Coding Style Guidelines

Help Center

In this class, we will expect you to follow our coding style conventions. These conventions will largely be enforced by the use of Pylint. Pylint is a source code bug and quality checker for the Python programming language. The machine-graders for this class will use a customized version of Pylint to check the style of your code. Pylint errors that violate of the style guidelines will result in deductions from the score of your mini-project. An explanation of common Pylint errors can be found here.

Here are some of the style guidelines that we will follow in this class. As you interact with Pylint, you will get more exposure to these guidelines.

Documentation

Documentation strings ("docstrings") are an integral part of the Python language. They need to be in the following places:

- At the top of each file describing the purpose of the module.
- Below each class definition describing the purpose of the class.
- Below each function definition describing the purpose of the function.

Docstrings describe what is being done in a module, class, method, or function, not how it is being done. Except in rare cases where the how is part of the contract (i.e., binary search, so you know it runs in time \log(n)). A docstring for a function should explain the arguments, what the function does, and what the function returns. This sample file demonstrates the use of docstrings. Note that the `__init__` methods of classes generally do not have docstrings because their purpose is obvious: to initialize the object. You may want to have one, though, to describe the arguments.

These docstrings are treated specially in Python, as they allow the system to automatically give you documentation for modules, classes, functions, and methods. At the command prompt, you can type `help(...)`, and it will return the docstring for whatever the argument you passed to help is. (Note that CodeSkulptor does not provide a command prompt, so you cannot use `help` in CodeSkulptor.)

Comments

Comments should describe how a section of code is accomplishing something. You should not comment obvious code. Instead, you should document complex code and/or design decisions. Comments and docstrings are not interchangeable. Comments start with the "#" character. While you will see some Python programmers do this, you should not comment your code by putting a multi-line string in the middle of your program. That is not actually a comment, rather it is just a string in the middle of your program!

A good example:

```
# This is a block comment
# that spans multiple lines
# explaining the following code.
val = some complicated expression
```

A bad example:

```
"""
Somebody told me that a multiline
string is a block comment.

It's not.
"""
val = some complicated expression
```

Note that docstrings are multi-line strings, but they do not violate this convention because docstrings and comments are different and serve different purposes in a program.

Global variables

Global variables should never be used in this class. Avoiding their use is good programming practice in any language. While programmers will sometimes break this rule, you should not break this rule in this class.

There is one exception to this rule: you may have global constants. Because the Python language does not actually support constants, by convention, Python programmers use global variables with names that are in all capital letters for constants. When you see a variable with a name in all capital letters, you should always assume that it is a constant and you should never change it. Again, such global constants are the only global variables that will be allowed in this class.

Indentation

Each indentation level should be indented by 4 spaces. As Python requires indentation to be consistent, it is important not to mix tabs and spaces. You should never use tabs for indentation. Instead, all indentation levels should be 4 spaces. Note that CodeSkulptor automatically converts all tab indents into 4 spaces.

Names

All variable, function, class, and method names must be at least 3 characters. The first character of a name should follow these conventions:

- Variable names should always start with a lower case letter. (Except for variables that represent constants, which should use all upper case letters.)
- Function and method names should always start with a lower case letter.
- Class names should always start with an upper case letter.

Further, we will follow the common Python convention that variable, function, and method names should not have any capital letters in them. You can separate words in a name with an underscore character, as follows: `some_variable_name`. Similarly, class names should not contain underscores, and instead use capitalization to separate words, as follows: `SomeClassName`.

As previously noted, constants should be in all capital letters, such as: `THIS_IS_A_CONSTANT`. Note that this means that your class names must have at least one lower case letter in them, to distinguish them from constants.

By convention in Python, you can "violate" the above rules and start a name with an underscore, `_`, to indicate that the name is *private* and should not be accessed outside of the context in which it is defined. In this case, the rest of the name after the underscore should follow the rules given above. This will arise mainly when you define instance fields in classes, as these style guidelines insist that such fields be private (discussed next).

Class and Instance Fields

Class and instance fields should never be accessed directly from outside the class. You should therefore always start your field names with an underscore. Even if you don't, you still should not access them from outside of the class.

You will often see public fields in Python programs (and in programs of other languages). This is not a good habit to get into. Instead, all fields should always be private. If there is good reason to make the data in the field accessible outside the class, you should create a method to do so. By convention in other languages, these methods are usually named `get_field`, where `field` is the name of the field. You should follow this convention.

Note that this is *not* common in Python. Rather, public fields or properties are used. However, we are trying to teach principles that transcend a particular programming language. All languages support writing so-called "getter" methods, whereas many do not support properties. There is nothing wrong with Python properties, it is just a different syntax for using methods. However, the use of public fields is not good practice in any programming language, whether the language allows it or not.

You should avoid the use of class fields, which are declared in the scope of the class itself and are common to all instances of the class. Instead, you should use instance fields (defined as attributes of `self`). These will be unique to each instance of the class.

Scope

You should not use names that knowingly duplicate other names in an outer scope. This would make the name in the outer scope impossible to access. In particular, you should never use names that are the same as existing Python built-in functions. For example, if you were to name one of your local variables `max` inside of a function, you would then not be able to call `max()` from within that function.

Arguments and local variables

While there is not necessarily a maximum number of arguments a function can take or a maximum number of local variables you can have, too many arguments or variables lead to unnecessarily complex and unreadable programs. Pylint will enforce maximum numbers of arguments, variables, methods, etc. If you run into limits that Pylint complains about, you should restructure your program to break it into smaller pieces. This will result in more readable and maintainable code.

Further, you should not have function arguments or local variables declared that are never used, except in rare circumstances. Sometimes, you do need to have a variable that you never use. A common case is in a loop that just needs to execute a certain number of times:

```
for num in range(42):
    # do something 42 times
```

In this case, you should name the variable with the `dummy_` prefix. This indicates clearly to you, others, and Pylint that the variable is intentionally unused.

```
for dummy_num in range(42):
    # do something 42 times
```

Created Wed 16 Apr 2014 3:22 PM PDT

Last Modified Wed 4 Feb 2015 8:24 AM PST