main

September 25, 2025

1 Part 0 of 2: Simple string processing review

Exercise ordering: Each exercise builds logically on previous exercises, but you may solve them in any order. That is, if you can't solve an exercise, you can still move on and try the next one. Use this to your advantage, as the exercises are **not** necessarily ordered in terms of difficulty. Higher point values generally indicate more difficult exercises.

Demo cells: Code cells starting with the comment `### define demo inputs` load results from prior exercises applied to the entire data set and use those to build demo inputs. These must be run for subsequent demos to work properly, but they do not affect the test cells. The data loaded in these cells may be rather large (at least in terms of human readability). You are free to print or otherwise use Python to explore them, but we did not print them in the starter code.

Debugging your code: Right before each exercise test cell, there is a block of text explaining the variables available to you for debugging. You may use these to test your code and can print/display them as needed (careful when printing large objects, you may want to print the head or chunks of rows at a time).

Exercise point breakdown:

- Exercise 0: **1** point
- Exercise 1: **2** points
- Exercise 2: **2** points
- Exercise 3: **3** points

Final reminders:

- Submit after **every exercise**
- Review the generated grade report after you submit to see what errors were returned
- Stay calm, skip problems as needed, and take short breaks at your leisure

```
In [140]: ### Global Imports

          # Use this cell to import anything common, e.g. numpy, pandas, sqlite3
          import string

          # Use this cell to bring in the starter data
```

```
In [141]: text = "sgtEEEr2020.0"
In [142]: # Strings have methods for checking "global" string properties
          print("1.", text.isalpha())

          # These can also be applied per character
          print("2.", [c.isalpha() for c in text])
```

```
1. False
2. [True, True, True, True, True, True, True, False, False, False, False, False, False]
```

```
In [143]: # Here are a bunch of additional useful methods
          print("BELOW: (global) -> (per character)")
          print(text.isdigit(), "-->", [c.isdigit() for c in text])
          print(text.isspace(), "-->", [c.isspace() for c in text])
          print(text.islower(), "-->", [c.islower() for c in text])
          print(text.isupper(), "-->", [c.isupper() for c in text])
          print(text.isnumeric(), "-->", [c.isnumeric() for c in text])
```

```
BELOW: (global) -> (per character)
False --> [False, False, False, False, False, False, False, True, True, True, True, False, Tru
False --> [False, False, False, False, False, False, False, False, False, False, False, False,
False --> [True, True, True, False, False, False, True, False, False, False, False, False, Fal
False --> [False, False, False, True, True, True, False, False, False, False, False, False, Fa
False --> [False, False, False, False, False, False, False, True, True, True, True, False, Tru
```

Exercise 0 (1 point). Create a new function that checks whether a given input string is a properly formatted social security number, i.e., has the pattern, `XXX-XX-XXXX`, *including* the separator dashes, where each `X` is a digit. It should return `True` if so or `False` otherwise.

```
In [144]: ### Define demo inputs
          demo_str_ex0_0 = '832-38-1847'
          demo_str_ex0_1 = '832 -38 -  1847'
          demo_str_ex0_2 = '832-bc-3847'
          demo_str_ex0_3 = '832381847'
```

The demos included in the solution cell below should display the following output:

```
is_ssn('832-38-1847') -> True
is_ssn('832 -38 -  1847') -> False
is_ssn('832-bc-3847') -> False
is_ssn('832381847') -> False
```

```
In [145]: def is_ssn(s):
              ###
              ### YOUR CODE HERE
              if len(s) == 11:
                  if s[0:3].isdigit() and s[4:6].isdigit() and s[7:11].isdigit():
                      if s[3] == "-" and s[6] == "-":
```

```
                    return True # all of these if statements need to be true in order to
            
            return False # if true is not returned, we return false. If one of the three if
            ###

        ### demo function call
        is_ssn(demo_str_ex0_0)
```

```
Out[145]: True
```

The cell below will test your solution for Exercise 0. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [146]: ### test_cell_ex0

          from tester_fw.testers import Tester

          conf = {
              'case_file':'tc_0',
              'func': is_ssn, # replace this with the function defined above
              'inputs':{ # input config dict. keys are parameter names
                  's':{
                      'dtype':'str', # data type of param.
                      'check_modified':False,
                  }
              },
              'outputs':{
                  'output_0':{
                      'index':0,
                      'dtype':'bool',
                      'check_dtype': True,
                      'check_col_dtypes': True, # Ignored if dtype is not df
                      'check_col_order': True, # Ignored if dtype is not df
                      'check_row_order': True, # Ignored if dtype is not df
                      'check_column_type': True, # Ignored if dtype is not df
                      'float_tolerance': 10 ** (-6)
                  }
              }
          }
          tester = Tester(conf, key=b'F24oYNyh8kq_wkZD_0o0ZCPHLco0-xNXHOYNiPnQmmY=', path='reso
          for _ in range(70):
              try:
                  tester.run_test()
```

```
            (input_vars, original_input_vars, returned_output_vars, true_output_vars) = 
        except:
            (input_vars, original_input_vars, returned_output_vars, true_output_vars) = 
            raise


    print('Passed! Please submit.')
```

```
Passed! Please submit.
```

2 Regular expressions

Exercise 0 hints at the general problem of finding patterns in text. A handy tool for this problem is Python's Regular Expression module, `re`.

A *regular expression* is a specially formatted pattern, written as a string. Matching patterns with regular expressions has 3 steps:

1. You come up with a pattern to find.
2. You compile it into a *pattern object*.
3. You apply the pattern object to a string to find *matches*, i.e., instances of the pattern within the string.

As you read through the examples below, refer also to the regular expression HOWTO document for many more examples and details.

```
In [147]: import re
```

2.1 Basics

Let's see how this scheme works for the simplest case, in which the pattern is an *exact substring*. In the following example, suppose want to look for the substring `'fox'` within a larger input string.

```
In [148]: pattern = 'fox'
          pattern_matcher = re.compile(pattern)

          input_string = 'The quick brown fox jumps over the lazy dog'
          matches = pattern_matcher.search(input_string)
          print(matches)
```

```
<re.Match object; span=(16, 19), match='fox'>
```

Observe that the returned object, `matches`, is a special object. Inspecting the printed output, notice that the matching text, `'fox'`, was found and located at positions 16-18 of the `input_string`. Had there been no matches, then `.search()` would have returned `None`, as in this example:

```
In [149]: print(pattern_matcher.search("This input has a FOX, but it's all uppercase and so wo
None
```

You can also write code to query the `matches` object for more information.

```
In [150]: print(matches.group())
          print(matches.start())
          print(matches.end())
          print(matches.span())
```

```
fox
16
19
(16, 19)
```

Module-level searching. For infrequently used patterns, you can also skip creating the pattern object and just call the module-level search function, `re.search()`.

```
In [151]: matches_2 = re.search('jump', input_string)
          assert matches_2 is not None
          print ("Found", matches_2.group(), "@", matches_2.span())
```

```
Found jump @ (20, 24)
```

Other Search Methods. Besides `search()`, there are several other pattern-matching procedures:

1. `match()` - Determine if the regular expression (RE) matches at the beginning of the string.
2. `search()` - Scan through a string, looking for any location where this RE matches.
3. `findall()` - Find all substrings where the RE matches, and returns them as a list.
4. `finditer()` - Find all substrings where the RE matches, and returns them as an iterator.

We'll use several of these below; again, refer to the HOWTO for more details.

2.2 A pattern language

An exact substring is one kind of pattern, but the power of regular expressions is that it provides an entire "*mini-language*" for specifying more general patterns.

To start, read the section of the HOWTO on "Simple Patterns". We highlight a few constructs below.

```
In [152]: # Metacharacter classes
          vowels = '[aeiou]'

          print(f"Scanning `{input_string}` for vowels, `{vowels}`:")
          for match_vowel in re.finditer(vowels, input_string):
              print(match_vowel)
```

```
Scanning `The quick brown fox jumps over the lazy dog` for vowels, `[aeiou]`:
<re.Match object; span=(2, 3), match='e'>
<re.Match object; span=(5, 6), match='u'>
<re.Match object; span=(6, 7), match='i'>
<re.Match object; span=(12, 13), match='o'>
<re.Match object; span=(17, 18), match='o'>
<re.Match object; span=(21, 22), match='u'>
<re.Match object; span=(26, 27), match='o'>
<re.Match object; span=(28, 29), match='e'>
<re.Match object; span=(33, 34), match='e'>
<re.Match object; span=(36, 37), match='a'>
<re.Match object; span=(41, 42), match='o'>
```

```
In [153]: # Counts: For instance, two or more consecutive vowels:
          two_or_more_vowels = vowels + '{2,}'
          print(f"Pattern: {two_or_more_vowels}")
          print(re.findall(two_or_more_vowels, input_string))
```

```
Pattern: [aeiou]{2,}
['ui']
```

```
In [154]: # Wildcards
          cats = "ca+t"
          print(re.search(cats, "is this a ct?"))
          print(re.search(cats, "how about this cat?"))
          print(re.search(cats, "and this one: caaaaat, yes or no?"))
```

```
None
<re.Match object; span=(15, 18), match='cat'>
<re.Match object; span=(14, 21), match='caaaaat'>
```

```
In [155]: # Special operator: "or"
          adjectives = "lazy|brown"
          print(f"Scanning `{input_string}` for adjectives, `{adjectives}`:")
          for match_adjective in re.finditer(adjectives, input_string):
              print(match_adjective)
```

```
Scanning `The quick brown fox jumps over the lazy dog` for adjectives, `lazy|brown`:
<re.Match object; span=(10, 15), match='brown'>
<re.Match object; span=(35, 39), match='lazy'>
```

```
In [156]: # Predefined character classes
          three_digits = '\d\d\d'
          print(re.findall(three_digits, "My number is 555-123-4567"))
```

```
['555', '123', '456']
```

> In the previous example, notice that the pattern search proceeds from left-to-right and does not return overlaps: here, the matcher returns `456` but not `567`. In fact, this case is an instance of the default *greedy behavior* of the matcher.

The backslash plague. In the "three-digits" example, we used the predefined metacharacter class, `'\d'`, to match slashes. But what if you want to match a *literal* slash? The HOWTO describes how things can get out of control in its subsection on "The Backslash Plague", which occurs because the Python interpreter processes backslashes in string literals (e.g., so that `\t` expands to a tab character and `\n` to a newline) while the regular expression processor also gives backslashes meaning (e.g., so that `\d` is a digit metaclass).

For example, suppose you want to look for the text string, `\section`, in some input string. Which of the following will match it? Recall that `\s` is a predefined metacharacter class that matches any whitespace character.

```
In [157]: input_with_slash_section = "This string contains `\section`, which we would like to m

          print(f"Searching: {input_with_slash_section}")

          print(re.search("\section", input_with_slash_section))
          print(re.search("\\section", input_with_slash_section))
          print(re.search("\\\\section", input_with_slash_section))
```

```
Searching: This string contains `\section`, which we would like to match.
None
None
<re.Match object; span=(22, 30), match='\\section'>
```

To help mitigate this case, Python provides a special type of string called a *raw string*, which is a string literal prefixed by the letter `r`. For such strings, the Python interpreter will not process the backslash.

> Although the interpreter won't process the backslash, the regular expression processor will do so. As such, the pattern string still needs *two* slashes, as shown below.

```
In [158]: print(re.search(r"\section", input_with_slash_section))
          print(re.search(r"\\section", input_with_slash_section))
          print(re.search(r"\\\\section", input_with_slash_section))
```

```
None
<re.Match object; span=(22, 30), match='\\section'>
None
```

Indeed, it is common style to always use raw strings for regular expression patterns, as we'll do in the examples that follow.

Creating pattern groups. Another handy construct are *pattern groups*, as we show in the next code cell.

Suppose we have a string that we know contains a name of the form, "(first) (middle) (last)", where the middle name is *optional*. We can use pattern groups to isolate each component of the name and tag the middle name as optional using the "zero-or-one" metacharacter, `'?'`.

The group itself is a subpattern enclosed within parentheses. When a match is found, we can extract the groups by calling `.groups()` on the match object, which returns a tuple of all matched groups.

> To make this pattern more readable, we have also used Python's multiline string literal combined with the `re.VERBOSE` option, which then allows us to include whitespace and comments as part of the pattern string.

```
In [159]: # Make the expression more readable with a re.VERBOSE pattern
          re_names2 = re.compile(r'''^              # Beginning of string
                                     ([a-zA-Z]+)    # First name
                                     \s+            # At least one space
                                     ([a-zA-Z]+\s)? # Optional middle name
                                     ([a-zA-Z]+)    # Last name
                                     $              # End of string
                                  ''',
                                  re.VERBOSE)
          print(re_names2.match('Rich Vuduc').groups())
          print(re_names2.match('Rich S Vuduc').groups())
          print(re_names2.match('Rich Salamander Vuduc').groups())
```

```
('Rich', None, 'Vuduc')
('Rich', 'S ', 'Vuduc')
('Rich', 'Salamander ', 'Vuduc')
```

Tagging pattern groups. You can also name pattern groups, which helps make your extraction code a bit more readable.

```
In [160]: # Named groups
          re_names3 = re.compile(r'''^
                                     (?P<first>[a-zA-Z]+)
                                     \s
                                     (?P<middle>[a-zA-Z]+\s)?
                                     \s*
                                     (?P<last>[a-zA-Z]+)
                                     $
                                  ''',
                                  re.VERBOSE)
          print(re_names3.match('Rich Vuduc').group('first'))
          print(re_names3.match('Rich S Vuduc').group('middle'))
          print(re_names3.match('Rich Salamander Vuduc').group('last'))
```

```
Rich
S
```

A regular expression debugger. Regular expressions can be tough to write and debug, but thankfully, there are several online tools to help! See, for instance, regex101, pythex, regexr, or debuggex. These all allow you to supply some sample input text and test what your pattern does in real time.

2.3 Email addresses

In the next exercise, you'll apply what you've read and learned about regular expressions to build a pattern matcher for email addresses. Again, if you haven't looked through the HOWTO yet, take a moment to do that!

Although there is a formal specification of what constitutes a valid email address, for this exercise, let's use the following simplified rules.

- We will restrict our attention to ASCII addresses and ignore Unicode. If you don't know what that means, don't worry about it---you shouldn't need to do anything special given our code templates, below.
- An email address has two parts, the username and the domain name. These are separated by an `@` character.
- A username **must begin with an alphabetic** character. It may be followed by any number of additional *alphanumeric* characters or any of the following special characters: `.` (period), `-` (hyphen), `_` (underscore), or `+` (plus).
- A domain name **must end with an alphabetic** character. It may consist of any of the following characters: alphanumeric characters, `.` (period), `-` (hyphen), or `_` (underscore).
- Alphabetic characters may be uppercase or lowercase.
- No whitespace characters are allowed.

Valid domain names usually have additional restrictions, e.g., there are a limited number of endings, such as `.com`, `.edu`, and so on. However, for this exercise you may ignore this fact.

Exercise 1 (2 points). Write a function `parse_email` that, given an email address `s`, returns a tuple, `(user-id, domain)` corresponding to the user name and domain name.

For instance, given `richie@cc.gatech.edu` it should return `('richie', 'cc.gatech.edu')`.

Your function should parse the email only if it exactly matches the email specification. For example, if there are leading or trailing spaces, the function should *not* match those. See the test cases for examples.

If the input is not a valid email address, the function should raise a `ValueError`.

> The requirement, "raise a `ValueError`" refers to a technique for handling errors in a program known as *exception handling*. The Python documentation covers exceptions in more detail, including raising `ValueError` objects.

We have provided wrapper function `eif_wrapper`. This wrapper will capture whether or not your function raised a `ValueError` and the returned value from the function. Additionally, any `ValueErrors` raised by your function will not halt execution of the notebook.

`eif_wrapper` is provided for you in the cell below. The function inputs are `s`, the input to be evaluated, and `func` the function that can raise a `ValueError`.

```
In [161]: # ValueError wrapper
          def eif_wrapper(s, func):
              """
              Returns a (bool, function return) pair where the first element is True when a Va
              and False if a Value Error is not raised. The second output is the return value
              """
              raised_value_error = False
              result = None
              try:
                  result = func(s)
              except ValueError:
                  raised_value_error = True
              finally:
                  return (raised_value_error, result)
```

```
In [162]: ### Define demo inputs
          demo_str_ex1_list = ['richie@cc.gatech.edu',
                               'what-do-you-know+not-much@gmail.com',
                               'x @hpcgarage.org',
                               'richie@cc.gatech.edu7']
```

The demo included in the solution cell below should display the following output:

```
eif_wrapper('richie@cc.gatech.edu', parse_email) -> (False, ('richie', 'cc.gatech.edu'))
eif_wrapper('what-do-you-know+not-much@gmail.com', parse_email) -> (False, ('what-do-you-know+
eif_wrapper('x @hpcgarage.org', parse_email) -> (True, None)
eif_wrapper('richie@cc.gatech.edu7', parse_email) -> (True, None)
```

```
In [163]: ### Exercise 1 solution
          def parse_email(s):
              """Parses a string as an email address, returning an (id, domain) pair."""
              ###
              ### YOUR CODE HERE
              re_emails = re.compile(r'''^
                                     ([a-zA-Z][a-zA-Z0-9._+\-]*)  # first character must begin
                                        @ # at symbol. But why would this work?
                                     ([a-zA-Z0-9._-]*
                                      [a-zA-Z])
                                      $
                                      ''',
                                    re.VERBOSE)
              ###

              match = re_emails.match(s)
              if not match:
                  raise ValueError
              else:
                  return match.group(1), match.group(2) # returns a tuple with two values.
```

```
### demo function call
for demo_str_ex1 in demo_str_ex1_list:
    print(f"eif_wrapper({demo_str_ex1}, parse_email) -> {eif_wrapper(demo_str_ex1, pa
```

```
eif_wrapper(richie@cc.gatech.edu, parse_email) -> (False, ('richie', 'cc.gatech.edu'))
eif_wrapper(what-do-you-know+not-much@gmail.com, parse_email) -> (False, ('what-do-you-know+no
eif_wrapper(x @hpcgarage.org, parse_email) -> (True, None)
eif_wrapper(richie@cc.gatech.edu7, parse_email) -> (True, None)
```

The cell below will test your solution for Exercise 1. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [164]: ### test_cell_ex1
          from tester_fw.testers import Tester

          conf = {
              'case_file':'tc_1',
              'func': lambda s: eif_wrapper(s, parse_email), # replace this with the function
              'inputs':{ # input config dict. keys are parameter names
                  's':{
                      'dtype':'str', # data type of param.
                      'check_modified':False,
                  }
              },
              'outputs':{
                  'output_0':{
                      'index':0,
                      'dtype':'bool',
                      'check_dtype': True,
                      'check_col_dtypes': True, # Ignored if dtype is not df
                      'check_col_order': True, # Ignored if dtype is not df
                      'check_row_order': True, # Ignored if dtype is not df
                      'check_column_type': True, # Ignored if dtype is not df
                      'float_tolerance': 10 ** (-6)
                  },
                  'output_1':{
                      'index':1,
                      'dtype':'NoneType',
                      'check_dtype': True,
                      'check_col_dtypes': True, # Ignored if dtype is not df
                      'check_col_order': True, # Ignored if dtype is not df
                      'check_row_order': True, # Ignored if dtype is not df
```

```
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                }
            }
        }
        tester = Tester(conf, key=b'F24oYNyh8kq_wkZD_Oo0ZCPHLco0-xNXHOYNiPnQmmY=', path='res
        for _ in range(70):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = 
            except:
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = 
                raise

        print('Passed! Please submit.')
```

```
Passed! Please submit.
```

2.4 Phone numbers

Exercise 2 (2 points). Write a function to parse US phone numbers written in the canonical "(404) 555-1212" format, i.e., a three-digit area code enclosed in parentheses followed by a seven-digit local number in three-hyphen-four digit format. It should also **ignore** all leading and trailing spaces, as well as any spaces that appear between the area code and local numbers. However, it should **not** accept any spaces in the area code (e.g., in '(404)') nor should it in the seven-digit local number.

For example, these would be considered valid phone number strings:

```
'(404) 121-2121'
'(404)121-2121      '
'   (404)       121-2121'
```

By contrast, these should be rejected:

```
'404-121-2121'
'(404)555 -1212'
' ( 404)121-2121'
'(abc) 555-12i2'
```

It should return a triple of strings, (`area_code`, `first_three`, `last_four`).
If the input is not a valid phone number, it should raise a `ValueError`.
The same wrapper function `eif_wrapper` from Exercise 1 will be used for evaluation.

```
In [165]: ### Define demo inputs
          demo_str_ex2_list = ['(404) 121-2121',
                               '404-121-2121']
```

The demo included in the solution cell below should display the following output:

```
eif_wrapper('(404) 121-2121', parse_phone1) -> (False, ('404', '121', '2121'))
eif_wrapper('404-121-2121', parse_phone1) -> (True, None)
```

Recall that `eif_wrapper` returns two outputs: - `bool` indicating whether or not your function raised a `ValueError` - `tuple` (the result of your parsing function) or `None` (when the parsing function raises an error)

```
In [166]: def parse_phone1(s):
              ###
              ### YOUR CODE HERE
              pattern = r'''
              \s*
              \((?P<areacode>\d{3})\)
              \s*
              (?P<local3>\d{3})
              -
              (?P<local4>\d{4})
              \s*
              '''
              ###

              matcher = re.compile(pattern, re.VERBOSE)
              matches = matcher.match(s) # we match our s, input string to our pattern (in mat

              if not matches:
                  raise ValueError
              else:
                  return (matches.group("areacode"), matches.group("local3"), matches.group("l



          print(parse_phone1('(291)255-4041 '))

          ### demo function call
          for demo_str_ex2 in demo_str_ex2_list:
              print(f"eif_wrapper('{demo_str_ex2}', parse_phone1) -> {eif_wrapper(demo_str_ex2
```

```
('291', '255', '4041')
eif_wrapper('(404) 121-2121', parse_phone1) -> (False, ('404', '121', '2121'))
eif_wrapper('404-121-2121', parse_phone1) -> (True, None)
```

The cell below will test your solution for Exercise 2. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]: ### test_cell_ex2
        from tester_fw.testers import Tester

        conf = {
            'case_file':'tc_2',
            'func': lambda s: eif_wrapper(s, parse_phone1), # replace this with the function d
            'inputs':{ # input config dict. keys are parameter names
                's':{
                    'dtype':'str', # data type of param.
                    'check_modified':False,
                }
            },
            'outputs':{
                'output_0':{
                    'index':0,
                    'dtype':'bool',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                },
                'output_1':{
                    'index':0,
                    'dtype':'NoneType',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                }
            }
        }
        tester = Tester(conf, key=b'F24oYNyh8kq_wkZD_OoOZCPHLcoO-xNXHOYNiPnQmmY=', path='resou
        for _ in range(70):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
            except:
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
                raise

        print('Passed! Please submit.')

In [ ]: print(input_vars)
        print(original_input_vars)
        print(returned_output_vars)
        print(true_output_vars)
```

Exercise 3 (3 points). Implement an enhanced phone number parser that can handle any of these patterns.

- (404) 555-1212
- (404) 5551212
- 404-555-1212
- 404-5551212
- 404555-1212
- 4045551212

As seen in the examples above, this parser should also handle the cases in which area code is not enclosed in parentheses. As before, it should not be sensitive to leading or trailing spaces. Also, for the patterns in which the area code is enclosed in parentheses, it should not be sensitive to the number of spaces separating the area code from the remainder of the number.

The same wrapper function `eif_wrapper` from Exercise 1 will be used for evaluation.

```
In [ ]: ### Define demo inputs
        demo_str_ex3_list = ['404-5551212',
                             '(404-555-1212']
```

The demo included in the solution cell below should display the following output:

```
eif_wrapper('404-5551212', parse_phone2) -> (False, ('404', '555', '1212'))
eif_wrapper('(404-555-1212', parse_phone2) -> (True, None)
```

Recall that `eif_wrapper` returns two outputs: - `bool` indicating whether or not your function raised a `ValueError` - `tuple` (the result of your parsing function) or `None` (when the parsing function raises an error)

```
In [ ]: ### Exercise 3 solution
        def parse_phone2(s):
            ###
            ### YOUR CODE HERE
            pattern = r'''
                \s*
                (?P<first>(\d{3}-?|\(\d{3}\)\s*))
                \s*
                (?P<local3>\d{3})
                -?
                (?P<local4>\d{4})
                \s*
            '''
```

```
    matcher = re.compile(pattern, re.VERBOSE)
    matches = matcher.match(s)
    if not matches:
        raise ValueError
    else:
        first = matches.group("first")
        local3 = matches.group("local3")
        local4 = matches.group("local4")
        # extract the digits from first (removes parentheses or dash)
        areacode = re.search(r'\d{3}', first).group()
        return (areacode, local3, local4)
    ###

### demo function call
for demo_str_ex3 in demo_str_ex3_list:
    print(f"eif_wrapper('{demo_str_ex3}', parse_phone2) -> {eif_wrapper(demo_str_ex3, 
```

The cell below will test your solution for Exercise 3. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]: ### test_cell_ex3
        from tester_fw.testers import Tester

        conf = {
            'case_file':'tc_3',
            'func': lambda s: eif_wrapper(s, parse_phone2), # replace this with the function d
            'inputs':{ # input config dict. keys are parameter names
                's':{
                    'dtype':'str', # data type of param.
                    'check_modified':False,
                }
            },
            'outputs':{
                'output_0':{
                    'index':0,
                    'dtype':'bool',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                },
                'output_1':{
                    'index':1,
                    'dtype':'NoneType',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                },

            }
        }
        tester = Tester(conf, key=b'F24oYNyh8kq_wkZD_OoOZCPHLcoO-xNXHOYNiPnQmmY=', path='resou
        for _ in range(70):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
            except:
                (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
                raise

        print('Passed! Please submit.')
```

Fin! This cell marks the end of Part 0. Don't forget to save, restart and rerun all cells, and submit it. When you are done, proceed to Parts 1 and 2.