Test-Driven Development (TDD)

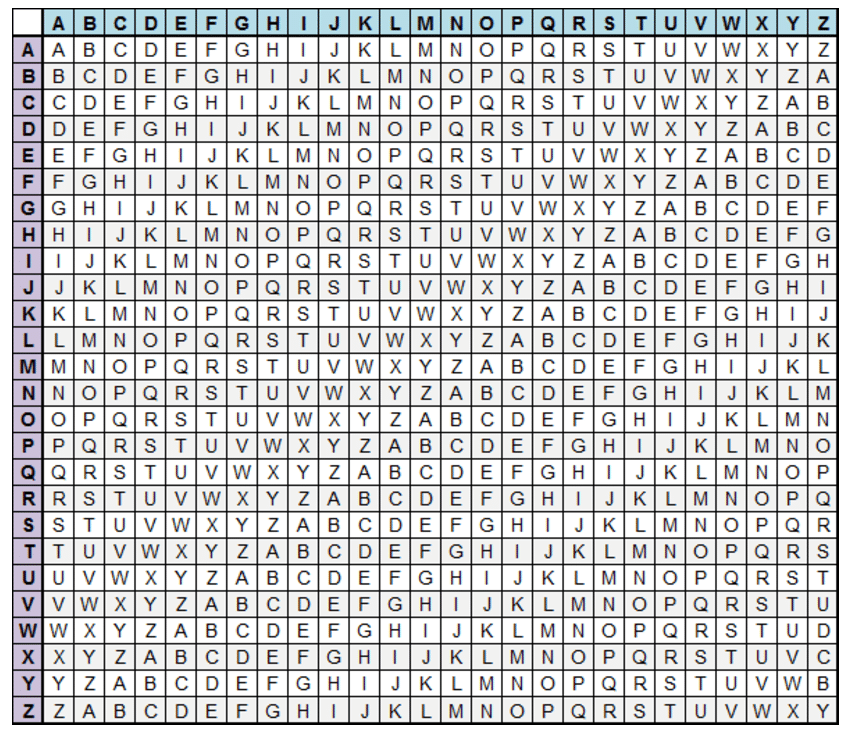
[**Test-driven development (TDD)**](https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1))is a software 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

* Open up your TestDrivenDevelopment Project – this should be configured to testing using PyTest
* Create a “vigenere.py” module
* Start building a bare bones VignereCipher class in main.py:

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

* To begin testing, right-click inside the class name and choose, Go To -> Test. This will create a “test\_vigenere.py” module with some starting code.
* Edit the code to create a TestVigenereCipher class as below

from vigenere import VigenereCipher

import pytest

class TestVigenereCipher:

@pytest.fixture()

def my\_cipher(self):

return VigenereCipher("TRAIN")

def test\_init(self, my\_cipher):

assert my\_cipher.keyword == "TRAIN"

* Note the @pytest.fixture() decorator means that you can write some code to set-up an instance of the class object. This is called a “fixture” and can be used in all your subsequent tests if you include my\_cipher in the parameters for the test.
* 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

Note that in Python convention, methods starting with an underscore are private methods: we are indicating that these functions should only be called from other VigenereCipher methods, they are not designed to be called from outside the function. Python is a *permissive* language, which means that it does not actually private methods from being used outside the function, other languages, such as Java and C++ have stricter conventions that prevent you using private methods outside the function.

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 more tests are shown below:

class TestVigenereCipher:

@pytest.fixture()

def my\_cipher(self):

return VigenereCipher("TRAIN")

def test\_init(self, my\_cipher):

assert my\_cipher.keyword == "TRAIN"

def test\_combine\_character(self, my\_cipher):

assert my\_cipher.\_combine\_character("D", "E") == "H"

def test\_combine\_non\_alpha(self, my\_cipher):

with pytest.raises(ValueError):

my\_cipher.\_combine\_character("@", "t")

## 

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

Now work on the other methods:

* Decide what the methods aught to do
* Write a tests for the methods in the test plan
* Write pytest unit tests to match the test plan
* Write the method
* Run the tests for the method
* If any tests fail de-bug the method (or the test)

## Test Plan – VigenereCipher Object

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Name** | **Method tested** | **Testing** | **Type** | **Input Data** | **Expected Output** |
| test\_init | .\_\_init\_\_ | keyword is successfully passed to keyword attribute | N | keyword = "TRAIN" | self.keyword = "TRAIN" |
| test\_combine\_character | .\_combine\_character | Character is correctly shifted | N | plain\_letter = "D"  keyword\_letter = "E" | "H" |
| test\_combine\_ character\_mod | .\_combine\_character | Character is shifted past the end of the alphabet | N | plain\_letter = "W"  keyword\_letter = "F" | "B" |
| test\_combine\_lower | .\_combine\_character | Check that combine character works with lower case characters | N | plain\_letter = "y" keyword\_letter = "t" | "R" |
| test\_combine\_  non\_alpha | .\_combine\_character | 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 if and only if 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**

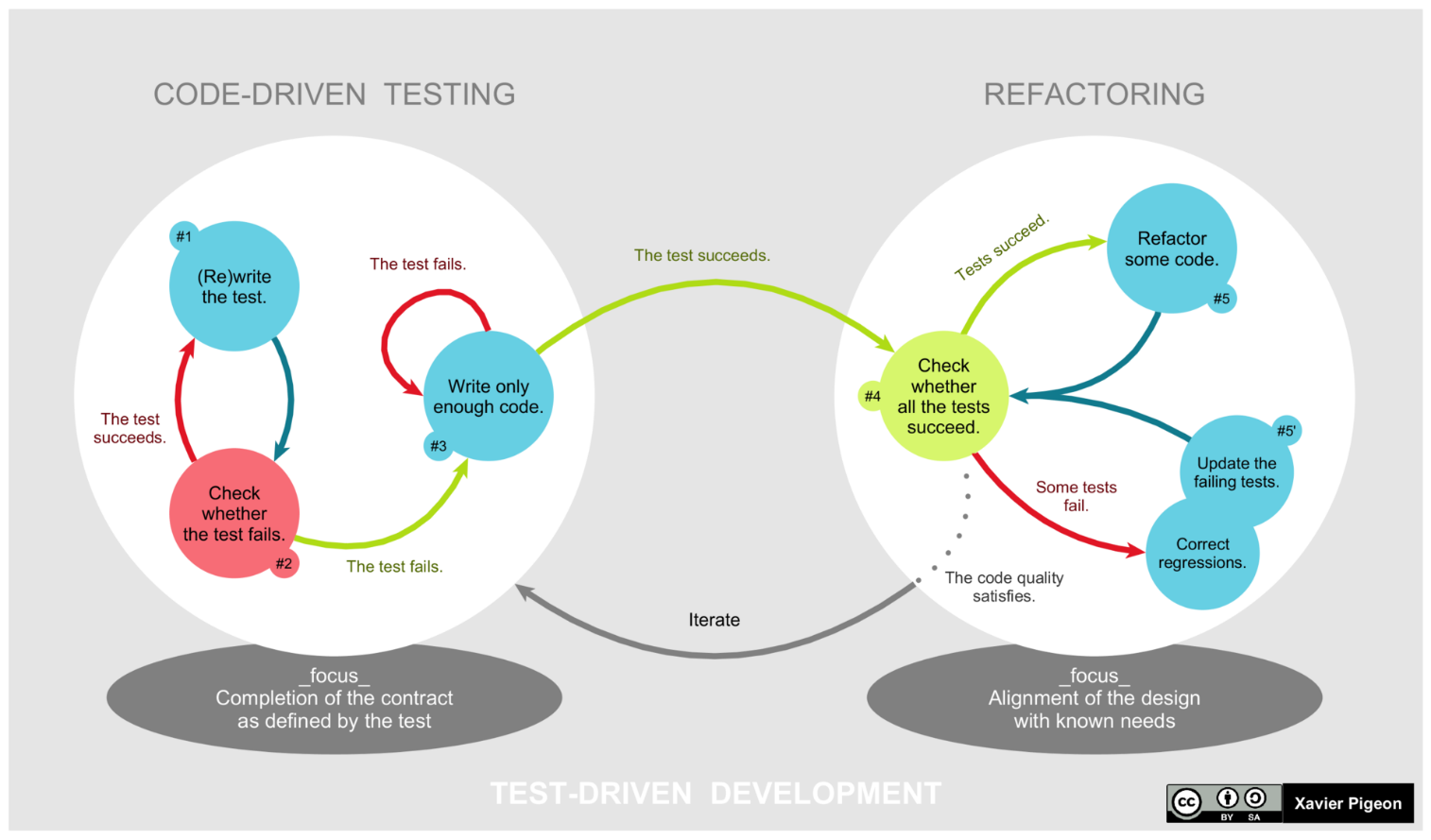
Inelegant or [hard code](https://en.wikipedia.org/wiki/Hard_code) is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.

**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.



# Refactoring Code

* 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, 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.

# Example – refactor Vigenère

In the Vigenère code the method code for encoding and decoding a string are extremely similar:

def encode(self, plaintext):  
 if not isinstance(plaintext, str):  
 raise TypeError("Plaintext must be a string")  
 plaintext\_msg = plaintext.replace(" ", "")  
 if not plaintext\_msg.isalpha():  
 raise ValueError("Plaintext must include only alphabetic characters (a-z or A-Z")  
 cipher\_letters = [self.\_combine\_character(p, k)  
 for p, k in zip(plaintext\_msg, self.extend\_keyword(len(plaintext\_msg)))]  
 return "".join(cipher\_letters)  
  
def decode(self, ciphertext):  
 if not isinstance(ciphertext, str):  
 raise TypeError("Ciphertext must be a string")  
 ciphertext\_msg = ciphertext.replace(" ", "")  
 if not ciphertext\_msg.isalpha():  
 raise ValueError("Ciphertext must include only alphabetic characters (a-z or A-Z")  
 cipher\_letters = [self.\_separate\_character(p, k)  
 for p, k in zip(ciphertext\_msg, self.extend\_keyword(len(ciphertext\_msg)))]  
 return "".join(cipher\_letters)

What is the only differences between the two pieces of code?

Why is it preferable to avoid having two separate functions that repeat much of the same code?

Re-factor the code to include a method:

def \_code(self, text, combine\_function)

This method should be called by the encode and decode methods and do most of the work. The \_encode method takes advantage of Python's first-class functions. ([Python's Functions Are First Class](https://dbader.org/blog/python-first-class-functions)). This means that the we can pass either \_combine\_characters or \_separate\_characters as arguments into \_encode, making it more flexible.

## Refactoring with Unittests

This kind of refactoring would be very risky in a project without unit tests. This is because there is a big chance that the refactor will introduce a bug which might break the functionality of the rest of the project.

However, if the project has sufficient unit tests, we can be much more confident about refactoring. Simply change the code and re-run the tests – any bugs that have been introduced can be spotted by the tests and corrected (called Red-Green refactoring).

# Vigenere Cipher Code and Tests in full

## VigenereCipher Class

class VigenereCipher:

def \_\_init\_\_(self, keyword):

if not isinstance(keyword, str):

raise TypeError("Keyword must be a string")

if not keyword.isalpha():

raise ValueError("Keyword must include only alphabetic characters (a-z or A-Z")

self.keyword = keyword.upper()

@staticmethod

def \_combine\_character(plain\_letter, keyword\_letter):

if not (plain\_letter.isalpha() and keyword\_letter.isalpha()):

raise ValueError("Plain letter and keyword letter must both be alphabetic (a-z or A-Z)")

plain\_num = ord(plain\_letter.upper()) - ord('A')

keyword\_num = ord(keyword\_letter.upper()) - ord('A')

cipher\_num = (plain\_num + keyword\_num) % 26

return chr(cipher\_num + ord('A'))

@staticmethod

def \_separate\_character(cipher\_letter, keyword\_letter):

cipher\_num = ord(cipher\_letter.upper()) - ord('A')

keyword\_num = ord(keyword\_letter.upper()) - ord('A')

plain\_num = (cipher\_num - keyword\_num) % 26

return chr(plain\_num + ord('A'))

def \_extend\_keyword(self, msg\_length):

d, m = divmod(msg\_length, len(self.keyword))

return self.keyword \* d + self.keyword[:m]

def encode(self, plaintext):

if not isinstance(plaintext, str):

raise TypeError("Plaintext must be a string")

plaintext\_msg = plaintext.replace(" ", "")

if not plaintext\_msg.isalpha():

raise ValueError("Plaintext must include only alphabetic characters (a-z or A-Z")

cipher\_letters = [self.\_combine\_character(p, k)

for p, k in zip(plaintext\_msg, self.\_extend\_keyword(len(plaintext\_msg)))]

return "".join(cipher\_letters)

def decode(self, ciphertext):

if not isinstance(ciphertext, str):

raise TypeError("Ciphertext must be a string")

ciphertext\_msg = ciphertext.replace(" ", "")

if not ciphertext\_msg.isalpha():

raise ValueError("Ciphertext must include only alphabetic characters (a-z or A-Z")

cipher\_letters = [self.\_separate\_character(p, k)

for p, k in zip(ciphertext\_msg, self.\_extend\_keyword(len(ciphertext\_msg)))]

return "".join(cipher\_letters)

### Refactored code

def \_code(self, text, combine\_function):

if not isinstance(text, str):

raise TypeError("Entered text must be a string")

text = text.replace(" ", "")

if not text.isalpha():

raise ValueError("Entered text must include only alphabetic characters (a-z or A-Z")

extended\_keyword = self.\_extend\_keyword(len(text))

coded\_letters = [combine\_function(p, k)

for p, k in zip(text, extended\_keyword)]

return "".join(coded\_letters)

def encode(self, plaintext):

return self.\_code(plaintext, self.\_combine\_character)

def decode(self, cipher\_text):

return self.\_code(cipher\_text, self.\_separate\_character)

## VigenereClass Unit Tests

from vigenere import VigenereCipher

import pytest

class TestVigenereCipher:

@pytest.fixture()

def my\_cipher(self):

return VigenereCipher("TRAIN")

def test\_init(self, my\_cipher):

assert my\_cipher.keyword == "TRAIN"

def test\_init\_lower(self):

my\_cipher = VigenereCipher("taxi")

assert my\_cipher.keyword == "TAXI"

def test\_init\_invalid(self):

with pytest.raises(ValueError):

VigenereCipher("$%FG")

with pytest.raises(TypeError):

VigenereCipher(56)

def test\_combine\_character(self, my\_cipher):

assert my\_cipher.\_combine\_character("D", "E") == "H"

def test\_combine\_character\_mod(self, my\_cipher):

assert my\_cipher.\_combine\_character("W", "F") == "B"

def test\_combine\_lower(self, my\_cipher):

assert my\_cipher.\_combine\_character("y", "t") == "R"

def test\_combine\_non\_alpha(self, my\_cipher):

with pytest.raises(ValueError):

my\_cipher.\_combine\_character("@", "t")

def test\_separate\_character(self, my\_cipher):

assert my\_cipher.\_separate\_character("W", "I") == "O"

assert my\_cipher.\_separate\_character("p", "e") == "L"

def test\_extend\_keyword(self, my\_cipher):

assert my\_cipher.\_extend\_keyword(12) == "TRAINTRAINTR"

cipher = VigenereCipher("CHEESE")

assert cipher.\_extend\_keyword(16) == "CHEESECHEESECHEE"

def test\_encode(self, my\_cipher):

assert my\_cipher.encode("ENCODEDINPYTHON") == "XECWQXUIVCRKHWA"

assert my\_cipher.encode("Give me cheese please") == "ZZVMZXTHMRLVPTRTJE"

cipher = VigenereCipher("computer")

assert cipher.encode("Meet me at the bridge at dawn") == "OSQIGXEKVVQQLBHXGOFSUPR"

def test\_decode(self, my\_cipher):

assert my\_cipher.decode("XECWQXUIVCRKHWA") == "ENCODEDINPYTHON"

cipher = VigenereCipher("trunk")

assert cipher.decode("MFGBBKFQAOOVLQSXJ") == "TOMORROWNEVERDIES"