Problem Set: Crypto

In contrast with the small isolated exercises you have been doing so far, the goal of this Problem Set assignment is to give you the opportunity to create something a little larger and more complicated, composed of multiple parts working together, which combine together to create a complete working program that interacts with a user.

This assignment consists of 4 parts:

1. **Initials**: a small exercise to get warmed up for the main event
2. **Caesar**: an encryption algorithm
3. **Vigenere**: an even cooler encryption algorithm
4. **Making Some Improvements**: a handful of small adjustments to add some features and improve the quality of your code

Ready? Let's do this!

> *NOTE* This problem set is inspired by the original Crypto Pset in CS50, adopted for Python rather than C.

Prerequisites

Before starting work on this, you should have completed the previous section, Using Python Locally (LocalPython.html).

Setup

For this assignment, you'll be writing code locally on your own machine, and running your code at the command line.

Open up a terminal window on your computer, and use the `cd` command to navigate to the folder where you save documents for this class. If you have not yet created such a folder, go ahead and create one now.

Once you are inside your directory for the class, create a new sub-directory called `crypto/`, and then `cd` into it:

```
$ mkdir crypto
$ cd crypto
```

Remember that in the example above, you should not actually type the dollar sign `$`. We use that symbol

just as a convention to indicate that the example takes place at the command-line prompt in a terminal window.

Part 1: Initials

Create a new file called `initials.py`. Open up that file in a text editor, and complete the following function:

```
def get_initials(fullname):
    """ Given a person's name, return the person's initials (uppercase) """
    # TODO your code here
```

Your function will receive one argument, `fullname`, a string representing someone's name, and should return a string with that name's capitalized initials. You may assume that the name will contain only letters (uppercase and/or lowercase) plus single spaces between words. This means you don't have to worry about Conan O'Brien, T.S. Eliot, or Cee-Lo Green.

Here are some examples of what your function should return for various `fullname` arguments:

`fullname`	**return value**
Ozzie Smith	OS
bonnie blair	BB
George	G
Daniel Day Lewis	DDL

If you were to invoke your function, it would look something like this:

```
answer = get_initials("Ozzie Smith")
print("The initials of 'Ozzie Smith' are", answer)
# => prints "The initials of 'Ozzie Smith' are OS"
```

Notes, Tips and Hints

- You'll need to collect the initials as you find them, and return them all together at the end. You may want to re-read about The Accumulator Pattern (../Strings/TheAccumulatorPatternwithStrings.html).
- Even if the name starts with a lowercase letter, you should always capitalize the initials. For example, if `fullname == "ozzie smith"`, you should still return `"OS"`.

Testing

When you are finished writing your `get_initials` function, you should test it to make sure it works. There are a few ways to do this:

1. You can import your script into a REPL (Python shell), and then feed various inputs into your function.
2. You can just add some print statements (like the example above) to `initials.py` script.

Technique 1 looks something like this:

```
$ python3
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from initials import get_initials
>>> get_initials("Ozzie Smith")
OS
>>> get_initials("bonnie blair")
BB
>>> get_initials("Daniel Day Lewis")
DDL
... etc
quit()
```

That looks complicated but its actually very easy. Try typing `python3` into your terminal and you'll see. Technique 1 is definitely recommended, because writing and changing bunch of print statements starts to get annoying very quickly.

But if you prefer Technique 2, here's how that works: Simply add print statements to your file, and then run your script on the command-line:

```
$ python3 caesar.py
The initials of 'Ozzie Smith' are OS
The initials of 'bonnie blair' are BB
The initials of 'Daniel Day Lewis' are DDL
... etc
```

After running your script, just test by hand that the output matches what you expected to see.

> *NOTE* Remember that we've been using Python 3 in this class. So when you try to run your program, make sure you type `python3 initials.py` , rather than simply `python initials.py` , which would run the Python 2 interpreter.

Either way, whether using the REPL or print statements, make sure to test your function agains a healthy variety of inputs.

Make It Interactive

Just for fun, let's turn this into an interactive program that a user can run from the terminal. All you have to do is add an `input` statement to ask the user for his/her name, and then a `print` statement to report the results back to him/her. Your program should work like this:

```
$ python3 initials.py
What is your full name?
Ozzie Smith
OS
```

Just to be clear about the example above:

- The user typed the first line, causing the program to run.
- Then, the program printed the second line asking for their name.
- Then the user typed the third line ("Ozzie Smith").
- Finally, the program printed the initials ("OS").

Part 2: Caesar

Now it's time for some encryption!

In chapter 9, you completed an exercise that had you write a function called `rot13`, which used the Caesar Cipher (https://en.wikipedia.org/wiki/Caesar_cipher#History_and_usage) to encrypt a message. If you need a refresher, this is want the exercise said:

> Write a function called `rot13` that uses the Caesar cipher to encrypt a message. The Caesar cipher works like a substitution cipher but each character is replaced by the character 13 characters to 'its right' in the alphabet. So for example the letter a becomes the letter n. If a letter is past the middle of the alphabet then the counting wraps around to the letter a again, so n becomes a, o becomes b and so on. *Hint*: Whenever you talk about things wrapping around its a good idea to think of modulo arithmetic.

The idea is to encrypt the message character by character, rotating each letter 13 places to the right. So for example, **a** becomes **n**, **b** becomes **o**, **c** becomes **p** and so on. At the end of the alphabet we wrap around, so that **m** shifts to **z** and then **n** shifts to **a**.

The end result is a *super secret coded message* that that looks like gibberish to any outsiders.

We're going to build a more general version of the rot13 algorithm that allows a message to be encrypted using *any* rotation amount, rather than just 13. Ultimately, users will be able to type a message in the terminal, and specify a rotation amount (13, 4, 600, etc), and your program will print the resulting encrypted message.

The final interactive program will run like this:

```
python3 caesar.py
Type a message:
Hello, World!
Rotate by:
5
Mjqqt, Btwqi!
```

We are going to do this in a few steps, so you can break the problem down into isolated pieces.

First, open up a file `caesar.py` in your editor.

alphabet_position

The first thing we are going to do is simply create a helper function which will prove useful in a few different places.

Write a function `alphabet_position(letter)`, which receives a letter (that is, a string with only one alphabetic character) and returns the 0-based numerical position of that letter within the alphabet. It should be case-insensitive.

Here are some example input parameter values, with the corresponding return values.

`letter`	**Return value**
a	0
A	0
b	1
y	24
z	25
Z	25

Don't worry about what might happen if somebody tries to use your function with an input parameter that is something other than a single letter, like `alphabet_position("grandpa")`

When you are finished, you should test your function to make sure it works.

rotate_character

Next, write another helper function `rotate_character(char, rot)` which receives a character `char` (that is, a string of length 1), and an integer `rot`. Your function should return a new string of length 1, the result of rotating `char` by `rot` number of places to the right.

Here are some example input values, with the corresponding return values.

`char`	`rot`	**Return value**
a	13	n
a	14	o
a	0	a

c	26	c
c	27	d
A	13	N
z	1	a
Z	2	B
z	37	k
!	37	!
6	13	6

A few important things to notice:

- The upper or lower case of the letter should be preserved. For example, `rotate_character("A", 13)` results in `"N"`, rather than `"n"`
- For non-alphabetical characters, you should ignore the `rot` argument and simply return `char` untouched. For example, see `"!"` and `"6"` in the table above.
- You should make use of your own `alphabet_position` function. If feeling confused, you may want to re-read about how Functions Can Call Other Functions (../Functions/Functionscancallotherfunctions.html)

Test `rotate_character` with various input values before moving on to the next stage. Use more tests than the examples we provide.

encrypt

At this point your caesar.py file should look like this:

```
def alphabet_position(letter):
    # blah blah
    # beautiful code is written here


def rotate_character(char, rot):
    # more beautiful code
```

Now let's get to the heart of the matter. Write one more function called `encrypt(text, rot)`, which receives as input a string and an integer. This is just like the `rot13` function, but instead of hardcoding the number 13, your function should receive a second argument, rot which specifies the rotation amount. Your function should return the result of rotating each letter in the `text` by `rot` places to the right.

Here are some example input values, with the corresponding return values.

`text`	`rot`	**Return value**
a	13	n
abcd	13	nopq
LaunchCode	13	YnhapuPbqr
LaunchCode	1	MbvodiDpef
Hello, World!	5	Mjqqt, Btwqi!

A few things to note:

- The `text` argument might contain non-alphabetic characters (spaces, numbers, symbols). You should leave these as they are.
- You should make use of your own `rotate_letter` function (which should make it very easy to satisfy the condition above).

When you're finished, you should of course test your function against a bunch of different inputs and make sure it works.

Make It Interactive

You're almost done with Caesar! The last step is simply to write some `print` and `input` statements so the user can run your program from the terminal. Remember, you should ask the user for their message and rotation amount, and then print the encrypted message:

```
$ python3 caesar.py
Type a message:
Hello, World!
Rotate by:
5
Mjqqt, Btwqi!
```

Part 3: Vigenere

If you're trying to pass notes in the back of class with your best friend Suzie, the Ceasar cipher would be fairly easy for a nosy outsider to decode. Let's implement a more complicated cipher algorithm.

Watch this short video (https://www.youtube.com/watch?v=9zASwVoshiM&feature=youtu.be) on the Vigenere cipher, courtesy of the CS50 folks at Harvard.

As you saw in the video, Vigenere uses a word as the encryption key, rather than an integer.

Your program will work like this:

```
$ python3 vigenere.py
Type a message:
The crow flies at midnight!
Encryption key:
boom
Uvs osck rmwse bh auebwsih!
```

Above, the user entered a message of “The crow flies at midnight” and an encryption key of “boom”, and the program printed “Uvs osck rmwse bh auebwsih!”.

How did we arrive at that result? Here is a detailed breakdown:

char from input string	**cipher char**	**rotation amount**	**result char**
T	b	1	U
h	o	14	v
e	o	14	s
(space)	n/a	n/a	(space)
c	m	12	o
r	b	1	s
o	o	14	c
w	o	14	k
(and so on…)			

Some important things to notice:

- As with Caesar, the upper or lower case of each character should be preserved.
- As with Caesar, non-alphabetical characters like `" "` and `"!"` do not get encrypted.
- When we encounter a non-alphabetical character, the encryption key *does not* use up another letter. For example, notice how the `"m"` in `"boom"` does not get “wasted”, so to speak, on the space character. Instead, it is “saved” for the next alphabetical character, the `"c"` in `"crow"`.
- Whenever we reach the end of the encryption key (“boom”) before reaching the end of the message, the encryption key wraps back around to the beginning again (the letter “b”).

Reusing your Caesar code

You probably noticed that Vigenere is very similar to Caesar. The only difference is that the rotation amount varies throughout the course of the message.

Whenever you find yourself in a situation like this–faced with a coding task that is very similar to one you did previously–your instinct should be to sniff around for ways to reuse the code you have already written. Ideally, all the work that is required by both tasks should be factored out into reusable components (like functions).

In this case, the majority of the logic that Vigenere has in common with Caesar is encapsulated in those two helper functions you wrote, `alphabet_position` and `rotate_character`. Indeed, that is why we intentionally guided you down the path of writing those functions. You are going to find both of those functions equally helpful for implementing Vigenere.

Go ahead and copy / paste those functions into `vigenere.py` so you can use them. (In reality, copy / pasting is not a very smart thing to do here, and there is a better way, which you will see farther down in this assignemnt. But for now, just do it.)

encrypt

Now that you have your helper functions, go ahead and write a new version of the `encrypt` function which uses the Vigenere cipher rather than Caesar. First, figure out what the function signature should say. How should it be different from the Caesar version, `def encrypt(text, rot)`?

As usual, don't move on until you have tested your function against a lot of different inputs and seen the results you expect.

Make It Interactive

Finally, add the appropriate `print` and `input` statements so that your program behaves as specified:

```
$ python3 vigenere.py
Type a message:
The crow flies at midnight!
Encryption key:
boom
Uvs osck rmwse bh auebwsih!
```

Part 4: Making Some Improvements

Congrats! You have created two very cool encryption programs.

Before calling this a done-deal, let's make a few improvements to the project by refactoring and adding a few new features. You will do three things:

A. **Refactor: Shared Code**

   Do some refactoring so that you share the two helper functions between files, rather than copy and paste.

B. **New Feature: Command-line Arguments**

   Add a feature that improves the user experience by allowing the user to type their rotation amount as a *command-line argument* rather than waiting for a prompt. (Caesar Only)

C. **New Feature: Validation**

   Add some validation on user input, so that if the user types something dumb, your program handles it gracefully, rather than crashing. (Caesar Only)

A. Refactor: Shared Code

Remember when we said that copy / pasting those helper functions is not a smart thing to do? Now let's do something better.

The reason that copy / pasting is a bad idea is that now you have two copies of the same exact code lying around. What happens if you realize you need to change the function? You will have to remember to make the same change in both copies. That might not sound so bad, but imagine if you had not two, but three copies, or six, or eleven? Given that you want to use the same function everywhere, that function should only ever be defined once.

Using `import`

If a function is only defined in one place, a particular file somewhere, then how can some other file use it? It is actually quite easy: the other file simply needs to `import` the function. You have already used the `import` keyword throughout this course, whenever you wanted to access code written by other people, such as the `math` and `random` modules. It is also possible to create and import your own code. Here's how:

1. In your editor, open up a new file called `helpers.py`. Paste both functions, `alphabet_position` and `rotate_character` into this new file.
2. Next, open up `caesar.py`, and delete both of those functions.
3. Finally, add this line to the top of `caesar.py`:

```
from helpers import alphabet_position, rotate_character
```

This says that we want to import code from a module `helpers`, but that we only want to import particular pieces of that module, in this case the functions `alphabet_position` and `rotate_character`.

Now we should be able to use those functions! Try running `python3 caesar.py` again, and you should find that it works just like it did before.

> *NOTE* In order for this to work, it is essential that `helpers.py` is in the same directory as `caesar.py`. Also note that the technique we are using here is a little simpler than the way this is normally done. For larger projects, where the structure is a tree of folders within folders, there is a slightly more involved procedure for reusing code, which does not require both modules to live together in the same folder. If you're curious, you can read up more about creating modules in Python in the Python module documentation (https://docs.python.org/3/tutorial/modules.html).

Once you have Caesar working, do the same thing for Vigenere: simply delete the two helper functions, and `import` them from `helpers.py`.

Now your helper functions are defined only once, and your code remains nice and DRY (Don't Repeat Yourself)!

B. New Feature: Command-line Arguments

Let's now make the following tweak to Caesar: instead of prompting the user for two things – the text message and the rotation amount – let's allow the user to include the rotation amount right away at the beginning.

Rather than behave like this:

```
python3 caesar.py
Type a message:
Hello, World!
Rotate by:
5
Mjqqt, Btwqi!
```

... we want our program to instead behave like this:

```
python3 caesar.py 5
Type a message:
Hello, World!
Mjqqt, Btwqi!
```

Notice how, on the first line, the user included the number `5` as an *argument* when running the program. This means that the program only needed to make one additional input prompt, asking for the text message. This makes for a slightly nicer user experience.

In order to implement this feature, you obviously need some way of knowing, inside your `caesar.py` script, that the user typed the number `5` as a command-line argument.

Conveniently enough, it just so happens that inside any Python program, you have access to a list containing each of the words the user typed on the command line. This list of strings lives in a module called `sys`, and has the variable name `argv` (short for "argument vector" ("vector" is another word for a list)).

Try adding the following two lines to the top of your `caesar.py` file:

```
from sys import argv
print("I know that these are the words the user typed on the command line: ", argv
)
```

Now run your program, and you should see output like this:

```
$ python3 caesar.py 5
I know that these are the words the user typed on the command line:  ['caesar.py',
'5']
Type a message:
... etc
```

The important part is the second line.

Notice that:

- The word “python3” is **not** included.
- The first item, `argv[0]` is always the name of your script (in this case, `'caesar.py'`).
- The other arguments follow. (In this case, we only have one additional argument, `'5'`).

Ok! Now you have all the tools you need to implement this feature. The `argv` variable is just a normal list like any other. The rotation number (5 or whatever it is), is sitting there inside that list, waiting for you.

To be clear, for this assignment, we only require that you update `caesar.py` to take a command-line argument. You can leave your Vigenere script as is.

C. New Feature: Validation

Let’s make one more improvement. You may or may not have noticed that if the user types certain things, your program will freak out.

There are two main cases to handle:

1. User fails to type a number when specifying rotation amount.

   ```
   python3 caesar.py grandpa
   ```

   If the user gives you something like “grandpa” instead of “5”, your program will crash, probably with this error:

   ```
   ValueError: invalid literal for int() with base 10: 'grandpa' on line X
   ```

2. User fails to provide a command-line argument.

   Now that you are expecting the user to specify the rotation amount via a command-line argument, there is a danger that the user will fail to type anything at all, i.e.:

   ```
   python3 caesar.py
   ```

   In this case, you will probably see:

   ```
   IndexError: list index out of range
   ```

   because you are trying to read from `argv` at an index that does not exist, since `argv` only contains one string, rather than two.

Rather than simply crash whenever one of these things happens, your program should handle it more gracefully. Write a function `user_input_is_valid(cl_args)`, which receives an array with the command-line arguments (you can just pass in `argv`), and returns a boolean indicating whether or not the user did everything correctly. You should return `False` if you see either of the two cases outlined above.

If your function returns `False`, your Caesar program should exit immediately, but first print a helpful "usage" message (explaining how to properly use your program). Below is an example of the user messing up, re-running the program, messing up again, etc:

```
$ python3 caesar.py
usage: python3 caesar.py n
$ python3 caesar.py grandpa
usage: python3 caesar.py n
$ python3 caesar.py 5.0
usage: python3 caesar.py n
$ python3 caesar.py 5
Type a message:
Hello, World!
Mjqqt, Btwqi!
```

4th time is the charm!

To check if the argument is an integer, there is a string method called `isdigit` which you should use. It works just like `isalpha`, but checks for integers rather than alphabetic characters:

```
>>> "grandpa".isdigit()
False
>>> "5.0".isdigit()
False
>>> "5".isdigit()
True
```

You can exit your program early by calling the `exit` function, which is part of the `sys` module. Just import the function by adding `exit` to your previous `import` statement:

```
from sys import argv, exit
```

and then invoke the function like this:

```
exit()
```

Ok, go forth and validate! As with the previous feature, this is only a requirement for Caesar. You do not have to update your Vigenere program.

Submitting Your Work

When you have finished, there is one more step you must do, in order to accommodate the fragile, picky grading-script: please comment out any `print` statements, `input` statements, and `exit()` commands.

For example, your final, submitted `caesar.py` file should look something like this:

```
from sys import argv, exit
from helpers import alphabet_position, rotate_character

def encrypt(text, rot):
    # (beautiful code)

def user_input_is_valid(cl_args):
    # (beautiful code)
    # (there should not be any print statements in here)

# EVERYTHING ELSE SHOULD BE COMMENTED OUT
```

Your version may look a little different, e.g. with some components in a different order. The important thing is that the last section (basically any code that actually executes when you run the script) should be commented out. Please do not delete this code entirely, because we do want to see it with our human eyes. It's just the sensitive robot-grader who must be shielded.

Once you have commented out all the side-effect-causing code from all your files, go to Vocareum and click the assignment titled *Problem Set: Crypto*. Rather than copy and paste your work, you can upload your files directly. In your Vocareum work environment, click the Upload button, and select all 4 files:

- initials.py
- casear.py
- vigenere.py
- helpers.py

Finally, as usual, click Submit!

(LocalPython.html) (../Studios/Chapter11.html) (LocalPython.html)
(../Studios/Chapter11.html)

Mark as completed