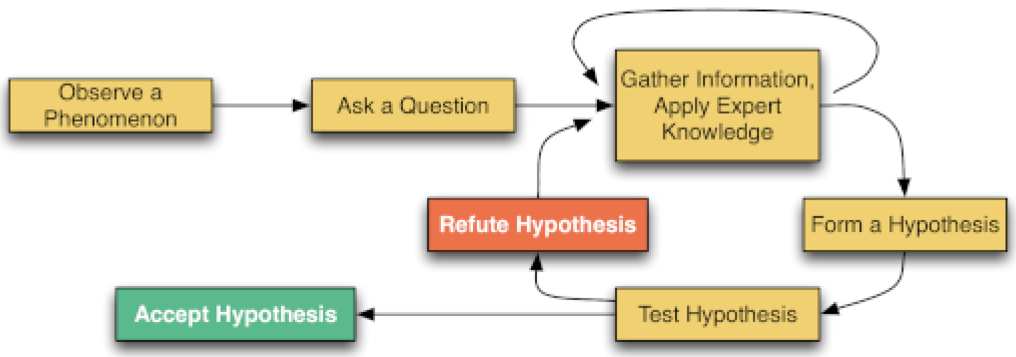
The Scientific Method



Observe a Phenomenon

The figure above shows a flow-chart of the scientific method. All science starts with observing some phenomenon. In the natural sciences this might be noticing that objects fall towards the ground when not supported by anything, realizing that finches in certain of the Galápagos Islands have different characteristics from other finches, or that the water level rises when you get into the bathtub.

In programming, our observation of phenomena relates to the behavior of our programs on certain inputs (“My program gives the wrong answer when I give it an input of 42!”). These observations typically arise from our test cases, but may happen under other circumstances (for example, the end user reports a problem that we did not discover in testing).

Ask a Question

Once you have observed a phenomena, the next step in the scientific method is to ask a question. Asking a good question here is crucial to the success of the rest of our scientific endeavor. While a broad question such as “What is wrong with my program and how do I fix it?” may seem appealing, it may be quite difficult to answer. Instead, we should aim for more focused questions: “On which line of code does my program crash?” or “Why does my program call *myFunction* with *x=3* and *y=-42*?”.

Answering one question often leads to an observation that leads to another question—restarting the scientific process all over again. Discovering what is wrong in this iterative fashion is perfectly fine, and in fact a great way to proceed. You start by asking “Which line does my program crash on?” then when you answer that, you ask “Why does it crash on this line?” the answer to that then leads you to ask “How is *x=3* and *y=-42*?” which in turn leads you to ask another question, and so on. Eventually, your chain of questions and answers leads you to the discovery of the problem, even if it somewhat far removed from the visible symptom.

Gather Information, Apply Expert Knowledge

Many people will say that forming a hypothesis is the next step of the scientific method. If you can form a hypothesis immediately, that is great. However, forming a good hypothesis is difficult, and forming one right away is often not possible.

The next step of the scientific method is actually to gather information and combine it with your expert knowledge. Going back to our example of visiting the doctor, the doctor gathers information by examining the patient, and combines it with her expert knowledge—years of training on symptoms of diseases and how the body works—to form a hypothesis.

In the case of debugging, you need to gather information about what is happening in your program, and combine this with your own expert knowledge on programming. Your expert knowledge comes in two parts here. One is your knowledge of programming in general—the rules for how to execute code by hand that we learned in previously (and will continue learning as we introduce more topics), and your domain knowledge of the particular program you are writing—the expected behaviors of each part of it.

Your expert knowledge will grow with practice in programming, and the domain for which you are writing programs. However, gathering information effectively is a skill of its own. The information gathering aspect of debugging is often conflated with the entirety of debugging—if you ask someone how they debug, they will often explain to you what techniques they use to gather information.

The simplest way to gather information is to insert print statements (in C, calls to *printf*) to display the values of various variables at various points in the program. The resulting output can give you information about the control flow of the program (which statements were executed, and in what order—as shown by what order your print statements print their output), and, of course the values of the variables that you print.

Gathering information by printing has the advantages that it is simple and requires no other knowledge or experience. However, it has several disadvantages as well. One is that changing what you print out requires recompiling and re-running your program. While this disadvantage may seem small, if your bug takes 15 minutes to manifest, restarting the program for each new piece of information you discover that you want can be quite time consuming. Another disadvantage is that the output may be overwhelming (*i.e.*, thousands of lines of output to sift through) if your program executes for even a modest time before experiencing the problem. A third disadvantage is that it cannot replicate or replace many features that debuggers offer.

Another approach to information gathering is to use a *debugger*—a tool specifically designed to aid programmers in the debugging process. The debugger is in fact primarily aimed at this piece of the debugging process—gathering information (sadly, it does not offer you hypotheses or expert knowledge). One widely used debugger is *gdb*, which we cover in detail in the future. We strongly recommend that you learn it, and if you intend to become a serious programmer, become an expert in it. We will mention generally what you can do with it here, but leave the details until later as *gdb* has features which we relate to topics that we have not learned yet.

For now, we will discuss the high-level points of a debugger. When you run your program inside the debugger, you can give the debugger a variety of commands to control the execution of your program, and get information from it. Note that *Emacs* understands how to interact with *gdb* and using them together makes the entire process go much more smoothly.

When you run your program inside of a debugger, it will run as normal until either (a) it exits (b) it crashes (c) it encounters a *breakpoint* (or *watchpoint*) that you have set. A breakpoint is set on a particular line of code, and instructs the debugger to stop the execution of your program whenever the execution arrow is on it. Breakpoints can be conditional—meaning you can specify that you only want to stop on some particular line when a conditional expression you specify is true. Watchpoints specify that you want to stop the program when a particular "box" changes.

Once your program is stopped, you can examine the state of the program by printing the values of expressions. The debugger will evaluate the expression, and print the result for you—giving you information about the state of the program. Most often, you will want to print the values of variables to see what is going on, though you may print much more complex expressions if you wish.

After printing some information, you will often want to continue executing in some fashion—either running until the debugger would stop naturally (as described above), or maybe just executing one more statement, then stopping. The debugger gives you the ability to choose either one. If you want to execute one statement at a time, and the current statement involves a function call, you have two options. You can either step over the call (asking the debugger to evaluate the entire function call, and stop on the next line of the current function), or you can step into the function call (asking the debugger to follow the execution arrow inside the function and let you explore what is happening inside it).

This approach to gathering information is more flexible than print statements—if you encounter one oddity, which suggests other things you need to explore, you can print them immediately. By contrast, if you print the value of a variable with a print statement, adding more print statements to investigate other variables requires recompiling and re-running the program.

The previous paragraph alludes to a common occurrence in the debugging process: recursive observations—gaining some information (*e.g.*, seeing the value you printed for one variable) leads you to want some other information (*e.g.*, to print some other variable). Often when you are investigating one phenomenon ("My program crashes when I enter 3…"), you observe some other phenomenon ("y is 0 on line 42..") which itself leads to a question meriting investigation ("How did that happen?"). The investigation of this second phenomenon proceeds according to the scientific method. In such a case, you are recursively applying the scientific method. We will learn about recursion as a programming technique later, but for now it suffices to say that recursion is when an algorithm (in this case, the scientific method) has a step which calls for you to apply the same algorithm to "smaller" (by some metric) inputs. Here, investigating the second observation may immediately solve your problem, may give you useful information to allow you to proceed, or may prove to be a red-herring (something that was actually fine, but just surprised you—in which case, you just continue gathering information for your original question).

Form a Hypothesis

The whole point of gathering all of this information is to help you form a hypothesis. Sometimes, you may be able to form a hypothesis right away—typically for problems that are simple relative to your experience level. However, forming a good hypothesis is generally hard, and requires significant information gathering.

Forming a good hypothesis is the key to proceeding effectively. A vague hypothesis is hard to test, and not so useful in identifying the problem. As an extreme example, the hypothesis “My program is broken” is easily verified, but rather useless. The hypothesis “My program is dividing by 0 on line 47 for certain inputs” is more useful, but could be improved. Even better would be “My program is dividing by 0 on line 47 if *y* is odd and *z* is a perfect square.” This (contrived) hypothesis is specific and clear—giving it two important characteristics for debugging.

The first characteristic of a good hypothesis that this exhibits is that it is testable. For a hypothesis to be testable, it must make specific predictions about the behavior of the program: when I give the program inputs that meet (*condition*), I will observe (*behavior*). For such a hypothesis, you can execute test cases to either refute this hypothesis (*e.g.*, if the program’s behavior does not match the predictions that the hypothesis makes) or to become confident enough in our hypothesis that we accept it. The contrived hypothesis we presented at the end of the previous paragraph is quite testable: we specify a certain category of inputs (*y* is odd and *z* is a perfect square) and exactly what behavior we expect to observe (division by 0 on line 47).

The second characteristic of a good hypothesis for debugging is that it is actionable—if we convince ourselves that it is true, it provides us with an indication of either how to fix the error in our code, or what the next steps towards it are. In the case of our contrived hypothesis, confirmation would likely suggest a special case of the algorithm which we did not consider. The fact that our hypothesis is specific (with regards to what types of inputs trigger the error) identifies the corner cases for us, guiding us to the path to fixing the problem.