# The Debug Perspective in Eclipse for Python
# A Reference and Tutorial

## Overview

Debugging is the process of locating the source of programming errors (often called bugs) and correcting them. Debuggers are software tools that programmers use to help with this process. Primarily, we use debuggers to **control and monitor** the execution of our scripts, to better understand them. Specifically, we can use a debugger to **run a script selectively**, **stopping it at interesting locations** and **examining its state** (the values that are bound to its variables, which are stored and modified as the script runs). In this way we can better understand how a script works —or more importantly, help pinpoint exactly where a script fails to work correctly as the first step towards determining what changes we must make to fix it. Thus, the process of debugging involves both locating and correcting the source of programming errors.

We can instruct a debugger to continually display the current values stored in all (or selected) variables; this process is called *observing* variables. Then, we can execute our script one line at a time (this process is called *single stepping* through a script). After each step, we can observe the new values bound to the variables. We can also use the debugger to automatically run the script with *unconditional breakpoints* set on lines of code, which instructs the debugger to stop the script whenever it is about to execute any breakpointed line; in addition, we can set a *conditional breakpoint* to also stop a script whenever it is about to execute a breakpointed line, but only when some specified condition (a `bool` expression) evaluates to `True` when that line is about to be executed.

This document explains and demonstrates how to use the Eclipse Debug perspective. While reading this document, look for the symbol, which instructs you to practice the debugger commands that were just discussed. Debuggers are sophisticated tools with many subtle features. Although this document is only a brief introduction, designed for novices, it covers the rudiments of the most important and useful debugger commands. Finally, during the quarter we will use the illustrative power of the Debug perspective to help us learn and understand new Python language features -an added bonus for knowing how to use it.

## Learn in haste; debug in leisure

A debugger is an important tool that can save us lots of time. I estimate that for every hour that you spend learning about the Debug perspective or practicing using it early in the quarter, you will save 3 hours later in the quarter, when you are locating and correcting execution errors in your scripts; few students will learn about the debugger in the middle of trying to debug their code, so spend the time now exploring. But debuggers are not a substitute for thinking: they help automate hand simulations, but they do not automate the inductive and deductive processes necessary to locate and fix bugs in a script. So, do not be misled, the most important tool for debugging is still your brain.

## Switching to the Eclipse Debug Perspective


Before we can debug a script, we must open its project and ensure it is runnable (no detected errors or warnings on its **Edit** tab). Then we are ready to run and debug it.

To follow along with this handout, first copy onto the desktop the `collatz` project (do it now), which linked on the **`Sample Programs`** page; in the **PyDev** perspective, load it into Eclipse as an existing project. Actually, this script is correct: it contains no bugs. But, we will use this script to study the features of the **Debug** perspective -the same features that we will use to actually debug our programs. Before proceeding, please read the comments at the top that describe this script; then, run it a few times, experimenting with it (supplying different values to its prompt) and observing its output. We must "understand" code before we can productively use the debugger to monitor it.

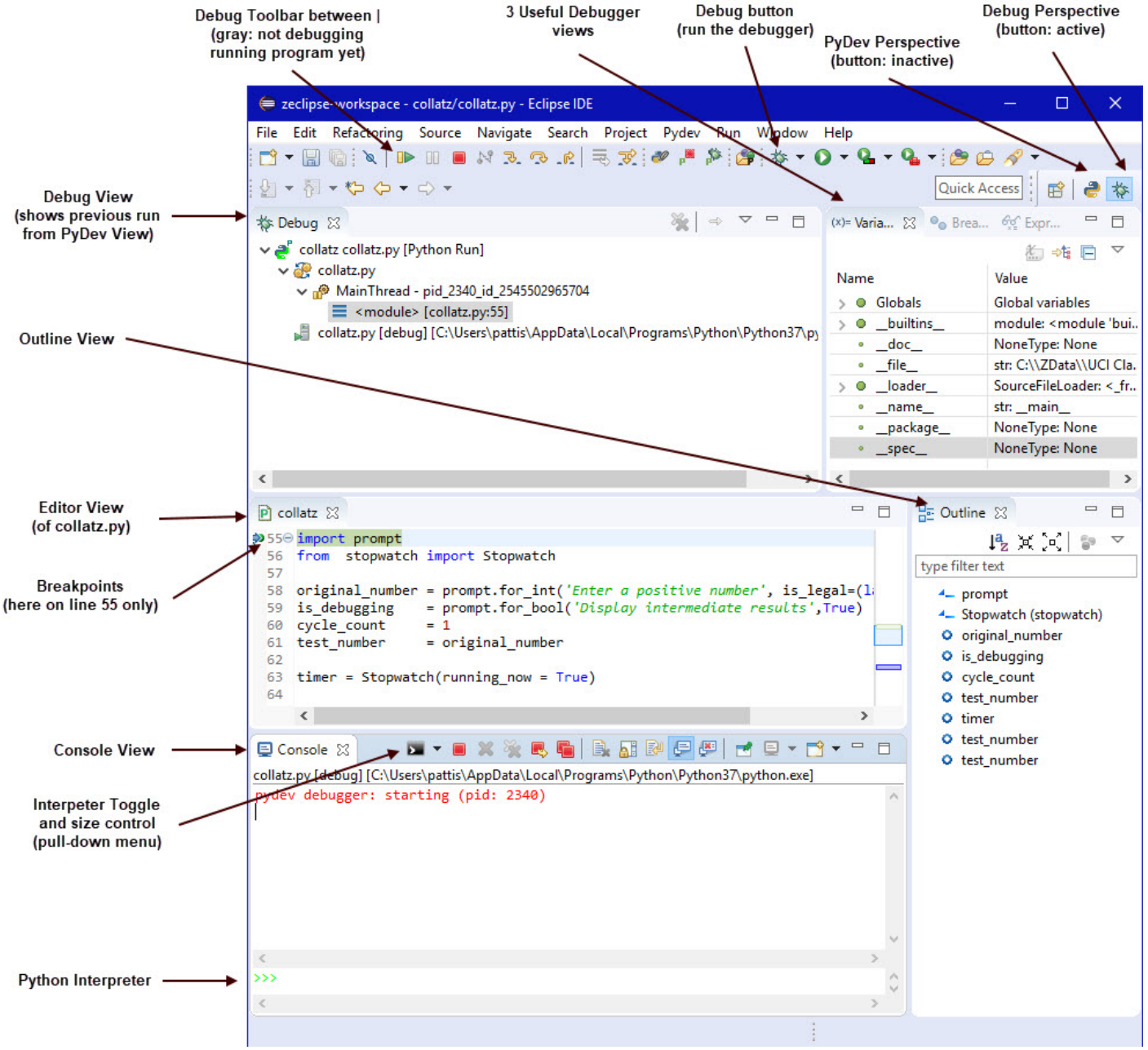
We are now ready to switch to the **Debug** perspective. The button for the **Debug** perspective should appear near the top-right of the workbench, to the right of the **PyDev** perspective button. If the **Debug** button is there, click it; if it is not there, you can put it there by clicking the **Open Perspective** button on the tab to the left of the **PyDev** button and selecting the **Debug** perspective (or by selecting **Window | Open Perspective | Debug** on the pull-down menu). The **PyDev** and **Debug** buttons can be annotated with the words **PyDev** and **Debug** by right-clicking either button and clicking the **Show Text** option (to remove the text, do this operation again, toggling the checkbox).

In the `collatz` project, click the **Debug** perspective button. You should see the Eclipse workbench change to the **Debug** perspective as illustrated in the figure on the next page. This picture illustrates (and labels) the standard size and layout of these windows in the **Debug** perspective, which was designed to make the most important debugging windows easily accessible. We can change the size and layout of any windows in this view, including removing any of their tabs or even the windows themselves. Sometimes it is useful to detach the tabs from the workbench (if you have enough screen space to display them).

**IMPORTANT**: Check the following only the **first time** that you use the **Debug** perspective: (a) Click the **Run** tab/pulldown menu, (b) and click the **Manage Python Exception Breakpoints** (about 2/3 the way down this list of optinos), (c) At the bottom, ensure the option **Suspend on django template render exceptions** is not checked (uncheck it if it is), and (d) Click **OK**.

Double-click in the margin to the left of the line 55 (`import prompt`) which appears in the editor window in the `collatz` tab. A green breakpoint icon ( ) should appear there, as is shown below. We will discuss the meaning of breakpoints in much more detail latter. Eclipse can now run the script under control of the **Debug** perspective. If we don't set this breakpoint, the Eclipse **Debug** perspective would run the script as it does in the **PyDev** perspective. But once we set this breakpoint, the Eclipse **Debug** perspective will stop the program before executing line 55, and allow us to perform the debugging commands discussed in this document.

Now click the **Debug** button ( ) on the Eclipse toolbar to the left of the run button ( ), not the **Debug** perspective button! Eclipse starts running the script under control of the **Debug** perspective and stops before executing line 55 to wait for more debugging commands. You should see the following workbench.




The **Debug** Toolbar now appears in color (not gray). The tiny icon after the **Console** tab toggles between showing (a) both the running program and Python Interpreter (into which we can type Python expressions to the >>> prompt, and have their value displayed) in the Console and (b) just showing the running program. Use the pull-down menu to its right to **Set Console Height** (how much shows the Python Interpreter): I suggest 20(%).

Finally, the breakpoint icon on line 55 is highlighted in green and has a blue arrow on it, indicating that the debugger is about to execute that line/statement in the script. Next we will look at the tabs in the **Debug** view.

### The Debug Tab

The **Debug** tab contains a *stack* under `MainThread - pid15052...`, listing all the modules (or functions) currently executing in the script. Currently it shows that the debugger is executing only the script in a module (`<module> [collatz.py:55]`). Generally, each module (or function) in the stack lists the line number (e.g., `line 55`) that it is executing. If the script calls a function, that function will be placed at the top of the stack. The top function is special: its line number always shows which line Python is about to execute (in a module or function). We can easily look at the context of any module/functions that are active, from the original script to the currently executing function. In the picture above, `collatz.py` is the active module and it is about to execute `line 55`.

The **Debug** tab allows us to monitor module execution (and function calls). We can click any line in this stack: the name is then highlighted, a tab in the **Editor** view displays that module's (or function's) code, and the **Variables** tab (described below) displays the values of that module's (or function's) variables. Whenever we start to debug a script with a breakpoint on the first line, we will see one highlighted entry in the **Debug** tab, indicating the script we are running and the line number it is about to execute: Here the `collatz.py` script which is about to execute line 55, shown in the `collatz` module/tab in the **Editor** view. The feature allows us to zoom-in on any executing code that we want to examine closely.

### The Editor Tab

The **Editor** window shows the line of Python code that is about to be executed, both by showing a blue right-pointing arrow in its left margin and by highlighting that line in green (in one of the file tabs in the editor window). As we execute the statements in a script (see single stepping/resume below), the arrow will move from line to line along with the highlighting. If we hover over any defined variable in an **Editor** tab, its name and current value will appear in a yellow window below the hover.

Whenever we set a breakpoint on a line, the **Debug** perspective stops on that line before its code executes: in this case the `import prompt`. Note that when the **Debug** perspective stops on a line, it has **not yet executed** that line: it is about to execute it. This detail often confuses beginners.

By selecting any module/function in the **Debug** tab, we can easily see where in that module our script is currently executing (either in the same module/file tab, or in another one the **Debug** perspective creates). As we see below, we can also see that module's variables in the **Variables** tab.

### The Variables Tab

The **Variables** tab lists the names and values of all the module/function variables that are defined by the highlighted module/function in the **Debug** tab. By selecting any module/function named in the **Debug** tab, we can see its code (in an **Editor** tab) and all of its defined variables (in the **Variable** tab). As the debugger executes the script, the the **Editor**/**Variables** tabs are updated to match the execution.

All variables refer to objects. The value of a simple object appears in the Value column in the **Variables** tab by showing the type of the object and its value: for example, the variable `__name__` (you will have to scroll to find it) is a `str` (string) with the value `__main__` (note a colon separates the type and value; the printed `str` is not enclosed in parentheses). If a variable refers to a complicated object (one that defines multiple values, like a `list`), the variable is prefaced by a disclosure triangle. If we click a `>`, it changes to a `V` and discloses more of the values in that object; if we click `V`, it changes to a `>` and, elides these names, so that they are not displayed. Later in the quarter, when we study more about Python, we will learn more about using these name-space boxes. If we click a name in the **Variables** tab its value will also appear in the small window below the **Variables** tab. Click `__name__`.

If there are too many entries in the **Debug** or **Variables** tab to display all at once, we can scroll through them. We can also simultaneously increase/decrease the sizes of the **Debug** and **Variables** tabs by pulling downward/upward on the horizontal line that separates these panes from the **Editor** tab. Doing so increases/decreases the size of these tabs (and an **Editor** tab). We will find it particularly useful to drag/drop the **Variables** tab outside of the Eclipse workbench to more easily see its contents; we can always drag/drop it back.

When a script starts, the **Variables** tab will always display the names `Globals`, `__builtins__`, `__doc__`, `__file__`, and others (all appearing alphabetically). Soon we will see how to step over the statements that define the new names `prompt`, `Stopwatch`, `original_number`, `is_debugging`, `cycle_count`, `test_number`, and `timer`, which will appear in the **Variable** tab along with their values.

### Outline Tab

The **Outline** tab shows every name that can be bound to a value in the module in the chosen **Editor** tab. If a name is bound multiple times in the module, it appears multiple times in the **Outline** tab. The names that are bound by `import` statements a prefaced by a different blue icon than the names that are bound to values by an assignment (=) statement. If we click on one name, the **Editor** tab will show the corresponding line in the module on which that name is bound to a value. Note "rebinding" is shown, not mutation: there is a difference.

The order of the names in the **Outline** tab correspond to the order in which names are bound in the module. But, we can display these names alphabetically by clicking the sort-by-name icon (  ). If we click this icon again, it toggles: the names, returning them to their original ordering.

We can use the downward triangle icon (four to the right of the sort-by-name icon) to hide certain categories of names (like imports) so they don't appear in the **Outline** tab.

### Console Tab

The **Console** tab starts showing the text `pydev debugger: starting ...`; as we debug the script, the **Console** tab will eventually show all the information it shows there when we run the script using the **PyDev** perspective.

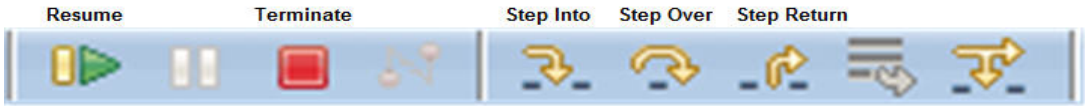
### Maximizing Tabs

Remember, to maximize any of the tabs, just double-click it (and double click it again to restore it). For more extensive manipulation, use the buttons on the right of the toolbar holding the tab. Try this now, with each of these three views (**Debug** tab, editor, and **Variable** tab).

### The Debug Tool Bar

Now we switch from examining the tables in the **Debug** perspective to **Debug** toolbar, focusing on the meaning of the debugger commands, followed by applying them to the **collatz.py** module.

The picture below shows the **Debug** toolbar; on top are the meanings of its most import/useful buttons.




The **Debug** toolbar has nine buttons (illustrated above). We click them to control the manner in which the Python executes statements in the script that we are debugging. Briefly, the most important buttons are:

- **Resume**
  - Execute the script until it ends, or until it stops at a breakpoint (unconditional or conditional).
- **Terminate**
  - Terminate a debugging session; we can always start a new debugging session by clicking the **Debug** button again, executing the script from its beginning.
- **Step Over** (for beginners, the most import stepping tool: sometimes calling **single stepping**)
  - Execute the highlighted statement in the **Editor** tab, ignoring the details of any called functions; stop at the statement following it.
- **Step Into**
  - Stop at the first statement inside the body of a function in the highlighted statement.
- **Step Return**
  - Execute all Python statements from the highlighted statement until the end of the function that the highlighted statement is in; stop at the statement that called the function (sometimes called **Step Out Of**); **Step Into/Step Return** help debug programs that call lots of functions/methods.

The **Step Into** and **Step Return** buttons are complimentary and useful only when we are debugging scripts that call functions/methods that we have written (not applicable in `collatz.py`), so we will defer discussion of these buttons until later in the quarter

If any of these buttons appears gray, it means that the button is not currently usable: clicking it has no effect. For example, if no script is running, all the buttons are gray; clicking them has no effect.

### Step Over and Single Stepping

**Step Over** allows us to execute a script very slowly: one line at a time. We use it to observe both the executing code in **Editor** tab and the values bound to the variables it defines in the **Variables** tab as the code executes. By doing so, we can carefully trace the execution of our script easily. Such tracing allows us to understand exactly how Python is executing our script: what code it is executing and how that code is changing the binding of its variables. These actions help us detect and correct any intent errors —differences between what Python is doing and what we wanted Python to do.

The **Step Over** button executes one line in a Python script: the highlighted statement that the arrow is pointing to. If that statement contains any function calls, it executes the complete function calls without showing us any of the function's details (any of the code/variables inside the function). We say that **Step Over** treats all functions as **black boxes**. If we want to monitor the code/variables inside a function, we will use **Step Into**, followed by **Step Over**/**Step Return** after the debugger starts executing code in the function.

Here are some details of using **Step Over** in a script

- The blue arrow/green highlighting indicates the line/statement that Python is **about to execute**; **Step Over** executes that line/statement; the blue arrow/green highlighting moves to the next line/statement that Python will execute. It skips over lines that are blank/contain comments (neither are statements).

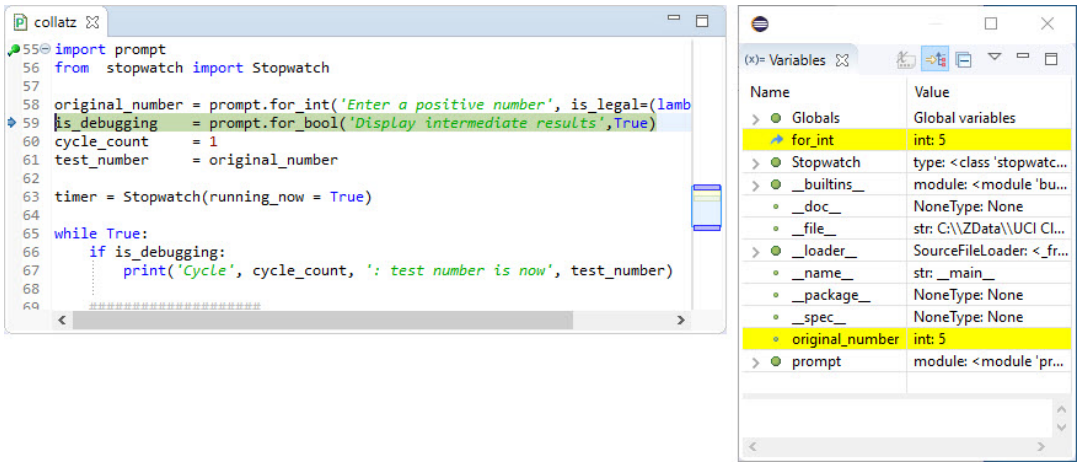
- If a variable is bound for the first time during the execution of the step, it is added to the **Variables** Tab (variables appear in alphabetically order) with its assigned value in the Value column, and both are highlighted in yellow.
- If a variable is already bound in the **Variables** tab is bound to a new value during the execution of a step, the new value appears in the Value column, and both are highlighted in yellow. If a variable's value does not change during the execution of a step, its background reverts to non-ye.
- If a function is called and returns a value, the following information appears in the **Variables** tab: a blue arrow, the name of the function, and the value it returns, all highlighted in yellow.

Start debugging the **`collatz.py`** script. The blue arrow/green highlighting refers to line 55/the statement **`import prompt`**. If you have the screen space, drag/drop the **Variables** tab out of the **Debugger** view onto your desktop and enlarge it to see more variables. Perform the following steps:

1) Click the **Step Over** button (Python imports the **`prompt`** module); notice that the name **`prompt`** appears in the **Variables** tab (and its value shows it to be a **module**) highlighted in yellow. The blue arrow/green highlighting now refers to the statement on line 56.

2) Click the **Step Over** button (Python imports the **`Stopwatch`** class from the **`stopwatch`** module); notice that the name **`Stopwatch`** appears in the **Variables** tab (and its value shows it to be a **class**) highlighted in yellow (the line for **`prompt`** returns to normal: it was not bound to a new value in this line). The blue arrow/green highlighting now refers to the statement on line 58.

3) Click the **Step Over** button once (Python calls the **`prompt.for_int`** function); notice that the prompt text (**`Enter a positive number:`**) appears in the **Console** tab and the debugging icons become gray: we have to enter a value in the **Console** tab before we can issue any more debugging commands. Enter the value **`5`** in the **Console** tab and press Enter. (a) The **Variables** tab indicates that the f**`or_int`** function returned the value **`5`** highlighted in yellow; (b) the name **`original_number`** appears in the **Variables** tab (and its value shows it to be the **int `5`**) highlighted in yellow (the line for **`Stopwatch`** returns to normal: it was not assigned a value in this line). The blue arrow/green highlighting now refers to the statement on line 59. At this point he **Editor** and **Variables** tab appear as:




4) Click the **Step Over** button once (Python calls the **`prompt.for_bool`** function); notice that the prompt text (**`Display intermediate results[True]:`**) appears in the Console. Press Enter to use the default value: **`True`**. (a) The **Variables** tab indicates that the f**`or_bool`** function returned the value **`True`** highlighted in yellow; (b) the name **`is_debugging`** appears in the **Variables** tab (and its value shows it to be the **bool `True`**) highlighted in yellow (the lines for **`original_number`** and the return from the **`for_int`** function return to normal). The blue arrow/green highlighting now refers to the statement on line 60.

5) Click the **Step Over** button once (Python binds the name **`cycle_count`**); notice that the name **`cycle_count`** appears in the **Variables** tab (and its value shows it to be the **int `1`**) highlighted in yellow (the lines for **`is_debugging`** and the return from the **`for_int`** function return to normal). The blue arrow/green highlighting now refers to the statement on line 61.

6) Click the **Step Over** button once (Python binds the name **`test_number`**); notice that the name **`test_number`** appears in the **Variables** tab (and its value shows it to be the **int `5`**) highlighted in yellow (the line for **`cycle_count`** returns to normal: it was not assigned a value in this line). The blue arrow/green highlighting now refers to the statement on line 63.

7-…) Continue clicking the **Step Over** button and observe how Python executes this script: how the control structures determine which lines are executed in the **Editor** tab and which variables change their state in the **Variables** tab. Observe that the standard information is printed in the **Console** tab. It will take `6` cycles before the test number becomes `1` and terminates the loop and prints the statistics. This is a slow but simple way to execute a script. Below we will discuss breakpoints and the Resume button, whi is a faster (and more focused) ways to jump to "lines of interest" and the possibly single step from these lines.

**Important Point**: The blue arrow points/green highlighting refer to the line that is **about to be executed**; when we click the **Step Over** button, Python executes the line being referred to. It is a common misconception that the blue arrow/green highlighting refers to the line that has just been executed. You need to know the difference, which is sometimes critical.

### Stepping over Code that inputs Information in the Console Window

Whenever we **Step Over** a line that requires input from the user, the **Console** tab becomes critical; typically, it contains a prompt telling the user what information to enter, and it waits for the user to enter this information. You will notice that the debugging buttons become gray (indicating we must do something else —enter the requested information— before returning to our stepping). The **Console** tab is automatically selected. Although the cursor appears at the front of the prompt, if we type a value it appears in the **Console** tab after the prompt.

### Terminating a Debugging Session (and possibly restarting one)

To terminate a debugging session (possibly to begin stepping through the script from the beginning again: it is easy to step too far, which requires going back to the beginning) click the **Terminate** button () either on the **Debug** toolbar or to the right of the **Console** tab. In both cases the debugger terminates the script immediately: the blue arrow/green highlighting disappear from the **Editor** view. The **Debug**/**Console** tabs show the script to be terminated. To start debugging over again, click the **Debug** button again. We can do both terminate/restart operations by clicking the **Restart** button(  ) to the right of the **Console** tab.

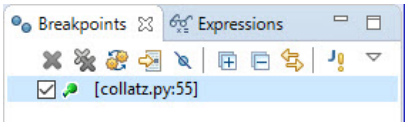
### Breakpoints: Unconditional and Conditional

The **Step Over** button allows us to observe the statements executed in our scripts. At each step, the **Variables** tab allows us to observe the changes made to the bindings of variables. **Step Over** allows (and forces) a very fine-grained view of our script. By setting a breakpoint on a line, we can run the script —at high speed— until the breakpointed line is about to be executed (instead of tediously single stepping to that line of interest). Often, we need to ignore many early lines in our script; stop somewhere in the middle of it; and then **Step Over** that line and subsequent lines carefully, observing changes to our variables. This requirement is easily met by setting breakpoints, which come in two varieties: *unconditional* and *conditional*. Let us look at each kind of breakpoint separately. We cover unconditional breakpoints first, because they are simpler. **Important**: We can set breakpoints only on lines containing Python statements: not blank lines nor on lines that contain only comments.

### Setting Unconditional Breakpoints

When we set an unconditional breakpoint on a line, the **Debug** perspective will stop execution of the script whenever it is about to execute that line. This might happen when we first click the **Debug** button, or while we debugging the script later, after we click the Resume button.

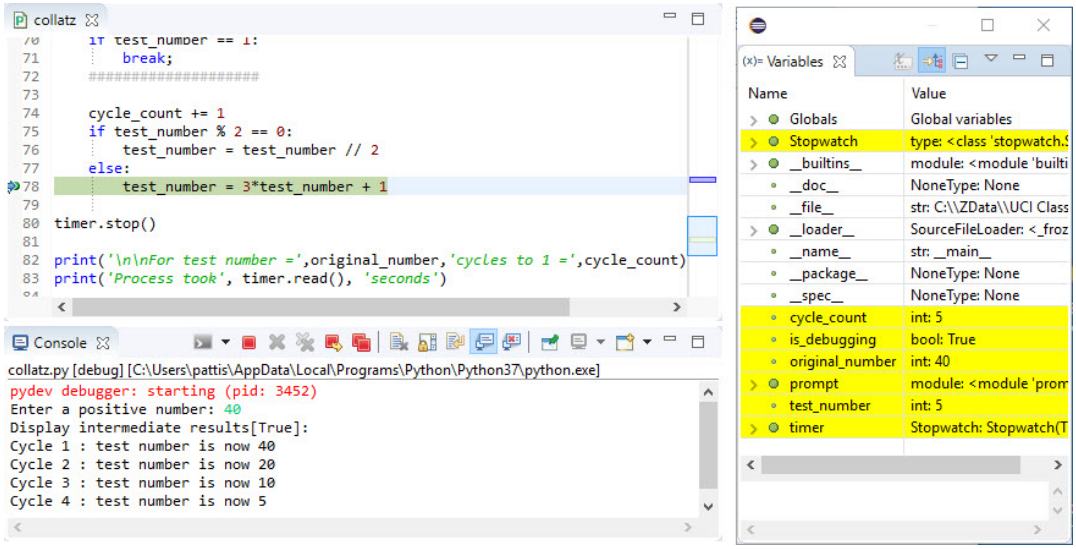
The easiest way to set an unconditional breakpoint on a line/statement is to double-click in the margin to the left of its line number in the **Editor** view (as we did above for `import prompt` line). When a line has a breakpoint set, its left margin changes to the breakpoint icon (  ); also, an entry for each breakpointed line appears in the **Breakpoints** tab: the entry shows whether the breakpoint is enabled (the box is checked if it is: we can easily enable/disable a breakpoint, or we can permanently remove it) the name of the function (if the breakpoint is in a function: ours aren't), and finally in brackets the name of the module the breakpoint appears in (and its line that module): if we select the Breakpoints tab in our example above, it displays as




We can simultaneously set breakpoints on many different lines in a module (in fact, we can set them in many different modules, if our program uses multiple files). When we start debugging the script, by clicking the debug button, Python runs the script, executing lines until it is about to execute any breakpointed line (if there are none, it executes the entire script, and the PyDev perspective does). The **Debug** perspective shows us which breakpointed line was reached first by the blue arrow/green highlighting in an **Editor** tab. When Python stops before executing that line, all changed bindings in the **Variables** tab (since the last debugging command) will appear with a yellow background.

In the **collatz.py** script (it should be stopped on the first line; if not, terminate it and click the **Debug** button again), click the column on the left of the **Editor** view, to the left of the line 78, which contains the statement **test_number = 3 * test_number + 1**, which is executed whenever **test_number** is odd (its execution is controlled by the **if** statement). Observe the debug icon to the left on this line and the entry in the **Breakpoints** window.

Click the Resume button to execute this script until the next breakpoint (or the end). When prompted, enter the value **40** and **True** –by just pressing Enter). The script stops on the breakpointed line 78 (with the blue arrow/green highlighting referring to this line), the first time that it is about to be executed: test number is **5** and now odd. Observe the **Editor**, **Console**, and **Variables** tabs when the script stops before executing this line. They should look as follows, showing in yellow all the script variables that were bound (and rebound) to values since we clicked the **Resume** button. Notice we are on **Cycle 4**. We are done with the breakpoint on line 78; double-click it to remove the breakpoint.




Click the **Resume** button again; again, observe that the program now terminates (there were no more odd numbers). Click the **Debug** and **Resume** buttons again to re-execute the script. This time enter a value of **17** to the first prompt and watch the more interesting behavior when clicking the **Resume** button to execute the program.

## Removing (or Unsetting) Breakpoints

If we no longer want the debugger to stop the script on a breakpointed line/statement, we can easily remove its breakpoint. The easiest way to remove a breakpoint from a line/statement is to double-click the breakpoint icon in the left margin of a line number. The breakpoint icon will disappear. The entry for that breakpoint will also disappear from the **Breakpoints** tab. We can also remove a breakpoint by right-clicking the breakpoint in the **Breakpoints** tab and selecting the **Remove** option.

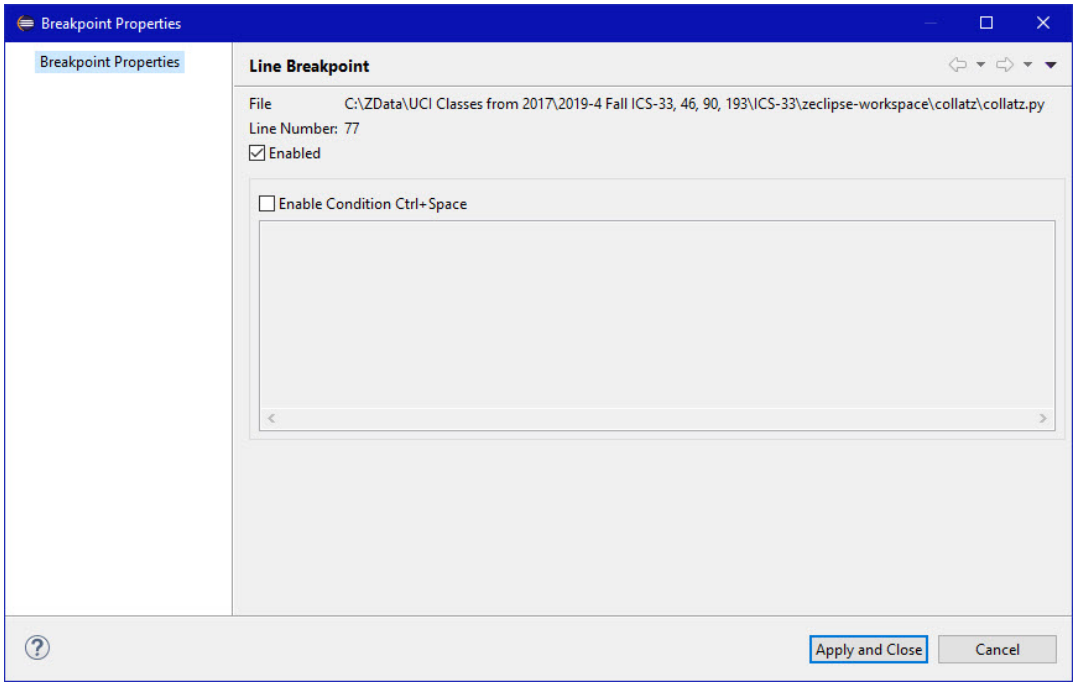
If we do not explicitly remove a breakpoint, and then terminate the script and run it again, that breakpoint (and any others that we set and did not remove) will still be there. With this feature, we can easily execute the script repeatedly, keeping the same breakpoints. In fact, if we terminate Eclipse, it will remember what we were debugging and what breakpoints were set when we restart it. But, if we terminate Eclipse while running the code (in the **PyDev/Debug** perspective), we must rerun the code.

Rerun the script in the **collatz.py** file a few times in the **Debug** perspective. Terminate Eclipse, restart it, and run the script again. Notice that the breakpoint that you set previously is still there; finally, remove the breakpoint at line 78, but leave the breakpoint at line 55.

## Setting Conditional Breakpoints

Sometimes we need to stop at line in a script, but not **every time** that the line is about to be executed: instead, we want to be more restrictive, and to **stop only when some special condition holds**. Conditional breakpoints are an advanced feature that allows us this extra control by allowing us to attach a **bool** condition to any existing breakpoint. When the script is about to execute a line with a conditional breakpoint, Python first evaluates the condition (written as a **bool** Python expression): it stops the script before executing that line only when the condition evaluates to **True**. This simple mechanism increases the utility of breakpoints tremendously, and is another reason to pay close attention to the structure and evaluation of **bool** expressions. Here is how to set a conditional breakpoint and specify its condition.

- Right click the breakpoint icon (and select **Breakpoint Properties…**); a box like the following will appear.

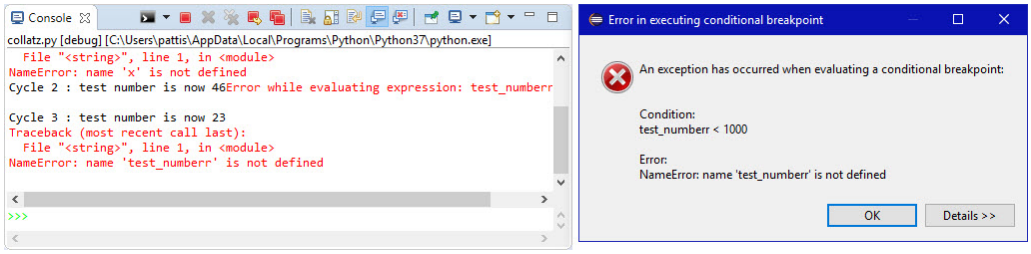



- Check the Enable Condition checkbox; the text area beneath it becomes white, allowing us to type the breakpoint's condition there.
- In the white text area, enter a valid condition (any legal Python **`bool`** expression). The expression must refer only to variable names that are bound to values at the time the breakpointed line will be executed.
- Click the **Apply and Close** button.

We can reexamine/change the condition for any breakpoint by retrieving its properties window and editing the text area that contains its condition (so if we make a mistake, we can easily fix it).

In the **`collatz.py`** script (it should be stopped on the first line), set an unconditional breakpoint on line 70: **`if test_number == 1:`** Then make it a conditional breakpoint by typing the Boolean expression **`test_number < 1000`** (don't type an **`if`**). Now click the **Debug** button to run this script. Click the **Resume** button and enter the value **`7777`** when prompted. The **Debug** perspective will stop on the breakpointed line the first time that the condition is **`True`**. Notice the variables in the **Variables** tab when the script stops: **`test_number`** is **`923`** (which is <**`1000`**) and **`cycle_count`** is **`17`**. Scroll the **Console** tab to see what output the script produced during these seventeen cycles; note that all prior values for **`test_number`** were ≥ **`1000`**. Click the **Resume** button again and observe what happens: Python stops with **`cycle_count`** storing **`33`** and **`test_number`** storing **`658`** (the only two variables in the **Variables** table that have changed). Terminate the script and debug it again, with the same conditional breakpoint; this time enter a different value when prompted (again, a value ≥ **`1000`**). Observe a similar pattern. Notice that when a program runs with conditional breakpoints, its execution speed diminishes (by a lot for a complex **`bool`** expression); this is the price we must pay for using such a powerful tool.

Finally, if we enter an illegal Boolean expression for a condition (typically illegal because of bad variables names or bad operator syntax), the **Debug** perspective does not notice the error immediately; it notices it only when the condition is checked by the **Debugger** perspective when Python runs the script (the first time the condition on the breakpointed line is checked). At that time, Python stops executing the script, displays an error message in the **Console** tab, and displays a pop-up window describing the error. Here is what things look like if we mistype the name **`test_number`** in a conditional breakpoint.




If we click **OK** on the pop-up window, we can continue executing the script (e.g., click the **Resume** or **Step Over** button). Each subsequent time it evaluates that breakpoint, the same thing happens. We

can re-edit the condition in the breakpoint properties box to correct the breakpoint expression before continuing to execute the script in the debugger, which will then use the updated condition.

### Disabling Breakpoints (and re-enabling them)

Sometimes it is useful to temporarily disable a breakpoint, rather than completely remove it. A disabled breakpoint has no effect, but it can easily be re-enabled to restore its effect. A removed breakpoint has to be set all over again (a removed conditional breakpoint requires further clicking and retyping the Boolean expression to recreate it). To disable a breakpoint:

- Right-Click the breakpoint icon (to the left of the breakpointed line/statement) and chose **Disable Breakpoint**; it will change to a gray icon and the checkbox in the **Breakpoints** tab will be unchecked. Caution: if we double click the breakpoint icon it will be removed! So think before you click.
- Uncheck the checked checkbox next to the breakpoint in the **Breakpoints** tab; the breakpoint icon in the Editor tab will become gray.

Finally, to re-enable a disabled breakpoint, either

- Right-Click its gray breakpoint icon (to the left of the breakpointed line/statement) and chose **Enable Breakpoint**; it will change back to a green icon and the checkbox in the **Breakpoints** tab will be checked. Caution: if you double click the white circle it will remove the breakpoint
- Check the unchecked checkbox in the **Breakpoints** tab; the effect will be the same.

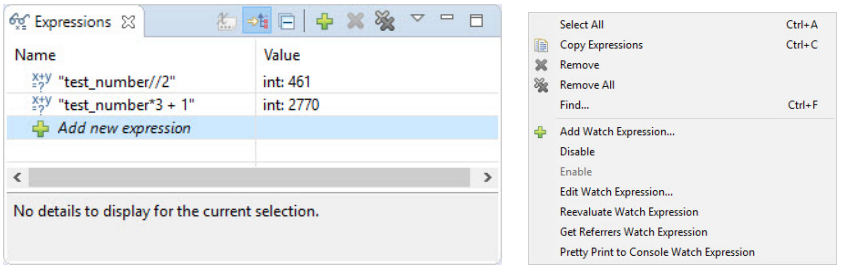
In the **`collatz.py`** script (it should be stopped on the first line/statement), disable the breakpoint on line 70 that you set above. Then, click the **Resume** button to run this script: it will not stop on a line with a disabled breakpoint. Then re-enable the breakpoint and restart the script, entering some value ≥ **`1000`**. Now the script stops on the line with the re-enabled breakpoint.

Finally, we can also disable a breakpoint (by unchecking the **Enabled** checkbox; not the **Enabled Conditional** checkbox). Furthermore, we can disable just the **condition** on a breakpoint (by unchecking the **Enable Condition** checkbox) in its **Breakpoint Properties** pop-up window. In the latter case, converting it from a conditional breakpoint to an unconditional breakpoint. The condition remains in the text area, though, so by rechecking this checkbox we can restore it to the its original state (as a conditional breakpoint with the original condition). So, we can easily toggle checking/not checking a breakpoint, and turning a conditional breakpoint into an unconditional one.

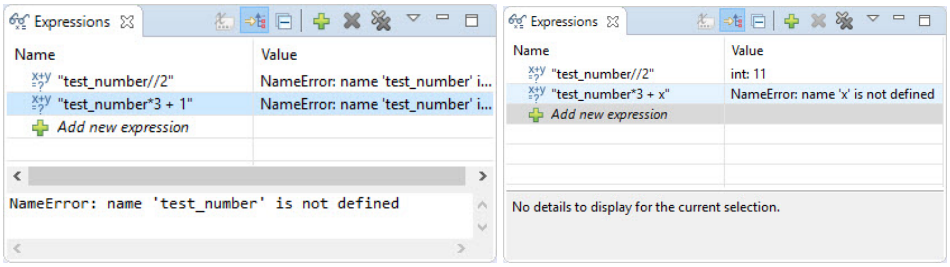
### Watching Expressions

It is possible to watch the values of arbitrary expressions, not just variables. For example, we can easily watch both the values **`test_number//2`** and **`3*test_number+1`** (the two values **`test_number`** might be rebound to) by viewing the **Expressions** tab (select **Window | Show View | Expressions** if you don't see it), clicking the green plus (**Add new expression**), typing any Python expression and pressing Enter. The **Expressions** tab below on the left will be added to the Debugger views holding the **Variables** and **Breakpoints** tabs, updating the values of these expressions whenever **`test_number`** changes. See how the Expressions tab would look below on the left.

We can further manipulate any expression in the Name list by right-clicking it. The menu below on the right will appear: the most commonly used operations are **Remove**, **Remove All**, **Disable**/**Enable**, and especially **Edit Watch Expression** to change the expression.

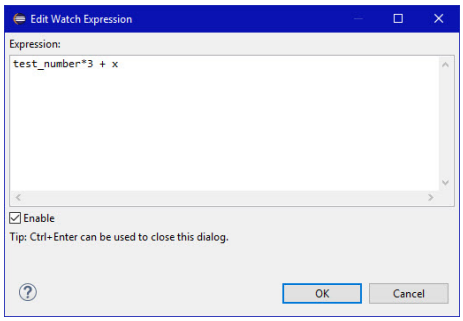



If there is no value yet for **`test_number`**, an error will appear in the Value column as follows.



If a watched expression contains a syntax error (e.g., an unknown name or poor use operators), an error message appears in the Value column (as illustrated above on the right).

We can always right-click this expression and select **Edit Watch Expression** to edit the expression (in a window like the conditional breakpoint window) and correct the error, if there is one. Eclipse will redisplay its Value (or show another error). We can also disable watching an expression in this window, and re-enable it later.




Last Word: Restoring Tabs

To restore any views (like the **Variables** or **Breakpoints** tabs) if you remove them, click the Window pull-down menu, then select Show View, and finally select the view you want to see.

Finally, when we terminate the debugger, Eclipse saves the contents and locations of all these windows. So, when we begin another debugging session (possibly after closing the entire project and reopening it at a later date), the desktop looks identical to when we last terminated a debugging session.

Pause and reflect

For now, you are done with this handout. Please ensure that you have an operational understanding of the various windows, views, and tabs that are part of the debug perspective, and how to use the Debug toolbar to single step through a script, and how to set, remove, and disable unconditional and conditional breakpoints.

A Peek at Other Debugger Features

**Viewing Complicated Data**: Complicated data (such as objects representing lists, dicts, and other classes) often show up in the **Variables** pane with disclosure triangle. Click the triangle to toggle between disclosing(seeing)/eliding(not-seeing) the elements of the data. There are also special watch windows for displaying the members of complex data. See the next bulleted item for why this is useful.

**Changing Values**: We can use the debugger to change the binding of simple variable or the values stored in variables referring to complicated objects. This operation is a bit delicate, and there are very very very (yes, not a typo) few times when it is useful. We accomplish this by right-clicking a variable in the **Variables** tab and selecting **Change Value** (and then entering the new value in the text box).

**Stopping an infinite loop**: If a loop is infinite (no progress is being made towards loop termination inside the body of the loop), we can stop it in the debugger by clicking the **Suspend** button. Typically, the **Debug** tab will have lots of names in it, and the programmer must click the **Step Return** button multiple times to return to the script. Then the programmer can use **Step Over** to locate the infinite loop, and continue to use stepping to diagnose why the loop is not making progress toward termination. Often, we observe an infinite loop by the **Console** tab printing nothing (it almost looks like the script has stopped) or printing the same thing repeatedly, over and over again.

**Step Into/Step Return**: When we begin writing our own functions (instead of just calling functions that are already written in the course library) we will discuss Stepping Into and Stepping Out Of, which treats functions as white/transparent boxes: when we Step Into a line containing a function call, the debugger goes into that function so that we can see its parameters, and how it works by executing its lines one at a time. We can single step in a function, Step into any functions that it calls, or Step Out that function to the line following the one where the function was called.

**Debugging Strategies**: We will also discuss general strategies for quickly locating errors in programs. Often once the location of a bug is detected, we can easily determine how to fix it.

⌨ Problems to Solve: with solutions on the next page (don't peek unless you are stuck)

Solve each of the following problems by setting breakpoints on various lines in the **`collatz.py`** script; most of the breakpoints also require conditions. **Do not change the script or single step to find the answers (use breakpoints to find them much faster)**. First determine where to set the breakpoint, then determine under what condition to stop at that breakpoint (this reduces one hard task to two simpler ones).

- For an input of **`5184`** what cycle is the first one where **`test_number`** is odd?
- For an input of **`18743`** on what cycle does **`test_number`** first drop below **`1000`**?
- For an input of **`77777`** does **`test_number`** ever equal **`1336`**? If so, on which cycle?
- For an input of **`77777`** what values does **`test_number`** have on cycles **`20`**, **`40`**, **`60`**, etc.?

There are simple and elegant ways to solve all of these problems.

## Solutions

Here are the solutions for the problems above. When answering questions about the current `cycle_count` and `test_number`, it is convenient to set a breakpoint on the `if test_number == 1:` line, which occurs directly after their current values are displayed to the console window.

- For an input of `5184` what cycle is the first one where `test_number` is odd?
  - Answer: on cycle `7` (`test_number` is `81`)
    1) Set a breakpoint at the line starting `if test_number == 1:` with the following condition: `test_number % 2 == 1`
    2) Set an unconditional breakpoint at the line `test_number = 3*test_number + 1;` this stops on the first odd number, but AFTER `cycle_count` has been incremented (so it will show as `8`, not `7`).

- For an input of `18743` on what cycle does `test_number` first drop below `1000`?
  - Answer: on cycle `81` (`test_number` is `572`)
    Set a breakpoint at the line starting `if test_number == 1:` with the following condition: `test_number < 1000`

- For an input of `77777` does `test_number` ever equal `1336`? If so, on which cycle?
  - Answer: on cycle `100` (`test_number` is `1336`)
    Set a breakpoint at the line starting `if test_number == 1:` with the following condition: `test_number == 1336`

- For an input of `77777` what values does `test_number` have on cycles `20`, `40`, `60`, etc.?
  - Answer: on cycle `20`, `test_number` is `6922`; on cycle `40`, `test_number` is `1853`; on cycle `60`, `test_number` is `496`; on cycle `80`, `test_number` is `137`; on cycle `100`, `test_number` is `1336`;
    Set a breakpoint at the line starting `if test_number == 1:` with the following condition: `cycle_count % 20 == 0`; here `cycle_count` divided by `20` has a remainder of `0` when it is `0`, `20`, `40`, `60`, etc.: some multiple of `20`.

Problem Set for Programming Assignment #0

Use the **Debug** perspective to observe the script in the **craps.py** script (find the **Craps Statistics** script by following the **Sample Programs** link on the course homepage). Before proceeding, please read all the comments at the top of that script, which describe it; then, run it a few times to understand how it works. Use **Step Over** repeated to observe the **Editor**/**Variables** tabs to understand in more detail how the program works. Below are a variety of questions about this script, all of which can be answered by using various combinations of the standard debugger commands: i.e., observing the **Variables** tab while stepping/running the script under the control of unconditional and conditional breakpoints (similarly to the problems/solutions in the handout above).

For example, question 1 (below) can be solved by observing the variable **game** (which keeps track of the game number: first, second, third, etc.) and setting a breakpoint at the unique line that is executed when a loss occurs on a first throw of the dice.

Determine **both** the answers to the questions and a general explanation of the functions (the debugger commands: where and with what conditions) that you used to answer these questions. For the solution you submit, just supply the answers (not the general explanation).

To start the assignment, you will have to download and modify the **crap.py** script.

- Download the **craps** project zip file onto your desktop and then unzip it.
- Open Eclipse and use the Python perspective to start the project.
- Open an Editor view of the **craps.py** script in it. Change blank line 61 to read
  `dice.standard_rolls_for_debugging(18894285)`
  - By making this change to the script, the **Dice** will generate a standard sequence of rolls, so its results will be identical each time that you run the script. This is a very useful feature when the script you are debugging, or trying to gather information about, uses a random number generator. You must use this number! Best to copy/paste it into your code to ensure fidelity.
- Run the script after this change. For the same number of games, you will always get the same statistics. Then switch to the **Debug** perspective. Ensure that the script runs correctly before setting any breakpoints.
- **Make no other changes to the script: this assignment does not require programming, just the ability to read a script, understand it, and run it using the debugger.**

Single step through the script for a few games to better understand the execution of this script and its control structures. Specifically, watch how the information stored in the variable **roll** controls what statements the script executes; also observe where/how the script changes **game**, **win_count** and **lose_count**.

Determine how to use debugger commands to help you answer the following questions. Each question requires its own/different debugging session, starting the program from the beginning. Make no changes to the script itself.

**1. What number game is the first to be lost on one roll of the dice (e.g., 2, 3, or 12)?**

**2. What are all the dice rolls in the seventh game (list them in the order thrown)?**

**3.What number game is the first to be won on a roll of 7? …on a roll of 11?**

**4.What number game is the first to be lost on a roll of 2? ..on a roll of 3? ..on a roll of 12?**

**5.What number game is the first to be won by making a point of 5? … a point of 10?**

**6. What are all the dice rolls in game 103?**

**7. In what game number does the 712th win occur?** (this script can run for a few minutes)

Hints

Enter **2000** when prompted **Enter # of games to play:** all answers correspond to earlier games than game **2000**. This bound is important because a script with breakpoints (especially conditional breakpoints) will run more **slowly** when compared to the same script without breakpoints (I estimate twice as slow with one breakpoint). So, once you set your breakpoints, it may take some time for the debugger to stop the script on the appropriate line (but still only a few seconds): if you set incorrect breakpoints, you do not want the script to execute for more than a few second before you find out.