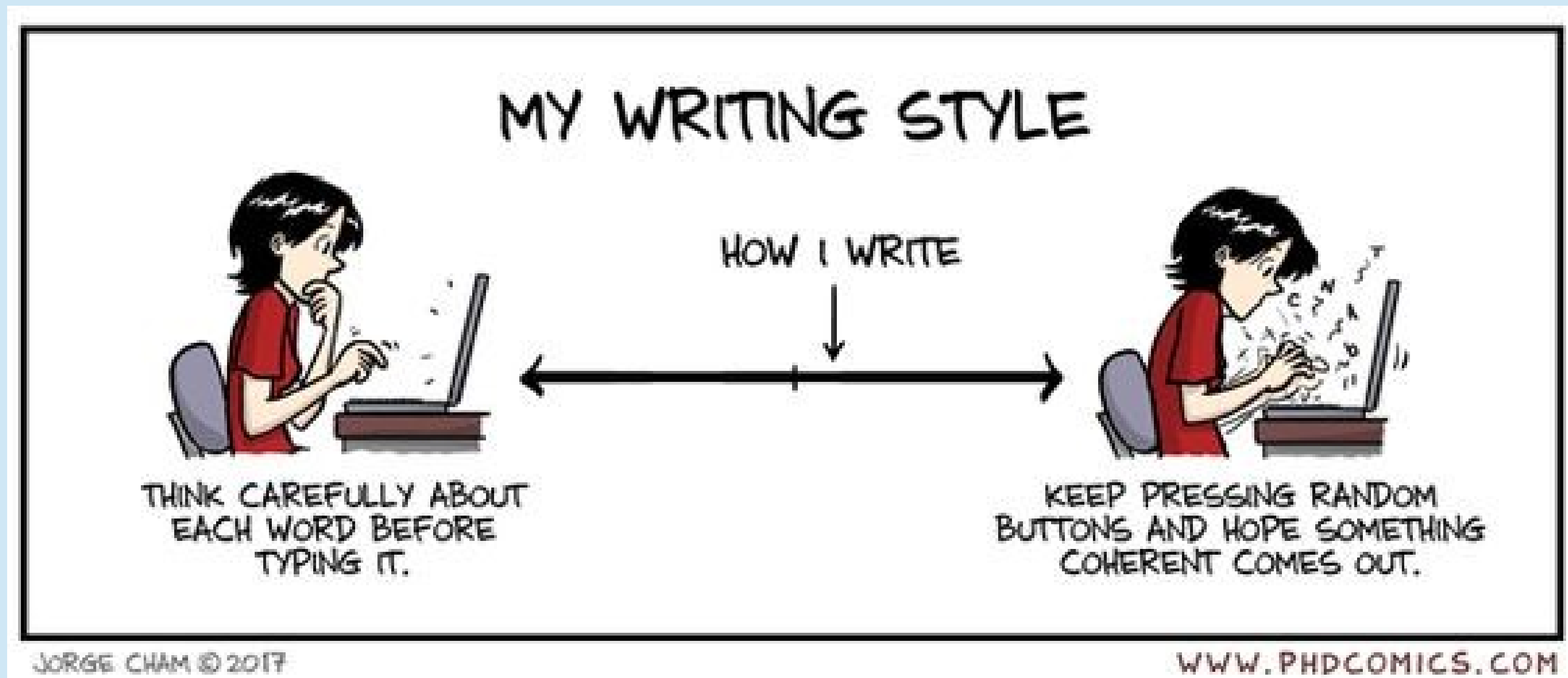
Writing for Computer Science

17 April 2018



Here are three descriptions of list comprehensions

Two are for Python, one for another language

Apart from the language, what differences do you notice?

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

Common syntax elements for comprehensions are:

```
comprehension ::=  expression comp_for
comp_for      ::=  [ASYNC] "for" target_list "in" or_test [comp_iter]
comp_iter     ::=  comp_for | comp_if
comp_if       ::=  "if" expression_nocond [comp_iter]
```

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't "leak" into the enclosing scope.

2.2 Comprehensions

Many functional languages provide a form of *list comprehension* analogous to set comprehension. For example,

$$[\,(x, y) \mid x \leftarrow [1, 2],\; y \leftarrow [3, 4]\,] \quad = \quad [(1, 3), (1, 4), (2, 3), (2, 4)].$$

In general, a comprehension has the form $[\, t \mid q \,]$, where t is a term and q is a qualifier. We use the letters t, u, v to range over terms, and p, q, r to range over qualifiers. A qualifier is either empty, Λ; or a generator, $x \leftarrow u$, where x is a variable and u is a list-valued term; or a composition of qualifiers, (p, q). Comprehensions are defined by the following rules:

$$\begin{array}{lrcl} (1) & [\, t \mid \Lambda \,] & = & \mathit{unit}\; t, \\ (2) & [\, t \mid x \leftarrow u \,] & = & \mathit{map}\,(\lambda x \rightarrow t)\; u, \\ (3) & [\, t \mid (p, q) \,] & = & \mathit{join}\,[\,[t|q] \mid p\,]. \end{array}$$

1) What differences did you notice?
(They are obviously very different, but can you put it into words?)

2) **Why** did the authors write them in these different ways?

3) **Who** is the reader for each piece?

4) **What** information does the reader want from the text?

5) **How** is the reader going to read the text?

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

Python language reference (www.python.org)

Common syntax elements for comprehensions are:

```
comprehension ::=  expression comp_for
comp_for      ::=  [ASYNC] "for" target_list "in" or_test [comp_iter]
comp_iter     ::=  comp_for | comp_if
comp_if       ::=  "if" expression_nocond [comp_iter]
```

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't "leak" into the enclosing scope.

Wadler, Philip. "*Comprehending monads*." Proceedings of the 1990 ACM conference on LISP and functional programming. ACM, 1990.

2.2 Comprehensions

Many functional languages provide a form of *list comprehension* analogous to set comprehension. For example,

$$[\,(x, y) \mid x \leftarrow [1, 2],\ y \leftarrow [3, 4]\,] \quad = \quad [(1, 3), (1, 4), (2, 3), (2, 4)].$$

In general, a comprehension has the form $[\,t \mid q\,]$, where t is a term and q is a qualifier. We use the letters t, u, v to range over terms, and p, q, r to range over qualifiers. A qualifier is either empty, Λ; or a generator, $x \leftarrow u$, where x is a variable and u is a list-valued term; or a composition of qualifiers, (p, q). Comprehensions are defined by the following rules:

$$\begin{array}{lll} (1) \quad [\,t \mid \Lambda\,] & = & \mathit{unit}\ t, \\ (2) \quad [\,t \mid x \leftarrow u\,] & = & \mathit{map}\,(\lambda x \rightarrow t)\,u, \\ (3) \quad [\,t \mid (p, q)\,] & = & \mathit{join}\,[\,[t|q] \mid p\,]. \end{array}$$

1) Python tutorial (*www.python.org*)

Teaching material

2) Python language reference (*www.python.org*)

Technical Documentation

3) Wadler, Philip. "*Comprehending monads*." Proceedings of the 1990 ACM conference on LISP and functional programming. ACM, 1990.

Theoretical / mathematical reasoning

Think of the reader’s needs,
not the writer’s wants

Reader’s Needs

- Quickly know what the text is about
- Decide if it is relevant for them
- Understand the content of the paper
- Learn the new ideas
- Find the information they need quickly
- Be convinced that the information is correct

Writer’s Wants

- Show off how much they know about a subject
- Show off how clever they are
- Surprise and entertain the reader
- Be liked by the reader
- Hide any problems with their work
- Write about something before doing the work to understand it

Scientific writing should be exact, clear and compact

Bad:

This class is fast and powerful, because the computer understands where the data is supposed to go. It also probably has a nice interface that makes the code extremely decoupled.

Good:

The class newList can be sorted in average case time O(n log n) and worst case time O(n log (log n)). Each node in the list carries both the value and the closest known values. It is accessed through the REST API.

Low-Level Rules

- Make definite assertions. Don’t hedge your bets.
- State your assumptions.
- Be precise, objective and unambiguous.
- Define any jargon you use.
- Use simple sentences.
- Be consistent. Use repetition.
- Every *if* should have a *then*.
- Don’t overload the comma.
- If you are using notation like $M_{i_{j,k_t}}^{O_b^c}$ then refactor!
- Use repetition

Exercise

Rewrite the following passage to make it easier to understand. (You may want to introduce variables and other mathematical notation.)

The cross-reference algorithm has two data structers: an array of documents, each of which is a linked list of words; and a binary tree of distinct words, each node of which contains a linked list of pointers to documents. When a document is added its linked list of words is traversed, and for each word in the list a pointer to the document is added to the word's linked list of documents. An order-one expansion of a document is achieved by pooling the linked lists of document pointers for each word in the document's linked list of words.

Types of Writing

Technical Writing in Computer Science

- **Mathematical / Scientific Reasoning**
- **Technical Documentation**
- **Selling / Advertising**
- **Teaching Material**
- **Criticism**

Minimise the number of new definitions and concepts

Problems in writing are often problems in thinking

Ethical Problems

- Plagiarism
- Ignore the problem and hope the reader doesn’t think of it
- “Authors” who are not actually authors
- Falsification of data

References

- Strunk and White. The Elements of Style
- Zimmer. Writing English as a Second Language
- Dupre. Bugs in Writing: A Guide to Debugging Your Prose
- Knuth, Larrabee, Roberts. Mathematical Writing

Method 1 – Grow the Tree

- Write the title
- Write the list of sections
- For each section: introductory paragraph, list of subsections
- For the ‘leaf’ subsections: List of paragraphs.
- Write the paragraphs.
- The tree should make sense at every depth
- Each leaf should see only its ‘scope’.

Common Mistakes – High Level

- Write everything you know about X
- General form before (instead of) specifics. (Examples First)
- Do everything at once

Common Mistakes – Low Level

- “This function allows to compute the square root” → “This function allows the user to compute the square root” or “This function allows the computation of the square root”
- “Exponentially” does not just mean “very fast”. It means $O\left(a^n\right)$ for some a.

The Perfect is the Enemy of the Good

No paper is perfect.
The best paper is a finished paper.
Research is never finished.