|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Student Name** | Colin Prater | | **Student Number** | | 469720981 |
| **Unit Code/s & Name/s** | ICTPRG405 Automate processes | | | | |
| **Assessment Type** | Portfolio | | | | |
| **Assessment Name** | Python programming project | **Assessment Task No.** | | | AT1 |
| **Assessment Due Date** | Mon 30th Aug 2021 | **Date submitted** | | |  |
| **Assessor Name** | Elan | | | | |
| **Student Declaration:** I declare that this assessment is my own work. Any ideas and comments made by other people have been acknowledged as references. I understand that if this statement is found to be false, it will be regarded as misconduct and will be subject to disciplinary action as outlined in the TAFE Queensland Student Rules. I understand that by emailing or submitting this assessment electronically, I agree to this Declaration in lieu of a written signature. | | | | | |
| **Student Signature** |  | | | **Date** |  |
| **PRIVACY DISCLAIMER:** TAFE Queensland is collecting your personal information for assessment purposes. The information will only be accessed by authorised employees of TAFE Queensland. Some of this information may be given to the Australian Skills Quality Authority (ASQA) or its successor and/or TAFE Queensland for audit and/or reporting purposes. Your information will not be given to any other person or agency unless you have given us written permission or we are required by law. | | | | | |

|  |  |
| --- | --- |
| **Instructions to Student** | **General Instructions:**  In this assessment, as an individual you will design, develop and test a software application using one or more algorithms to perform a desired operation. Full details of the work you need to do are given below. You will also use an integrated development environment (IDE) to do your software development.  **Information / Materials provided:** You will be provided with access to a suitable IDE for software development.  **Assessment Criteria:**  To achieve a satisfactory result, your assessor will be looking for your ability to demonstrate the following key skills/tasks/knowledge to an acceptable industry standard:   * *Design algorithms to perform specific tasks* * *Design the algorithms using an abstract pseudo-code* * *Use structures, sequence, selection and iteration in the algorithms to perform their task* * *Translate the pseudo-code of the algorithms into the Python language* * *Test the Python code for errors, omissions and for mishandling of inputs, and fix these errors* * *Provide internal documentation in the Python code* * *Provide external user documentation for a Python program*   **Number of Attempts:**  You will receive up to two (2) attempts at this assessment task. Should your 1st attempt be unsatisfactory (U), your teacher will provide feedback and discuss the relevant sections / questions with you and will arrange a due date for the submission of your 2nd attempt. If your 2nd submission is unsatisfactory (U), or you fail to submit a 2nd attempt, you will receive an overall unsatisfactory result for this assessment task. Only one re-assessment attempt may be granted for each assessment task. **For more information, refer to the Student Rules.** |
| **Submission details** | Insert your details on page 1 and sign the Student Declaration. Include this template with your submission.  *Assessment to be submitted via*   * *TAFE Queensland Learning Management System: Connect url:* [*https://connect.tafeqld.edu.au/d2l/login*](https://connect.tafeqld.edu.au/d2l/login) * *Username; 10 digit student number* * *For Password: Reset password go to* [*https://passwordreset.tafeqld.edu.au/default.aspx*](https://passwordreset.tafeqld.edu.au/default.aspx)*>* |
| **Instructions for the Assessor** | Students need to have access to a suitable IDE to work on the assessment tasks. However, they are allowed to work on this assessment off-campus; it is up to the student to organise off-campus access to an IDE. |
| **Note to Student** | An overview of all Assessment Tasks relevant to this unit is located in the Unit Study Guide. |

# The Purpose of this Assessment

Welcome to the programming assessment for this unit. By completing this assessment you will:

* develop one or more algorithms that are used to perform specific software functions
* understand how a large software application can be broken into a number of components called functions
* design each of the function components in pseudo-code, and then translate them to Python
* observe that each function provides a *contract* (what it guarantees to do) to the other components through its Application Program Interface (API)
* test each function’s API to verify that it meets the defined API contract that it is supposed to, or identify errors or omissions in the function’s behaviour (and then fix them)
* produce both internal documentation for the software application and the external documentation for the users of the application

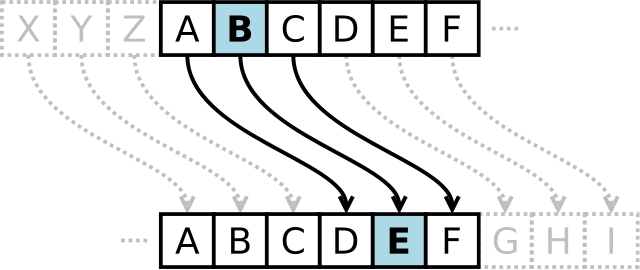
In terms of Cyber Security, what you are doing here is software testing and software verification. You are scripting a complicated application out of simpler parts. You will prove, by testing, that each part does exactly what it is supposed to do and nothing else. Once the software is verified, there should be few or no “bugs” in the software for would-be attackers to exploit.

Finally, to make this assessment even more relevant to Cyber Security, the software application that you develop performs a simple encryption and decryption algorithm. You will be able to encrypt text messages with your program and send them to your friends for decryption!

# The Purpose of the Program

The purpose of the program which you will design and write is to perform the Caesar cipher. This is a simple encryption cipher that uses the 26 uppercase letters of the English alphabet. Given a specific encryption key, the Caesar cipher *shifts* the letters in the plaintext (the unencrypted message) so that they are mapped to new letters in the *ciphertext* (the encrypted message).

As an example, imagine that the encryption key is the number +3. This means that each letter in the plaintext is shifted three places to the right:



(from Wikimedia, public domain)

The letter ‘B’ becomes the letter ‘E’, the letter ‘D’ becomes the letter ‘G’ etc. For the letters near the end of the English alphabet, we “wrap around” as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Plaintext letter | U | V | W | X | Y | Z |
| Ciphertext letter | X | Y | Z | A | B | C |

In this way, all 26 letters in the plaintext are mapped to 26 different letters in the ciphertext.

Here is an example of a plaintext message encrypted by the Caesar cipher with a shift of +3:

**MEETMEINTHEHOTELLOBBYATTHREEPM**

**PHHWPHLQWKHKRWHOOREEBDWWKUHHSP**

(The spaces in the plaintext are removed because they cannot be ciphered).

To decrypt this message, we apply a negative shift of the same distance, i.e. -3, to shift each letter 3 places to the left, revealing the original plaintext:

**PHHWPHLQWKHKRWHOOREEBDWWKUHHSP**

**MEETMEINTHEHOTELLOBBYATTHREEPM**

For more details of the Caesar cipher and its history, read this Wikipedia article:

<https://en.wikipedia.org/wiki/Caesar_cipher>

# The User’s Interface to the Program

By the end of this assessment, you will write a Python program called *caesar.py*. It will run from the command line in either Windows or Linux. The user will provide the shift value (i.e. the key) and the plaintext message on the command line. Your program will perform the Caesar cipher encryption on this message and print out the new text. An example of your program in action would be:

$ python caesar.py 3 ‘MEETMEINTHEHOTELLOBBYATTHREEPM’

PHHWPHLQWKHKRWHOOREEBDWWKUHHSP

And because the shift value can be negative as well as positive, you can use the same program for decryption as well as encryption:

$ python caesar.py -3 ‘PHHWPHLQWKHKRWHOOREEBDWWKUHHSP’

MEETMEINTHEHOTELLOBBYATTHREEPM

# Defensive Programming and Your APIs

In every section of the program, you will practice defensive programming. You will assume that any input that you are provided could be wrong, either by accident or on purpose (e.g. by an attacker). You will always check the inputs in every section of your program and deal with them. This means that:

* the user’s command line values must be checked
* for each and every function in your program, the function’s arguments must be checked

Your options may include:

* printing out an error message and stopping the program
* converting the input into something more suitable
* accepting the input as it is

Regardless, what you expect as inputs to each section and how you deal with all possible inputs must be documented, as this forms your *contract* (your API) with the other components of your program.

As an example, what would you do if a user ran your caesar.pl program with these inputs?:

$ python caesar.py +4000 ‘Words... $#^!&\*%! words, we hates them!!!’

What does a shift of 4,000 mean? And can the Caesar cipher deal with punctuation and lowercase letters?

# Pseudo-code for the Main Program

As someone who is still learning a programming language, it is good practice to first design the sections of your program in an abstract pseudo-code in (nearly) English, as you can understand this language. You can check your pseudo-code algorithm for obvious errors, and “run” your program by hand on a piece of paper. Once you are happy with the pseudo-code, you can then translate this into the real source code language that you are using, e.g. Python.

Here is the top-level pseudo-code for the Caesar program:

*Check that we have the right number of arguments on the command line: if not, an error*

*Read the values of these arguments: the shift value and the original text*

*Check that the values make sense*

*If possible, convert the values into new values that do make sense*

*If this is not possible, it’s an error*

*Using the shift value, shift all the letters in the original text to make the new text*

*Print out the new text*

You will take these top-level pseudo-code and break it into three section which will form the three tasks in this assessment.

* In task 1, you will write the code to convert the user’s input text into values that do make sense
* In task 2, you will write the code to shift all the text letters to do the encryption
* In task 3, you will write the rest of the code for the top-level of the program

# Task 1: Converting the Original Text

As a defensive programmer, we have to assume that the user will give us an input message on the command line which has characters which are not uppercase. But the Caesar algorithm can only encrypt uppercase letters. We need to deal with this.

We need a function which takes as input the string of text that the user gave us on the command line. It will return a new string of text which is suitable for encryption by the Caesar cipher, i.e. it must be all uppercase letters.

Here is the (partial) API for this function. It will be called *convert\_to\_Caesar()*. It will take one argument: a string of text. It will return: a string of text. The letters in the argument will be processed as follows:

* uppercase letters will be left untouched
* lowercase letters will be converted to uppercase letters: ‘a’ →’A’, ‘b’ → ‘B’ etc.
* the full stop ‘.’ will be translated to the letter ‘X’ (as ‘X’ is rare, we can use it to separate sentences)
* all other letters in the argument will be discarded

will convert all letters in the string to uppercase

user\_input = user\_input.upper()

if (next\_letter >= 65 and next\_letter <=90):

if next\_letter = X:

### Step 1: Pseudo-Code for This Function

You will design the algorithm for this function in pseudo-code, i.e. semi-English. The algorithm must have these properties:

* it must be guaranteed to stop and return a string regardless of the argument it is given
* it must obey the API contract defined above regardless of the argument it is given
* it must not crash or do unexpected things regardless of the argument it is given
* the algorithm must be written in simple enough steps that it can be translated into a computer language like Python
* the algorithm must be exact and well-defined, i.e. not open to interpretation

You will use English words like “if”, “while” and “for” where they are suitable. These make sense to an English reader, but they also indicate programming structures such as sequence, selection and iteration to a programmer.

Performance Criteria

1.1 Develop an algorithm that is an exact and sufficient description of the solution

1.2 Develop an algorithm that takes account of all possible situations

1.3 Develop an algorithm that is guaranteed to end

2.1 Use structures, sequence, selection and iteration

2.2 Use structures to describe algorithmic solutions to a problem

3.1 Create an abstract design, to fulfil the requirements of the proposed process`

### Step 2: Write Tests for the Function

The only way to guarantee that this function can meet its API contract is to give it all possible string inputs and check that the resulting output string is correct in each case. As there are an infinite number of input strings, this is impossible.

To verify that your algorithm is correct, you will write a set of at least 10 test cases for your function. Each test case consists of three things:

* the input string to the function
* what should be the output string from the function
* any comment about this test case

As a test case writer, it is your job to think deviously, cunningly and cruelly. What nasty test cases can you think of that may cause the function to break (do the wrong thing)? There is no point in testing your function with just nice and easy inputs.

Here are three example test cases. You have to write another 10 test cases:

|  |  |  |
| --- | --- | --- |
| Input string | Output String | Comment |
| Hello there | HELLOTHERE | Spaces are lost |
| It’s me. Do you understand? | ITSMEXDOYOUUNDERSTAND | Full stops become Xs |
| I studied in 2018. | ISTUDIEDINX | Digits are discarded |

On paper, use both your algorithm in pseudo-code and the inputs in your test cases to review your code and determine if there are any errors or omissions in your pseudo-code.

Below your test cases, document how you performed the on-paper review of your pseudo-code and any changes that you had to make as part of this review.

### Step 3: Convert the Pseudo-Code into Python

You have your algorithm in pseudo-code and you have tested it on paper to confirm the API contract. Now, translate your pseudo-code into Python. Create a Python file called *task1.py*. In this file, write a Python function called *convert\_to\_Caesar()* which is the pseudo-code translated to Python.

Include internal documentation in your Python code as follows:

* At the top of your function, put in these comment lines:

# Author: <your name>

# Date: <date that you wrote the code>

* Before the function, use comments to document the function’s purpose and API. What does it do? What input(s) does it expect? What type of result does it return? What will it do to the input(s)? What sorts of input count as bad inputs? What will the function do with bad inputs?
* Put comments before each main section of your function’s Python code. Remember: a useful comment explains what you are doing and why, not how this is being done. A programmer can read the code and see how the code works, but they can’t understand why you are doing it.

**Hint, hint:** Your pseudo-code already explains how your algorithm works, and possibly even explains why you are doing things that way. Why not reuse it as the comments in your Python code?

Performance Criteria

3.2 Review the abstract design for any omissions or errors

3.3 Translate the abstract design to the chosen language

3.4 Create the internal documentation

5.1 Create technical-level documentation

### Step 4: Test Your Python Code

Put lines of code in the main part of your Python program to test your Python function using the test cases that you defined above. Here are some examples using the three test cases shown above:

# Test 1

result= convert\_to\_Caesar('Hello there')

if (result == 'HELLOTHERE'):

print('Test 1 OK')

else:

print('Test 1 error: ' + result)

# Test 2

result= convert\_to\_Caesar("It's me. Do you understand?")

if (result == 'ITSMEXDOYOUUNDERSTAND'):

print('Test 2 OK')

else:

print('Test 2 error: ' + result)

# Test 3

result= convert\_to\_Caesar('I studied in 2018.')

if (result == 'ISTUDIEDINX'):

print('Test 3 OK')

else:

print('Test 3 error: ' + result)

Use these tests to check your function for errors and identify any inputs that are not covered correctly by your code. If you find errors, use the debugging functionality in your IDE to single-step your code, view the values of variables etc. to identify your bugs and fix them.

Performance Criteria

4.1 Check the script or code, for syntax and semantic errors

4.2 Identify any areas that are not covered, or are covered incorrectly, in the script or code

### Step 5: Upload Your Work

For task 1, you are to upload these two files to Connect:

* A text or Word document containing your pseudo-code for the *convert\_to\_Caesar()* function, and your test cases for the function.
* Your review of your pseudo-code and any changes that you had to make
* A working Python program with the *convert\_to\_Caesar()* function, with testing code in the main program to verify that it works correctly.

These are due in by the end of Friday, Week 3.

# Task 2: Encrypting the Converted Text

Now that we can convert the text entered by the user into a form which can be encrypted by the Caesar cipher, it is time to write a function to do this.

We need a function which takes as input the shift value and the string of text that has been converted. It will return a new string of text which has been encrypted by the Caesar cipher.

This is all I’m going to give you as the API for this function. It is up to you to specify more exactly what the API (*contract*) is that this function provides.

### Step 1: Pseudo-Code for This Function

You will design the algorithm for this function in pseudo-code, i.e. semi-English. The algorithm must have these properties:

* it must be guaranteed to stop and return a string regardless of the argument it is given
* it must obey the API contract defined above regardless of the argument it is given
* it must not crash or do unexpected things regardless of the argument it is given
* the algorithm must be written in simple enough steps that it can be translated into a computer language like Python
* the algorithm must be exact and well-defined, i.e. not open to interpretation

You will use English words like “if”, “while” and “for” where they are suitable. These make sense to an English reader, but they also indicate programming structures such as sequence, selection and iteration to a programmer.

Performance Criteria

1.1 Develop an algorithm that is an exact and sufficient description of the solution

1.2 Develop an algorithm that takes account of all possible situations

1.3 Develop an algorithm that is guaranteed to end

2.1 Use structures, sequence, selection and iteration

2.2 Use structures to describe algorithmic solutions to a problem

3.1 Create an abstract design, to fulfil the requirements of the proposed process`

### Step 2: Write Tests for the Function

To prove that your algorithm is correct, you will write a set of at least 10 test cases for your function. Each test case consists of four things:

* the input shift value
* the input string to the function
* what should be the output string from the function
* any comment about this test case

Here are three example test cases. You have to write another 10 test cases:

|  |  |  |  |
| --- | --- | --- | --- |
| Input Shift | Input string | Output String | Comment |
| 3 | TESTTHISFUNCTION | WHVWWKLVIXQFWLRQ | Encryption |
| -3 | WHVWWKLVIXQFWLRQ | TESTTHISFUNCTION | Decryption |
| 13 | PIANOSTOOL | CVNABFGBBY | ROT-13 |

### Step 3: Convert the Pseudo-Code into Python

You have your algorithm in pseudo-code and you have tested it on paper to confirm the API contract. Now, translate your pseudo-code into Python. Create a Python file called *task2.py*. In this file, write a Python function called *encrypt\_Caesar()* which is the pseudo-code translated to Python.

Include internal documentation in your Python code as follows:

* At the top of your function, put in these comment lines:

# Author: <your name>

# Date: <date that you wrote the code>

* Before the function, use comments to document the function’s purpose and API. What does it do? What input(s) does it expect? What type of result does it return? What will it do to the input(s)? What sorts of input count as bad inputs? What will the function do with bad inputs?
* Put comments before each main section of your function’s Python code. Remember: a useful comment explains what you are doing and why, not how this is being done. A programmer can read the code and see how the code works, but they can’t understand why you are doing it.

Performance Criteria

3.2 Review the abstract design for any omissions or errors

3.3 Translate the abstract design to the chosen language

3.4 Create the internal documentation

5.1 Create technical-level documentation

### Step 4: Test Your Python Code

Put lines of code in the main part of your Python program to test your Python function using the test cases that you defined above. Use the Python testing code given in Task 1 as a template to write your tests for Task 2.

Use these tests to check your function for errors and identify any inputs that are not covered correctly by your code.

Performance Criteria

4.1 Check the script or code, for syntax and semantic errors

4.2 Identify any areas that are not covered, or are covered incorrectly, in the script or code

### Step 5: Upload Your Work

For task 2, you are to upload these two files to Connect:

* A text or Word document containing your pseudo-code for the *encrypt\_Caesar()* function, and your test cases for the function.
* A working Python program with the *encrypt\_Caesar()* function, with testing code in the main program to verify that it works correctly.

These are due in by the end of Friday, Week 6.

# Task 3: The Full Caesar Program

We now have:

* The top-level pseudo code for the Caesar program: get inputs from the user, deal with the input, encrypt the input, print out the encrypted version,
* A function called *convert\_to\_Caesar()* which converts the user’s input into something suitable to be encrypted, and
* A function called *encrypt\_Caesar()* which does the encryption work

Your task now is to create a single Python program called *caesar.py*. It will contain the two functions from the previous tasks. It will read in the user’s inputs from the command line and perform the top-level pseudo-code. As you have already been given this pseudo-code, there is no need for you to write it.

### Step 1: Write Tests for the Program

This time, we are writing a full Python program (i.e. a software application) and not just a function. However, we can still test it, as the user will enter two input values on the command line, and your program will output a piece of encrypted text.

To prove that your Python program is correct, you will write a set of at least 5 test cases for your function. Each test case consists of four things:

* the input shift value which the user would enter on the command line
* the input string which the user would enter on the command line
* what your program should print out
* any comment about this test case

As this is a program and not a function, it can behave quite differently to a function. For example:

* the program might decide to print an error message
* the program might decide to exit and not print out a line of text

Do not use any of the 10 test cases that you previously wrote. Instead, think of new test cases that exhibit the behaviour of your program, not of the functions in your program.

### Step 2: Convert the Pseudo-Code into Python

Take the top-level pseudo code and translate it into Python. Use the two functions that you previously wrote to perform some of the functionality. Remember to be a defensive programmer and check the inputs from the user for validity.

Document your Python code as follows:

* At the top of your program, put in these comment lines:

# Author: <your name>

# Date: <date that you wrote the code>

* After these lines, use comments to document the program’s purpose and API. What does it do? What input(s) does it expect, and where do they comes from? What type of output will it produce? What sorts of input count as bad inputs? What will the function do with bad inputs? Will any error messages be printed out? Will the program exit on error inputs?
* Put comments before each main section of your main program’s Python code. Remember: a useful comment explains what you are doing and why, not how this is being done. A programmer can read the code and see how the code works, but they can’t understand why you are doing it.

Run your program from the command-line, and use your tests to check your program for errors and identify any inputs that are not covered correctly by your code.

Performance Criteria

4.1 Check the script or code, for syntax and semantic errors

4.2 Identify any areas that are not covered, or are covered incorrectly, in the script or code

### Step 3: User Documentation

All software should come with internal documentation for both the programmer (so that the software can be maintained) and external documentation for the user (so that the user knows how to use the program). Historically, the Unix operating system came with a set of manual pages; each program had its own manual page with these sections:

* NAME: name and purpose of the program
* SYNOPSIS: an abstract description of how to run it on the command line
* DESCRIPTION: a description of the program’s purpose, how to use it, what errors it deal with etc.
* SEE ALSO: if needed, a list of related programs and commands
* BUGS: if the program cannot deal with all possible user inputs, these are described here.

As as example, read [this example of the cat command from the 7th Edition of Unix](http://man.cat-v.org/unix_7th/1/cat).

You will write a Word document which is a user manual for your *caesar.py* program which is

structured like the Unix manual pages.

Performance Criteria

5.2 Create user-level documentation

### Step 4: Upload Your Work

For task 3, you are to upload these two files to Connect:

* A Word document containing the user manual for your *caesar.py* program
* A text or Word document containing your test cases for the program.
* A working Python program called *caesar.py* which is fully documented.

These are due in by the end of Friday, Week 9.