This programming assignment is designed to show how we can get Python to check function annotations whenever annotated functions are called. For each of Python's built-in data types, we will develop an interpretation (and write code to check that interpretation) of how the data type specifies type information that is checkable when a function is called.

For example, we specified the main **dict** for the NDFA by the notation

```
  {str : {str : {str}}}
```

Recall the outer-**dict** associates a **str state** key with a value that is an inner-**dict**; and the inner-**dict** associates a **str transition** with a value that is a set of **str states**.

Note that this is an actual data structure in Python, where **str** is a reference to the string class object. It is an outer-**dict** whose single key is **str** and whose associated value is an inner-**dict** whose single key is a **str** and whose value is a **set** that contains just element, **str**.

We will write the **Check_Annotation** class and use it as a decorator for functions whose annotations we want to check each time the function is called. Internally it will overload the **__call__** method to call the function only after checking its annotation by using mutual recursion (not direct recursion), in a natural way, to process the nesting of data types inside data types illustrated in the notation above. We will write code that ensures that this checking works for the standard classes defined in Python. The code will also know how to process a special annotation-checking protocol (via the **__check_annotation__** method) that we can implement in any new classes that we write, so that that class can become part of the annotation language (I have done this for two classes: **Check_All_OK** and **Check_Any_OK**).

I suggest that you look at the code in the modules that appear in the project folder that you will download. Then (see the detailed instructions in this document) we can add/test/debug capabilities for each of the built-in data types we can use in the annotation language, iteratively enhancing our code until we can use all the built-in data types in annotations. Once we do this for **list** (about a dozen lines; the biggest of my checking functions), all the others are variants and therefore much easier to write (but still with some interesting details).

Download the program4 project folder and use it to create an Eclipse project. We can test our class (put it in the **checkannotation.py** module, which already includes some useful code) at the end of the class itself, or in a special driver that is included, which uses a batch file to test progressively more and more complex data types in the annotation language.

Problem Summary:

Write a class named **Check_Annotation** that decorates a function, such that when the decorated function is called, the decorator class check's the decorated function's annotation, using the annotation language described below in detail. We can use this decorator by writing either

```
  def f(params-annotation) -> result-annotation:
      ...
  f = Check_Annotation(f)
```

or

```
  @Check_Annotation
  def f(params-annotation) -> result-annotation:
      ...
```

which is a special Python syntactic form that expands to the former assignment. Thus, when the decorated **f** is called, Python calls **Check_Annotations.__call__** in the decorator, which first checks the annotations (and raises an exception if it is not met) and second computes/returns the decorated function **f**: the original one written.

This class defines four major attributes:

- the **checking_on** instance name to turn off/on annotation checking in all decorated functions; it starts on.
- the **__init__** method to remember the function being decorated and initialize a per-function name that helps controls annotation checking; it also starts on: for a function call to check its annotation, both **checking_on** and its per-function name must be **True**: if either is **False** the annotation is not checked.
- the **__call__** method that intercepts each call to the decorated function and decides whether to check the annotation, and if so implements annotation checking, both for parameters and returned results, if they are specified; if annotation checking succeeds, this method computes/returns the result of calling the decorated function.
- the **check** method (specified in more detail below) that does the annotation checking: it either succeeds silently or raises an **AssertionError** exception with useful information specified in the details below. Note that the unconditional assertion,

  ```
  assert False, message
  ```

  is a simple way to raise **AssertionError** with a message. I wrote many nested helper functions in **check**, one for each data type annotation that can be checked: e.g., **check_dict**.

Details

Let's explore the meaning of the **__call__** and **check** methods in more detail.

I. The **__call__** method intercepts calls to the decorated function; it specifies ***args** and ****kargs** to handle all calls, regardless of their parameter structure. My method was about 40 lines (but about 17 lines were comments/blank, and 7 comprise the **param_arg_binding** local function supplied in the download; this function computes an **ordereddict** of the parameter names (each associated to its argument) in the order that the parameters are defined in the function. The **__call__** method

   - determines whether to check the annotations (see above); if not just call the decorated function and return its result.
   - determines the parameters of the function and the matching arguments they are bound to. The **param_arg_bindings** function (written locally in this method) returns an ordered dictionary of parameter/value bindings; ordered means that when iterated, keys always appear in the same order: the order the parameters appear in in the function's definition. It uses the various attributes in the **inspect** module to do the job. You might be interested in reading the documentation for the **inspect** module: it is quite interesting and many of its (powerful) features are new to Python. **It would be an excellent idea to print this data structure to see what information it accumulates for various annotated function that you test in your script).**
   - determines the annotations of the parameters by using the **__annotations__** attribute of any function object. This name is bound to a dictionary containing as keys every annotated parameter name; the associated value for each parameter name is its annotation. If we defined the function **f** using **def f(x:int,y,z:int)->str:...** its **__annotations__** dictionary is

     ```
     {'x': <class 'int'>, 'z': <class 'int'>, 'return':
     <class 'str'>>}
     ```

     Notice that parameter **y** has no annotation so it does not appear as a key in this dictionary, and the key **return** is associated with the annotation for the returned value (after the **->**).

   - If any checked annotations (parameters or returned result) raise the **AssertionError** handle it by printing the relevant source lines for the function (see the **getsourcelines** function in the **inspect** module's documentation) and reraise the exception, skipping the rest of the code in this method.
     - Checks every parameter that has an annotation
     - Call the decorated function to compute its returned result (and save it).
     - If **'return'** is in the dictionary of annotions: (a) add the result as the value associated with the key **_return** in the dictionary

of parameter and argument bindings; (b) check the annotation for **return**

    - Return the result.

II. The **check** method has the following header

```
def check(self,param,annot,value,check_history=''):
```

where

- **self** is an instance of the **Check_Annotation** class
- **param** is a string that specifies the name of the parameter being checked (or **'_return'** for checking the returned value)
- **annot** is a data structure that specifies the annotation
- **value** is the value of **param** that the annotation should be checked against (to ensure it is legal)
- **check_history** is a string that embodies the history of checking the annotation for the parameter to here (it is extended by concatenation in each recursive call to provide context for any annotation violations to be checked later); it is printed after the details of any annotation violation, to suppy context for the failure.

Each call to **check** decodes the **annot** to check, and checks it against the **value**: **check**'s body is one big **if/elif/.../else** determining which local function to call to check the specific annotation (and letting that local function do the real work). Most annotations are checked by calling a function defined locally in **check** that can use the parameters of **check** freely, because these functions are defined in **check**'s local scope (in fact these local functions are often parameterless: many get all the information they need from check's parameters). The more complicated local functions also call **check**; so **check** calls a local function which can call **check**: this is indirect recursion. My method was about 100 lines: about 13 lines were comments/blank, and 60 more appeared in 5 locally declared functions - including one to solve the extra credit (**str**) part of this assignment- so I had about a dozen lines per local function.

The annotation checking language comprises the following components (for Python's built-in types). I **strongly** suggest writing/testing each component before moving on to the next: all are similar and understanding/testing/debugging **list** (the first really interesting one) will supply tremendous insight for writing all. **Write the required exception messages exactly to match the ones shown.**

- **annot** is **None**: do nothing (succeed silently). note that **def f(x):** has no annotation to check for its parameter **x**, but **def f(x:None):** has an annotation to check for **x**, but it never fails. **None** has more interesting uses inside more complicated data types, illustrated below (see the last example for **list**).

- **annot** is any **type** (e.g., **type(annot) is type)**: fail if **value** is not an instance of the specified type, with an exception messages matching the following examples. The **isinstance** function (covered in the inheritance lectures) generalizes checking the type of an object. Instead of writing **type(x) is someclass** we write **isinstance(x,someclass)**: it checks whether **x**'s object is constructed from **someclass** or any base class of **someclass**, which is the correct test to perform here.

  For **def f(x:int):...** called as **f('abc')** or **f(x='abc')** the exception message would be:

  ```
  AssertionError: 'x' failed annotation check(wrong type):
  value = 'abc'
    was type str ...should be type int
  ```

  For **def f(x:list):...** called as **f({1,2})** the exception message would be:

  ```
  AssertionError: 'x' failed annotation check(wrong type):
  value = {1, 2}
    was type set ...should be type list
  ```

  All exception messages described in the sections below follow this same general format, although the more complicated ones supply extra context via the **check_history** parameter.

- **annot** is a **list** (not the **list** class object, but an instance of **list**: a real list of one or more values; see the examples below) where each element in **list** is an annotation. Fail if
  1. **value** is not a list
  2. **annot** has just one element-annotation, and any of the elements in the **value** list fails the element-annotation check
  3. **annot** has more than one element-annotation, and
     a. the **annot** and **value** lists have a different number of elements, or
     b. any element in the **value** list fails its corresponding element-annotation check

  Here are some examples of failures:

  4. For **def f(x:[int]):...** called as **f({1,2})** the exception message would be:

  5.
     ```
     AssertionError: 'x' failed annotation check(wrong
     type): value = {1, 2}
       was type set ...should be type list
     ```

6. For **def f(x:[int]):...** called as **f([1,'a'])** the exception message would be:

```
7. AssertionError: 'x' failed annotation check(wrong
   type): value = 'a'
8.   was type str ...should be type int
   list[1] check: <class 'int'>
```

Note that when each element in the list is tested, it appends the index it is checking and the annotation it is checking to the **check_history** (which prints after the actual annotation that fails: here the line starting **list[1] check: ...**): it means the element at index **0** did not fail this annotation but the element at index **1** did.

9. For **def f(x:[int,str]):...** called as **f([1])** the exception message would be:

```
10.     AssertionError: 'x' failed annotation
   check(wrong number of elements): value = [1]
     annotation had 2 elements[<class 'int'>, <class
   'str'>]
```

11. For **def f(x:[int,str]):...** called as **f([1,2])** the exception message would be:

```
12.     AssertionError: 'x' failed annotation
   check(wrong type): value = 2
13.       was type int ...should be type str
   list[1] check: <class 'str'>
```

Note that the annotation **def f(x:list):...** and the annotation **def f(x:[None]):...** have the same meaning (but the former is faster to check): the first checks only that **x** is an instance of **list**; the second checks that **x** is an instance of **list** and then checks each of its values agains the annotation **None**, which according to that rule's annotation does not checking and never fails -so really the checks are the same.

Likewise note that for **def f(x:[int,None]):...** called as **f([1,'a'])** no exception is raised, because the annotation for the list element at index **1** is **None**, which according to that rule's annotation does no checking of the list's value at index **1** and never fails.

Note also that for **def f(x:[[str]]):...** called as **f([['a','b'],['c','d']])** no exception is raised, because the annotation says **x** is a list containing lists that contain only strings. The code to check list annotations will indirectly call itself (recursively) in the process of checking this annotation. Think about this now, when there are few data types being processed; it will be natural to perform other recursive annotation checks in the **check** method. In fact, spend a good amount of time simplifying the local function that performs this check, because most of the other annotations listed below look very similar.

Finally, note if we called **f([['a',1],['c','d']])** the exception message would be

```
AssertionError: 'x' failed annotation check(wrong type):
value = 1
  was type int ...should be type str
list[0] check: [<class 'str'>]
list[1] check: <class 'str'>
```

which indicates that the annotation of **list[0]** was being checked when the annotation for **list[1]** was being checked (each of its values should be a **list** of **str**), when Python found a non-string that violated the annotation.

- **annot** is a **tuple** (not the **tuple** class object, but an instance of **tuple**: a real tuple of values), where each element in **annot** is an annotation.

  Structurally, checking tuples is equivalent to checking lists (all 3 rules apply). In fact, I parameterized the local function that I originally wrote for checking lists to work for checking tuples as well). Of course, the error messages should use the word **list** and **tuple** where appropriate. **Caution**: remember for tuples of one value we must write f(x:(int,)):...; notice the comma after **int**.

- **annot** is a **dict** (not the **dict** class object, but an instance of **dict**: a real dictonary; see the examples below), with exactly one key: both the key and its associated value are each an annotation. Note, this annotation should work for subclases of **dict**, e.g., **defaultdict**. Check it not by **type(annot) is dict** but using the **isinstance** function (covered in the inheritance lectures): **isinstance(annot,dict)** Fail if
  0. **value** is not a **dict** or a subclass of **dict**
  1. **annot** has more than one key/value association: this is actually a bad/illegal annotation, not a failed annotation
  2. **annot** has one key/value association, and
     a. any key in the **value** dictionary fails the key-annotation check or
     b. any value in the **value** dictionary fails the value-annotation check

  Here are some examples of failures:

  3. For **def f(x:{str : int}):...** called as **f(['a',0])** the exception message would be:

4. `AssertionError: 'x' failed annotation check(wrong type): value = ['a', 0]`
```
  was type list ...should be type dict
```

5. For **def f(x:{str : int, int : int}):...** called as **f({'a':0})** the exception message would be:
6. `AssertionError: 'x' annotation inconsistency: dict should have 1 item but had 2`
```
  annotation = {<class 'str'>: <class 'int'>,
<class 'int'>: <class 'int'>}
```

7. For **def f(x:{str : int}):...** called as **f({1:0})** the exception message would be:
8. `AssertionError: 'x' failed annotation check(wrong type): value = 1`
9. `  was type int ...should be type str`
   `dict key check: <class 'str'>`

10. For **def f(x:{str : int}):...** called as **f({'a':'b'})** the exception message would be:
11. `    AssertionError: 'x' failed annotation check(wrong type): value = 'b'`
12. `      was type str ...should be type int`
    `dict value check: <class 'int'>`

Of course, if a dictionary had many keys, it would check the required annotations for each of its keys and their associated values.

- **annot** is a **set** (not the **set** class object, but an instance of **set**: a real set of values; see the examples below) where its has exactly one value that is an annotation. Fail if
  0. **value** is not a **set**
  1. **annot** has more than one value: this is actually a bad/illegal annotation, not a failed annotation
  2. **annot** has one value, and any value in the **value** set fails the value-annotation check

Here are some examples of failures:

3. For **def f(x:{str}):...** called as **f(['a','b'])** the exception message would be:
4. `AssertionError: 'x' failed annotation check(wrong type): value = ['a', 'b']`
```
  was type list ...should be type set
```

5. For **def f(x:{str,int}):...** called as **f({'a',1})** the exception message would be:
6. `AssertionError: 'x' annotation inconsistency: set should have 1 value but had 2`
```
  annotation = {<class 'str'>, <class 'int'>}
```

7. For **def f(x:{str}):...** called as **f({'a',1})** the exception message would be:
8. **`AssertionError: 'x' failed annotation check(wrong type): value = 1`**
9. **`  was type int ...should be type str`**
   `set value check: <class 'str'>`

- **annot** is a **frozenset** (not the **frozenset** class object, but an instance of **frozenset**: a real frozenset of values) where its one value is an annotation.

  Structurally, checking frozensets are equivalent to checking sets (all 3 rules apply). In fact, I parameterized the local function that I originally wrote for checking sets to work for checking frozensets as well, similarly to the general function I wrote for checking lists/tuple. Of course, the error messages should use the word **set** and **frozenset** where appropriate.

- **annot** is a **lambda** (or any function object) that is a predicate with one parameter and returning a value that can be interpreted as a **bool**. Fail if
  0. **annot** has zero/more than one parameters: this is actually a bad/illegal annotation, not a failed annotation
  1. Calling the lambda/function on **value** returns **False**
  2. Calling the lambda/function on **value** raises an exception

  Note that we can recognize a function/lambda object by calling the **inspect** module's **isfunction** predicate; we can determine the number of parameters in a function/lambda object by accessing its **__code__.co_varnames** attribute. You might be interested in reading the documentation for the **inspect** module: it is quite interesting and many of its (powerful) features are new to Python.

  Here are some examples of failures: in the first two, the argument fails the **lambda** directly; in the others the argument is a list on which the **lambda** is checked for every value and fails for one.

3. For **def f(x:lambda x,y : x>0):...** called as **f(1)** the exception message would be:
4. **`AssertionError: 'x' annotation inconsistency: predicate should have 1 parameter but had 2`**
   `  predicate = <function <lambda> at 0x02BDDC90>`

5. For **def f(x:lambda x : x>0):...** called as **f(0)** the exception message would be:
6. **`AssertionError: 'x' failed annotation check: value = 0`**

```
  predicate = <function <lambda> at 0x02BDDC90>
```

7. For **def f(x:[lambda x : x>0]):...** called as **f([1,0])** the exception message would be:

```
8. AssertionError: 'x' failed annotation check: value
   = 0
9.   predicate = <function <lambda> at 0x02BDDC90>
   list[1] check: <function <lambda> at 0x02BDDC90>
```

   Note that in this example we are checking the **lambda** annotation for every value in a **list**, just as the annotation **[int]** would check that every value in a **list** was an instance of the **int** class.

10. For **def f(x:[lambda x : x>0]):...** called as **f([1,'a'])** the exception message would be:

```
11.     AssertionError: 'x' annotation
   predicate(<function <lambda> at 0x0022C540>)
   raised exception
12.       exception = TypeError: '>' not supported
   between instances of 'str' and 'int'
   list[1] check: <function <lambda> at 0x0022C540>
```

   Note that for **def f(x:[lambda x : isinstance(x,int) and x>0]):...** called as **f([1,'a'])** the exception message would be the more reasonable:

```
AssertionError: 'x' failed annotation check: value
= 'a'
  predicate = <function <lambda> at 0x02BDDC90>
list[1] check: <function <lambda> at 0x02BDDC90>
```

- **annot** is not any of the above (or **str**, specified in the extra credit part below if you implemented it). Assume it is an object constructed from a class that supports annotation checking, by that class defining the the **__check_annotation__** method. Fail if
  0. There is no **__check_annotation__** method in the class: e.g., calling the **__check_annotation__** method raises the **AttributeError** exception (the object was not constructed from a class that supports the annotation checking protocol): this is actually a bad/illegal annotation, not a failed annotation
  1. calling its **__check_annotation__** method fails
  2. calling its **__check_annotation__** method raises any other exception

  Note that I have written the **Check_All_OK** and **Check_Any_OK** classes that support the annotation checking protocol; check them out.

Here are some examples of failures. The first assumes the **Bag** class does not support the annotation checking protocol; the second assumes it does; the third assumes it supports the protocol but raises some other exception (not **AssertionError**).

3. For **def f(x:Bag([str])):...** called as **f(Bag('a'))** the exception message would be:

   ```
   AssertionError: 'x' annotation undecipherable:
   Bag(<class 'str'>[1])
   ```

4. For **def f(x:Bag([str])):...** called as **f(Bag(['a',1]))** the exception message would be:

```
5. AssertionError: 'x' failed annotation check(wrong
   type): value = 1
6.   was type int ...should be type str
   Bag value check: <class 'str'>
```

7. For **def f(x:Bag([lambda x : x > 0])):...** called as **f(Bag(['a',1]))** the exception message would be:

```
8. AssertionError: 'x' annotation predicate(<function
   <lambda> at 0x006482B8>) raised exception
9.   exception = TypeError: '>' not supported between
   instances of 'str' and 'int'
   Bag value check: <function <lambda> at 0x006482B8>
```

   The **checkannotation.py** module defines the **Check_All_OK** and **Check_Any_OK** classes, which implement the check annotation protocol. Note that with the **Check_Any_OK** class, we can specify that every value in a list must contain a string or integer. So for **def f(x:[Check_Any_OK(str,int)]):...** called as **f(['a',1])** there is no exception raised. Likewise with the **Check_All_OK** class, we can specify that every value in a list must be an integer and must be bigger than 0. So for **def f(x:[Check_All_OK(int,lambda x : x > 0)]:...** called as **f([1,2])** there is no exception raised.

Extra credit: Implement the following annotations as well.

- **annot** is a **str** object, which when **eval**uated using a dictionary in which all the parameters are defined (and the returned result is the value of the key **'_return'**) returns a value that can be interpreted as a **bool**. This specifiction is similar to lambdas/functions, but more general, because the expressions can name multiple names, not just the parameter. Fail if
  0. Evaluating the string returns **False**

1. Evaluating the string raises an exception

Here are some examples of failures.

2. For **def f(x,y:'y>x'):...** called as **f(0,0)** the exception message would be:
3.
```
AssertionError: 'y' failed annotation check(str
predicate: 'y>x')
  args for evaluation: x->0, y->0
```

   Notice that with this form of annotation, we can check properties that depend on values of multiple parameters (not just type information). The values of all the parameters are included in the error message. Likewise we can check properties that depend on the returned values. For **def f(x,y)->'_return < x or _return < y': return x + y** called as **f(3, 5)** the exception message would be:

```
AssertionError: 'return' failed annotation
check(str predicate: '_return < x or _return < y')
  args for evaluation: x->3, y->5, _return->8
```

   Notice the value of **_return** is listed with all the parameter values. Of course, such strings are easier to read than what Python prints for lambdas/functions.

4. For **def f(x:'x>0'):...** called as **f('a')** the exception message would be:
5.
```
AssertionError: 'x' annotation check(str
predicate: 'x>0') raised exception
  exception = TypeError: '>' not supported between
instances of 'str' and 'int'
```

.

A Largish Example: Full Output

When I put the following code in the script (before the driver) in the **checkannotation.py** module).

```
@Check_Annotation
def f(x:[[int]]): pass

f([[1,2],[3,4],[5,'a']])
```

the result printed was the following , although I edited out some of the code that Python displays from my program: lines that start with ...

```
--------------------------------------------------------------
------------------
    @Check_Annotation
    def f(x:[[int]]): pass
--------------------------------------------------------------
------------------
Traceback (most recent call last):
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 209, in <module>
    f([[1,2],[3,4],[5,'a']])
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 183, in __call__
    ...my call to self.check
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 138, in check
    ...my call to check a list
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 70, in check_sequence
    ...my call to check a value in the list
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 138, in check
    ...my call to check a list
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 70, in check_sequence
    ...my call to check a value in the list
  File
"C:\Users\USER\workspace\program4\checkannotationsolution.py",
line 137, in check
    ...my call to check a type (which failed the assertion
causing the following exception)
AssertionError: 'x' failed annotation check(wrong type): value
= 'a'
  was type str ...should be type int
list[2] check: [<class 'int'>]
list[1] check: <class 'int'>
```

Feel free to put the small tests shown in this document (or in the **bsc.txt** file) in the same position (before the driver) to test the annotations as you write them.

Testing

The sections above present various tests for elements of the annotation language: they are easy to specify because the parameter annotations involve only the header: the body can be **pass**; when checking return annotations, we can put one return statement in the body of the code, to return a value that does/doesn't satisfy the annotation.

I provided an **if __name__ == '__main__':** section in the **checkannotation.py** module. Again, it is easy to test a simple function there by annotating it and then calling it (described in the previous section).

I provided code to call **driver.driver()** which can be used to run individual and **batch_self_check**, using the file **bsc.txt**.

Here is an example of running individual tests. After importing and abbreviating the name of the **Check_Annotation** class it defines a simply annotated function, decorates it, and then calls the function with good and bad arguments (which in the latter case rasise an exception because of an annotation failure).

```
Command[!]: from checkannotation import Check_Annotation as ca
Command[from checkannotation import Check_Annotation as ca]:
def f(x:int): pass
Command[def f(x:int): pass]: f = ca(f)
Command[f = ca(f)]: f(1)
Command[f(1)]: f('a')
Traceback (most recent call last):
  File "C:\Users\USER\workspace\courselib\driver.py", line
225, in driver
    exec(old,local,globl)
  File "<string>", line 1, in <module>
  File "C:\Users\USER\workspace\program4\checkannotation.py",
line 183, in __call__
    self.check(p,annot[p],self._args[p])
  File "C:\Users\USER\workspace\program4\checkannotation.py",
line 137, in check
    '\n  was type '+type_as_str(value)+' ...should be type
'+type_as_str(annot)+'\n'+check_history
AssertionError: 'x' failed annotation check(wrong type): value
= 'a'
  was type str ...should be type int
Command[f('a')]:
```

When runing **batch_self_check**, you might want to start by removing all but the earliest test (or comment them out with #) as you start testing your code).

IMPORTANT for running batch-self-check: To use the **batch_self_check** you must remove the part of your **__call__** method that prints out the source lines when an assertion exception is raised: otherwise Python will raise a strange exception (**OSError**), which disrupts **batch_self_check**. Comment out the following lines so that your code looks like

```
except AssertionError:
```

```
#     print(80*'-')
#     for l in inspect.getsourcelines(self._f)[0]: # ignore
starting line #
#     print(l.rstrip())
#     print(80*'-')
    raise
```

IMPORTANT: Comment-out these lines in the code you submit.