

Lab 2: Modules, variables, Strings, for loops

This lab continues the introduction to the Python programming/scripting language, focussing on modules and variables, particularly the built-in Python String Object data type.

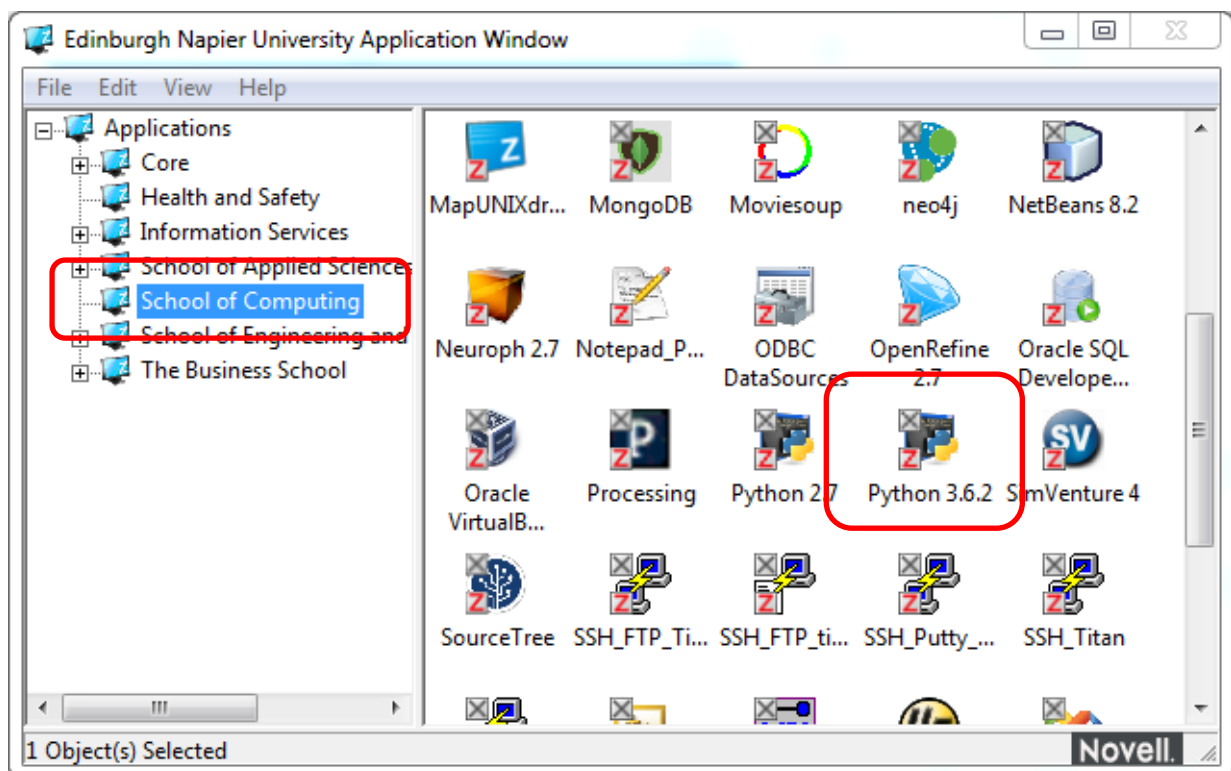
WARNING: In Python 2.x, print statements could be used without brackets. I have tried to change all print statements in this lab to work for Python 3.x, but may have missed one or two.

NOTE: The Caesar Cipher exercise is the first part of the Portfolio assessment (see separate assessment brief for details).

1. (OPTIONAL) Running Python in JKCC and other labs

If you are using a computer on campus in a general lab, Python may not be installed.

In this case you can run it from the “Edinburgh Napier University Application Window”. Find Python 3.6.2 in School of Computing.



The default save location is the local C: drive, so make sure you change this e.g. to your H: drive (Documents points there) - otherwise your work will disappear down a black hole when you log off.

2. Python Modules

Scripts and Functions can be shared and reused between projects or developers, using Python Modules. Modules are simply Python .py files containing code such as functions. The modules can then be imported in any script, giving access to the reusable code.

Using Functions

To use a function from a module, the following syntax is used:

```
from <module_name> import <function_name>
```

Or more commonly, to import all functions:

```
import module_name
```

From the IDLE Python Shell, start a new session with **Shell>Restart Shell**.

Q: What happens if you try calling your print_msg function from last week?

The function is not defined in the shell, so you cannot call it.

```
>>> ===== RESTART =====
>>> print_msg("hello")

Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    print_msg("hello")
NameError: name 'print_msg' is not defined
```

Now try importing the function using the syntax above, and then calling it.

Q: What happens if you try calling your print_msg function after importing it?

```
>>> ===== RESTART =====
>>> from hello_world import print_msg
>>> print_msg("brian")
hello brian
>>> |
```

Q: What is the difference when calling the function, if you import the entire module?

The function then needs to be prefixed with the module name space.

```
>>> ===== RESTART =====
>>> import hello_world
>>> hello_world.print_msg("John")
hello John
>>> |
```

This way, local functions and functions from other modules cannot be confused.

3. Module Boiler-plate

A boiler-plate in Python is a block at the end of the script. In its simplest form, it could look like:

```
if __name__ == '__main__':  
    main()
```

The main reason for having a boiler-plate is that it allows the same module (.py file) to be used in two different ways – it can be executed directly as a script, or be imported and used in another module. By doing the “main” check, you can have that code only execute when you want to run the module as a program and not have it execute when someone just wants to import your module and call your functions themselves.

Let’s test this out.

In IDLE, create a new file boiler_demo.py which contains:

```
# boiler_demo.py  
# demonstrates the function of a boiler plate  
# modified from http://ibiblio.org/g2swap/byteofpython/read/module-name.html  
  
if __name__ == '__main__':  
    print ('This program is being run by itself')  
else:  
    print ('I am being imported from another module')
```

Q: What is the output when you run this script using run>module or F5?

Q: What is the output when you import the module with `>>> import boiler_demo`

Now add boiler-plate code to your hello_world module from last week, so it can be run as a script, or imported as a module:

```
*hello_world_boiler.py - C:/Users/Petra/Dropbox/CSN08114 Python/hello_world_boiler.py (3.6.2)*
File Edit Format Run Options Window Help

# script: hello_world with boilerplate
# author: Petra
# created: 23/9/17
#
def print_msg(msg):
    '''Print a hello along with the given argument'''
    print("hello " + msg)

def main():
    print('running test cases from main()')
    print_msg("Petra")
    print_msg("everyone")

# Standard boilerplate code to call the main() function
if __name__ == '__main__':
    main()
```

Try running as a script, and importing as a module.

Similarly, change your keyspace calculator script from last week to a module.
Check that you can import the module and then use the function from the interactive interpreter.

4. Doc strings and other documentation

In the lecture, we looked at the documentation that is available for all built in and external modules. Obviously (!) it would make sense to provide such documentation for all your modules and functions as well. In this exercise, we will look in depth at how to do this, using the built in "os" module as an example.

First, we need to import the module.

```
>>> import os
```

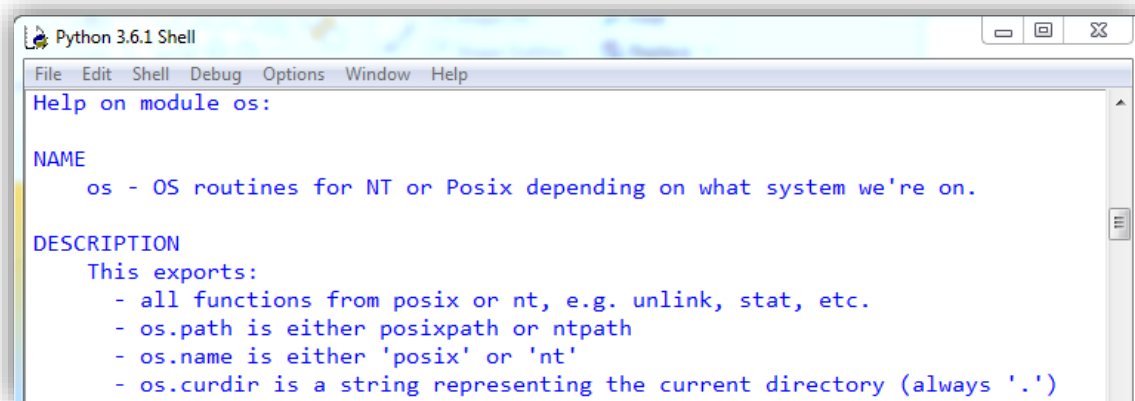
Now try the following

```
>>> dir(os)
>>> help(os)
```

Q: What is the output from the `dir(os)` command?

Q: What is the filename and path of the module? (Hint: look at the bottom of the `help(os)` output.)

Scroll up to the top of the `help(os)` output. As you can see, there is help for the module as a whole here, a longer name and a description.



```
Python 3.6.1 Shell
File Edit Shell Debug Options Window Help
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

DESCRIPTION
    This exports:
      - all functions from posix or nt, e.g. unlink, stat, etc.
      - os.path is either posixpath or ntpath
      - os.name is either 'posix' or 'nt'
      - os.curdir is a string representing the current directory (always '.')

```

Let's find out how this was created.

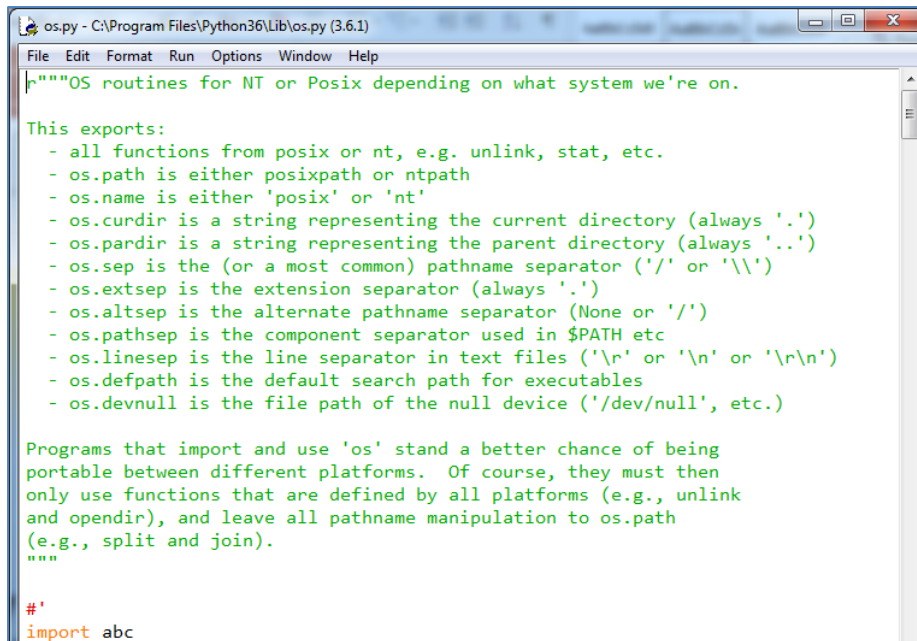
Find the main Python directory, and in it the file that contains the `os` module.

Make a copy of this file for yourself and open the copy in IDLE or Notepad++.

Q: Why should you give this copy a different filename (e.g. `my_os.py`) from the original?

Q: What does the first line of the file start with?

Q: How does the `help()` function know what should be the "name" and what the "description" of the module?



```
os.py - C:\Program Files\Python36\Lib\os.py (3.6.1)
File Edit Format Run Options Window Help
"""OS routines for NT or Posix depending on what system we're on.

This exports:
- all functions from posix or nt, e.g. unlink, stat, etc.
- os.path is either posixpath or ntpath
- os.name is either 'posix' or 'nt'
- os.curdir is a string representing the current directory (always '.')
- os.pardir is a string representing the parent directory (always '..')
- os.sep is the (or a most common) pathname separator ('/' or '\\')
- os.extsep is the extension separator (always '.')
- os.altsep is the alternate pathname separator (None or '/')
- os.pathsep is the component separator used in $PATH etc
- os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
- os.defpath is the default search path for executables
- os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms. Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
"""

#
import abc
```

What about the function doc strings? Look back through the output of the `help(os)` command. You can see that it gives a short explanation for each of the functions. For example:

```
pipe()
    Create a pipe.

    Returns a tuple of two file descriptors:
    (read_fd, write_fd)
```

This is called the **function doc string**.

Q: Where can you find this function doc string in the module file?

Q: type the command:

```
>>> os.pipe(
```

What happens when you pause typing at the open bracket?

You should ensure that all your own functions have helpful doc strings, and your modules have an appropriate brief description. For the assessments, you will get marks for this. Most of the starting scripts provided for you will have place holders for the doc strings which you need to change. You should also add a module description.

5. Python Basic Data Types – Scientific notation for numbers

Last week, with the `keyspace.py` script, you saw some output in scientific notation. For example, you may have found that there are $6.302494097246094e+17$ (`math.pow(95,9)`) possible ascii passwords with 9 characters.

Q: How can you use python to find out what $6.302494097246094e+17$ is when written as a standard integer, without scientific notation? (there are several methods – how many can you think of? – see lecture slides)

Q: The chances of randomly guessing a 3-character ascii password at the first attempt are $p=1/(95^3)$. When Python displays the result of this, in scientific notation, what is the e-value? How many zeroes would appear after the decimal point when this is written out in full?

Q: Use a print statement with f-string notation to display these chances as a float with 7 decimal places (`.7f`). [refer to the lecture slides and/or later exercises below!]

The screenshot below shows how I used the `.format()` method. Can you work out what is greyed out here?

```
>>> out = 'The chances of randomly guessing a 4 character password \
at the first attempt are {0} or {1}'.format(0.00000001, 1.2277376631548254e-08)
>>> print(out)
The chances of randomly guessing a 4 character password at the first att
empt are 0.00000001 or 1.2277376631548254e-08
```

6. Python Basic Data Types – Strings

Python identifies strings as a **sequence/collection** of characters between quotation marks.

Python strings can be enclosed in single or double quotation marks. This can be useful if the string contains quotation marks, as you don't need to use an escape character.

```
>>> str_var1 = 'Don\'t Panic!!!'          # Using an escape character
>>> str_var2 = "Don't Panic!!!"          # No escape character needed
```

Q: When printed out, are `str_var1` and `str_var2` similar?

```
>>> str_var1
"Don't Panic!!"
>>> str_var2
"Don't Panic!!"
```

Python strings on multiple lines can be enclosed in triple quotation marks, or an escape character can be used to show the string spans multiple lines.

```
>>> str_var1 = "Don't \
Panic!"          # Using an escape character to span lines
>>> str_var2 = """Don't
Panic!"""        # Using """ to span lines
```

Q: When printed out, what do `str_var1` and `str_var2` contain now?

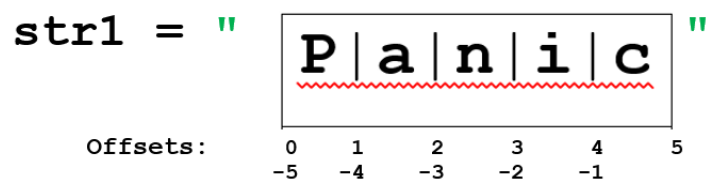
Python can perform a variety of operations on strings.

Use the BIF `len()` to calculate the length of a string:

```
>>> len(str_var1)
12
```

7. Slicing Strings

The slicing operator `[]` works with the string offsets/index positions to returning one or more characters from the string. The offsets for the example `str1="Panic"` are shown on the right:



Let's use our previously defined `str_var1` to experiment with slicing.

```
>>> str_var1[1]
'o'
>>> str_var1[11]
!
```

Q: What is the index of the first character in a string?

Q: What is the value of `str_var1[3]`?

Python can also slice backwards from the end of the string:

```
>>> print (str_var1[-3])      # Print 3rd last char in string
i
>>> print (str_var1[-12])     # Print 12th last char
D
```


Q: What negative index returns the last char in the string?

Q: How could the last letter be returned without negative indexing?

Q: What is the value of `str_var1[-7]`?

Python can also return sub-sections of a string, using slicing, using the `[:]` operator.

```
>>> print (str_var1[1:3]) # Slice start 1 to end 3
on
>>> print (str_var1[1:]) # Slice from start 1 to the end of string
on't Panic!
```

Q: What slice statement returns the first 3 chars of the string?

Q: What does `str_var1[:-2]` remove from the string?

Q: What is the value of `str_var1[::-1]`?

8. Concatenating or Joining Strings and basic string formatting

Strings can be concatenated using the `+` operator:

```
>>> print (str_var1 + " and carry on...")
"Don't Panic!! and carry on..."
```

Q: What is the value of `str_var1` now? Has it changed?

Strings can also be defined using the concatenation operator:

```
>>> str_var3 = "Please " + str_var1[:-2] + " and carry on..."
>>> print (str_var3)
```

Q: In one line of Python, how can we create a new string `str_var4`, using a slice to return the original string `str_var1` without the exclamation marks, and then concatenate `"???"` onto the end?

Strings can be *joined* in a *more efficient way and formatted at the same time* using `.format` or f-string notations:

```
>>> home='Forfar'
>>> away='East Fife'
>>> home_score, away_score=4,5
>>> str_var5 = "Final score was {} {} {} {}..." .format(home, home_score,
away, away_score)
>>> print(str_var5)
Final score was Forfar 4 East Fife 5...
>>> str_var6=f"Final score was {home} {home_score} {away} {away_score}."
>>> print(str_var6)
Final score was Forfar 4 East Fife 5.
```

Try this out yourself!

f-strings are a new feature introduced for Python 3.6. Find out more by reading about PEP 498, Literal String Formatting (f-strings) at <https://docs.python.org/3/whatsnew/3.6.html> and https://docs.python.org/3/reference/lexical_analysis.html#f-strings. The full specification is at <https://www.python.org/dev/peps/pep-0498/>.

Strings can be *joined* in an even *more efficient* way by using the string object `join()` method on a list of strings. If joining thousands of strings this should improve performance. We will cover lists more fully later. For now, just note that they are enclosed with square brackets, and list elements are separated with commas.

```
>>> list_var = [str_var1, "and", "carry", "on"]
>>> "".join(list_var)
"Don't Panic!andcarryon"
```

The above joins a list of strings without adding any separating characters in between – because the string before the `.` (dot) is an empty string.

To join a list of strings with a space as separating character use:

```
>>> list_var = [str_var1, "and", "carry", "on"]
>>> " ".join(list_var)
"Don't Panic! and carry on"
```

You can have multiple separating characters:

```
>>> list_var = ["Lions", "Tigers", "Bears"]
>>> " and ".join(list_var)
'Lions and Tigers and Bears'
```

Q: What are the results of the two joins below? Can you explain this?

```
>>> "?".join(str_var1)

>>> str_var1.join('???')
```

The `*` operator can be used to repeat the same string. It concatenates the string multiple times with itself.

```
>>> print ('ha'*4)
'hahahaha'
```

9. String Membership Operators

The `in` operator can be used to check if strings contain other strings.

```
>>> str_var1 = "Don't Panic!!"
>>> "Pan" in str_var1
True
>>> "pan" in str_var1      # string comparison is case sensitive
False
```

The `not in` operator can be used to check if strings do not contain other strings.

```
>>> "Snake" not in str_var1
True
>>> "Pan" not in str_var1
False
>>>
```

`string.find()` and `string.index()` methods

If you want the position where a substring occurs in a string, use `.find()` or `.index()`.

In lecture 2 (slides 29-30), we went through some `.find()` examples. Repeat these for practice.

```
>>> s = 'Good morning every1'
```

Q: What is the difference between `.find()` and `.index()`?

Hint: Try `s.index('m')`, `s.find('m')`, `s.find('z')` and `s.index('z')`.

10. Case of Strings

Python can convert strings to upper or lower case using the built in string methods `lower()` and `upper()`. These methods return a new string; all upper or lower case (except non letters, which are copied without change).

```
>>> str_var5 = "Please Don't Panic! Count to 10"
>>> str_upper = str_var5.upper()
>>> str_lower = str_var5.lower()
>>> print (str_upper)
"PLEASE DON'T PANIC! COUNT TO 10"
>>> print (str_lower)
"please don't panic! count to 10"
```

Q: What Python command(s) could be used to create a new string `str_capital` from `str_var5` with only the first character capitalised?

Hint: You could use slice to return the first letter, then `.upper()`, then slice the rest of the string, and use `.lower()`, then join the two parts together.

Python actually has a built in string method to capitalise strings in exactly this way!

```
>>> print (str_var5.capitalize())
Please don't panic!
```

As Python is interpreted, string literals are converted into string objects, and so methods can be called on them directly.

```
>>> print ("DON'T PANIC, take 5 mins ...".capitalize())
Don't panic, take 5 mins ...
```

Python also has a built in string method to capitalise each word in a string, `.title()`.

```
>>> print ("DON'T PANIC, take 5 mins ...".title())
Don'T Panic, Take 5 Mins ...
```

Python also has built in string methods to check whether strings are upper or lower case:

```
>>> "DON'T".isupper()
True

>>> "DON'T".islower()
False
```

Q: What is the Python string method to check if a string is numeric?

11. String Characters and Numeric Encodings

Python has BIFs for turning characters into ASCII codes and vice versa.

```
>>> print (ord('A'))      # Return the ASCII for the char A
65
>>> print (chr(65))       # Return the char for the ASCII code 65
A
```

Q: What are the ASCII codes for B, C & D?

Q: What are the characters represented by ASCII codes 64, 62, and 97?

12. Iteration of Strings

A string can be iterated through, with tasks performed on each character. A script could be written to print out each item of the string s:

```
>>> s = "Don't Panic and Code Some Python"
>>> for char in s:
    print (char)
```

Q: How many chars per line are printed?

Q: What could we use to print all chars on one line?

13. Password Strength Checker Script

Create a script in a file called **passwd_checker.py**. You can copy&paste the code from below to create the initial script, or download it from moodle.

Using the lecture/lab notes as reference, change the function `check_strength()`, to **check if a password has a length of more than 6 chars, and that it has at least one uppercase char, at least 3 numeric chars and also at least one lower case char.**

Add a line or two at a time, and run to test if syntax is ok using the test case password in main and other test cases, and check each change you make does what you expect.

Hint: The script is based around the `passwd_strong` Boolean variable, so this needs to be updated to False if any of the checks above are not passed.



Script starting point:

password_checker_start.py

(in Moodle, Week3)

The script should run without errors, but not do too much yet!

```
# Script:  passwd_checker_start.py
# Desc:    Check strength of a password.
# Author:  Rich Macf, Petra L
# modified: Sept 2017

import sys, math

def check_strength(passwd):
    """checks the strength of a password, printing the result"""
    print ('[*] passwd_checker')
    passwd_strong = True;
    # check password length
    if len(passwd) < 5:
        print ('Password must be at least 5 chars')
        passwd_strong = False;
        return

    # check password for upper and numeric chars
    hasupper = False;
    haslower = False;
    digitcount = 0;
    #
    # ... ADD YOUR CODE HERE ...
    #
    if passwd_strong:
        print (f'[*] Password {passwd} is strong')

def main():
    # test case
    passwd = 'PASWord123'
    # call strength checker function
    check_strength(passwd)

if __name__ == '__main__':
    main()
```

14. Caesar Cipher script

This exercise is the first part of the portfolio assessment. It is worth 25% of the module grade.

Save a copy of your finished script as a .txt file and upload this to moodle by 9am of the day of your lab 2 weeks after this sheet was meant to be done in the lab.

Assuming this sheet is the lab for Monday 2/10/17, that means that you need to **upload no later than 9am Monday 16 October 2017**.

You will be asked to demo your script during your lab on the 16 October, where your work and demo will be marked according to the rubric shown at the end of this exercise.

Feedback will be given verbally during the demo; marks will be entered in moodle.

Create a module called **caesar.py**. You can copy&paste the code from the link below to create the initial script or download it from moodle.

The script should run without errors, but not do too much yet!



Script starting point:

caesar_start.py

(in Moodle, Week3)

```
# Script:  caesar.py
# Desc:    encrypt and decrypt text with a Caesar cipher
#          using defined character set with index
# Author:  your name(s). Based on a template
# Created: 23/9/17
# note that you should add a module doc string!

charset="ABCDEFGHIJKLMNOPQRSTUVWXYZ" # characters to be encrypted
numchars=len(charset) # number of characters, for wrapping round

def caesar_encrypt(plaintext,key):
    """put an appropriate function doc string here"""
    print (f'[*] ENCRYPTING - key: {key}; plaintext: {plaintext}')

    # plaintext=      # convert plaintext to upper case
    ciphertext=''     # initialise ciphertext as empty string

    for ch in plaintext:
        if ch in charset:
            new='' # replace this with your code, may use extra lines
        else:
            new=ch # do nothing with characters not in charset
        ciphertext=ciphertext+new
    print (f'[*] ciphertext: {ciphertext}')
    return ciphertext # returns ciphertext so it can be reused

def caesar_decrypt(ciphertext,key):
    """put an appropriate function doc string here"""
```

```

    # very similar to caesar_encrypt(), but shift left
    print (f'[*] DECRYPTING - key: {key}; ciphertext: {ciphertext}')
    #
    plaintext=''    # replace this with your code
    #
    print (f'[*] plaintext: {plaintext}')
    return plaintext # returns plaintext so it can be reused

def caesar_crack(ciphertext):
    """put an appropriate function doc string here"""
    # how could you brute force crack a caesar cipher?
    # your code here

def main():
    # test cases
    key=2
    plain1 = 'Hello Suzanne'
    cipher1 = 'IQQfOQtPkpIGXGtaQPG'
    crackme = 'PBATENGHYNGVBAFLBHUNIRPENPXRQGURPBQRNAQGURFUVSGJNFGUVEGRRA'
    # call functions with test cases
    caesar_encrypt(plain1, key)
    caesar_decrypt(cipher1, key)
    # caesar_crack(crackme) # remove comment to test cracking

# boilerplate
if __name__ == '__main__':
    main()

```

Your task is to complete the script appropriately so that the functions work as expected. Use the lecture notes to help you get started.

Ideally, any spaces should also be removed from the string during encryption.

As implementation of the cracking function is marked separately, it is recommended that you focus on encryption and decryption initially.

To gain more than 32 marks out of 40 (80%), you will need to implement additional features that demonstrate Python skills you gained by doing your own research. Discuss your ideas for these with your tutor(s) in the lab. Some example suggestions are listed below.

Your work will be marked according to the rubric overleaf. The rubric contains typical descriptors for categories such as "good/very good". However, you may be given intermediate marks if your work is between the descriptions.

It's perfectly ok to use and adapt code used from external sources (e.g. stackoverflow), provided you have not directly asked someone else to do the work on your behalf. As always, you must fully attribute any such code - failure to do this could constitute academic deceit.

Marking will take place only when you demo your work in class. Failure to do a demo will result in 0 marks for this exercise.

Make sure that you can easily run your script with precisely the test cases given in main() of the starting script for demo purposes.

The marks for this exercise will be aggregated with the marks for the class test (later in the module) to give you an overall portfolio mark. The Portfolio coursework as a whole constitutes 50% of the marks for the module, so this exercise is worth 25%.

You are encouraged to do this task in pairs. The rules for this are:

- Add both names as the script author
- Both members of the pair must upload the same script to moodle
- You must do the demo together, and both demonstrate that you each fully understand the script. In this case you will get the same marks. If one of you misses the demo, or cannot answer questions, this will result in different marks being given.

Criteria	Excellent 7-8 Points each	Good/very good 5-6 Points each	Acceptable 3-4 points each	Poor 0-2 points each
Encrypt and decrypt	Both functions work as expected and spaces are stripped out.	Both functions work correctly, but spaces not stripped or other minor issues.	One function works correctly, or both work but with issues.	Neither function works - Substantial functionality missing.
Cracking	Function works fully, uses decrypt function, and gives clean output similar to lecture example.	Function works, but minor issues such as convoluted output.	Function partially working.	Cracking function not attempted.
Additional Features	Useful, complex additional feature(s) that substantially enhance functionality and demonstrate own research	Useful additional feature(s) that somewhat enhance functionality and demonstrate own research	Some minor Additional features attempted.	No additional features
Read-ability	Excellent, full commenting and logical and efficient structure. Any code from external sources fully attributed	Code tidy and easy to read. Some comments.	Either poor structure or very few or no annotations.	Poor structure and no commenting. Code from external sources not attributed
demo	Excellent understanding. Answers questions fully and confidently. Script runs by default with prescribed test cases.	Answers questions well. Acceptable understanding of more complex code.	Understands core constructs but struggles to answer questions.	Little understanding of code demonstrated.

Example suggestions for additional features:

NOTE: These are some suggestions. You can come up with your own ideas and YOU ARE NOT EXPECTED TO IMPLEMENT ALL OF THESE! Please discuss your plans with a lab helper!

- Encryption and decryption with the Caesar cipher use effectively the same algorithm, except that they shift in the opposite direction. So, this could be implemented using a single function that takes an extra parameter describing the mode. You could replace the two separate functions with this.
- Add a menu system to main so that the user can e.g. input the string and key at runtime, and choose whether to encrypt or decrypt. (Remember that it should remain possible to run the script easily with the prescribed test cases, for demo purposes.)

- Read up on PEP8, which is the de-facto code style guide for Python. A high quality, easy-to-read version of PEP 8 is available at pep8.org. Then follow this style in your code.
- Use exception handling to deal with any errors at runtime – for example, to deal with text containing characters that are not in the predefined charset. (exception handling required for CW2 but here an additional feature as we haven't covered it in class yet).
- Inspect the cracked strings automatically (e.g. using a dictionary) to select or highlight the ones that are more likely correct. E.g. "PYTHON" would be more likely correct than "YHCQXW" or "XGBPWW".
- Show the time taken to execute the cracking function.
- If you really really want a very tough nut to crack you could try putting spaces back into the cracked text, but this is practically impossible to do automatically so that it works without fail.
- Any additional suggestions will be posted in the moodle forum.

15. References

Read <http://stackoverflow.com/questions/419163/what-does-if-name-main-do> to find out more about reasons for using boiler plate code.

How to use extended slices <https://docs.python.org/2/whatsnew/2.3.html#extended-slices>

f-strings are a new feature introduced for Python 3.6. Find out more by reading about PEP 498, Literal String Formatting (f-strings) at <https://docs.python.org/3/whatsnew/3.6.html> and https://docs.python.org/3/reference/lexical_analysis.html#f-strings. The full specification is at <https://www.python.org/dev/peps/pep-0498/>.