Test-Driven Development (TDD)

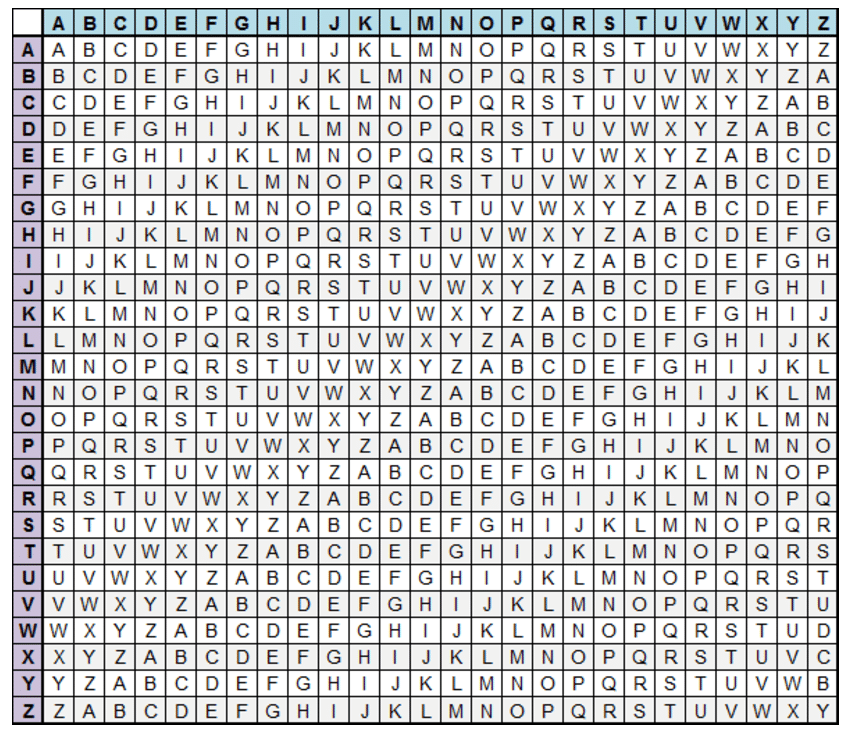
[**Test-driven development (TDD)**](https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1))is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is opposed to software being developed first and test cases created later.

*You should use test-driven development for at least part of your computer project.*

Use this sheet to practise test-driven development in the case of creating a Vigenère cipher encode and decode object.

## The Vigenère Code

Recall that the Vigenère code uses the Vigenère grid and a key word. The key word is repeated along the top of the plaintext word. For each letter in the plaintext, the ciphertext letter is created by cross referencing the key word letter on the column and the plain text letter on the row.



For example, encode this message using the keyword "TRAIN":

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Key Word | T | R | A | I | N | T | R | A | I | N | T | R | A | I | N |
| Plain Text | E | N | C | O | D | E | D | I | N | P | Y | T | H | O | N |
| Cipher Text |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## Set-up a testing case with a Vigenère Cipher class

* Start-up a new PyCharm project and call it Vigenere (you don't need a Virtualenv, you can choose a "previously configured interpreter and choose the interpreter C:\Pythopn381-64\python.exe)
* Create a main.py file for your code and then do File -> New -> Python file and choose Python unit test. Call the file "test\_main"
* Start building a bare bones VignereCipher class in main.py:

class VigenereCipher:  
 def \_\_init\_\_(self, keyword):  
 self.keyword = keyword

* Pycharm will have already started writing some of the the test\_main module. Edit it to test that an instance of the VigenereCipher can be created with a keyword attribute:

import unittest from main import VigenereCipher  
  
  
class VigenereTestCase(unittest.TestCase):  
 def setUp(self):  
 self.cipher = VigenereCipher("TRAIN")  
  
 def test\_init(self):  
 self.assertEqual(self.cipher.keyword, "TRAIN")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

* The setUp function runs before any other tests – in this case it creates a VigenereCipher instance in self.cipher, with the keyword "TRAIN"
* Run the test – it should pass successfully!

STOP – DISCUSS HOW YOU CAN BREAK DOWN THE PROBLEM OF CODING AND DECODING MESSAGES USING THE VIGENERE CIPHER. WHAT METHODS WILL YOU WRITE FOR THE VIGENERECIPHER CLASS TO BREAK DOWN THE PROBLEM

## VigenereCipher Methods

Let us assume that we decide to break the Vigenere cipher down into the following methods:

.combine\_character(plain\_letter, keyword\_letter) # encode a single character  
.extend\_keyword(self, rqd\_length) # copies keyword to require length  
.encode(self, plaintext) # encodes plaintext using keyword  
.separate\_character(cipher\_letter, keyword\_letter) # translate a cipher letter to plaintext  
.decode(self, ciphertext) # decode ciphertext using keyword

The test-driven cycle now proceeds as follows:

1. Write tests in your test plan describing the input and the expected output of each test.
2. Write a unit test in the test.main module that will use assert statements to check that the output from your functions or methods is as expected
3. Finally write the method code. If the tests don't pass, the code should be debugged to ensure that it works before moving onto other methods.

The test plan on the next page includes some initial tests. Two of the tests are shown below:

class VigenereTestCase(unittest.TestCase):  
 def setUp(self):  
 self.cipher = VigenereCipher("TRAIN")  
  
 def test\_init(self):  
 self.assertEqual(self.cipher.keyword, "TRAIN")  
  
 def test\_combine\_character(self):  
 self.assertEqual(self.cipher.combine\_character("D", "E"), "H")  
  
 def test\_combine\_non\_alpha(self):  
 self.assertRaises(ValueError, self.cipher.combine\_character, "@", "t")

## 

Write the remaining tests in the test plan as TestCases. Then write the .combine\_charater method and check that it passes all of the tests.

Proceed to fill in the Test Plan, write TestCases and finally write methods which satisfy the requirements of the Test Plan.

## Test Plan – VigenereCipher Object

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Name** | **Testing** | **Type** | **Input Data** | **Expected Output** |
| test\_init | keyword is successfully passed to keyword attribute | N | keyword = "TRAIN" | self.keyword = "TRAIN" |
| test\_combine\_character | Character is correctly shifted | N | plain\_letter = "D"  keyword\_letter = "E" | "H" |
| test\_combine\_ character\_mod | Character is shifted past the end of the alphabet | N | plain\_letter = "W"  keyword\_letter = "F" | "B" |
| test\_combine\_lower | Check that combine character works with lower case characters | N | plain\_letter = "y" keyword\_letter = "t" | "R" |
| test\_combine\_  non\_alpha | Check that an error is raised if inputs are not alphabetic (a-z or A-Z) | E | plain\_letter = "@" keyword\_letter = "t" | ValueError raised |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

## Refactoring

One of the principles of programming is Don't Repeat Yourself (DRY). Look at the code that you have written. Have you repeated yourself anywhere? Could you [*refactor*](https://en.wikipedia.org/wiki/Code_refactoring) your code so that repeated lines are put into a single helper method? One of the benefits of the test-driven approach is that the refactoring should not change the functionality of the code so that your unit tests should still work.

# The Test-Driven Development Environment

The [test-driven development process is described in Wikipedia](https://en.wikipedia.org/wiki/Test-driven_development):

**1. Add a test**

The adding of a new feature begins by writing a test that passes [iff](https://en.wikipedia.org/wiki/Iff" \o "Iff) the feature's specifications are met. The developer can discover these specifications by asking about [use cases](https://en.wikipedia.org/wiki/Use_case) and [user stories](https://en.wikipedia.org/wiki/User_story). A key benefit of test-driven development is that it makes the developer focus on requirements *before* writing code. This is in contrast with the usual practice, where unit tests are only written *after* code.

**2. Run all tests. The new test *should fail* for expected reasons**

This shows that new code is actually needed for the desired feature. It validates that the [test harness](https://en.wikipedia.org/wiki/Test_harness) is working correctly. It rules out the possibility that the new test is flawed and will always pass.

**3. Write the simplest code that passes the new test**

Inelegant or [hard code](https://en.wikipedia.org/wiki/Hard_code) is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.

**4. All tests should now pass**

If any fail, the new code must be revised until they pass. This ensures the new code meets the [test requirements](https://en.wikipedia.org/wiki/Software_requirements) and does not break existing features.

**5. Refactor as needed, using tests after each refactor to ensure that functionality is preserved**

Code is [refactored](https://en.wikipedia.org/wiki/Code_refactoring) for [readability](https://en.wikipedia.org/wiki/Computer_programming#Readability_of_source_code) and maintainability. In particular, hard-coded test data should be removed. Running the test suite after each refactor helps ensure that no existing functionality is broken.

* Examples of refactoring:
  + moving code to where it most logically belongs
  + removing [duplicate code](https://en.wikipedia.org/wiki/Duplicate_code)
  + making [names](https://en.wikipedia.org/wiki/Name) [self-documenting](https://en.wikipedia.org/wiki/Self-documenting_code)
  + splitting methods into smaller pieces
  + re-arranging [inheritance hierarchies](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

**Repeat**

The cycle above is repeated for each new piece of functionality. Tests should be small and incremental, and commits made often. That way, if new code fails some tests, the programmer can simply [undo](https://en.wikipedia.org/wiki/Undo) or revert rather than [debug](https://en.wikipedia.org/wiki/Debug) excessively. When using [external libraries](https://en.wikipedia.org/wiki/Library_(computing)), it is important not to write tests that are so small as to effectively test merely the library itself,[[4]](https://en.wikipedia.org/wiki/Test-driven_development#cite_note-Newkirk-4) unless there is some reason to believe that the library is buggy or not feature-rich enough to serve all the needs of the software under development.