Writing clean code

credit: Chad Vernon's blog
Clean Code: A Handbook of Agile Software Craftsmanship
Clean Code: Writing Code for Humans

Don’t Repeat Yourself (DRY) principle

```
sphere = create_poly_sphere(name='left_eye')
assign_shader(sphere, 'blinn1')
parent_constrain(head, sphere)

sphere = create_poly_sphere(name='right_eye')
assign_shader(sphere, 'blinn2')
parent_constrain(head, sphere)
```

```
def create_eye(name, shader):
    sphere = create_poly_sphere(name=name)
    assign_shader(sphere, shader)
    parent_constrain(head, sphere)

create_eye('left_eye', 'blinn1')
create_eye('right_eye', 'blinn2')
```

Use descriptive names

```
curr= os.environ.get('CURRENT_CONTEXT')
if curr:
    cl= curr.split('/')
self.__curr= [None] * 6
self.setType( cl[0] )
self.setSeq( cl[1] )
if len( cl ) > 3:
    self.setSubseq( cl[2] )
    self.setShot( '/'.join( cl[2:] ) )
else:
    self.setShot( cl[-1] )

if wa: self.__wa= wa
else: self.__wa= os.environ.get('CURRENT_WORKAREA')
```

Use descriptive names

```
context_type, sequence, subsequence, shot = self.get_current_context()
self.set_type(context_type)
self.set_sequence(sequence)
if subsequence:
    self.set_subsequence(subsequence)
if shot:
    self.set_shot(shot)
self.__work_area = work_area if work_area else self.get_current_work_area()
```

Copy

Naming Classes

Bad class names include:

- ShapeIE
- Utility
- Common
- MyFunctions

Good class names include:

- ShapeExporter
- RigPublisher
- Project
- User

Naming Functions & Methods

Bad method names include:

- proc_new
- pending
- process1
- process2

Good method names include:

- create_process
- is_pending
- send_notification
- import_mesh

More on scope & naming of Classes and Functions/Methods

- Classes are nouns.
- Classes should be named as simply and as specifically as possible.
- If you can’t make the name specific, consider splitting it into two classes.
- Classes should have a single responsibility.

- Functions & Methods are verbs.
- Functions & Methods should only return what the name describes.
  - Everything else is called a side effect and should be avoided.
- Functions & Methods should do one thing (if it’s two or more, split into separate functions).

Avoid abbreviations

Bad names:

- sjData
- jid
- sjid
- nm
- sjState

Good names:

- subjob_data
- job_id
- subjob_id
- name
- subjob_state

Naming booleans

Bad boolean names:

- open
- status
- login

Good Boolean names:

- is_open
- logged_in
- is_valid
- enabled
- done

Working with booleans

Compare booleans implicitly:

```
# Don't do this
if (is_valid == True):
    # do something

# Instead do this
if is_valid:
    # do something
```

Working with booleans

Assign booleans implicitly:

```
# Don't do this
if len(items) == 0:
    remove_entry = True
else:
    remove_entry = False

# Instead do this
remove_entry = len(items) == 0
```

Working with booleans

Avoid boolean names that signify negative values:

```
# Don't do this
if not not_valid:
    pass

# Instead do this
if valid:
    pass
```

Use ternaries

For simple if/then/else constructions:

```
# Don't do this
if height > height_threshold:
    category = 'giant'
else:
    category = 'hobbit'

# Instead do this
category = 'giant' if height > height_threshold else 'hobbit'
```

Use dictionary to replace case/switch statements

For complicated if/then/else constructions with categories:

```
def f(x):
    return {
        'a': 1,
        'b': 2
    }.get(x, 9)    # 9 is default if x not found
```

Don’t use strings as types

```
if component_type == 'arm':
    # do something
elif component_type == 'leg':
    # do something else
```

```
class ComponentType(object):
    arm = 'arm'
    leg = 'leg'

if component_type == ComponentType.arm:
    # do something
elif component_type == ComponentType.leg:
    # do something else
```

Don’t use magic numbers

```
if run_mode < 3:
    run_mode = 5
elif run_mode == 3:
    run_mode = 4
```

```
class JobStatus(object):
    waiting = 1
    starting = 2
    running = 3
    aborting = 4
    done = 5

    def __init__(self, value=JobStatus.waiting):
        self.status = value

    def not_yet_running(self):
        return self.status < JobStatus.running

    def abort(self):
        if self.not_yet_running():
            self.status = JobStatus.done
        elif self.status == JobStatus.running:
            self.status = JobStatus.aborting

# job_status is the new run_mode
job_status.abort()
```

Writing clean functions

- Strive for 0-3, parameters, with maximum of 7-9
  - The more parameters, the harder it is for the reader of the code to understand
- Shorter is better, < 100 lines of code
- Indicators that your function may be too long:
  - You need to separate sections of code with comments or whitespace
  - Scrolling is required to view all the code
  - The function is hard to name
  - There are conditionals several layers deep
  - There are more than 7 variables and parameters in the scope at any one time

Writing clean functions: Extracting a method

```
# Instead of this
if something:
    if something_else:
        while some_condition:
            # do something complicated

# Do this instead
if something:
    if something_else:
        do_complicated_things()

def do_complicated_things():
    while some_condition:
        # do something complicated
```

Writing clean functions: Return early

```python
# Instead of this
def validate_mesh(mesh):
    result = False
    if has_uniform_scale(mesh):
        if has_soft_normal(mesh):
            if name_is_alphanumeric(mesh):
                result = name_is_unique(mesh)
    return result

# Do this
def validate_mesh(mesh):
    if not has_uniform_scale(mesh):
        return False
    if not has_normal(mesh):
        return False
    if not name_is_alphanumeric(mesh):
        return False
    return name_is_unique(mesh)
```

Writing clean functions/methods: proximity

```
def add_take():
    if not validate_take():  # First method referenced should be directly below
        raise ValueError('Take is not valid')
    save_take()  # Second method referenced should be below first

def validate_take():
    return take.endswith(‘.mov’)

def save_take():
    # save in database
```

Writing clean Classes

Should read like an outline, with higher levels of abstraction above fine-grained details

- Class
  - Method 1
    - Method 1a
      - Method 1ai
      - Method 1aii
    - Method 1b
    - Method 1c
  - Method 2
    - Method 2a

Writing clean Classes

Strive for “high cohesion”:

- all functionality in the class is closely related
- methods interact with the rest of the class
- class has attributes used by multiple methods
- class can be re-used frequently over time

Writing clean Classes: Cohesion

```python
# Low cohesion class
class Vehicle(object):

    def edit_options():
        pass

    def update_pricing():
        pass

    def schedule_maintenance():
        pass

    def send_maintenance_reminder():
        pass

    def select_financing():
        pass

    def calculate_monthly_payment():
        pass
```

```python
# High cohesion classes
class Vehicle(object):
    def __init__(self)

    def edit_options():
        pass

    def update_pricing():
        pass

class VehicleMaintainer(object):
    def schedule_maintenance():
        pass

    def send_maintenance_reminder():
        pass

class VehicleFinancer(object)
    def select_financing():
        pass

    def calculate_monthly_payment():
        pass
```

Comments: Don’t be redundant

Don’t be redundant - clean code will require fewer comments!

```
# Clear the node combo box then add items
self.node_combobox.clear()
if nodes:
    # Sort the nodes
    nodes.sort()
    # Check to see if there is a shape controller associated with the node
    self.find_shape_controllers(nodes)

    # Now add the list of nodes to the combo box
    self.node_combobox.addItems(nodes)

    # If a node is specified set the combo box
    if node:
        # Find the node's index
        index = self.node_combobox.findText(
            node,
            QtCore.Qt.MatchExactly | QtCore.Qt.MatchCaseSensitive)
        self.node_combobox.setCurrentIndex(index)
```

Comments: Dividers

Sign that you should divide into more functions/classes

```
# Now create the new group object and insert it into the table
# ------------------------------------------------------------------
# Create the group object
group = slidergroup.SliderGroup(name)
self._slider_groups[name] = group
# Tell the group what its start row is
group.setRow(row)
# Apply color
if color:
    group.setColor(color)

# Generate sliders from the attributes attached to the group
# ------------------------------------------------------------------
row_index = row + 1
sliders_to_add = []
for attr in attributes:
    if cmds.objExists(attr):
        slider = self.add_slider(attr, rowIndex, group)
        row_index += 1
```

Examples of nice code from past student projects

- https://github.com/ewellinger/election_analysis/blob/master/NMF_Clustering.py
- https://github.com/erindesmond/ABC-MUSIC/blob/master/src/lstm_class.py
- https://github.com/ecgill/flip_risk_indexer/blob/master/src/run.py