University of
Applied Sciences

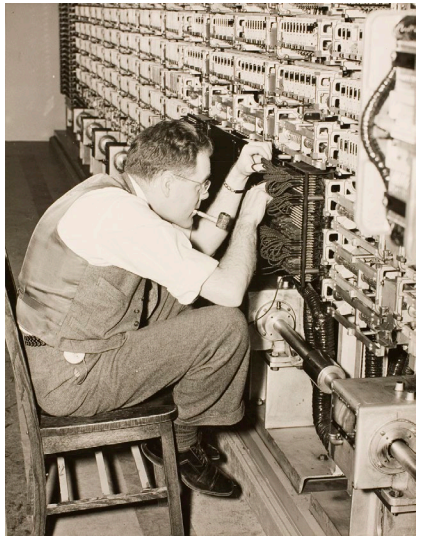
Programming 2

Testing (with Pytest)

Dr. Dhungana
Dr. Klikovits
SS2024 • Krems

Bugs Bugs Bugs

Grace Murray Hopper was the one who recorded the first computer bug attributed to a moth, which caused technical issues with the Harvard Mark II computer. The Mark II was one of the earliest computers and performed faster than its predecessor, the Mark I.



Debugging

IMPACT EDUCATION EVENTS FUNDING OPPORTUNITIES

Sep 9, 1947 CE: World's First Computer Bug

On September 9, 1947, a team of computer scientists reported the world's first computer bug—a moth trapped in their computer at Harvard University.

GRADES
3 - 12+

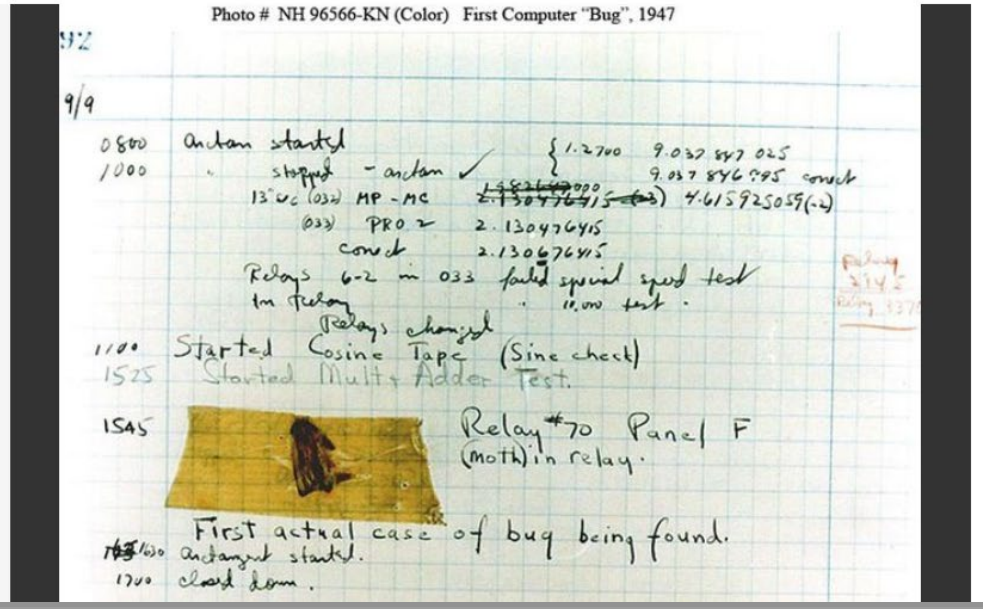
SUBJECTS
English Language Arts, Experiential Learning

CONTENTS
1 Image

Computer Bug

"First actual case of bug being found," according to the brainiacs at Harvard, 1945. The engineers who found the moth were the first to literally "debug" a machine.

PHOTOGRAPH COURTESY NAVAL SURFACE WARFARE CENTER, DAHLGREN, VIRGINIA



Ariane 5

- Rocket catastrophe in 1996
- The flight exploded after 40 seconds.
- The Ariane 5 software reused the specifications from Ariane 4
- Specifically the 64 bit floating point number relating to the horizontal velocity was converted to the 16 bit signed integer.
- The number was larger then 32,767 which was greater then the largest integer store able in the 16 bit signed integer and hence the conversion failed.
- The rocket cost over $7 billion, destroyed rocket and its cargo were valued at $500 million.

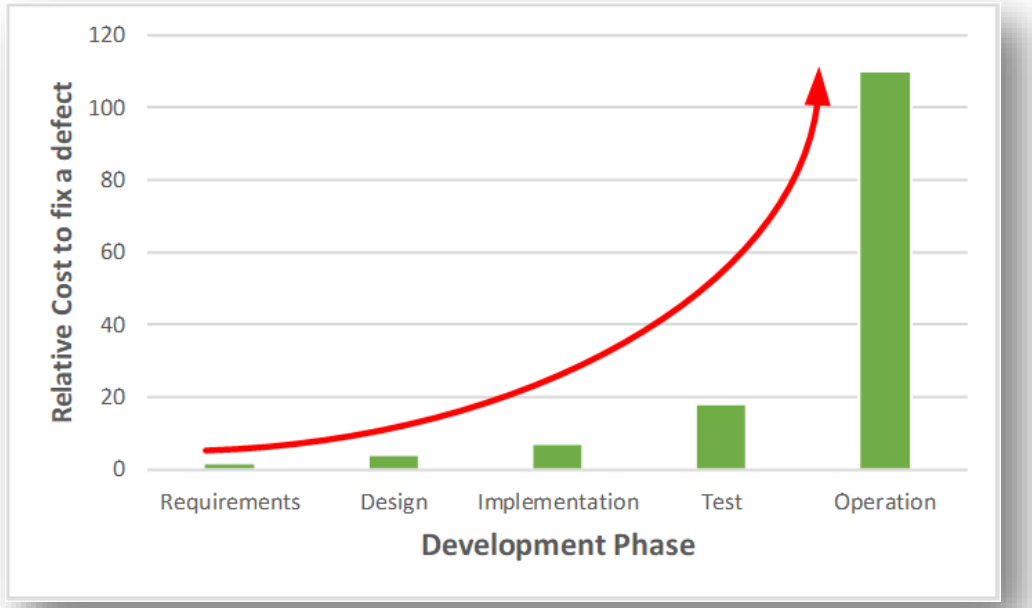
https://de.wikipedia.org/wiki/Ariane_5

Boeing puts cost of 737 Max crashes at $19bn as it slumps to annual loss

Company lost $636m last year after two deadly crashes forced new airliner to be grounded

Objectives of Software Testing

- Uncover as many as errors (or bugs) as possible in a given timeline.
- Demonstrate a given software product matching its requirement specifications.
- Validate the quality of a software testing using the minimum cost and efforts.
- Generate high quality test cases, perform effective tests, and issue correct and helpful problem reports.
- uncover the errors in the software, including errors in:
    - requirements from requirement analysis
    - design documented in design specifications
    - coding (implementation)
    - system resources and system environment
    - hardware problems and their interfaces to software



Verification and Validation

Validation

does the software do what was wanted?

- "Are we **building the right system**?"

This is difficult to determine and involves subjective judgments (reviews, etc.)

Verification

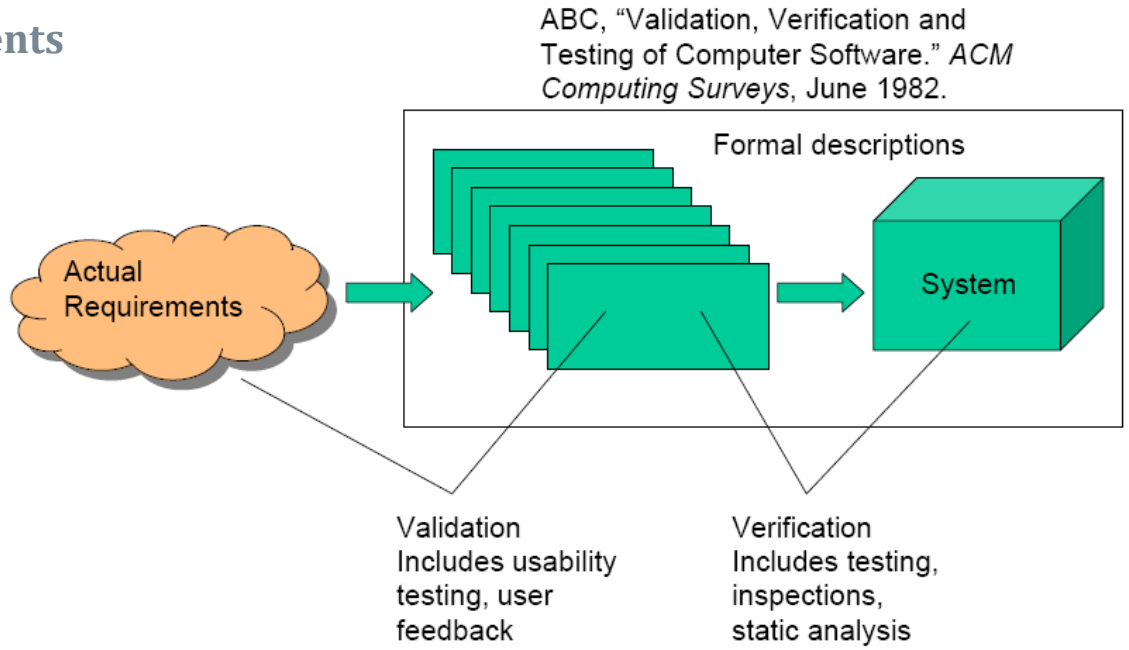
does the software meet its specification?

- "Are we **building the system right**?"

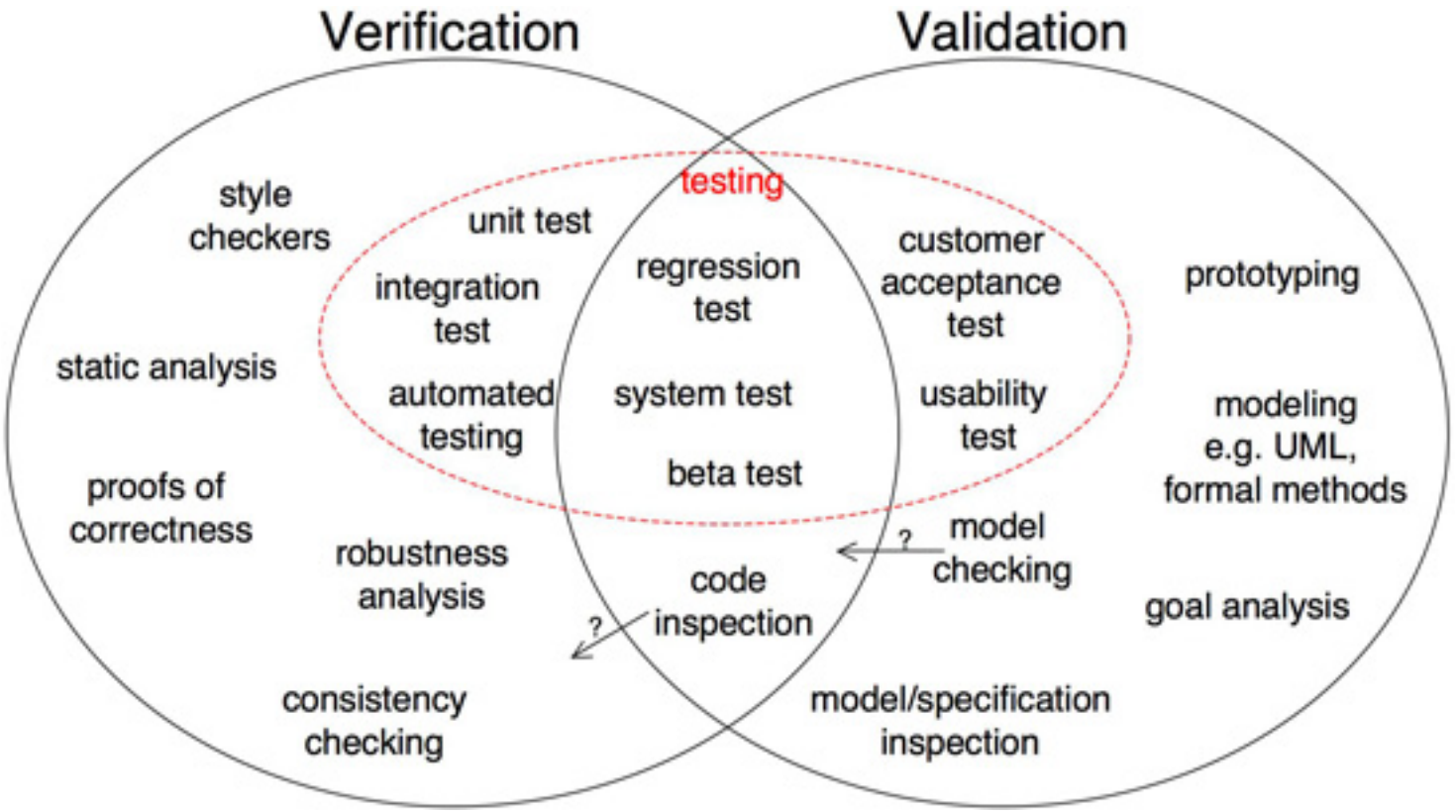
Implications?

Everything must be verified

...including the verification process itself

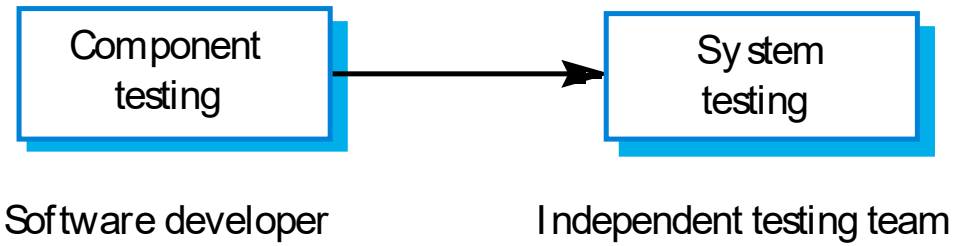


Testing Procedures



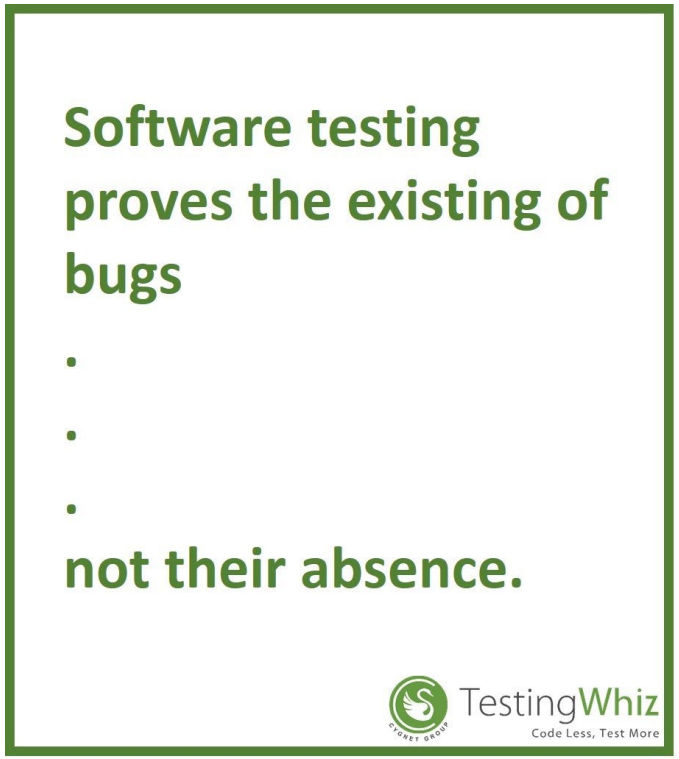
The testing process

- **Component testing**
  - Testing of individual program components;
  - Usually the responsibility of the component developer (except sometimes for critical systems);
  - Tests are derived from the developer's experience.
- **System testing**
  - Testing of groups of components integrated to create a system or sub-system;
  - The responsibility of an independent testing team;
  - Tests are based on a system specification.



Test Characteristics

- **Testing should be repeatable**
  - if you find an error, you want to repeat the test to show others
  - if you correct an error, you want to repeat the test to check that you fixed it
- **Testing should be systematic**
  - random testing is not enough
  - select test sets that are representative of real uses
  - select test sets that cover the range of behaviors of the program
- **Testing should be documented**
  - keep track of what tests were performed, and what the results were

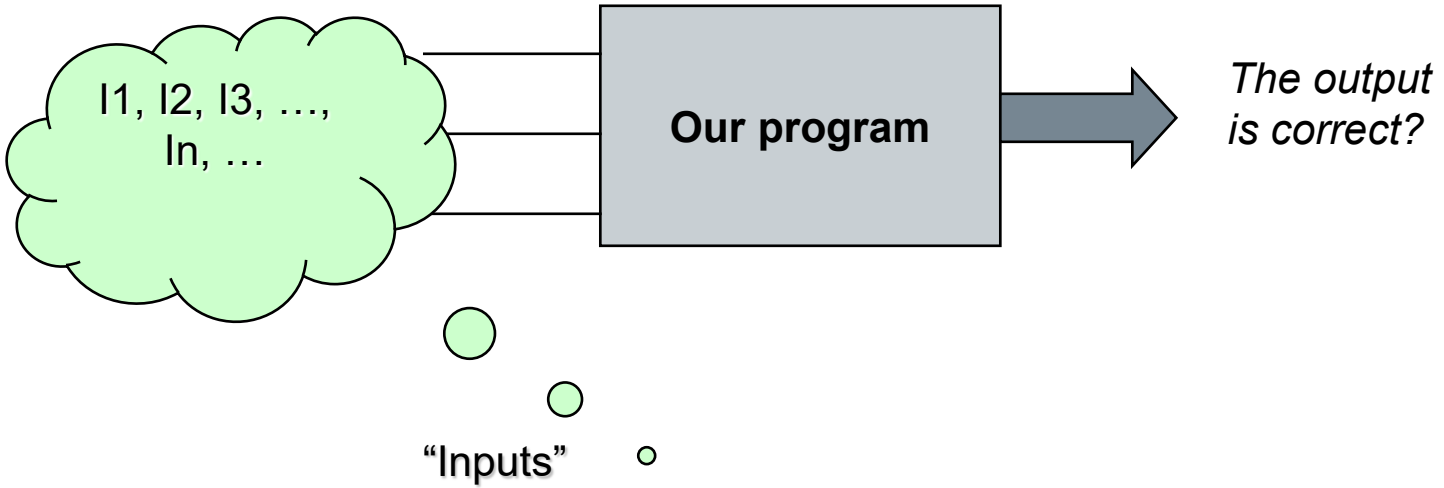


What is a Test Case

Typically consists of

- the input
- the expected output
- the actual result

These test plans/scripts are critical to project success!



Expected results =? Obtained results

Testing Strategies

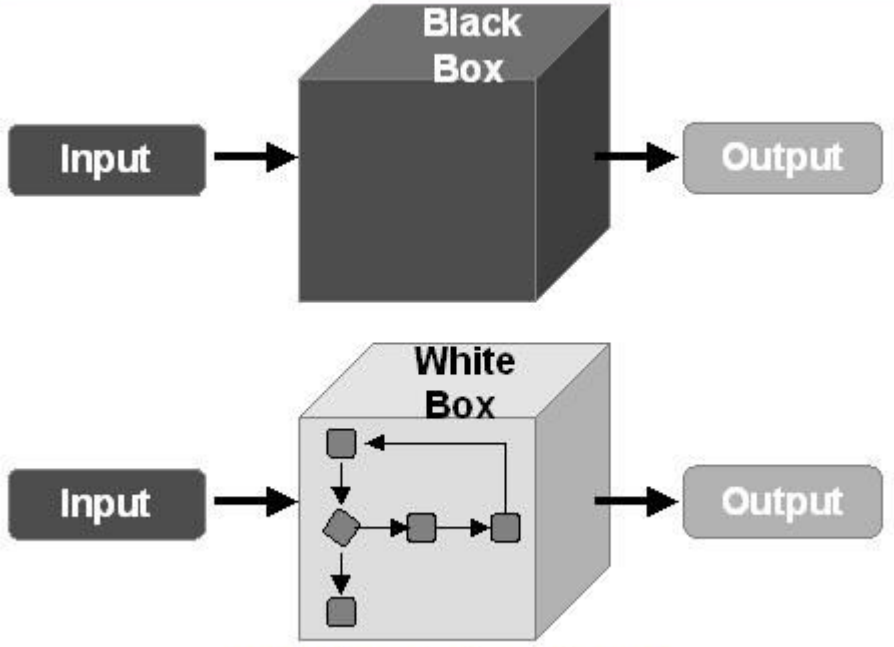
Never possible for designer to anticipate every possible use of system. Systematic testing is therefore essential.

Offline strategies:

- coding convention/style checkers
- walkthroughs (“dry runs”)
- inspections

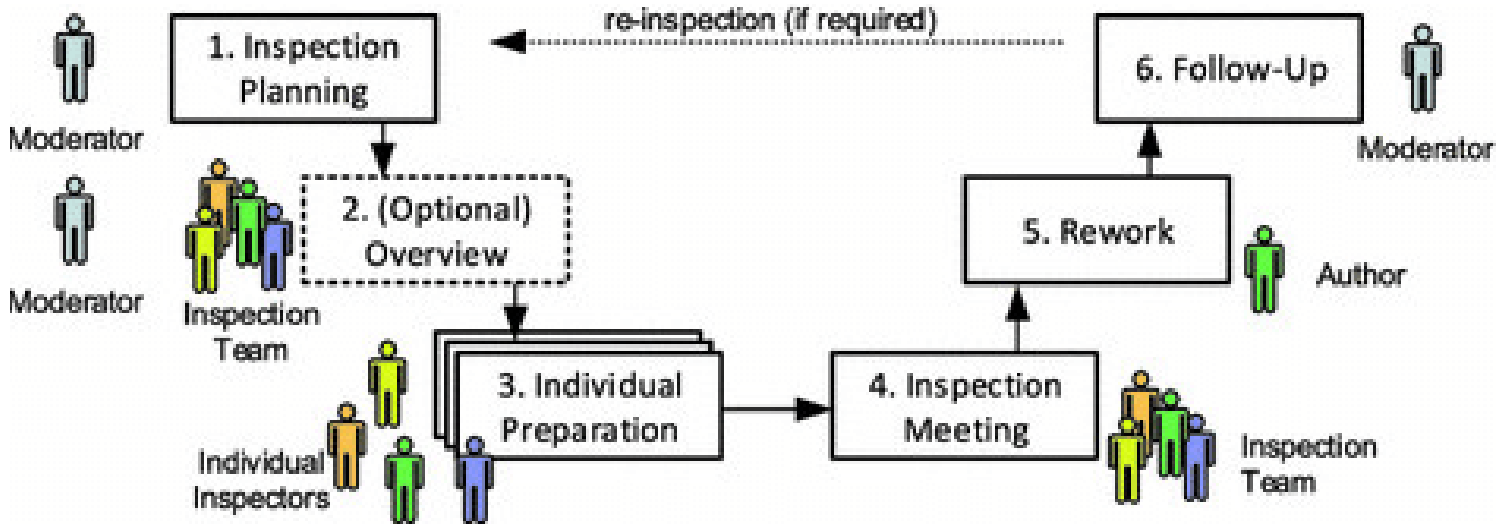
Online strategies

- black box testing
- white box testing



Inspections

- Formal procedure, where a team of programmers read through code, explaining what it does.
- Inspectors play “devils advocate”, trying to find bugs.
- Time consuming process!
- Can be divisive/lead to interpersonal problems.
- Often used only for critical code.



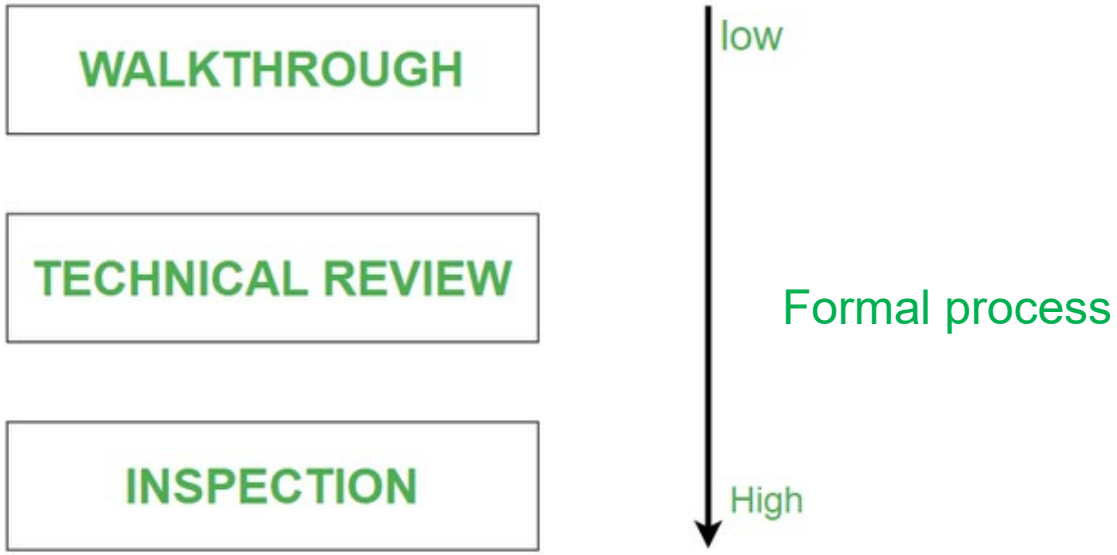
Walkthroughs

Similar to inspections, except that inspectors “mentally execute” the code using simple test data.

Expensive in terms of human resources.

Impossible for many systems.

Usually used as discussion aid.

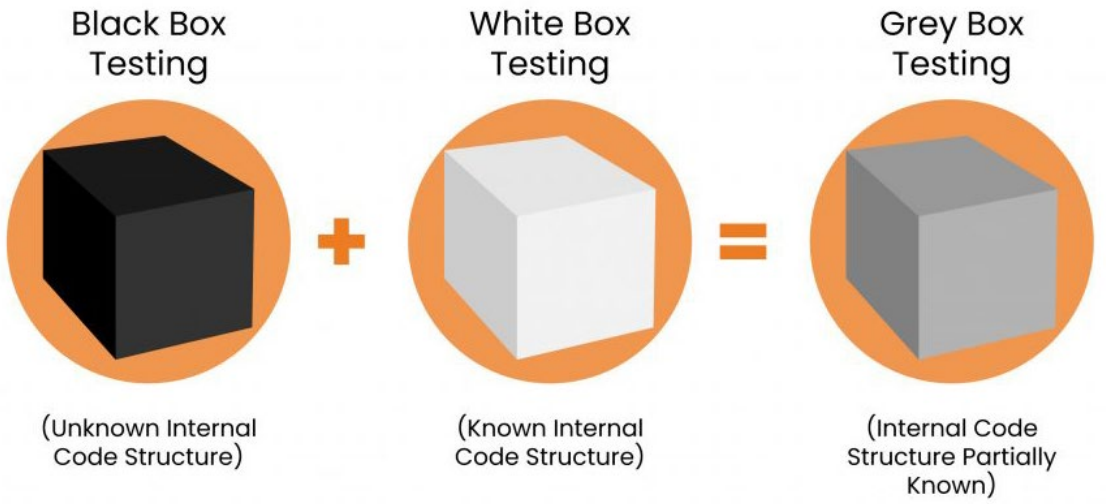


Unit Testing

Individual components are tested independently

- data structure in a component
- program logic and program structure in a component
- component interface
- functions and operations of a component

Unit testers: developers of the components.



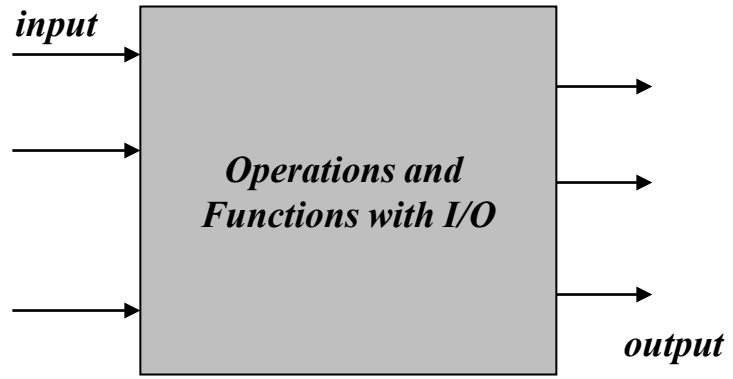
Black Box Testing

Generate test cases from the specification

i.e. don’t look at the code

Advantages:

- avoids making the same assumptions as the programmer
- test data is independent of the implementation
- results can be interpreted without knowing implementation details

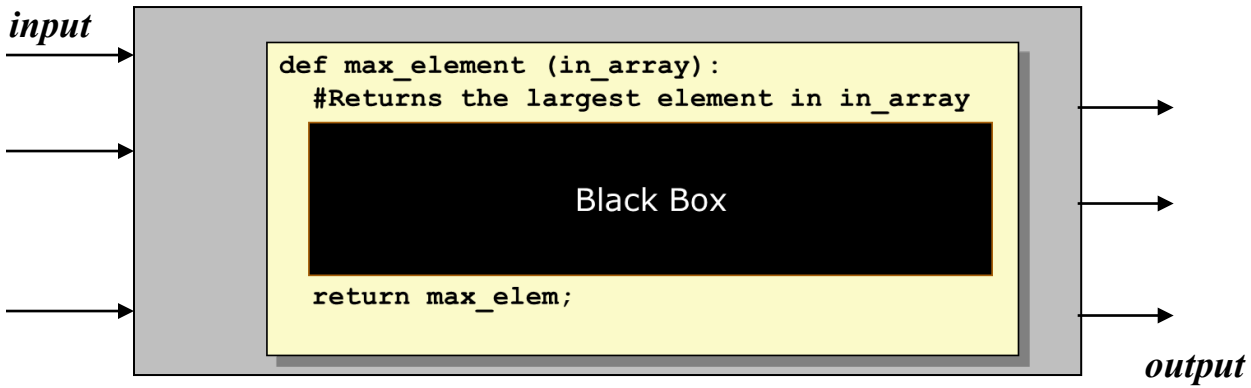


Black Box Testing

Example: Black Box Testing

Is this enough testing?

input	output	OK?
3 16 4 32 9	32	yes
9 32 4 16 3	32	yes
22 32 59 17 88 1	88	yes
1 88 17 59 32 22	88	yes
1 3 5 7 9 1 3 5 7	9	yes
7 5 3 1 9 7 5 3 1	9	yes
9 6 7 11 5	11	yes
5 11 7 6 9	11	yes
561 13 1024 79 86 222 97	1024	yes
97 222 86 79 1024 13 561	1024	yes



Black Box Testing

Equivalence Partitioning

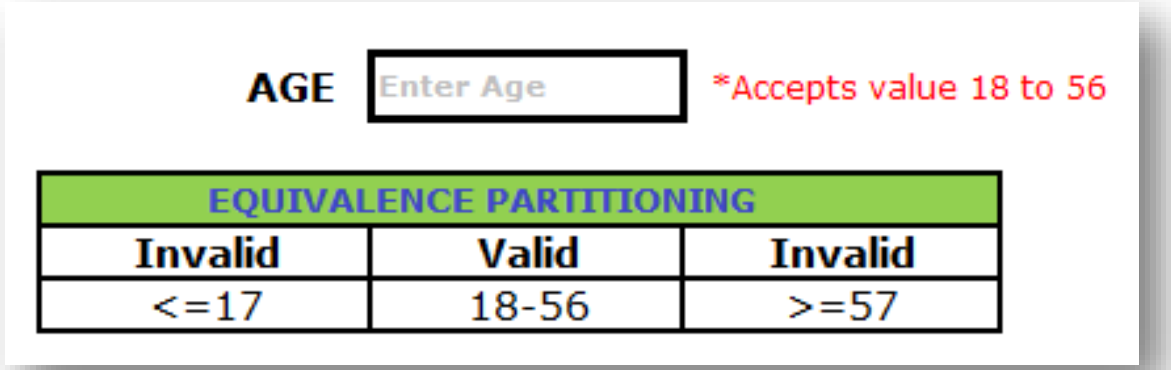
Suppose system asks for “a number between 100 and 999 inclusive”.

This gives three equivalence classes of input:

- less than 100
- 100 to 999
- greater than 999

We thus test the system against characteristic values from each equivalence class.

Example: 50 (invalid), 500 (valid), 1500 (invalid).



Boundary Analysis

Arises from the observation that most programs fail at input boundaries.

Suppose system asks for “a number between 100 and 999 inclusive”.

The boundaries are 100 and 999.

We therefore test for values:

99 100 101

lower boundary

998 999 1000

upper boundary

AGE Enter Age *Accepts value 18 to 56

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
17	18, 19, 55, 56	57

Selecting Test Cases for Black Box Testing

Paths through the specification

- e.g. choose test cases that cover each part of the preconditions and postconditions

Boundary conditions

- choose test cases that are at or close to boundaries for ranges of inputs

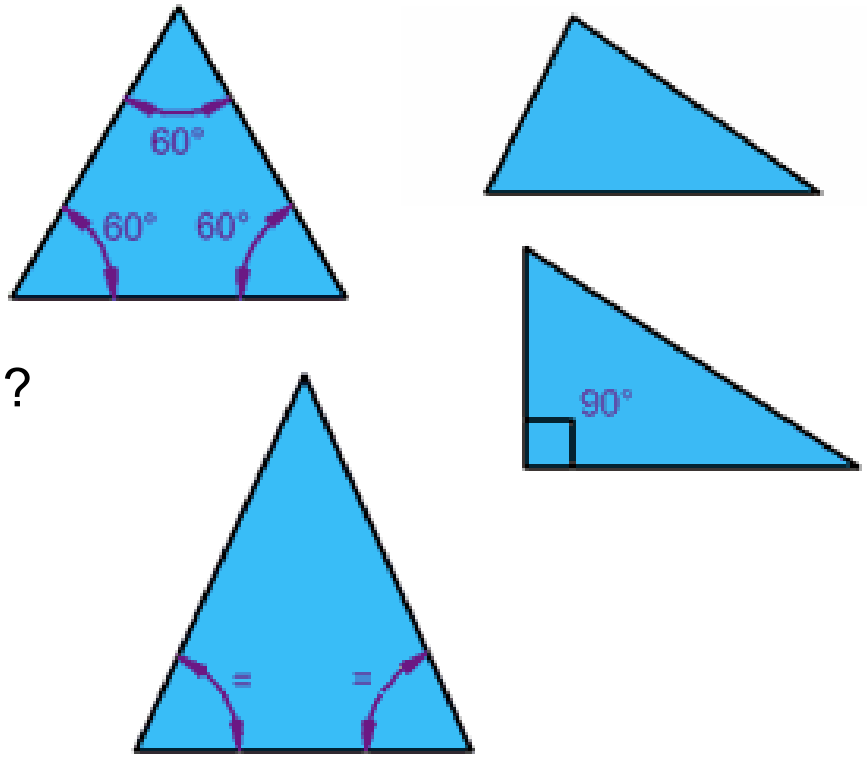
Off-nominal cases

- choose test cases that try out every type of invalid input (the program should degrade gracefully, without loss of data)

Example: Testing a Triangle

Specification: a component gets 3 numerical values (a, b, and c) as input and has to determine, whether the values describe a specific type of triangle.

- Equilateral triangle
- Equal-sided triangle
- Orthogonal triangle
- Other valid triangle.



- Which values would you take to test these specification?
- How many test cases would you use?
- After how many test cases would you stop testing?

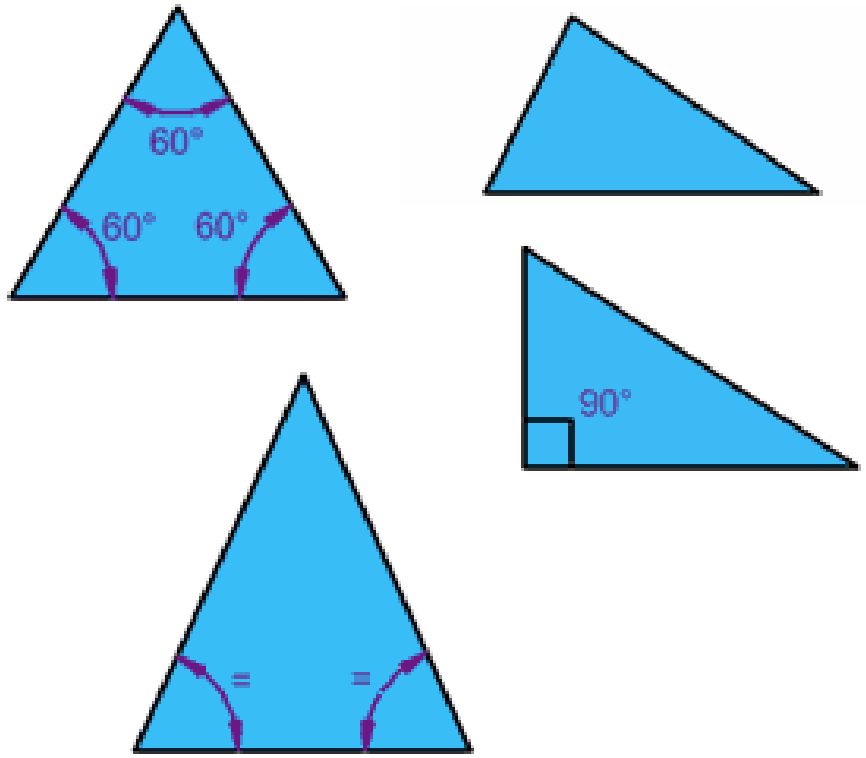
Sample Test cases

Valid Triangles

- Equilateral: 3, 3, 3
- Equal-sided : 5, 5, 3
- Orthogonal: 3, 4, 5
- Others: 3, 5, 7

Invalid Triangles

- a + b < c: 3, 3, 7
- a + b = c: 3, 4, 7
- Negative values: 3, 4, -5
- Zeros: 0, 0, 0



White Box Testing

In white box testing, we use knowledge of the internal structure to guide development of tests.

The ideal: examine every possible run of a system.

- Not possible in practice!

Instead: aim to test every statement at least once

- because black box testing can never guarantee we exercised all the code

Path completeness

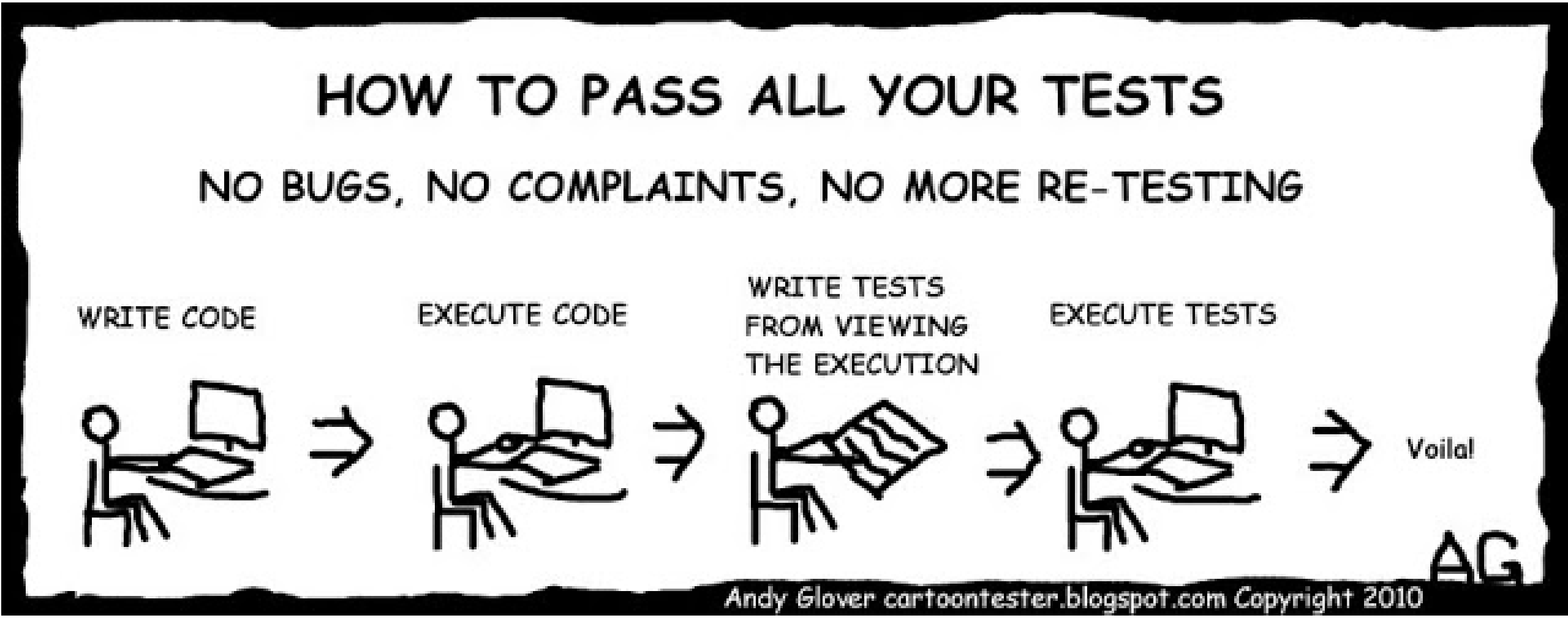
- A test set is path complete if each path through the code is exercised by at least one case in the test set

Example: White Box Testing

What are the test cases to adequately test greetme?

```
def greetme(signal):
    if (signal > 5):
        greet = (“hello”)
    else:
        greet = (‘goodbye’)
    return greet
```

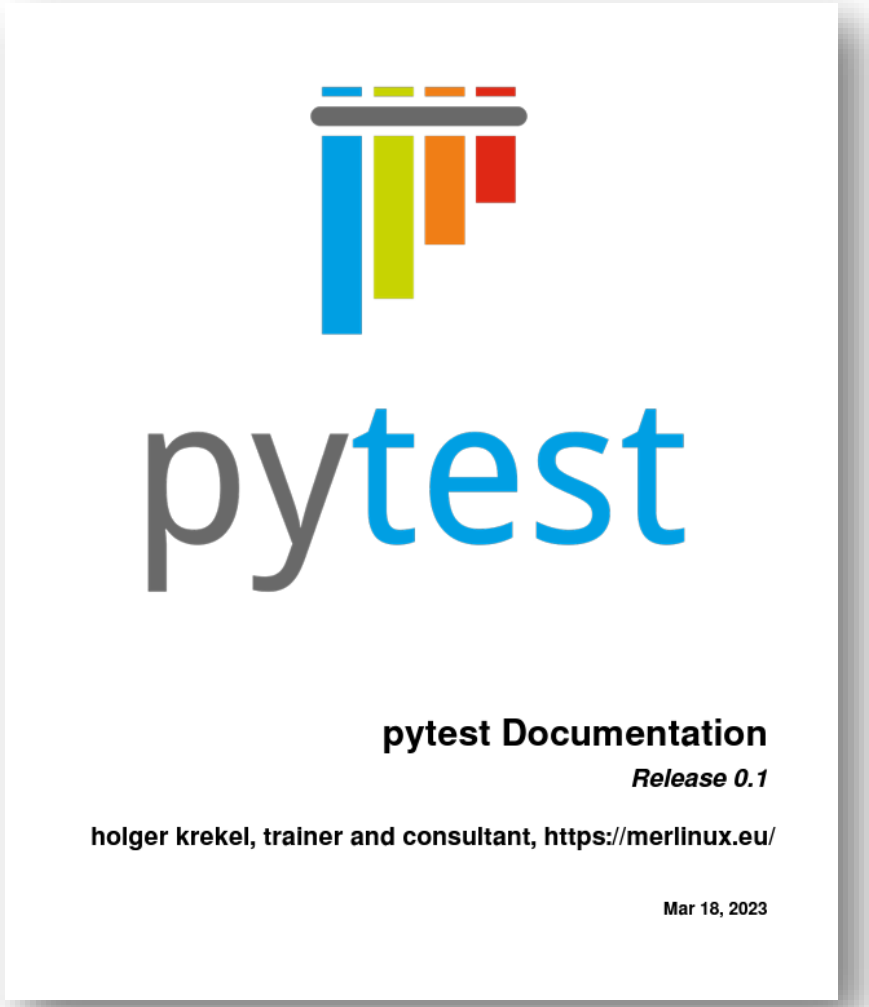
There are two possible paths through this code, corresponding to signal > 5 and signal <= 5. Aim to execute each one.


HOW TO PASS ALL YOUR TESTS
NO BUGS, NO COMPLAINTS, NO MORE RE-TESTING
WRITE CODE
EXECUTE CODE
WRITE TESTS FROM VIEWING THE EXECUTION
EXECUTE TESTS
Voila!
AG
Andy Glover cartoontester.blogspot.com Copyright 2010

Unit Testing with pytest

```
python -m pip install pytest
```

Most functional tests follow the Arrange-Act-Assert model:
1.Arrange, or set up, the conditions for the test
2.Act by calling some function or method
3.Assert that some end condition is true



https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf

A simple Test Case using pytest

The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError → not when using pytest

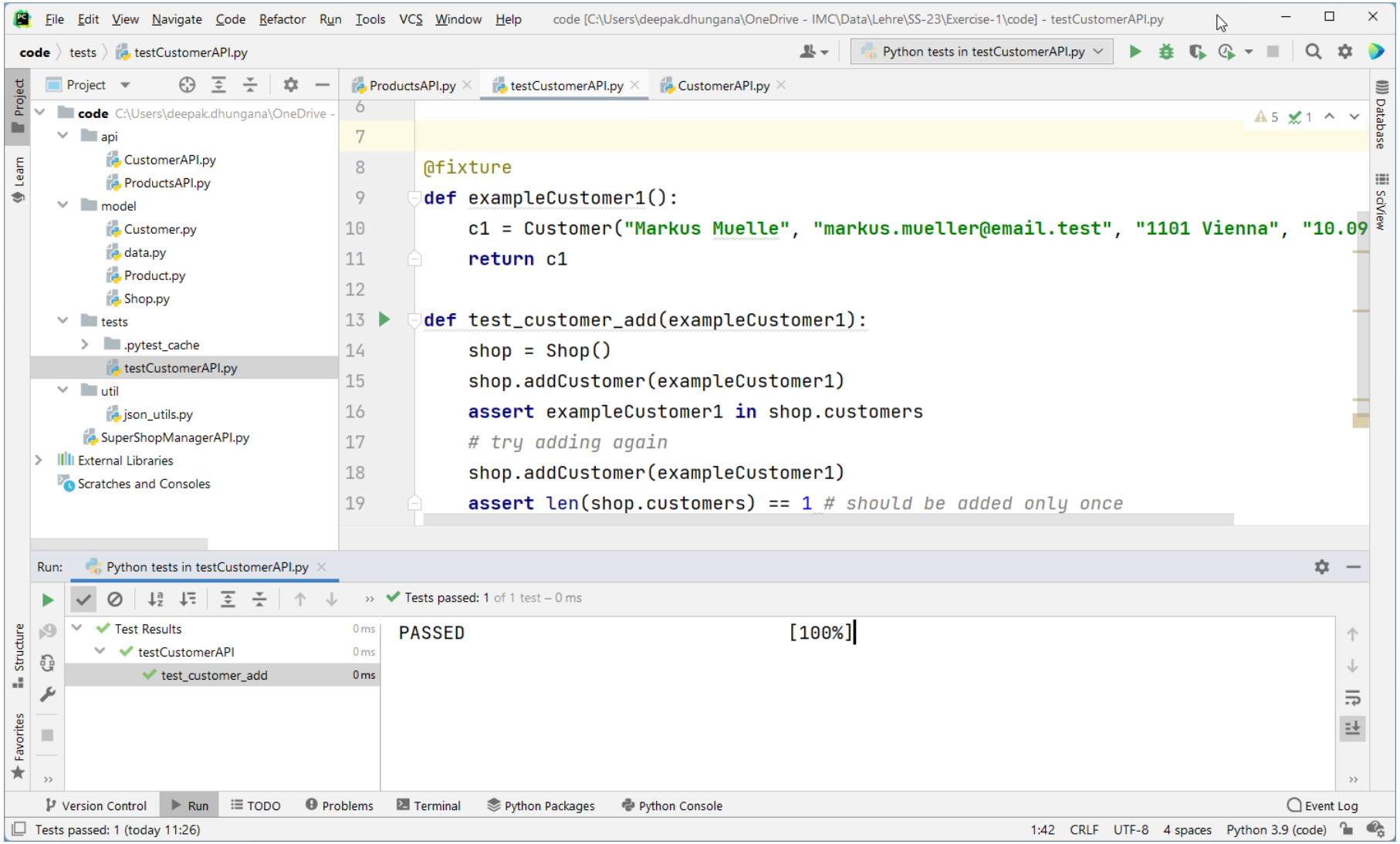
A test case is a function, that starts with “test_”

```
# function to be tested
def capital_case(x):
  return chr(ord(x)-ord('a')+ord('A'))
```

```
def test_capital_case():
  assert capital_case('a') == 'A'
  assert capital_case('x') == 'X'
  assert capital_case('z') == 'Z'
```

```
def test_capital_case2():
  assert capital_case('9') == '9'
  assert capital_case('(') == '('
  assert capital_case('#') == '#'
```

Running Unit Tests in PyCharm



Exercise: Black Box Testing with Pycharm

Write Test Cases to test the implementation of **revWord, which** takes a sentence as input and prints the words in the sentence in the reverse order.

```
# revWord("this is an interesting task")

# Returns "task interesting an is this"


def revWord(s):

        # code here
```

Test Methods in a class

Methods of a class with a prefix **test** are automatically identified as test cases by Pytest

```
class TestClass():
    def test_one(self):
        x = "hello"
        assert 'h' in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, 'check')
```

Test prefixed test classes (without an __init__ method)

https://doc.pytest.org/en/latest/explanation/goodpractices.html#conventions-for-python-test-discovery

Parameterizing Tests

Provide input tuples as parameters of a test!

```
@pytest.mark.parametrize ('inp, outp',
                          [('a', 'A'),
                           ('1', '1'),
                           ('#', '#'),
                           ('-', '-'),
                           ('c', 'C'),
                          ]
                         )
def test_capital_case3(inp, outp):
    assert capital_case(inp) == outp
```

Write Tests cases to test foreign code

Task 4.1 Parameterizing Tests

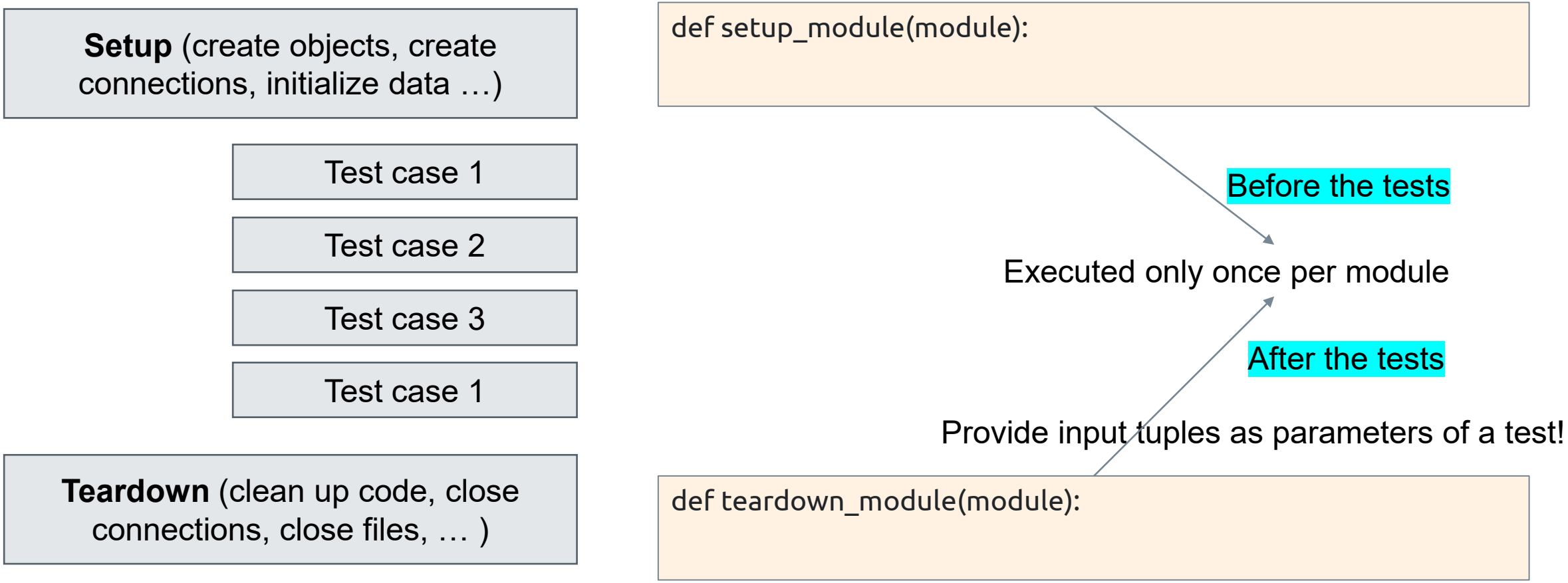
random_password_generator.py is provided in Github

It generates random password with following rules:

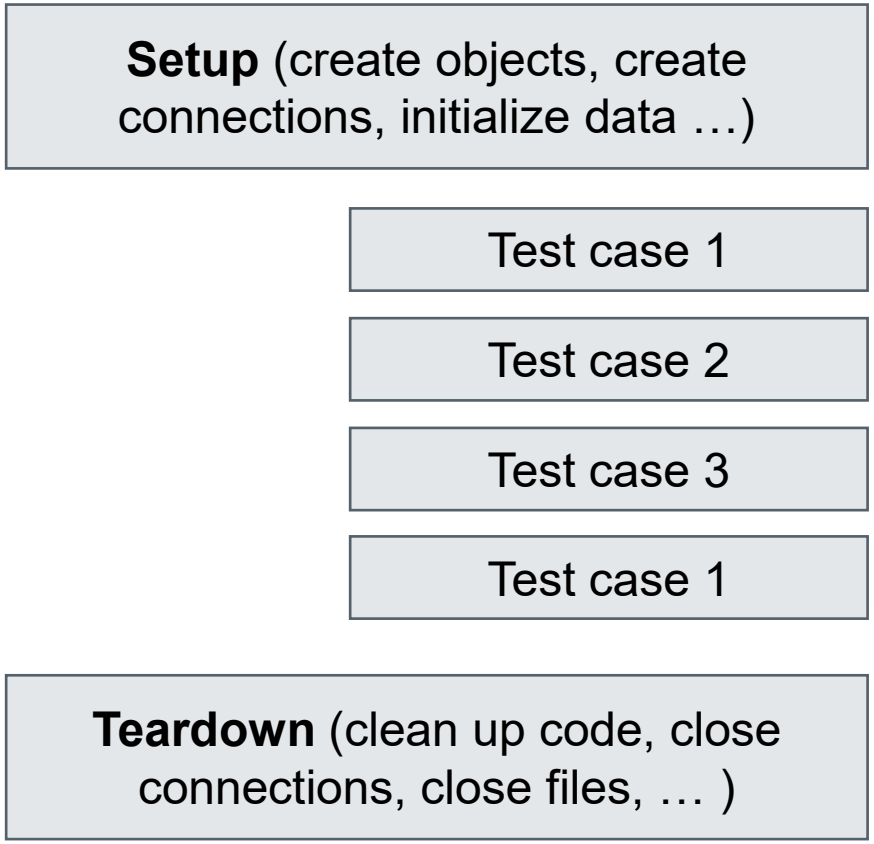
1. 6-20 characters
2. at least one uppercase character
3. at least one lowercase character
4. at least one digit
5. at least one special character (!, @, #, $, %, ^, &, *)
6. no more than 2 characters repeating consecutively

Make appropriate use of `@pytest.mark.parametrize`

Setup and Tear Down Tests per Module



Fixtures



- Software test fixtures initialize test functions. Declared using the decorator
  @pytest.fixture
- They provide a fixed baseline so that tests execute reliably and produce consistent, repeatable, results.
- Initialization may setup services, state, or other operating environments.
- These are accessed by test functions through arguments

Fixtures Example

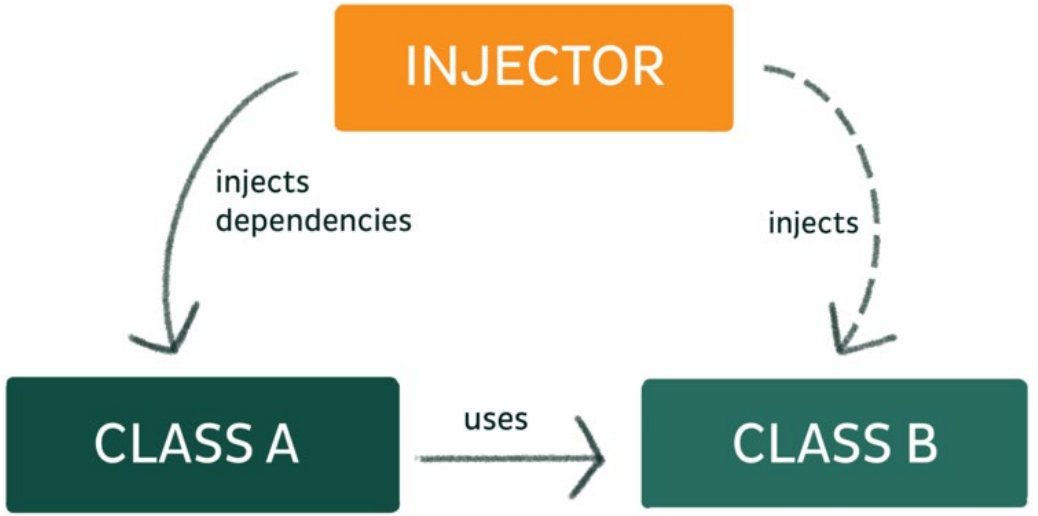
```
@pytest.fixture
def exampleCustomer1():
    c1 = Customer("Markus Muelle", "markus.mueller@email.test", "1101 Vienna", "10.09.2001")
    return c1
```

```
def test_customer_add(exampleCustomer1):
    shop = Shop()
    shop.addCustomer(exampleCustomer1)
    assert exampleCustomer1 in shop.customers
    # try adding again
    shop.addCustomer(exampleCustomer1)
    assert len(shop.customers) == 1 # should be added only once
```

Inversion of Control or dependency injection

Traditional "Pull" approach:
- Direct instantiation
- Asking a factory for an implementation

"Push" approach:
- Something outside of the object "pushes" its dependencies into it.
- The Object has no knowledge of how it gets its dependencies
- It just assumes they are there.



Scope of Fixtures

Sharing a fixture instance across tests in a class, module or session
add a scope="…" parameter to the @pytest.fixture invocation function, class, module, package or session.

```
def test_customer_add (exampleCustomer1):

def test_customer_delete (exampleCustomerList):

def test_customer_verfication (exampleCustomer1):
```

```
@pytest.fixture (scope= "module")
def exampleCustomer1 ():


@pytest.fixture (scope= "session")
def exampleCustomerList(exampleCustomer1):
```

Assertions about expected exceptions

Use context manager pytest.raises

```
import pytest


def test_zero_division():
        with pytest.raises(ZeroDivisionError) as ex:
                print (1 / 0)
  # you can use ex to assert something here
```

Write Tests cases to test foreign code

Task 4.2 Writing Pytest test cases

Consider the programming challenge **https://adventofcode.com/2023/day/1**

Somebody wrote a solution to this challenge here:

https://github.com/stefanoandroni/advent-of-code/blob/master/2023/day-1/part-1/main.py

Write appropriate Pytest test cases to test this implementation.

And here is another implementation for the same challenge. Can you reuse your test cases to test the second implementation?

https://github.com/LiquidFun/adventofcode/blob/main/2023/01/01.py