BEST PRACTICES OF CODE ORGANIZATION

Mikhail Dozmorov

Summer 2018

Automating everything

- Little automation is better than no automation
- It's more work to do things properly, but it could save you a ton of aggravation down the road

What to automate?

- what you're trying to do
- what you're thinking about
- what you're seeing
- what you're concluding and why

Good vs. bad programming

- “Any fool can write code that a computer can understand”
- “Good programmers write code that humans can understand”
  - Martin Fowler, 2008

Bad programmer explains him/herself with comments, good programmer explains him/herself with code

Good code = Clean code

Follow coding conventions

- **PEP-8** for Python, **PSR-2** for PHP, google "<language_name> coding conventions" for more
- **Google's R Style Guide**, **R style guide** by Hadley Wickham

Clean code is

- Understandable at first glance
- Neat and elegant
- Unambigious
- Not necesserily computationally efficient
- Self-explanatory
- Maintainable

http://www.cbs.dtu.dk/courses/27610/clean_code_index.html#clean-code-and-refactoring

https://www.python.org/dev/peps/pep-0008/

http://www.php-fig.org/psr/psr-2/

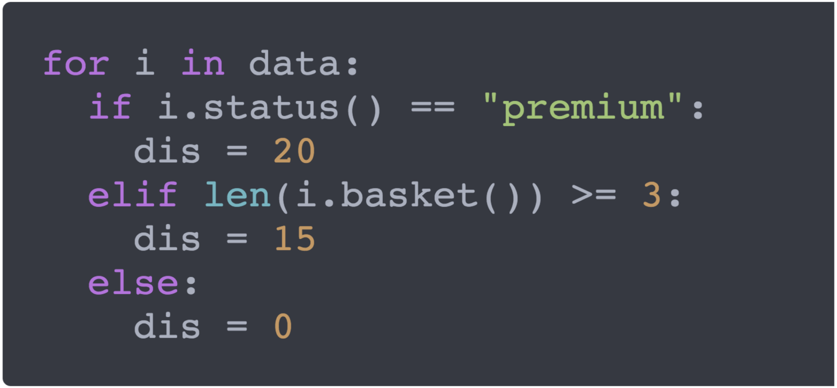
https://google.github.io/styleguide/Rguide.xml

Bad code

- Full of “magic” – variables/values noone can understand
- Cluttered, or too loose
- Redundant
- Poorly commented
- Does not follow conventions
- Hardly maintainable

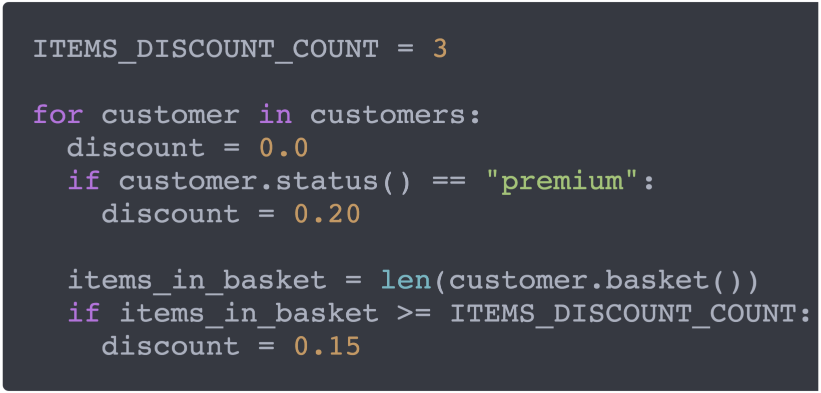
Code represents you – don’t write a bad code

Good vs. Bad code



```python
for i in data:
  if i.status() == "premium":
    dis = 20
  elif len(i.basket()) >= 3:
    dis = 15
  else:
    dis = 0
```

Good vs. Bad code



```python
ITEMS_DISCOUNT_COUNT = 3

for customer in customers:
  discount = 0.0
  if customer.status() == "premium":
    discount = 0.20

  items_in_basket = len(customer.basket())
  if items_in_basket >= ITEMS_DISCOUNT_COUNT:
    discount = 0.15
```

Good variable names

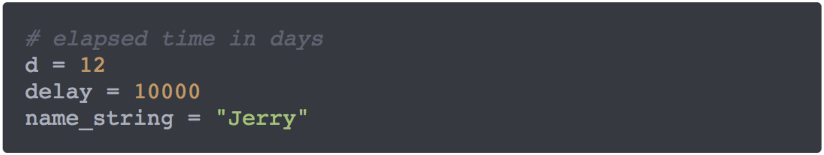
Variable names – nouns

- informative
- unambigious
- descriptive
- variables are in lower case, constants are in UPPER case

Good variable names

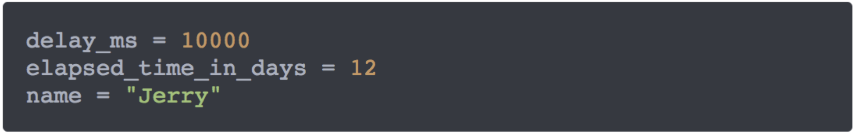
- **Tab completion** - Almost all modern text editors provide tab completion, so that typing the first part of a variable name and then pressing the tab key inserts the completed name of the variable. Employing this means that meaningful, longer variable names are no harder to type than terse abbreviations.
- choose and follow conventions
  - **underscore_convention**
  - **camelCaseConvention**
  - **dot.convention**

Good vs. bad variable names



```
# elapsed time in days
d = 12
delay = 10000
name_string = "Jerry"
```

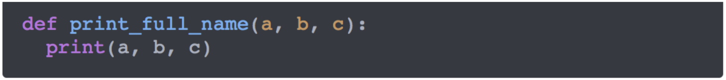
Good vs. bad variable names



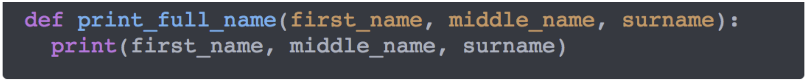
```
delay_ms = 10000
elapsed_time_in_days = 12
name = "Jerry"
```

Good function names

- Function names – verbs
  - "verb first" rule, e.g., print_full_name
  - informative, unambigious, descriptive, etc., as for variables



```python
def print_full_name(a, b, c):
  print(a, b, c)
```



```python
def print_full_name(first_name, middle_name, surname):
  print(first_name, middle_name, surname)
```

Good naming practices

- Short but meaningful
- Don't use spaces, either in variable names or file names, use underscore "_" instead, e.g., "tcga_first_batch"
- Be careful about extra spaces within cells, "female" is not the same as " female" or "female "
- Avoid special characters, except for underscores and hyphens. Other symbols ($, @, %, #, &, *, (,), !, /, etc.) often have special meaning in programming languages, and so they can be harder to handle

good name	good alternative	avoid
Max_temp_C	MaxTemp	Maximum Temp (▫ C)
Precipitation_mm	Precipitation	precmm
Mean_year_growth	MeanYearGrowth	Mean growth/year
sex	sex	M/F
weight	weight	w.
cell_type	CellType	Cell type
Observation_01	first_observation	1st Obs.

Refactoring

Refactoring – making better code

- Make code understandable by other developers. Here we ask ourselves a question; If I would give the code to my grandma, would she understand it?
- Increase readability of the code = reduce cluttering of the code. Make code loose in tight places and tight in loose places
- **Globally search-and-replace bad variable/function names**

Principles of good code development

- **DRY**
  - Don't Repeat Yourself
  - Do everything to avoid code repetition!
- **WET**
  - Write Everything (more than) Twice
  - The first time you write a code, you are writing it for the solution, second for comprehension, third for efficiency and last for your sake
- **KISS**
  - Keep It Small and Simple
  - Simplicity over complicity, shorter over longer

Computational reproducibility in plain language

- **Write code that uses relative paths.**
  - Don't use hard-coded absolute paths (i.e. `/Users/stephen/Data/seq-data.csv` or `C:\Stephen\Documents\Data\Project1\data.txt`).
  - Put the data in the project directory and reference it *relative* to where the code is, e.g., `data/gapminder.csv`, etc.
- **Always set your seed.** If you're doing anything that involves random/monte-carlo approaches, always use `set.seed()`.
- **Document everything and use code as documentation.**
  - Document why you do something, not mechanics.
  - Document your methods and workflows.
  - Document the origin of all data in your project directory.
  - Document **when** and **how** you downloaded the data.
  - Record **data** version info.
  - Record **software** version info with `session_info()`.
  - Use dynamic documentation to make your life easier.

Summary of good software development practices

1. Place a brief explanatory comment at the start of every program
2. Decompose programs into functions
3. Be ruthless about eliminating duplication
4. Always search for well-maintained software that do what you need
5. Test libraries before relying on them
6. Give functions and variables meaningful names
7. Make dependencies and requirements explicit
8. Do not comment and uncomment sections of code to control a program's behavior
9. Provide a simple example or test dataset
10. Submit code to a reputable DOI-issuing repository

The core realization in these practices is that being readable, reusable, and testable are all side effects of writing modular code, i.e., of building programs out of short, single-purpose functions with clearly-defined inputs and outputs