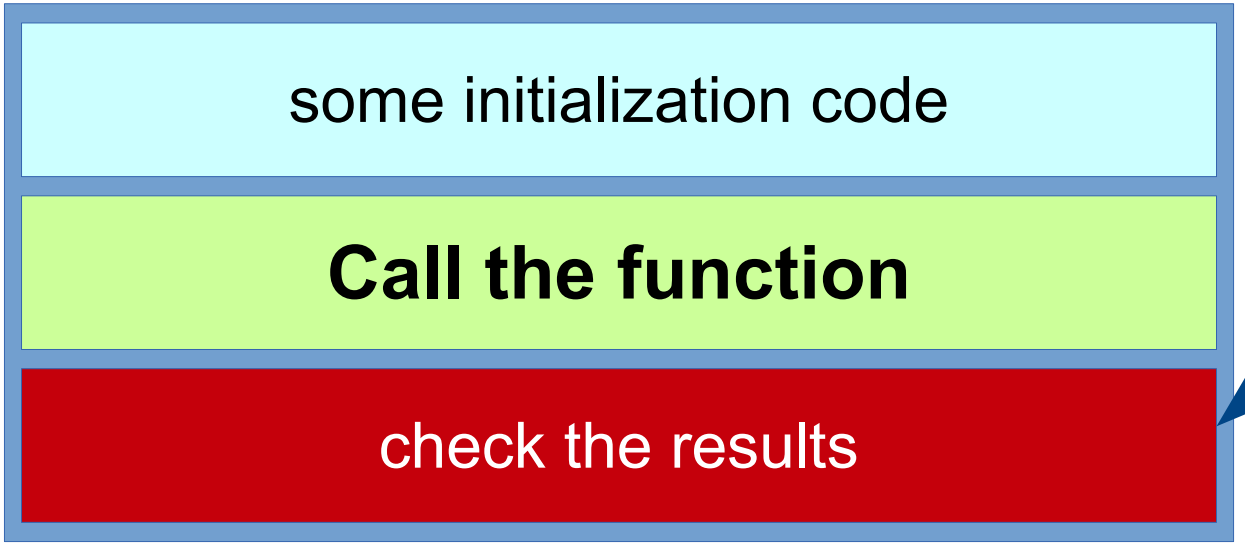
Unit testing in Python with doctest (1)

01219116/01219117
Programming 2

Common structure of a test case

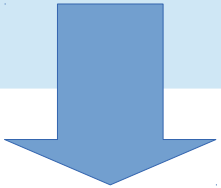
- How do you test a function?
  - You need to call it,
  - and check if it works correctly,
  - by looking at its return value.
- Your code would contain:



The checking code is usually written as a set of **assertions**.

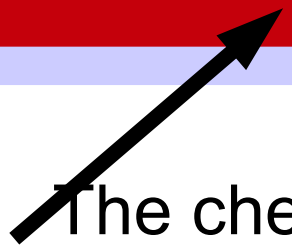
Our test code in Flappy Dot

```
"""
>>> check_player_pillar_collision(100, 100, 300, 200)
False
>>> check_player_pillar_collision(300, 300, 300, 200)
True
"""
```



```
result = check_player_pillar_collision(100, 100, 300, 200)

assert(result, False)
```



The checking code is usually written as a set of **assertions**.

Testing Tools

- Test framework: doctest
  - Write test cases in Python docstring.

The first (finished) example

```
def max3(a, b, c):
    """
    >>> max3(10, 5, 2)
    10
    >>> max3(2, 15, 5)
    15
    >>> max3(10, 7, 20)
    20
    >>> max3(20, 7, 20)
    20
    >>> max3(100, 100, 20)
    100
    >>> max3(100, 200, 200)
    200
    """

    if a >= b and a >= c:
        return a
    if b >= a and b >= c:
        return b
    if c >= a and c >= b:
        return c
```

*spaces between lines are removed so that the code fit in one page.

What do you see?

- A code with corresponding test cases.
- Enough test cases to make you feel confident about the correctness of the code.
  - Ask yourself: hide the code and look at only the test, does it make you feel comfortable to use the code?
- Enough test examples to explain what the function does.

How can we get there?

- Traditional approach
  - Write code, then write test.

- Test-driven development
  - Write test, then write code.

A few words before we start

- TDD is a well-established practice in software development in general.
- But in Game development, TDD (or even unit testing) is not a standard practice.

1st example: max3

- Let's try to work with **max3** to get to the final code as shown previously.

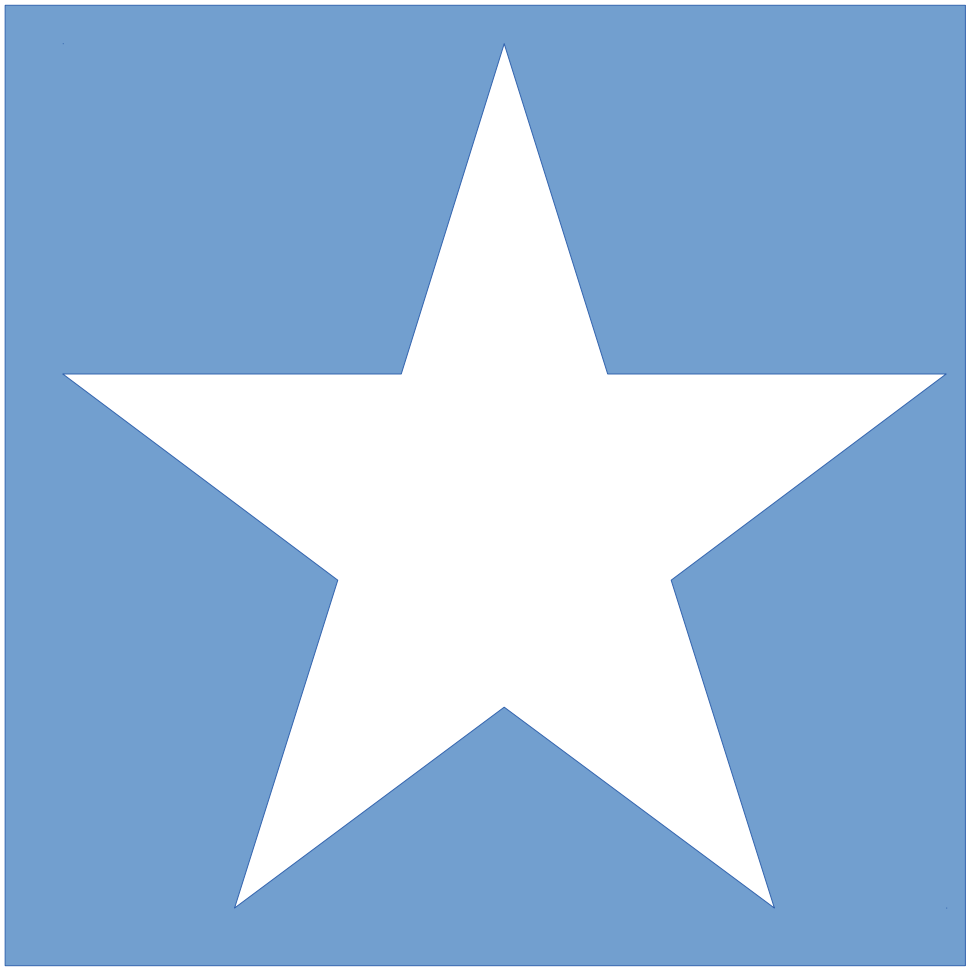
```
def max3(a, b, c):
    # ...
```

- This function returns the maximum of **a**, **b**, and **c**.

How to get started

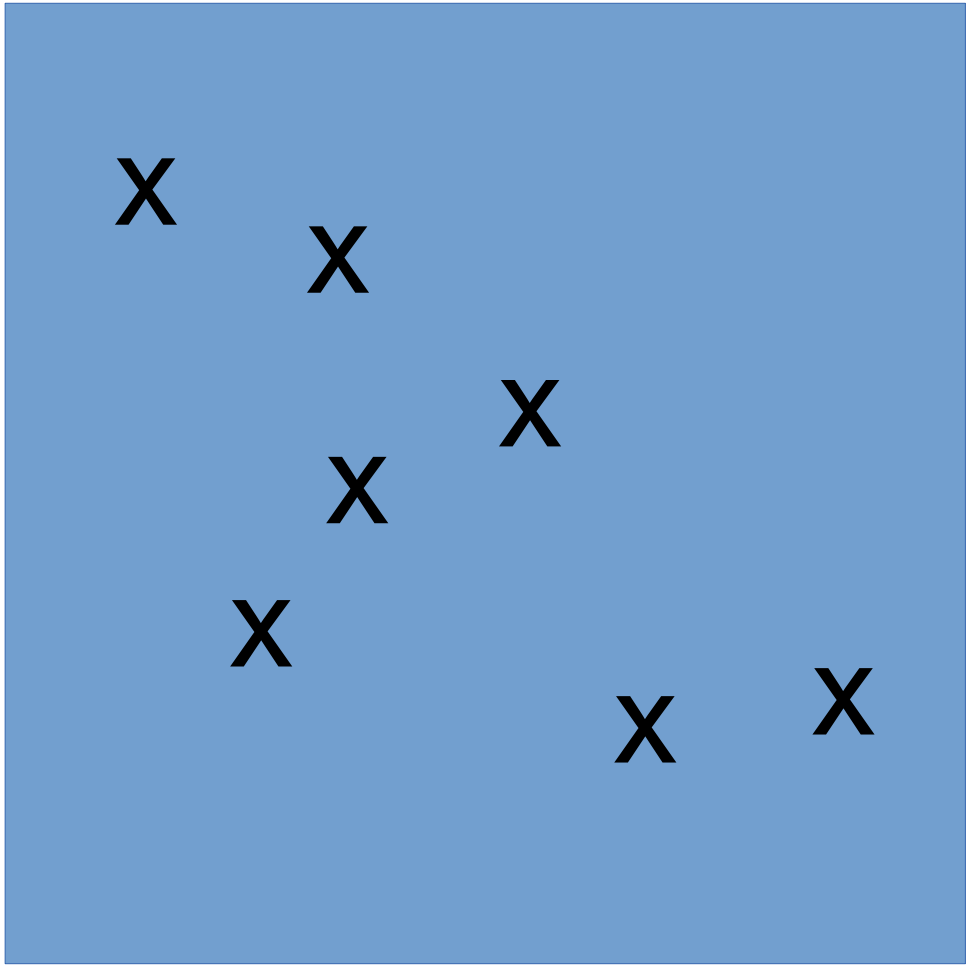
- If you are fluent with the techniques, you can just start writing test cases right away.
- But sometimes it might be easier to start by thinking about what you would like to test.
- In other words, let ask:
  - how do we know that max3 works correctly?

What's in this box?



Is it a star-shaped object?

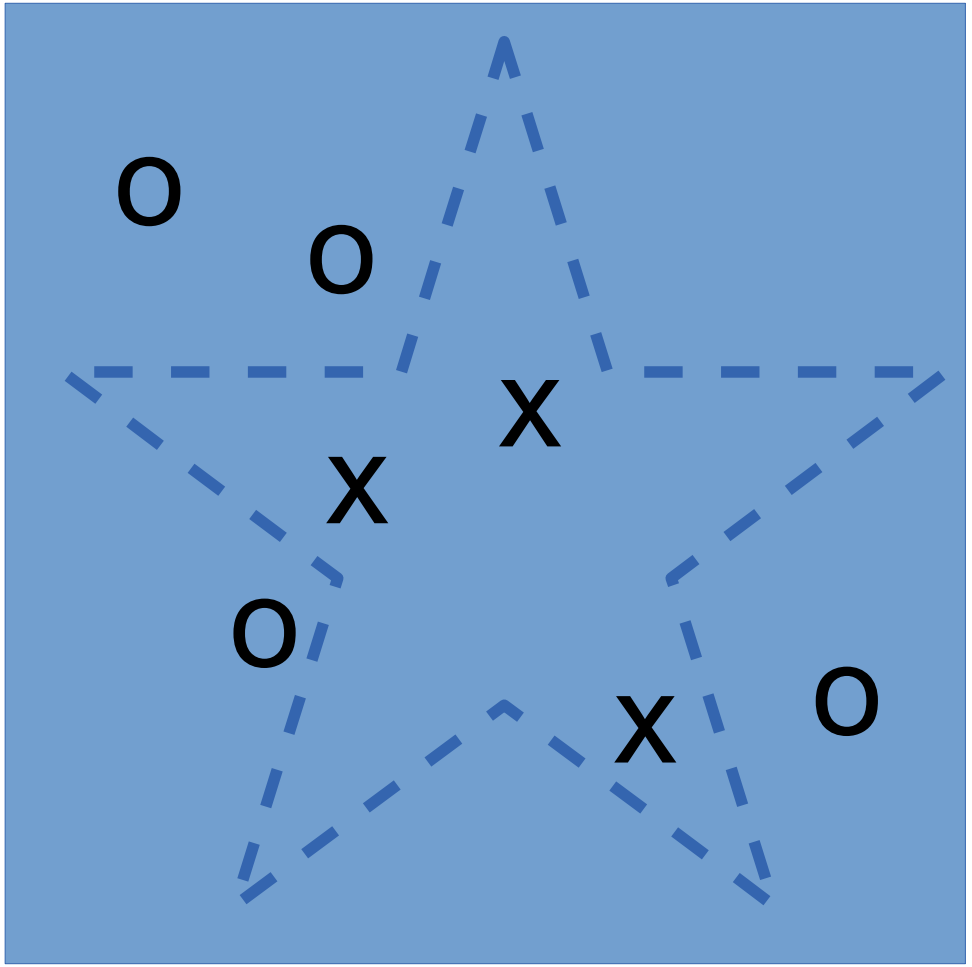
Let's try to “peak” into the box with a pin



These are the positions that we plan to use a pin to check if there is anything at that position

Is it a star-shaped object?

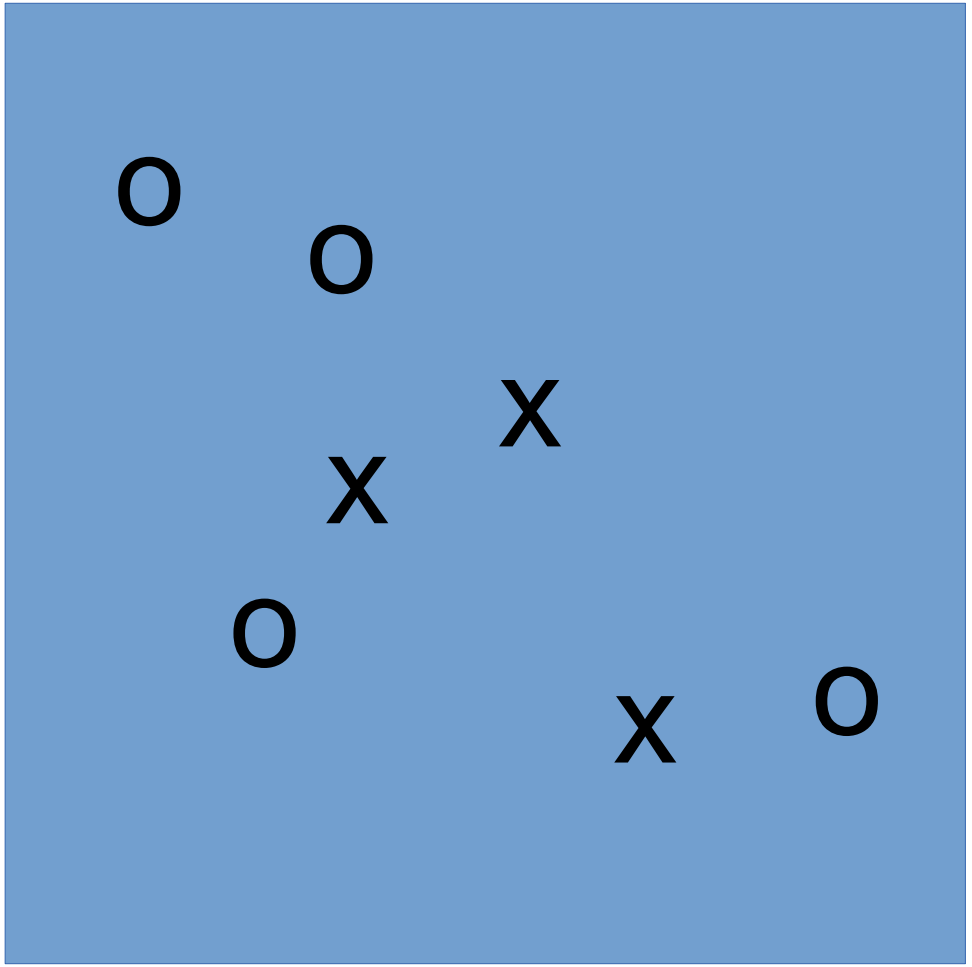
Expectations:
if there is a star in the box



o = nothing
x = something

Is it a star-shaped object?

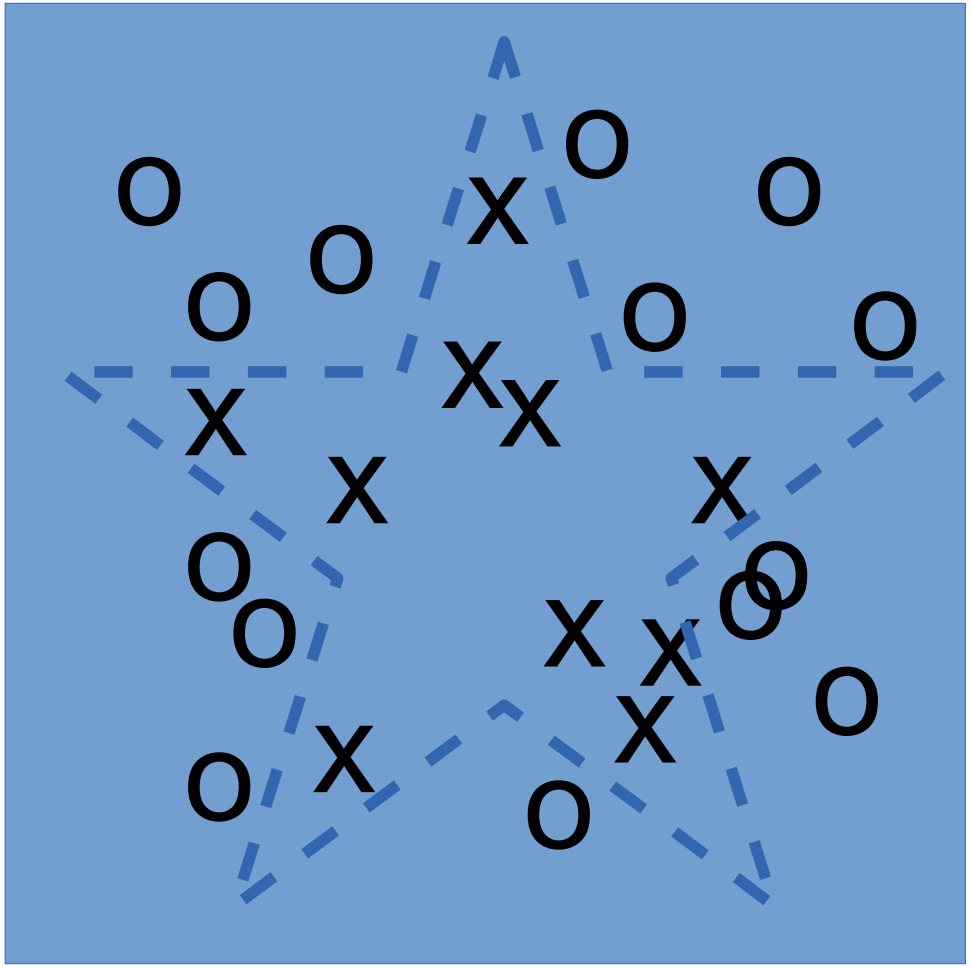
Actual results



o = nothing
x = something

Do you **believe** that it is a star-shaped object?

Actual results with more tests



o = nothing
x = something

Do you believe that it is a star-shaped object?

Usage examples

- Think about the test cases as usage examples for the function.

a	b	c	expected results

Try to be lazy

- Many usage examples look at the same situation.
- We don't need to include all of them.

a	b	c	expected results
10	20	5	20
~~50~~	~~700~~	~~12~~	~~700~~
~~13~~	~~15~~	~~12~~	~~15~~
1	2	3	3
~~9~~	~~30~~	~~40~~	~~40~~
10	10	5	10

Pick one to start

- We need to get started.
- Pick one example, and let's code.
  - Which one? Let's try the one that is easiest to code.

a	b	c	expected results
10	20	5	20
1	2	3	3
10	10	5	10

See the demo

Test structure

Let's try

- Let's start with a simple function:

```
def add_with_cap(a, b, cap):
    # ...
```

- This function adds **a** and **b**, but ensure that the return value is not greater than **cap**. (Think about the HP in game after you drink a magic recovery portion.)

Examples

- Before you start writing the test and code, think about the examples that you would need to show that `add_with_cap` works correctly.
- Think about a table like the one below.
- After you have listed a few test cases, think about which one to start testing first.

a	b	cap	expected results

Practice time

Function `get_top_k`

- Write function get_top_k that takes a list of integers and returns the k-th largest integer.

```
def get_top_k(lst, k):
    # ...
```

- For example:
  - `get_top_k([1, 2, 3, 4], 3)` should return 2
  - `get_top_k([10, 9, 8, 100], 2)` should return 10

Function `fizz_buzz`

- Write a function that takes a number **x** and returns a string according to the following rules.
  - If x is a multiple of 3, return “fizz”.
  - If x is a multiple of 5, return “buzz”
  - If x is both multiple of 3 and multiple of 5, return “fizzbuzz”
  - otherwise return the number (in string), e.g., fizz_buzz(7) should return “7”

Function pronounce

- Write function **`pronounce`** that takes an integer **`x`** from 1 to 999 and return how **`x`** is pronounced in English.

```
def pronounce(x):
    # ...
```

- For example:
  - pronounce(1) should return 'one'
  - pronounce(57) should return 'fifty-seven'