## Part 1. PyCharm features

My program has a couple of modules. One module contains my TripleString class and some other stuff, and another module contains all my test code.

- Project tab. Here we can choose "Project files" vs "Project" view.
- Structure tab. Notice buttons "Show fields", "Show inherited", "Sort alphabetically".
- Option "Show members" in Project tab combines both views.

Show Run tab, Terminal tab (Tools), Python console tab (Build), Version control tab.
(By default, Terminal and Python console windows open to the project's root directory)

Extra tip:
Editor > General > Smart Keys > "Surround selection on typing quote or brace"

## Part 1.a. Module's name

When Python interpreter sees an import statement, it has to read the entire imported module first.
It reads and executes every line of code strictly from top to bottom. But it only has to do it once.

```
print("module name:", __name__)
```

Within each module, the module's name is available as the value of the variable __name__.
This line of code prints different module names depending on which module we run: triplestring or test.

## Part 1.b. Current vs imported module's name

```
if __name__ == '__main__':
```

This statement determines whether the current module is run directly or is it being imported:
- if this module is run directly, the code after this statement will be executed
- if this module is being imported, this code will be skipped

## Part 2. Module level variables

If the module's name is too long, to avoid typing it every time, we can use an alias name.

ms._MIN_LEN and ms._MAX_LEN

When Python interpreter reads a module, it creates a global namespace for that module. Each module has its own global space. When Python interpreter finds an import statement, it imports everything from the imported module's namespace into the current module's namespace.

The only true global variables in Python are module-level variables. They are visible inside every local scope of that module. We can capitalize variable names, however, that doesn't turn them into true, read-only, constants.
Our global "constants" can be easily reassigned. The underscore in the name indicates that our constants/variables are not public, meaning not intended for use outside of the module.

## Part 3. Import statements

import math
from math import *
from math import sqrt, log, sin

The first import form is good for documenting purposes in larger modules. Example: *math.pi*. Fully-qualified attribute names are completely unambitious. Math.sqrt is not going to conflict with your own pi.

The second form imports everything (whether you need it or not). All functions, classes, and variables now can be used without the dot prefix, and this can be very confusing, creates namespace collision.

The last import form is most specific, it imports only what you listed. Imported names can be used without the dot prefix. And notice that private global variables of the imported module are out of the way.

However, nothing in Python makes it possible to enforce data hiding — it is all based upon convention.

## Part 4. Scopes. Call stack

Set breakpoints on t = TripleString.from_list(['one']), the last print statement, and inside the static method _is_valid(). Step through the code and watch the call stack.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope contains the local names
- any number of enclosing scopes
- scope containing the current module's global names
- the outermost scope is the namespace containing built-in names

If you want to see built-in names, try in Python console

```
for i in __builtins__:
    print('\t', i)
```

For a function, the local namespace is created during run time and deleted when the function returns (the frame is added to and popped from the call stack).

Class definition is yet another namespace in the local scope.
When a class definition is entered, a new namespace is created, and used as the local scope — all assignments to local variables go into this new namespace. When a class definition is left, a class object is created.

Assignments to names always go into the innermost scope, unless global keyword is used.

## Part 5. Class level variables

t1.LUCKY_NUM = 666

creates a new local variable that shadows the global variable.

t1.__dict__   prints out t1's namespace.

This doesn't only have to do with class variables, creation of new object variable is always possible.

## Part 6. Class methods, Dunder methods, Instance methods

Class methods are often used for alternative constructors.

Dunder methods overload methods (operators) in object.
Run with and without __repr__ methods.
Run with and without __eq__ method.

Classes have data attributes and function attributes.

Instance methods are just function attributes of a class, they expect an object instance passed in as the first argument.
To put it another way: attributes of a class that are function objects define corresponding methods of their instances.
That's why the alternative syntax works.

## Part 7. Property object

If our class has self.string1 together with a self.set_string1() methods, then we have created two ways to change the value of string1.

The Zen of Python
>>> import this
"There should be one -- and preferably only one -- obvious way to do it."

If you need improved data encapsulation, use @property (property decorator).

decorators are also called "wrappers".
@decorator syntax is just syntactic sugar for the special wrapper function property().
This function returns a property object, which has its own methods: getter, setter, deleter.

Properties may be writable or read-only.

Name mangling:
Any identifier of the form **__name** is textually replaced with **_classname__name**.
Name mangling is helpful for creating "private" names.

## Part 8 Class attributes and Object attributes

Both classes and objects have data attributes and function attributes.

| **Class attributes** | **Object attributes** |
|---|---|
| **Constructor** | |
| __init__ <**function** TripleString.__init__> | __init__ = <**bound method** TripleString.__init__ of TripleString ("a", "b", "c")> |
| **Instance method** | |
| abbreviate <**function** TripleString.abr> | abbreviate = <**bound method** TripleString.abbreviate of TripleString ("a", "b", "c")> |
| **Class method** | |
| from_list <**bound method** TripleString.from_list of <class ''triplestring .TripleString'>> | from_list <**bound method** TripleString.from_list of <class 'triplestring.TripleString'>> |
| **Static method** | |
| _is_valid <**function** TripleString._is_valid > | _is_valid <**function** TripleString._is_valid > |
| Special methods - override methods in object | |
| __repr__ <**function** TripleString.__repr__ > | __repr__ = <**bound method** TripleString.__repr__ of TripleString ("a", "b", "c")> |
| __eq__ <**function** TripleString.__eq__ > | __eq__ = <**bound method** TripleString.__eq__ of TripleString ("a", "b", "c")> |
| | __str__ = <**bound method** TripleString.__str__ of TripleString ("a", "b", "c")> |
| **Class variables** | |
| _count 0 | _count 1 |
| LUCKY_NUM 3 | LUCKY_NUM 3 |
| **Properties** | |
| string1 <**property object** ><br>string2 <**property object** ><br>string3 <**property object** > | _TripleString__s1 a<br>_TripleString__s2 b<br>_TripleString__s3 c |
| **Instance variables** | |
| - | id 1<br>string1 a<br>string2 b<br>string3 c |