Import, Docstrings, Special Methods

Modules should have short, all-lowercase names.

syntax

```
import <module>
```

to import a specific element

```
from <module import <element>
```

for all the elements(**Wildcard import**)

```
from <module import *
```

to import as a different name

```
import <module> as <name>
```

imports should be grouped in the following order

- standard library imports
- standard third party imports
- local application specific imports

Types of Imports

There are two types of imports:

- **Absolute Imports**
  - They contain the full path to your script (python file), starting from the root folder.
  - The folders are separated with a period.
  - It can be as long as needed.
  - Advantage: They are the preferred type of import statement because they clearly show the path to the module.
  - Disadvantage: They can be quite long if the project has many subfolders.

Example:

```
from <package>.<module> import <module>
import <package>.<module>.<function>
```

Note: <package>, <module>, <function> are placeholders for the real names. You need to replace them in the project.

- **Relative Imports**
  - The path specifies where the modules are located **relative** to the current code file.
  - The syntax uses **leading dots**: one dot indicates the directory of the current script. Two dots indicate the parent directory. Three dots indicate the grandfather directory, and so on, one directory per dot.
  - For relative imports, the dots can go up to the directory that contains the script that is run.

Example:

```
import other
from .<module> import <element>
from ..<subpackage>.<module> import <element>
```

According to the Python Documentation (PEP 328):

"Guido has Pronounced that relative imports will use leading dots. A single leading dot indicates a relative import, starting with the current package. Two or more leading dots give a relative import to the parent(s) of the current package, one level per dot after the first". - source.

⚠️ Important

According to the Python Documentation:

Relative imports must always use from <> import; **import <> is always absolute**. Of course, absolute imports can use from <> import by omitting the leading dots. - source.
To learn more about absolute and relative import, please refer to the PEP 328:

- PEP 328 -- Imports: Multi-Line and Absolute/Relative

DOCSTRINGS

It refers to documentation strings.They are

- String literals
- first statement in a class,method , module
- can be one line or multiple lines
- to expain purpose,logic and data types
- Specific and relvant information
- docstrings can also be easily converted to documentation and are linked to the elements using the __doc__ attribute

Docstrings can be read in the help(<elem>) in the interactive shell but comments can be only be read from the source code

One line docstrings are used mostly for methods or functions. **For one-line docstrings, the closing quotes should be in the same line as the opening quotes.**
ex:

```
def add(a,b):
    """add two integers and return the resulting integer"""
    return a + b
```

multiline doctrings are used for classes,method and modules
for a method:

- arguments
- optional arguments
- return value

- return value
- side effects (ex:mutation)
- Exceptions raised
- Restrictions on when it can be called

for classes

- purpose
- list public methods
- list public instance variables
- effects of inheritnece
- __init__() ducmented seperately
- indivdual methods should have their own docstrings

example

```
class Triangle:
    # Body

    def find_area(self, base, height):
        """Return the area of a triangle.
        Find the area of a triangle using the base
        and the height provided. These values must be
        positive or zero.
        Args:
            base: A positive integer that represents the length
                of the base of the triangle. This value can be zero.
            height: A positive integer that represents the length
                of the height of the triangle. This value can be zero.
        Returns:
            A float that represents the area of the triangle.
        Raises:
            ValueError: the base or the height or both are not valid.
        """
        if base < 0 or height < 0:
            raise ValueError("The base and height must be either positive or zero")

        return (base * height)/2
```

Document classes

- while working with properites we should only document the getters as only that will be displayed in the documentation
- The class constructors should be documented in the docstring for its __init__() method

```
class TissueSample:
    """Class that represents a tissue sample.
    Attributes:
        patient (Patient): the patient associated with the tissue sample.
        organ (str): the organ associated with tissue sample.
        code (str): a unique alphanumeric sequence that identifies the tissue sample.
        diagnosis (str): the diagnosis associated with this tissue sample. By default,
            it is initialized to None when the instance is created.
    Methods:
        show_data(): displays the four main properties of the tissue sample
            in a human-readable format
```

```
        in a human readable format.
    """

    def __init__(self, patient, organ, code):
        """Initalize the values of the instance attributes of an instance of TissueSample.
        Args:
            patient (Patient): the patient associated with the tissue sample.
            organ (str): the organ associated with tissue sample.
            code (str): a unique alphanumeric sequence that identifies the tissue sample.
        """
        self.patient = patient
        self.organ = organ
        self._code = code
        self._diagnosis = None
    def show_data(self):
        """Display tissue sample data in a human-readable format."""
        print(f"==== Tissue Sample (code #{self.code})====")
        print(f"Patient:", self.patient.name)
        print(f"Organ:", self.organ)
        if self.diagnosis:
            print(f"Diagnosis:", self.diagnosis)
    # Read-only property (only a getter)
    @property
    def code(self):
        """Code of the tissue sample."""
        return self._code
    @property
    def diagnosis(self):
        """Diagnosis associated with the tissue sample."""
        return self._diagnosis
    @diagnosis.setter
    def diagnosis(self, diagnosis):
        self._diagnosis = diagnosis
```

Quick Intro: Exceptions

Exceptions will be mentioned in this section because you should document them in your docstrings.

Please refer to this article if you would like to read a quick introduction to the concept of exceptions:

- **Errors and Exceptions - Python Documentation**

Basically:

- Exceptions are errors detected during the execution of a program.
- They can be handled appropriately to avoid an unexpected crash of the program.

Example: Documenting Functions

This is a Python function without any documentation. Please take a moment and try to determine what the function does, the expected data type of the arguments, and the data type and format of the return value.

```
def frequency_dict(data):
    if not data:
        raise ValueError("The list cannot be empty")

    freq = {}

    for elem in data:
        if elem not in freq:
            freq[elem] = 1
        else:
            freq[elem] += 1
    return freq
```

That took a few seconds and it wasn't so intuitive, right? 😠

Now, this is the same function with its corresponding docstring that documents the purpose of the function, the arguments, the return value, and the exceptions.

Notice how documentation makes it much easier for other developers to understand the logic behind a function (in this case), its arguments, and return value. 👍

```
def frequency_dict(data):
    """Return a dictionary with the number of occurrences of each value in the list.
    Create a dictionary that maps each element of the list
    to the number of times it occurs in the list.
    Args:
        data: A list of values of an immutable data type.
            These values can be integers, booleans, tuples, or strings.
    Return:
        A dictionary mapping each value with its frequency.
        For example, this function call:
        a = [1, 6, 2, 6, 2]
        returns this dictionary:
        {1: 1, 6: 2, 2: 2}
    Raises:
        ValueError: if the list is empty.
    """
    if not data:
        raise ValueError("The list cannot be empty")

    freq = {}

    for elem in data:
        if elem not in freq:
            freq[elem] = 1
        else:
            freq[elem] += 1
    return freq
```

There are three main styles used to write docstrings:

- Sphinx Style

- Google Style
- Numpy Style

__doc__ attribute

helps to retrun the doc string

```
def add(a,b):
    """add two integers and return the resulting integer"""
    return a + b
#using docstring
#help(add) gives info
print(add.__doc__)
```

OUTPUT

```
add two integers and return the resulting integer
```

Special Methods

Some example are:

- __add__()
- __str__()
- __repr__()
- __len__()

They have double leading and trailing underscores.It acts as a type of method overriding
Special Methods are also called:

- Magic Methods
- Dunder Methods

💡 **Tips:** The term "dunder" refers to the "double underscore" that is characteristic of their syntax. It was created to provided a more concise way to refer to the methods without saying "underscore underscore init underscore underscore", we can say "dunder init".

example:

```
print((5).__add__(6))
```

OUTPUT

```
11
```

__str__ is used when we do print(obj) and return a customised string

example:

```
class Point:

    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

p1 = Point(5,4)
print(p1)
```

OUTPUT

```
<__main__.Point object at 0x7f3ca4a771d0>
```

without the __str__

with the __str__

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"
```

OUTPUT

```
Cordinates : 5 , 4
```

__str__() vs. __repr__()

These special methods might look similar at first when you read the documentation because they **both return a string** that describes the object, but they are very different. Let's see why:

- **__str__()**
  - Used to provide an **informal** representation of the object meant for the final **users**.
  - Favores readability over details or precision.
  - Called by the str() , format(), and print() built-in functions.

According to the Python documentation:

"This method differs from object.__repr__() in that there is **no expectation that __str__() return a valid Python expression**: a more convenient or concise representation can be used."
Source: __str__() - Python Documentation

- **__repr__()**
  - Used to provide a **formal** representation of the object meant for **developers**.
  - They are used for debugging.
  - Called by the repr() built-in function.

According to the Python Documentation:

If at all possible, this **should look like a valid Python expression** that could be used to recreate an object

with the same value (given an appropriate environment). If this is not possible, a string of the form <...some useful description...> should be returned.
Source: __repr__() - Python Documentation.

💡 **Note:** The default implementation of __str__() calls __repr__()

🔺 Example
Here we have an example of a class that implements these two special methods:

```
class Shirt:
    def __init__(self, color, size, brand):
        self.color = color
        self.size = size
        self.brand = brand
    def __str__(self):
        return f"Color: {self.color}; Size: {self.size}; Brand: {self.brand}"
    def __repr__(self):
        return f'Shirt("{self.color}", "{self.size}", "{self.brand}")'
shirt = Shirt("Blue", "XL", "Retro Land")
print("__str__() -", str(shirt))
print("\n__repr__() -", repr(shirt))
# The object can be recreated if needed
new_shirt = eval(repr(shirt))
print("\n(New object)", new_shirt)
```

The output is:

```
__str__() - Color: Blue; Size: XL; Brand: Retro Land
__repr__() - Shirt("Blue", "XL", "Retro Land")
(New object) Color: Blue; Size: XL; Brand: Retro Land
```

💡 **Note:** Notice how the __str__() representation is more "informal" and how the __repr__() representation is much more formal (it is a copy of the Python expression used to recreate the object). Using the eval() function, we can use the string returned by repr() to recreate the object.

This code uses f-strings, a feature supported by Python 3.6 and the latest versions.

__len__()
It is indirectly called when we use len(obj).**IT must return a integer**

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"

    def __len__(self):
        """Distance from the origin"""
        return round((self.x**2 + self.y**2)**(1/2))

p1 = Point(5,4)
print(len(p1))
```

OUTPUT

```
6
```

__getitem__

- takes one parameters
- helps to return the item that we are trying access

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cordinates = {"x": self.x, "y": self.y}

    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"



    def __len__(self):
        """Distance from the origin"""
        return round((self.x**2 + self.y**2)**(1/2))

    def __getitem__(self, cord):
        return self.cordinates[cord]

p1 = Point(5,4)
print(p1["x"])
```

OUTPUT

```
5
```

__call__

Python has a set of built-in methods and __call__ is one of them. The __call__ method enables Python programmers to write classes where the instances behave like functions and can be called like a function. When the instance is called as a function; if this method is defined, x(arg1, arg2, ...) is a shorthand for x.__call__(arg1, arg2, ...).

```
object() is shorthand for object.__call__()
```

Example 1:

```
class Example:
    def __init__(self):
        print("Instance Created")

    # Defining __call__ method
    def __call__(self):
        print("Instance is called via special method")

# Instance created
e = Example()

# __call__ method will be called
e()
```

Output :

```
Instance Created
Instance is called via special method
```

The forward method in pytorch acts the same way

__add__

It is used as a default of the add operator

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cordinates = {"x": self.x, "y": self.y}

    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"

    def __len__(self):
        """Distance from the origin"""
        return round((self.x**2 + self.y**2)**(1/2))

    def __getitem__(self, cord):
        return self.cordinates[cord]

    def __add__(self, other):
        x = self.x + other.x
        y = self.y  + other.y
        return f"Point: ({x},{y}) "

p1 = Point(5,4)
p2 = Point(6,3)
p3 = p1 + p2
print(p3)
```

OUTPUT

```
Cordinates : 11 , 7
```

__bool__

It always returns True unless defined

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cordinates = {"x": self.x, "y": self.y}

    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"

    def __len__(self):
        """Distance from the origin"""
        return round((self.x**2 + self.y**2)**(1/2))

    def __getitem__(self, cord):
        return self.cordinates[cord]

    def __add__(self, other):
        x = self.x + other.x
        y = self.y  + other.y
        return Point(x,y)

    def __bool__(self):
        return self.x > self.y

p1 = Point(2,4)
p2 = Point(6,3)

print(bool(p2))
print(bool(p1))
```

OUTPUT

```
True
False
```

Relevant Connection: __bool__() and __len__()

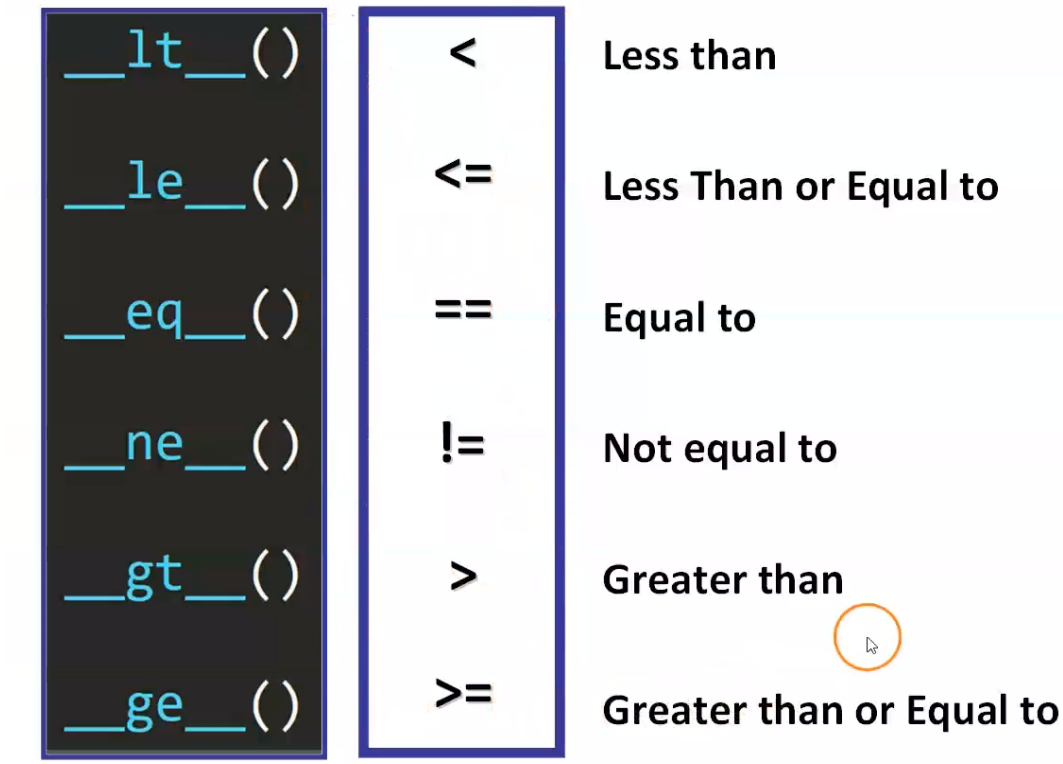
💡 **According to the Python Documentation:**

"When __bool__() is not defined, __len__() is called, if it is defined, and the object is considered true if its result is nonzero."
and...

"If a class defines neither __len__() nor __bool__(), all its instances are considered true."
Source: __bool__() - Python Documentation

source: __bool__() - Python Documentation

Rich Comparison Methods



example:

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cordinates = {"x": self.x, "y": self.y}

    def __str__(self):
        return f"Cordinates : {self.x} , {self.y}"

    def __len__(self):
        """Distance from the origin"""
        return round((self.x**2 + self.y**2)**(1/2))

    def __getitem__(self, cord):
        return self.cordinates[cord]

    def __add__(self, other):
        x = self.x + other.x
        y = self.y  + other.y
        return Point(x,y)

    def __bool__(self):
        return self.x > self.y
```

```
    def __lt__(self, other):
        return self.x < other.x

    def __le__(self, other):
        return self.x <= other.x

    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)

    def __ne__(self, other):
        return (self.x != other.x) or (self.y != other.y)

    def __gt__(self, other):
        return self.x > other.x

    def __ge__(self, other):
        return self.x >= other.x

p1 = Point(2,4)
```