Assignment 1: Currency

(Source: Deptt. Of C.S., University of Cornell)

Submission deadline: November 30, 2022

Thinking about that trip overseas? Whenever possible, it is best to go when the exchange rate is in your favor. When your dollars buy more in the foreign currency, you can do more on your vacation. This is why it would be nice to have a tool that could tell you how much your cash is worth in another currency.

However, there is no set mathematical formula to compute this conversion. The value of a currency (with respect to another) is constantly changing. In fact, in the time that it takes you to read this paragraph, the exchange rate between the dollar and the Euro has probably changed several times. So how do we write a program to handle something like that?

One solution is to make use of a web service. A web service is a program that, when you send it a request, automatically generates a web page with the information that you asked for. In our case, the web service will tell us the current exchange rate for most of the major international currencies. Your job will be to use string-manipulation methods to read the web page and extract the exact information we need. Full instructions are included below.

Learning Objectives

This assignment is designed to give you practice with the following skills:

- How to write a self-contained module in Python
- How to write a script that uses a self-contained module
- How to use string methods in Python

- How to connect Python to a web service
- How to read specifications and understand preconditions
- How to use docstrings appropriately for specifications
- How to follow the coding conventions for this course
- How to test a program using doctests

Academic Integrity

This assignment is supposed to be attempted individually. DO NOT copy. Strict action will be taken against plagiarism. There will be a viva after the assignment submission to ensure that the solution submitted by you was your own. However, you will have a couple of tutorial sessions where you may clarify your doubts (and possibly ask for clues) with the instructor.

Assignment Scope

Everything that you need to complete this assignment should have been covered by Lessons 7 and 8 (Specifications and Testing) in class. In particular, **you may not use if-statements anywhere in this assignment**, as they are not necessary. Submissions containing if-statements will be returned for you to revise. Similarly, students with prior programming experience **should not try to use loops or recursion**.

Grading Policy

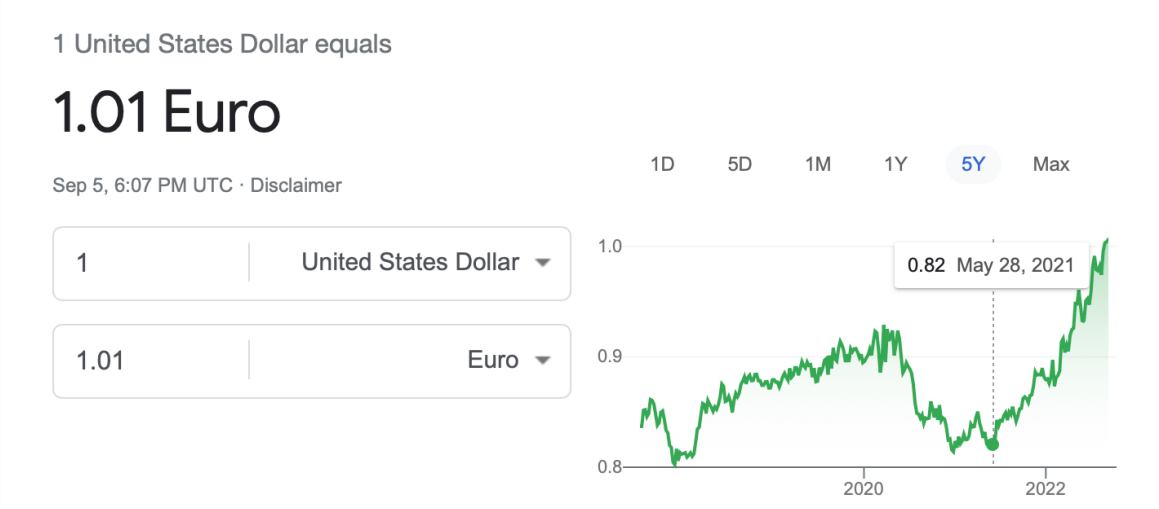
In grading your code, we will focus on the following issues in order:

- Correct function specifications and/or formatting*
- Adequate test cases
- Correct implementations (does it pass our test cases?)

*Formatting has to be done as per the conventions given at the end of the document.

The Currency Exchange Web Service

Before you do anything at all, you should play around with the currency exchange web service. You do not need any Python to do this; just a web browser.



For this assignment, you will use a simulated currency exchange service that never changes values. This is important for **testing in Part C**. If the answer is always changing, it is hard to test that you are getting the right answers. To use the service, you connect your web browser to the following address:

```
http://cs1110.cs.cornell.edu/2022fa/a1?
```

This prefix is followed by a currency query. A currency query has three pieces of information in the following format (without spaces; we have included spaces here solely for readability):

```
old=currency1 & new=currency2 & amt=value
```

where currency1 is a three-letter code for the original currency, currency2 is a three-letter code for the new currency and value is a float value for the amount of

money in currency1. For example, if you want to know the value of 2.5 dollars (USD) in Cuban Pesos (CUP), the query is:

```
old=USD&new=CUP&amt=2.5
```

The query is not enough by itself. To use it, you have to make it part of a web page URL. The full URL for this query combines the prefix and the query together, like this

```
http://cs1110.cs.cornell.edu/2022fa/a1?old=USD&new=CUP&amt=2.5
```

Click on the link to see it in action.

You will note that the "web page" in your browser is just a single line in the following format:

```
{ "lhs" : "2.5 United States Dollars", "rhs" : "64.375 Cuban Pesos", "err" : "" }
```

This is what is known as a **JSON** representation of the answer. JSON is a way of encoding complex data so that it can be sent over the Internet. You will use what you know about string operations and methods to pull out the relevant data out of the JSON string.

You should try a few more currency queries to familiarize yourself with the service. Note that if you enter an invalid query (for example, using a non-existent currency code like `AAA`), you will get the following response in error:

```
{ "lhs" : "", "rhs" : "", "err" : "Source currency code is invalid." }
```

Similarly, if you enter a query with two valid currency codes, but with an invalid quantity value, you will get the following error:

```
{ "lhs" : "", "rhs" : "", "err" : "Currency amount is invalid." }
```

For all error responses, the `lhs` and `rhs` values are blank, while the value `err` is a specific error message describing the problem. If `err` is blank, then there was no error. This will be important for error handling in this assignment.

Currency Application

Your primary goal in this assignment is to write an interactive application that queries the user for input and responds as follows:

```
[user@machine]:a1 > python a1app.py
Enter original currency: USD
Enter desired currency: EUR
Enter original amount: 2.5
You can exchange 2.5 USD for 2.517745 EUR.
```

To create this application, you will create two files:

- `a1.py` : a module with a collection of functions to perform the calculation.
- `a1app.py` : a script to receive user input and provide an answer.

Of the two files, `a1app.py` will be the easier. It will be no more complex than the scripts you wrote for the lab. The vast majority of the work will be in the file `a1.py`. This file will contain a definition for following function:

```
def exchange(old, new, amt):
    """
    Returns the amount of currency received in the given
exchange. In this exchange, the user is changing amt money in
currency old to the currency new. The value returned represents
the amount in currency new.

    The value returned has type float.

    Parameter old: the currency on hand
    Precondition: old is a string for a valid currency code
        Parameter new: the currency to convert to
    Precondition: new is a string for a valid currency code

    Parameter amt: amount of currency to convert
    Precondition: amt is a float
    """
```

This function will involve several steps. You will get the JSON string from the web service, break up the string to pull out the numeric value (as a substring), and then convert that substring to a float. As this is the very first assignment, we are going to take you through this process step-by-step. **However, not every function that we ask you to implement will be used by** `exchange`. The functions in a1.py will be tested using doctests. As told earlier, the exchange rates on the server are fixed so that they never change, making it possible for you to design test cases.

This assignment might feel like you are working in reverse. You will write the functions to break up the string first, and the functions to interact with the web service last. This is because we want you to develop the following programming habit: *always complete and test the helper functions before finishing the functions that use them*.

Part I: The Assignment Files

One of the most important outcomes of this assignment is that you understand the importance of testing. This assignment will follow an *iterative development cycle*. That means you will write a few functions, then fully test them before you write any more. This process makes it easier to find bugs; you know that any bugs must have been part of the work you did since the last test.

Iterative development requires that you write doctests for a function in `a1.py`, implement the function and then move to writing doctests for another function followed by its implementation. To get started, first create both the files. The very first thing you should do is specifically create a directory for this assignment. This folder should contain the two assignment files and nothing else.

The Module `a1`

In your newly created directory, you should create the module `a1` (with file name `a1.py`). This will be the main module for this assignment. Following the **style**

guidelines your file should start with a descriptive docstring, and the last two lines should be (1) the name and (2) the date the file was last edited. This is the docstring that we would like you to use:

```
"""
Module for currency exchange

This module provides several string parsing functions to implement a
simple currency exchange routine using an online currency service.
The primary function in this module is exchange.

Author: YOUR NAME HERE
Date:   THE DATE COMPLETED HERE
"""
```

Cut-and-paste this docstring into a1, making sure to insert your name and date as appropriate.

Part II: The Core Functions

Iterative Development

In this part of the assignment, you will work on the file `a1.py`. For each function in this file, you will do the following:

Add a function stub to `a1`

We will give you the function header to write. We will also give you a detailed docstring specification for the function. You should copy-and-paste the specification into the function body, indented.

Add test cases. Yes, this means you are writing doctests before writing the function body. We talked about this in the **lesson on testing**. Furthermore, your tests should be representative (you may revise the concept from the classroom slides).

Write the function body

Make sure that the function satisfies the specifications exactly. If the specification says to return something, you need a return statement. Make sure that the value returned is of the correct type.

Test the function.

If errors are found, fix them and re-test. Keep doing this until no more errors are found.

Function Specifications

The descriptions that we provide in each part below represent the level of completeness and precision we are looking for in your docstring comments. In fact, it is best to copy-and-paste these descriptions to create the first draft of your docstring comments. If you do not cut and paste, please adhere to the conventions we use, such as using a single line, followed by a blank line and a more descriptive paragraph, or by using "Returns …" for fruitful-functions. While we have provided the contents of the specification, we have not always formatted them properly for you.

If you want to see if your specifications are written correctly, start an interactive Python shell and type

```
>>> import a1
>>> help(a1)
```

This should list all the functions with their specifications.

Part A: Breaking Up Strings

A large part of this assignment is breaking up a JSON string. Conceptually, you want to separate the currency amount from the currency name. For example, if we are given the string `"4.502 Euros"`, then we want to break it up into `"4.502"` and `"Euros"` .

This is the motivation for the two functions below. The implementation of these functions should be relatively simple.

`before_space(s)`

```
Returns a copy of s up to, but not including, the first space

Parameter s: the string to slice
Precondition: s is a string with at least one space
```

`after_space(s)`

```
Returns a copy of s after the first space

Parameter s: the string to slice
Precondition: s is a string with at least one space
```

Implement these functions according to their specification. In other words,

- Write the header and specification in `a1.py`.
- Place test cases in the form of doctests.
- Implement the functions in `a1.py`.
- Test for and correct errors until no errors remain.

When you think about what test cases you want to include, consider the following:

- Does the specification allow for strings with more than one space?
- Does the specification allow for strings that start with a space?

- Does the specification allow for strings that don't have any spaces?

When you write your test cases **write a single-line comment (#) just above the test case explaining what it is testing**. If you make it clear what you are testing, this can help us provide better feedback when we return the graded assignment.

Part B: Processing a JSON String

All of the responses to a currency query, whether valid or invalid, contain the keywords `"lhs"` and `"rhs"`. If it is a valid currency query, then the answer is in quotes after the keyword `"rhs"`. If it is invalid, then the quotes after `"rhs"` are empty. Hence the next step is to extract the information in quotes after these keywords.

While working on each of the functions below, remember to write doctests before implementing the body. You should thoroughly test each function before implementing the next one.

`first_inside_quotes(s)`

```
Returns the first substring of s between two (double) quotes

A quote character is one that is inside a string, not one that
delimits it.  We typically use single quotes (') to delimit a
string if we want to use a double quote character (") inside of
it.

Examples:
first_inside_quotes('A "B C" D') returns 'B C'

first_inside_quotes('A "B C" D "E F" G') returns 'B C',

because it only picks the first such substring
```

```
Parameter s: a string to search

Precondition: s is a string containing at least two double quotes
```

Once you have this function completed, you should move on to the following functions.

`get_lhs(json)`

```
Returns the lhs value in the response to a currency query

Given a JSON response to a currency query, this returns the

string inside double quotes (") immediately following the keyword

"lhs". For example, if the JSON is

'{ "lhs" : "1 Bitcoin", "rhs" : "19995.85429186 Euros", "err" : "" }'

then this function returns '1 Bitcoin' (not '"1 Bitcoin"').

This function returns the empty string if the JSON response

contains an error message.

Parameter json: a json string to parse

Precondition: json is the response to a currency query
```

`get_rhs(json)`

Returns the rhs value in the response to a currency query

Given a JSON response to a currency query, this returns the

string inside double quotes (") immediately following the keyword

"rhs". For example, if the JSON is

```
'{ "lhs" : "1 Bitcoin", "rhs" : "19995.85429186 Euros", "err" : "" }'
```

then this function returns '19995.85429186 Euros' (not

'"38781.518240835 Euros"').

This function returns the empty string if the JSON response contains an error message.

Parameter json: a json string to parse

Precondition: json is the response to a currency query

has_error(json)

Returns True if the query has an error; False otherwise.

Given a JSON response to a currency query, this returns True if there

is an error message. For example, if the JSON is

```
'{ "lhs" : "", "rhs" : "", "err" : "Currency amount is invalid." }'
```

```
then the query is not valid, so this function returns True (It
does NOT return the message 'Currency amount is invalid.').
```

```
Parameter json: a json string to parse
Precondition: json is the response to a currency query
```

Note that three of the functions above have **json is the response to a currency query** in the precondition. This just means that your test cases should only consider strings that can be returned by the server. So the string

```
'{ "lhs":"2 Namibian Dollars", "rhs":"2 Lesotho Maloti",
"err":"" }
```

is okay to test, but the string

```
' "lhs" : "2 Namibian Dollars", "rhs" : "2 Lesotho Maloti",
"err" : "" '
```

is not. Again, **write a single-line comment (#) just above each test case explaining what it is testing**. This will help us give you better feedback.

You should **not use a conditional statement to implement these functions** (except for the one place in a1app.py where it has been used in the already written code provided to you). Simply find the position of the appropriate keyword and extract the value in quotes immediately after it. Your implementation must make use of the `find` or `index` string methods.

Part C: Contacting the Server

Now it is time to interact with the web service. In this part, you will implement a single function. Do remember to write the test cases (doctests) for this function properly.

`query_website(old, new, amt)`

```
Returns a JSON string that is a response to a currency query.

A currency query converts amt money in currency old to the

currency new. The response should be a string of the form


'{ "lhs":"<old-amt>", "rhs":"<new-amt>", "err":"" }'


where the values old-amount and new-amount contain the value

and name for the old and new currencies. If the query is

invalid, both old-amount and new-amount will be empty, while

"err" will have an error message.

Parameter old: the currency on hand
Precondition: old is a string with no spaces or non-letters

 Parameter new: the currency to convert to

Precondition: new is a string with no spaces or non-letters

 Parameter amt: amount of currency to convert

Precondition: amt is a float
```

The way to implement this function is as follows: import the module named `requests`.

Thereafter use the following line of code to get the required json object and convert it to text:

`json = (requests.get(target_url)).text`

Here `target_url` is of the form given below (as discussed earlier):

`http://cs1110.cs.cornell.edu/2022fa/a1?old=USD&new=CUP&amt=2.5`

`requests.get(target_url)` returns the json response, which we convert to text format to be able to apply various string operations on it.

When you test this function, you need to ensure that it returns exactly the right JSON string for the value given. The best way to test this is to use a web browser to manually get the right JSON answer. For example, one test case can be constructed by seeing the result of going to the URL

`http://cs1110.cs.cornell.edu/2022fa/a1?old=USD&new=CUP&amt=2.5`

You can copy the value from this webpage into the test case expected output (**but do not forget to add quotes**). Then check that the function returns the same JSON string. Remember to be thorough with your choice of test cases. One is not enough.

Part D: Computing the Exchange

We are now ready for the final part of the assignment. Implement the following specifications, again using our test-case-before-function-body approach.

`is_currency(code)`

```
Returns: True if code is a valid (3 letter code for a) currency

It returns False otherwise.


Parameter code: the currency code to verify

Precondition: code is a string with no spaces or non-letters.
```

In implementing `is_currency`, you must use `query_website` and `has_error` as helper functions. However, you may use the **table of currencies** in determining correct answers for your test cases (note that table of currencies is a hyperlink).

`exchange(old, new, amt)`

```
Returns the amount of currency received in the given exchange.


In this exchange, the user is changing amt money in currency

old to the currency new. The value returned represents the

amount in currency new.

The value returned has type float.


Parameter old: the currency on hand

Precondition: old is a string for a valid currency code

Parameter new: the currency to convert to
```

```
Precondition: new is a string for a valid currency code

Parameter amt: amount of currency to convert

Precondition: amt is a float
```

You may also use the table of currencies to craft some test cases for the function `exchange`. However, you might find it easier to use a currency query URL to look up the correct answer, and then paste the answer into your test case.

A bigger issue with testing `exchange` is that problem that we saw in class: real numbers cannot be represented exactly. This creates problems when you try to test equality between floats. We therefore allow you to avoid writing test cases for exchange for the time being.

Finally, bear in mind that, like `query_website`, these functions connect to the web service, and so are not instantaneous. In our solution, with complete test procedures for everything, it can take a while to run the unit test.

Part III: The Application Script

You are essentially done. The last part of the assignment is to complete the file `a1app.py`. Given below is a skeleton of the file contents. Some lines of code are simply to be copied and pasted in your script.

```
import a1

***Receive the required data from the user***
```

```
# DO NOT modify the following code
# if the source currency is not valid, quit
if(not(a1.is_currency(old))):
        print(old," is not a valid currency")
        quit()

# if the target currency is not valid, quit
if(not(a1.is_currency(new))):
        print(new," is not a valid currency")
        quit()
```

Compute the amount of currency received in the given exchange and print it

It is important to import the module `a1` for two reasons:

- Function `is_currency(code)` defined in `a1` is being called in the already written code
- You will need to access the `exchange` function in this script. This is the only function of `a1.py` that you will need to use (besides `is_currency`). Everything else will be either an `input` function, a `print` function, or some other built-in function.

The script should prompt the user and provide an answer, as shown below.

```
[user@machine]:a1 > python a1app.py
Enter original currency: USD
Enter desired currency: CUP
```

```
Enter original amount: 2.5
You can exchange 2.5 USD for 64.375 CUP.
```

Obviously you will get different answers for different currencies and amounts. But otherwise, the output displayed must look exactly like it does above. You must use those exact words for your final print statement and end with a period.

Formatting conventions

Tabs vs. Spaces

Python will allow you to indent with either tabs or spaces. However, beginning programmers should *always indent with spaces*. And whatever you do, you should absolutely **never mix tabs and spaces* (doing so will cause your program to crash). Always use *4 spaces for each indentation level*.

Blank Lines

Because Python is whitespace significant, blank lines should be used sparingly and to give the most emphasis to code divisions. The following important guideline should be observed: **Functions** should be separated by two blank lines. In addition, you may use blank lines to separate logical sections of code within a function or method, but this should be done sparingly.

Import Statements

Imports should usually be on separate lines. The following code

```
   import os
    import sys
```

is preferable to

```
   import sys, os
```

However, it is okay to use commas when importing multiple items from a single module.

The following is acceptable:

```
   from subprocess import Popen, PIPE
```

The `from` keyword should be limited to imports that are used heavily within the current module, and which are guaranteed to not collide with any existing functions or variables.

Comment Conventions

Python has two types of comments: single line comments (which start with a # sign) and docstrings (which are enclosed in triple quotes). The following is a general rule regarding commenting:

Specifications are docstrings; all other comments are single line comments.

You will see a lot of Python code that ignores this guideline. Do not emulate that code.

Specifications

Do as discussed in class (for both functions and modules).

Statement Comments

Single-line comments are never required. While they help make the code more maintainable, they do not serve the same software engineering role that specifications do. With that said, they do serve a useful role as *statement comments*.

Just as essays are grouped in paragraphs, code should be grouped into logical units. A good unit should be one that can be described in a **single sentence**, and that sentence should be written as a single-line comment at the start of the unit. If a unit of code requires more than a sentence to describe, that means it is complicated enough to pull out as a separate function (with specification).

Here is an example of a code unit with a statement-comment:

```
    # Ensure that x >= y

    if (x < y):
        tmp = x
```

```
        x = y
        y = tmp
```

Note that a statement comment should explain what the group of statements does, not how it does it.

Each time you start a new unit or block of code, you should add a new statement comment. Here is an extended example with several statement comments.

```
   # Eliminate whitespace from the beginning and end of t
    while (len(t) != 0 and t[0].isspace()):
        t= t[1:]


    # If t is empty, print an error message and return
    if (len(t) == 0):
        ...
        return False;


    # If t is a proper name, print a greeting.
    if (contains_capitals(t)):
        ...


    # Store the French translation of t in t_french
```