Test-Driven Development (TDD) Example

History of Unit Testing

Unit testing is not a new concept, but it has gained considerable significance in the past two decades, largely due to the evolution of extreme programming methodologies.
In 1975, it was referenced in Frederick Brooks' "The Mythical Man-Month."
In 1979, Glenford Myers' book "The Art of Software Testing" delved into it in detail.
In 1987, IEEE introduced a specific standard for software unit testing.
In 1999, Kent Beck's book "Extreme Programming Explained" laid out the fundamental principles of TDD - a methodology that prioritizes testing.

Development Cycle

TDD is built around the "red -> green -> refactor" cycle. In the first phase, the programmer writes a failing test (red), in the second phase, they write the minimal code needed for the test to pass (green), and in the third phase, they clean up the code through refactoring, if necessary. The order of these phases is crucial.

TDD Example

In this example, no testing frameworks are used; unit tests are performed "manually". This approach simplifies the understanding of unit tests for first-time learners. More sophisticated methods of crafting tests will be presented in subsequent materials.

Create Tests and Code Stubs

1.1. Write Unit Tests

Let's say our task is to implement a list sort function. Instead of diving right into the code, we start by writing unit tests. This allows us to immediately outline the function's specification, capturing it in a set of tests.

```
def test_sort():
    print("Test #1")
    print("testcase #1: ", end="")
    A = [4, 2, 5, 1, 3]
    A_sorted = [1, 2, 3, 4, 5]
```

```
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")
    print("testcase #2: ", end="")
    A = []
    A_sorted = []
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")
    print("testcase #3: ", end="")
    A = [1, 2, 3, 4, 5]
    A_sorted = [1, 2, 3, 4, 5]
    sort_algorithm(A)
    passed = A == A_sorted
    print("Ok" if passed else "Fail")
test_sort()
```

In the sort_algorithm() interface we've used, we've assumed that the list sorts in-place. Accordingly, an empty list stays empty, and a list sorted in ascending order remains sorted.

1.2. Write a Stub

Now we create a stub function to be tested, necessary to run the code. As soon as the stub is written, we need to run the tests, and they should not pass yet.

```
def sort_algorithm(A):
    pass
test_sort()
```

Run the tests:

```
Test #1
testcase #1: Fail
testcase #2: Ok
testcase #3: Ok
```

Regrettably, the test design is flawed. The stub, which does nothing, has passed two-thirds of the test scenarios. This might give the false impression that the sorting function is almost functional, which is far from the truth. Therefore, we should issue a final Fail, even if Fail occurs only once. Moreover, we need to create more functional action tests, rather than inaction tests. At the very least, we could add a reversed list case, as well as a case where the list contains repeating numbers.

Hmm... By the way, shouldn't we check if the sorting is stable (i.e., does not switch identical values)? And should we test the sorting of other objects - floating point numbers, strings, tuples, not just integers?

Now, what about the acceptable list length? What length list should our function be able to sort in a reasonable time frame? After all, there are sorting algorithms with vastly different time complexities...

These questions, arising at the first stage of TDD-based development, represent the most valuable consequence of using this methodology! TDD literally forces us to focus on the function's interface specifications. At this point, we must pause further development and clarify requirements: look into the project documentation, consult with the supervisor/customer/team leader, or if we're self-supervising, make clear, reasoned decisions and capture them in the tests. Moreover, it would be beneficial to name the tests in a way that makes it clear what exactly they are testing.

Let's say we consulted with the team leader and got answers to all our questions:

1. Should the sorting be stable? - Yes.
2. Should the sorting be universal? - Yes.
3. Maximum length of the list to be sorted? - 100 elements.
4. What time complexity is required? - Quadratic, O(N^2).
   (For now, let's not discuss that Python has a standard universal pragmatic sort with a time complexity of O(N*logN). Our goal, using Bubble sort as a known "reinvented wheel", is to illustrate the TDD development process.)

1.3. Refining Our Tests

The key aim here is to revise our tests in such a way that running the stub will return a negative result. Most importantly, we want to ensure all our assertions about the sort are documented in the tests.

In developing our tests, we'll adhere to structured programming and a top-down approach. We'll break the tests into separate functions with understandable names (matching the specifications we defined earlier) and call them from our main function.

```
from random import shuffle  # it randomizes order of elements
```

```
def test_sort():
    print("Test sorting algorithm:")
    passed = True
    passed &= test_sort_works_in_simple_cases()
    passed &= test_sort_algorithm_stable()
    passed &= test_sort_algorithm_is_universal()
    passed &= test_sort_algorithm_scalability()
    print("Summary:", "Ok" if passed else "Fail")
def test_sort_works_in_simple_cases():
    print("- sort algorithm works in simple cases:", end=" ")
    passed = True
    for A1 in ([1], [], [1, 2], [1, 2, 3, 4, 5],
               [4, 2, 5, 1, 3], [5, 4, 4, 5, 5],
               list(range(20)), list(range(20, 1, -1))):
        A2 = sorted(list(A1))  # yes, we are cheating here to shorten example
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))
    print("Ok" if passed else "Fail")
    return passed
def test_sort_algorithm_stable():
    print("- sort algorithm is stable:", end=" ")
    passed = True
    for A1 in ([[100] for i in range(5)],
               [[1, 2], [1, 2], [2, 2], [2, 2], [2, 3], [2, 3]],
               [[5, 2] for i in range(30)] + [[10, 5] for i in range(30)]):
        shuffle(A1)
        A2 = sorted(list(A1))  # here we are cheating: standard sort is stable
        sort_algorithm(A1)
        # to test stability we will check A1[i] not equals A2[i], but is A2[i]
        passed &= all(x is y for x, y in zip(A1, A2))
    print("Ok" if passed else "Fail")
    return passed
def test_sort_algorithm_is_universal():
    print("- sort algorithm is universal:", end=" ")
    passed = True
    # testing types: str, float, list
    for A1 in (list('abcdefg'),
               [float(i)**0.5 for i in range(10)],
               [[1, 2], [2, 3], [3, 4], [3, 4, 5], [6, 7]]):
        shuffle(A1)
        A2 = sorted(list(A1))
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))
    print("Ok" if passed else "Fail")
    return passed
def test_sort_algorithm_scalability(max_scale=100):
    print("- sort algorithm on scale={0}:".format(max_scale), end=" ")
```

```
    passed = True
    for A1 in (list(range(max_scale)),
               list(range(max_scale//2, max_scale)) +
list(range(max_scale//2)),list(range(max_scale, 0, -1))):
        shuffle(A1)
        A2 = sorted(list(A1))
        sort_algorithm(A1)
        passed &= all(x == y for x, y in zip(A1, A2))
    print("Ok" if passed else "Fail")
    return passed
def sort_algorithm(A):
    "Sorting of list A on place."
    pass
test_sort()
```

With these modifications, our testing is detailed enough to specify the task of this particular sort. Reading the main function, test_sort(), allows us to quickly grasp the properties of the algorithm, and studying any specific test function provides us with a detailed understanding of a certain property.

It's important to note that this 'documentation' serves as the active criteria that will only approve our code when it meets the standards.

1.4. Ensuring That the Stub Does Not Pass the Updated Tests

Execute the code provided above and you should receive a testing summary:

```
Test sorting algorithm:
- sort algorithm works in simple cases: Fail
- sort algorithm is stable: Fail
- sort algorithm is universal: Fail
- sort algorithm on scale=100: Fail
Summary: Fail
```

We've achieved victory! The 'red color' has been reached! Unit tests are written, and we can now move on to implementation.

It's worth mentioning that using the unittest library or the doctest library significantly simplifies the formatting and shortens the length of the behavioral requirements for the function, which are organized into unit tests. The capabilities of these two libraries will be demonstrated in the next materials of the course.

2. Implementing the Required Functionality

We need to act swiftly and decisively here, as our specification of requirements is at our fingertips, and the task is extremely clear and specific.

In accordance with the "Test First" principle, we should only write the code that is absolutely necessary for the tests to run successfully. We can even 'insert a crutch' that does not cause the tests to fail.

For the sake of simplicity, let's first make a small error in bubble sort:

```
def sort_algorithm(A):
    """
    Sorting of list on place. Using Bubble Sort algorithm.
    """
    N = len(A)
    for i in range(N-1):
        for k in range(N-1):
            if A[k] >= A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]
```

The testing result shows that this version is unstable:

```
Test sorting algorithm:
- sort algorithm works in simple cases: Ok
- sort algorithm is stable: Fail
- sort algorithm is universal: Ok
- sort algorithm on scale=100: Ok
Summary: Fail
```

Identify the error (typo) and strive to achieve 'green color' on your own. Do not completely overhaul the algorithm, as a radical improvement is what we aim to do once we've reached the 'green color'.

3. Refactoring

When a typical programmer begins refactoring, they often feel nervous and extremely tense. This is usually because refactoring tends to occur when the code has become so convoluted that it has stopped working, and pinpointing exactly where can be a big question... Such "red light" refactoring can not only fail to solve problems, but it can also break the program, potentially leading to the collapse of the project akin to the "Tower of Babel" scenario. At any point during refactoring, the programmer may inadvertently break something while rewriting the code and not even notice it. It's no wonder that the word "refactoring" can be like a red rag to a bull for many IT managers: as a leader, they often don't understand the purpose of this "fussing over already written code", and they're also fearful of regression.

The solution is clear — refactoring should only begin when there are unit tests in place, and only when the "green light" is on. This ensures that you're always prepared to roll back to a working version of the code if something goes wrong.

We're currently on "green light", so we confidently and boldly rewrite the code, occasionally running our tests, until we're satisfied that our preferred version of the sort (which is visually appealing to us as it's understandable, clean, and practical) passes all tests and receives the final OK.

Let's settle on this version:

```
def sort_algorithm(A):
    """
    Sorting of list on place. Using Bubble Sort algorithm.
    """
    N = len(A)
    list_is_sorted = False
    bypass = 1
    while not list_is_sorted:
        list_is_sorted = True
        for k in range(N - bypass):
```

```
            if A[k] > A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]
                list_is_sorted = False
bypass += 1
```

Running the tests:

```
 Test sorting algorithm:
- sort algorithm works in simple cases: Ok
- sort algorithm is stable: Ok
- sort algorithm is universal: Ok
- sort algorithm on scale=100: Ok
Summary: Ok
```

Conclusion

The result of applying TDD is the happiness of the programmer, which specifically means:

1. Confidence in one's own code.
2. Absence of fear and peace of mind during refactoring.
3. Having a relevant and automatically verifiable behavior specification for the function.

However, it's important not to turn TDD into a dogma, assuming that code written according to other methodologies is inherently bad. Even the most wonderful idea can be taken to absurd extremes.