CS 161

Module 1

Exploration: Problem Solving

Introduction

How do you get from a set of requirements to a working program? There is no cut-and-dried recipe to follow. If there were, someone would write an app for it, and programmers would be out of a job. It can be tempting to just sit down at the keyboard and start typing out code, and that might even work okay for tiny simple problems, but for anything else, it can easily get you into a time-wasting morass of errors that you're not sure how to fix. To help you avoid that outcome, here are some general principles you can follow to help guide you toward a solution:

1. Understand the requirements
2. Design tests
3. Outline your approach
4. Translate your plan into code
5. Run your tests
6. Debug your code
7. Reflect on your process

You may not find all of these steps useful (or at least not have to think about them consciously) for very small programs, but you should definitely keep them in mind for when you get to problems whose solutions are less obvious.

Understand the requirements

In real life this is sometimes the most challenging part, since sometimes your clients themselves won't have a super clear idea of exactly what they want, but for the assignments I give you all the requirements. However you may find it useful to go through the assignment specifications and put them in the form of a checklist. You may also think of something I haven't considered, in which case you should ask me for clarification.

Design tests

It's important to come up with test cases that are representative of how your code could be used. You want them to be good enough that if your code passes all your tests, you will feel confident that it correctly meets all of the requirements. It's also important that for each test, you decide what result(s) you expect from that test. If you don't do that ahead of time, then it can be easy to run a test, briefly look at the output, and convince yourself that it looks reasonable.

How do you know what cases to test?

* Special cases: for example if your program should compute the factorial of a number, the normal rule is that you take the product of all integers from 1 to that number, but zero is a special case, since the factorial of zero is 1, so zero should definitely be one of the test cases.
* Boundary values: for example if a customer gets a special discount when they order 10 or more items, then you should check the numbers on both sides of that boundary (9 and 10) to make sure they're handled correctly.
* Path coverage: you should have test cases that check different possible paths through your logic, to make sure the different paths give correct results.
* Edge cases: extreme or unusual cases that might expose problems.
* Unit tests: when your solution consists of multiple parts that each solve a piece of the problem, it's good to test each part separately, since it's easier to debug a small part at a time.
* Integration tests: once you're satisfied that the different parts work correctly on their own, you need to make sure they work correctly together.

It's always better for you to find cases that break your code than for a customer (or employer) to do so.

Outline your approach

You will want come up with an outline of the steps your code will need to take to meet the requirements. Two useful strategies to help with this are stepwise refinement and working through examples of the problem by hand.

Stepwise refinement

One of the most common methods for outlining your problem-solving approach is *stepwise refinement*. This is a process of breaking up the original problem into some number of smaller sub-problems. Next, you take each sub-problem and break it up into even smaller sub-problems. This is repeated until you get a list of steps that are detailed enough to be translated into code. This list of steps is often referred to as *pseudocode*.

As an example of stepwise refinement, consider the problem of wanting to paint your house, which is a big job. It might seem overwhelming at first, but you can break that top-level task into the following sub-tasks:

* buy paint
* paint house
* clean up

Those sub-tasks can in turn be broken down further:

* buy paint
  + choose colors
  + choose brand of paint
* paint house
  + tape over parts that shouldn't get painted
  + purchase/borrow supplies (ladders, rollers, etc.)
  + bribe friends with pizza
  + schedule a day/time
  + actually paint the house
* clean up
  + return/put away supplies
  + un-tape the taped bits
  + finish any fine detail work

This process continues until you've defined everything to a level of detail where you feel comfortable that you know every small step that needs to happen.

Work through examples by hand

Brains are pretty smart (most of the time). They know how to do lots of stuff without us really being conscious of the steps we're going through to make that stuff happen. In this method, you solve multiple examples of the problem yourself with pencil and paper, *paying close attention* to each tiny step. Doing this will often help you notice the pattern(s) in how you solve different examples of that problem. This strategy can be used together with stepwise refinement, to help you figure out what the sub-tasks should be.

Translate your outline into code

Take the outline (or pseudocode) you've created and translate each small sub-task into code, but don't do it all at once (unless it's a tiny program). Instead, you should produce a short section of code and then try running it to see if that part does what you expect. If it does, then you can add another short section and run it again, until you've translated your whole outline. If you avoid making lots of changes at once, it's easier to track down any new bugs that pop up.

Run your tests

How do you know if your code does what you expect? By running the tests you designed back in step #2. If you've thought of more tests while coding, add those in. The more testing you do, the more bugs you'll catch.

Debug your code

What if your code doesn't do what you expect? Well then you need to find the problem and fix it. Fortunately, since you're only making a few changes to your code at a time, that means any new problem is almost certainly among those few changes. Often error messages will also give you information about what the problem is and where it occurred. If you don't see the problem at first, try to **methodically** narrow down where it could be. After making corrections, re-run your tests to make sure that part works now, and that you haven't accidentally broken anything else.

Reflect on your process

Reflecting on your problem solving process can help you fine tune it to work better next time. What did you learn about the problem as you went? What tests didn't work out the way you expected? What alterations did you have to make to your program due to failed tests? How could your planned tests have been more complete? What was missing or needed to be altered from your initial design, and why? What problems did you encounter during implementation? How were you able to solve those problems? What outside sources (sites, books, or other materials) did you find helpful? How can you generalize any parts of your problem solving experience in a way that might help you on future assignments?

Exploration: Some Context

Algorithms

An algorithm is a pattern for how to manipulate and transform information. It gives a sequence of step-by-step instructions for accomplishing some task. For example, you can think of a recipe as a kind of algorithm, since it gives you a sequence of step-by-step instructions for making some food item. So why do I say that an algorithm is a pattern? Because you will usually want your algorithm to work for all cases of a problem, not just one specific case. For example, if you write a program that applies a filter to a photo, you probably want it to work for any photo, not just one specific photo. In order to write an algorithm for this task, you have to see the pattern of what's similar about applying the filter to different photos and express that pattern in your algorithm.

Computers

We take computers for granted - they're a ubiquitous part of modern society. However they're really a pretty amazing concept. Not so much dedicated computing devices that just carry out one kind of task - those are more or less an advanced form of the tool-making that people have engaged in for millennia. But the idea of a general purpose programmable machine that can carry out any computational process you describe to it (as an algorithm) was revolutionary. Not right away - Charles Babbage first described the idea for his Analytical Engine in 1837, and it wasn't until 1941, more than a century later, that the first such machine was built - but once technology and people's imaginations caught up with the idea, it transformed our world.

Most modern computers follow a similar architecture that has the following basic components:

1. A CPU (central processing unit) is the processor that decodes and executes machine language instructions. These days many computers contain multiple processors.
2. RAM (random access memory) is not actually random. "Random access" just means that you can access data anywhere in that memory. RAM is volatile, meaning that everything stored there disappears when you turn off the computer.
3. Secondary storage devices are where everything that you install or save on your computer is stored. This used to include magnetic tapes, floppy disks, and CDs, but now the most common examples are internal hard drives and USB drives, and sometimes external hard drives for backup purposes or for storing large amounts of data. This kind of memory is slower than RAM, but it doesn't go away when you turn off the computer. When you run a program or open a file, the computer will copy what you need into RAM for faster access. When you install or save something, a copy is made on a secondary storage device.

Computer Languages

I mentioned in the overview that learning to program gives you a language in which to express algorithms. Algorithms are not tied to any specific form of expression, but for a computer to understand and execute our algorithms, we must use a mathematically precise grammar and syntax, i.e. a computer language. Such a language is also useful because it forces us to be **very** specific about how we want to manipulate or transform the relevant information - there is no room for ambiguity.

Computer CPUs only understand their own **machine language**, which is pretty tedious to read or write programs in. "High-level" languages were written to be easier for humans to work with, but in order for the computer to understand them, they have to be translated to the machine language of the computer they're running on. This can be done either all at once beforehand (in a **compiled** language), or as the program is running (in an **interpreted** language). Python is an interpreted language.

Computer Programs

A computer program is just an algorithm that is expressed in some specific computer language.

Python

Python is a computer language created by Guido van Rossum, and was first released in 1991. It was named for the comedy troupe Monty Python, so in material about Python you'll occasionally run across references to their work. There are two main branches of Python - Python 2 (which is being phased out) and Python 3. They are very similar in many ways, but for this course we'll stick to Python 3.

In this course and the next one, you will not learn everything there is to know about Python - that would be impossible. You will, however, learn a lot about Python and about computer programming in general.

Interactive mode and Vocareum examples

In future lessons, you will often see interactive boxes with Python code examples for you to run and experiment with. They will look like this (this first time you'll need to agree to their use conditions, but that shouldn't happen with later ones):

You can execute the sample code by clicking on the "run" button. If you do this now, you should see "Hello world" appear in the bottom box. That is the program output. If a program asks for input, you can type that in the lower box and hit "Enter" on your keyboard. By default, when the script finishes, you'll be in Python interactive mode, so you can print out the values of variables and such.

Under "work" on the left, you can see the file(s) that are part of the example, and you can click on one to see its code. You can also modify that code and re-run it. To do that, **first exit the Python interactive mode by typing "exit()" and hitting the <enter> key**. Now you can click the "run" button again to see the results of any modifications you made. I will sometimes give instructions for things to try, but I encourage you to also try your own experiments, to satisfy your curiosity and make sure you understand the topic before moving on. If at any point you want to revert back to the original code, you can click on the "reset" button at the top right.