DUY TÂN UNIVERSITY
International School

CAPSTONE PROJECT 2

CMU-SE-451 / CMU-IS-451 / CMU-CS-451

CODE STANDARD

Version 2.0

Date: 1 - Mar - 2021

EXPERT-DRIVEN SMART DASHBOARD APPLICATION

Submitted by

Vo Van Hoa
Pham Van Tin
Ky Huu Dong
Tran Thi Thanh Kieu

Approved by

Capstone Project 2 - Mentor:

Name	Signature	Date
Binh, Thanh Nguyen		26 - May - 2020

PROJECT INFORMATION			
Project Acronym	EDSDA		
Project Title	Expert-Driven Smart Dashboard Application		
Project Web URL	https://sda-research.ml/		
Start Date	01 - Mar - 2021		
End Date:	02 - Jun - 2021		
Lead Institution	International School, Duy Tan University		
Project Mentor	Ph.D Binh, Thanh Nguyen		
Scrum Master	Hoa, Vo	hoavo.dng@gmail.com	0935.193.182
Team Members	Tin, Pham Van	tinphamvan123@gmail.com	0932.535.175
	Dong, Ky Huu	kyhuudong@gmail.com	0898.246.980
	Kieu, Tran Thi Thanh	thanhkieutran391@gmail.com	0358.583.251

DOCUMENT INFORMATION			
Document Title	Code Standard		
Author(s)	Team C2SE.06		
Role	[EDSDA] Code Standard v2.2		
Date	01 - Mar - 2021	Filename	[EDSDA] 010 Code Standard
URL	https://github.com/sdateamdtu2020/SDA-v2.0		
Access	Project and CMU Program		

REVISION HISTORY

Version	Person(s)	Date	Description	Approval
Draft	Hoa, Vo	01 - Mar - 2021	Initiate document	x
2.0	All members	26 - May - 2020	Finish content of document	x

Table of Contents

Types

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.
  - `string`
  - `number`
  - `boolean`
  - `None`

```
foo = 1
bar = foo

bar = 9

print(foo, bar) # => 1, 9
```

- 1.2 **Complex**: When you access a complex type you work on a reference to its value.
  - `dict`
  - `list`
  - `function`

```
foo = [1, 2]
bar = foo

bar[0] = 9

print(foo[0], bar[0]) # => 9, 9
```

⬆ back to top

References

- 2.1 Use `CONST` for all of your references; avoid using `var`. Python does not have `constant` type, so you need to observe the convention using UPPERCASE for constants and never modify them.

> Why? This ensures that you can't reassign your references, which can lead to bugs and difficult to comprehend code.

```
# bad
foo = 1
bar = 2

# good
FOO = 1
BAR = 2
```

⬆ back to top

Dictionaries

- 3.1 Use the literal syntax for dictionary creation.

```

```

```
# bad
item = dict()

# good
item = {}
```

- 3.2 Use computed key names when creating dictionaries with dynamic key names.

  > Why? They allow you to define all the key of a dictionary in one place.

```
def get_key(k):
    return f'a key named {k}'

# bad
obj = {
    'id': 5,
    'name': 'San Francisco',
}
obj[get_key('enabled')] = True

# good
obj = {
    'id': 5,
    'name': 'San Francisco',
    get_key('enabled'): True,
}
```

- 3.3 Prefer the dictionary spread operator over `copy()` to shallow-copy and extend dictionaries.

```
# bad
original = {'a': 1, 'b': 2}
clone = original.copy()
clone.update({'c': 3})

# good
original = {'a': 1, 'b': 2}
clone = {**original, 'c': 3}

# good
original = {'a': 1, 'b': 2}
original_2 = {'c': 3, 'd': 4}
long_clone = {**original, **original_2, 'e': 5}
```

- 3.4 Use line breaks after open and before close dictionary braces only if a dictionary has multiple lines.

```
# bad - single item will not exceed one line
single_map = {
    'a': 1,
}

# bad - single line
item_map = {
    'a': 1, 'b': 2, 'c': 3,
}

# good
single_map = {'a': 1}

item_map = {
    'a': 1,
    'b': 2,
    'c': 3,
}
```

- 3.5 Use `dict.get(key)` to get properties.

> Why? Getting via `dict[key]` will break on missing key, and requires bloated code to guard against.

```
item_map = {
    'a': 1,
    'b': 2,
}

# bad - throws error
item_map['c']

# bad - bloated code
try:
    item_map['c']
except KeyError:
    item_map['c'] = 3
    return item_map['c']

# good
item_map.get('c')

# good
item_map['c'] = item_map.get('c') or 3
```

⬆ back to top

Lists

- 4.1 Use the literal syntax for list creation.

```
# bad
items = list()

# good
items = []
```

- 4.2 Use list spreads * to copy and extend lists.

```
# bad
items = ['a', 'b']
clone = items.copy() + ['c']

# good
items = ['a', 'b']
clone = [*items, 'c']

# good
items = ['a', 'b']
items_2 = ['c', 'd']
clone = [*items, *items2, 'e']
```

- 4.3 Use line breaks after open and before close list brackets only if a list has multiple lines.

```
# bad - single line
items = [
    [0, 1], [2, 3], [4, 5],
]

# bad - no line break after bracket
dict_list = [{
    'id': 1
}, {
    'id': 2
}]

number_list = [
    1, 2,
]

# good
items = [[0, 1], [2, 3], [4, 5]]

dict_list = [
    {'id': 1},
    {'id': 2},
]
```

```
number_list = [
    1,
    2,
]
```

⬆ back to top

Destructuring

- 5.1 Use list destructuring.

```
items = [1, 2, 3, 4, 5]

# bad
first = items[0]
second = items[1]

# good
first, second, *tail = items
first, second, *rest, last = items
```

⬆ back to top

Strings

- 6.1 Use single quotes `''` for strings.

> Why? Less escaping for double quote `""`, less bloat, and makes code more searchable.

```
# bad
name = "Capt. Janeway"

# bad
json_string = "{\"a\": 1}"

# bad - f string should contain interpolation or newlines
name = f'Capt. Janeway'

# good
name = 'Capt. Janeway'

# good
json_string = '{"a": 1}'
```

- 6.2 Strings that cause the line to go over 79 characters should not be written across multiple lines using string concatenation.

> Why? Broken strings are painful to work with and make code less searchable.

```
# bad
error_msg = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do
\
with this, you would get nowhere \
fast.'

# bad
error_msg = 'This is a super long error that was thrown because ' + \
    'of Batman. When you stop to think about how Batman had anything
to do ' + \
    'with this, you would get nowhere fast.'

# good
error_msg = 'This is a super long error that was thrown because of
Batman. When you stop to think about how Batman had anything to do
with this, you would get nowhere fast.'
```

- 6.3 When programmatically building up strings, use template strings instead of concatenation.

> Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.

```
# bad
def say_hi(name):
    return 'How are you, ' + name + '?'

# bad
def say_hi(name):
    return ''.join(['How are you, ', name, '?'])

# bad
def say_hi(name):
    return f'How are you, { name }?'

# good
def say_hi(name):
    return f'How are you, {name}?'
```

> For python < 3.6, convert f-string to template string by `format()`:

```
a = 1
b = 2
c = 3

# python >=3.6
```

```
f'a: {a} b: {b} c: {c}'

# python <3.6
'a: {a} b: {b} c: {c}'.format(a=a, b=b, c=c)

param = {'a': a, 'b': b, 'c': c}
'a: {a} b: {b} c: {c}'.format(**param)

'a: {} b: {} c: {}'.format(a, b, c)
```

- 6.4 Never use `eval()` on a string, it opens too many vulnerabilities.
- 6.5 Do not unnecessarily escape characters in strings.

> Why? Backslashes harm readability, thus they should only be present when necessary.

```
# bad
foo = '\'this\' \i\s \"quoted\"'

# good
foo = '\'this\' is "quoted"'
foo = f'my name is "{name}"'
```

⬆ back to top

Functions

- 7.1 Use default parameter syntax rather than mutating function arguments.

```
# really bad
def do_something(opt):
    # No! We shouldn't mutate function arguments.
    # Double bad: if opt is falsy it'll be set to an object which may
    # be what you want but it can introduce subtle bugs.
    opt = opt or 'foo'
    # ...

# still bad
def do_something(opt):
    if (opt is None):
        opt = 'foo'
    # ...

# good
def do_something(opt='foo'):
    # ...
```

- 7.2 Do not use complex data type as default parameter.

> Why? Variable to a complex type is a reference, and so the single instance will be modified.

```
# bad
def init_list(value, new_list=[]):
    new_list.append(value)
    return new_list

init_list(1)
# => [1]
init_list(2)
# => [1, 2], instead of the new init [2]

# good
def init_list(value, new_list=None):
    if new_list is None:
        new_list = []
    new_list.append(value)
    return new_list

init_list(1)
# => [1]
init_list(2)
# => [2]
```

- 7.3 No spacing in a function signature.

> Why? Consistency is good, and eases code search.

```
# bad
def foo(a): print(a)
def bar (b): print(b)

# good
def foo(a): print(a)
def bar(b): print(b)
```

- 7.4 Never reassign parameters.

> Why? Reassigning parameters can lead to unexpected behavior.

```
# bad
def fn_1(a):
    a = 1
    # ...

def fn_2(a):
    if (!a): a = 1
    # ...
```

```
# good
def fn_3(a):
    b = a or 1
    # ...

def fn_4(a=1):
    # ...
```

- 7.5 Functions with multiline signatures, or invocations, should be indented just like every other multiline list in this guide: with each item on a line by itself, with a trailing comma on the last item.

```
# bad
def some_fn(foo,
            bar,
            baz):
    # ...

# good
def some_fn(
    foo,
    bar,
    baz,
):
    # ...

# bad
some_fn(foo,
  bar,
  baz)

# good
some_fn(
  foo,
  bar,
  baz,
)
```

- 7.6 Call function with parameters by specifying their names.

> Why? Clarity of parameters and future-proofing. When updating source code function parameters, it can be done reliably with minimal propagation.

```
def move(x, y, roll=False):
    # ...

# bad - unclear what the params mean
move(1, 0, True)
```

```
# good
move(x=1, y=0, roll=True)

# later when updating method, no need to propagate function calls
since they will auto-assume z=0 reliably
def move(x, y, z=0, roll=False):
    # ...
```

- 7.7 Break down code logic into digestible chunks, and refactor a lot.

> Why? Other programmers and your future self will thank you for writing understandable code.

```
# bad - short but extremely confusing
def long_logic():
    return [a for a in small_list for small_list in large_matrix if
len(small_list) else ['replacement'] if len(a) > 2]

# good - longer but very clear
def long_logic():
    result = []
    for small_list in large_matrix:
        if len(small_list) > 0:
            used_list = small_list
        else:
            used_list = ['replacement']
        for a in used_list:
            if len(a) > 2:
                result.append(a)
    return result
```

⬆ back to top

Classes & Constructors

- 8.1 Avoid duplicate class members.

> Why? Duplicate class member declarations will silently prefer the last one - having duplicates is almost certainly a bug.

```
# bad
class Foo():
    def bar(): return 1
    def bar(): return 2

# good
class Foo():
    def bar(): return 1
```

⬆ back to top

Modules

- 9.1 Do not use wildcard imports.

> Why? To prevent namespace pollution and conflicts, and to know which modules your variables or functions come from.

```
# bad
from common.util import *

# good
from common import util
```

- 9.2 Do not import unused modules.

> Why? Performance, reliability, containment. If a module breaks, your code that should be isolated from the module will break too. This causes more errors and makes it harder to debug.

- 9.3 Only import from a path in one place.

> Why? Having multiple lines that import from the same path can make code harder to maintain.

```
# bad
from foo import bar
# ... some other imports
from foo import baz, qux

# good
from foo import bar, baz, qux

# good
from foo import (
    bar,
    baz,
    qux,
)
```

- 9.4 Put all `import`s above non-import statements.

> Why? Since `import`s are hoisted, keeping them all at the top prevents surprising behavior.

```
# bad
import a_module
from b_module import foo
foo.init()
```

```
from c_module import bar

# good
import a_module
from b_module import foo
from c_module import bar

foo.init();
```

- 9.5 Sort the `import`s by `import` then `from`, and sort alphabetically.

  > Why? `import` are often more generic that `from`; sort to ease manual inspection and for maintainability.

```
# bad
from a_module import foo
import e_module
import b_module
from c_module import c_fn, b_fn

# good
import b_module
import e_module
from a_module import foo
from c_module import b_fn, c_fn
```

- 9.6 Multiline imports should be indented just like multiline list and dictionary literals.

  > Why? The parentheses follow the same indentation rules as every other bracket or brace block in the style guide, as do the trailing commas.

```
# bad
from a_module import long_name_a, long_name_b, long_name_c,
long_name_d

# good
from a_module import (
  long_name_a,
  long_name_b,
  long_name_c,
  long_name_d,
)
```

⬆ back to top

Iterators and Generators

(Pending)

⬆ back to top

Variables

- 11.1 Use UPPERCASE to declare constants, and observe the convention - do not modify them in the program. Python has no `constant` type, so it must be observed manually.

```
# bad
a_constant = 1
os.environ['py_env'] = 'development'

# good
A_CONSTANT = 1
os.environ['PY_ENV'] = 'development'
```

- 11.2 Declare one constant per line.

> Why? For clarity, and it's easier to add/remove declarations this way, and with minimal git-diffs. You can also step through each declaration with the debugger, instead of jumping through all of them at once.

```
# bad
FOO, BAR, BAZ = 1, 2, 3

# good
FOO = 1
BAR = 2
BAZ = 3
```

- 11.3 Group all your `CONST`s and then group all your `var`s.

> Why? For clarity and ease of reference. This is also helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

```
# bad
FOO = 1
counter = 0
BAR = 2
length = counter

# good
FOO = 1
BAR = 2

counter = 0
length = counter
```

- 11.4 Assign variables with the minimally sufficient scope at where you need them, but place them in a reasonable place.

> Why? Prevent variable scope-leak and conflicts

```
# bad - leak to sibling
counter = 0 # mean to count group_a only
for list_a in group_a:
    counter += len(list_a)

for list_b in group_b:
    counter += len(list_b)

# bad - leak into smaller scope
counter = 0 # mean to count within groups
for group in super_group:
    counter += len(group)
    for list in group:
        counter += len(list)

# good
counter_a = 0
for list_a in group_a:
    counter_a += len(list_a)

counter_b = 0
for list_b in group_b:
    counter_b += len(list_b)

# good
group_counter = 0 # mean to count within groups
for group in super_group:
    group_counter += len(group)
    list_counter = 0 # mean to count within lists
    for list in group:
        list_counter += len(list)
```

- 11.5 Prepend underscore _ when naming variables that are unused. Also a part of PEP8.

> Why? To be aware of data usage and side effects.

```
# bad
first, unused, last = [1, 2, 3]

# bad - finder is not used though expected to be
for finder, replacer in some_map.items():
    do_something_without_key(replacer)

# bad - lose track of what the first key is
```

```
for _, replacer in some_map.items():
    do_something_without_key(replacer)

# good
first, _unused, last = [1, 2, 3]

# good - we know what the variable is, and it is unused
for _finder, replacer in some_map.items():
    do_something_without_key(replacer)
```

⬆ back to top

Comparison Operators & Equality

- 12.1 Use concise boolean conditionals, refactor long compound statements.

  > Why? Long boolean statements are hard to read and understand.

```
# bad
if (can_move_x() and can_move_y() or is_light() or is_dry() or
has_high_drag()):
    execute_operation_tumbleweed()
else:
    execute_operation_cactus()

# good
can_move = can_move_x() and can_move_y()
movable = is_light() or is_dry() or has_high_drag()
if (can_move or movable):
    execute_operation_tumbleweed()
else:
    execute_operation_cactus()
```

- 12.2 Be direct with booleans, avoid unnecessary negations.

  > Why? Negations are harder to understand and longer to write.

```
# bad
if not a_is_legal():
    do_b()
else:
    do_a()

# good
if a_is_legal():
    do_a()
else:
    do_b()
```

- 12.3 Use shortcuts for booleans, but explicit comparisons for strings and numbers.

```
# bad
if is_valid == true:
    # ...

# good
if is_valid:
    # ...

# bad
if name:
    # ...

# good
if name != '':
    # ...

# bad
if len(a_list):
    # ...

# good
if len(a_list) > 0:
    # ...
```

- 12.4 Ternaries should not be nested and generally be single line expressions.

```
# bad
foo = 'bar' if maybe_1 > maybe_2 else 'baz' if value_1 > value_2 else
None

# best
maybe_none = 'baz' if value1 > value2 else None
foo = 'bar' if maybe1 > maybe2 else maybe_none
```

- 12.5 Avoid unneeded ternary statements.

```
# bad
foo = a if a else b
bar = True if c else False
baz = False if c else True

# good
foo = a or b
bar = c
baz = not c
```

⬆ back to top

Comments

- 13.1 Use `'''...'''` for multi-line comments.

```
# bad
def make(tag):
    # make() returns a new element
    # based on the passed in tag name
    #
    # @param {string} tag
    # @return {Element} element

    # ...
    return element

# good
def make(tag):
    '''
    make() returns a new element
    based on the passed in tag name

    @param {string} tag
    @return {Element} element
    '''

    # ...
    return element
```

- 13.2 Use # for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment unless it's on the first line of a block.

```
# bad
active = True  # is current tab

# good
# is current tab
active = True

# bad
def get_type():
    print('fetching type...')
    # set the default type to 'no type'
    type = self.type or 'no type'

    return type

# good
def get_type():
```

```
    print('fetching type...')

    # set the default type to 'no type'
    type = self.type or 'no type'

    return type

# also good
def get_type():
    # set the default type to 'no type'
    type = self.type or 'no type'

    return type
```

- 13.3 Start all comments with a space to make it easier to read.

```
# bad
#is current tab
active = True

# good
# is current tab
active = True
```

- 13.4 Prefixing your comments with `TODO` helps yourself and other developers be aware of items to revisit or implement. It keeps the issues visible and easy to find. These are different than regular comments because they are actionable.

```
def complex_calculator():
    compute_basic()

    # TODO figure out improvements to the logic
    return compute_core_logic()
```

⬆ back to top

Whitespace

- 14.1 Use soft tabs (space character) set to 4 spaces as per PEP8.

```
# bad
def foo():
··return bar

# bad
def foo():
·return bar
```

```
# good
def foo():
····return bar
```

- 14.2 Set off operators with spaces.

```
# bad
x=y+5

# good
x = y + 5
```

- 14.3 End files with a single newline character.

```
# bad
import util
# ...
def foo():
    return bar
```

```
# bad
import util
# ...
def foo():
    return bar↵
↵
```

```
# bad
import util
# ...
def foo():
    return bar↵
```

- 14.4 Leave a blank line after blocks and before the next statement.

```
# bad
if foo:
    return bar
return baz

# good
if foo:
```

```
    return bar

return baz

# bad
results = [
    fn_1(),
    fn_2(),
]
return results

# good
results = [
    fn_1(),
    fn_2(),
]

return results
```

- 14.5 Do not pad your blocks with blank lines.

```
# bad
def bar():

    print(foo)

# bad
if (baz):

    print(qux)
else:
    print(foo)

# good
def bar():
    print(foo)

# good
if (baz):
    print(qux)
else:
    print(foo)
```

- 14.6 Do not add spaces inside parentheses.

```
# bad
def bar( foo ):
    return foo
```

```
# good
def bar(foo):
    return foo

# bad
if ( foo and fux ):
    print(foo)

# good
if (foo and fux):
    print(foo)
```

- 14.7 Do not add spaces inside brackets or braces.

```
# bad
foo = [ 1, 2, 3 ]
print(foo[ 0 ])
bar = { 'a': 1 }

# good
foo = [1, 2, 3]
print(foo[0])
bar = {'a': 1}
```

- 14.8 Avoid having lines of code that are longer than 79 characters (including whitespace) as per PEP8. Note: per above, long strings are exempt from this rule, and should not be broken up.

> Why? This ensures readability and maintainability.

```
# bad
foo = nested_object and nested_object.foo and nested_object.foo.bar
and nested_object.foo.bar.baz and nested_object.foo.bar.baz.quux and
nested_object.foo.bar.baz.quux.xyzzy

# bad
http_call({'method': 'POST', 'url': 'https://airbnb.com/', 'data':
{name: 'John', 'age': 20}})

# good
foo = (nested_object and
    nested_object.foo and
    nested_object.foo.bar and
    nested_object.foo.bar.baz and
    nested_object.foo.bar.baz.quux and
    nested_object.foo.bar.baz.quux.xyzzy)

# good
http_call({
    'method': 'POST',
```

```
    'url': 'https://airbnb.com/',
    'data': {name: 'John', 'age': 20},
})
```

⬆ back to top

Commas

- 15.1 Leading commas: **Nope.**

```
# bad
story = [
      once
    , upon
    , a_time
]

# good
story = [
    once,
    upon,
    a_time,
]

# bad
hero = {
    'first_name': 'Ada'
    , 'last_name': 'Lovelace'
    , 'birth_year': 1815
    , 'super_power': 'computers'
}

# good
hero = {
    'first_name': 'Ada',
    'last_name': 'Lovelace',
    'birth_year': 1815,
    'super_power': 'computers',
}
```

- 15.2 Additional trailing comma: **Yup.**

> Why? This leads to cleaner git diffs during code change.

```
# bad - git diff without trailing comma
hero = {
     'first_name': 'Ada',
-    'last_name': 'Lovelace'
+    'last_name': 'Lovelace',
```

```
+    'super_power': 'computers',
}

# good - git diff with trailing comma
hero = {
     'first_name': 'Ada',
     'last_name': 'Lovelace',
+    'super_power': 'computers',
}
```

```
# bad
hero = {
    'first_name': 'Ada',
    'last_name': 'Lovelace'
}

heroes = [
    'Batman',
    'Superman'
]

# good
hero = {
    'first_name': 'Ada',
    'last_name': 'Lovelace',
}

heroes = [
    'Batman',
    'Superman',
]

# bad
def create_hero(
    first_name,
    last_name,
    superpower
):
    # ...

# good
def create_hero(
    first_name,
    last_name,
    superpower,
):
    # ...

# good - note that a comma must not appear after a "spread" element
def create_hero(
    first_name,
```

```
    last_name,
    superpower,
    **kwargs
):
    # ...

# bad
create_hero(
    first_name,
    last_name,
    superpower
)

# good
create_hero(
    first_name,
    last_name,
    superpower,
)

# good - note that a comma must not appear after a "spread" element
create_hero(
    first_name,
    last_name,
    superpower,
    **kwargs
)
```

⬆ back to top

Naming Conventions

- 16.1 Use snake_case when naming variables, functions, and instances. Use it for file names too as they will be used in imports.

```
# bad
import myModule
OBJEcttssssss = {}
thisIsMyObject = {}
def thisIsMyFunction():

# good
import my_module
objects = {}
this_is_my_object = {}
def this_is_my_function():
```

- 16.2 Use PascalCase only when naming classes.

```
# bad
class prioritizedMemoryReplay():
    # ...

memory = prioritizedMemoryReplay()

# good
class PrioritizedMemoryReplay():
    # ...

memory = PrioritizedMemoryReplay()
```

- 16.3 Avoid single letter names. Use descriptive and meaningful names - tell what the function does, or what data type an object is. Use `description_object` instead of `object_description`.

```
# bad
def q():
    # ...

# good
def query():
    # ...

# bad - no convention to know what data type it is
df_raw_data = pd.DataFrame(some_data)
id_map_num = {'a': 1, 'b': 2}

# good - convention to tell data type by the last term
raw_data_df = pd.DataFrame(some_data)
id_num_map = {'a': 1, 'b': 2}

# bad - meaningless names, lost context
LIST_1 = ['Jack', 'Alice', 'Emily']
# ... many lines of code later
for item in LIST_1:
    register_human(item)

# good
NAME_LIST = ['Jack', 'Alice', 'Emily']
# ... many lines of code later
for name in NAME_LIST:
    register_human(name)
```

- 16.4 Avoid using close naming to prevent typo and confusion.

```
# bad
objects = ['rock', 'paper', 'scissors']
for object in objects:
```

```
    register_item(object)
close_box(objects)

# okay
objects = ['rock', 'paper', 'scissors']
for obj in objects:
    register_item(obj)
close_box(objects)

# best
object_list = ['rock', 'paper', 'scissors']
for object in object_list:
    register_item(object)
close_box(object_list)
```

- 16.5 Use singular or base words in naming; avoid using plural and instead append singular with the data type.

> Why? To prevent inconsistencies and second-guesses when using variables. Also, plurals are 1 letter away from a typo, are hard to read, and are ambiguous on the data type.

```
# bad
def moves_object(x, y):
    # ...

# good
def move_object(x, y):
    # ...

# bad - inconsistent naming for same data type and usage
teacher = ['Michael']
students = ['Jack', 'Alice', 'Emily']
books = pd.DataFrame({'title': ['lorem', 'ipsum']})

for t in teacher:
    register_human(t)

for student in students:
    register_human(student)

for book in books:
    register_item(book) # wrong; iterate column name instead of book

# good
teacher_list = ['Michael']
student_list = ['Jack', 'Alice', 'Emily']
book_df = pd.DataFrame({'title': ['lorem', 'ipsum']})

for teacher in teacher_list:
    register_human(teacher)
```

```
for student in student_list:
    register_human(student)

# naming as df suggests it shall be treated as a dataframe
for _idx, book in book_df.iterrow():
    register_item(book)
```

- 16.6 Use singular naming for modules and source files.

```
# bad
from commons import utils

utils.read_string()

# good
from common import util

util.read_string()
```

- 16.7 Use abbreviations if they are clear and make for more readable and writable code.

> Why? Names are for humans, so always make code readable and easy to spell.

```
# bad
flight_prerequisites_checklist = ['landing gear', 'engine', 'flaps']
initialize_flight(flight_prerequisites_checklist)

# good
flight_prereq_checklist = ['landing gear', 'engine', 'flaps']
init_flight(flight_prereq_checklist)
```

- 16.8 Use simple, concise names over long, explicit ones.

> Why? Names are for humans to read, and should make the code clean.

```
# bad - explicit Java-style naming clutters code and harms readability
poscode_to_city_name_map = {11223: 'brooklyn'}
poscode_to_city_name_to_state_name_map =
map_city_to_state(poscode_to_city_name_map)
poscode_to_city_name_to_state_name_to_country_map = {}

# good - understandable and fast to read
poscode_city_map = {11223: 'brooklyn'}
poscode_state_map = map_city_to_state(poscode_city_map)
poscode_country_map = {}
```

⬆ back to top

React/JSX Style Guide

Basic Rules

- Always use JSX syntax.
- Do not use `React.createElement` unless you're initializing the app from a file that is not JSX.

Class vs `React.createClass` vs stateless

- If you have internal state and/or refs, prefer `class extends React.Component` over `React.createClass`. eslint: `react/prefer-es6-class` `react/prefer-stateless-function`

```
// bad
const Listing = React.createClass({
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  },
});

// good
class Listing extends React.Component {
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
}
```

And if you don't have state or refs, prefer normal functions (not arrow functions) over classes:

```
// bad
class Listing extends React.Component {
  render() {
    return <div>{this.props.hello}</div>;
  }
}

// bad (relying on function name inference is discouraged)
const Listing = ({ hello }) => <div>{hello}</div>;

// good
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

Mixins

- Do not use mixins.

> Why? Mixins introduce implicit dependencies, cause name clashes, and cause snowballing complexity. Most use cases for mixins can be accomplished in better ways via components, higher-order components, or utility modules.

Naming

- **Extensions**: Use `.jsx` extension for React components. eslint: `react/jsx-filename-extension`
- **Filename**: Use PascalCase for filenames. E.g., `ReservationCard.jsx`.
- **Reference Naming**: Use PascalCase for React components and camelCase for their instances. eslint: `react/jsx-pascal-case`

```
// bad
import reservationCard from "./ReservationCard";

// good
import ReservationCard from "./ReservationCard";

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- **Component Naming**: Use the filename as the component name. For example, `ReservationCard.jsx` should have a reference name of `ReservationCard`. However, for root components of a directory, use `index.jsx` as the filename and use the directory name as the component name:

```
// bad
import Footer from "./Footer/Footer";

// bad
import Footer from "./Footer/index";

// good
import Footer from "./Footer";
```

- **Higher-order Component Naming**: Use a composite of the higher-order component's name and the passed-in component's name as the `displayName` on the generated component. For example, the higher-order component `withFoo()`, when passed a component `Bar` should produce a component with a `displayName` of `withFoo(Bar)`.

> Why? A component's `displayName` may be used by developer tools or in error messages, and having a value that clearly expresses this relationship helps people understand what is happening.

```
// bad
export default function withFoo(WrappedComponent) {
  return function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}

// good
export default function withFoo(WrappedComponent) {
  function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }

  const wrappedComponentName = WrappedComponent.displayName
    || WrappedComponent.name
    || 'Component';

  WithFoo.displayName = `withFoo(${wrappedComponentName})`;
  return WithFoo;
}
```

- **Props Naming**: Avoid using DOM component prop names for different purposes.

> Why? People expect props like `style` and `className` to mean one specific thing. Varying this API for a subset of your app makes the code less readable and less maintainable, and may cause bugs.

```
// bad
<MyComponent style="fancy" />

// bad
<MyComponent className="fancy" />

// good
<MyComponent variant="fancy" />
```

Declaration

- Do not use `displayName` for naming components. Instead, name the component by reference.

```
// bad
export default React.createClass({
  displayName: 'ReservationCard',
```

```
  // stuff goes here
});

// good
export default class ReservationCard extends React.Component {
}
```

Alignment

- Follow these alignment styles for JSX syntax. eslint: `react/jsx-closing-bracket-location` `react/jsx-closing-tag-location`

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// if props fit in one line then keep it on the same line
<Foo bar="bar" />

// children get indented normally
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>

// bad
{showButton &&
  <Button />
}

// bad
{
  showButton &&
    <Button />
}

// good
{showButton && (
  <Button />
)}

// good
{showButton && <Button />}
```

```
// good
{someReallyLongConditional
  && anotherLongConditional
  && (
    <Foo
      superLongParam="bar"
      anotherSuperLongParam="baz"
    />
  )
}

// good
{someConditional ? (
  <Foo />
) : (
  <Foo
    superLongParam="bar"
    anotherSuperLongParam="baz"
  />
)}
```

Quotes

- Always use double quotes (") for JSX attributes, but single quotes (') for all other JS. eslint: `jsx-quotes`

> Why? Regular HTML attributes also typically use double quotes instead of single, so JSX attributes mirror this convention.

```
// bad
<Foo bar='bar' />

// good
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />
```

Spacing

- Always include a single space in your self-closing tag. eslint: `no-multi-spaces`, `react/jsx-tag-spacing`

```
// bad
<Foo/>

// very bad
<Foo                  />

// bad
<Foo
 />

// good
<Foo />
```

- Do not pad JSX curly braces with spaces. eslint: `react/jsx-curly-spacing`

```
// bad
<Foo bar={ baz } />

// good
<Foo bar={baz} />
```

Props

- Always use camelCase for prop names, or PascalCase if the prop value is a React component.

```
// bad
<Foo
  UserName="hello"
  phone_number={12345678}
/>

// good
<Foo
  userName="hello"
  phoneNumber={12345678}
  Component={SomeComponent}
/>
```

- Omit the value of the prop when it is explicitly `true`. eslint: `react/jsx-boolean-value`

```
// bad
<Foo
  hidden={true}
/>

// good
```

```
<Foo
  hidden
/>

// good
<Foo hidden />
```

- Always include an `alt` prop on `<img>` tags. If the image is presentational, `alt` can be an empty string or the `<img>` must have `role="presentation"`. eslint: `jsx-a11y/alt-text`

```
// bad
<img src="hello.jpg" />

// good
<img src="hello.jpg" alt="Me waving hello" />

// good
<img src="hello.jpg" alt="" />

// good
<img src="hello.jpg" role="presentation" />
```

- Do not use words like "image", "photo", or "picture" in `<img>` `alt` props. eslint: `jsx-a11y/img-redundant-alt`

> Why? Screenreaders already announce `img` elements as images, so there is no need to include this information in the alt text.

```
// bad
<img src="hello.jpg" alt="Picture of me waving hello" />

// good
<img src="hello.jpg" alt="Me waving hello" />
```

- Use only valid, non-abstract ARIA roles. eslint: `jsx-a11y/aria-role`

```
// bad - not an ARIA role
<div role="datepicker" />

// bad - abstract ARIA role
<div role="range" />

// good
<div role="button" />
```

- Do not use `accessKey` on elements. eslint: `jsx-a11y/no-access-key`

> Why? Inconsistencies between keyboard shortcuts and keyboard commands used by people using screenreaders and keyboards complicate accessibility.

```
// bad
<div accessKey="h" />

// good
<div />
```

- Avoid using an array index as `key` prop, prefer a stable ID. eslint: `react/no-array-index-key`

> Why? Not using a stable ID is an anti-pattern because it can negatively impact performance and cause issues with component state.

We don't recommend using indexes for keys if the order of items may change.

```
// bad
{
  todos.map((todo, index) => <Todo {...todo} key={index} />);
}

// good
{
  todos.map((todo) => <Todo {...todo} key={todo.id} />);
}
```

- Always define explicit defaultProps for all non-required props.

> Why? propTypes are a form of documentation, and providing defaultProps means the reader of your code doesn't have to assume as much. In addition, it can mean that your code can omit certain type checks.

```
// bad
function SFC({ foo, bar, children }) {
  return (
    <div>
      {foo}
      {bar}
      {children}
    </div>
  );
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};
```

```
// good
function SFC({ foo, bar, children }) {
  return (
    <div>
      {foo}
      {bar}
      {children}
    </div>
  );
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};
SFC.defaultProps = {
  bar: "",
  children: null,
};
```

- Use spread props sparingly.

  > Why? Otherwise you're more likely to pass unnecessary props down to components. And for React v15.6.1 and older, you could pass invalid HTML attributes to the DOM.

Exceptions:

- HOCs that proxy down props and hoist propTypes

```
function HOC(WrappedComponent) {
  return class Proxy extends React.Component {
    Proxy.propTypes = {
      text: PropTypes.string,
      isLoading: PropTypes.bool
    };

    render() {
      return <WrappedComponent {...this.props} />
    }
  }
}
```

- Spreading objects with known, explicit props. This can be particularly useful when testing React components with Mocha's beforeEach construct.

```
export default function Foo {
  const props = {
    text: '',
    isPublished: false
```

```
  }

  return (<div {...props} />);
}
```

Notes for use: Filter out unnecessary props when possible. Also, use prop-types-exact to help prevent bugs.

```
// bad
render() {
  const { irrelevantProp, ...relevantProps } = this.props;
  return <WrappedComponent {...this.props} />
}

// good
render() {
  const { irrelevantProp, ...relevantProps } = this.props;
  return <WrappedComponent {...relevantProps} />
}
```

Refs

- Always use ref callbacks. eslint: `react/no-string-refs`

```
// bad
<Foo
  ref="myRef"
/>

// good
<Foo
  ref={(ref) => { this.myRef = ref; }}
/>
```

Parentheses

- Wrap JSX tags in parentheses when they span more than one line. eslint: `react/jsx-wrap-multilines`

```
// bad
render() {
  return <MyComponent variant="long body" foo="bar">
           <MyChild />
         </MyComponent>;
}

// good
render() {
```

```
  return (
    <MyComponent variant="long body" foo="bar">
      <MyChild />
    </MyComponent>
  );
}

// good, when single line
render() {
  const body = <div>hello</div>;
  return <MyComponent>{body}</MyComponent>;
}
```

Tags

- Always self-close tags that have no children. eslint: `react/self-closing-comp`

```
// bad
<Foo variant="stuff"></Foo>

// good
<Foo variant="stuff" />
```

- If your component has multiline properties, close its tag on a new line. eslint: `react/jsx-closing-bracket-location`

```
// bad
<Foo
  bar="bar"
  baz="baz" />

// good
<Foo
  bar="bar"
  baz="baz"
/>
```

Methods

- Use arrow functions to close over local variables. It is handy when you need to pass additional data to an event handler. Although, make sure they do not massively hurt performance, in particular when passed to custom components that might be PureComponents, because they will trigger a possibly needless rerender every time.

```
function ItemList(props) {
  return (
```

```
    <ul>
      {props.items.map((item, index) => (
        <Item
          key={item.key}
          onClick={(event) => {
            doSomethingWith(event, item.name, index);
          }}
        />
      ))}
    </ul>
  );
}
```

- Bind event handlers for the render method in the constructor. eslint: `react/jsx-no-bind`

> Why? A bind call in the render path creates a brand new function on every single render. Do not use arrow functions in class fields, because it makes them challenging to test and debug, and can negatively impact performance, and because conceptually, class fields are for data, not logic.

```
// bad
class extends React.Component {
  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv.bind(this)} />;
  }
}

// very bad
class extends React.Component {
  onClickDiv = () => {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />
  }
}

// good
class extends React.Component {
  constructor(props) {
    super(props);

    this.onClickDiv = this.onClickDiv.bind(this);
  }

  onClickDiv() {
```

```
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}
```

- Do not use underscore prefix for internal methods of a React component.

> Why? Underscore prefixes are sometimes used as a convention in other languages to denote privacy. But, unlike those languages, there is no native support for privacy in JavaScript, everything is public. Regardless of your intentions, adding underscore prefixes to your properties does not actually make them private, and any property (underscore-prefixed or not) should be treated as being public. See issues #1024, and #490 for a more in-depth discussion.

```
// bad
React.createClass({
  _onClickSubmit() {
    // do stuff
  },

  // other stuff
});

// good
class extends React.Component {
  onClickSubmit() {
    // do stuff
  }

  // other stuff
}
```

- Be sure to return a value in your `render` methods. eslint: `react/require-render-return`

```
// bad
render() {
  (<div />);
}

// good
render() {
  return (<div />);
}
```

Ordering

- Ordering for `class extends React.Component`:

1. optional `static` methods
2. `constructor`
3. `getChildContext`
4. `componentWillMount`
5. `componentDidMount`
6. `componentWillReceiveProps`
7. `shouldComponentUpdate`
8. `componentWillUpdate`
9. `componentDidUpdate`
10. `componentWillUnmount`
11. *event handlers starting with 'handle'* like `handleSubmit()` or `handleChangeDescription()`
12. *event handlers starting with 'on'* like `onClickSubmit()` or `onChangeDescription()`
13. *getter methods for `render`* like `getSelectReason()` or `getFooterContent()`
14. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
15. `render`

- How to define `propTypes`, `defaultProps`, `contextTypes`, etc...

```
import React from "react";
import PropTypes from "prop-types";

const propTypes = {
  id: PropTypes.number.isRequired,
  url: PropTypes.string.isRequired,
  text: PropTypes.string,
};

const defaultProps = {
  text: "Hello World",
};

class Link extends React.Component {
  static methodsAreOk() {
    return true;
  }

  render() {
    return (
      <a href={this.props.url} data-id={this.props.id}>
        {this.props.text}
      </a>
    );
  }
}

Link.propTypes = propTypes;
```

```
Link.defaultProps = defaultProps;

export default Link;
```

- Ordering for `React.createClass`: eslint: `react/sort-comp`

1. `displayName`
2. `propTypes`
3. `contextTypes`
4. `childContextTypes`
5. `mixins`
6. `statics`
7. `defaultProps`
8. `getDefaultProps`
9. `getInitialState`
10. `getChildContext`
11. `componentWillMount`
12. `componentDidMount`
13. `componentWillReceiveProps`
14. `shouldComponentUpdate`
15. `componentWillUpdate`
16. `componentDidUpdate`
17. `componentWillUnmount`
18. *clickHandlers or eventHandlers* like `onClickSubmit()` or `onChangeDescription()`
19. *getter methods for* `render` like `getSelectReason()` or `getFooterContent()`
20. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
21. `render`

`isMounted`

- Do not use `isMounted`. eslint: `react/no-is-mounted`

> Why? `isMounted` is an anti-pattern, is not available when using ES6 classes, and is on its way to being officially deprecated.

⬆ back to top