Python Code Review

May 8, 2021

0.1 General

- (**Item:2**) Follow PEP8
  - use in line negation (`if a is not b`) instead of negation of postive expressions (if not a is b)
  - try to reduce level of `nesting` using `classes`, `generators`, `etc.`
  - imports should be in a section in the following order: `standard library` modules, `third-party` modules, `your` own modules. Each subsection should have imports in alphabetical order
  - in loops use `_` for unused variables
  - try to return function instead of calling function and then returning it
  - try combining exception handling
  - try to make your function generalizable as possible
  - use `long` and descriptive variable names
  - optimize `if` statements to ensure failure occurs as quickly as possible
- (**Item 5**) Write `Helper Functions` Instead of Complex Expressions
- (**Item 6**): Perfer Multiple Assignment `Unpacking` over Indexing
- (**Item 7**): Perfer `enumerate` over `range`
- (**Item 8**): Use `zip` top process Iterators in Parallel
- (**Item 9**): Avoid `else` Blocks After `for` and `while` Loops
- (**Item 10**): Prevent Repetition with Assignment Expressions / `Walrus Operator`

0.2 Lists and Dictionaries

- (**Item 12:**) Avoid `Striding` and `Slicing` in a Signle Expression
  - If you need all three parameters, consider doing two assignments (one to stride and another to slice) or using `islice` from the `itertools`
- (**Item 13:**) Perfer `Catch-All` Unpacking Over Slicing (`*unpacking`)
- (**Item: 14**) Sort by `Complex` Criteria Using the `Key` Parameter
  - tuples have built in `__it__` and you can compare them
  - sort(key=lambda x: (x.weight, -x.name)
- (**Item: 16**) Perfer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys
- (**Item: 17**) Perfer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State
  - try to use `if (names := votes.get(key)) is None:`
- (**Item: 18**) Know How to Construct Key-Dependent Default Values with `__missing__`

0.3 Functions

- (**Item: 19**) Never Unpack More Than Three Variables from Functions

- (**Item: 20**) Perfer Raising Exceptions to Returning `None`
- (**Item: 22**) Reduce Visual Noise with Variable Positional Arguments (`*args`)
  - not good pratice to use this with generators
- (**Item: 23**) Provide Optional Behavior with Keyword Arguments (`**kwargs`)
- (**Item: 24**) Use `None` and Docstrings to Specify Dynamic Default Arguments
  - during function definition at module load time. This can cause odd behaviors for dynamic values (like {}, [], or datetime.now())
- (**Item: 25**) Enforce Clarity with Keyword-Only and Positional-Only Arguments
  - `safe_division_d(x, y, /, *, found=False, ignore=False)`
- (**Item: 26**) Define Function Decorators with `functools.wraps`

0.4 Comprehensions and Generators

- (**Item: 27**) Use Comprehensions Instead of `map` and `filter`
- (**Item: 28**) Avoid More Than Two Control Subexpressions in Comprehension
  - meaning have two `for` loops or one for loop and one `if`
- (**Item: 29**) Avoid Repeated Work in Comprehensions by Using Assignment Expression
- (**Item: 30**) Consider `Generators` Instead of Returning `Lists`
- (**Item: 31**) Be Denfensive when Iterative Over `Arguments`
- (**Item: 32**) Consider `Generator Expressions` for Large `Lists` Comprehensions
  - Generator expressions execute very quickly when chained together and are memory efficient
- (**Item: 33**) Compose Multiple Generators with `yeild from`
- (**Item: 35**) Avoid Causing State Transitions in Generators with ‘throw
- (**Item: 36**) Consider `itertools` foe working with iterators and generators
  - `chain`, `repeat`, `cycle`, `tee`, `zip_longest`, `islice`, `takewhile`, `dropwhile`, ‘,filterfalse,accumulate,product,permutations,combinations‘

0.5 Classes and Interfaces

- (**Item: 37**) Compose Classes Instead of Nesting Many Levels of Built-in Types
  - bascially if you have to futher then one level of nesting, i.e, a dictionay in a tuple or a tuple in a dictionary , re-think aprpoach
  - it is time to use classes to create a layer of abstraction between your interfaces and concrete implementations
  - use `namedtuple` for lightweight immutable data containers before you need the flexibility of a full class
  - move your code to using multiple classes when you internal state dictionaries get complicated
- (**Item: 38**) Accept Functions instead of Classes for Simple Interfaces
  - you can pass fuction or class methods to functions as API hooks
  - using a helper class to provide the behavior of a stateful closure is clearner
  - when you need a function to maintain state, consider defining a class that provides the`__cal__` method instead of defining a stateful closure
- (**Item: 39**) Use `@classmethod` Polymorphism to Construct Objects Generically
  - Use `@classmethod` to define alternative constructors for your classes
  - Use class method polymorphism to provide generic ways to build and connect many concrete subclasses

  - essentially what this will allow you to do is generically connect and initialize things like `mapreduce`
- (**Item: 40**) Initialize Parent Classes with `super`
  - use `.mro()` to see order of function calls
- (**Item: 41**) Consider Composing Functionality with Mixin Classes
- (**Item: 42**) Perfer Public Attributes over Private Ones
  - Use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes.
- (**Item: 43**) Iherit from `collections.abc` for Custom Container Types

0.6 Metaclasses and Attributes

- (**Item: 44**) Use Plain Attributes Instead of `Setter` and `Getter` Methods
  - Use `@property` to define special behavior when attributes are `geters` and `seters`
  - Ensure that `@property` methods are fast; for slow or complex work— especially involving I/O or causing side effects—use normal methods instead
- (**Item: 45**) Consider `@property` Instead of Refactoring Attributes
  - dont overuse `@property`. When you keep extending @property, it's time to refactor the class
- (**Item: 46**) Use Descriptors for Reusable `@property` methods
  - the problem with the @property is reuse; the methods `@property` decorates cant be reused for multiple attributes of the same or unrelated class
  - Reuse the behavior od @property methods by defining your own `descriptor protocol` classes with `__get__` and `__set__`
  - Use WeakKeyDictionary to ensure that your descriptor classes don't cause memory leaks
- (**Item: 47**) Use `__getattr__`. `__getattribute__` and `__setattr__` for Lazy Attributes
  - Use `__getattr__` and `__setattr__` to lazily load and save attributes
  - `__getattribute__` is more advance then `__getattr__` and will be called on every call even if attribute is set
  - there is considerable overhead added; use `super()` to avoid infinite recursion for an object
- (**Item: 48**) Validate Subclasses with `__init_subclass__`
  - Metaclasses can be used to inspect or modify a class after it's defined but before it's created, but they're often more heavyweight than what you need
  - Use `__init_subclass__` to ensure that subclasses are well formed at the time they are defined, before objects of their type are constructed and does not require `metaclasses` or `type` inheritane
- (**Item: 49**) Register Class Existence with `__init_subclass__`
  - class registration is a helpful pattern for building modular Python programs
  - `metaclasses` let you run registration code automatically each time a base class is subclassed in a program
- (**Item: 50**) Annotate Class Attributes with `__set_name__`
  - `metaclasses` enable you to modify a class's attributes before the class is fully defined
  - define `set_name__` on your descriptor classes to allow them to take into account their surrounding class and its property names
  - avoid memory leaks and the `weakref` module by having descriptors store data they manipulate directly withing a class's instance dictionary
- (**Item: 51**) Perfer Class Decorators Over Metaclasses for Composable Class Extensions

  - A class decorator is a simple function that receives a class instance as a parameter and returns either a new class or a modified version of the original class
  - Class decorators are useful when you want to modify every method or attribute of a class

0.7 Robustness and Performance

- (**Item: 65**) Take Advantage of Each Block in `try/except/else/finally`
  - use `try/finally` when you want exceptions to propagate up but also want to run cleanup code even when exceptions occur
  - use `try/else` to make it clear which excaptions will be handled by your code and which exceptions will propagate up
- (**Item: 66**) Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior
  - The `contextlib` built-in module provides a `contextmanager` decorator that makes it easy to use your own functions in `with` statements
  - The value `yielded` by context managers is supplied to the `as` part of the with statement. your code an directly access the cause of a special context
- (**Item: 67**) Use `datetime` Instead of `time` for Local Clocks
- (**Item: 68**) Make `pickle` Reliable with `copyreg`
- (**Item: 69**) Use `decimal` or `fraction` when Precision is Paramount
- (**Item: 70**) Profile Before Optimizing
  - use `cProfiler` over `Profiler`
  - `Stats` lets you select what data you want to see
- (**Item: 71**) Perfer `deque` for Producer-Consumer Queues
- (**Item: 72**) Consider Searching Sorted Sequences with `bisect`
- (**Item: 73**) Know How to Use `heapq` for Priority Queues
  - To use heapq, the items being prioritized must have a natural sort order, which requires special methods like **lt** to be defined for classes
- (**Item: 74**) Consider `memoryview` and `bytearray` for zero-copy interactions

0.8 Testing and Debugging

- (**Item: 75**) Using `repr` Strings for Debugging Output
  - `repr` can be used to type-check
  - you can reach into the object instance dictionary, which is stored in the **dict** attribute
- (**Item: 76**) Verify Related Behaviors in `TestCase` Subclasses
  - use `help(TeseCase)` to find methods like `assertEqual` or `assertTrue`
  - consider writing data-driven tests using the subTest helper method in order to reduce boilerplate
- (**Item: 77**) Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule` and `tearDownModule`
  - it's important to write both `unit tests` (for isolated functionality) and `integration tests` (for modules that interact with each other)
- (**Item: 78**) Use Mocks to Test Code with Complex Dependencies
  - use `ANY` to indicate any value is ok for an argument
  - use `call` to test how many times a function was called
- (**Item: 79**) Encapsulate Dependencies to Facilitate Mocking and Testing
- (**Item: 80**) Consider Interactive Debugging with `pdb`

    - The pdb module can be used for debug exceptions after they happen in independent Python programs (using `python -m pdb -c continue <program path>`) or the interactive Python interpreter (using import `pdb; pdb.pm()`)
- (**Item: 81**) Use `tracemalloc` to Understand Memory Usage and Leaks
- `gc` module can help you understand which object exist
- `tracemalloc` helps to understanding the source of memeory usage

0.9 Collaboration

- (**Item: 84**) Write Docstrings for Every `Function`, `Class` and `Module`
- (**Item: 85**) Use Packages to Organize Modules and Provide Stable `APIs`
- (**Item: 87**) Define a Root `Exception` to Insulate Callers from APIs
- (**Item: 88**) Know How to Break Circular Dependencies
- (**Item: 89**) Consider `Warnings` to Refactor and Migrate Usage
- (**Item: 90**) Consider Static Analysis via typing to Obviate Bugs