11. The Power of Variable Names

Code Complete

Contents

11.0 Python Keywords

https://docs.python.org/3/reference/lexical_analysis.html#keywords

2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

11.1 Considerations in Choosing Good Names

- You can't give a variable a name the way you give a dog a name—because it's cute or it has a good sound. Unlike the dog and its name, which are different entities, a variable and a variable's name are essentially the same thing.

- Java Example of Poor Variable Names

  Example 11-1.

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

  What's happening in this piece of code? What do x1, xx, and xxx mean? What does fido mean?

  Example 11-2. Java Example of Good Variable

```
Names balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) +
monthlyTotal;
balance = balance + Interest( customerID, balance );
```

  In view of the contrast between these two pieces of code, a good variable name is readable, memorable, and appropriate.

The Most Important Naming Consideration

- The most important consideration in naming a variable is that the name fully and accurately describe the entity the variable represents. An effective technique for coming up with a good name is to state in words what the variable represents. Often that statement itself is the best variable name. It's easy to read because it doesn't contain cryptic abbreviations, and it's unambiguous. Because it's a full description of the entity, it won't be confused with something else. And it's easy to remember

Problem Orientation

- A good mnemonic name generally speaks to the problem rather than the solution. A good name tends to express the what more than the how. In general, if a name refers to some aspect of computing rather than to the problem, it's a how rather than a what.

Avoid such a name in favor of a name that refers to the problem itself. A record of employee data could be called inputRec or employeeData. inputRec is a computer term that refers to computing ideas—input and record. employeeData refers to the problem domain rather than the computing universe.

Optimum Name

- Length: The optimum length for a name seems to be somewhere between the lengths of x and maximumNumberOfPointsInModernOlympics. Names that are too short don't convey enough meaning.
- Gorla, Benander, and Benander found that the effort required to debug a program was minimized when variables had names that averaged 10 to 16 characters (1990). Programs with names averaging 8 to 20 characters were almost as easy to debug.

Table 11-2. Variable Names That Are Too Long, Too Short, or Just Right

Too long:
numberOfPeopleOnTheUsOlympicTeam
numberOfSeatsInTheStadium
maximumNumberOfPointsInModernOlympics
Too short:
n, np, ntm
n, ns, nsisd
m, mp, max, points
Just right:
numTeamMembers, teamMemberCount
numSeatsInStadium, seatCount
teamPointsMax, pointsRecord

The Effect of Scope on Variable Names

- Cross-Reference Scope is discussed in more detail in Scope. Are short variable names always bad? No, not always. When you give a variable a short name like i, the length itself says something about the variable—namely, that the variable is a scratch value with a limited scope of operation.
- A study by W. J. Hansen found that longer names are better for rarely used variables or global variables and shorter names are better for local variables or loop variables (Shneiderman 1980).

- If you have variables that are in the global namespace (named constants, class names, and so on), consider whether you need to adopt a convention for partitioning the global namespace and avoiding naming conflicts.

Computed-Value Qualifiers in Variable Names

- Many programs have variables that contain computed values: totals, averages, maximums, and so on. If you modify a name with a qualifier like Total, Sum, Average, Max, Min, Record, String, or Pointer, put the modifier at the end of the name.
- This practice offers several advantages. First, the most significant part of the variable name, the part that gives the variable most of its meaning, is at the front, so it's most prominent and gets read first. Second, by establishing this convention, you avoid the confusion you might create if you were to use both totalRevenue and revenueTotal in the same program.
- An exception to the rule that computed values go at the end of the name is the customary position of the Num qualifier. Placed at the beginning of a variable name, Num refers to a total: numCustomers is the total number of customers. Placed at the end of the variable name, Num refers to an index: customerNum is the number of the current customer.
- But, because using Num so often creates confusion, it's probably best to sidestep the whole issue by using Count or Total to refer to a total number of customers and Index to refer to a specific customer.

Common Opposites in Variable Names

Cross-Reference

For a similar list of opposites in routine names, see "Use opposites precisely" in Good Routine Names.

- Use opposites precisely. Using naming conventions for opposites helps consistency, which helps readability. Pairs like begin/end are easy to understand and remember. Pairs that depart from common-language opposites tend to be hard to remember and are therefore confusing.

11.2 Naming Specific Types of Data

In addition to the general considerations in naming data, special considerations come up in the naming of specific kinds of data. This section describes considerations specifically for loop variables, status variables, temporary variables, boolean variables, enumerated types, and named constants.

Naming Loop Indexes

Cross-Reference
For details on loops, see Chapter 16.

- Guidelines for naming variables in loops have arisen because loops are such a common feature of computer programming. The names i, j, and k are customary:

  Example 11-4. Java Example of a Simple Loop Variable Name

  ```
  for ( i = firstItem; i < lastItem; i++ )
              {     data[ i ] = 0; }
  ```

- If a variable is to be used outside the loop, it should be given a name more meaningful than i, j, or k. For example, if you are reading records from a file and need to remember how many records you've read, a name like recordCount would be appropriate:

  Example 11-5. Java Example of a Good Descriptive Loop Variable Name

  ```
  recordCount = 0; while ( moreScores() )
         {     score[ recordCount ] = GetNextScore();
  recordCount++; }
  ```

- Because code is so often changed, expanded, and copied into other programs, many experienced programmers avoid names like i altogether.

  Example 11-6. Java Example of Good Loop Names in a Nested Loop

  ```
  for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ )
   {    for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ )
           {      score[ teamIndex ][ eventIndex ] = 0;    } }
  ```

- Carefully chosen names for loop-index variables avoid the common problem of index cross-talk: saying i when you mean j and j when you mean i. They also make array accesses clearer:

`score[ teamIndex ][ eventIndex ]` is more informative than `score[ i ][ j ]`.

- If you have to use i, j, and k, don't use them for anything other than loop indexes for simple loops—the convention is too well established, and breaking it to use them in other ways is confusing.

Naming Status Variables

Status variables describe the state of your program.
Here's a naming guideline: Think of a better name than flag for status variables. It's better to think of flags as status variables. A flag should never have flag in its name because that doesn't give you any clue about what the flag does. For clarity, flags should be assigned values and their values should be tested with enumerated types, named constants, or global variables that act as named constants.

Example 11-8.
C++ Examples of Better Use of Status Variables

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded == True ) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Naming Temporary Variables

Temporary variables are used to hold intermediate results of calculations, as temporary placeholders, and to hold housekeeping values.
They're usually called temp, x, or some other vague and nondescriptive name. In general, temporary variables are a sign that the programmer does not yet fully understand the problem.

Example 11-11. C++
Example with a "Temporary" Variable Name Replaced with a Real Variable

```
// Compute roots of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
discriminant = sqrt( b^2 - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
```

This is essentially the same code, but it's improved with the use of an accurate, descriptive variable name.

Naming Boolean Variables

- `Done`

Use done to indicate whether something is done. The variable can indicate whether a loop is done or some other operation is done. Set done to false before something is done, and set it to true when something is completed. error Use error to indicate that an error has occurred.

- `found`

  Use found to indicate whether a value has been found.

- `success` or `ok`

  Use success or ok to indicate whether an operation has been successful.

- If you can, replace success with a more specific name that describes precisely what it means to be successful.

- Give boolean variables names that imply `true` or `false`. Names like done and success are good boolean names because the state is either true or false; something is done or it isn't; it's a success or it isn't. Names like status and sourceFile, on the other hand, are poor boolean names because they're not obviously true or false.

- Some programmers like to put Is in front of their boolean names. Then the variable name becomes a question: isdone? isError? isFound? isProcessingComplete? Answering the question with true or false provides the value of the variable. A benefit of this approach is that it won't work with vague names: isStatus? makes no sense at all. A drawback is that it makes simple logical expressions less readable: if (isFound) is slightly less readable than if (found).

- Use positive boolean variable names. Negative names like notFound, notdone, and notSuccessful are difficult to read when they are negated—

Naming Enumerated Types

Cross-Reference
For details on using enumerated types, see Enumerated Types.

When you use an enumerated type, you can ensure that it's clear that members of the type all belong to the same group by using a group prefix, such as Color_, Planet_, or Month_.

Example 11-12.
Visual Basic Example of Using a Prefix Naming Convention for Enumerated Types

```
Public Enum Color
      Color_Red
      Color_Green
      Color_Blue
End Enum
```

Naming Constants

Cross-Reference
For details on using named constants, see Named Constants.

When naming constants, name the abstract entity the constant represents rather than the number the constant refers to. FIVE is a bad name for a constant (regardless of whether the value it represents is 5.0). CYCLES_NEEDED is a good name. CYCLES_NEEDED can equal 5.0 or 6.0. FIVE = 6.0 would be ridiculous.

11.3 The Power of Naming Conventions

Some programmers resist standards and conventions—and with good reason. Some standards and conventions are rigid and ineffective—destructive to creativity and program quality. This is unfortunate since effective standards are some of the most powerful tools at your disposal. This section discusses why, when, and how you should create your own standards for naming variables.

Why Have Conventions?

- Conventions offer several specific benefits: They let you take more for granted. By making one global decision rather than many local ones, you can concentrate on the more important characteristics of the code. They help you transfer knowledge across projects. Similarities in names give you an easier and more confident understanding of what unfamiliar variables are supposed to do. They help you learn code more quickly on a new project.

- They reduce name proliferation. Without naming conventions, you can easily call the same thing by two different names. For example, you might call total points both pointTotal and totalPoints.

- They compensate for language weaknesses. You can use conventions to emulate named constants and enumerated types. The conventions can differentiate among local, class, and global data and can incorporate type.

- They emphasize relationships among related items. If you use object data, the compiler takes care of this automatically. If your language doesn't support objects, you can supplement it with a naming convention. Names like address, phone, and name don't indicate that the variables are related. But suppose you decide that all employee-data variables should begin with an Employee prefix. employeeAddress, employeePhone, and employeeName leave no doubt that the variables are related. Programming conventions can make up for the weakness of the language you're using.

- The power of naming conventions doesn't come from the specific convention chosen but from the fact that a convention exists, adding structure to the code and giving you fewer things to worry about.

When You Should Have a Naming Convention

- When multiple programmers are working on a project
- When you plan to turn a program over to another programmer for modifications and maintenance (which is nearly always)
- When your programs are reviewed by other programmers in your organization
- When your program is so large that you can't hold the whole thing in your brain
- When the program will be long-lived enough that you might put it aside for a few weeks or months before working on it again
- When you have a lot of unusual terms that are common on a project and want to have standard terms or abbreviations to use in coding

11.4 Informal Naming Conventions

Most projects use relatively informal naming conventions such as the ones laid out in this section.

Guidelines for a Language-Independent Convention

Here are some guidelines for creating a language-independent convention:

- Differentiate between variable names and routine names. The convention this book uses is to begin variable and object names with lower case and routine names with upper case: variableName vs. RoutineName(). Differentiate between classes and objects.

Example 11-17.
Each of these options has strengths and weaknesses.

Option 1: Differentiating Types and Variables via Initial Capitalization

```
Widget widget;
LongerWidget longerWidget;
```

Option 2: Differentiating Types and Variables via All Caps

```
WIDGET widget;
LONGERWIDGET longerWidget
```

Option 3: Differentiating Types and Variables via the “t_” Prefix for Types

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

Option 4: Differentiating Types and Variables via the “a” Prefix for Variables

```
Widget aWidget;
LongerWidget aLongerWidget;
```

6 Option 5: Differentiating Types and Variables via Using More Specific Names for the Variables

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

- Identify global variables. One common programming problem is misuse of global variables.

- Identify member variables. Identify a class's member data. Make it clear that the variable isn't a local variable and that it isn't a global variable either.

- Identify type definitions. Naming conventions for types serve two purposes: they explicitly identify a name as a type name, and they avoid naming clashes with variables. To avoid confusion, you can prefix the type names with t_, such as t_Color and t_Menu.

- Identify named constants. Named constants need to be identified so that you can tell whether you're assigning a variable a value from another variable (whose value might change) or from a named constant.
  In C++ and Java, the convention is to use all uppercase letters, possibly with underscores to separate words, RECSMAX or RECS_ MAX

- Identify elements of enumerated types. Elements of enumerated types need to be identified for the same reasons that named constants do—

- Identify input-only parameters in languages that don't enforce them. Sometimes input parameters are accidentally modified.

- Format names to enhance readability. Two common techniques for increasing readability are using capitalization and spacing characters to separate words.
  other languages also allow the use of the underscore (_) separator.

- Try not to mix these techniques; that makes code hard to read.

Guidelines for Language-Specific Conventions

Follow the naming conventions of the language you're using.

- PEP Index > PEP 8 -- Style Guide for Python Code

  http://legacy.python.org/dev/peps/pep-0008/

- Code Style - Python Guide

  http://docs.python-guide.org/en/latest/writing/style/

- Google Python Style Guide

  https://google-styleguide.googlecode.com/svn/trunk/pyguide.html

- Java Conventions Further Reading For more on Java programming style, see The Elements of Java Style, 2d ed. (Vermeulen et al. 2000).

In contrast with C and C++, Java style conventions have been well established since the language's beginning: i and j are integer indexes. Constants are in ALL_CAPS separated by underscores. Class and interface names capitalize the first letter of each word, including the first word—for example, ClassOrInterfaceName. Variable and method names use lowercase for the first word, with the first letter of each following word capitalized—for example, variableOrRoutineName. The underscore is not used as a separator within names except for names in all caps. The get and set prefixes are used for accessor methods.

Sample Naming Conventions

The standard conventions above tend to ignore several important aspects of naming that were discussed over the past few pages—including variable scoping (private, class, or global), differentiating between class, object, routine, and variable names, and other issues.

Table 11-3. Sample Naming Conventions for C++ and Java

11.5 Standardized Prefixes

Further Reading
For further details on the Hungarian naming convention, see "The Hungarian Revolution" (Simonyi and Heller 1991).

Standardizing prefixes for common meanings provides a terse but consistent and readable approach to naming data. The best known scheme for standardizing prefixes is the Hungarian naming convention, which is a set of detailed guidelines for naming variables and routines (not Hungarians!) that was widely used at one time in Microsoft Windows programming.

is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value. Standardized prefixes are composed of two parts: the user-defined type (UDT) abbreviation and the semantic prefix.

User-Defined Type Abbreviations

The UDT abbreviation identifies the data type of the object or variable being named. UDT abbreviations might refer to entities such as windows, screen regions, and fonts. A UDT abbreviation generally doesn't refer to any of the predefined data types offered by the programming language.

Semantic Prefixes

Semantic prefixes go a step beyond the UDT and describe how the variable or object is used. Unlike UDTs, which vary from project to project, semantic prefixes are some-what standard across projects.

Table 11-7 shows a list of standard semantic prefixes.

Semantic prefixes are formatted in lowercase or mixed uppercase and lowercase and are combined with the UDTs and with other semantic prefixes as needed.

11.6 Creating Short Names That Are Readable

The desire to use short variable names is in some ways a remnant of an earlier age of computing. Older languages like assembler, generic Basic, and Fortran limited variable names to 2–8 characters

In modern languages like C++, Java, and Visual Basic, you can create names of virtually any length; you have almost no reason to shorten meaningful names.

It's a good idea to be familiar with multiple techniques for abbreviating because no single technique works well in all cases.

General Abbreviation Guidelines

Here are several guidelines for creating abbreviations. Some of them contradict others, so don't try to use them all at the same time.

- Use standard abbreviations (the ones in common use, which are listed in a dictionary).
- Remove all nonleading vowels. (computer becomes cmptr, and screen becomes scrn. apple becomes appl, and integer becomes intgr.)
- Remove articles: and, or, the, and so on.
- Use the first letter or first few letters of each word.
- Truncate consistently after the first, second, or third (whichever is appropriate) letter of each word.

- Keep the first and last letters of each word.
- Use every significant word in the name, up to a maximum of three words.
- Remove useless suffixes—ing, ed, and so on.
- Keep the most noticeable sound in each syllable.
- Be sure not to change the meaning of the variable.

Iterate through these techniques until you abbreviate each variable name to between 8 to 20 characters or the number of characters to which your language limits variable names.

Phonetic Abbreviations

Some people advocate creating abbreviations based on the sound of the words rather than their spelling. Thus skating becomes sk8ing, highlight becomes hilite, before becomes b4, execute becomes xqt, and so on.

Comments on Abbreviations

You can fall into several traps when creating abbreviations. Here are some rules for avoiding pitfalls:

- Don't abbreviate by removing one character from a word.
- Abbreviate consistently. Always use the same abbreviation. For example, use Num everywhere or No everywhere, but don't use both.
- Create names that you can pronounce. Use xPos rather than xPstn and needsComp rather than ndsCmptg.
- Create names that you can pronounce. Use xPos rather than xPstn and needsComp rather than ndsCmptg. Apply the telephone test—if you can't read your code to someone over the phone, rename your variables to be more distinctive (Kernighan and Plauger 1978).
- Avoid combinations that result in misreading or mispronunciation. To refer to the end of B, favor ENDB over BEND.
- Use a thesaurus to resolve naming collisions.
- Document extremely short names with translation tables in the code. In languages that allow only very short names, include a translation table to provide a reminder of the mnemonic content of the variables. Include the table as comments at the beginning of a block of code.
- Document all abbreviations in a project-level "Standard Abbreviations" document.

The general issue illustrated by this guideline is the difference between write-time convenience and read-time convenience. This approach clearly creates a write-time inconvenience, but programmers over the lifetime of a system spend far more time reading code than writing code.

Remember that names matter more to the reader of the code than to the writer. Read code of your own that you haven't seen for at least six months and notice where you have to work to understand what the names mean.

Kinds of Names to Avoid

- Avoid misleading names or abbreviations.
  FALSE is usually the opposite of TRUE and would be a bad abbreviation for "Fig and Almond Season."

- Avoid names with similar meanings.
  input and inputValue, recordNum and numRecords, and fileNumber and fileIndex are o semantically similar

- Avoid names like clientRecs and clientReps. They're only one letter different from each other, and the letter is hard to notice.

- Avoid names that sound similar, such as wrap and rap.

- Avoid numerals in names.
  If the numerals in a name are really significant, use an array instead of separate variables. If an array is inappropriate, numerals are even more inappropriate. For example, avoid file1 and file2, or total1 and total2.

- misspelling highlight as hilite to save three characters makes it devilishly difficult for a reader to remember how highlight was misspelled.

- Avoid words that are commonly misspelled in English.

- Don't differentiate variable names solely by capitalization. If you're programming in a case-sensitive language such as C++,

- Avoid multiple natural languages. In multinational projects, enforce use of a single natural language for all code, including class names, variable names, and so on.

- more subtle problem occurs in variations of English. If a project is conducted in multiple English-speaking countries, standardize on one version of English

- Avoid the names of standard types, variables, and routines.

- Don't use names that are totally unrelated to what the variables represent. Sprinkling names such as margaret and pookie throughout your program virtually guarantees that no one else will be able to understand it. Avoid your boyfriend's name, wife's name, favorite beer's name,

- Avoid names containing hard-to-read characters. Be aware that some characters look so similar that it's hard to tell them apart.