ICS3U: Internal Documentation

Inline Comments

It is possible to make **comments** (information that is not to be read by the interpreter, but by humans) by prefacing a line with the `#` character. All characters after the `#` will be ignored. While there are some that insist that comments are unnecessary ("the code should speak for itself" is a common reply), it is generally good practice to use them in our programs to explain steps or clarify procedures.

It is not necessary to comment every line of code. In fact, this is probably more distracting and time-consuming than useful. Consider the following program that reads two integers from the user, finds their product, and displays it to the screen.

```
# Read the first integer from the user
num1 = int(input("Enter the first integer: "))
# Read the second integer from the user
num2 = int(input("Enter the second integer: "))
# Find the product of the integers
product = num1 * num2
# Display the product to the screen
print("The product of the two integers is", product)
```

Much of the information in the comments is "obvious" to someone who has some familiarity with Python. It is also redundant: the first two comments are virtually identical, as are the lines of code to which they refer. A better solution would be to comment blocks of related code and use vertical whitespace to separate the blocks, as in the following example.

```
# Read two integers from the user
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))

# Find the product of the integers and display it to the screen
product = num1 * num2
print("The product of the two integers is", product)
```

This is less cluttered, but still conveys the same information. Since they are only for the benefit of the reader, comments should be informative. If possible, they should describe the purpose of the commands or how they contribute to the functioning of the program, rather than literal interpretations of the commands. Consider the following piece of code.

```
# Add 1 to item_count, then print "Item added"
item_count += 1
print("Item added.")
```

This does not give the reader any additional information that cannot already be obtained from the relatively simple Python commands, so the comment is fairly useless. A better comment might be this:

```
# Update the number of items in inventory and inform the user.
item_count += 1
print("Item added.")
```

This clearly conveys the *intent* of the code, and is far more useful than the previous comment.

Program and Function Headers

Most of our major programs so far have been prefaced by **program headers**. These comments usually include various pieces of information about the program and the programmer. This makes it easy for a reader to obtain a general overview of the program, and to see who was responsible for the code.

Many larger programs include a **log** of dates when various sections of code were modified, along with a description of what was changed, added or removed. This is important, because any changes made to one section of code might have effects on other sections. If there is found to be a problem, the code can be reverted back to an earlier state by reversing the changes. These days, much of this is handled automatically by **version control software**.

A typical program header might look something like the following.

```
# ======================================================================
# MY_AWESOME_PROGRAM.PY
# John Q. Programmer
# This program does the totally awesome things described in Specification
# Document A-15243. It uses the SuperDuper Algorithm for efficiency.
# REVISION HISTORY
# 2017-03-14: Initial creation.
# 2017-03-17: Fixed a bug where negative values gave incorrect results.
# 2017-04-04: Added support for Elbonian language.
# 2017-06-11: Disabled code which would allow program to become sentient.
# ======================================================================
```

As programs become larger and more complex, much of their functionality is moved into smaller functions that are responsible for handling certain actions. It is good practice to add a **function header**, with information about the function (e.g. its purpose, revision dates, etc.), similar to the program header. Again, this helps a programmer obtain an overview about what the function does, who wrote or modified it, and so on.

Pre- and Post-Conditions

While not everyone agrees, including **pre-conditions** and **post-conditions** to a function's header can provide useful information, if accurate. A pre-condition is a requirement that *must* be true in order for a function to run correctly. A post-condition is a result that *will* be true once the function has completed. Using a mathematical analogy, consider the process of determining the square root of a value. Since it is not possible to determine the square root of a negative value (at least not in the real number system), a pre-condition might be "the number for which we are determining the square root must be zero or greater". Once we have determined the square root, we will have a new value that is zero or greater. A post-condition, then, might be "a number with a value of zero or greater is generated".

A header for a function that calculates the square root of a number might look something like this:

```
# ======================================================================
# square_root(n)
# John Q. Programmer
# Calculates the square root, r, of a number, n.
# ----------------------------------------------------------------------
# Pre-conditions: n must be a number greater than or equal to zero.
# Post-conditions: r will be a number greater than or equal to zero.
# ======================================================================
```

More complicated functions may have multiple pre- or post-conditions.

```
# ========================================================================
# shortest_path(G, start, end)
# John Q. Programmer
# Calculates the shortest path in a graph, G, beginning at start and
# finishing at end. Uses a modified version of Djikstra's algorithm
# to accommodate paths with negative values.
# ------------------------------------------------------------------------
# Pre-conditions: G must be a non-empty graph, start and end must be
# defined points on the graph, and it must be possible to reach end from
# start using some combination of paths.
# Post-conditions: Will generate a non-empty sequence of paths from
# start to end with the smallest value. In the case of a tie, the path
# connecting the fewest nodes is given. In the case of another tie,
# the first such path is given.
# ========================================================================
```

Note that pre- and pos-conditions are only useful if they are accurate. It is the job of the programmer to ensure that they are, and that any changes to code in the functions do not alter them. If so, the conditions must be updated to reflect the new code.