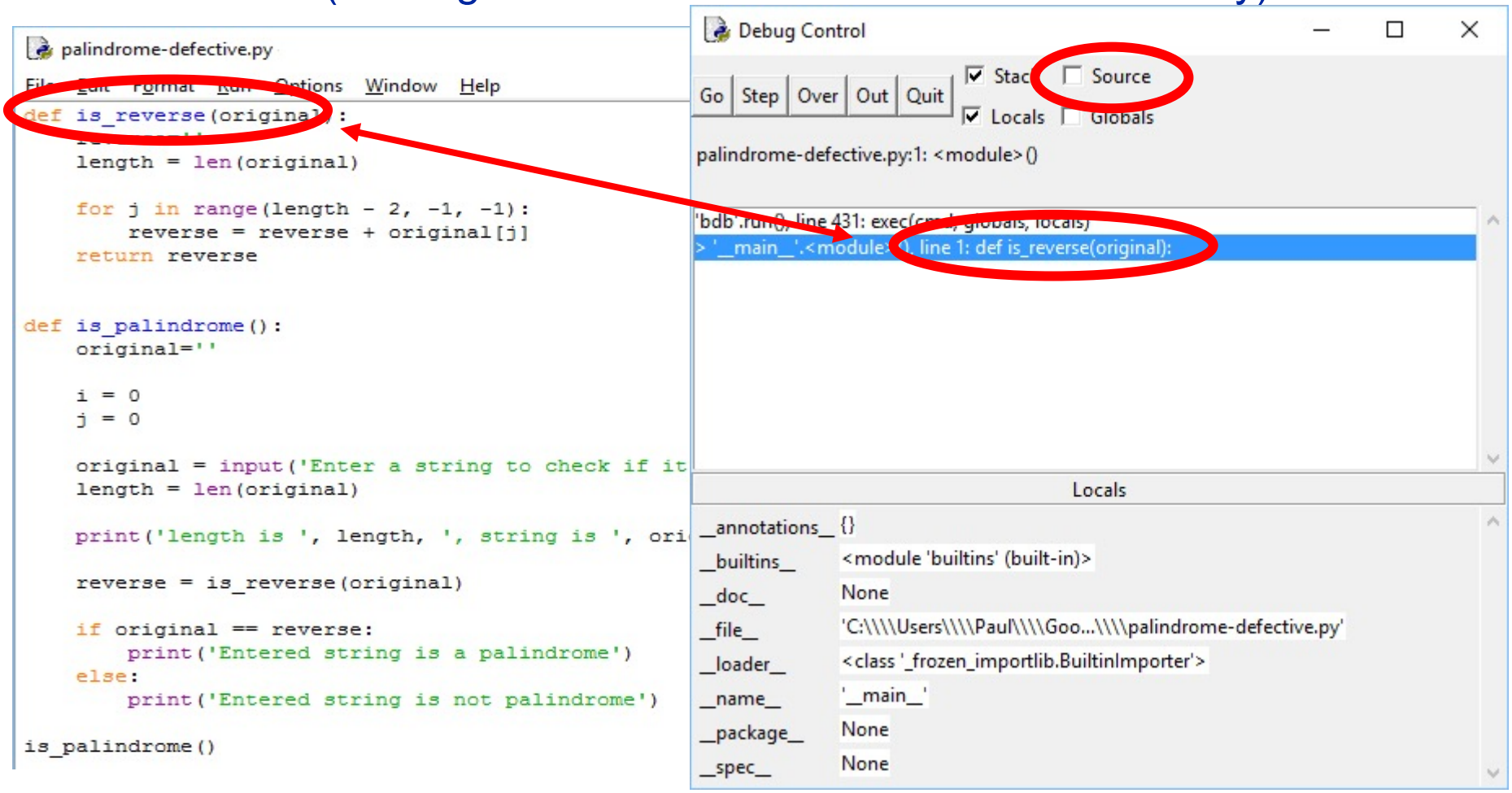
Debugging the Code (2)

2. The debugger activates and automatically stops at the first line of our programme.



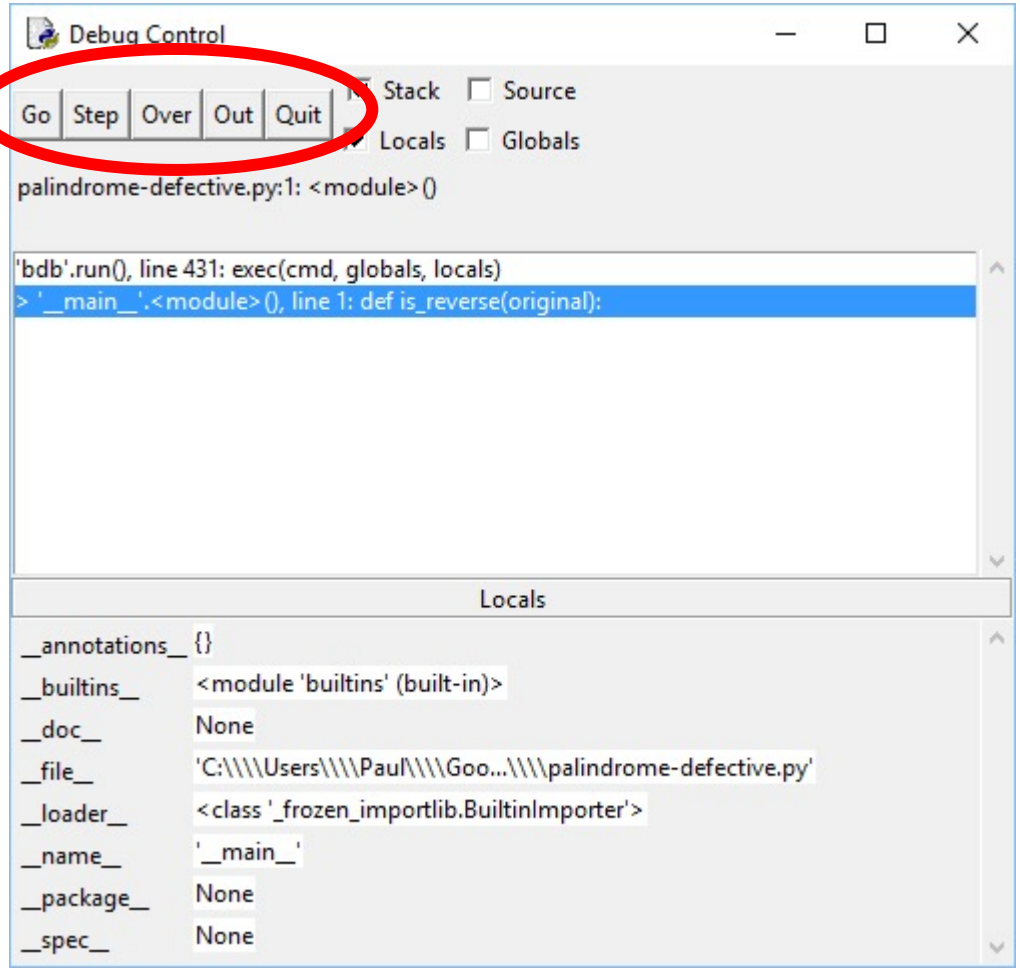
Tip: View the Source Code

2. It is usually helpful to have both the source code and the Debugger in view. (clicking the “source” box does this automatically)



Debugging the Code (3)

3. The 5 IDLE Debugger options.



Debugging the Code (3)

3. The 5 IDLE Debugger options.

Go - Executes the rest of the code as normal, or until it reaches a *break point* (we discuss break points later.)

Step - Step one instruction. If the line contains a function call, the debugger will *step into* the function.

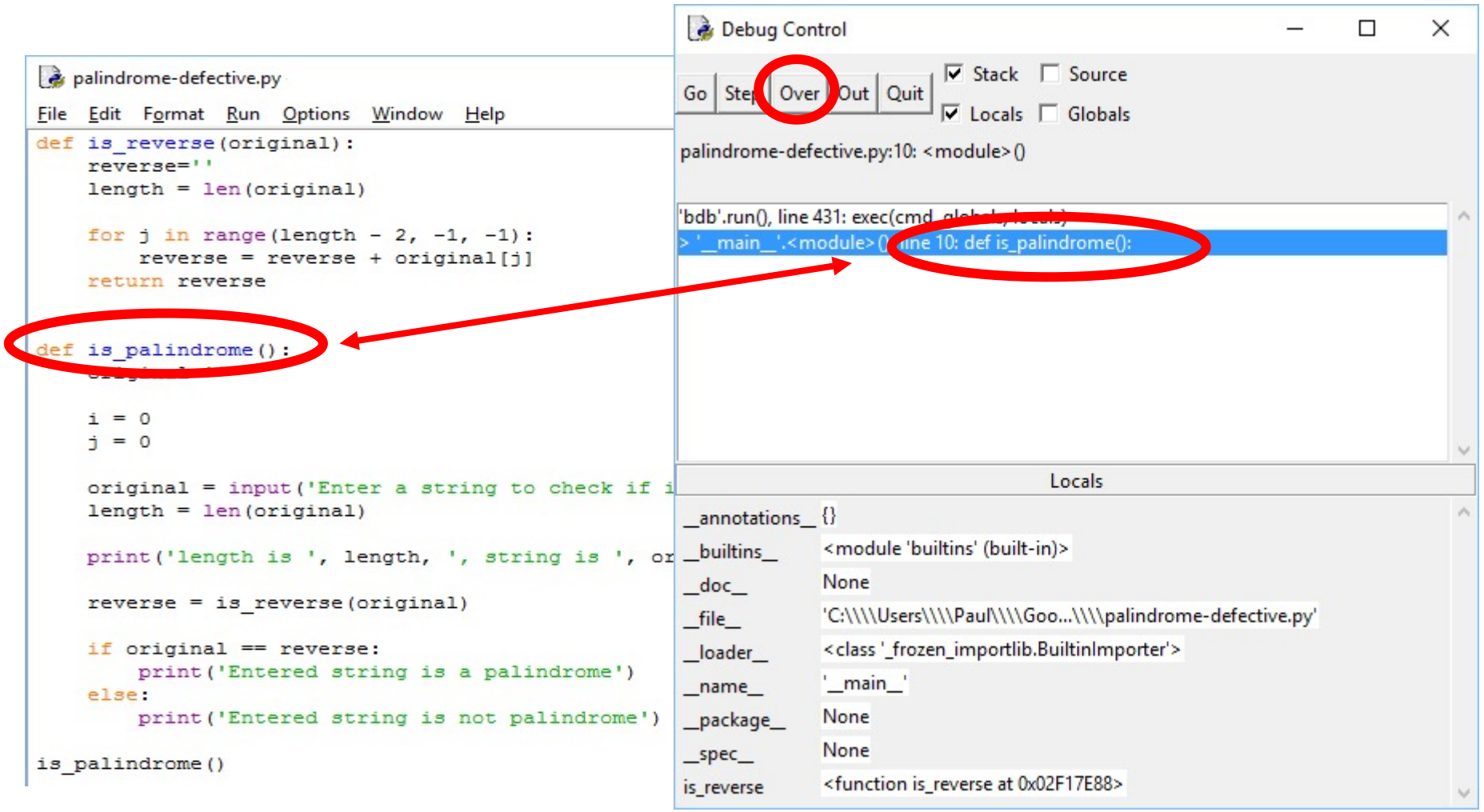
Over - Step one instruction. If the line is a function call, the debugger won't *step into* the function, but instead *step over* the call.

Out - Keeps stepping over lines of code until the debugger leaves the function it was in when **Out** was clicked. This *steps out* of the function.

Quit - Immediately terminates the program.

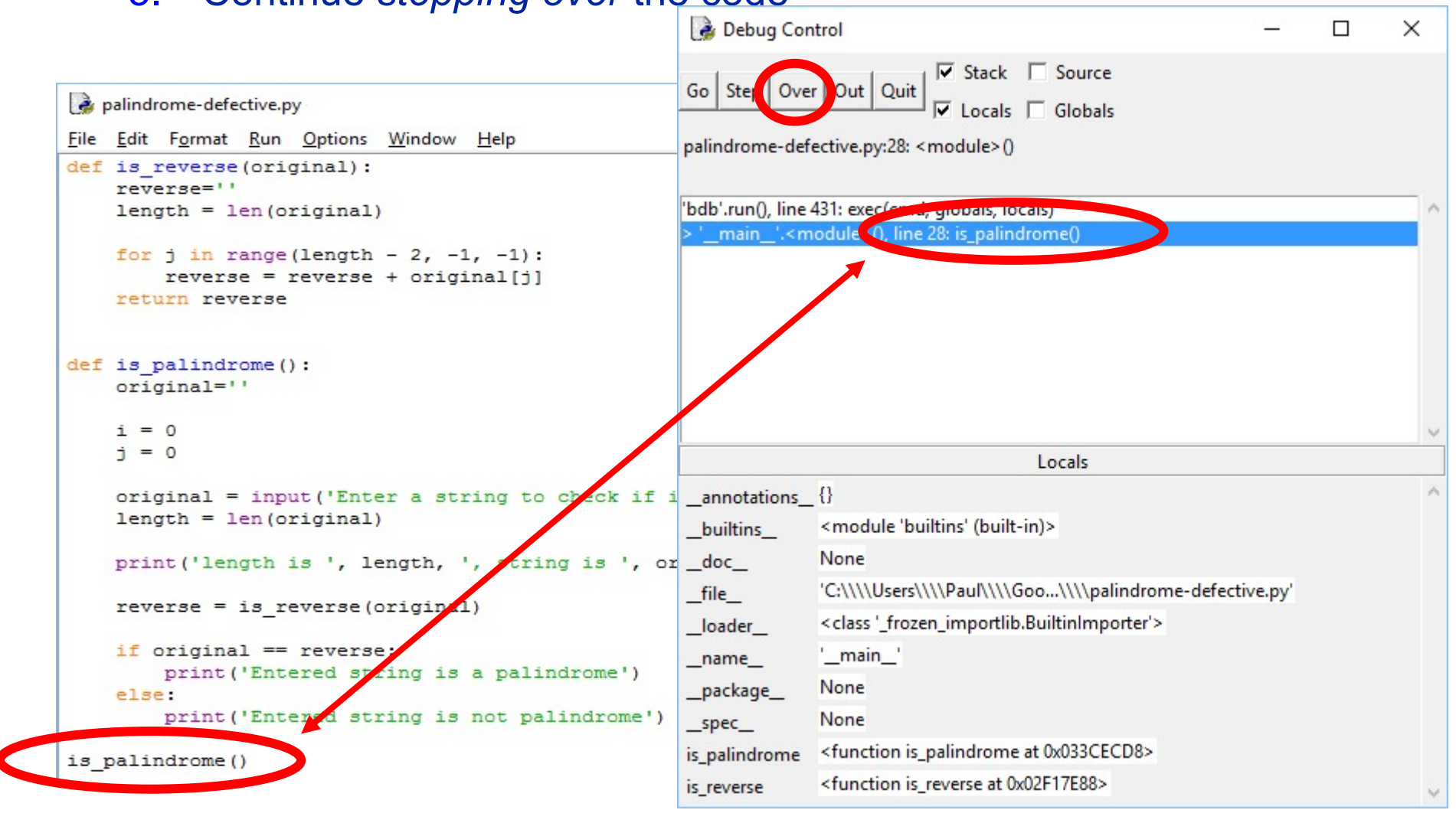
Debugging the code (4)

4. *Stepping over* the code



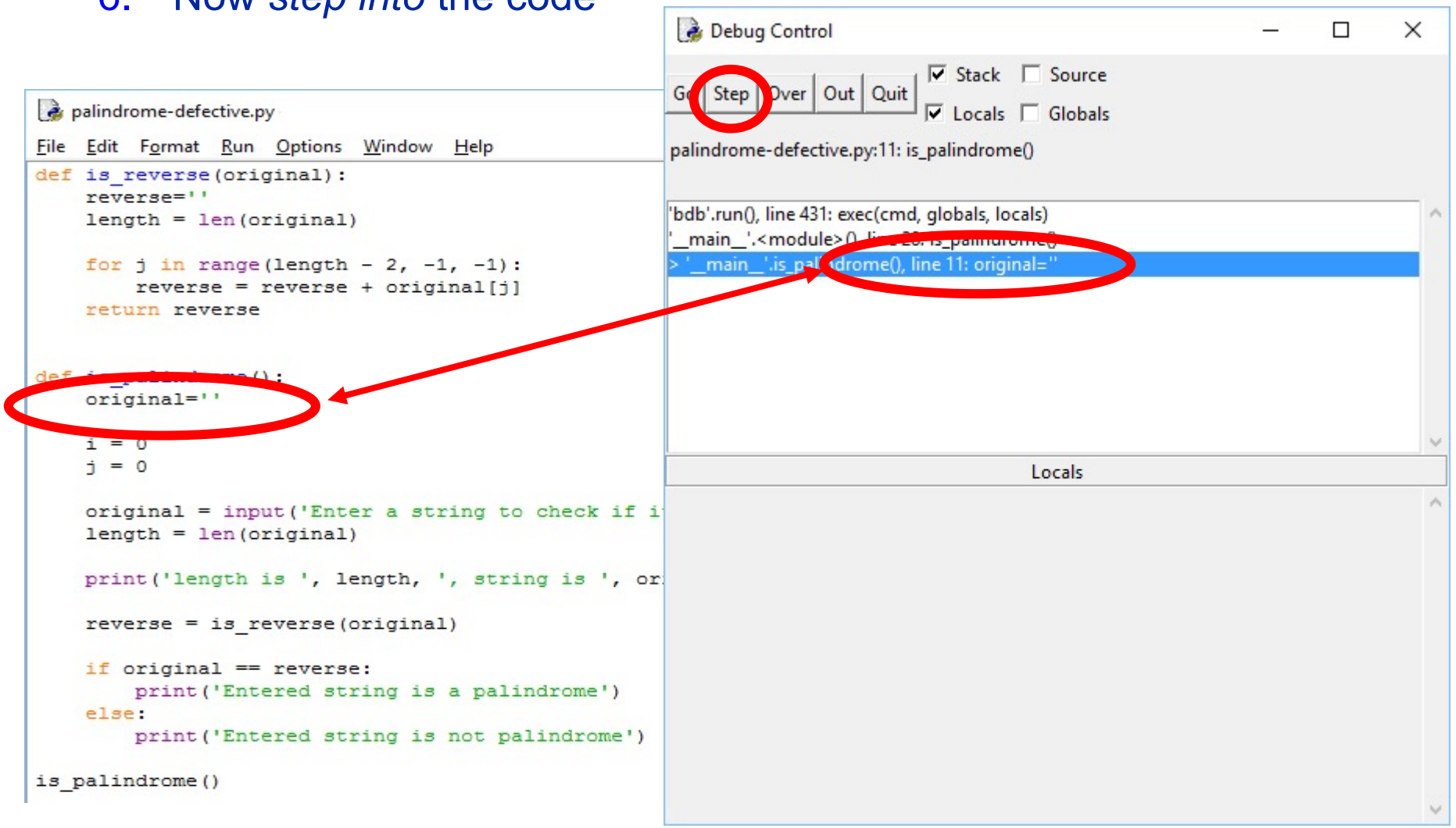
Debugging the code (5)

5. Continue *stepping over* the code



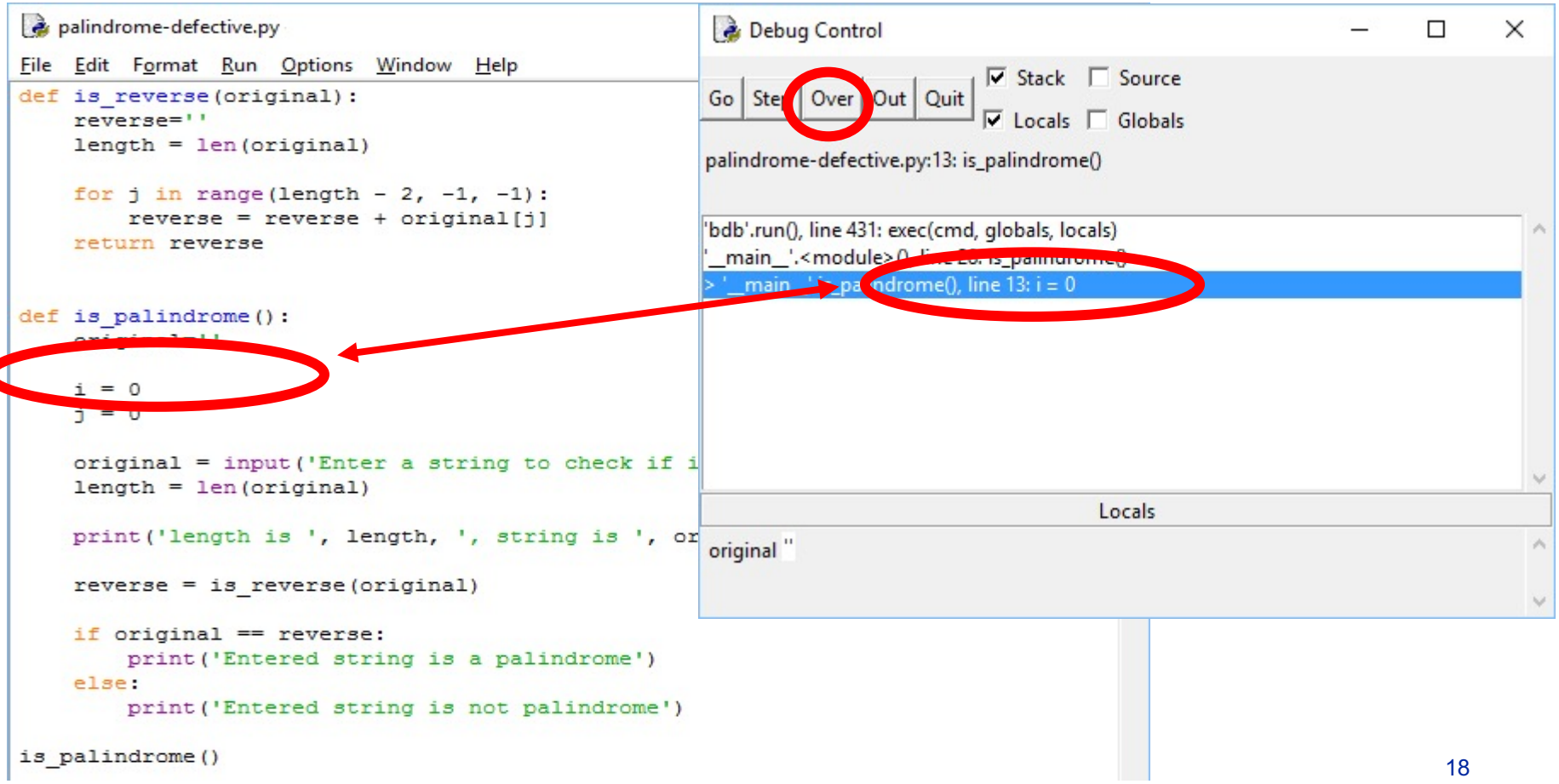
Debugging the code (6)

6. Now *step into* the code



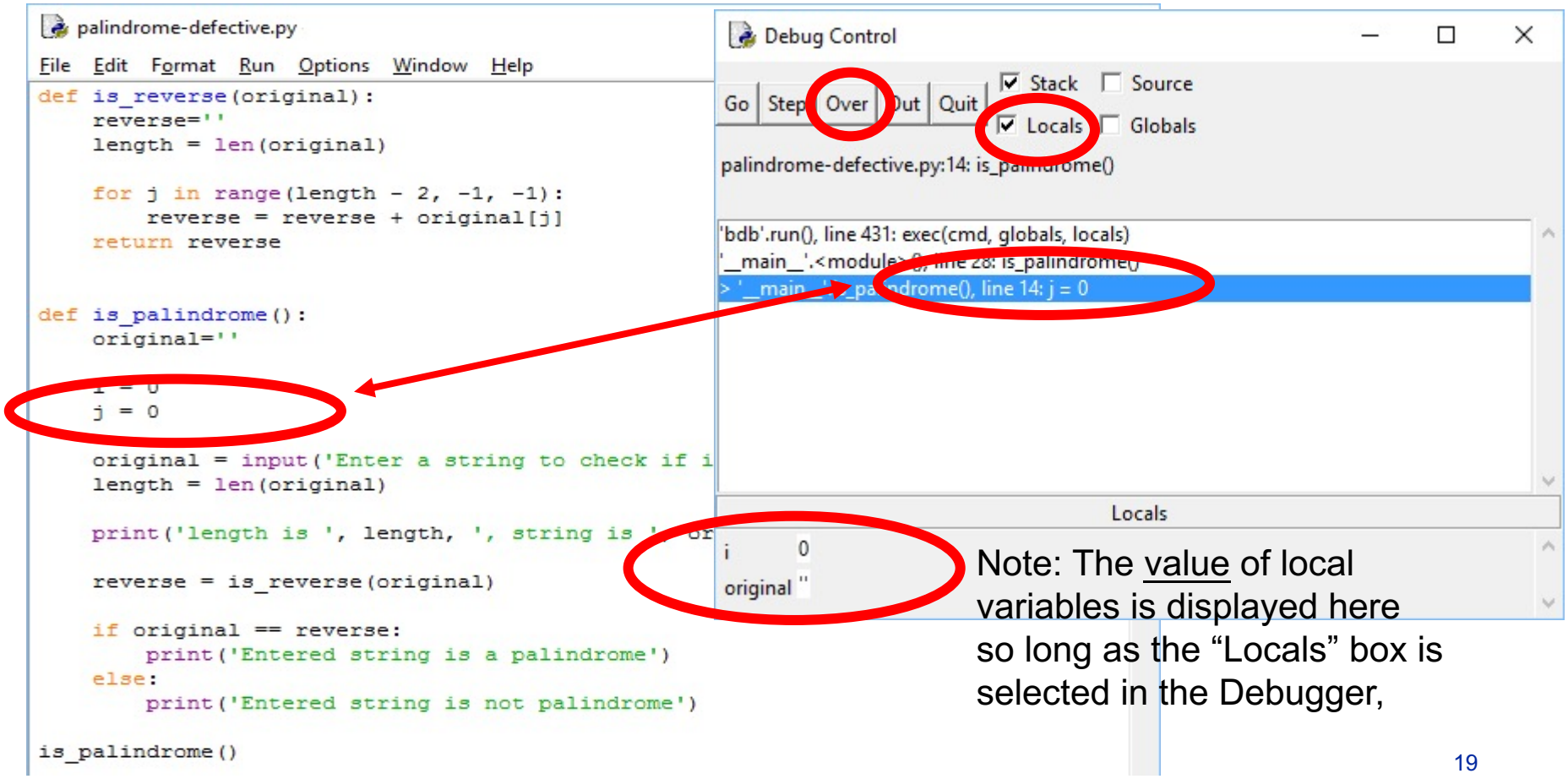
Debugging the code (7)

7. And now *step over* the code again…



Debugging the code (8)

8. And now *step over* the code again…



Note: The value of local variables is displayed here so long as the “Locals” box is selected in the Debugger,

Debugging the Code – breakpoints (i)

- But stepping *over* (and especially *into*) all of the code is not necessary to get to the problem area – and it is far too inefficient.
- In the case of our code, we know that there is an issue when checking if a string is a palindrome.
- We can therefore often ignore all of the code leading up to the problem area.
- This is achieved through the use of *breakpoints*:
  - First, we set a breakpoint in the part of the code of direct interest
  - Second, we jump directly to that breakpoint by using the “Go” option
  - Recall that “Go” will simply execute the code either to completion or to the first breakpoint it encounters.

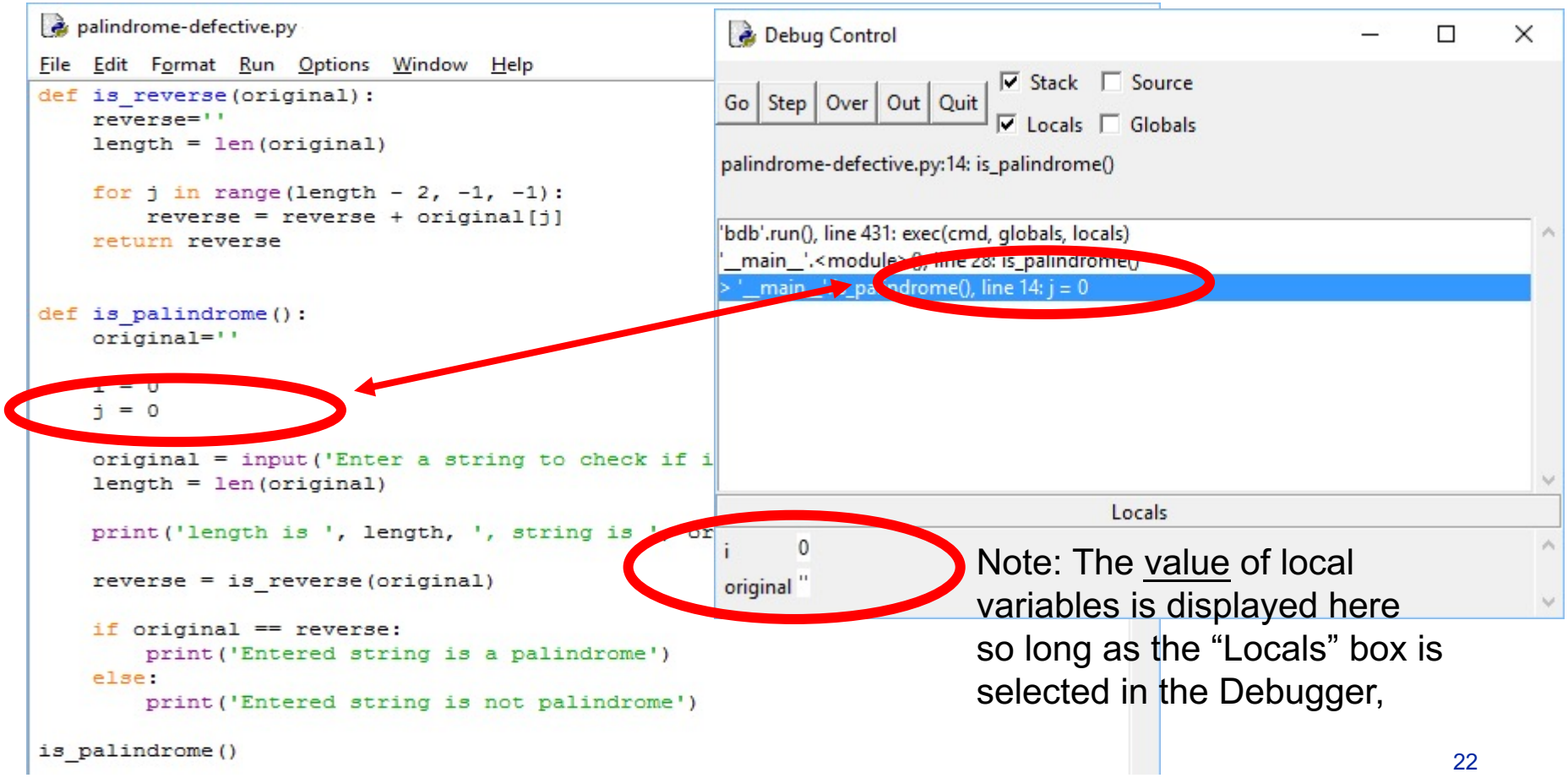
Debugging the Code – breakpoints (ii)

- To set a break point, right-click on the line in the file editor and select **Set Breakpoint** from the menu that appears.
- The file editor will highlight that line with yellow.
- You can set break points on as many lines as you want.
- To remove the break point, click on the line and select **Clear Breakpoint** from the menu that appears.

- In practice, programmers set the breakpoint(s) of interest without the need to step into or over any code that they feel will not bring them directly to the problem area… and keep doing this until the source of the defect is identified.

Debugging the code (9)

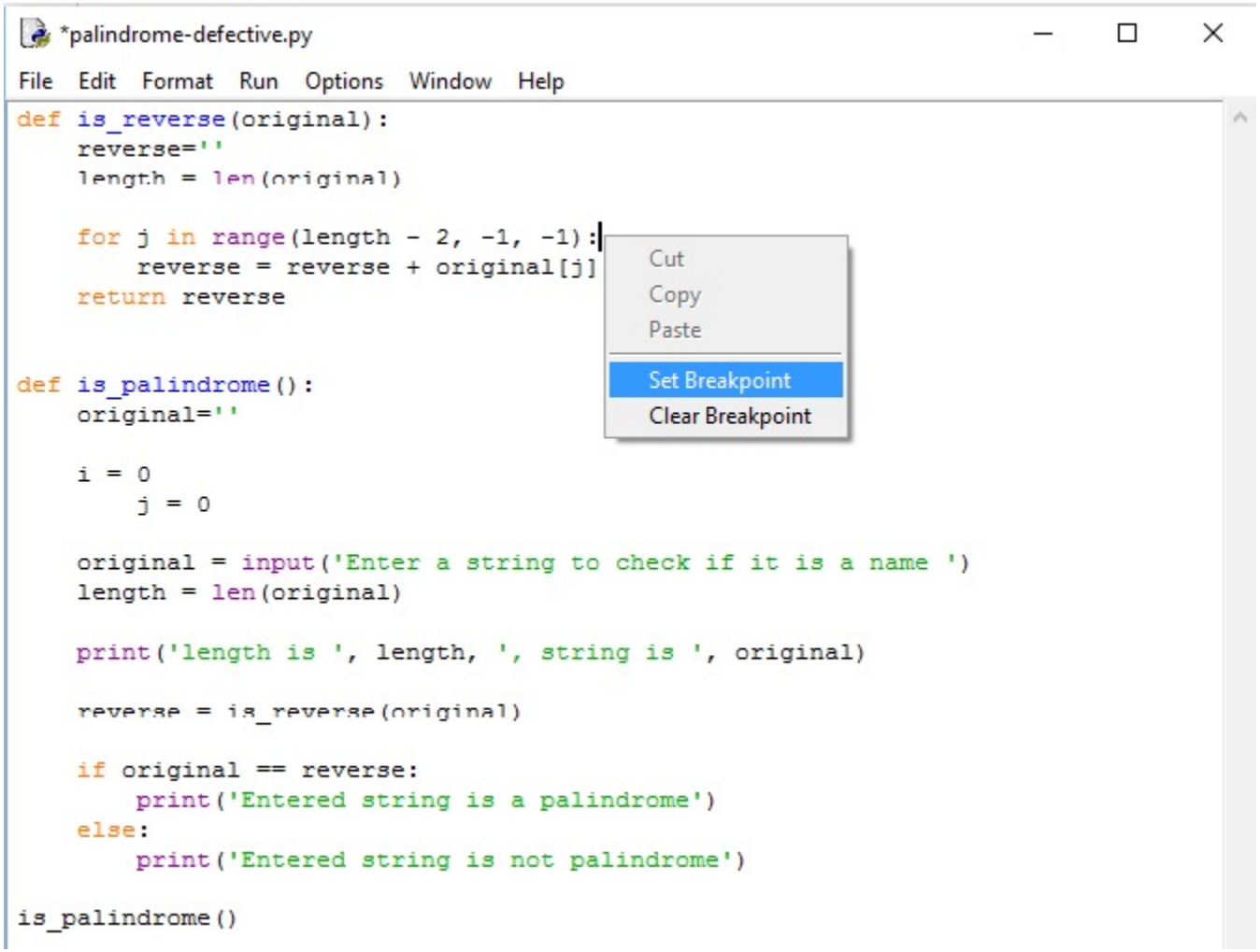
9. Setting our breakpoint… in the is_reverse() function



Note: The value of local variables is displayed here so long as the “Locals” box is selected in the Debugger,

Debugging the code (10)

10. Setting our breakpoint…



Debugging the code (10)

10. Our breakpoint is now set…

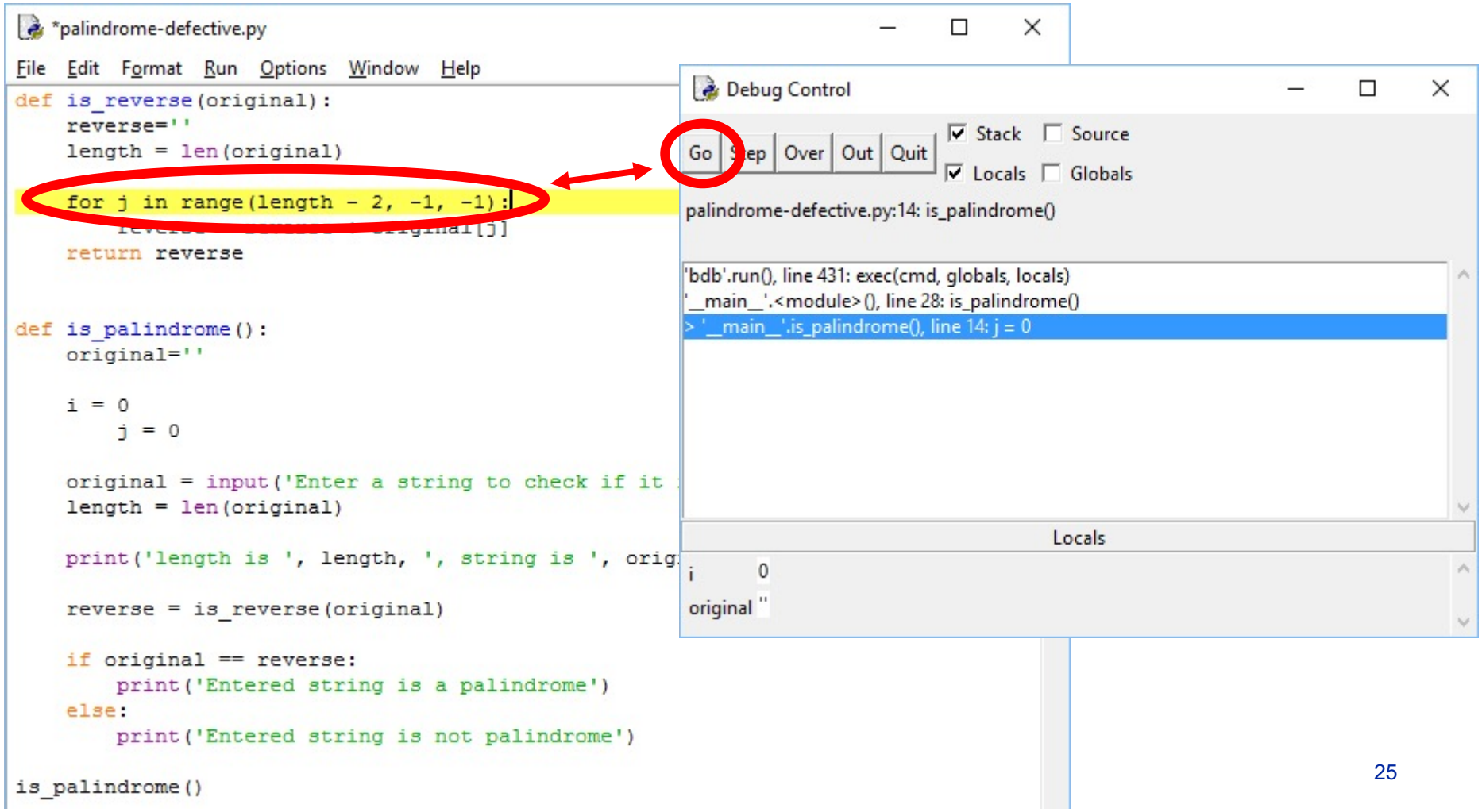
*palindrome-defective.py

File Edit Format Run Options Window Help

```
def is_reverse(original):
    reverse=''
    length = len(original)

    for j in range(length - 2, -1, -1):
        reverse = reverse + original[j]
    return reverse


def is_palindrome():
    original=''

    i = 0
        j = 0

    original = input('Enter a string to check if it is a name ')
    length = len(original)

    print('length is ', length, ', string is ', original)

    reverse = is_reverse(original)

    if original == reverse:
        print('Entered string is a palindrome')
    else:
        print('Entered string is not palindrome')

is_palindrome()
```

Debugging the code (11)

11. “Go” directly to our breakpoint

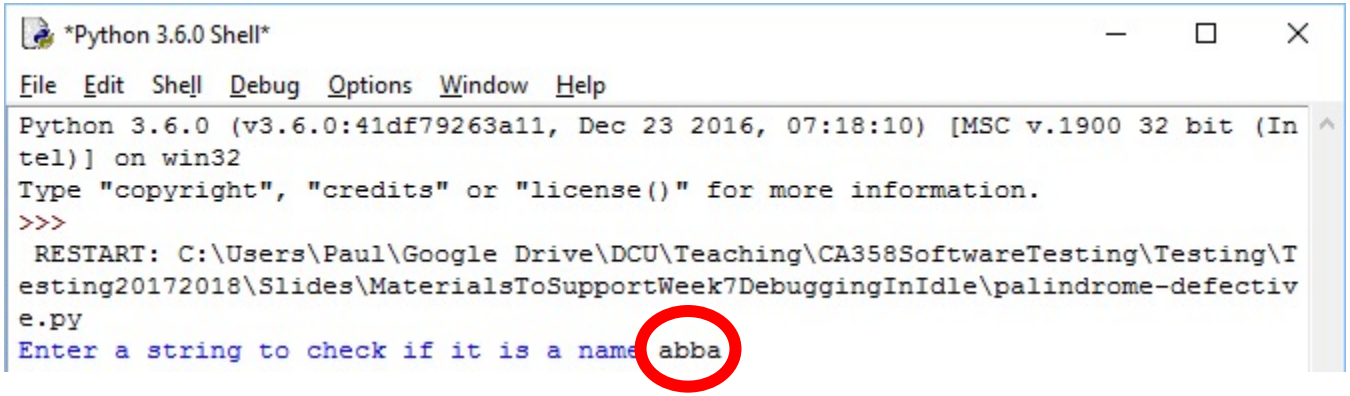


Debugging the code (12)

12. In the case of our code, going directly to our breakpoint now involves entering a candidate string in the Python Shell.

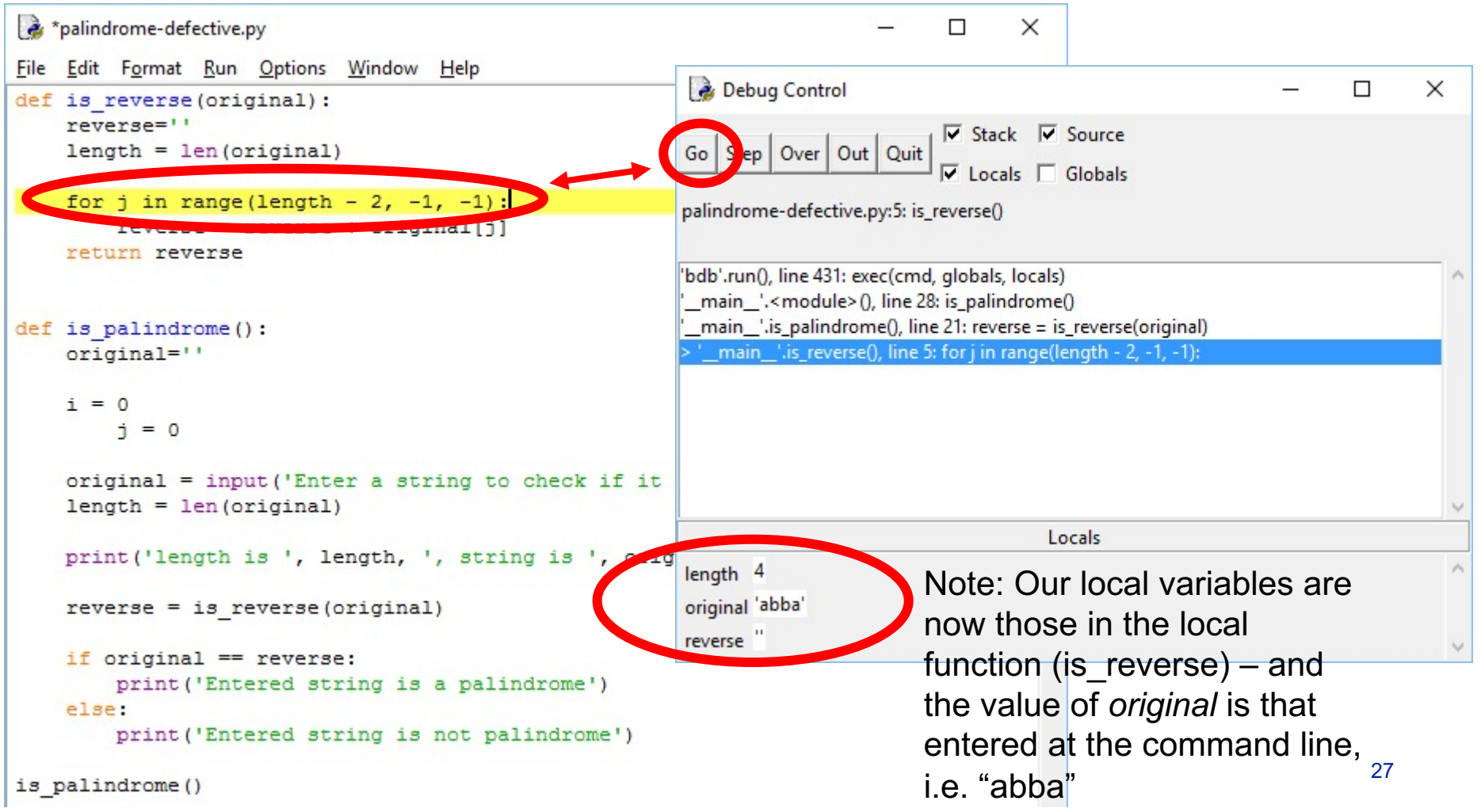
    We enter the string “abba”…

    DON’T FORGET TO ENTER A STRING!



Debugging the code (13)

13. Debugger takes us directly to our breakpoint



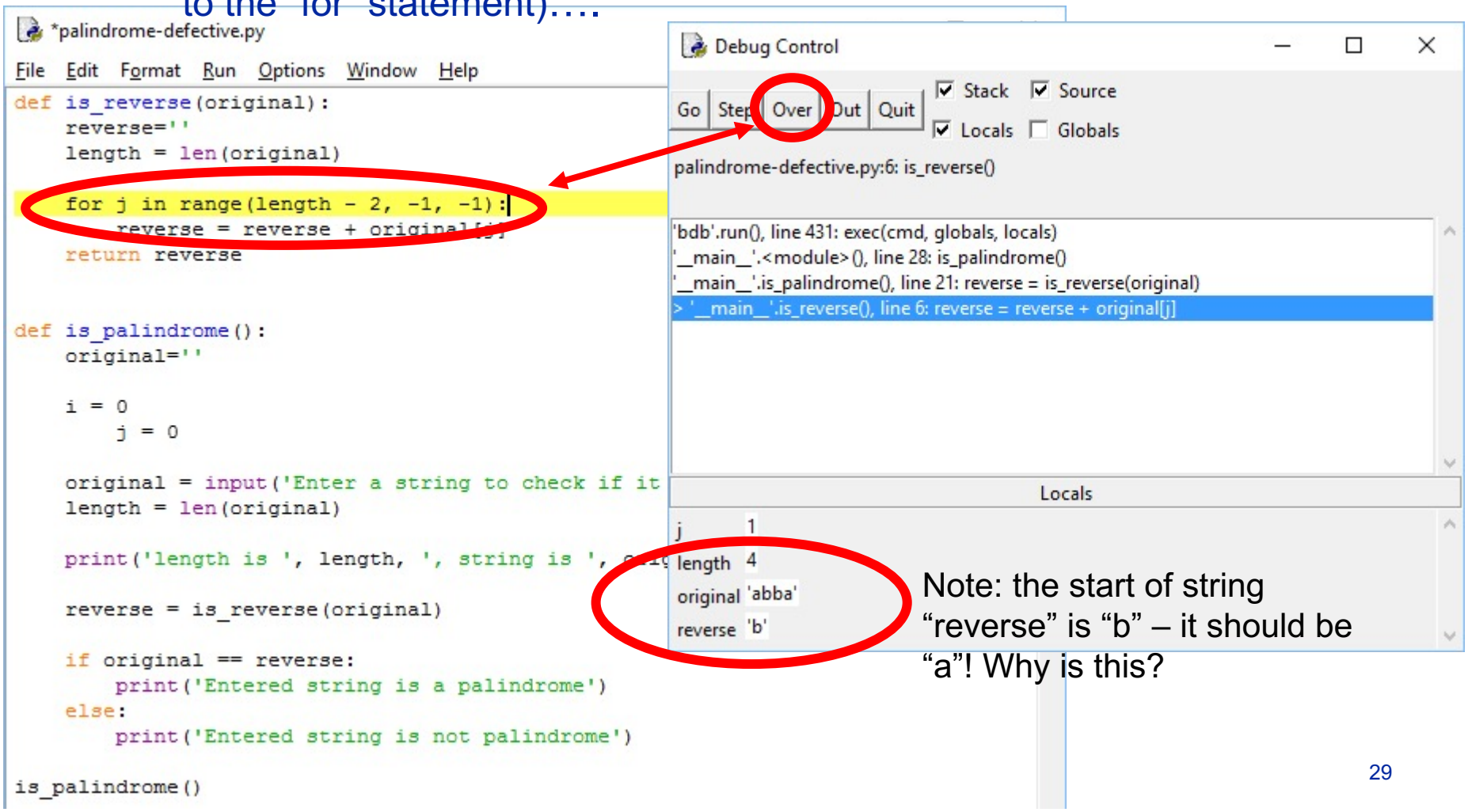
Note: Our local variables are now those in the local function (is_reverse) – and the value of *original* is that entered at the command line, i.e. “abba”

Debugging the problem

- Now that we have got close to where we feel the source of the problem exists, we want to:
  - Step over the lines of code and examine the changes to the local variables
  - In particular in our case, we are interested to see how the string *reverse* is built, as it is a strong potential source for our defect.
  - So, we step *over* the code….
  - And immediately we notice an issue – the first letter in reverse is “b” and not “a” as was expected.

Debugging the code (13)

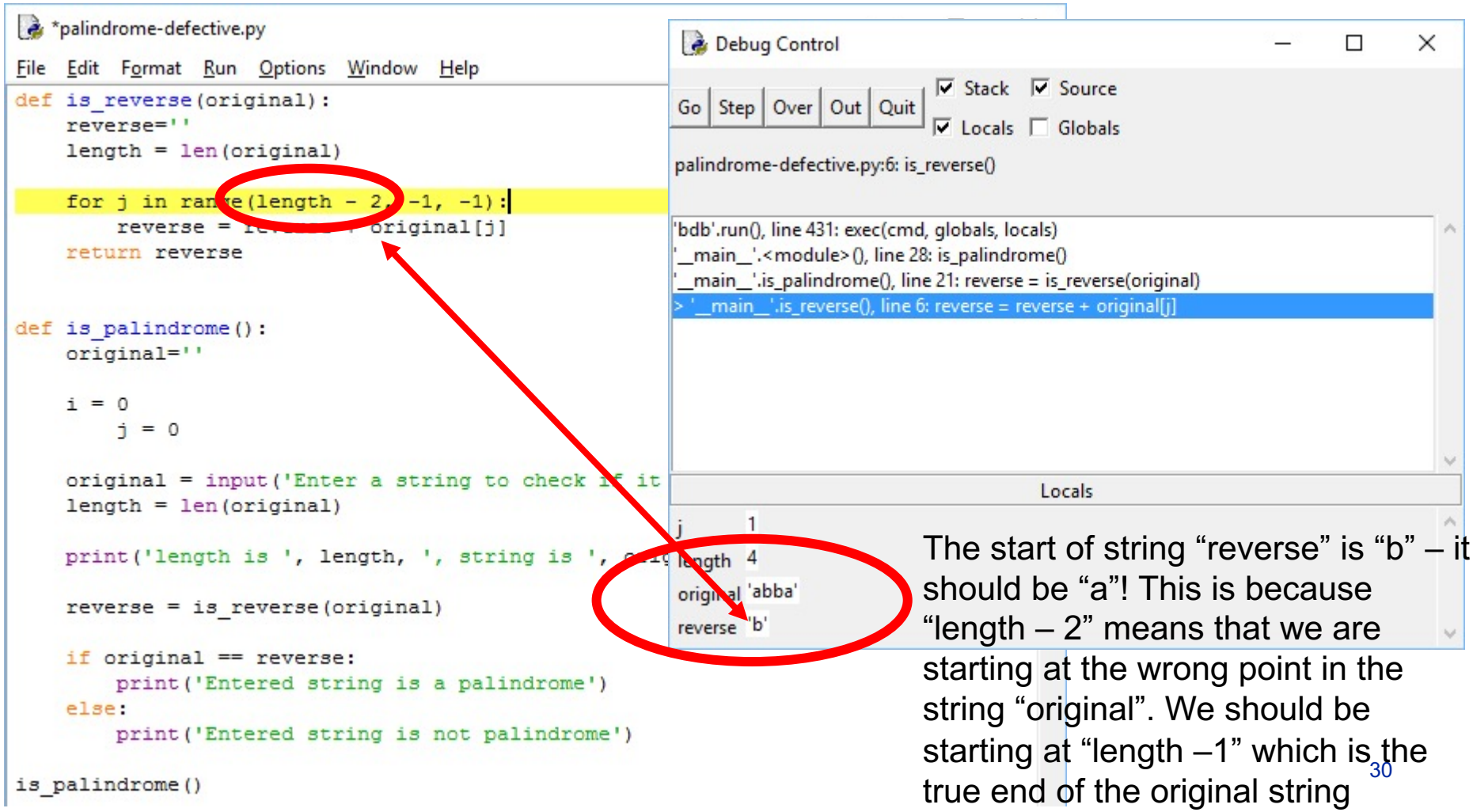
13. Step *over* the code twice (i.e. click “step” twice thereby bringing us back to the “for” statement)….



Note: the start of string “reverse” is “b” – it should be “a”! Why is this?

Debugging the code (14)

14. The source of the defect is revealed



The start of string “reverse” is “b” – it should be “a”! This is because “length – 2” means that we are starting at the wrong point in the string “original”. We should be starting at “length –1” which is the true end of the original string

Debugging the problem

- And we can use this information to fix the defect.
- Change “length – 2” to “length -1”.
- Run the program and see if it now works…. And it does.
- Most programmers would clear the breakpoint – and probably also exit the debugger at this point and simply execute the code as normal, expecting the defect to be fixed.

- This is the power of the debugger!

Over to you

Any Questions?