

Coding Guidelines For I2B2

Limited use of globals:

Global variables can be altered by any part of the code, making it difficult to remember or reason about every possible use.

A global variable can have no access control. It can not be limited to some parts of the program.

Using global variables causes very tight coupling of code.

Standard headers for different modules:

For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:

Name of the module

Date of module creation

Author of the module

Modification history

Synopsis of the module about what the module does

Different functions supported in the module along with their input output parameters

Global variables accessed or modified by the module

```
#
# Copyright (c) 2020-2021 Massachusetts General Hospital. All rights reserved.
# This program and the accompanying materials are made available under the terms
# of the Mozilla Public License v. 2.0 ( http://mozilla.org/MPL/2.0/) and under
# the terms of the Healthcare Disclaimer.
#
"""
:mod:`addProject` -- Add i2b2 project
=====

.. module:: addProject
   :platform: Linux/Windows
   :synopsis: module contains methods for creating project & user in i2b2

.. moduleauthor:: Shreyash Joshi <shreyashj10@gmail.com>

"""
```

Naming conventions for local variables, global variables, constants and functions:

Some of the naming conventions are given below:

Meaningful and understandable variables name helps anyone to understand the reason of using it.

Local variables should be named using camel case lettering starting with small letter (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData). Constant names should be formed using capital letters only (e.g. CONSDATA).

It is better to avoid the use of digits in variable names.

The names of the function should be written in camel case starting with small letters.

The name of the function must describe the reason of using the function clearly and briefly.

EXAMPLE:

```
# class name follows camelcase convention
class StudentDetails:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    # Method name, variable names in lowercase joined with an underscore
    def grade(self, marks_obtained):
        # constants in capital
        GRACE = 2
        marks_obtained = GRACE + marks_obtained
        if marks_obtained > 90:
            self.student_grade = 'A'
        elif marks_obtained > 70:
            student_grade = 'B'
        else:
            student_grade = 'C'
```

Imports, Blank Lines, and the Indentations:

The import should be in a particular sequence. At first, the standard libraries, then the third party, and at the last, the local libraries should be imported. If you only need a single function/class from the

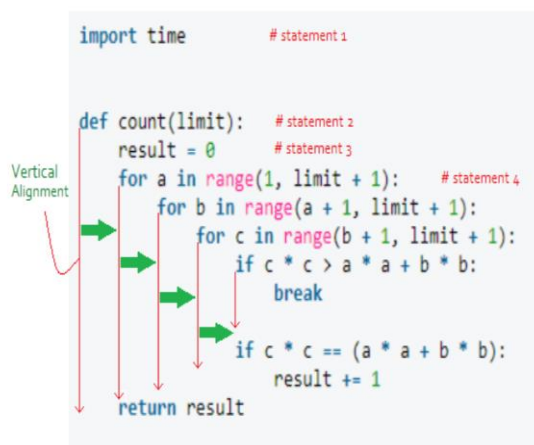
import, do an absolute import. It makes your code much cleaner, accurate, and easy to identify. Don't forget to add a space between different types of imports.

There should be two blank lines surrounding classes and top-level functions. The methods inside of the class should be surrounded by a single blank line only. The preferred method of indentation is spaces, the 4 spaces indentation is accepted and accurate, but still, most people prefer tab indentation. Please keep in mind not to mix both spaces and tabs for indentation.

Proper Indentation should be there at the beginning and at the end of each block in the program.

All braces should start from a new line and the code following the end of braces also start from a new line.

EXAMPLE:



```
import time # statement 1

def count(limit): # statement 2
    result = 0 # statement 3
    for a in range(1, limit + 1): # statement 4
        for b in range(a + 1, limit + 1):
            for c in range(b + 1, limit + 1):
                if c * c > a * a + b * b:
                    break
            if c * c == (a * a + b * b):
                result += 1
    return result
```

Indentation

```
# Don't forget to add a space between different group of imports

# first of all, the standard library imports
import standard_library_import_a
import standard_library_import_b

# then, the third party imports
import third_party_import_a
import third_party_import_b
import third_party_import_c

# at the last, local library import
from local_library import local_a, local_b
from local_library_two import local_c
```

Sequence of imports

Code should be well documented:

The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.

Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.

EXAMPLE:

```
# documenting a function
def get_grades(marks):
    """
    Summary: getting grades from marks
    Description: This function takes marks as an argument and returns grades
    params:
    marks(int): marks obtained
    :
    grade(string): grade achieved
    """
    if marks > 90:
        grade = 'A'
    elif marks > 70:
        grade = 'B'
    else:
        grade = 'C'
```

Length of functions should not be very large:

Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks. Methods should do only one logical thing.

The given example only finds whether the number is prime or not, it has only one functionality, so for better understanding the function should not be lengthy, should only have one functionality.

```
def is_prime(n):
    if n in [2, 3]:
        return True
    if n % 2 == 0:
        return False
    r = 3
    while r * r <= n:
        if n % r == 0:
            return False
        r += 2
    return True
print(is_prime(78), is_prime(79))
```

Use DRY (Don't Repeat Yourself):

Always use the DRY principle to reuse the code. The best way to do it is to use functions and classes. The common functions can be put into a separate utils.py file and can be used several times instead of creating similar functions again and again.

EXAMPLE:

```

text1 = 1
text2 = 2
text3 = 3
text4 = 4
text5 = 5

print(f"Number: {text1}")
print(f"Number: {text2}")
print(f"Number: {text3}")
print(f"Number: {text4}")
print(f"Number: {text5}")

# DRY

texts = [1, 2, 3, 4, 5]

for i in range(len(texts)):
    print(f"Number: {texts[i]}")

```

Defensive coding

- Program defensively, i.e., assume that errors are going to arise, and write code to detect them when they do.
- Put assertions in programs to check their state as they run, and to help readers understand how those programs are supposed to work.

Python

```

numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for num in numbers:
    assert num > 0.0, 'Data should only contain positive values'
    total += num
print('total is:', total)

```

Error

```

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-19-33d87ea29ae4> in <module>()
      2 total = 0.0
      3 for num in numbers:
----> 4     assert num > 0.0, 'Data should only contain positive values'
      5     total += num
      6 print('total is:', total)

AssertionError: Data should only contain positive values

```

- A [precondition](#) is something that must be true at the start of a function in order for it to work correctly. Use preconditions to check that the inputs to a function are safe to use.
- Use postconditions to check that the output from a function is safe to use.
- Write tests before writing code in order to help determine exactly what that code is supposed to do.

Use the 'with' statement while opening a file, the 'with' statement closes the file even if there is an exception raised

```
import csv

# opening a file, with statement is used
with open('filename.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for line in csv_reader:
        print(line)
```

Try not to use GOTO statement:

GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

"GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.

For HTML

Always declare the document type as the first line in your document.

```
<!DOCTYPE html>
```

Close All HTML Elements

```
<section>
  <p>This is a paragraph.</p>
  <p>This is a paragraph.</p>
</section>
```

Always Quote Attribute Values

```
<table class="striped">
```

For CSS

Formatting

All CSS documents must use **two spaces** for indentation and files should have no trailing whitespace.

Other formatting rules:

- Use soft-tabs with a two space indent.(**Soft tabs are just spaces**. Soft tabs are usually either 2 or 4 spaces)
- Use double quotes.
- Put spaces after `:` in property declarations.
- Put spaces before `{` in rule declarations.
- Use hex color codes `#000` unless using `rgba()` .
- Always provide fallback properties for older browsers.
- Use one line per property declaration.
- Always follow a rule with one line of whitespace.
- Always quote `url()` and `@import()` contents.

```
.media {
  overflow: hidden;
  color: #fff;
  background-color: #000; /* Fallback value */
  background-image: linear-gradient(black, grey);
}

.media .img {
  float: left;
  border: 1px solid #ccc;
}

.media .img img {
  display: block;
}

.media .content {
  background: #fff url("../images/media-background.png") no-repeat;
}
```