universität
innsbruck

198.801

Introduction to Programming:

Programming in Python

Non-interactive Testing with DocTest

Joanna Chimiak-Opoka, PhD
University of Innsbruck, Digital Science Center

Agenda

- Overview of testing concepts and methods
- Approaches to define a test suite
- Technical part: how to use DocTest
- Further remarks

INTUITION

behind

TESTING

Cooking Soup Analogy

What to do when you are cooking a soup and are **bugs falling down** from the ceiling?

- declare it as a high protein soup
  → calling bug a feature
  it is a bad practice!
- check soup for bugs
  → **testing**
- remove bugs from soup
  → **debugging**
- keep lid closed
  → **defensive programming**
- clean kitchen
  → **eliminate source of bugs**
  ideal, but not realistic

Photo © Louis R

analogy by Srini Devadas, after Ana Bell

Precision Levels

What would you say
*about **color of cows** in Scotland?*

Person:
The cows in Scotland are brown.

Logician:
No, there are cows in Scotland
of which at least one is brown!

Computer Scientist:
No, there is at least one cow in Scotland,
of which one side appears to be brown!!

Program Verification
Introduction

Reiner Hähnle

Everyday examples of error detection

critical numbers such as

credit cards or **bar-codes** have

check digits to detect typical mistakes with high probability

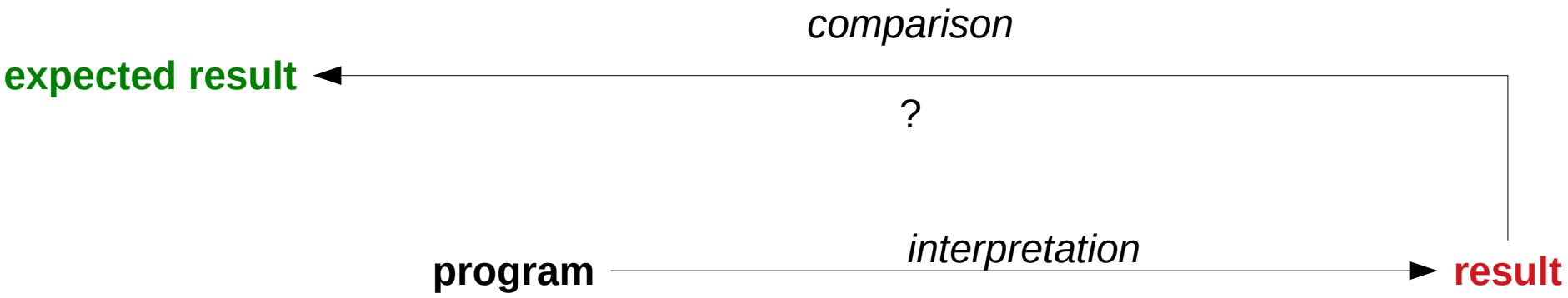
The above example barcode is from a product that a student might have selected, so they would give you these 12 digits (they should keep the 13th digit secret):

94005470098?

http://www.financetwitter.com/2014/08/cracking-sixteen-digits-credit-card-numbers-what-do-they-mean.html →

Testing results of a program

Testing a program against expected/known values



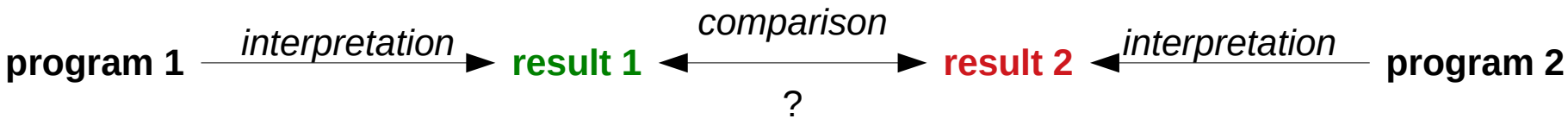
Conditionals with Cards

```
If (CARD is RED or CARD > 9)
        Award YOUR team 1 point
Else
    If (CARD is Ace and CARD is Hearts)
            Award YOUR team 2 points
    Else
            Award OTHER team 1 point
```

Card	Expected Result
A ♥	+1
A ♦	+1
K♠	+1
9 ♦	+1
9 ♠	-1

Testing results of two programs

Testing a program against another program



Conditionals with Cards

```
If (CARD is RED or CARD > 9)
        Award YOUR team 1 point
Else
        If (CARD is Ace and CARD is Hearts)
                Award YOUR team 2 points
        Else
                Award OTHER team 1 point
```

Initial	Card	Simplified
+1	A ♥	+1
+1	A ♦	+1
+1	K ♠	+1
+1	9 ♦	+1
-1	9 ♠	-1

```
If (CARD is RED or CARD > 9)
        Award YOUR team 1 point
Else
        Award OTHER team 1 point
```

FROM PREVIOUS UNITS:

The process of code improvement without changing its functionality is called **refactoring** and should be supported by **tests**.

Testing shows that a program is **probably correct**.
The better the **quality of the testing sample** the higher the probability of correctness.

Error detection

Everyday examples of error detection:

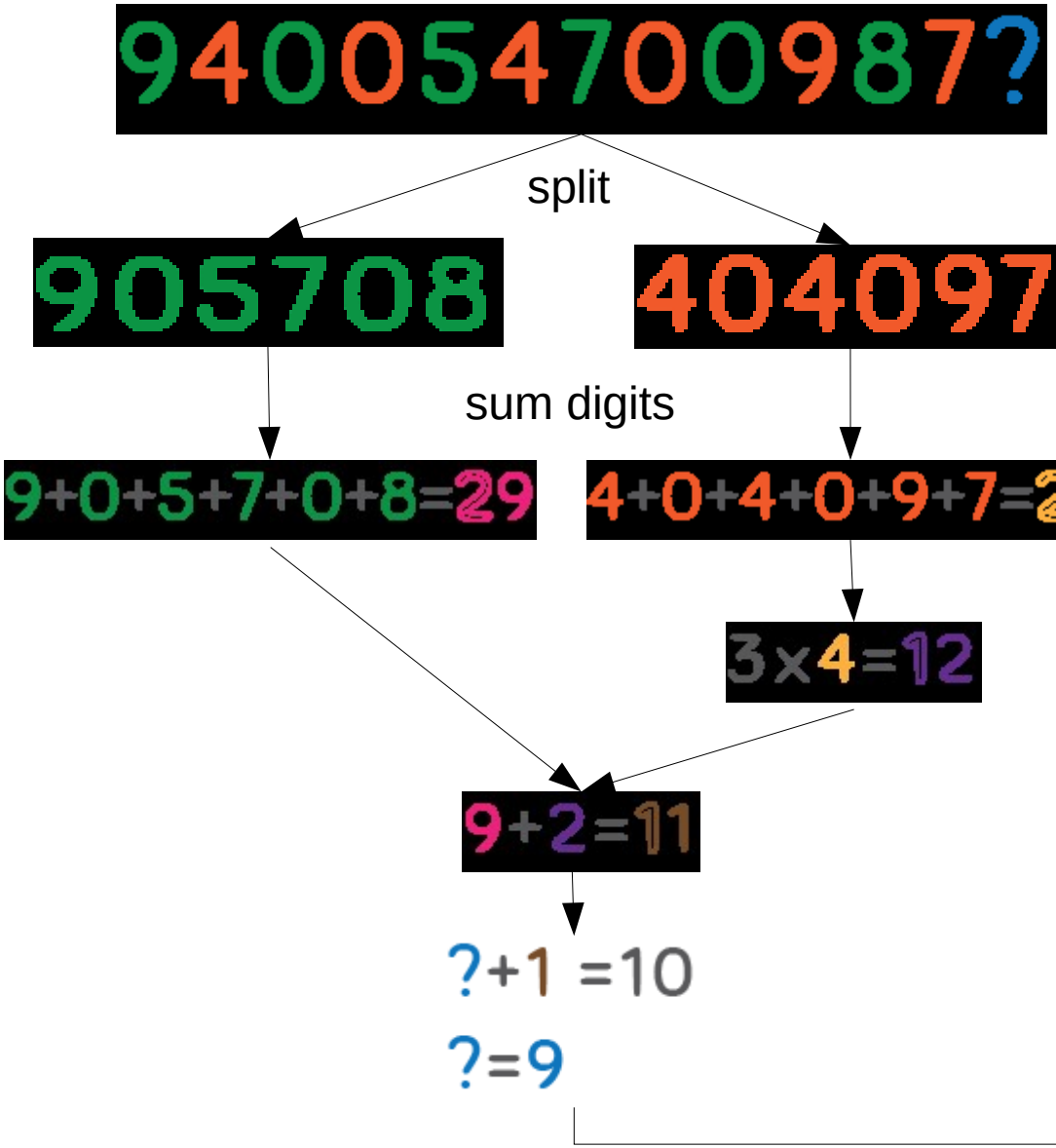
critical numbers such as **credit cards**
or **bar-codes** have **check digits**
to detect typical mistakes in the numbers
with high probability

The above example barcode is from a product that a student might have selected, so they would give you these 12 digits (they should keep the 13th digit secret):

94005470098?

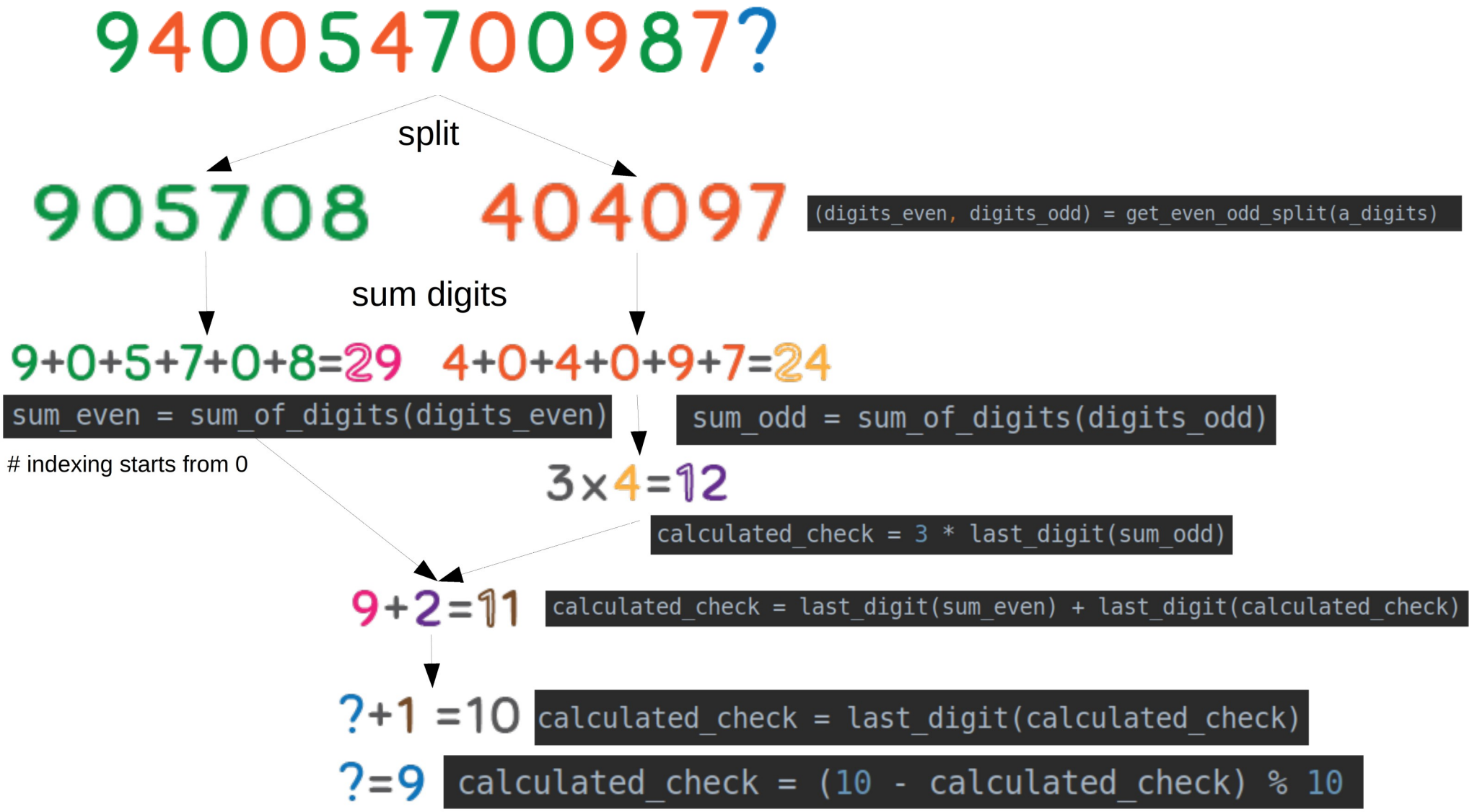
http://www.financetwitter.com/2014/08/cracking-sixteen-digits-credit-card-numbers-what-do-they-mean.html →

Example calculation



The link
to the description
of the algorithm
is in OLAT

Example calculation



Algorithm usage and analysis

Work in pairs:

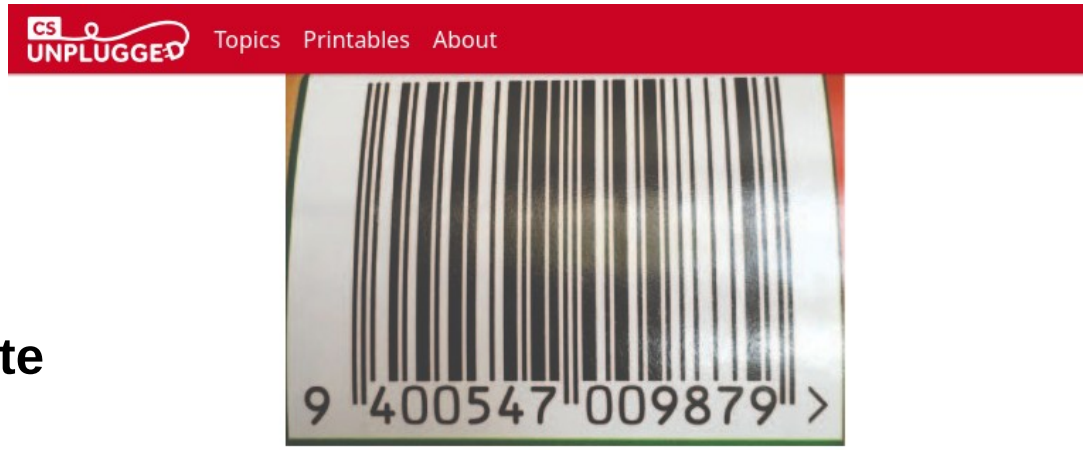
- give each other
  a product number
  without last digit

  and let the other person **calculate**
  it according to
  the algorithm

  and **compare** your results with the
  actual last digit of the barcode

- give each other a product number
  with the last digit but **with an error**

  and let the other person calculate
  the check sum to **detect** a
  problem



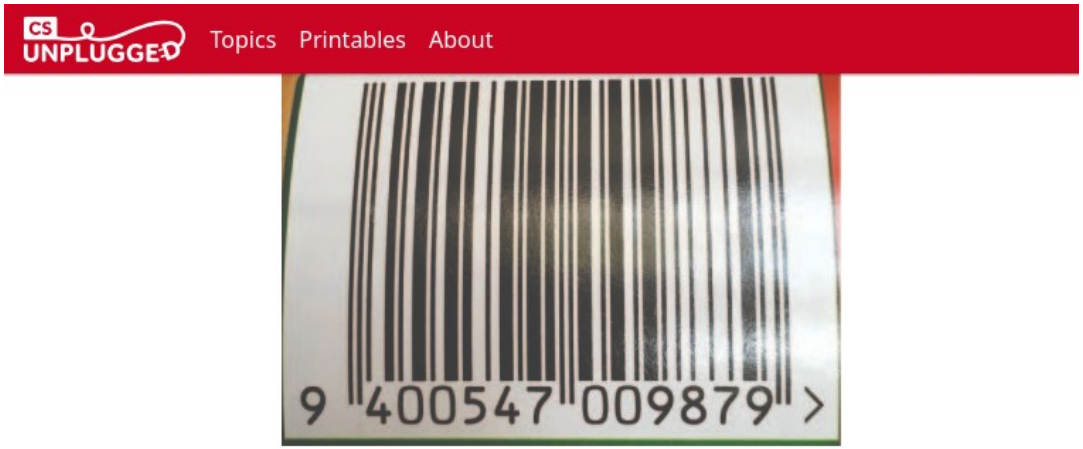
The above example barcode is from a product that a student might have selected, so they would give you these 12 digits (they should keep the 13th digit secret):

940054700987?

- **Discuss**:
  - What kind of errors the checksum is **guaranteed to detect** and
  - what kind of them **wouldn't be detected**?
  - Give examples and check them.

Possible mistakes

- A digit is inserted in the number.
- A digit is removed from the number.
- A digit has its value changed.
- Two adjacent digits are swapped with each other.
- Two adjacent digits are modified.
- More digits are modifies.



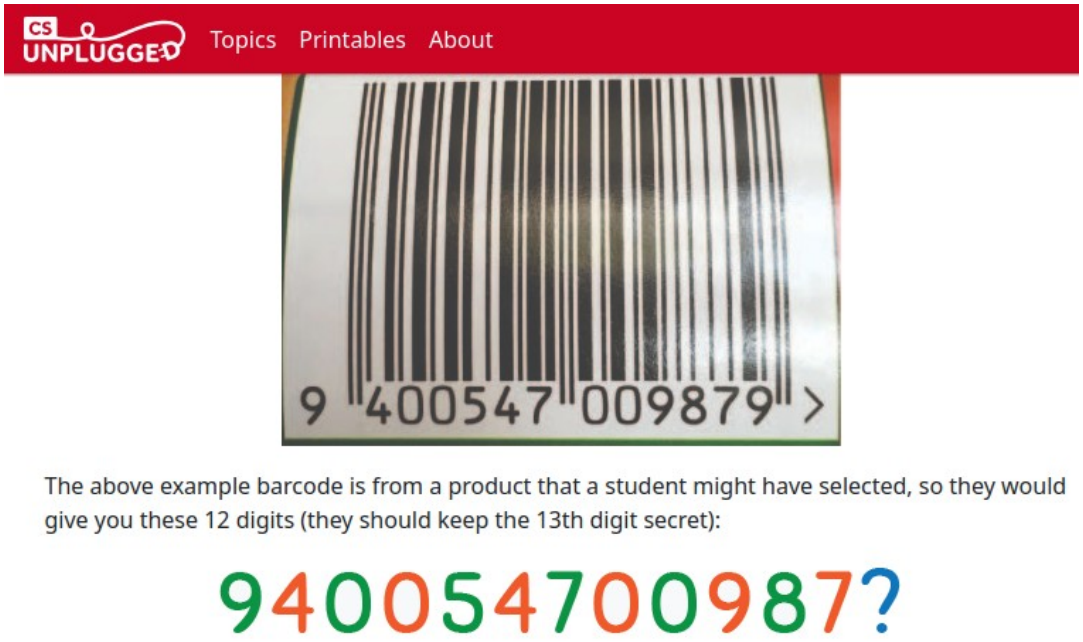
The above example barcode is from a product that a student might have selected, so they would give you these 12 digits (they should keep the 13th digit secret):

940054700987?

Calculate check sums for your examples
or use the program (from course material) to see if your guess was correct.

Error detection

940054700987	→ 9	✔	☺
940054100987	→ 5	✘	☺
940054190987	→ 8	✘	☺
940054120987	→ 9	✔	☹
940054100937	→ 0	✘	☺
940154790987	→ 9	✔	☹



The above example barcode is from a product that a student might have selected, so they would give you these 12 digits (they should keep the 13th digit secret):

940054700987?

- **Error detection:** *One digit* errors can be always detected. Mistakes *in two consecutive digits* in most* cases can be detected, but some multi-digit mistakes can *compensate*.
- **Abstract explanation**: when calculating a check sum a big part of information is lost, thus many** barcodes have the same last digit.
- **Pragmatic approach**: reasonable detection rate for the most probable mistakes.

*) 9:10 **) without any restriction on a product number: 10^{12}:10^{1}

TESTING

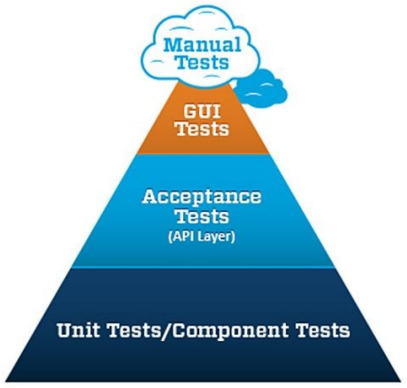
ASPECTS

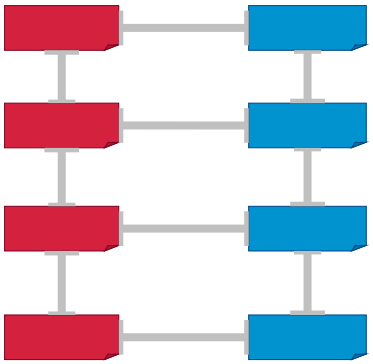
Aspects of Testing

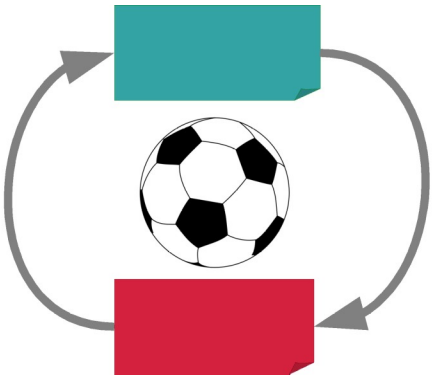
For better understanding, we need to look at the software testing from different perspectives.

Let‘s look at the following aspects:

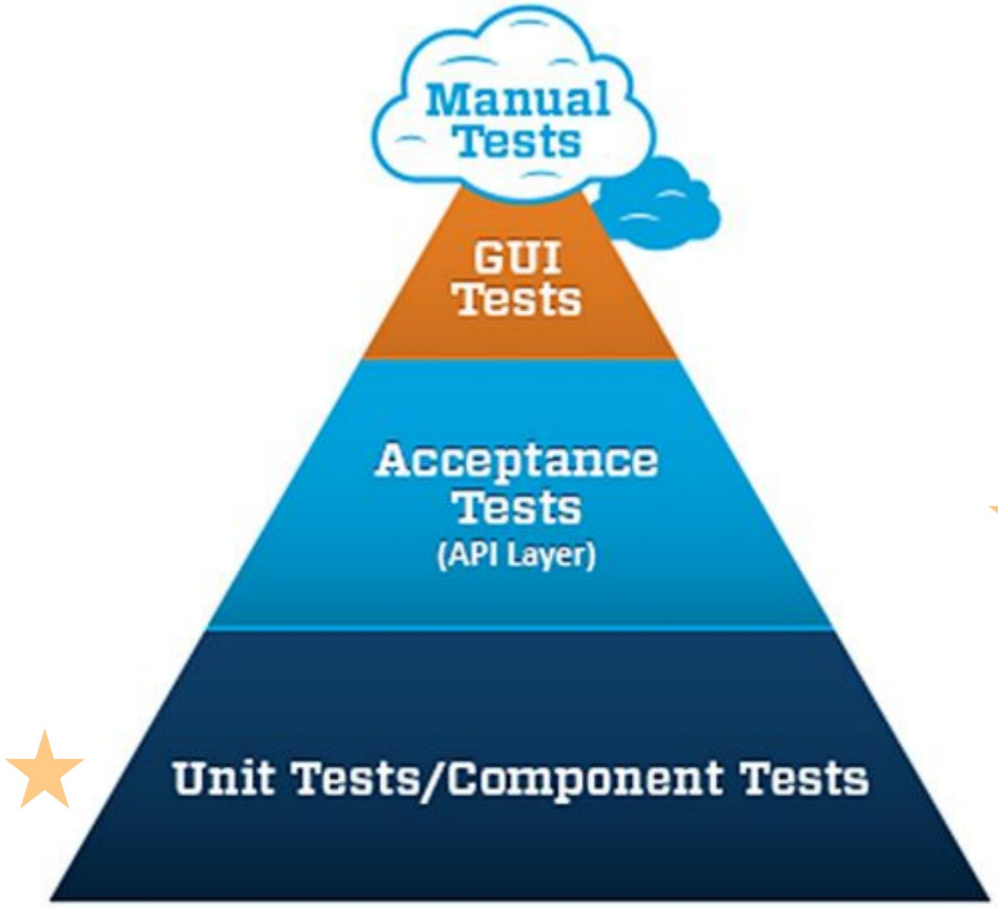
- level of automation
- scope of testing
- testing workflow
- extend of testing







Level of Automation



[QATestLab]

In **manual testing**,
a tester to play the role of an end user
whereby he uses most of the program's
features to ensure correct behavior.

Automated testing is the practice of
using software to test software.

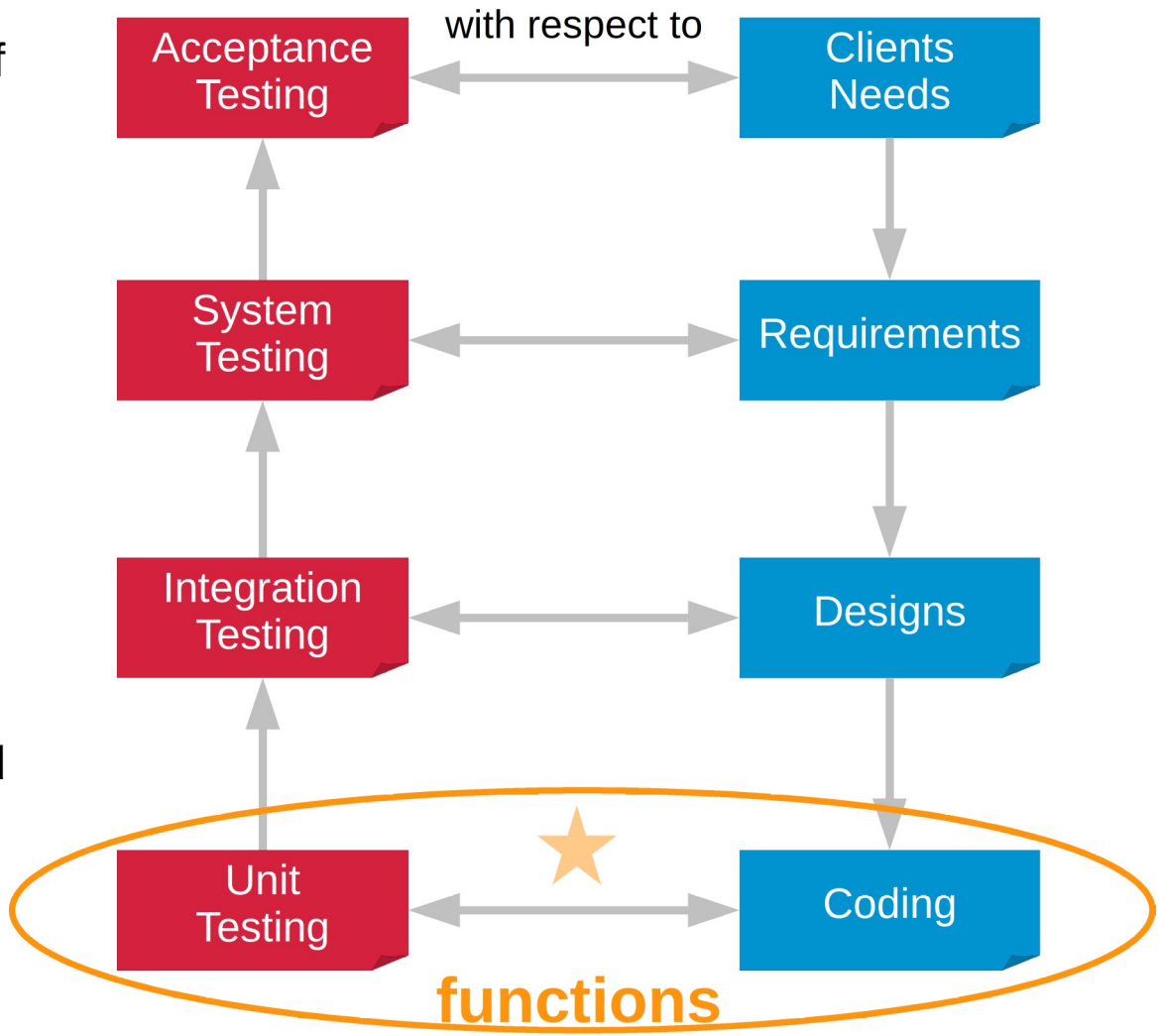
An automated test case consists of
a series of **commands**
that execute the steps of a test,
as well as any **data** needed for the test
and the **expected result**.

Scope of Testing

Software testing involves the execution of a software component to indicate the extent to which the component or system under test:

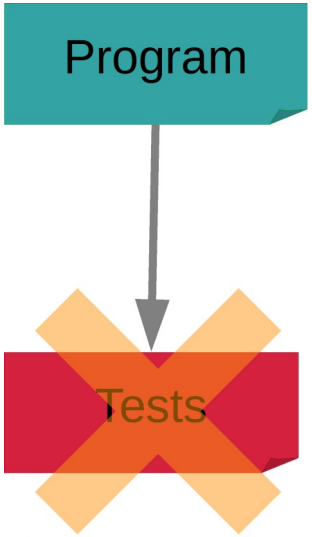
- **meets** the requirements that guided its design and development,
- **responds** correctly to all kinds of inputs,
- **performs** its functions within an acceptable time,
- it is sufficiently **usable**,
- can be **installed** and run in its intended environments, and
- **achieves** the general result its stakeholders desire.

[Wikipedia]



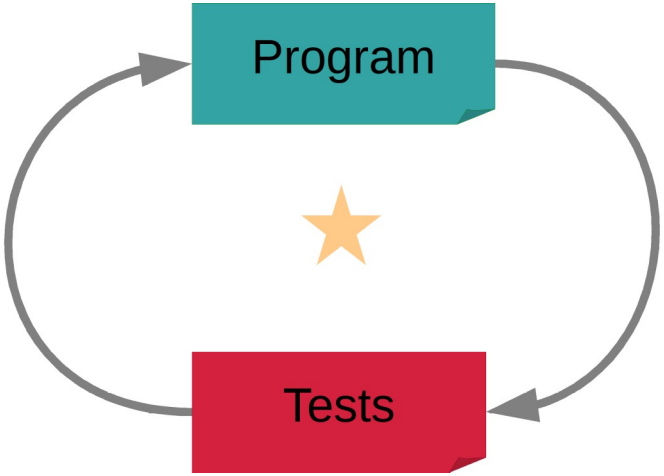
As in V-model of Software Development

Testing Workflow



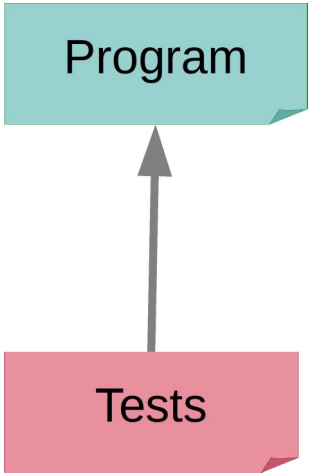
Program first
Test next

e.g.
in V-model,
waterfall model
…
no tests!



Incremental programming
and testing

e.g.
test-driven development
regression testing
before refactoring



Define **tests** first
Write program next

not useful/applicable

Extend of Testing

The test intensive approaches are in safety- or mission-critical systems (e.g. nuclear power plant control) or in... our course.

Pragmatic, minimal approach would be to use

Regression testing
- add **tests for bugs** as you find them
- catch reintroduced errors
  that were previously fixed

Testing for **refactoring**
- writing tests to cover behavior of a working program
  before modifying it
- remember to **be critical**
  when reusing program's results in tests

Approaches to define a

TEST SUITE

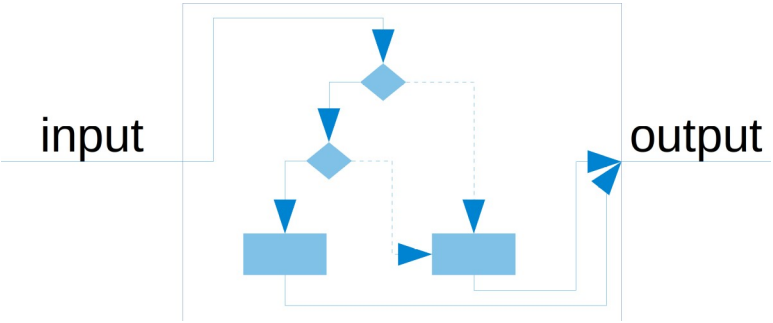
White and Black Box

Example methods to design a test suite:

White-box testing
(knowing the source code of the program)

→ Statement Coverage

→ Branch Coverage



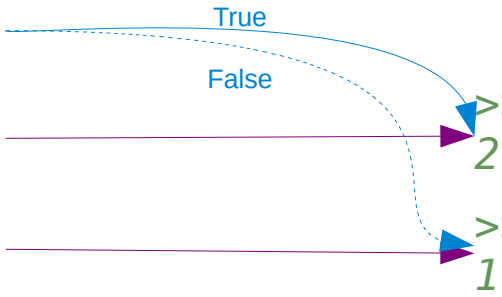
Black-box testing
(knowing restrictions on the program)

→ Boundary-value analysis

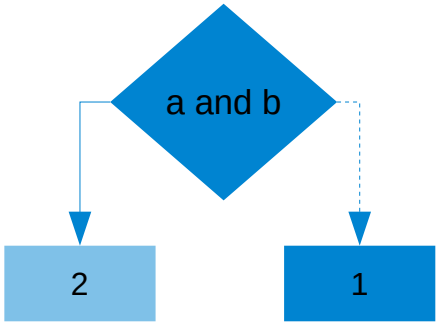
Statement and Branch Coverage

```
def ab_fun(a, b):
    if a and b:
        return 2
    return 1
```

STATEMENT COVERAGE == BRANCH COVERAGE



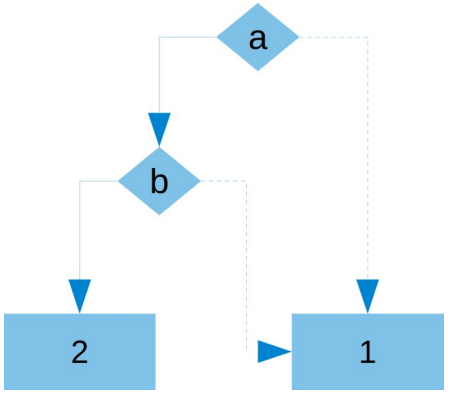

```
>>> ab_fun(True, True)
2
>>> ab_fun(True, False)
1
```




```
def a_b_fun(a, b):
    if a:
        if b:
            return 2
    return 1
```

STATEMENT COVERAGE < BRANCH COVERAGE

```
>>> a_b_fun(True, True)
2
>>> a_b_fun(True, False)
1
>>> a_b_fun(False, True)
1
```



Boundary-value analysis (1)

specification

(conditions)

▼

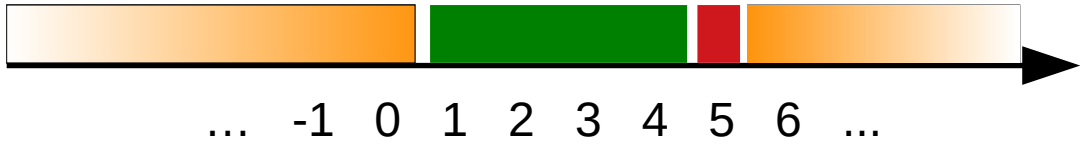
partition

▼

test values

a valid grade is from 1 to 5

a positive grade is lower then 5



0 1 4 5 6

Boundary-value analysis (2)

Rules for determining if a grade is positive

based on number of points in test(s)

1 test

2 tests

invalid points from a test are

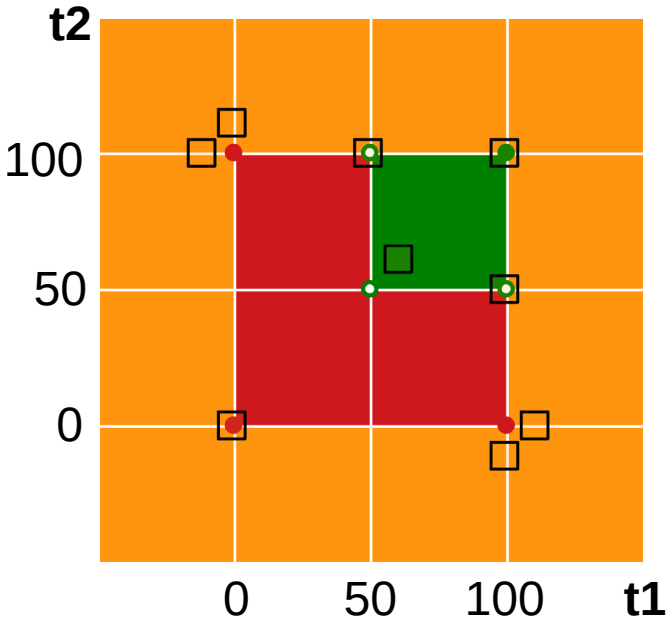
outside a range from **0 to 100 points**

for **a positive grade**

both tests must be **above 50 points**

for **the negative grade**

any of tests must be **50 points or lower**



t1, t2	t1, t2	comments	
101, 0	**0, 101**	the minimal value above the possible (valid) maximal number of points for a test	☐
-1, 100	**100, -1**	the maximal value below the possible (valid) minimal number of points for a test	☐
51, 51		the minimal values of points for a positive grade for both tests	☐
100, 100		the maximal values of points for a positive grade for both tests	☐
50, 100	**100, 50**	the maximal value of points for the negative grade for a test	☐
0, 0		the minimal values of points for the negative grade for both tests	☐

Boundary-value analysis (3)

Input:

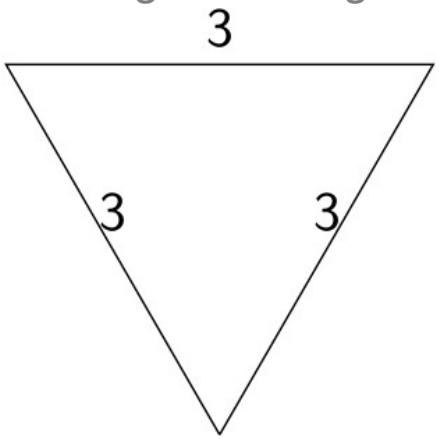
3 numbers as sides to **construct a triangle**

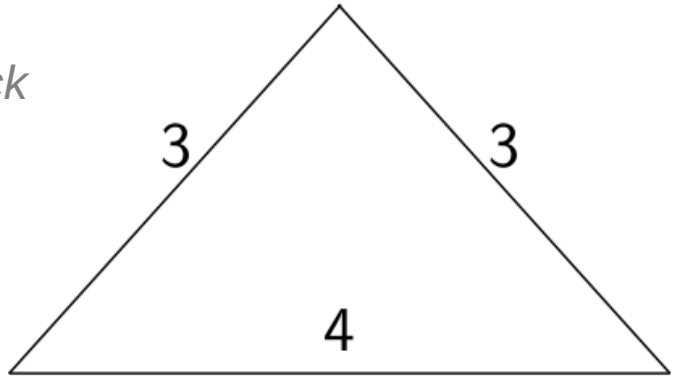
Output:

error – invalid sides, not possible to create a triangle
scalene – no two sides are equal
isosceles – exactly two sides are equal
equilateral – all sides are equal

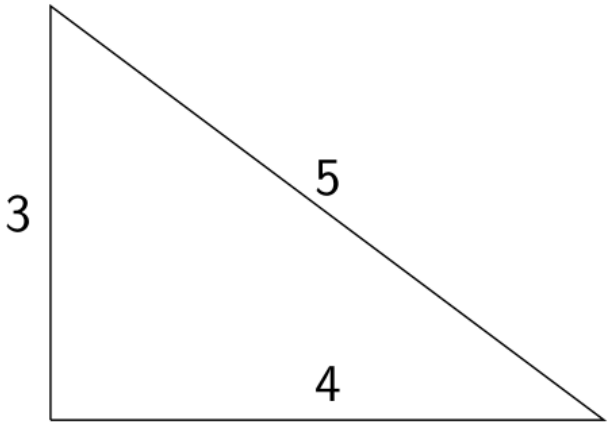
Define a set of test cases
(think of all possible cases)
and write a program.
Do not Google for a solution!

(3, 3, 3)
*en: an **equilateral** triangle*
de: ein gleichseitiges Dreieck





(3, 3, 4)
*en: an **isosceles** triangle*
de: ein gleichschenkliges Dreieck



(3, 4, 5)
*en: an **scalene** triangle*
de: ein Dreieck

Using

DocTest

package

docstrings + doctest

```
def sqfeet_to_sqmeters(a_value):
    """
    Converts square feet to square meters.
    Checks if the argument is a real number.

    :param a_value: a real number of square feet
    :type a_value: float

    :return: a real number of square meters
    :rtype: float

    Examples:
    Correct input type:
    >>> sqfeet_to_sqmeters(107.64)
    10.0
    >>> sqfeet_to_sqmeters(10.764)
    1.0
```

docstrings
– Python documentation strings

provide a convenient way of associating documentation with Python modules, functions, classes, and methods.
An object's docsting is defined by including a string constant as the **first statement** in the object's definition.

doctest
– Testing Through Documentation

Write automated tests as part of the documentation for a module.

The doctest module *tests* source code by **running examples** embedded in the documentation and verifying that they produce the **expected results**.

doctest

```
def sqfeet_to_sqmeters(a_value):
    """
    Converts square feet to square meters.
    Checks if the argument is a real number.

    :param a_value: a real number of square feet
    :type a_value: float

    :return: a real number of square meters
    :rtype: float

    Examples:
    Correct input type:
    >>> sqfeet_to_sqmeters(107.64)
    10.0
    >>> sqfeet_to_sqmeters(10.764)
    1.0
```

parsing the help text
to find examples

strating with '**>>> **'

```
>>> sqfeet_to_sqmeters(107.64)
```

running them

10.0

?

comparing the output text
against the **expected value**

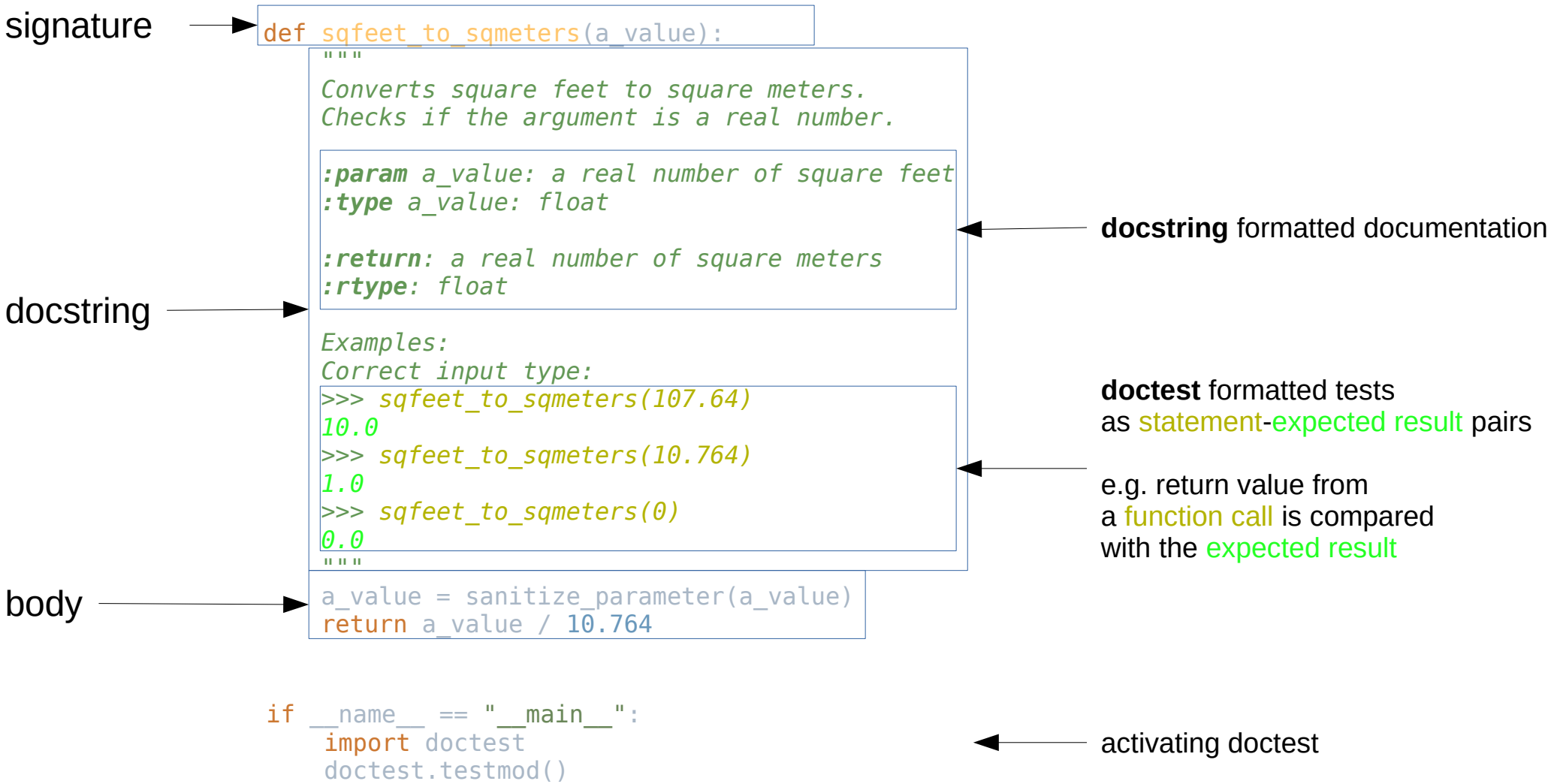
```
10.0
```

ending with a new line
or a next example

syntax like in an interactive
Python interpreter session

```
bpython version 0.17.1 on top of Python 3
>>> import math
>>> math.sqrt(1024)
32.0
>>>
```

Function definition: Structure



DEFENSIVE

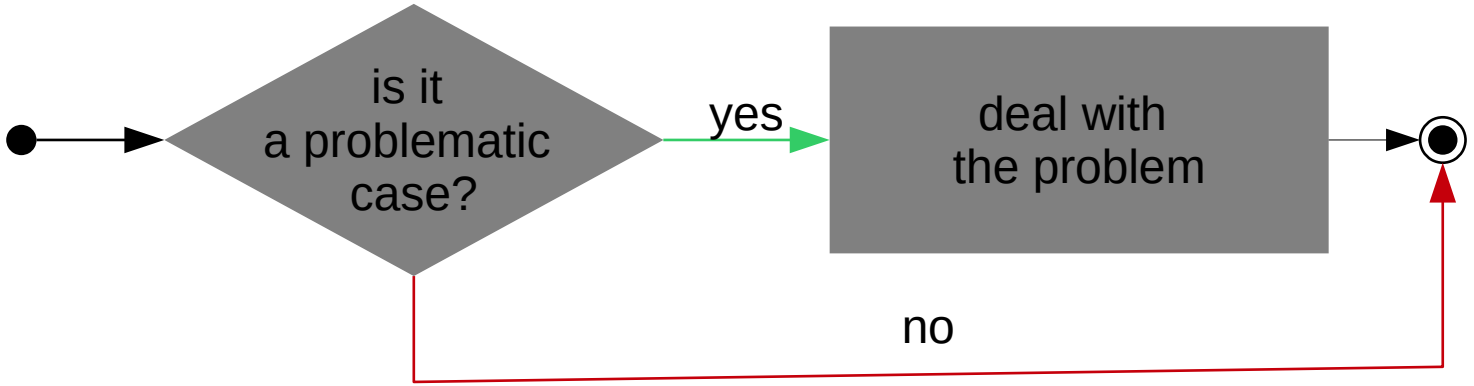
PROGRAMMING

What is defensive programming?

Defensive programming is an approach intended to ensure the continuing function of a program under **unforeseen circumstances**.

One of the goals is to make a program behave in a predictable manner despite unexpected inputs or user actions.

How to deal unforeseen circumstances?



	detect	act
built-in error handling	# Python interpreter	# Python interpreter
user defined exceptions	**if** / **else** **try** / **except**	**raise Exception***
user defined assertions	**assert**	

*) we can defined another behavior for unexpected cases

Where to detect a problem?

- **Precondition** is a condition that must always be true just **prior to the execution** of some section of code.
  In function's precondition you can check all constraints on the function's arguments (types, range).

- **Postcondition** is a condition that must always be true just **after the execution** of some section of code.
  In function's postcondition you can check all constraints on the function's result (e.g. range).

- **Invariant** is a condition that must **always** be true during the execution of a program, or during some portion of it (e.g. inside a loop).

When take care of errors?

- **Sources of data**
  - **Unpredicted sources** of data, e.g. user input or calls from third-party programs, should be checked for correctness
  - **Secure sources** of data, e.g. internal communication within your program, does not need to be checked
- Communication with user
  - if you need messages for **more informative error messages** you should use exception handling or
    if you need to end your program gracefully and safely
  - otherwise you could relay on standard exception handling
- Two approches: outside vs. **inside function**
  - Checking inside have two advantages: the check happens only once
    and it is close to specification and usage