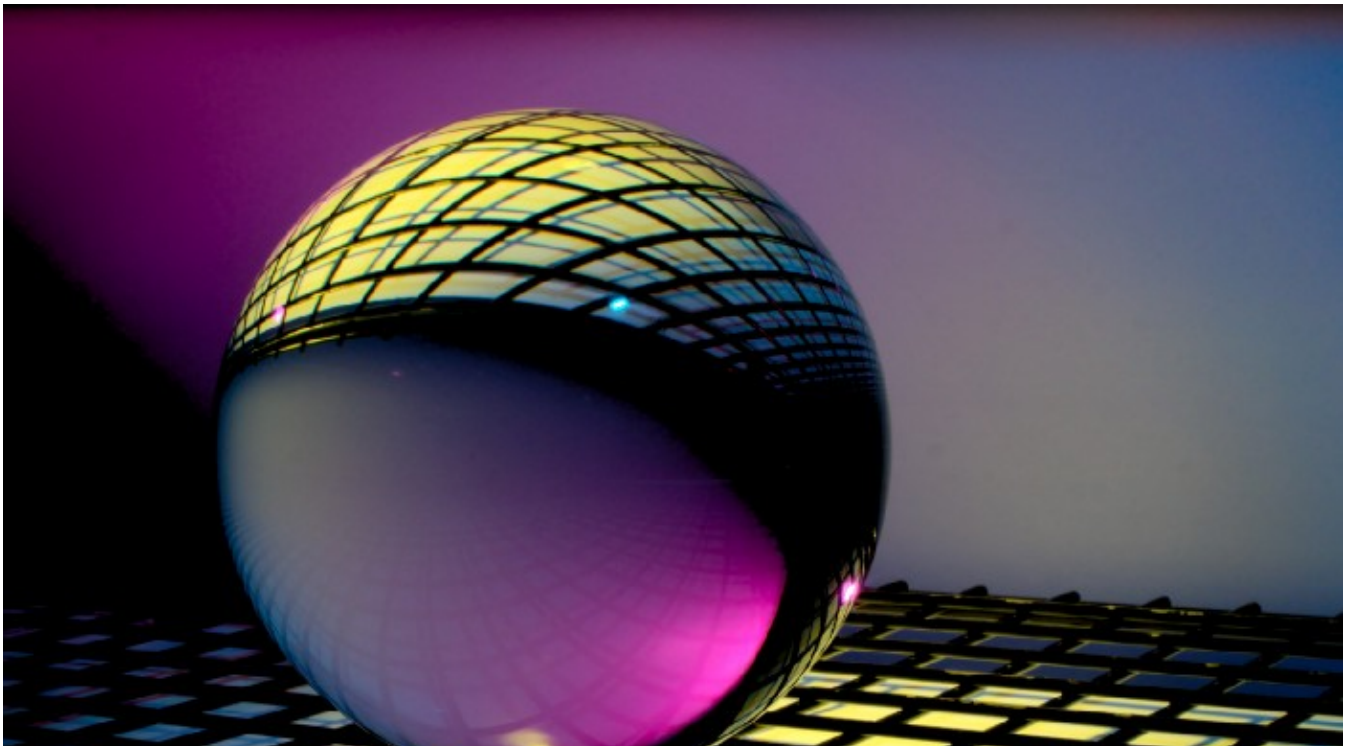
32 Advanced Techniques for Better Python Code

Tips for Python documentation, coding, testing, verification, and continuous integration

Bruce H. Cottman, Ph.D.

Mar 4 · 11 min read ★



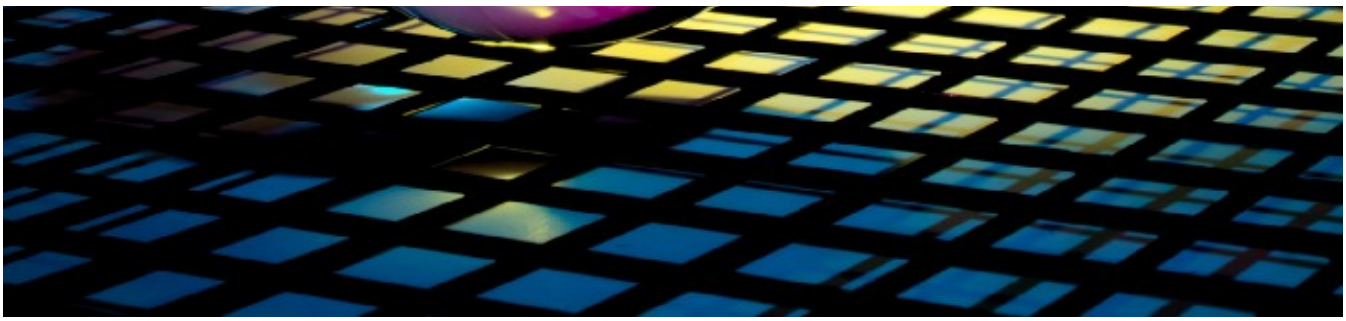


Photo by Michael Dziedzic on Unsplash

I've pulled together these techniques from 35 years of programming in multiple languages on multiple projects. There are before and after Python code examples that apply each technique.

The techniques divide into five categories:

- Documentation techniques
- Coding techniques
- Testing techniques
- Verification techniques
- Continuous integration (CI) techniques

But first, I'll introduce the projects or code repositories we'll use to apply these techniques.

What's PHOTONAI?

PHOTONAI incorporates scikit-learn and other machine learning (ML) or deep learning (DL) frameworks with one unifying paradigm. PHOTONAI adopts scikit-learn's `Transformer` class method architecture.

PHOTONAI adds code that reduces manual coding and errors by transforming pre- and post-learner algorithms into elements with their argument signature. Examples of elements are several data cleaners, scalers, imputers, class balancers, cross-validators, hyper-parameter tuners, and ensembles.

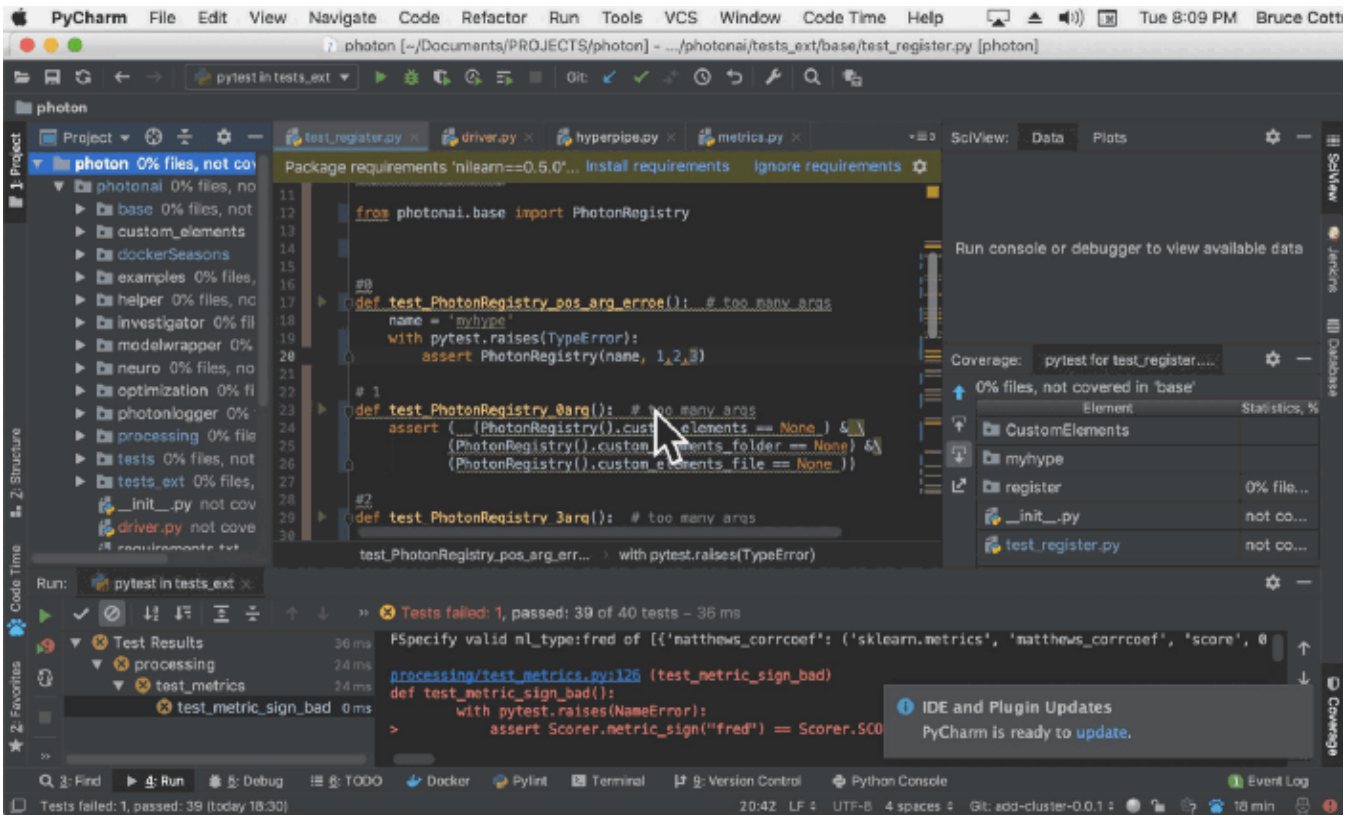
I usually give code examples from before and after Python to show the transformation of PHOTONAI using the technique.

Documentation techniques

Technique #1: Create new version, and document major changes

- Create a new version, **a+1.0.0,** if there are a significant amount of architectural changes, a significant amount of code changed, and/or a significant amount of new behavior added.
- Create a new subversion, **a.b+1.0**, if there are minor architectural changes, a minor amount of code changed, and/or a minor amount of new behavior added.
- Create a new sub-subversion, **a.b.c+1**, if adding a minor amount of code and/or there are bug fixes.

I use PyCharm to `git add` all change and new files of the project PHOTONAI 1.1.0.



PyCharm git add for all changed files in project PHOTONAI 1.1.0. Animation by Rachel Cottman.

Note: You can `git add` a single file in the same manner as a project. Left-click on the file name instead of the project directory name.

The changes shown here result in a new subversion, 1.1.0 *from 1.0.0). I use Git.

```
git checkout -b 1.1.0
git status
```

Terminal output:

```
On branch 1.1.0

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)
modified:   photonai/base/registry/element_dictionary.py
modified:   photonai/base/registry/registry.py
new file:   photonai/custom_elements/CustomElements.json
new file:   photonai/driver.py
modified:   photonai/processing/metrics.py
modified:   photonai/tests/processing_tests/metrics_test.py
new file:   photonai/tests_ext/CustomElements/CustomElements.json
new file:   photonai/tests_ext/base/test_hyperpipe.py
new file:   photonai/tests_ext/base/test_register.py
new file:   photonai/tests_ext/processing/test_metrics.py
new file:   photonai/util.py
```

Note: The above is a list of all files changed or added for the project PHOTONAI 1.1.0.

Technique #2: Document locally any significant addition or change in code

```
class Scorer(object):
    """
    Transforms a string literal into a callable instance of a
    particular metric
    BHC 1.1.0 - support cluster scoring by add clustering
    scores and type
            - added METRIC_METADATA dictionary
            - added ML_TYPES = ["Classification", "Regression",
                                "Clustering"]
            - added METRIC_<>ID Postion index of metric
              metadata list
            - added SCORE_TYPES
            - added SCORE_SIGN
    """
```

Technique #3: Create long and descriptive names for constants, variables, functions, and class methods

Technique #4: Transform behaviorally inappropriate comments into behaviorally correct comments to lower maintenance cost and bug generation

Technique #5: Add comments to increase the readability of code

Comments have a maintenance cost, and inappropriate comments have a higher maintenance cost.

Comments are one part of the code. They have a maintenance cost associated with any behavior change.

The comment before the change:

```
@staticmethod
def get_json(photon_package):
"""
Load JSON file in which the elements for the PHOTON submodule are
stored.

The JSON files are stored in the framework folder by the name
convention 'photon_package.json'

Parameters:
-----------
* 'photon_package' [str]:
  The name of the photonai submodule

Returns:
--------
JSON file as dict, file path as str
"""
```

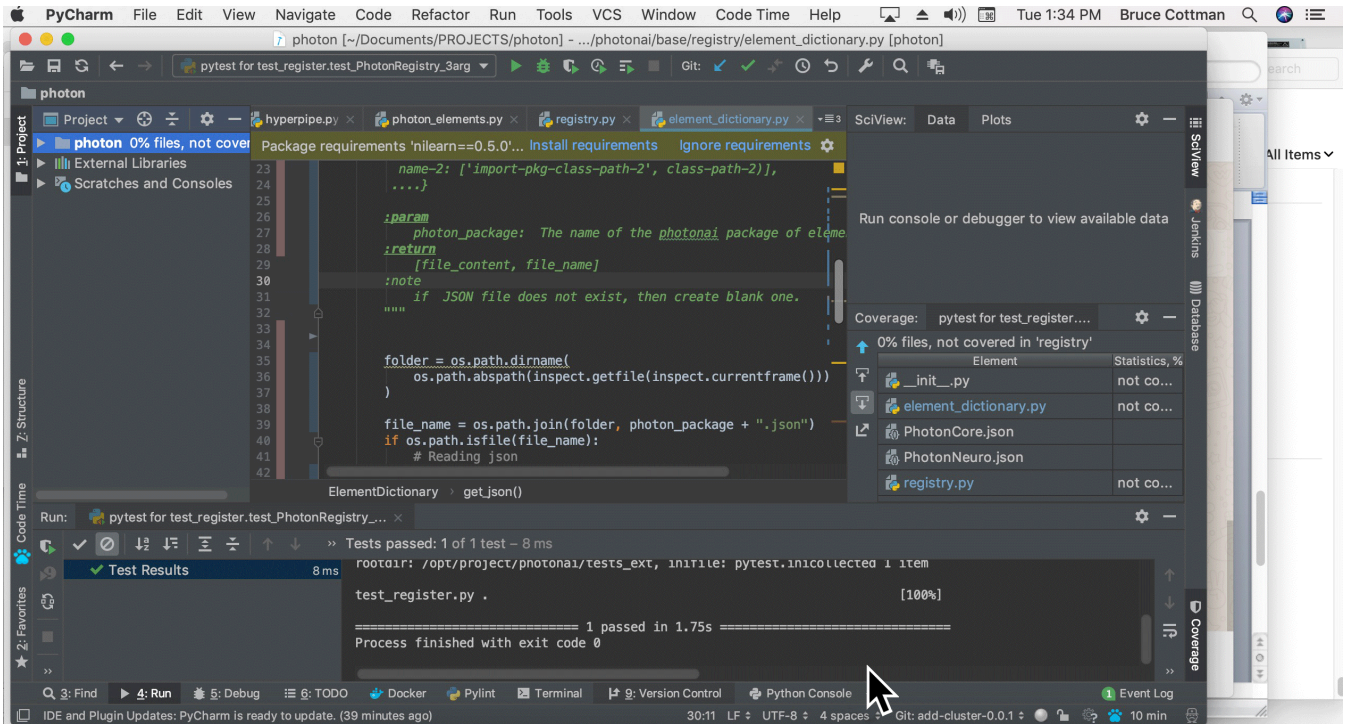
The comment after a change:

```
@staticmethod
def load_json(photon_package: str) -> Any:
    """
    load_json Loads JSON file.

The class init PipelineElement('name',...)
    stores the element metadata in a json file.

The JSON files are stored in the framework folder
    by the name convention 'photon_<package>.json'.
```

```
(example:$HOME/PROJECTS/photon/photonai/base/registry/PhotonCore.jso
n)

The file is of format
    { name-1: ['import-pkg-class-path-1', class-path-1)],
      name-2: ['import-pkg-class-path-2', class-path-2)],
     ....}

Parameters
    ----------
        photon_package:  The name of the photonai package of element
metadata
    Returns
    -------
        [file_content, file_name]
    Notes
    -------
    if  JSON file does not exist, then create blank one.
"""
```

Technique #6: Choose a docstring style, and use it consistently throughout the project (package, library, …). We use NumPy Style because it is suited for the detailed documentation of classes, methods, functions, and parameters.

Note: The PyCharm docstring format is set from the top ribbon sequence PyCharm | Preferences | Tools | Python Integrated Tools, as shown in the following GIF. I am on a Macintosh. It is different on Windows or Linux.



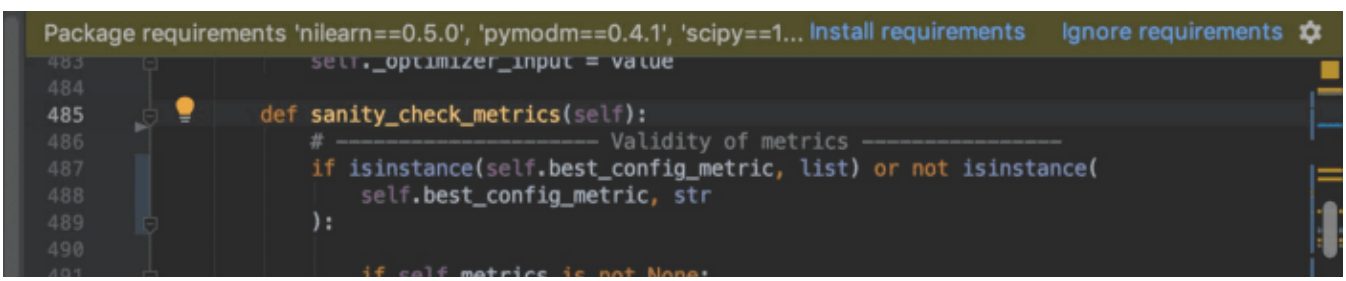
PyCharm set the docstring style for the project PHOTONAI 1.1.0. Animation by Rachel Cottman.

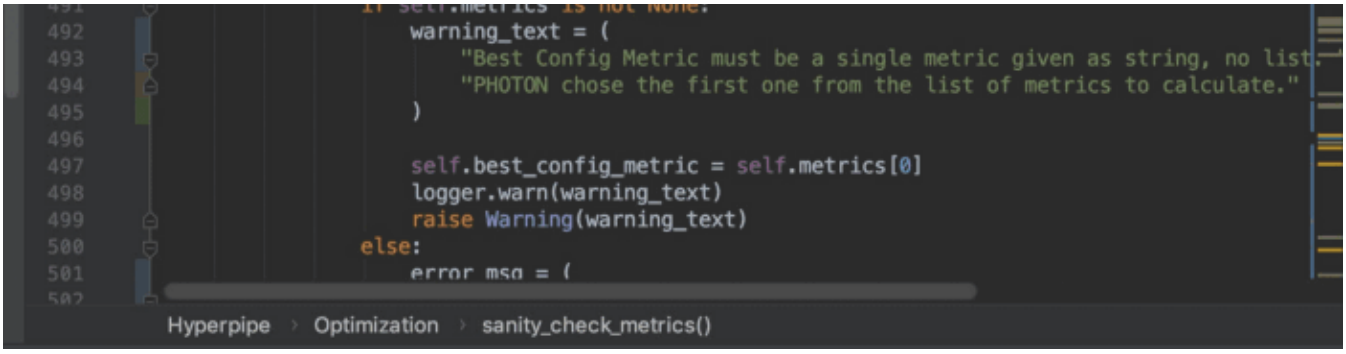
Using NumPy Style and correcting for any gaps, the comment becomes:

```
@staticmethod
def get_json(photon_package: str) -> Any:
    """
    get_json Loads JSON file.

The class init PipelineElement('name',...)
    stores the element metadata in a json file.

The JSON files are stored in the framework folder
    by the name convention 'photon_<package>.json'.

(example:$HOME/PROJECTS/photon/photonai/base/registry/PhotonCore.jso
n)

The file is of format
    { name-1: ['import-pkg-class-path-1', class-path-1)],
      name-2: ['import-pkg-class-path-2', class-path-2)],
     ....}

Parameters
    ----------
    photon_package:  The name of the photonai package
    of element metadata.

Returns
    -------
    [file_content, file_name]

Notes
    -------
    If  JSON file does not exist, then create a blank one.
    """
```

Note: Type hints create less complicated documentation of the signature of the method.

Note: PyCharm enters the NumPy docstring template. PyCharm only inserts `return` because the function has no parameters. If you right-click, it lists all allowed NumPy docstring keywords. Cool!





Technique #7: Apply PEP-8 naming conventions

In the below example, a global constant, `ELEMENT_TYPE`, is uppercase, and the variable `machine_learning_type` is lowercase.

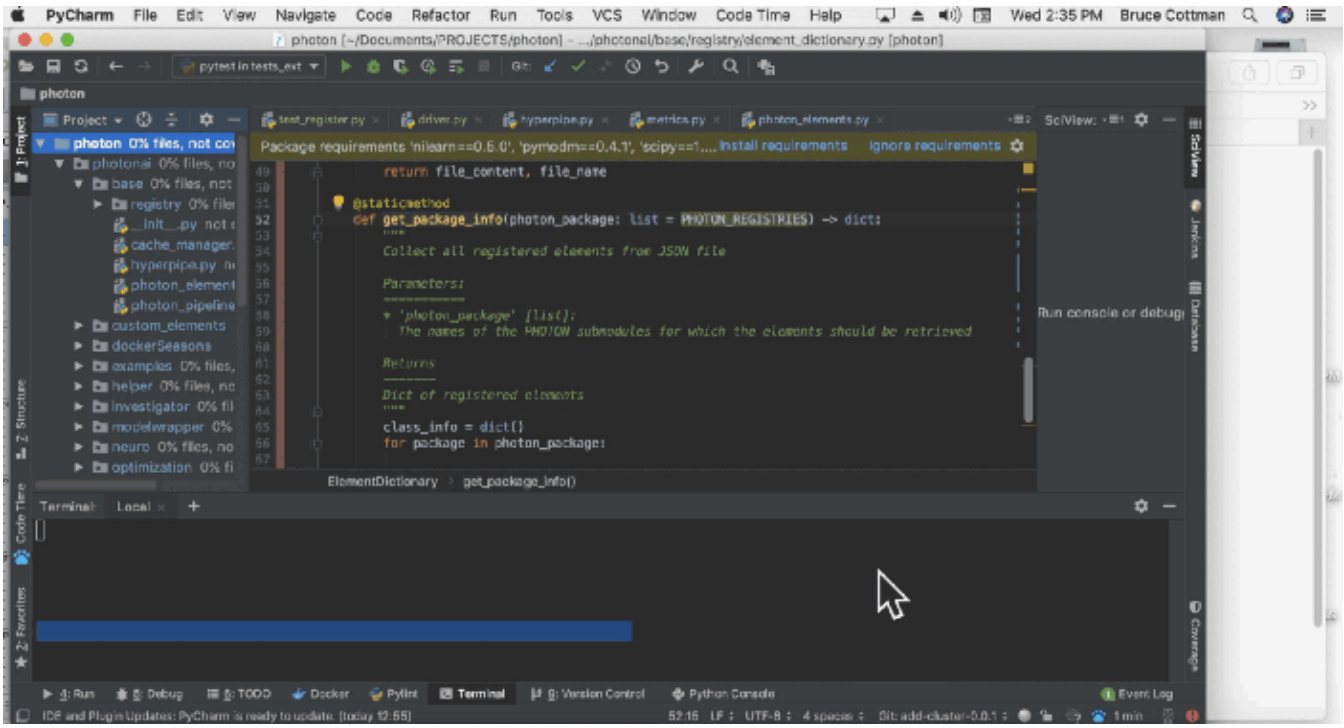
Globals:

```
ELEMENT_TYPE -> ML_TYPE
ELEMENT_DICTIONARY   ->  METRIC_METADATA
```

variables:

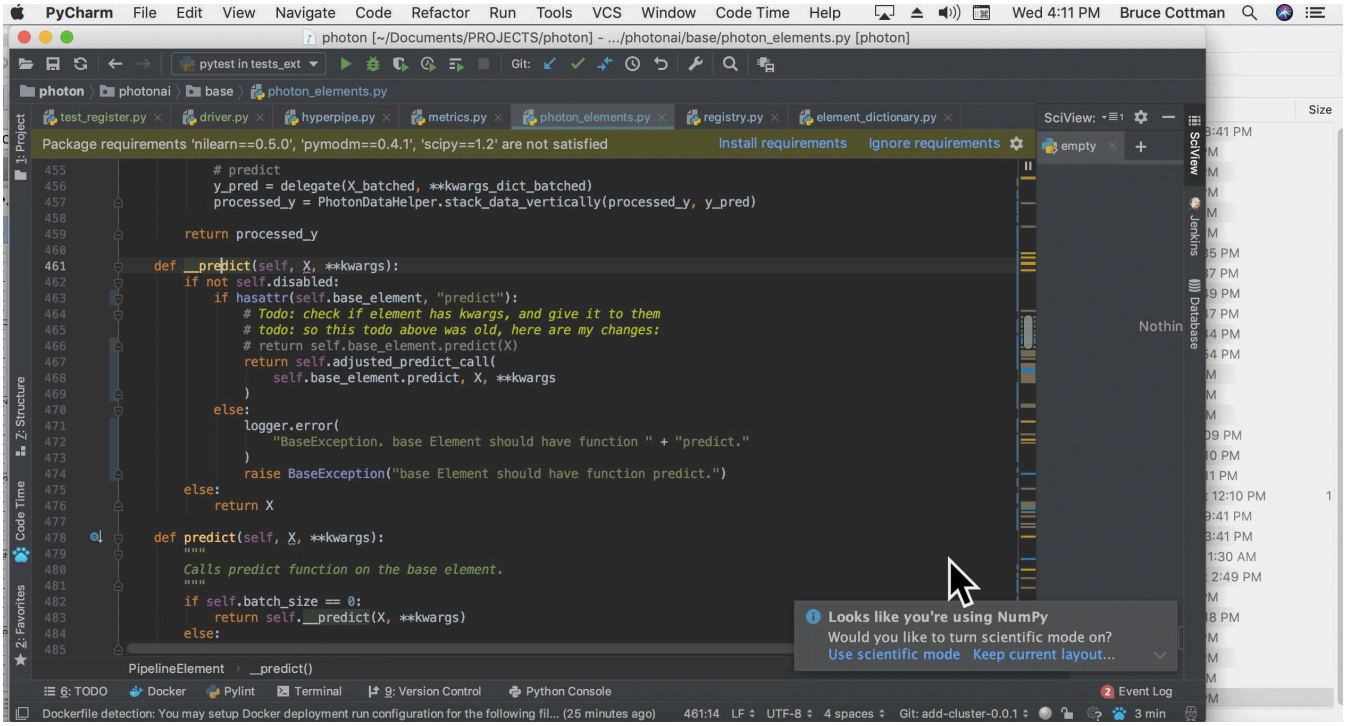
machine_learning_type -> element_type

Note: PyCharm can run an external tool for formatting. I use black, which formats a `.py` file or the entire project into a PEP-8 compliant format. The result is that all files are formatted the same. The follow-on result is an increased readability score.



PyCharm external tool: black applied to all .py files of the project PHOTONAI 1.1.0. Animation by Rachel Cottman.

Note: PyCharm can change names wherever used in a package. I understand that the word refactor may be considered harmful by some. However, changing a name throughout the project is useful. Please ignore how PyCharm categorizes its features. Use `rename` from the beginning so that you don't have to refactor your code!



PyCharm git rename feature. Animation by Rachel Cottman.

Note: PyCharm `find usage` finds all the files in your project reference — e.g., a constant, variable, function, class, or class method.

PyCharm Find Usages feature. Animation by Rachel Cottman.

Technique #8: Add type hinting to every function or class method.

Python is a dynamically typed language. Python 3.5 and higher allows you to enable type hints (PEP 484). I want to emphasize the word *hints* because type hints do not affect the Python interpreter. As far as you are concerned, the Python interpreter ignores type hints.

Type hints (note: not strong type checking) make it possible to accomplish bug hunting, finding security holes, and static type checking after the first pass of coding and unit testing.

```
def is_machine_learning_type(ml_type: str) -> bool:
```

Note: The docstring no longer needs each `arg` and the `return` datatype to be documented. The signature becomes self-documenting.

Note: The readability, as well as information of post-code-checking tools, has increased.

Technique #9: Transform irrelevant comments to relevant comments

Version 1.0.0 (original)

```
# register new object
PhotonRegister.save("ABC1", "namespace.filename.ABC1",
"Transformer")
```

Version 1.1.0

```
# register new element
PhotonRegister.register("ABC1", "namespace.filename.ABC1",
"Transformer")
```

Technique #10: Keep only architectural `#todo` comments

Note: PyCharm has found all `#Todo` for the project PHOTONAI 1.1.0.

PyCharm find all Todo of the project PHOTONAI 1.1.0. Animation by Rachel Cottman.

Technique #11: Delete old code that is commented out.

Version 1.1.0 (before commented out 1.1.0 code):

```
def __post_init__(self):
    if self.custom_elements_folder:
        self._load_custom_folder(self.custom_elements_folder)

# base_PHOTON_REGISTRIES = ["PhotonCore", "PhotonNeuro"]
# PHOTON_REGISTRIES = ["PhotonCore", "PhotonNeuro"]

# def __init__(self, custom_elements_folder: str = None):
#     if custom_elements_folder:
#         self._load_custom_folder()
#     else:
#         self.custom_elements = None
#         self.custom_elements_folder = None
#         self.custom_elements_file = None
```

Version 1.1.0 (after removal of commented out 1.1.0 code):

```
def __post_init__(self):
    if self.custom_elements_folder:
        self._load_custom_folder(self.custom_elements_folder)
```

Note*:* Lines-of-code (LOC) complexity is reduced by elimination of commented-out old code. The readability score is increased.

Coding Techniques

Technique #12: Don't reinvent the wheel

Before you write a line of code for your idea or task, search for solutions that others have coded. There are *awesome* repositories out there. Discover your favorites.

To get you started, here are some of my favorites:

- Papers with code
- Awesome Git
- Awesome lists
- Awesome Awesomeness (Programming Languages)
- Awesome Machine Learning
- Vision Machine Learning

Technique #13: Learn from others' code.

The packages NumPy, pandas, Keras, fast.ai, TensorFlow, PyTorch, Kubernetes, Hadoop, PySpark, and hundreds of other packages teach advanced computer science and the best coding techniques.

Take advantage of the open source universe of Python and other languages.

Technique #14: Log; don't print

Technique #15: Do not code global data structures; use parameter files

Logging and parameter files are subjects that are lightly broached or not taught at all in your brick-and-mortar or online classes.

You either learn about logging and parameter files in the school of hard knocks or, if you're lucky, by watching a senior software engineer (or by using techniques 12 and 13).

I detail the advantages of logging and parameter files in the following article:

Technique #16: Create a function or method to encapsulate value checking

The example below creates a method to check if a value is in a global constant.

```
@staticmethod
def is_machine_learning_type(ml_type: str) -> bool:
    """
    :raises
    if not known machine_learning_type

:param machine_learning_type
    :return: True
    """
    if ml_type in Scorer.ML_TYPES:
        return True
    else:
        logger.error(
            "Specify valid ml_type to choose best config:
{}".format(ml_type)
        )
    raise NameError(becomes
```

Technique #17: Create the package error type

For the package Photoai, we created the error type `Photonai`.

Technique #18: Create a raise function with the package error type

For the error type, we created the function `raise_PhotonaiError`.

```
import logging

class PhotoaiError(Exception):
    pass

def raise_PhotoaiError(msg):
    logger.error(msg)
    raise PhotoaiError(msg)
```

Method `is_machine_learning_type` becomes:

```
@staticmethod
def is_machine_learning_type(ml_type: str) -> bool:
    """
    Parameters:
    -----------
    machine_learning_type

Returns
   -----------
   True

Raises
   -----------
   if not known machine_learning_type
   """
   if ml_type in Scorer.ML_TYPES:
        return True

raise_PhotoaiError(
        "Specify valid ml_type:{} of [{}]".format(ml_type,
        Scorer.ML_TYPES))
```

Note: `raise_PhotoaiError` combines the log and runtime error. The case of *not* `true` halts and displays the stack. As a runtime error, it behaves as we would expect if it passes an unknown value. Notice the improvement in readability and maintenance costs.

Technique #19: Use `faulthandler`

Starting with Python 3.3, a library called fault handler displays the calling stack tracebacks on an error or with a Python segfault.

Technique #20: Eliminate global variables — or at least try

You use `globals()` to list out all globals in your package.

Technique #21: Eliminate all unused local variables

Usually, your IDE identifies all dangling variables for you. If not, you use `locals()` and use search.

Technique #22: Apply the Python 3.7+ `@dataclass` decorator before the class definition.

`@dataclass` is an added feature of Python 3.7. The main driving force was to eliminate boilerplate associated with the state in a `def class` definition.

`@dataclass` decorates a `def class` definition and automatically generates the five double dunder methods: `__init__()`, `__repr__()`, `__str__`, `__eq__()`, and `__hash__()`.

`@dataclass` virtually eliminates repetitive boilerplate code required to define a basic class. Notice that all these five double dunder methods work directly with the class's encapsulated state.

```
class PhotonRegistry:
    """
    Helper class to manage the PHOTON Element Register
    ...
    """

base_PHOTON_REGISTRIES = ['PhotonCore', 'PhotonNeuro']
PHOTON_REGISTRIES = ['PhotonCore', 'PhotonNeuro']

def __init__(self, custom_elements_folder: str = None):
    if custom_elements_folder:
        self._load_custom_folder(custom_elements_folder)
    else:
        self.custom_elements = None
        self.custom_elements_folder = None
        self.custom_elements_file = None
```

After the `@dataclass` decorator:

```
@dataclass
class PhotonRegistry:
    """
    Helper class to manage the PHOTON Element Register
    ...
    """
    custom_elements_folder: str = None
    custom_elements: str = None
    custom_elements_file: str = None
    base_PHOTON_REGISTRIES: ClassVar[List[str]] =/
   ["PhotonCore", "PhotonNeuro"]
    PHOTON_REGISTRIES: ClassVar[List[str]] =/
   ["PhotonCore", "PhotonNeuro"]

def __post_init__(self):
    if self.custom_elements_folder:
        self._load_custom_folder(self.custom_elements_folder)
```

A detailed blog article on the use of `dataclass`:

> 12 Examples of How To Write Better Code Using @dataclass
>
> We are adding clustering algorithms using scikit-learn, Keras, and other packages to the Photonai package. With twelve...
>
> towardsdatascience.com

Note: Readability has increased substantially by using `@dataclass` with `type hinting`.

Testing Techniques

Technique #23: The developers of the code develop unit tests

Note: Set the tool pytest in PyCharm.

Setting the tool pytest in PyCharm for the project PHOTONAI 1.1.0. Animation by Rachel Cottman.

Technique #24: Unit tests have 80%+ coverage. We use coverage.

Technique #25: Integration tests must have 100% coverage of all external (API) functions, classes, class attributes, and class methods.

Technique #26: The developers do *not* accomplish acceptance testing.

For example, we selected these unit tests from the 24 unit tests in `test_metrics.py`.

```
# 22
def test_is_machine_learning_type_bad():
    with pytest.raises(PhotoaiError):
        assert Scorer.is_machine_learning_type("fred")
# 23
def test_is_machine_learning_type():
    assert Scorer.is_machine_learning_type("Clustering")
```

Note: Once the test tool is selected, PyCharm marks individual runnable tests. These are marked with a filled arrow in the left bar. Example of individual testing using pytest:

With PyCharm, you can run a test file or an entire test folder using pytest, and the results show. Reference bottom panel-ribbon (passed 39/40 test — 32 ms).

PyCharm test tool specified for project PHOTONAI 1.1.0. Animation by Rachel Cottman.

Technique #27: Code type hinting checking.

Type hints are just hints that third-party tools like Mypy can use to point out potential bugs.

Technique #28: Test coverage

Technique #29: Performance profiling

Technique #29:Check security

Technique #29:Code reliability

The following article details tools you can use for test coverage, performance profiling, checking input data security, and code reliability.

Verification Techniques

Technique #30: Code review

All the package code is reviewed before the push to the master VCS. A developer can review their code using the tool Codacy.

Technique #31: Version control

Choose a version control system and use it. We use GitHub and GitLab. We're required to push every day, and local commits are usually several times a day. The frequency of commits is left to the developer to decide.

Note: PyCharm has a `local history`. Click-right in any file and PyCharm pulls up file changes since the launch of PyCharm. `local history` is a very light VCS that has saved many developers.

PyCharm Local (file) History feature. Animation by Rachel Cottman.

Note: PyCharm can add a file, folder, or an entire package to the VCS. I use Git in the following animation.

PyCharm git add for all changed files in the project PHOTONAI 1.1.0. Animation by Rachel Cottman.

DevOps for Your Sandbox

Technique #32: Continuous integration (CI) automation

I cover a development operation pipeline (DevOps) specification in Python code in the article below. You can use this code to transform your favorite interactive development environment (IDE) into a DevOps tool.

Transform Your Favorite Python IDE Into a DevOps Tool Capturing CI/CD in customizable Python code betterprogramming.pub	

Summary

My like-minded software engineers and I appreciate PyCharm because it enables a good hunk of our development pipeline to be automated without learning a CI/CD package. The natural clustering of objects in PyCharm results in Technique #32.

If you use PyCharm, hopefully the animations help you with your PyCharm setup and usage.

I have given only Python examples, but most of these techniques apply to other languages.

All 32 techniques result in better understandability. You'll have a better understanding ofthe results, better testing, less bugs, and a lower maintenance cost.

Python Zen masters use these techniques for better readability.

Happy coding!