coding-tipps

January 1, 2021

1 Coding tipps

Much has been written on code quality and coding style. This document will probably not add much to the body of work already existing, but could nontheless be a helpful first step.

1.1 Checklist

I don't believe in checklist for code review, however, for guidance they might still yield some value. Note that these points are not meant to be rigourisly checked, but to be kept in mind while reviewing code.

1.1.1 Minimum

- Code meets specifications (ie, solves the exercise)
- Code uses lists and other structures where applicable (not variable1, variable2…)
- Purpose(!) of code is explained in text cells, if neccessary
- Notes on Implementation are conveyed in code comments, if neccessary

1.1.2 Great

- Code is easy to follow by using appropriate variable names and good structure
- Code uses list comprehension where appropriate
- Code follows a consistent style and layout (intendation, blank lines, line length)
- Code organized in functions where appropriate
- Code uses packages like numpy, pandas etc. where applicable

1.1.3 Advanced

- Code uses generators/iterators where appropriate
- Code organized in pure and un-pure functions where appropriate
- If applicable, complex codes are refactored into modules (not in this course)
- Complies largly to PEP8
- Uses doctests or other tests
- Code cannot reasonably made shorter without compromising readability

2 On Naming

> There are only two hard things in Computer Science: cache invalidation and naming things — Phil Karlton

Naming it *hard*, and most programmers struggle with comming up with usable, readable naming schemes that help readers to comprehend their code. Yet consistent, clever naming is the most important aspect of readable code, and possible the biggest productivity boost one can hope for. Lets try with an example.

Consider the following example:

[1]:
```
# Please take a moment to examine this code and guess its function and purpose..
 ↪.

# bad
def genf(ext):
    f1 = fln()
    f2 = []
    for f in f2:
        f3 = replext(n, ext)
        f2.append(n)
    return f2
```

...do you understand the purpose of this function? Could you fix or extend it? If so, stop now and apply to google and don't forget us on your glorious adventures, you rockstar. If not, know that the code is simple, but the names make it hard to read and understand. Now, lets try to fix this code by adding comments:

[2]:
```
# still bad, though commented
def genf(ext):
    f1 = fln() # get names
    f2 = [] #create list f2
    for f in f2: #iterate over all names
        f3 = replext(n, ext) # change extensions
        f2.append(n) # add to list
    return f2 # return result to calling code
```

Now, is this code easier to understand? No. But it is longer, and more akward to read. Comments rarely make bad code easier to understand, renaming and restructuring is the better way to improve readability.

[3]:
```
# great code
def generate_output_filenames(extension):
    input_filenames = get_filenames()
    output_filenames = []
    for filename in input_filenames:
        output_filename = replace_extension( filename, extension )
        output_filenames.append(output_filename)
    return output_filenames
```

Now, this code is way easier to understand, even to the inexperience programmer. Notice we don't need additional comments to explain ourselfs, the code document itself.

However, this code is hard to test. Why? Because it is "un-pure", that is, its return value is not (only) dependant on its arguments, but on other system state (specifically, the result from the get_filenames() call).

We can rewrite our function to be pure *and* include unit tests in the docstring, which also helps the reader to understand the functions purpose and usage.

[4]:
```
# even better code
def generate_output_filenames(input_filenames, extension):
    """Generate a list of filenames with extension.

    >>> generate_output_filenames(['a.pdf', 'b.pdf'], 'jpg')
    ['a.jpg', 'b.jpg']
    """
    output_filenames = []
    for filename in input_filenames:
        output_filename = replace_extension( filename, extension )
        output_filenames.append(output_filename)
    return output_filenames
```

2.1 Some notes on Readability

Code is written for humans, not computers, so treat your code a you would your manuscripts. Make it painfully clear to read, and trivial to understand. Good code is boring and read many time more than written, and the reader doesn't know what you know now. And that reader will probably be you, a few weeks, months or years from now. Be kind to future-you .

It clearly helps to make your code look like any other code, by using pythonic idioms and adhering to an established style guide.

Comments are like the manual to your smartphone. You won't read them. And if you have to read the comments, the code is propably not well written.

> Comments. A delicate matter, requiring taste and judgement. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses good type names and variable names, it should explain itself. Second, comments aren't checked by the compiler, so there is no guarantee that they're right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code.
>
> Rob Pike, "Notes on Programming in C"

When you do have to write comments (or jupyter text cells), explain the concepts, methods and reasoning behind the code, not the implementation.

Not a helpful comment:

[6]:
```
a = [] # create list
a.append(1) # add an item to the list
```

These comments add nothing of substance to the code, make it more difficult to read and will break sooner or later.

Helpful comments:

[7]:
```
def get_filenames_from_database():
    pass # not yet implemented, todo

temp = 50 # in milli-celsius

velocity = 3500  # literature value, see doi:10.2466/pms.1983.57.2.566.

temp = temp * 1000  # compensate for weird input data, convert to full °C

# work around issue that database sometimes returns None when uninitialized
names = get_filenames_from_database() or []
```

These comments explain decisions that lead to an implementation, *not* the implementation itself. Additional commentary should be reserved for text cells, especially considering equations.

2.2 Common mistakes w/ Jupyter Notebooks

Jupyter has it own sets of unique traps and pitfalls, so be aware of those while reviewing Notebooks.

- *Out of Order Execution and Hidden State*: it is trivial to get to an inconsistent state in jupyter notebooks, because cells are not neccessarily run from top to bottom. So ALWAYS ALWAYS run "Kernel -> Restart & Run All" before submitting a notebook.
- *Spaghetti code*: Resist the tendency to write non-modular spaghetti code. Write functions, classes and even importable modules where appropriate. Jupyter is all about data exploration, application and illustration of code, not about implementing complex codes or algorithms.

2.3 Code structure

Spaghetti code is code that is as long and intractable as spaghetti in a bowl that eventually develops into a big ball of mud. Unfortunattely, Jupyter makes it especially tempting to write code that just rambles on and on, with no decernable structure. Try to make your code not "just work", but also readable and elegant to the best of your abilities.

Lets look at an example, a jupyter script that converts a meter value into some sensible length unit, eg. 0.1m -> 10cm, 150000000000 -> 1AU etc.

[8]:
```
# this is bad

# distanceParser
# parse meters into next convenient unit
# supports mm, cm, m, km, au, parsecs, light years and others

distanceIn_meters = 101
isSolved=False
distance=distanceIn_meters
meter = distance
mm = .01
#print(isSolved)
```

```
if meter/mm < 10:
    isSolved = True
    result = str(meter/mm) + "mm"
#else:
#    isSolved= False
#    result = "?"

cm = .1
#print(isSolved)
if meter/cm < 10 and not isSolved:
    isSolved = True
    result = str(meter/cm) + "cm"
#else:
#    isSolved= False
#    result = "?"

m = 1
# print(isSolved)
if meter/m < 10 and not isSolved:
    isSolved = True
    result = str(meter/m) + "m"
#else:
#    isSolved= False
#    result = "?"
doubledeckerbus = 10
# print(isSolved)
# print(doubledeckerbus, meter/doubledeckerbus, isSolved)
if meter/doubledeckerbus < 1E3 and not isSolved:
    #print('doubledeckerbus')
    isSolved = True
    result = str(meter/doubledeckerbus) + "double-decker buses"
#else:
#    isSolved= False
#    result = "?"

footballfield = 110
if meter/footballfield < 1E3 and not isSolved:
    isSolved = True
    result = str(meter/footballfield) + "football fields"
#else
#    isSolved= False
#    result = "?"

km = 1E3
if meter/km < 1E3 and not isSolved:
    isSolved = True
    result = str(meter/km) + "km"
```

```
#else:
#    isSolved= False
#    result = "?"

au = 149E9
if meter/au < 1E3 and not isSolved:
    isSolved = True
    result = str(meter/au) + "au"
#else:
#    isSolved= False
#    result = "?"

parsec = 30E15
if meter/parsec < 1E3 and not isSolved:
    isSolved = True
    result = str(meter/parsec) + "pc"
#else:
#    isSolved= False
#    result = "?"

lightyear = 9E15
if meter/lightyear < 1E3 and not isSolved:
    isSolved = True
    result = str(meter/lightyear) + "ly"
#else:
#    isSolved= False
#    result = "?"

if not isSolved:
    result = "?"

print(result)
```

10.1double-decker buses

Do you *like* this code? Do you feel confident that this code is bug-free, and can be extended later on? Could you re-use this code, or parts of it?

If the answers to all these questions is "no", you should strongly consider refactoring your code.

Lets try this again:

[9]:
```
# this is better
distance = 101  # in meter

units = {
    'mm': .001,
    'cm': .01,
    'm': 1,
```

```
    'Double-Decker Buses': 10,
    'football fields': 110,
    'km': 1E3,
    'au': 149E9,
    'parsec': 30E15,
    'lightyear': 9E15
}

def find_unit(value, units):
    """Find first match of needle in min-max-haystack."""
    ratios = { unit: value/units[unit] for unit in units if value/units[unit]␣
 ↪>= 1}

    # if ratios contained any ratios over 1, take the last unit that fits
    # else just use the smallest input unit
    if ratios:
        target_unit = list(ratios.keys())[-1]
    else:
        target_unit = list(units.keys())[0]
    return '{} {}'.format(value/units[target_unit], target_unit)


print( find_unit(distance, units) )
```

```
10.1 Double-Decker Buses
```

While not perfect, this is much better. Adding a new unit is trivial, and re-use (eg. for time etc) is possible. You can even write tests easily for this code, and reason about its behaviour. The point is, your work is not done once your code runs and seems to work. Try to produce the best code you can.

2.4 Advanced: try to be *pythonic*

Python has a number of control structures that it *really* likes, and that differentiate itself from other languages. These might not be intuitive on first glance, but make a lot of sense to experienced python programmers.

Going back to the earlier example, we might want to refactor our code to use list comprehension instead of for loops. This reduces our code into an elegant one-liner:

[2]:
```
# elegant, pythonic implementation
def generate_output_filenames(filenames, extension):
    """Generate a list of filenames with extension.

    >>> replace_extensions(['a.pdf', 'b.pdf'], 'jpg')
    ['a.jpg', 'b.jpg']
    """
    return [replace_extension(name, extension) for name in filenames]
```

2.5 Further reading

Read How to Write Bad Code: The Definitive Guide, Part I and Part II