**What is steganography and how can I use it to conceal a message in a digital image, and then retrieve it?**

**Abstract**

This paper discusses the science of steganography and its uses in the world of cyber security and protecting digital data, including past examples and an in-depth analysis of some modern techniques. Examples are provided to illustrate the processes used, how they can be countered, and the future potential of steganography in avenues such as cyber forensics and watermarking. In particular, it explores Least Significant Bit modification and how this technique can be implemented to hide a message within a cover image. It details how I have reproduced this technique in Python, creating a program which produces a stegoimage from a provided image and message, and then extracts the message after concealment.

**Introduction**

Humans have always felt the need to communicate privately with one another. Some information, such as banking details or military secrets, should never be shared with the public. In recent times, the integration of technology into our lives has been accompanied by frequent cases of data theft and violation of privacy. In order to protect our data from prying eyes, two techniques have arisen: cryptography, the act of encrypting data so that potential eavesdroppers cannot comprehend the contents; and its counterpart, steganography – hiding a message’s very existence so eavesdroppers aren’t aware of it at all. Encryption proved essential to communicating covertly in the past, a famous example being the Enigma machine, which the Germans used to scramble military messages during World War 2. However, while steganography is less well-known, it may become a crucial part of data security.

In this project I aim to explore the idea of steganography, discussing its origins and early examples before delving into its modern uses, including who may use it and its advantages and disadvantages compared to cryptography. Then, I will discuss how I recreated one of the techniques into a Python program, including my planning, development process and final result. I will explain in detail how the technique works and the steps taken to integrate it into Python, including the challenges that arose, in order to answer my question: What is steganography and how can I use it to conceal a message in a digital image, and then retrieve it?

**Secondary Research**

Definition

Steganography is defined as ‘the art and science of invisible communication.’ (Morkel, Eloff, & Olivier, 2005). This trio represent the University of Pretoria; numerous listed sources and links to each author’s academic email address mark their article as trustworthy. The aim of steganography is to conceal the existence of a message as it is sent to a recipient, ‘without possible eavesdroppers even knowing there is a form of communication in the ﬁrst place’ (Krenn, 2004). Krenn’s article, on the other hand, contains no background information about its purpose or reliability, but it is cited in many other sources. In other words, ‘trying to hide the fact that we’ve sent a message at all’ (Pound, 2015).Dr Pound is an Image Analyst at the University of Nottingham, and part of a channel called Computerphile, renowned for providing well-researched educational videos.

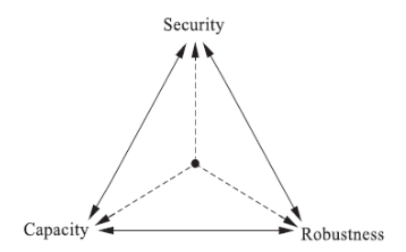
The word ‘steganography’ is a combination of Greek words ‘stegos’, meaning ‘cover’, and ‘grafia’, meaning ‘writing’(Morkel, Eloff, & Olivier, 2005). Typically, a message is embedded into another file, called a carrier, which combined with the message is known as a stego-carrier. For example, an image concealing a message is known as a stego-image (Kumar & Pooja, 2010). Kumar and Pooja’s source is unclear in places and contains uncited facts with the same wording as in Krenn’s earlier article, making this source seem less credible. ‘Hiding information may require a stego key’ (Kumar & Pooja, 2010) –additional information required for embedding and retrieving the message, much like keys used in encryption. Another journal article identifies the three objectives of steganography techniques:to be undetectable, robust and have a high capacity. However, it is difficult to account for all three at once and many techniques favour one aspect over the others. (Cheddad, Condell, Curran, & McKevitt, 2010). Representing the University of Ulster, these four authors have all worked in the computer science industry, some as professors, so their work seems very reliable.

Figure 1: The relationship between security, robustness and capacity (Febryan, Purboyo, & Saputra, 2017).

Febryan, Purboyo and Saputra complement this idea by providing Figure 1, illustrating how as you favour one aspect, you tend to reduce the other two. Their journal article is from Telkom University and contains many references, although its grammar is unclear in places reducing its trustworthiness. Steganography itself also has a contradictory nature – the recipient must be able to retrieve the secret, but any other interceptors should not be able to find it.

History

In conjunction with the word’s origin, ‘it is believed that steganography was first practiced during the Golden Age in Greece’ (Kumar & Pooja, 2010). Some of the earliest known cases were recorded by Greek historian Herodotus, referred to as ‘the father of history’ (Kahn, 1996). Kahn is an acclaimed author who has written extensively about military intelligence and cryptography, so it follows that his work on steganography, its counterpart, is very dependable. An example is ‘Harpagus, a man who killed a hare and hid a message inside its body, sending it with a messenger who pretended to be a hunter’ (Kahn, 1996). Herodotus also writes of a nobleman named Histaieus, who was anxious to give Aristagoras orders to revolt. Afraid the roads were guarded, he took his most trusted slave and shaved the hair off his head, pricking a message upon the skin and dispatching him only once the hair had regrown (Herodotus, 430 BC). This is the same book referred to by Kahn, translated by Robin Waterfield. Herodotus’ account is one of few historical texts from this period, and while it is difficult to closely verify its reliability, it is widely trusted by many historians nowadays. Another method was to write on the underside of a wax tablet, then cover it with a new wax layer, giving the appearance of a blank tablet (Krenn, 2004). These methods would prove impractical in the 21st century; however, a lack of modern technology made them much more difficult to detect than techniques used today.

Another technique was developed by Aeneas the Tactician, where near-invisible pinpricks were placed above certain letters in a magazine or news article, indicating them as part of a hidden message. This was used throughout the Renaissance period(Kahn, 1996)*,* when England was divided between the Protestants, supporting Queen Elizabeth I, and the Catholics, ‘who sought to unseat Elizabeth and return a Catholic monarch [Mary Queen of Scots] to the throne’ (Budiansky, 2011). This is a biography of Sir Francis Walsingham, principal secretary to Queen Elizabeth I, famous for developing the highly effective predecessor to our intelligence system, including double agents, propaganda and code breaking (Budiansky, 2011). Budiansky has written numerous historical books and biographies; it seems his work is very trustworthy. Walsingham uncovered coded messages to Mary smuggled inside beer barrels, providing enough evidence of her plots against Queen Elizabeth for Mary’s conviction and execution (Budiansky, 2011). This proved crucial to protecting Elizabeth’s crown.

Historically, using text to conceal messages was the most common form of steganography (Morkel, Eloff, & Olivier, 2005). During the Second World War, a technique called the Microdot came into use: a page-sized photograph is reduced in size to just 1mm (Kipper, 2003). Kipper has written two books detailing digital crime, in which he references many sources, making his work seem dependable. This allowed it to fit inside a full stop, which was pasted onto a document and sent to the recipient.



Figure 2: An example of a microdot (Microdot Technology, 2019).

This was extremely difficult to detect (Morkel, Eloff, & Olivier, 2005), making it very useful at a time when secret communication was of the essence.

Modern applications

However, the introduction of the Internet has dramatically changed the way we communicate. We can talk to people across the globe in an instant, and control our banking and shopping from the comfort of home. The Internet is a public, shared system, yet we use it to exchange private information such as credit card details and passwords (Epner, 2019) Epner is a cybersecurity worker for an unnamed US national intelligence agency. She provides an educational video on behalf of Khan Academy, and these factors combined make her source seem very trustworthy. This information needs protecting, or anyone could intercept it and see your data. For most, the solution is cryptography. Literally meaning ‘secret writing’ (Kahn, 1996), it is defined by Network Encyclopaedia as ‘the use of codes to convert data so that only a specific recipient will be able to read it, using a key.’ (Network Encyclopaedia, n.d.). This digital encyclopaedia’s articles must be thoroughly checked before being posted, so it is likely they are accurate. Anyone without the key is met with a random string of characters – while they cannot understand the message, it is quite obvious that one has been hidden. This issue is addressed by steganography.

According to Morkel, Eloff and Olivier, ‘research in steganography has mainly been driven by a lack of strength in cryptographic systems’ (Morkel, Eloff, & Olivier, 2005). This may not have been true for classical steganography, but it has certainly become a reason to invest in steganography techniques in the digital age. Currently, online communication uses a technique called asymmetric encryption: a public key is used to encrypt messages, and can be shared with anyone; but only the private key can decrypt them, and this is kept secret (Epner, 2019). This technique exploits the fact that taking a very large number (the public key) and finding its factors is very slow, even using powerful computers (Reich, 2019). Reich is a physicist and creator of popular educational YouTube channel MinutePhysics. The multiple references in the video’s description suggest that the information in his source is credible. Epner points out that ‘public key cryptography is the foundation of all secure messaging on the open internet’ (Epner, 2019). However, Reich’s source discusses a new technique that is guaranteed to threaten this system – by manipulating quantum physics using a method called Shor’s Algorithm, finding factors of the public key becomes much faster, allowing the user to very quickly decrypt data encrypted in this way (Reich, 2019). This would threaten the security of almost all current internet-based communications. Luckily, Reich also states that the current implementations of Shor’s Algorithm are far from having enough memory to factorise public keys currently in use (Reich, 2019). However, in the future, asymmetric encryption could be very vulnerable to quantum attacks – another reason to investigate steganography.

Another advantage is that as a result of it being less common, steganography receives far less attention from security services than cryptography (Cameron, 2018).Cameron is a senior writer for the IEEE Computer Society and holds multiple qualifications, writing frequently for the Computer magazine. Her work appears quite credible. This means that there are less efforts to find data hidden using steganography compared with decrypting codes. Although Pound counters that as the amount of steganography has increased, so have the attempts, and number of approaches, to try and find it (Pound, 2015). However, one of the biggest advantages of steganography is that it ‘can be combined with [encryption](https://cyberhoot.com/cybrary/encryption/) as an extra step for hiding or protecting data.’ (Mezquita, 2020). Mezquita has written multiple articles and blogs on behalf of CyberHoot, a renowned cybersecurity education platform. His article appears reliable. Once the steganography fails and the message is noticed, the interceptor must still decode it in order to read it (Krenn, 2004). Also, using certain types of encryption can make detecting a message more difficult, as the data appears to be naturally occurring within the carrier and not adjusted (Kumar & Pooja, 2010). This is a major step towards finding the most potent method of protecting data – something governments and secret services are desperate to develop.

Most modern steganography techniques are designed to hide digital data as it is sent across a network. Text, images, video and audio cover files are frequently used because they contain a lot of redundancy (data which provides unnecessary detail) (Morkel, Eloff, & Olivier, 2005). For example, typical images used on the internet could hide more than a megabyte of data (Pound, 2015). Although Mezquita states that almost any type of digital content can be used (Mezquita, 2020). Methods within text include null ciphers – taking the nth letter from each word to reconstruct a hidden message – or altering the whitespace between words in a document (Kipper, 2003).

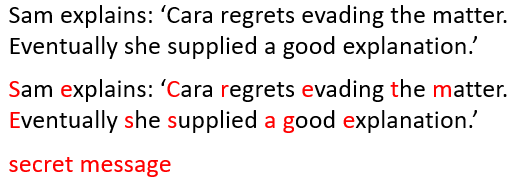
******

Figure 3: An example of a null cipher.

To demonstrate this, I created Figure 3 to present a null cipher, where the first letter of each word is used to hide a phrase. It is difficult to immediately spot the message; but its presence may be indicated by the imperfect flow. Krenn warns that these methods can easily be removed, however, by rewriting the contents. Null ciphers will no longer work as the words have been changed, and differentiations in whitespacing are eliminated by the new layout (Krenn, 2004).

Data can be hidden in audio files by making alterations to channels with a frequency too high for humans to hear, causing the changes to go unheard. (Krenn, 2004). ‘The ‘normal’ hearing frequency range of a healthy young person is about 20 to 20,000Hz’ (Amplifon, 2020). Amplifon, previously National Hearing Care, is a service dedicated to providing hearing care across the globe. Their information is provided by expert audiologists and must be accurate. Uncompressed audio files often contain sounds outside this range, and as we cannot hear it, it is considered redundant. However, a technique called lossy compression, which removes the redundant data of a file to save space, can cause these frequencies to be cut out, meaning information hidden there is also lost. Video files are a combination of images (which are discussed in depth later on) and audio files, so data can be hidden using any of the techniques used in these files (Krenn, 2004). Febryan, Purboyo and Saputra agree, providing examples including discrete cosine transformation, which uses a mathematical formula to modify images, and dynamic cover generation, where a new carrier is created specifically to hide information instead of using an already-existing video (Febryan, Purboyo, & Saputra, 2017). Advantages of using video include the generous capacity for storing data, and the fact that the images and sounds are continually changing, so any noticeable alterations will likely be unobserved (Krenn, 2004). Although one of the most effective forms of steganography in a video file is using code signals and gestures that only the recipient understands, meaning interceptors and computers alike have great difficulty picking up on them.This is also a rare technique because it cannot be removed by compressing or altering the file (Krenn, 2004).

But who exactly uses steganography? As mentioned earlier, the technique is used far less frequently than cryptography, as most people don’t need to hide the fact that they are communicating with others, so long as the actual information is kept private. Unfortunately, a large portion of its usage comes from cybercriminals, who are ‘well aware of the increased focus on encryption and are looking for other ways to make malicious software stay under the radar’ (Cameron, 2018). Mezquita confirms that common illegitimate uses of steganography include ‘transporting malware hidden inside otherwise safe files, or ex-filtrating stolen intellectual property’ (Mezquita, 2020). Cameron describes a technique called traffic-type obfuscation, which hides the existence of packet (digital data) flow. This is useful for criminals to obscure their actions (Cameron, 2018). It may be used to hide pornographic content (Mezquita, 2020), or executable code such as a virus (Cameron, 2018). In 2016, the Criminal Use of Information Hiding (CUIng) initiative was set up to ‘tackle the problem of criminal exploitation of information hiding techniques’ (CUIng.org, 2016). This source is the CUIng initiative’s official website, and so will be the best place to look for their aims. They, along with other security services, work to uncover information hidden by cybercriminals using steganography. CUIng also reports cases where the technique was used by spies, such as the Russian spy ring in 2010, and terrorists, including members of al Qaeda (CUIng.org, 2016).

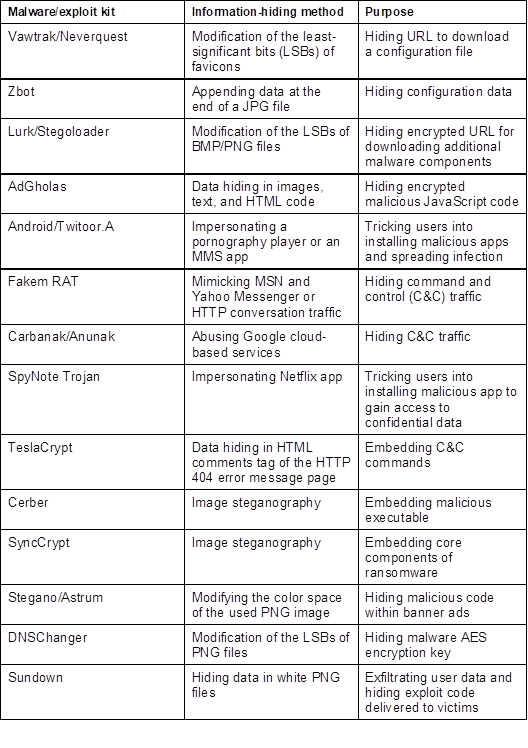


Figure 4: Examples of information-hiding malware (Cameron, 2018)

Cameron provides a table showing 14 real-world examples of steganography techniques used by cybercriminals from 2011 to 2017. As well as making their malware harder to detect, steganography prevents it from being easily traced back to a criminal or organisation (Cameron, 2018). The acronyms JPG, BMP and PNG signify that the cover source used is a digital image – it is therefore obvious that the majority of these examples use them. Notice the examples that mention LSB – it stands for Least Significant Bit, and is a fundamental part of my Python project later on. This source provides evidence that LSB modification is used by real-world cybercriminals also.

Other purposes of steganography include simply storing and hiding information, such as bank details or passwords. In countries where freedom of speech is regulated, it allows people to communicate without censorship or fear of messages being traced back to them(Krenn, 2004). Steganography allows watermarks, trademarks and similar data to be hidden in content files while appearing unchanged. These can identify the source of illegally shared files (Mezquita, 2020), without altering the content itself as visible watermarks do. Stock photo companies may use this technique, repeating a small piece of text over and over to make it difficult to remove (Pound, 2015). In fact, this will likely become a very important avenue for steganography, as new methods of plagiarism arise and content providers become more eager to protect their work against illegal distribution. Having a watermark present will allow creators to track down violators of copyright and take legal action against them (Krenn, 2004). Steganography, it seems, is full of potential – with enough research and development it could become a fundamental part of internet communication.

Steganography in Digital Images

According to Morkel, Eloff and Olivier, the most common form of cover source is the image, due to their widespread use on the internet such as in websites, social media updates and advert banners (Morkel, Eloff, & Olivier, 2005). Another article agrees, claiming that they are ‘the most popular carrier objects’ (Shetye, Vanmali, Fernandes, & Patil, 2016) This article’s authors are part of an Indian engineering college, and include multiple references, however some of their arguments lack justification and seem less dependable. There are two types of digital image: vectors, constructed using mathematical shapes and used in logos, text or simple designs (McMaster, n.d.); and bitmaps, composed of tiny individual squares called pixels which when pieced together create an intricately detailed image. This tutorial website is written by a graphics and web designer, but provides little background information, appearing less reliable.

Figure 5: A bitmap image zoomed in to reveal individual pixels (McMaster, n.d.).

McMaster provides Figure 5, depicting how a pixel is the smallest indivisible part of an image, containing only one colour. To represent a pixel on a computer, we must convert its colour into data that the computer can understand.

Humans are familiar with the decimal (base 10) number system – that means representing numbers using digits, with a different digit for each power of 10 (Brain, 2000). Brain’s article is part of educational website HowStuffWorks, and is likely checked before publication, appearing accurate. For example, 583 has five 100s, eight 10s and three 1s. However, being at its simplest, a machine, a computer represents data using a switch in one of two states: on or off (Heddings, 2018). Heddings is a technical expert and experienced writer for similar educational website HowToGeek, dedicated solely to technology. His work is backed up by diagrams adding to its credibility. We represent this using 1 for on and 0 for off, and this is called binary. Numbers in binary use base 2 instead of 10 (Brain, 2000), meaning each digit represents a power of 2: 1011 has one 8, no 4s, one 2 and one 1. 8+2+1=11 so 1011 is 11 in binary. Each digit of binary is known as a bit, and each bit is worth two times more than the next (Heddings, 2018). We use this system to represent everything on a computer.

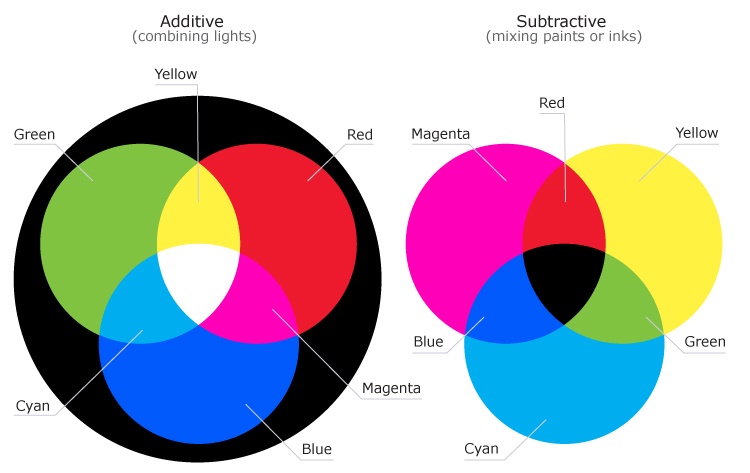
So how do we represent colour? On a computer, colours are made by mixing red, green and blue channels, similarly to how paint colours are made using yellow, magenta and cyan (Science Learning Hub – Pokapū Akoranga Pūtaiao, 2019). This source is from a renowned company, containing diagrams and a video from a professor, making it seem very reliable.

Figure 6: The different primary colours used for computer screens and mixing paints. (Science Learning Hub – Pokapū Akoranga Pūtaiao., 2019).

Science Learning Hub present this diagram, clearly illustrating the differences between colour models used on screens and in paints. Mixing red, green and blue produces white, and black is the absence of these colours. The majority of images on the internet use True color, a specification that uses 24 bits to represent a pixel, with eight for red green and blue each (Techopedia, n.d.). This encyclopaedia article should have been thoroughly researched before publication, however a lack of author and date makes the source seem less dependable. Each channel has a value from 00000000 to 11111111 (0 to 255 in decimal), with higher numbers meaning more of that colour. A white pixel would have all three channels as 255, while a black pixel would have all three channels as 0.

= 01111000

= 11110000

= 10101111

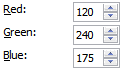


Figure 7: A green pixel and its colour values.

I created Figure 7 to show a green pixel and the values of each colour channel. In binary, this pixel would be stored as 01111000 11110000 10101111. Using True color, over 16.5 million colours can be represented (Techopedia, n.d.) – that is 256 combinations for each of the red, green and blue channels (Frich, 2016). Frich’s source is a collection of articles about colour management, written by a professional photographer. However, there are no signs that he is accessing trustworthy information and I am hesitant to trust facts from this source. Frich claims that humans can distinctly perceive up to 300,000 separate colours, however Techopedia and Science Learning Hub argue that this value should be closer to 10 million (Science Learning Hub – Pokapū Akoranga Pūtaiao, 2019). Either way, a large portion of the possible colours are indistinguishable from one another (Techopedia, n.d.). This is exploited by ‘the most widely known algorithm for image steganography’, Least Significant Bit modification (UKEssays, 2018). While the author of this essay is not named, they are a university student and so likely have access to very reliable information; this is confirmed by the numerous references.

The bit in the last column of a binary number is considered the Least Significant Bit (LSB), because changing it has little effect on the overall total (Kipper, 2003), altering it by just 1.

= 01111001

= 11110001

= 10101110

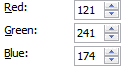


Figure 8: The green pixel with its LSBs changed.

In Figure 8, the LSB of each colour channel has been swapped. The decimal values are very close to the original, and the new pixel appears identical. Such a change is almost imperceptible (Pound, 2015).

= 11111000

= 11110000

= 00101111

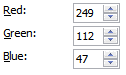


Figure 9: The green pixel with its MSBs changed.

In contrast, Figure 9 shows the same pixel with its most significant bits changed. The decimal values are wildly different and the pixel is now a vibrant orange. Images have a large capacity for LSB modification – in a typical image of 800\*600 pixels, there could be up to 1,440,000 bits of secret information (Krenn, 2004). But in order to hide a message, it needs to be converted to binary.

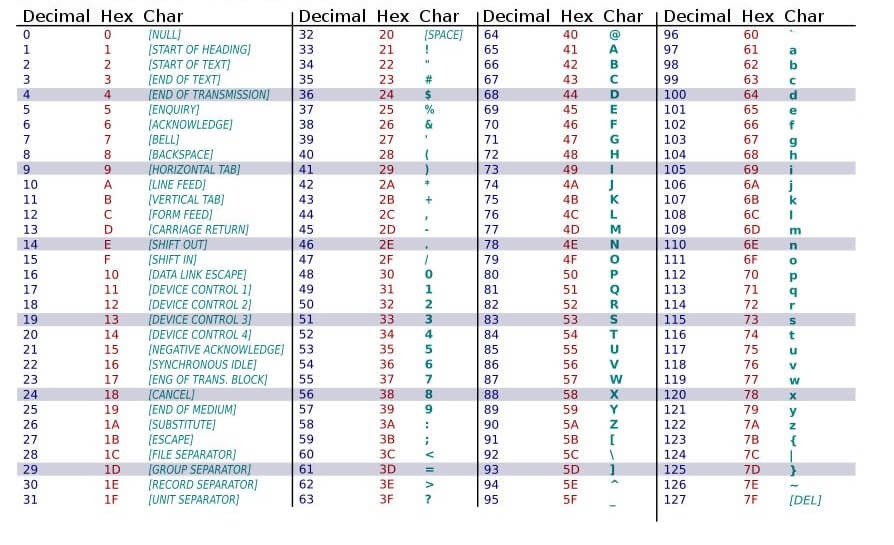
This is done using the American Standard Code for Information Interchange (ASCII), which assigns each character of text a number. For example, ‘A’ corresponds to ‘65’ or ‘01000001’ in binary, and this is universal across devices. Numbers, symbols and control characters such as BACKSPACE also have an ASCII value (BitMerge, 2016). This video is from a small educational channel dedicated to technology. However, there is little information available on its sources or credibility. This information can be found in an ASCII table:

Figure 10: An example of an ASCII table (Programming Electronics Academy, n.d.)

This table shows the first 128 characters’ ASCII codes. Most devices add an extra bit to double the space available (Brain, 2000) – this is called extended ASCII. Now that the message is in binary form, it can be embedded into the image by modifying the LSBs of a specific set of pixels to become the ASCII message. It can then be sent to a recipient, who extracts the LSBs and converts them back into characters, while anyone else who looks at the image will have no idea that there is a secret inside.

While it isn’t the most secure steganography technique, LSB modification is popular because it is fairly simple to implement (Krenn, 2004), and a lot of information can be embedded without visibly altering the stego-image (UKEssays, 2018). However, the technique has many drawbacks, including the fact that, like in audio files, cropping the image or compressing it using lossy compression could cause parts of a secret message to be lost (Morkel, Eloff, & Olivier, 2005). Krenn agrees that ‘the hidden message will not survive this operation and is lost after the transformation’ (Krenn, 2004). This can be prevented by sharing a key with the recipient, which defines a random selection of pixels to be changed from across the image. This also makes it much harder for anyone without the key to discover the message, even if they suspect one has been hidden (Morkel, Eloff, & Olivier, 2005). Pound suggests to never release the source image: an interceptor cannot then compare the two to see that changes have been made (Pound, 2015).

However, this will not protect against statistical analysis, a technique whichcan easily detect modified images by making use of the fact that while LSBs are insignificant, they still contain some data about the image (Krenn, 2004)*.* To illustrate this, I created Figure 11, a simple image of a house, using 4 bits to represent each pixel:

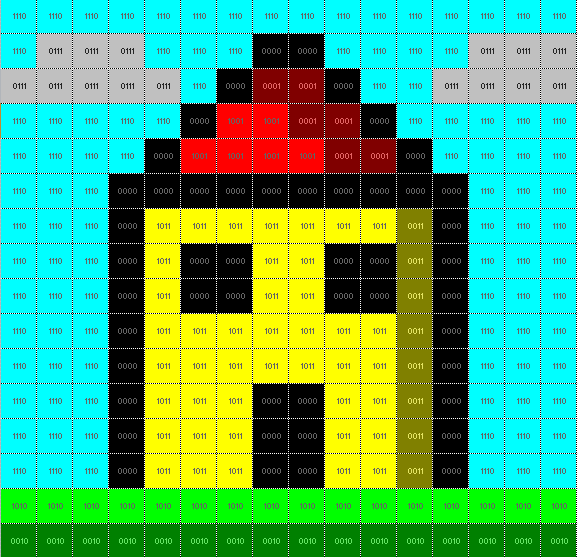


Figure 11: A house represented using 4 bits per pixel.

It is clear that different binary codes represent different colours. 0000 is black, as per convention, 1001 is bright red and 1110 is blue. Now look at Figure 12:

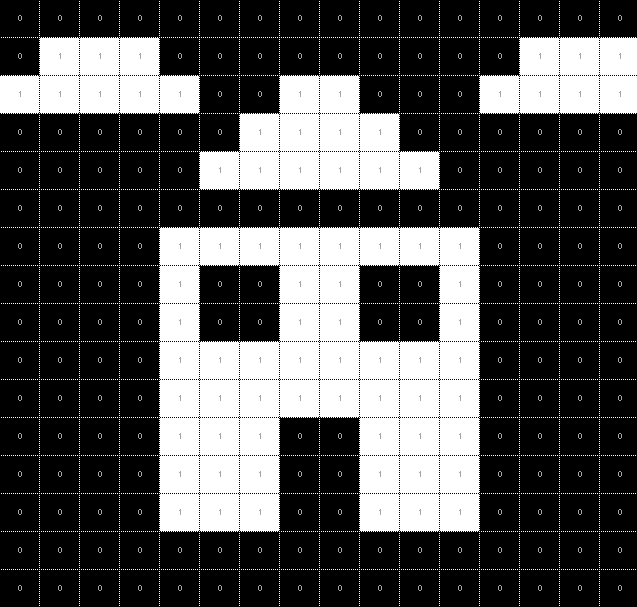
******This image shows the same house recreated using only its LSBs, with 0 representing black and 1 representing white. While a lot of the detail has been lost, the image is still clearly identifiable as a house. However, if the LSBs had been modified, this pattern would be scrambled and unrecognisable, revealing additional noise that wouldn’t be present in the source image (Pound, 2015). The reason LSB modification is effective is that, along with many other steganography techniques, interceptors are unlikely to find the secret without explicitly searching for one.

Figure 12: The same house represented using only its LSBs.

**Methodology**

Planning

The first stage of my artefact was to create a plan for the project and how it should function, without considering the specific syntax (wording and format) of Python, the programming language I would use. I produced the following mind map, highlighting each section of the program:

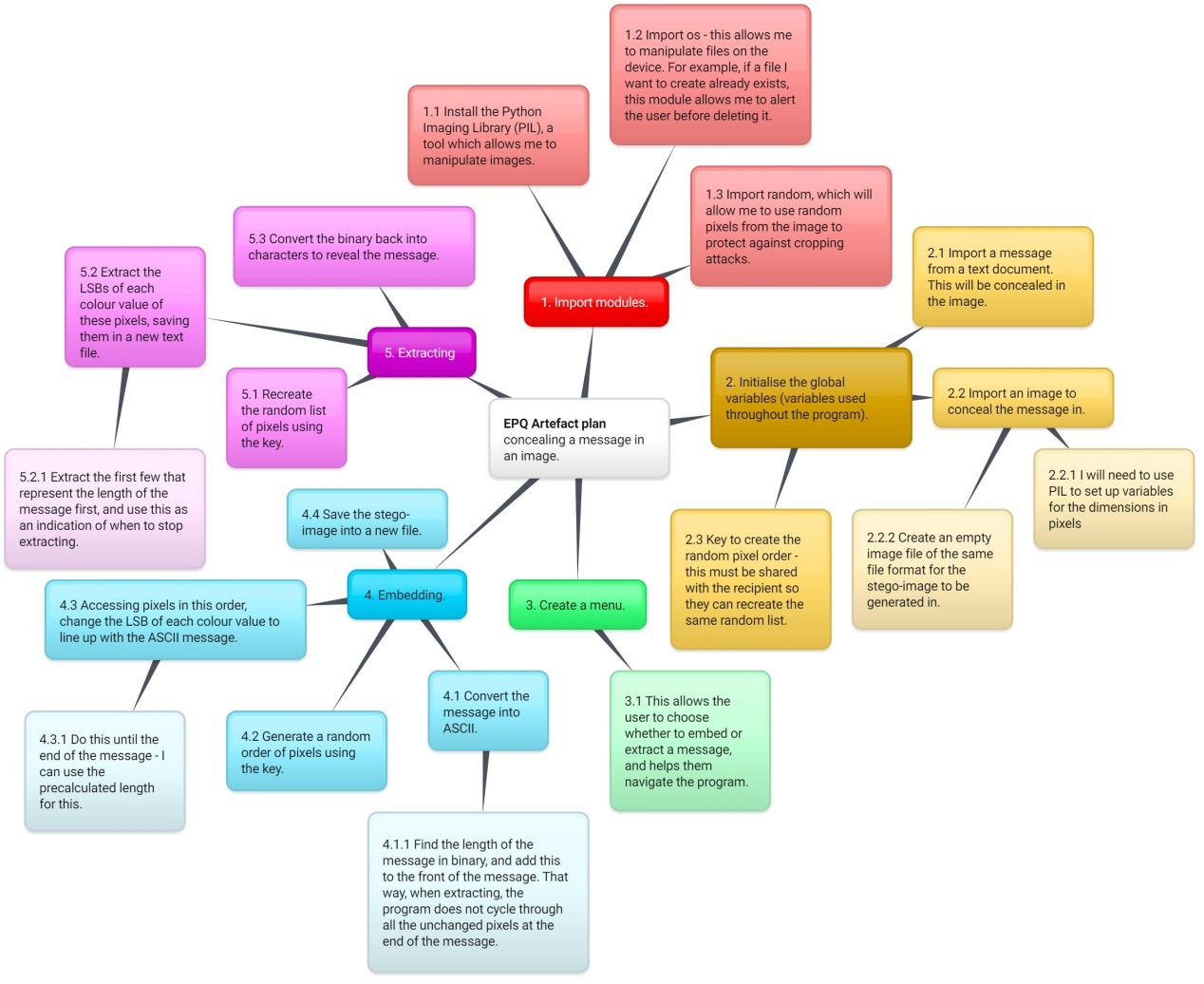
While creating the plan I had to consider which library I should import in order to access image properties (Python alone cannot accomplish this; however, importing a library would give me access to new commands, allowing me to manipulate the image). Having checked multiple websites, I realised that the Python Imaging Library (PIL) was the simplest option, requiring only one installation. This was relatively simple, although it will have to be installed on every device that the program runs on, so I included an explanation of how to do this in the program.

Figure 13: Initial plan of the Python project.

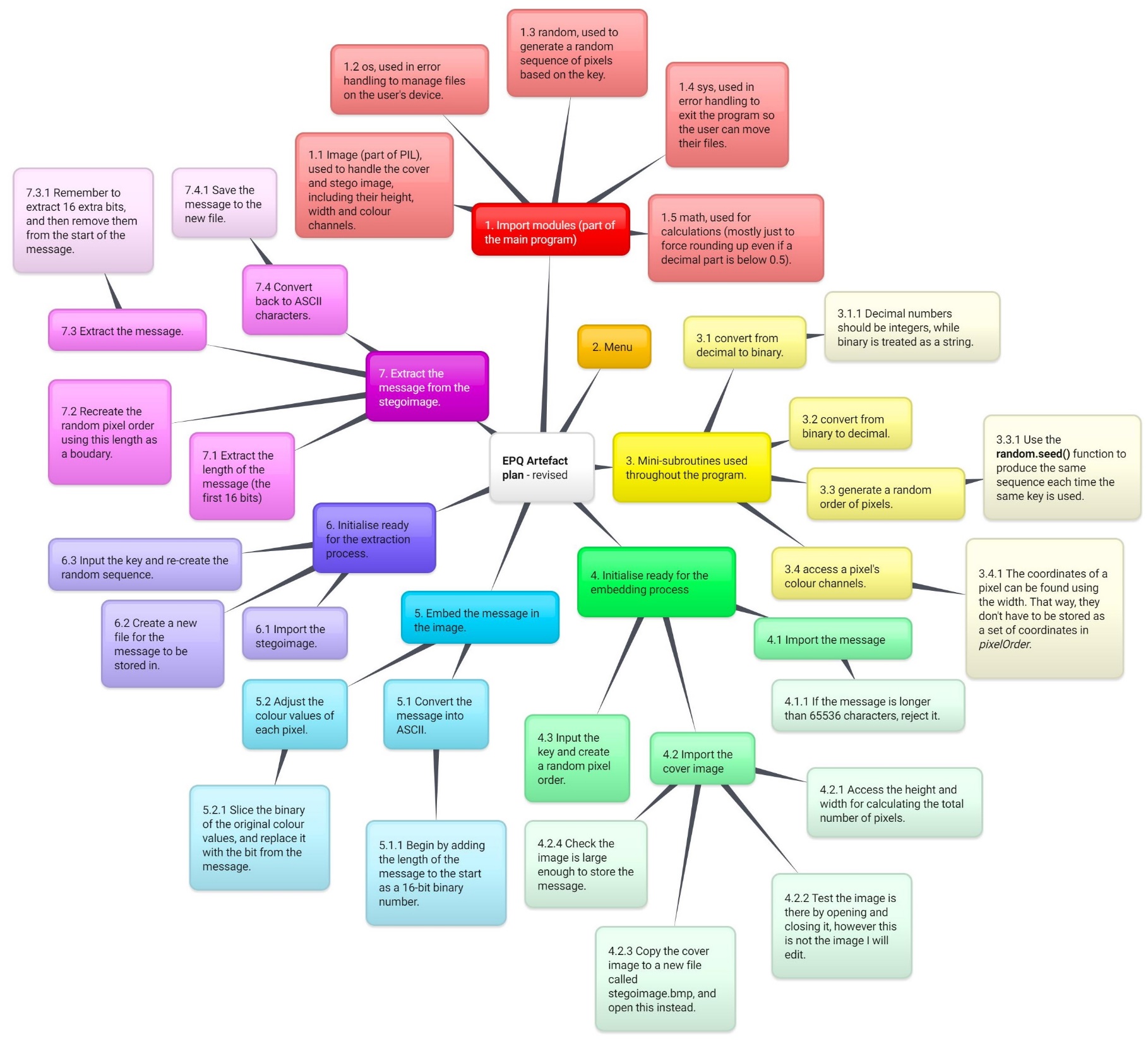
Once the code was finished, I decided to revisit the plan, updating it with regard to changes made in the program, such as splitting the initialise part into two subroutines. This second plan better represents the final flow of the program, but note that I only referred to the original plan during development, and that was sufficient enough to produce a working program.

Figure 14: Revised plan of the Python project.

The Solution

First, some information about Python itself. IDLE is where I created my program, and the shellis the interface it ran on. In IDLE, different colours represent different objects. Black is the default, used for variable names and operations. Orange denotes a keyword, each having a different use. For example, **if** is used to select between two statements. Purple is used for predefined functions such as **print**, which outputs to the shell. In addition, I could define my own functions (referred to as subroutines), and call them throughout the program. Blue appears after the **def** keyword, and indicates the name of the subroutine being defined. Green is a string, a set of characters between quotation marks. Red is a comment, after a hashtag, which have no effect on the actual program and are used to annotate and explain the code (I always included #endifafter an **if** statement, and similar keywords. Even though this isn’t required by Python, I found that it helped to make the structure clearer). In the shell, text produced by the program is blue, and inputs are black.

Throughout the program, a large proportion of the code was used for error handling. This means dealing with incorrect inputs, missing files and other exceptions. It is a crucial part of any program, however not necessarily specific to this project. Therefore, it has been omitted from the following screenshots. (See Appendix A for the final IDLE code including error handling).

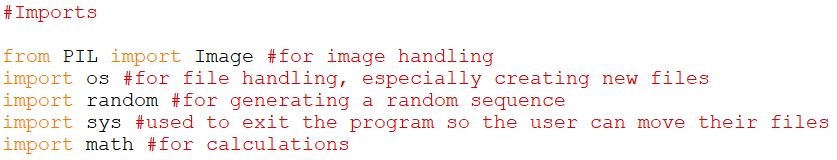
My first step was to import PIL and the other libraries required: **os** to handle files on the user’s device; **random** to generate a sequence of pixels; **sys** to exit the program during error handling; and **math** to make calculations easier. Only PIL required an extra installation – the other four were imported immediately:

Figure 15: Imports (IDLE).

When the user starts the program, they are first greeted with the menu. These are required in almost every project, so it didn’t take long to generate a simple menu with options to embed or extract a message.

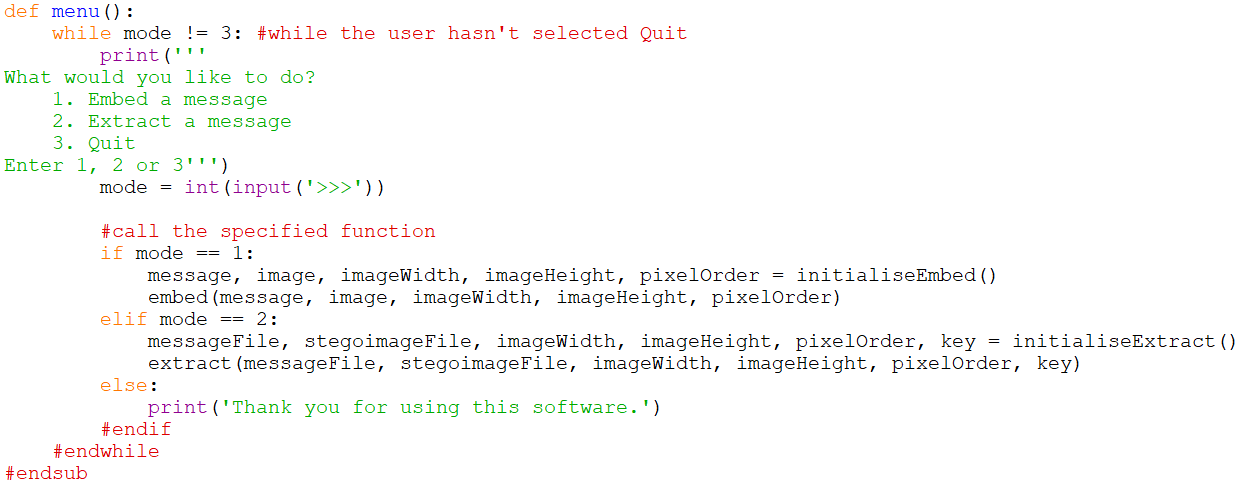
The **while** loop caused the menu to be displayed until the user selected Quit, so each time a process finished, it would be presented again. The user is then directed to the chosen subroutines. A list of variables precedes **initialiseEmbed()** and **initialiseExtract()**, and is placed within the parenthesis of the **embed()** and **extract()** subroutines. This directed the flow of variables: they were created in the initialise subroutines, passed back to the menu, and then into the embed and extract subroutines where they were used. To the user, the menu looked like this:

Figure 16: Menu (IDLE)

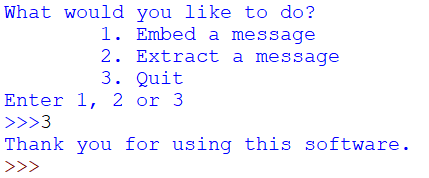


Figure 17: **menu()** (shell).

Occasionally, there would be a certain process that I wanted to use repeatedly in multiple subroutines. I prepared these in separate, smaller subroutines, using different variable names to the ones passed into them. This allowed me to call them at any point in the program, and test them separately from the main subroutines, making it easier to isolate faults in the code. The first two allowed me to convert from decimal to binary and vice versa:

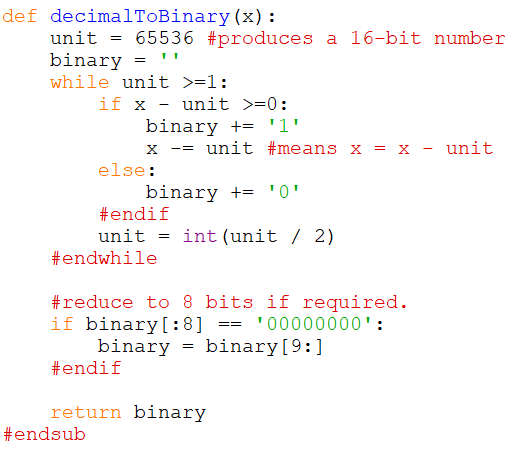
To convert from decimal to binary, I took a decimal unit (starting with 65536 as this created a 16-bit binary number) and checked whether the decimal number was greater or equal to it. If so, I wrote a 1 in the *binary* variable, taking the unit away from the number, and if not I wrote 0. I divided the unit by 2, and repeated this until it was lower than 1. For example, if I had decimal number 43260: 43620–65536 is less than 0, so write 0. 65536/2=32768. 43260–32768 is greater than 0, so write 1 and take 32768 away from 43260, producing 10492. Repeating this process produces the binary equivalent of 43260.

Figure 18: Converting from decimal to binary (IDLE)

I decided to leave *binary* as a string, instead of converting it to a number. This was important because it allowed me to easily replace the LSB later on, and also helped when converting back to decimal. Sometimes when converting a number, I wanted the subroutine to return 16 bits, and other times only 8 bits. In general, it was useful to remove the first 8 bits if they were all 0, and if a specific instance of the subroutine required 16 bits instead, I would change this afterwards. This made the subroutine much more versatile.

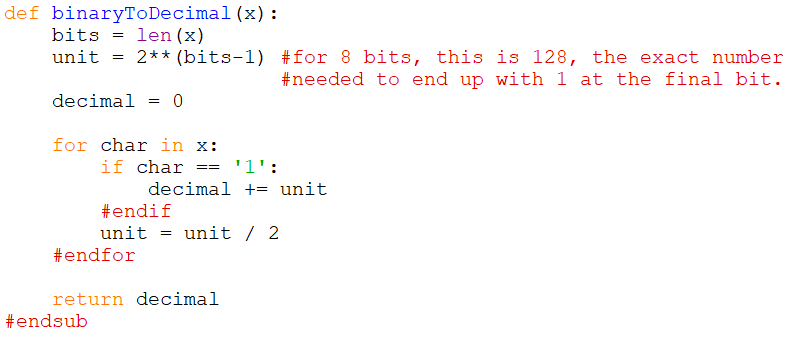
Converting to decimal was almost the reverse of converting to binary. I calculated a unit (this could depend on the length of the binary number, unlike in the previous subroutine). Then I checked each bit in turn; if it was 1 I added *unit* to *decimal,* before dividing *unit* by 2. For example, binary number 110 is calculated to have a starting unit of 4. The first bit is 1 so 4 is added to *decimal*. 4/2=2, and the second bit is also 1 so 2 is added to *decimal*, which now stores 6. 2/2=1, but the last bit is 0, so this 1 is not added and the subroutine returns 6.

Figure 19: Converting from binary to decimal (IDLE)

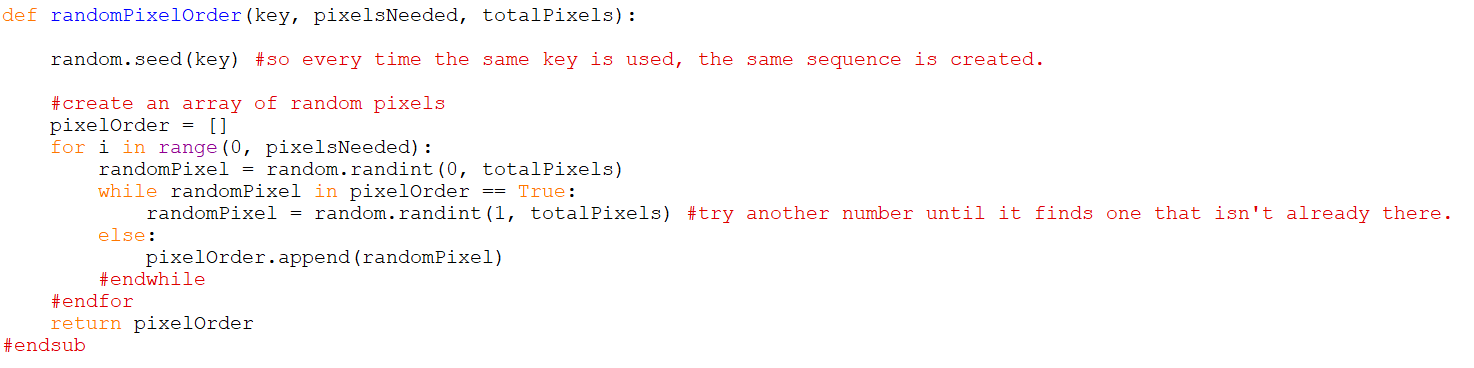
In reality, I only passed 8-bit and 16-bit values through this subroutine. However, as demonstrated it worked with any binary value, adding to its versatility. At the start of this academic year, I was tasked with creating a program to convert 8-bit binary to decimal and vice versa in my Computer Science class, and these two subroutines were inspired by this program.

Figure 20: Shuffling the pixels in a way that can be recreated by a recipient (IDLE)

The next subroutine handled generating a random sequence of pixels. A weakness of LSB modification is that data can be affected if the image is cropped, especially if it is all stored close together. To protect against this, I decided to alter the pixels in a random sequence as discussed earlier, which the recipient could recreate if they knew the key (Morkel, Eloff, & Olivier, 2005). This was possible due to the line ‘**random.seed(***key***)**’, which initialised the random function to always produce the same sequence every time it was used. Different keys produced a completely different sequence, so only someone who knew the original key could recreate it.

I then used the modified function to create a random list of pixels. *pixelOrder* was defined as an empty array (list), then a random pixel number was selected. If this number wasn’t already in *pixelOrder*, I added it to the array. The **for** loop caused this to repeat for however many pixels were required to hide/find the message – otherwise the program would loop over every pixel in the image, wasting time.

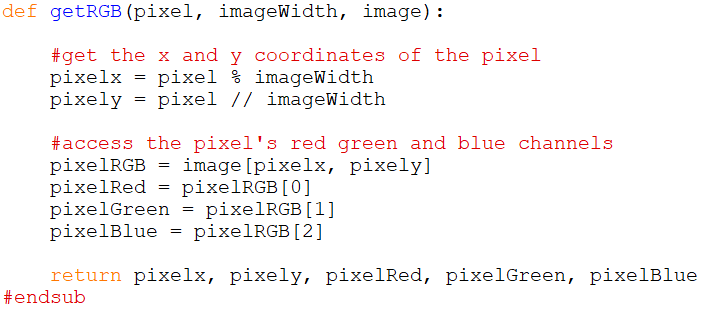
The final mini-subroutine was used to retrieve a pixel’s red, green and blue (RGB) colour values. I used this subroutine many times in quick succession when both embedding and extracting. First, I calculated the coordinates of the given pixel, as the command provided by PIL to access colour values required it to be in coordinate form. In an image, coordinates are measured from the top left corner, as shown in Figure 22:

Figure 21: Accessing the RGB values of a specific pixel (IDLE)

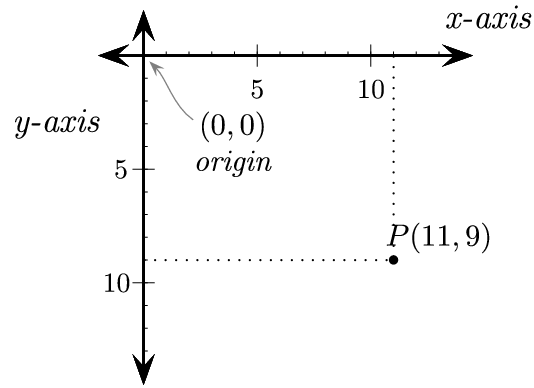


Figure 22: The coordinate system used in images (Craven, 2017)

Conversely to Cartesian coordinates, the y-axis increases going down. Therefore, the first pixel is indexed as 0, with coordinates, and the index number increases by a multiple of the width of the image (in pixels) going down each row. Manipulating this fact allowed me to find the coordinates.

Referring back to Figure 21, the ‘%’ is an operand called ‘modulus’, meaning to find the remainder when dividing *pixel* by *imageWidth*. It provided the x-coordinate. The ‘//’ means whole number division – divide *pixel* by *imageWidth* and discard the remainder. It provided the y-coordinate. The line ‘*pixelRGB = image*[*pixelx, pixely*]’ allowed me to access the colour channels of the pixel, returning the results in a tuple such as (121, 40, 203). The next three lines simply separated these values. (In some images, this returned a fourth value, which represents the alpha channel used in transparency. However, as not all images use it, I chose to ignore it.)

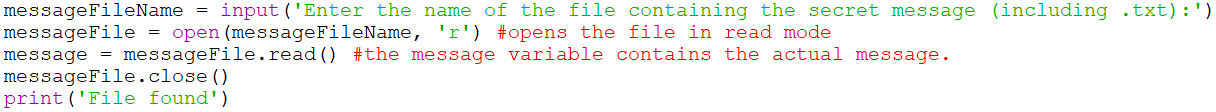
******The first major subroutine was **initialiseEmbed()**, which set up all the variables required for embedding. It began by accessing the message:

Figure 23: Accessing the message (IDLE).

First, the user would input the name of the text file containing the secret message.

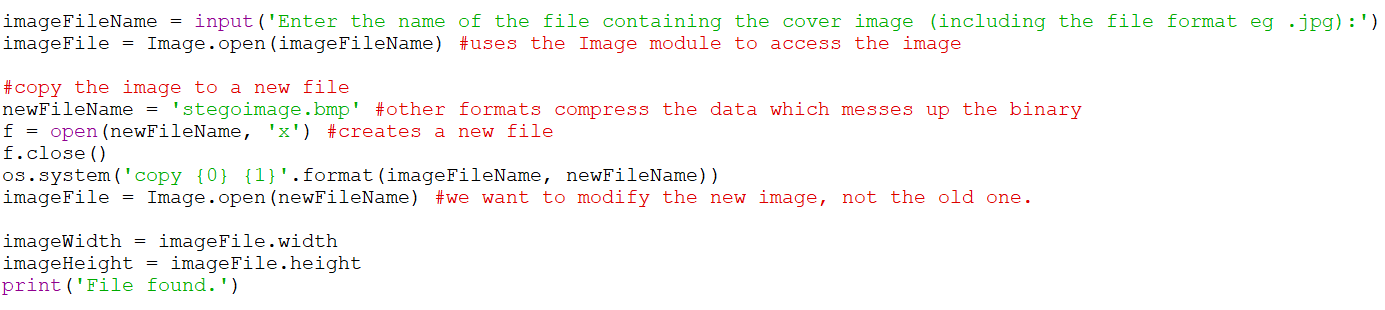
I opened this file and stored it in a new variable, *messageFile*, and its contents (the actual message) in *message*. Next, I imported the cover image in a similar manner:

Figure 24: Accessing the image (IDLE)

In this section, I accessed the image and copied it to a new file called ‘stegoimage.bmp’. It is important to notice the ‘.bmp’ extension: originally, I had planned to retain the original image’s format. I tested the program with jpegs, which use lossy compression. As mentioned earlier, this can destroy parts of a hidden message (Morkel, Eloff, & Olivier, 2005). This is especially true for LSB modification. So when the stegoimage was saved, it was immediately compressed by the jpeg algorithm and the hidden message was lost. I discovered this in my early testing – the aim of steganography is that changes to the source must be imperceptible, which made testing rather difficult. As I didn’t currently have an extraction algorithm, I tested that the pixels were being modified correctly by changing their colour values to a bright pink. When this failed, I realised that there was an issue with compression, so I changed the file format to bitmap (.bmp), which use lossless compression instead, retaining the exact binary data of the image. This made the pixels change colour as I’d hoped. I then opened the new image and extracted its height and width.

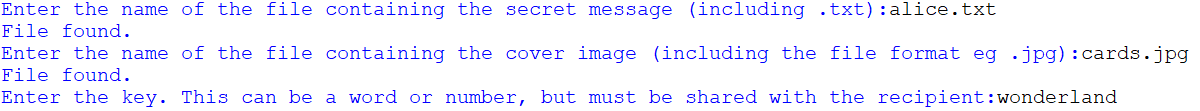
Lastly, the user would input a key, and I used the **randomPixelOrder()** subroutine to generate a list of pixels using it. To the user, **initialiseEmbed()** looked like this:

Figure 25: **initialiseEmbed()** (shell)

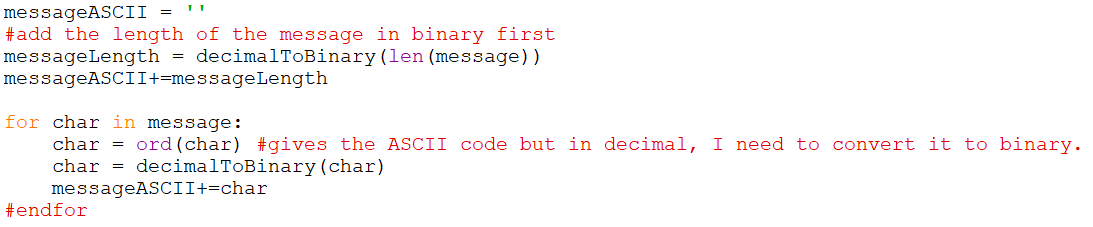
Next was to embed the message into the image. First, I converted the message into ASCII binary:

Figure 26: Converting the message into ASCII binary (IDLE)

I calculated the length of the message and added it to the start. This meant that, when extracting, the program would have an indication of when to stop, so that it wouldn’t waste time extracting data from every pixel. However, when I tested the whole program, I encountered an issue here. I had used a text file with just a couple of characters for the very first testing, and when I tried the code on it there was no problem. So I then tried the larger file documented below, and it produced a string of seemingly random characters instead. By forcing the program to output the binary values extracted, I noticed that the numbers produced were actually very close to the characters of the message – it turned out that I gave the length calculated here one too many zeroes at the beginning if it had more than 8 bits. So the smaller text file was fine, but the larger one was affected. This caused the extraction algorithm to extract one too few digits of the length, messing up the entire rest of the message. This was easy to fix by including the line ‘*messageLength* = *messageLength*[1:]’, but it took me a long time to work out what was wrong. Finally, I converted each character from the message into its ASCII code using the **ord** function, converted this to binary and added it to a variable named *messageASCII*.

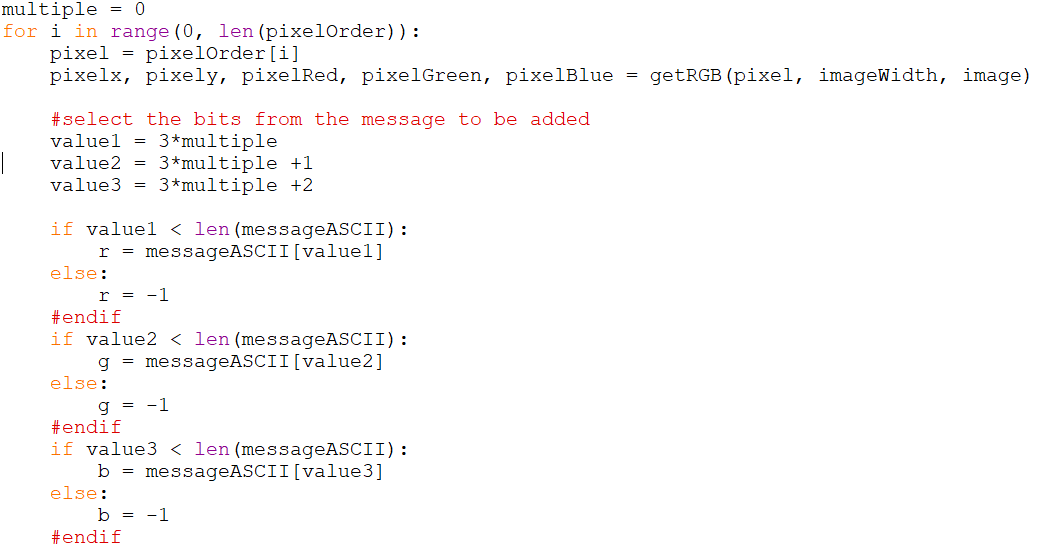
Next was to embed this ASCII into the image:

Figure 27: Taking a single bit for each colour channel from the message (IDLE)

This section was encompassed in a **for** loop, which caused it to iterate for each pixel in *pixelOrder*. First I retrieved the pixel’s coordinates and colour values using the **getRGB()** subroutine. Then I calculated three values, denoting which character would be taken from the message next. *multiple* was initially set to 0, so the first three values were 0, 1, 2, in the next iteration they were 3, 4, 5 and so on. The three **if** statements checked whether these values were smaller than the length of the message (meaning the end of the file hadn’t been reached). If so, they defined a variable *r*, *g* or *b* as the next bit in *messageASCII*, and if not returned -1. This prevented the program from failing if the message ended with the last pixel only changing one or two bits instead of three. *r*, *g* and *b* contained the new LSBs to be added to the image.

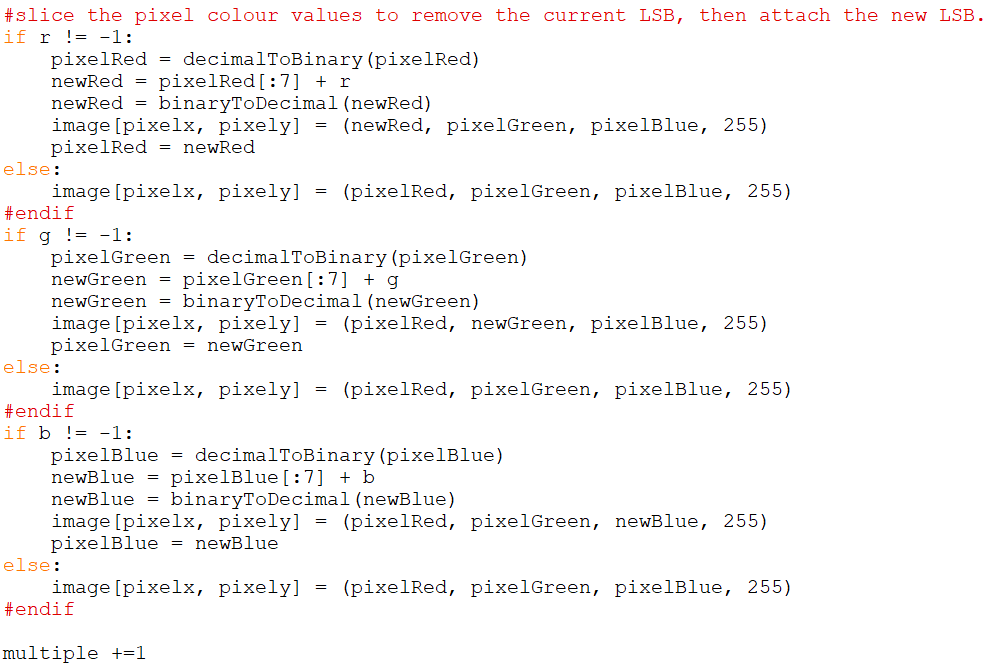
To achieve this, I used these three blocks of code. If *r* contained a character from the message, I converted the collected red value (*pixelRed*) into binary. Because it was a string, I could manipulate it by ‘slicing’ off the LSB using the line ‘*newRed* = *pixelRed*[:7] + *r*’. This essentially removed the LSB in *pixelRed*, replaced it with *r*, and stored it in a variable called *newRed*. I then converted it back to decimal and implemented it back into the image. Finally, I assigned the value of *newRed* to *pixelRed* so that if a change also occurred in the green and blue channels, the change in the red channel was retained. If *r* instead contained -1, the pixel retained its original red value (this line was required for changes to occur). Similar code was used for the green and blue channels. Finally, I increased *multiple*, then saved and closed the image outside the loop.

Figure 28: Replacing the LSBs in the image with bits from the message (IDLE)

In **initialiseExtract()** I set up the variables needed for the extraction process:

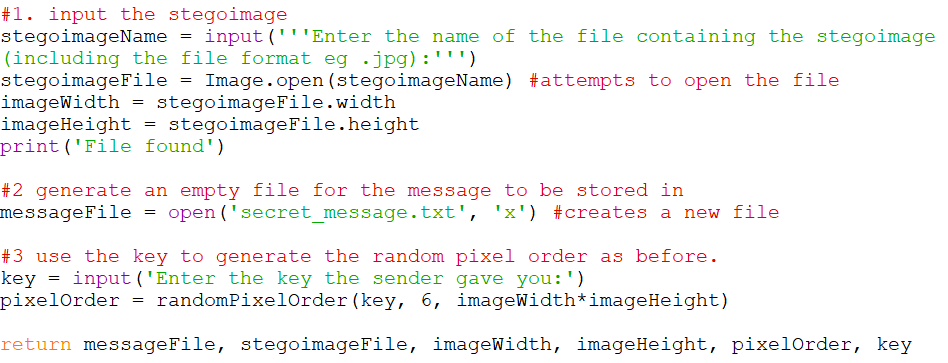


Figure 29: **initialiseExtract()** (IDLE)

First was the stegoimage, imported similarly to in **initialiseEmbed()**. Then I created a new text file on the user’s system for the secret message to be stored in. Finally, I prompted the user for the key, using it to generate the same pixel order as before. Here it only takes six pixels – these contained the length of the message I prepended to it earlier. Once these had been extracted, I could use the length to determine how many pixels to keep extracting.

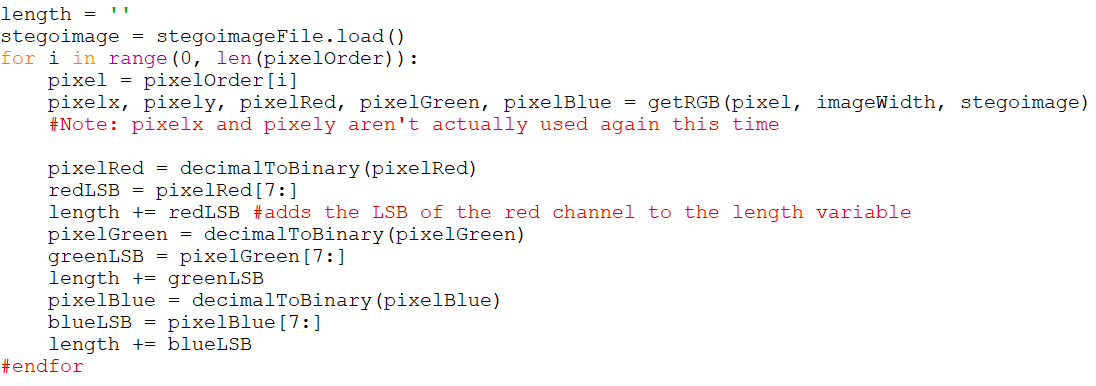
As such, the **extract()** subroutine began by extracting the first six pixels:

Figure 30: Extracting the length of the message (IDLE)

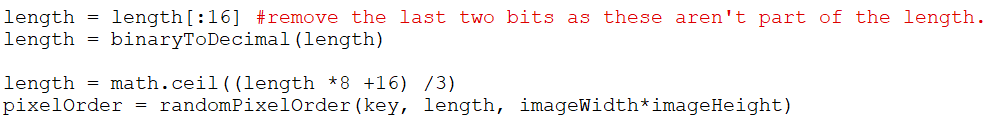
I began by creating a variable *length* to store the length in. I accessed each pixel in the list and retrieved its colour values using the **getRGB()** subroutine. These were converted into binary and then sliced to leave just the LSB, which I attached to *length*.

Figure 31: Preparing the length (IDLE)

This resulted in *length* having 18 bits, so I removed the last two as they were not part of the length. The third line converted *length* into the number of pixels required to reproduce *pixelOrder*: I multiplied by 8 because the message was stored in binary, using 8 bits per character; added 16 to account for the length stored at the start; and divided by 3 because each pixel contained three bits of the message (one in each colour channel). The **math.ceil()** function rounded up if there was a remainder – this caused pixels with one or two bits of the message to still be included.

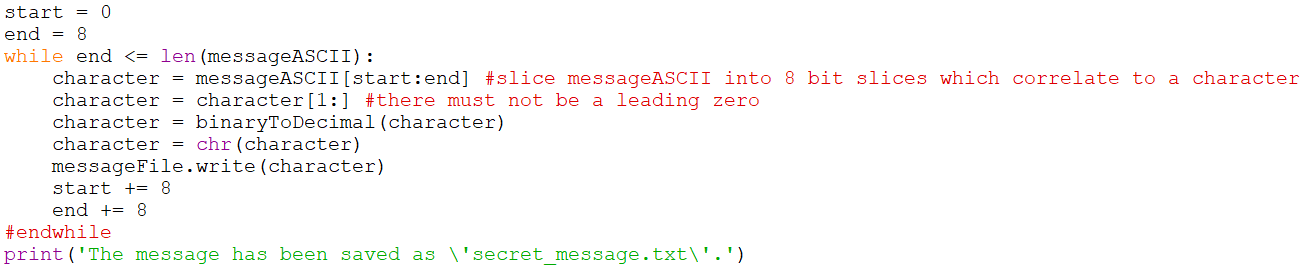
I then extracted the message in exactly the same way, and stored in a variable called *messageASCII*. The first 16 bits, containing the length, were removed, and then I converted the message back into ASCII characters:

Figure 32: Converting back into ASCII characters (IDLE)

I used two variables*, start* and *end,* to slice the message into 8-bit binary values, each containing one character of the message. The **while** loop checked that the end of the file has not been reached, and each time it looped, a character was extracted from *messageASCII.* I converted it to decimal, and then used the **chr** function to convert to its equivalent ASCII character. This was written to the new text file and *start* and *end* were incremented by 8. To the user, the two extracting functions produced this:

Figure 33: **initialiseExtract()** and **extract()** (shell)

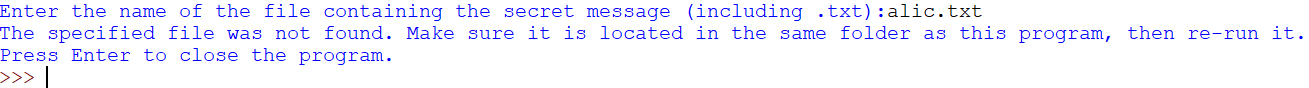
Finally, I would like to mention error handling (see Appendix A). I spent a significant amount of time during and after writing the code testing for errors, using a range of different inputs to try to mess with the outcome. This means that if the user were to accidentally make a mistake, the program would prompt them with how to solve this issue. For example, I entered an invalid filename into the **initialiseEmbed()** subroutine, and was met with this:

Figure 34: An example of error handling (shell)

The program searches for ‘alic.txt’, and, not finding it, returns this message. I tried other invalid inputs such as using numbers out of range in the menu, pressing enter without typing anything, using the wrong file types and more. A suitable error message was provided each time.

**The Result**

I will now showcase the final result of the program. To test it, I used a short excerpt from ‘Alice’s Adventures in Wonderland’ by Lewis Carroll, saved as ‘alice.txt’:

******I hid the message in this jpeg image, simply named ‘cards.jpg’:

Figure 35: The message to be hidden (Carroll, 1865)

******As shown in previous screenshots, I used the key ‘wonderland’. During the embedding half of the program, the file ‘stegoimage.bmp’ was created on my computer:

Figure 36: The image the message was hidden in (Lear, 2014).

**

Figure 37: The stegoimage produced by the program.

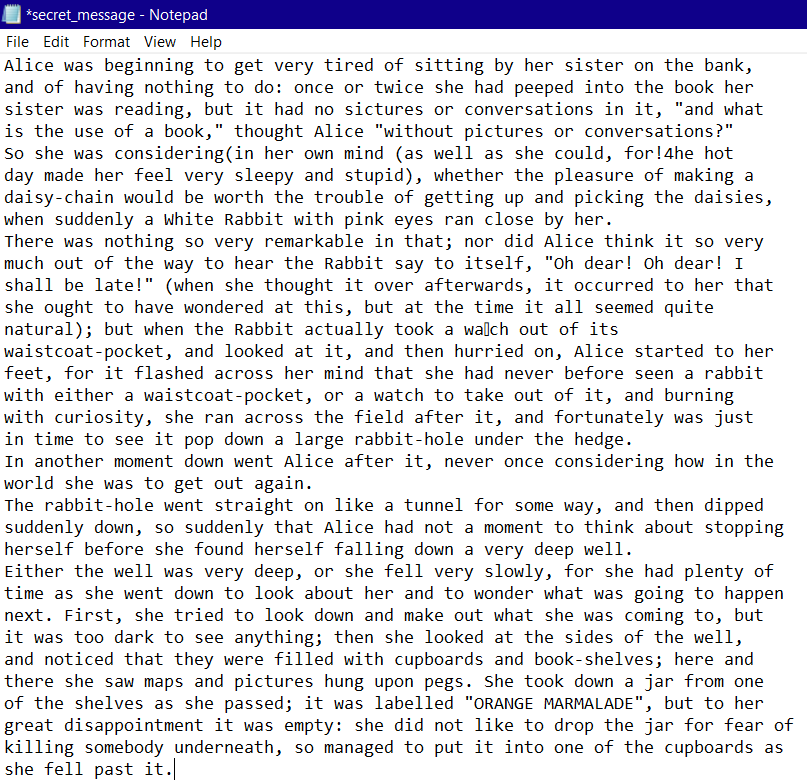
It appears identical to the original, just as I had hoped. It would be impossible to discern that a message is hidden inside. However, the message could be retrieved using the extraction half of the program:

Figure 38: The message retrieved from the stegoimage

While the message can be clearly understood, you will notice a few errors, such as ‘for!4he’ instead of ‘for the’. This brings me to the final issue in the program, and one I see no solution for. Every so often, at random intervals, a character would be extracted incorrectly, with just one bit being erroneous. Most of the message was extracted perfectly, so it appears that a very small proportion of the bits have been misinterpreted. Such a minor change can only be due to issues copying or saving the file. Having researched both methods and found nothing, I used an online tool to test them myself, and learned that the **Image.save()** method was very rarely saving the colour values incorrectly – an imperceptible change. I have been unable to think of a way to counter this, however, while experimenting I noticed that the glitch is less common using a larger cover image. Figure 38 shows the clearest example I could create, while using smaller images resulted in more errors. A much worse case, using an image that was excessively small, looked like this:

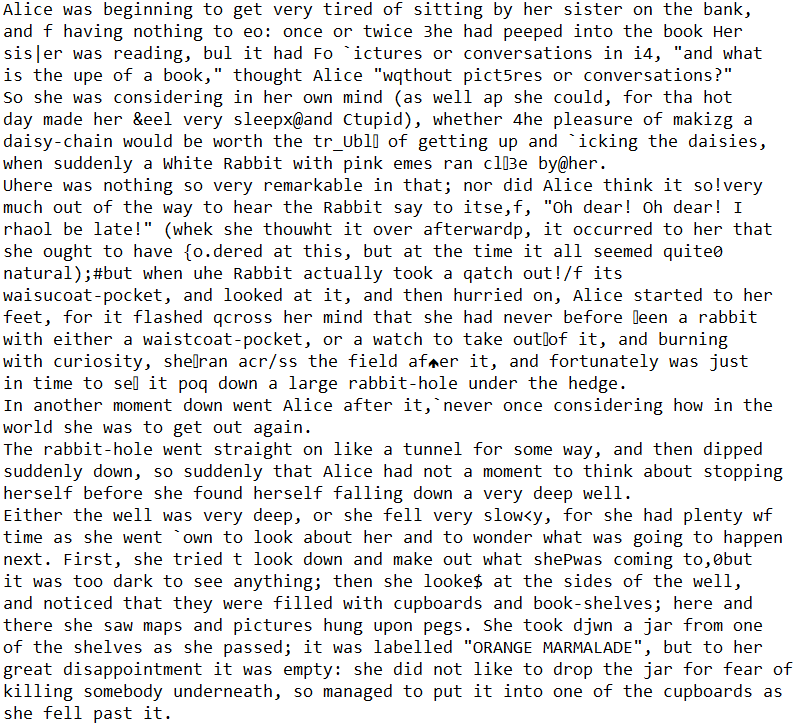
Even in this example the message can still be understood relatively well. Aside from this minor issue, I have successfully hidden a message within a digital image, and retrieved it.

Figure 39: An example showing the prominence of the issue in a smaller image

**Conclusion**

Steganography is the science of concealing the existence of a secret within something much less conspicuous. The technique has changed over time, having become integrated into the digital world as a tool for watermarking or secret communication, used by cybercriminals, security services and others in a variety of cover sources. One such source is the digital image, which can be manipulated using the process of Least Significant Bit modification. Despite the challenges I encountered, I have successfully implemented this technique into a Python program, using an image of a pack of playing cards and an excerpt from Alice in Wonderland. My research explains in further detail how the technique works, as well as its potential weaknesses. If I were to continue this project, I would further discuss methods used to counteract steganography, as well as remove the almost negligible glitch in my final product. Despite this minor error, I believe that I have successfully achieved my aim to hide a message within a digital image, and then retrieve it.

**Table of Figures**

[Figure 1: The relationship between security, robustness and capacity (Febryan, Purboyo, & Saputra, 2017). 2](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656463)

[Figure 2: An example of a microdot (Microdot Technology, 2019). 4](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656464)

[Figure 3: An example of a null cipher. 5](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656465)

[Figure 4: Examples of information-hiding malware (Cameron, 2018) 7](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656466)

[Figure 5: A bitmap image zoomed in to reveal individual pixels (McMaster, n.d.). 8](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656467)

[Figure 6: The different primary colours used for computer screens and mixing paints. (Science Learning Hub – Pokapū Akoranga Pūtaiao., 2019). 9](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656468)

[Figure 7: A green pixel and its colour values. 10](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656469)

[Figure 8: The green pixel with its LSBs changed. 10](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656470)

[Figure 9: The green pixel with its MSBs changed. 10](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656471)

[Figure 10: An example of an ASCII table (Programming Electronics Academy, n.d.) 11](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656472)

[Figure 11: A house represented using 4 bits per pixel. 12](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656473)

[Figure 12: The same house represented using only its LSBs. 13](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656474)

[Figure 13: Initial plan of the Python project. 14](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656475)

[Figure 14: Revised plan of the Python project. 15](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656476)

[Figure 15: Imports (IDLE). 16](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656477)

[Figure 16: Menu (IDLE) 17](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656478)

[Figure 17: **menu()** (shell). 17](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656479)

[Figure 18: Converting from decimal to binary (IDLE) 18](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656480)

[Figure 19: Converting from binary to decimal (IDLE) 19](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656481)

[Figure 20: Shuffling the pixels in a way that can be recreated by a recipient (IDLE) 19](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656482)

[Figure 21: Accessing the RGB values of a specific pixel (IDLE) 20](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656483)

[Figure 22: The coordinate system used in images (Craven, 2017) 20](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656484)

[Figure 23: Accessing the message (IDLE). 21](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656485)

[Figure 24: Accessing the image (IDLE) 21](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656486)

[Figure 25: **initialiseEmbed()** (shell) 22](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656487)

[Figure 26: Converting the message into ASCII binary (IDLE) 22](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656488)

[Figure 27: Taking a single bit for each colour channel from the message (IDLE) 23](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656489)

[Figure 28: Replacing the LSBs in the image with bits from the message (IDLE) 24](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656490)

[Figure 29: **initialiseExtract()** (IDLE) 25](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656491)

[Figure 30: Extracting the length of the message (IDLE) 25](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656492)

[Figure 31: Preparing the length (IDLE) 25](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656493)

[Figure 32: Converting back into ASCII characters (IDLE) 26](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656494)

[Figure 33: **initialiseExtract()** and **extract()** (shell) 26](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656495)

[Figure 34: An example of error handling (shell) 26](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656496)

[Figure 35: The message to be hidden (Carroll, 1865) 27](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656497)

[Figure 36: The image the message was hidden in (Lear, 2014). 28](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656498)

[Figure 37: The stegoimage produced by the program. 28](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656499)

[Figure 38: The message retrieved from the stegoimage 29](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656500)

[Figure 39: An example showing the prominence of the issue in a smaller image 30](file:///C:\Users\emfor\Documents\Homework\Year%2012%20-%2013\EPQ\Essay%20draft%20and%20planning\EPQ%20Essay%201.docx#_Toc64656501)

**Bibliography**

Amplifon. (2020, May 18). *The Human Hearing Range.* Retrieved from Amplifon Hearing Care Professionals website: https://www.amplifon.com/au/blog/human-hearing-range

BitMerge. (2016, August 8). *What is ASCII? [Video].* Retrieved from YouTube: https://www.youtube.com/watch?v=zB85kTs-sEw

Brain, M. (2000, April 1). *How Bits and Bytes Work.* Retrieved from How Stuff Works: https://computer.howstuffworks.com/bytes.htm

Budiansky, S. (2011). *Sir Francis Walsingham.* Encyclopaedia Britannica.

Cameron, L. (2018, November 15). *Hackers' latest weapon: Steganography*. Retrieved from Computer Publications - How Steganography Works: https://publications.computer.org/computer-magazine/2018/11/15/how-steganography-works/

Carroll, L. (1865). *Alice's Adventures in Wonderland.* Macmillan.

Cheddad, A., Condell, J., Curran, K., & McKevitt, P. (2010). Digital image steganography: Survey and analysis of current methods. *Signal Processing*, Abstract.

Craven, P. (2017). *Chapter 5: Introduction to Graphics.* Retrieved from Program Arcade Games With Python And Pygame: http://programarcadegames.com/index.php?chapter=introduction\_to\_graphics

CUIng.org. (2016). *CUIng.org Main Page*. Retrieved from CUIng Web site: https://cuing.org/index.html

Epner, M. (2019, April 23). *Encryption and public keys | Internet 101 | Computer Science | Khan Academy [Video].* Retrieved from Youtube: https://www.youtube.com/watch?v=6-JjHa-qLPk

Febryan, A., Purboyo, T. W., & Saputra, R. E. (2017). Steganography Methods on Text, Audio, Image and Video: A Survey. *International Journal of Applied Engineering Research [Volume 12, Number 21]*, 10485-10490.

Frich, A. (2016, April 14). *Colours and Computer Science.* Retrieved from Colour Management Guide: https://www.color-management-guide.com/computer-science-and-colors.html

Heddings, A. (2018, October 1). *What is Binary, and Why Do Computers Use It?* Retrieved from How To Geek: https://www.howtogeek.com/367621/what-is-binary-and-why-do-computers-use-it/

Herodotus. (430 BC). *The Histories.* Oxford University Press.

Kahn, D. (1996). *The history of steganography.* New York: Springer.

Kipper, G. (2003). *Investigators Guide to Steganography.* Auerbach Publications.

Krenn, J. (2004, January). *Steganography and Steganalysis.* Retrieved from retawprojects: http://www.retawprojects.com/uploads/steganalysis.pdf

Kumar, A., & Pooja, K. (2010). Steganography - A data hiding technique. *International Journal of Computer Applications* , 19 - 23.

Lear, N. (2014, December 23). *The magic of Christmas.* Retrieved from Nukelear Fishing [Blog]: https://nukelear.me.uk/category/playing-cards/

McMaster, W. (n.d.). *Creator Impact*. Retrieved from Vector vs Bitmap Images: https://creatorimpact.com/vector-vs-bitmap-images/

Mezquita, T. (2020, March 30). *CyberHoot.* Retrieved from Cybrary - Steganography: https://cyberhoot.com/cybrary/steganography/

Microdot Technology. (2019, July 30). *Microdot Technology*. Retrieved from Valjiram & Ravi IAS Study Center: https://vajiramias.com/current-affairs/microdot-technology/5d3fe2d21d5def4ee848bfe6/

Morkel, T., Eloff, J., & Olivier, M. (2005). An overview of image steganography. *Proceedings of the ISSA 2005 New Knowledge Today Conference* (pp. 1-11). Pretoria, South Africa: ISSA, Pretoria, South Africa. Retrieved from https://www.researchgate.net/publication/220803240\_An\_overview\_of\_image\_steganography

Network Encyclopaedia. (n.d.). *Cryptography*. Retrieved from Network Encyclopaedia: https://networkencyclopedia.com/cryptography/

Pound, M. (2015, August 4). *Secrets Hidden in Images (Steganography) - Computerphile [Video].* Retrieved from Youtube: https://youtu.be/TWEXCYQKyDc

Programming Electronics Academy. (n.d.). *Arduino issue: LED flickers (on then off) when using switchcase and serial monitor input via the keyboard.* Retrieved from Programming Electronics Academy: https://www.programmingelectronics.com/arduino-issue-switchcase/

Reich, H. (2019, May 1). *How Quantum Computers Break Encryption | Shor's Algorithm Explained [MinutePhysics video].* Retrieved from Youtube: https://www.youtube.com/watch?v=lvTqbM5Dq4Q

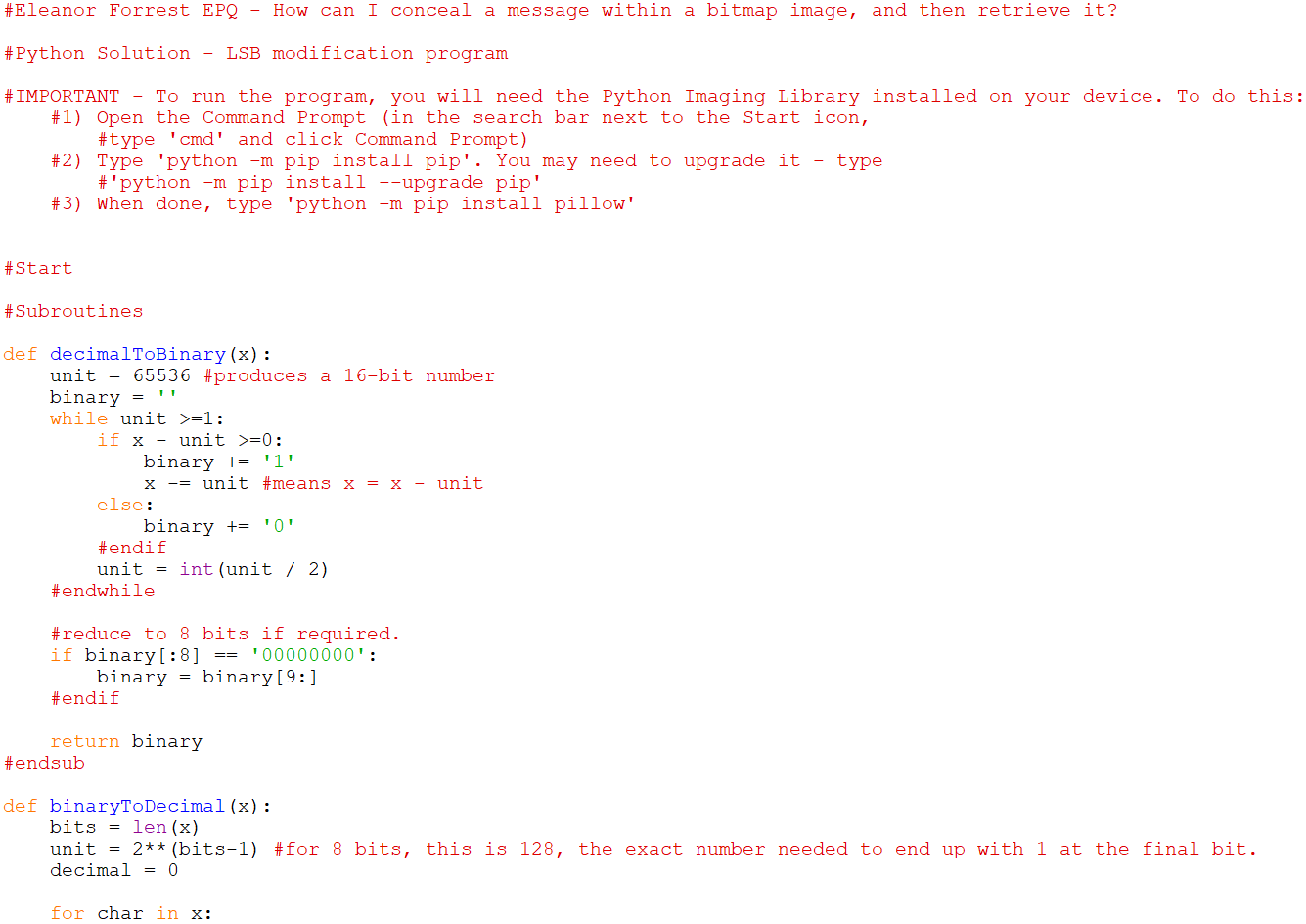
Science Learning Hub – Pokapū Akoranga Pūtaiao. (2019, April 24). *Colours of Light.* Retrieved from Science Learning Hub – Pokapū Akoranga Pūtaiao: https://www.sciencelearn.org.nz/resources/47-colours-of-light

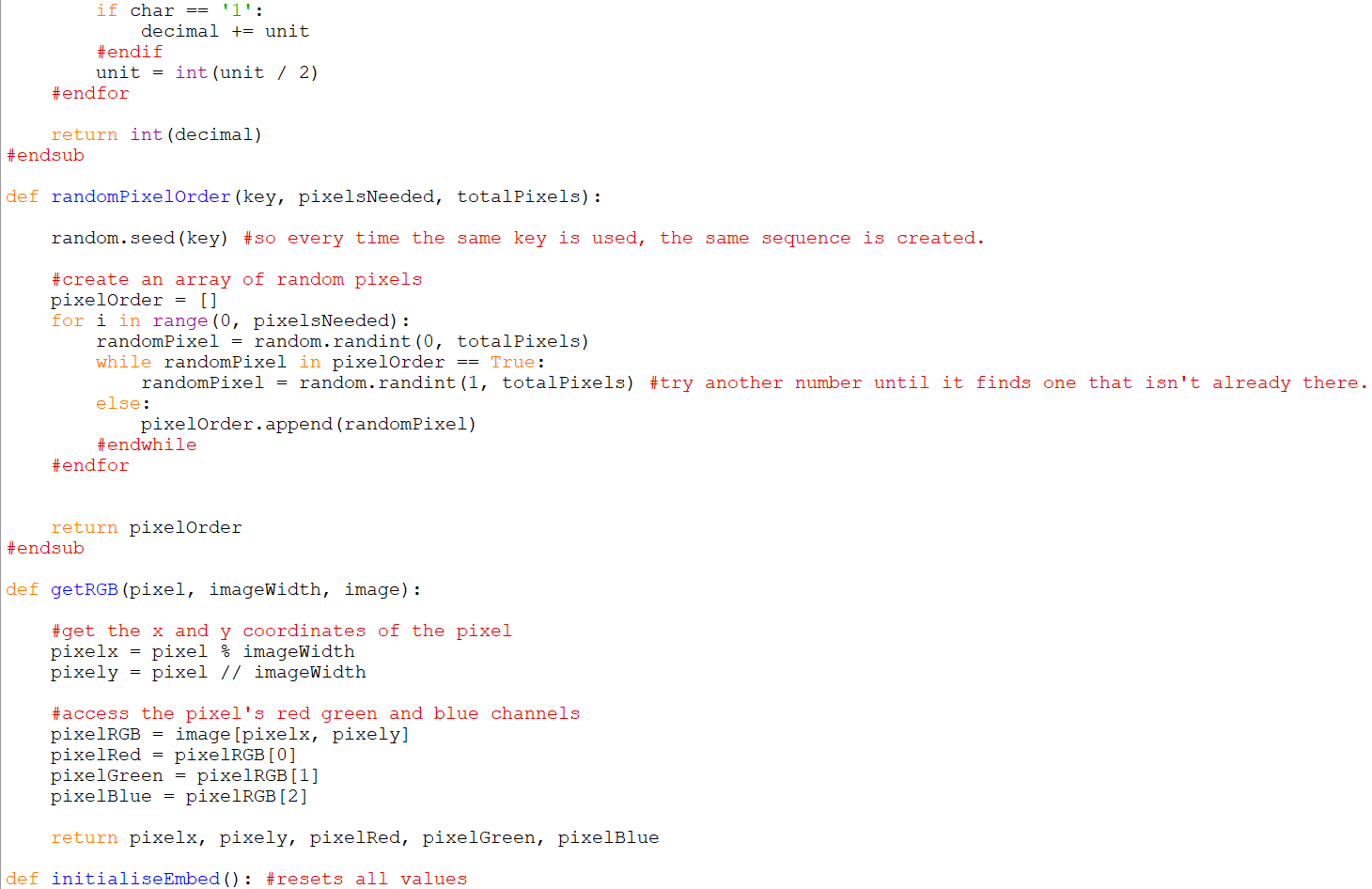
Shetye, O., Vanmali, C., Fernandes, M., & Patil, P. (2016). Survey on Different Techniques of Image Steganography. *International Journal of Computer Applications Volume 138*, 36-38.

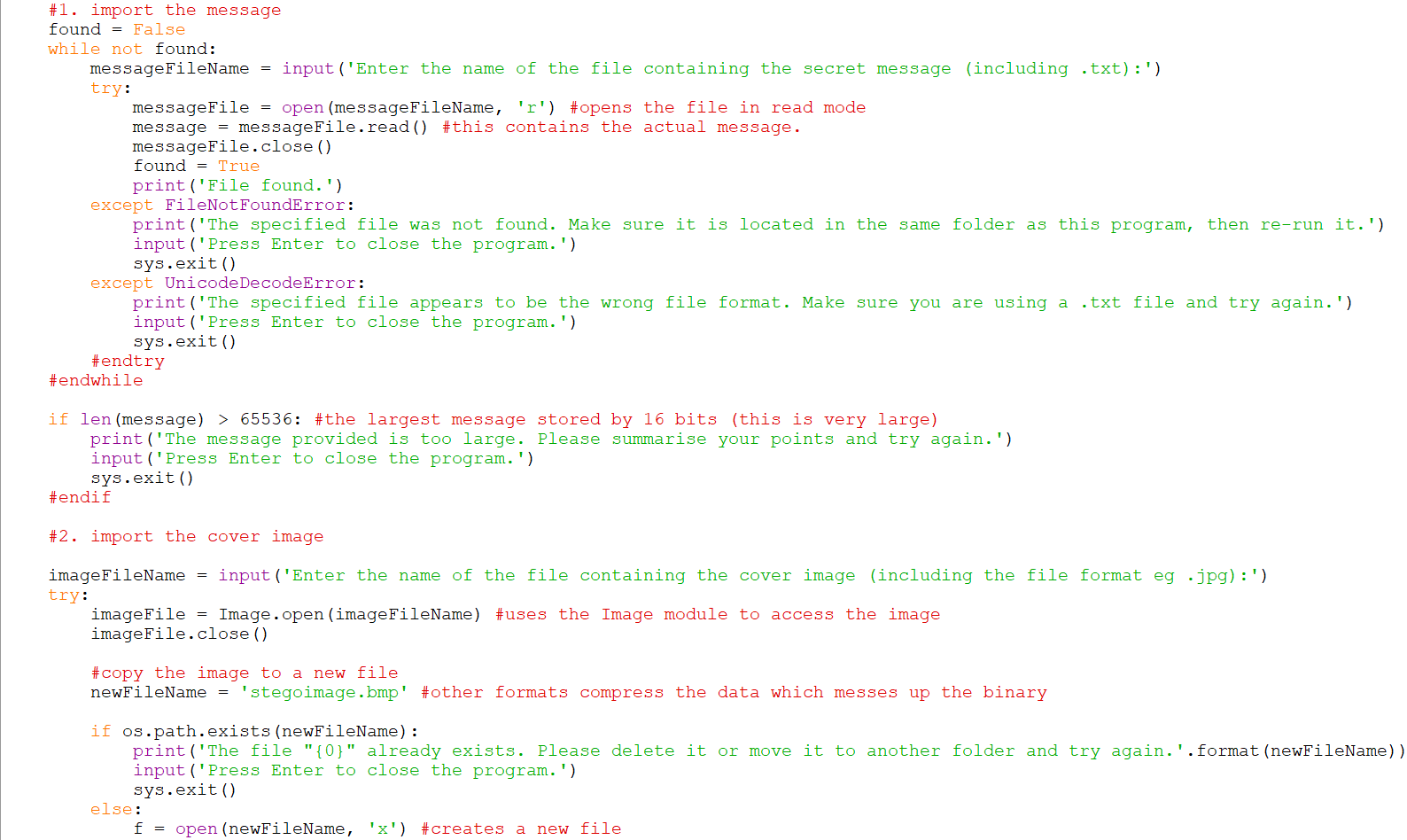
Techopedia. (n.d.). *True Color.* Retrieved from Techopedia: https://www.techopedia.com/definition/496/true-color

UKEssays. (2018, November). *A Review of Image Steganography Techniques.* Retrieved from UKEssays: https://www.ukessays.com/essays/sciences/a-review-of-image-steganography-techniques.php?vref=1

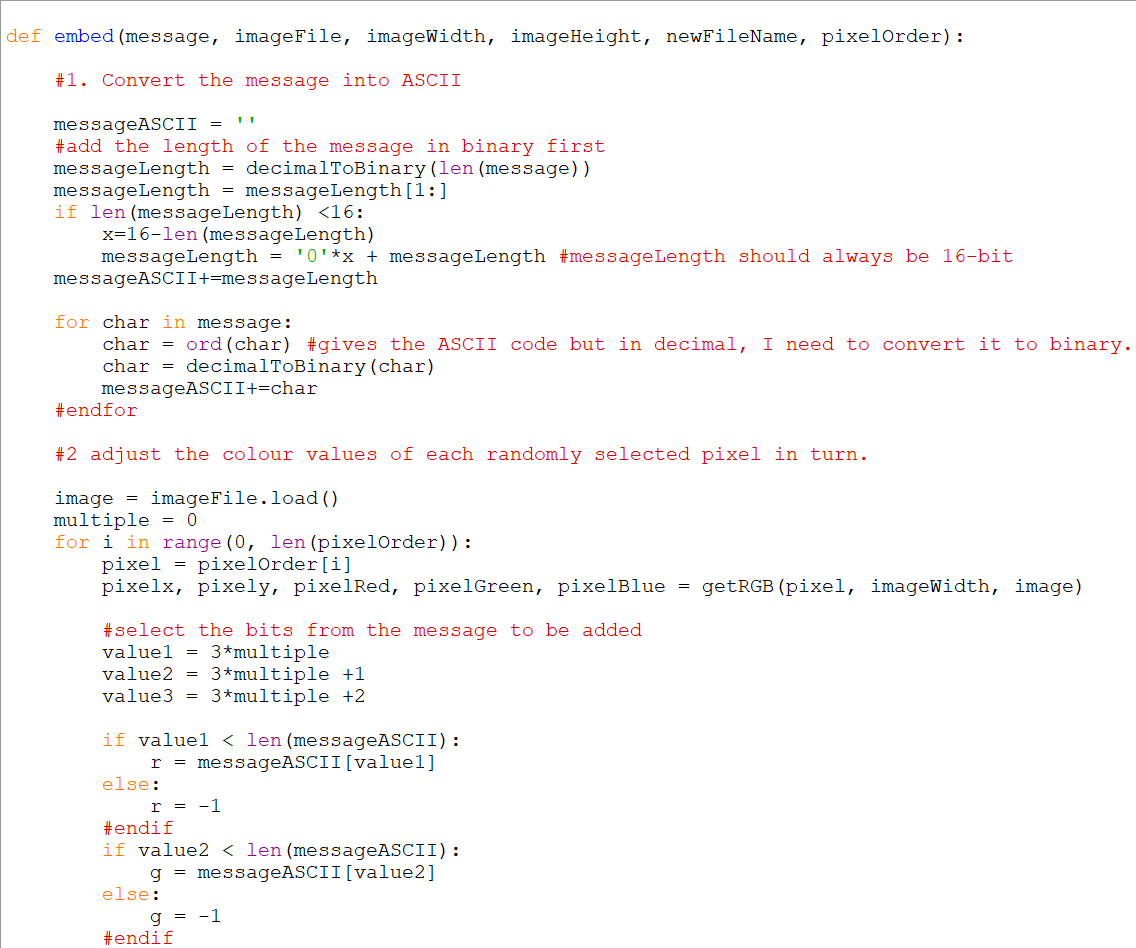
**Appendix**

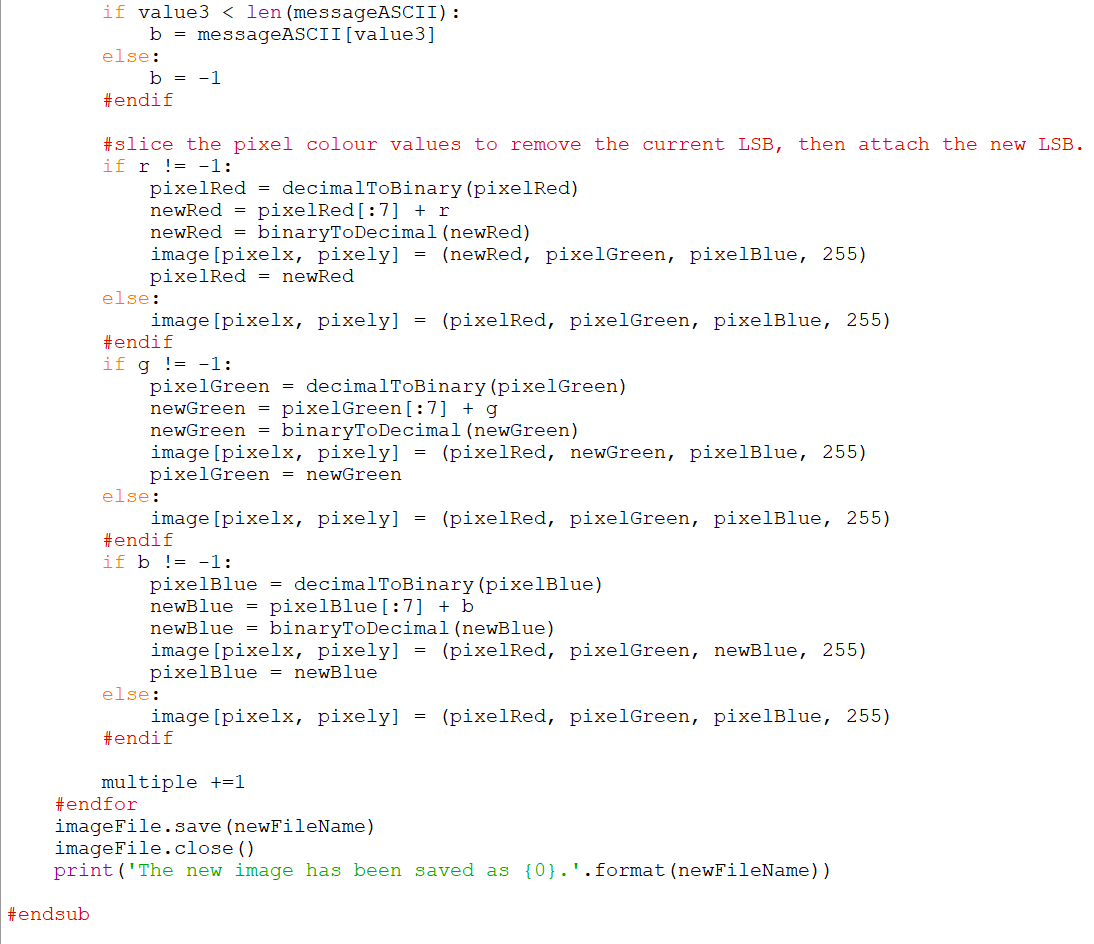
*Appendix A*. Evidence of the final program, including error handling:

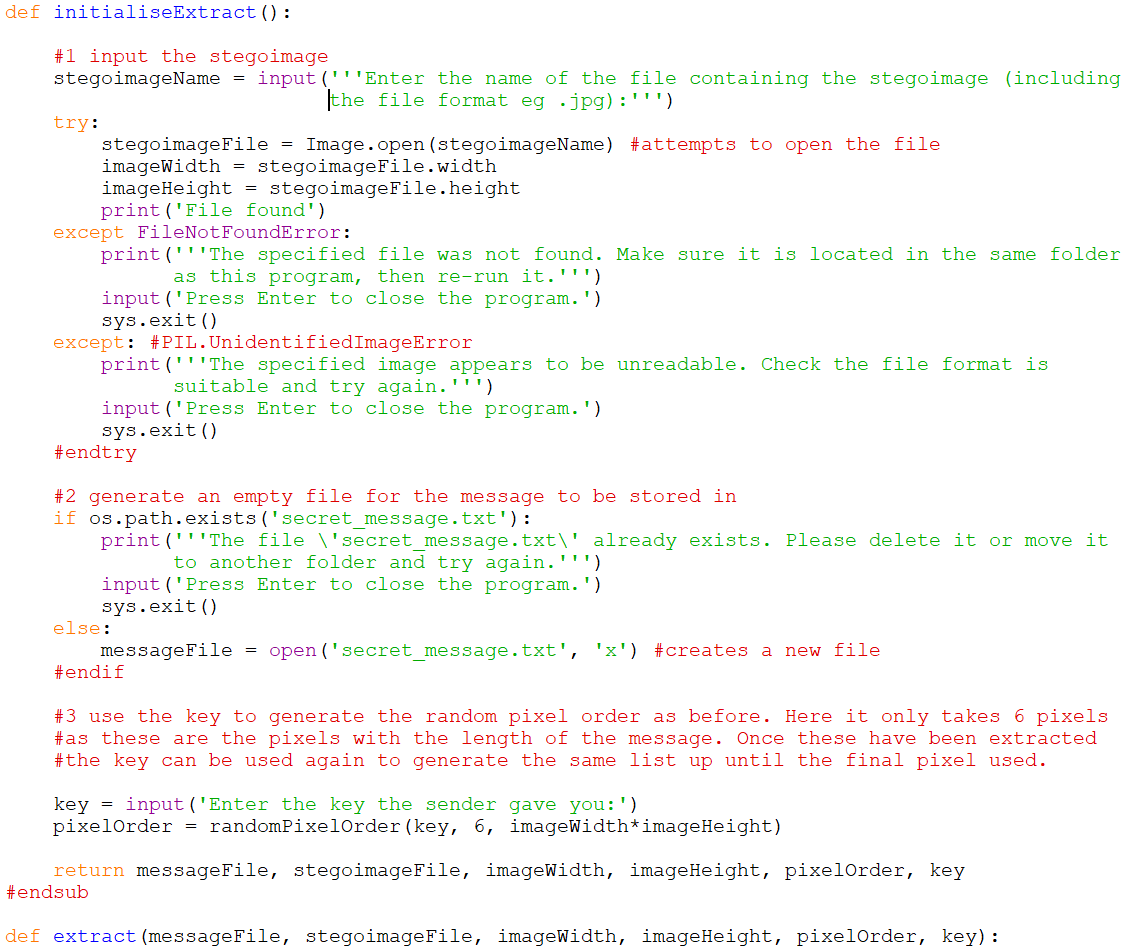
******

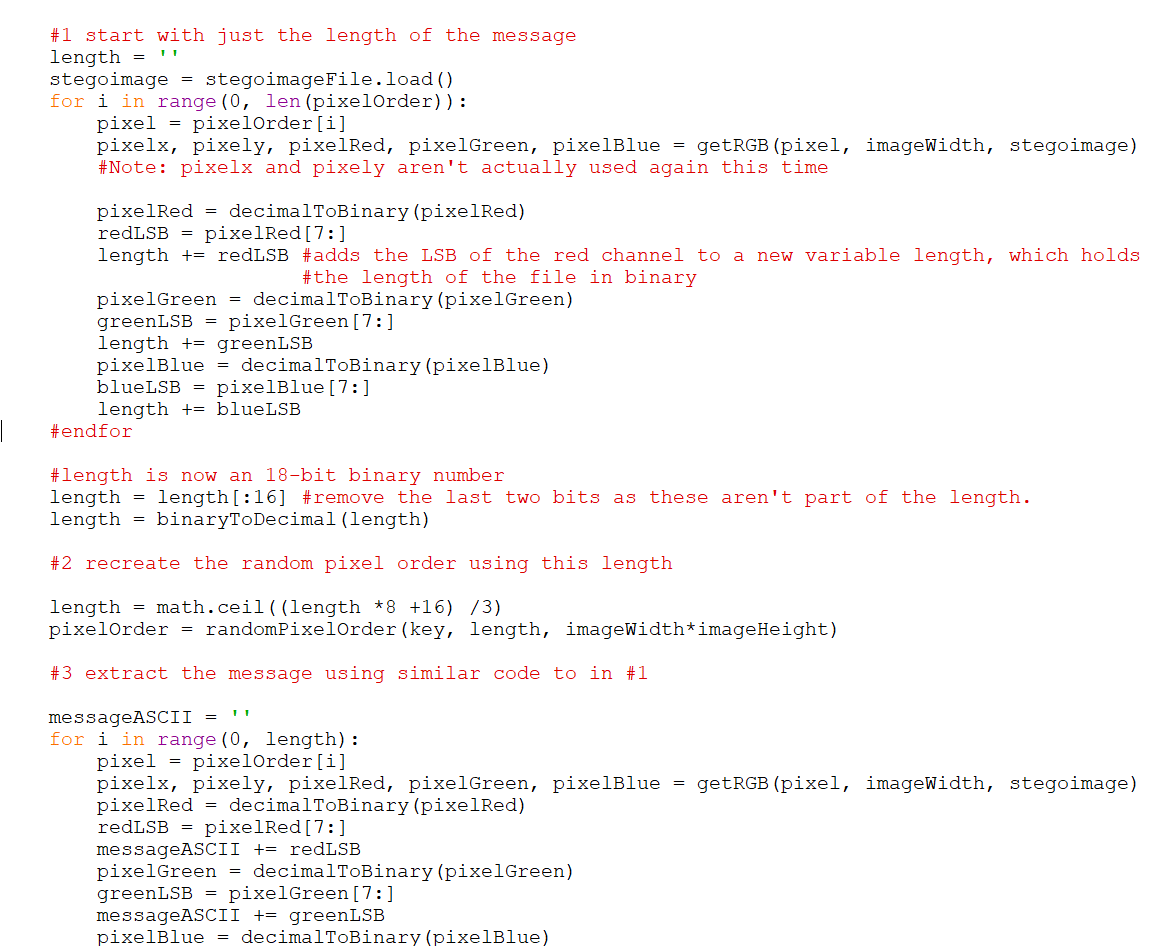
******

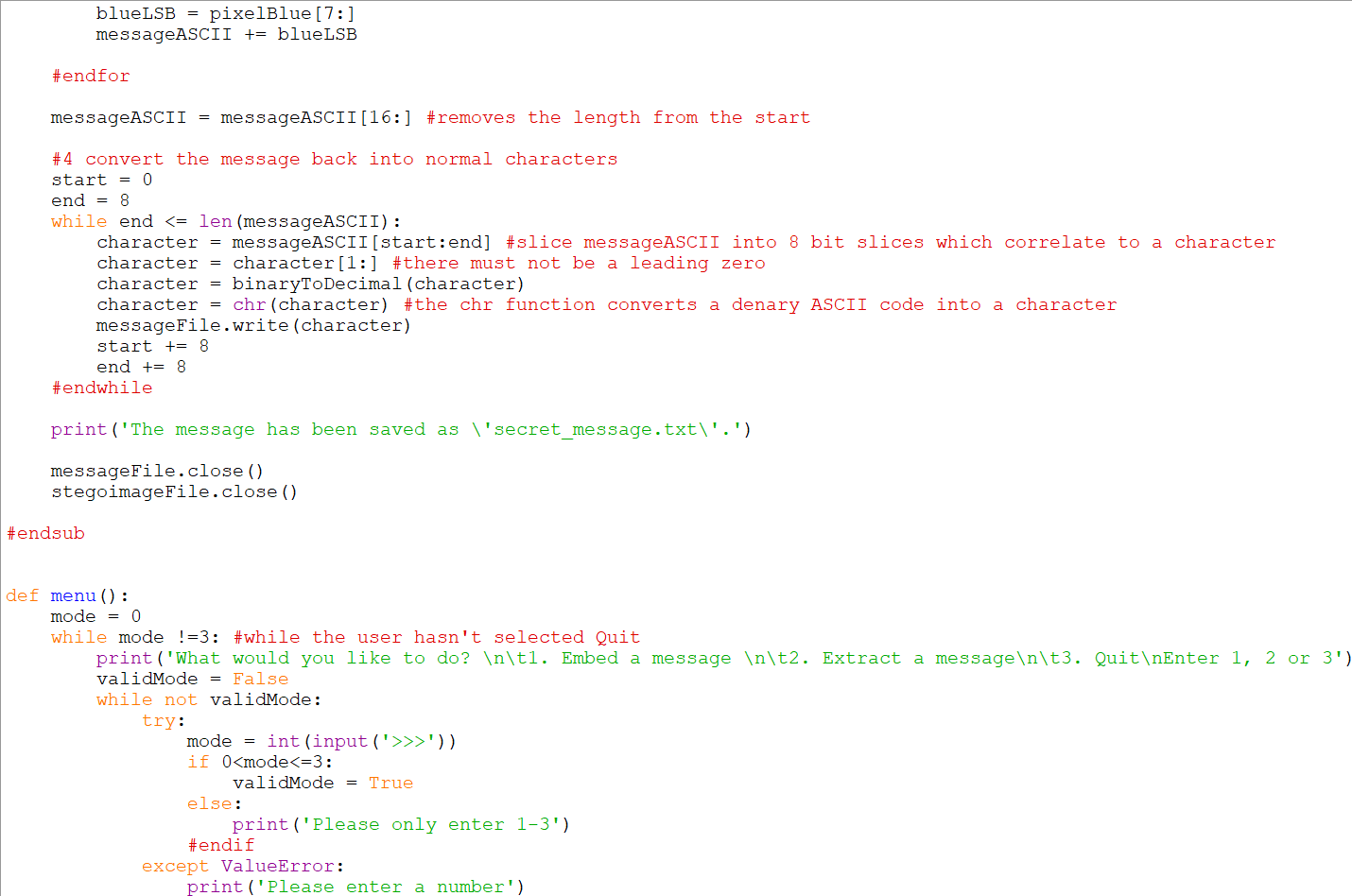
******

******

******

******

******

******

******