Introduction to Steganography – Worksheet

Task

Create an image-based stegosystem, such that a secret message can be embedded into an image, passed between recipients without arousing suspicion, and successfully extracted.

The system should include two separate scripts, one for embedding and one for extraction. This worksheet will guide you through the creation of these scripts in Python on replit.com.

Setup

The following materials should be available to you:

- two blank scripts: embed.py and extract.py
- a secret message: secret.txt
- three bitmap images: dice.bmp, flowers.bmp, and tiger.bmp

```
To be, or not to be: that is the question: Whether 'tis nobler in the mind to
suffer The slings and arrows of outrageous fortune, Or to take arms against a sea
of troubles, And by opposing end them? To die: to sleep; No more; and by a sleep
to say we end The heart-ache and the thousand natural shocks That flesh is heir
to, 'tis a consummation Devoutly to be wish'd. To die, to sleep; To sleep:
perchance to dream: ay, there's the rub; For in that sleep of death what dreams
may come When we have shuffled off this mortal coil, Must give us pause: there's
the respect That makes calamity of so long life; For who would bear the whips and
scorns of time, The oppressor's wrong, the proud man's contumely, The pangs of
despised love, the law's delay, The insolence of office and the spurns That
patient merit of the unworthy takes, When he himself might his quietus make With
a bare bodkin? who would fardels bear, To grunt and sweat under a weary life, But
that the dread of something after death, The undiscover'd country from whose
bourn No traveller returns, puzzles the will And makes us rather bear those ills
we have Than fly to others that we know not of? Thus conscience does make cowards
of us all; And thus the native hue of resolution Is sicklied o'er with the pale
cast of thought, And enterprises of great pith and moment With this regard their
currents turn awry, And lose the name of action.--Soft you now! The fair Ophelia!
Nymph, in thy orisons Be all my sins remember'd!
```

secret message

images

This worksheet will assume the use of Python 3.6 and the Python Image Library (PIL) on replit.com.

You should have access to a replit environment called 'IntroToSteg' containing the materials listed above.

To run a python script on replit.com (*e.g.*, embed.py), navigate to the 'Shell' tab and run the command:

```
python embed.py
```

To output information to the console from a Python script (*e.g.*, for debugging), use the print command:

```
print('output')
```

To insert a comment in a Python script, use #.

Introduction to Steganography – Worksheet

embed.py

1. Choose the configurations. Start by setting the following variables (replacing ░ with your choice of values):

```
# Choose the configurations
key = ░
colourPlane = ░  # 0 is red, 1 is green, 2 is blue
significantBit = ░  # 7 is the least significant bit, 0 is the most
coverImage = '░'
secret = '░'
```

   The key will be used to protect the secret by embedding it into the cover-image in such a way that the recipient needs to have the same key to extract it. The colour plane sets whether to embed information into the red, green, or blue pixel values. The significant bit sets which bit of the binary representation of the pixel value to embed into; initially this should be set to 7, the least significant bit (LSB). The cover-image is the image file into which the secret will be embedded; for example, 'img/flowers.bmp'. The secret is the secret message that is being sent; initially, this should be set to the string found in secret.txt.

2. Read the cover-image data. Start by importing the *Image* module from PIL. Then, open the cover-image file to access its data and metadata.

```
from PIL import Image
```

```
# Read the cover-image data
image = Image.open(coverImage)
dimensions = image.size
pixels = image.load()
```

   The `dimensions` variable is a 2-tuple and holds the image size metadata; `dimensions[0]` holds the width and `dimensions[1]` holds the height. This can be used to calculate how many pixels are in the image and therefore how much information can be embedded into it.

   The `pixels` variable holds the pixel colour values. For example, `pixels[0,0]` will return a list of the colour values of the pixel at position (0,0), such as (255,0,0) if the pixel is red. In Python, the origin (*i.e.*, the (0,0) position) is the top left of the image. Furthermore, `pixels[0,0][colourPlane]` will return just the colour value in the given colour plane. This can be used to see the relevant colour plane value of a pixel, so as to decide whether or not to modify it when embedding information into it.

3. Determine the pixel embedding order. Start by importing the *random* module. Then, create a list the same size as the number of pixels in the cover-image and re-arrange its entries in a deterministic way (such that the extractor can generate the same sequence).

```
import random
```

```
# Determine the pixel embedding order
shuffledIndices = list(range(0, dimensions[0] * dimensions[1]))
random.seed(key)
random.shuffle(shuffledIndices)
```

   The `shuffledIndices` variable is initiated as a list of consecutive positive integers (0, 1, 2, 3,…) with a number of entries equal to the number of pixels in the image, calculated by multiplying the width of the image by its height. Each entry in the list represents a pixel in the image—so when the list is re-arranged in a repeatable way, it can be used as a guide for the embedding and extraction processes.

   The `random.seed()` function initialises Python's pseudo-random number generator (PRNG). By default, this function initialises using the current time; however, by providing a parameter (in this case, the key), it is forced to use the parameter value instead, making any subsequent randomness deterministic. This means that, as long as the embedding algorithm and the extraction algorithm use the same parameter, their random number sequences will be generated the same.

   The `random.shuffle()` function re-arranges the entries in the `shuffledIndices` list in a randomly-generated order. As mentioned, this random ordering is deterministic due its seeding, and can be recreated at any time by using the same seed.

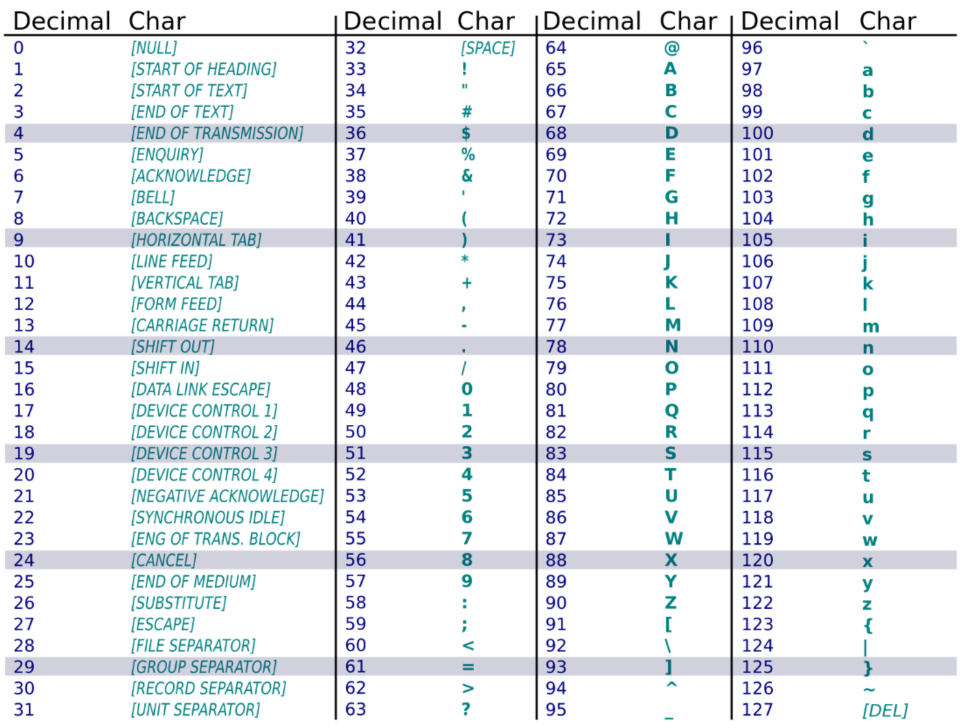
4. Convert the secret into ASCII bits and prepend the length of the secret to it to aid extraction.

```
# Convert the secret into ASCII bits
# Prepend the length of the secret to it to aid extraction
sbits = ‘’.join(format(ord(char), ‘b’).zfill(7) for char in secret)
lbits = format(len(secret), ‘b’).zfill(14)
bits = lbits + sbits
```

ASCII-encoding is used to convert characters into 7-bit representations for electronic communication (see the table below). This means that each character is assigned a value between 0 and 127, which can then be represented by 7 binary digits. For example, the character *A* is 65 in ASCII, and 65 is 1000001 in binary.

The code above iterates over the secret, converting each character into a 7-bit ASCII representation, and concatenates the results into a binary string that is then stored in `sbits`. Then, it calculates the length of the secret (the number of characters), converts that number to binary, and pads it to 14 bits (*i.e.*, it inserts 0s at the start to occupy all 14 allocated bits) that it stores in `lbits`. Finally, it concatenates `lbits` and `sbits` and stores the result in `bits`.

Now that the secret is represented in 0s and 1s only, it is ready to be embedded.



Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
0	[NULL]	32	[SPACE]	64	@	96	`
1	[START OF HEADING]	33	!	65	A	97	a
2	[START OF TEXT]	34	"	66	B	98	b
3	[END OF TEXT]	35	#	67	C	99	c
4	[END OF TRANSMISSION]	36	$	68	D	100	d
5	[ENQUIRY]	37	%	69	E	101	e
6	[ACKNOWLEDGE]	38	&	70	F	102	f
7	[BELL]	39	'	71	G	103	g
8	[BACKSPACE]	40	(	72	H	104	h
9	[HORIZONTAL TAB]	41	)	73	I	105	i
10	[LINE FEED]	42	*	74	J	106	j
11	[VERTICAL TAB]	43	+	75	K	107	k
12	[FORM FEED]	44	,	76	L	108	l
13	[CARRIAGE RETURN]	45	-	77	M	109	m
14	[SHIFT OUT]	46	.	78	N	110	n
15	[SHIFT IN]	47	/	79	O	111	o
16	[DATA LINK ESCAPE]	48	0	80	P	112	p
17	[DEVICE CONTROL 1]	49	1	81	Q	113	q
18	[DEVICE CONTROL 2]	50	2	82	R	114	r
19	[DEVICE CONTROL 3]	51	3	83	S	115	s
20	[DEVICE CONTROL 4]	52	4	84	T	116	t
21	[NEGATIVE ACKNOWLEDGE]	53	5	85	U	117	u
22	[SYNCHRONOUS IDLE]	54	6	86	V	118	v
23	[ENG OF TRANS. BLOCK]	55	7	87	W	119	w
24	[CANCEL]	56	8	88	X	120	x
25	[END OF MEDIUM]	57	9	89	Y	121	y
26	[SUBSTITUTE]	58	:	90	Z	122	z
27	[ESCAPE]	59	;	91	[	123	{
28	[FILE SEPARATOR]	60	<	92	\	124	|
29	[GROUP SEPARATOR]	61	=	93	]	125	}
30	[RECORD SEPARATOR]	62	>	94	^	126	~
31	[UNIT SEPARATOR]	63	?	95	_	127	[DEL]

ASCII table

5. Embed the secret bits into the re-ordered pixels in the chosen colour plane. Start by defining a function that takes as parameters the pixel values to be modified, the chosen colour plane, the significant bit, and a modifier instructing whether the pixel value should be increased or decreased (*i.e.*, changed from a 0 to a 1, or vice versa).

```
def modify_pixel(pixel, plane, bit, modifier):
        m = modifier * (2 ** (7 - bit))
        r = pixel[0] + m if plane == 0 else pixel[0]
        g = pixel[1] + m if plane == 1 else pixel[1]
        b = pixel[2] + m if plane == 2 else pixel[2]
        return (r, g, b)
```

This is required for programmatical reasons. The `m` variable is used to modify the correct bit in the pixel value. Since the pixel value is stored as an integer in the `Image` module, `m` uses the significant bit to calculate how much to modify this integer to achieve the desired modification in the binary representation. The modification is then made to the relevant colour plane only and the pixel values (one modified, two not) are returned.

```
# Embed the secret bits into the re-ordered pixels in the chosen colour plane
for i in range(len(bits)):
        x = shuffledIndices[i] % dimensions[0]
        y = int(shuffledIndices[i] / dimensions[0])
        p = format(pixels[x, y][colourPlane], ‘b’).zfill(8)

        # If the existing value is 0, only change it if the secret bit is 1
        if p[significantBit] == ‘0’:
                if bits[i] == ‘1’:
                        pixels[x, y] = modify_pixel(pixels[x, y], colourPlane, significantBit, 1)
        # If the existing value is 1, only change it if the secret bit is 0
        else:
                if bits[i] == ‘0’:
                        pixels[x, y] = modify_pixel(pixels[x, y], colourPlane, significantBit, -1)
```

The code above implements the embedding algorithm. It iterates over the `bits` variable, in order to embed one bit at a time. It uses the `shuffledIndices` list to determine the selection order of pixels in which to embed.

For each bit, first determine the (`x`,`y`)-coordinates of the pixel into which the bit will be embedded by converting the corresponding value in `shuffledIndices`, then convert the relevant colour value of that pixel into its 8-bit binary representation (since the colour values can range from 0 to 255) and store it in `p`. In `p`, the significant bit is compared with the bit to be embedded and the pixel is only changed if the values are different. This means that the resulting bit in the stego-image will be precisely the bit embedded.

6. Output the stego-image.

```
# Output the stego-image
image.save(‘stego-image.bmp’)
```

The final step is to output the modified image.

extract.py

1. Set the configurations. In order to extract the correct information, the extraction algorithm must use the same key, colour plane, and significant bit as the embedding algorithm.

```
# Set the configurations
key = 
colourPlane = 
significantBit  = 
stegoImage = ‘stego-image.bmp’
```

2. Read the stego-image data. As before, open the stego-image file to access its pixel colour values (`pixels`) and size metadata (`dimensions`).

3. Determine the pixel extraction order. As before, create and seed an identical `shuffledIndices` list.

4. Extract the embedded data by reversing the embedding process.

```
# Extract the embedded data by reversing the embedding process
extractedBits = []
for i in shuffledIndices:
        x = i % dimensions[0]
        y = int(i / dimensions[0])
        p = format(pixels[x, y][colourPlane], ‘b’).zfill(8)
        extractedBits.append(p[significantBit])
```

The `extractedBits` variable is initiated as a blank list. The `shuffledIndices` list is used to select the pixels in the same order as during the embedding process, so as to extract the data in the same order as it was embedded. Since the extracted bit is precisely the bit that was embedded into it, no further processing is needed and it is then stored in `extractedBits`.

`extractedBits` should therefore contain all of the embedded data in the correct order.

5. Determine the length of the secret.

```
# Determine the length of the secret
extractedLengthBits = extractedBits[:14]
extractedLength = 0
for i in range(len(extractedLengthBits)):
        extractedLength += int(extractedLengthBits[i]) * (2 ** (13 - i))
```

   Since the first 14 bits of the extracted data contain the length of the secret, these bits are treated as a binary representation of an integer; the converted integer is then stored in `extractedLength`.

6. Convert the ASCII bits to the secret. Display the secret in the console.

```
# Convert the ASCII bits to the secret
extractedSecretASCII = []
for i in range(0, extractedLength):
        a = 0
        for j in range(0, 7):
                a += int(extractedBits[14 + i * 7 + j]) * (2 ** (6 - j))
        extractedSecretASCII.append(chr(a))

print(‘’.join(extractedSecretASCII))
```

   Using `extractedLength`, the remaining bits in `extractedBits` are processed. Since these bits represent ASCII-encoded characters, they are parsed 7-at-a-time to reconstruct each 7-bit ASCII value, which is then converted to a character. These characters are concatenated to form the secret, which is printed to the console.

If the embedding and extraction algorithms are both correct, and both are using the same configurations, then the secret message should be successfully retrieved and displayed.

Questions

1. (a) If a single bit of data is embedded into each pixel, which image can hold the most embedded data?

	dice.bmp	**flowers.bmp**	**tiger.bmp**
filesize	532 kB		
dimensions	400*454		
pixels	181,600		
capacity	22.17 kB		

   (b) What is the relationship between the filesize and the capacity?

   (c) How might the capacity be increased and what might be the drawback of doing so?

2. (a) The secret consists of 1,488 characters. If these characters are ASCII-encoded, how many times could the secret be embedded into each image?

   (b) What might be the benefit of embedding the secret multiple times?

3. Experiment with the stegosystem. Try different images, try difference colour planes, try different significant bits, try embedding more bits (either embed the secret multiple times or use a longer secret). What do you notice?

Extensions

1. Make a copy of your embed script and call it highlight.py, keeping the same configurations. In this script, modify your embedding algorithm to set pixels to a fixed colour, instead of embedding bits, and output an image called ‘highlighted-image.bmp’. Use this to see which pixels are being modified. Are they evenly spread across the image? Is your data protected against cropping attacks? How might you improve the system against cropping?

2. Create a small graphical logo (or find an existing one). How would you modify your scripts to embed such a logo into a cover-image and extract it again? Plan the changes that you will need to make and implement them. What might be the benefits and drawbacks of using a graphical secret compared with a text-based secret?