2.1 Assertion

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0-discount))
    assert 0 <= price <= product['price']
    return price

shoes = {'name': 'fancy shoes', 'price': 15_001}
price = apply_discount(shoes, 0.25)
# price = apply_discount(shoes, 2.0)
print(price)
```

- run python -O test.py OR python -OO test.py. It will disable the assertion statement
- **DO NOT** use asserts for data validation'''

Example code

```
def delete_product(store,prod_id, user):
    assert user.is_admin(), 'Must be admin'
    assert store.has_product(prod_id), 'Unknown product'
    store.get_product(prod_id).delete
```

- The reason is if the debug flag is turned off, it will cause unpleasant effects.

What we should do

```
def delete_product(store,prod_id, user):
    if not user.is_admin():
        raise AuthError('Must be an admin to delete')
    if store.has_product(prod_id):
        raise ValueError('Unknown product id')
    store.get_product(prod_id).delete
```

The correct way

```
counter = 1
assert counter == 10, 'It should fail'
```

- For some reason, if you pass in a tuple, it will always be true. Why? This is because non-empty tuples always return true in Python'''

The wrong way

```
assert (counter == 10, 'It should fail')
```

2.2 Comma placement

```
names = ['Alice', 'Bob', 'Dilbert']
```

Problem

Whenever you make a change to this list of names, it iwll be hard to tell what was modified by looking at a Git diff.

How to fix it

```
names = [
    'Alice',
    'Bob',
    'Dilbert',
]
```

- Notice the comma after Dilbert, this will make changes in a Git Diff very apparent

Common mistake

```
names = [
    'Alice',
    'Bob',
    'Dilbert' # <- Missing Comma!
    'Jane'
]
```

- if you inspect the element, it will look like this

```
['Alice', 'bob', 'DilbertJane']
```

- This is because of something called String Literal concatenation
    - **String literal concatenation** is a double edge sword, it can make our life convenient and confusing at the same time
    - helpful example:

```
str = ('Hello! This is a line'
'Another line'
'Another line to complete the sentence.')
```

2.3 Context Manager

Focus on the 'with' keyword

```
with open('hello.txt', 'w') as f:
    f.write('hello world!')
```

Internally, the above example translate into something like this

```
f = open('hello.txt', 'w')
try:
    f.write('hello world')
finally:
    f.close()
```

Without the try catch block it will waste resource when we can't write to the file

Another example using the with keyword dealing with concurrent threading class

```
import threading
some_lock = threading.Lock()
some_lock.acquire()
# harmful
try:
    #do something
    print('bruh')
finally:
    some_lock.release()

# better
with some_lock:
    # do something
    print('bruh')
```

Supporting with in your own objects

In order to do that, you'll need to add `__enter__` and `__exit__` method

```
class ManagedFile:
    def __init__(self, name):
        self.name = name
    def __enter__(self):
        self.file = open(self.name, 'w')
        return self.file
    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

how to use it with 'with'

```
with ManagedFile('hello.txt', 'w') as f:
    f.write('hello world!')
```

Or your function can support 'with' using the contextlib library

```
from contextlib import contextmanager

@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

with managed_file('hello.txt') as f:
    f.write('hello world')

# Example 2
with Indenter() as indent:
    indent.print('hi')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')

class Indenter():
    def __init__(self) -> None:
        self.level = 0

    def __enter__(self):
        self.level += 1
        return self
```

```
    def __exit__(self):
        self.level -= 1

    def print(self, text):
        print('    ' * self.level + text)
```

2.4 Underscores , Dunders, etc

```
from my_module import *
'''According to PEP 8, wildcard imports is not recommended'''
# Correct way to import
import my_module
```

single trailing underscore

```
def make_obj(name, class_):
    print('this is correct way so that it avoid conflicts')

def make_obj(name, class):
   print('invalid syntax because class is a keyword')
```

Single preceding underscore may raise conflict with class modules

```
from my_module import *
external_func()
>> it's fine
_internal_func()
>> raises error
```

Double preceding underscore

causes the python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses.

This is called Name mangling