How to Program Software

Do you have a perfect idea for a program, but don't know how to turn it into a reality? Learning a programming language takes time, but many successful programmers are self-taught. Once you learn to think like a programmer and get the...

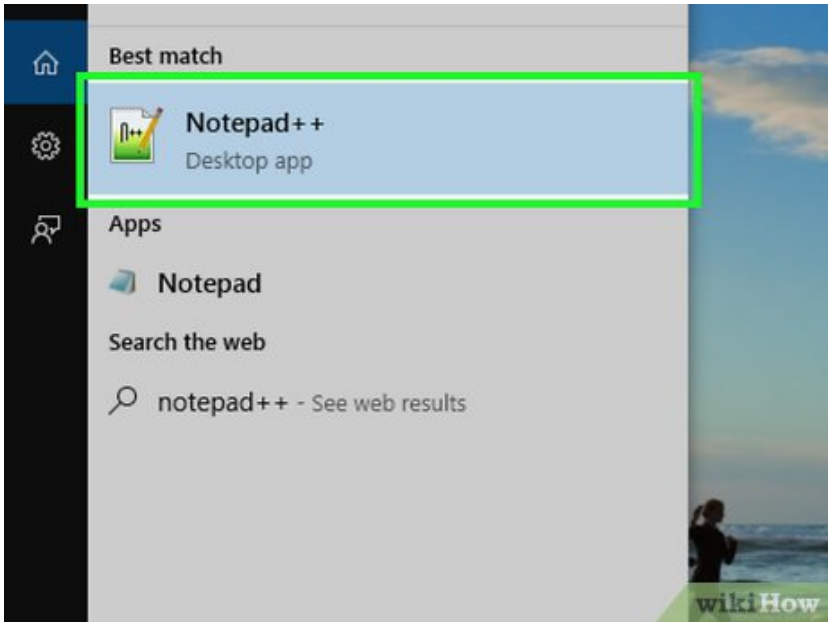
Part 1 of 7:

Learning a Programming Language

1. 

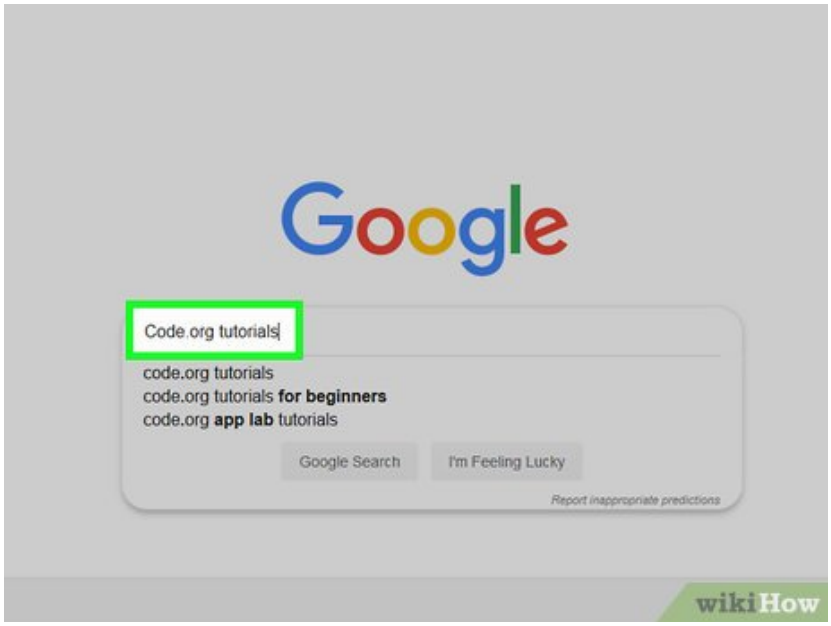
Decide on a starting language. If you've never coded before, you'll want to start with a language that is geared towards beginners, but still lets you work towards accomplishing your goals with your program. There are dozens of languages to choose from, and all excel at different tasks and implementations. Some of the most popular languages for new developers include:[1]

1. C - One of the older computer languages, but still widely-used. Learning C will also give you a leg up when you expand to C++ and Java.
2. C++ - One of the most popular languages in use today, especially in software development. Learning C++ will take a while, and mastering it even longer, but there are countless opportunities once you know it.
3. Java - Another incredibly popular language that can be scaled to work on nearly any operating system.
4. Python - This is one of the simpler languages in use, and the basics can be learned in just a couple days. It is still quite powerful, and used in a lot of server and website applications.

2. 

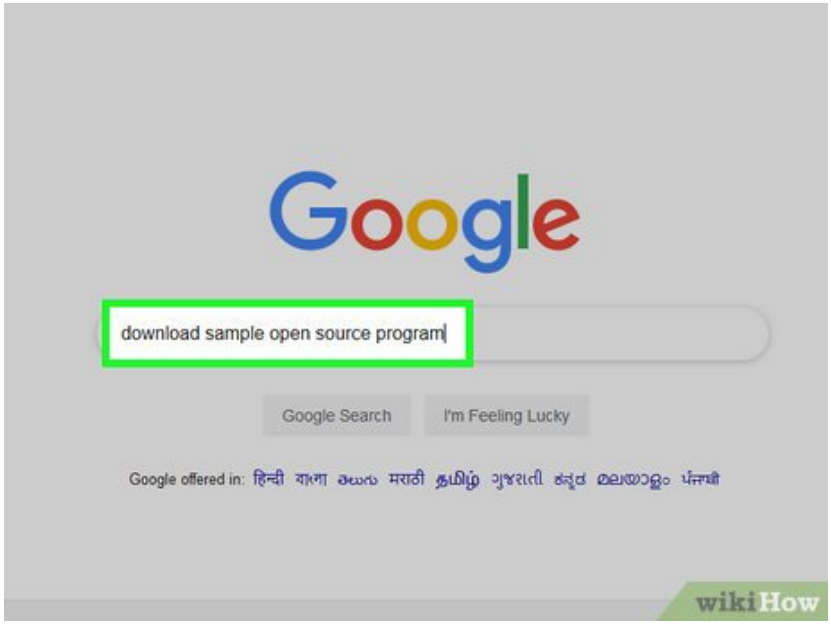
Set up a development environment. You will need a few tools in order to start writing code. These tools are referred to as your "development environment". What you will need varies depending on the language you are coding.

1. Code editor - Nearly all programmers will benefit from having a dedicated code editor installed. While you can write code using a simple text editor like Notepad, you'll find the process much easier if you have a program that highlights syntax and automates a lot of the repetitive programming tasks. Some popular code editors include Notepad++, TextMate, and JEdit.
2. Compiler or interpreter - Many languages, such as C and Java, need to be compiled before you can execute the code. You will need a compiler for your chosen language installed on your computer. Most compilers will also perform bug-reporting duties.
3. IDE (Integrated Development Environment) - Some programming languages have a code editor, compiler, and error-catcher all integrated into one program called an IDE. You can usually get this IDE from the programming language's website.

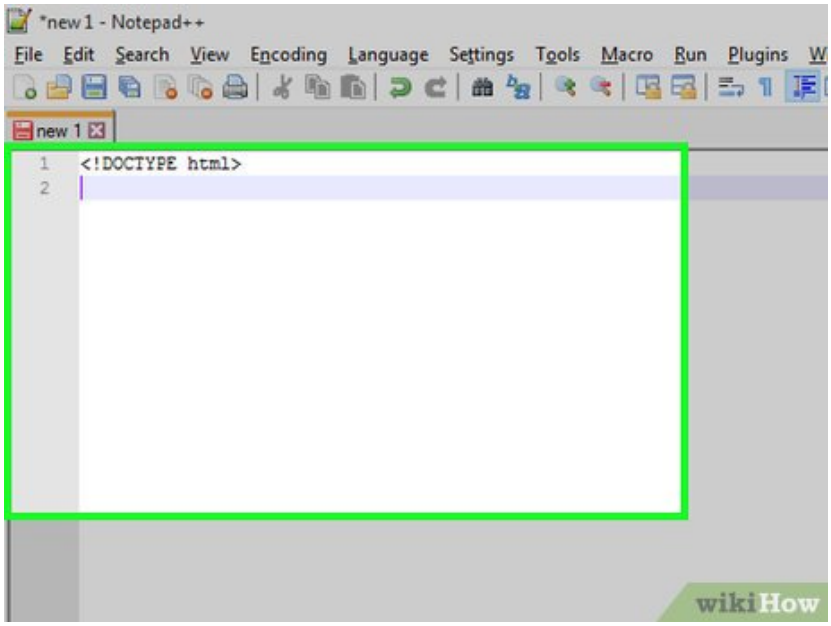
3. 

Complete some tutorials. If you've never programmed before, you're going to need to start small. Find some tutorials online that can walk you through the basic concepts of your chosen language. This could include learning about syntax, variables, functions, routines, conditional statements, and how they all fit together.

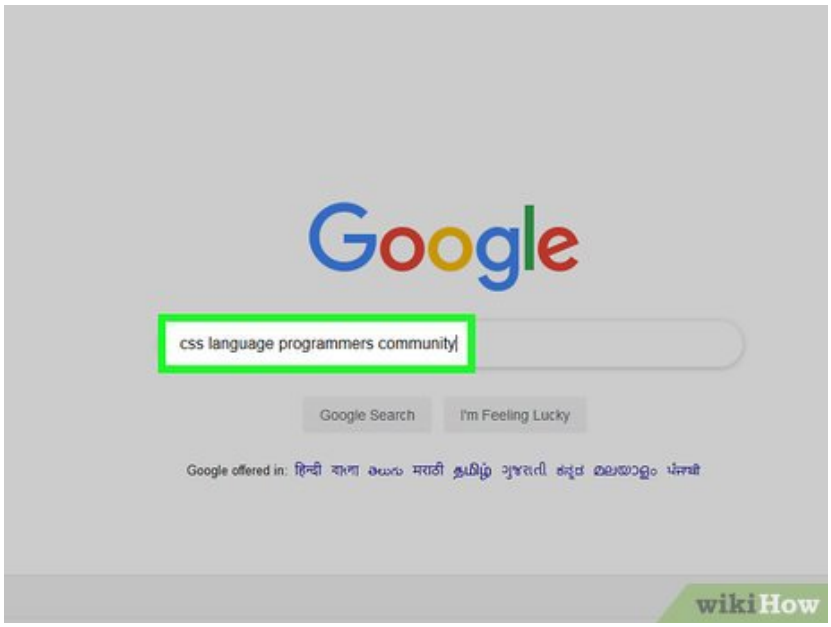
1. There are a variety of places online that provide good tutorials, including Udemy, Khan Academy, Codecademy, Code.org, and many more.[2]

4. 

Download some sample and open-source programs. Manipulating sample code can help you learn how to perform tasks using that language. There are countless sample and open-source programs available that let you access all of the code that makes the program work. Start with simple programs that are related to the kind of programs you want to make.

5. 

Create simple programs to learn the basics. When it comes time to start writing your own code, start with the basics. Write a few programs with simple inputs and outputs. Practice techniques you'll need with more complex programs, such as data handling and subroutines. Experiment and try to break your own programs.

6. 

Join coding communities. Being able to talk to a knowledgeable programmer about any issues you have is invaluable. You can find countless like-minded programmers on various sites and communities around the internet. Join a few related to your chosen language and read everything you can. Don't be afraid to ask questions, but be sure that you've tried to come up with a solution on your own first.

7. 

Understand that learning any programming language takes time. You won't be able to make a program the first time you sit down at your keyboard (not a complex program, anyway). Learning how to use the programming language effectively takes time, but with practice you'll soon be coding much faster and more efficiently.[3]

Part 2 of 7:

Designing Your Program

1. 

Write up a basic design document. Before you start coding your program, it will help to have some written material to refer to during the programming process. The design document outlines the goals of the program and describes the features in depth. This allows you to stay focused on the function the program.

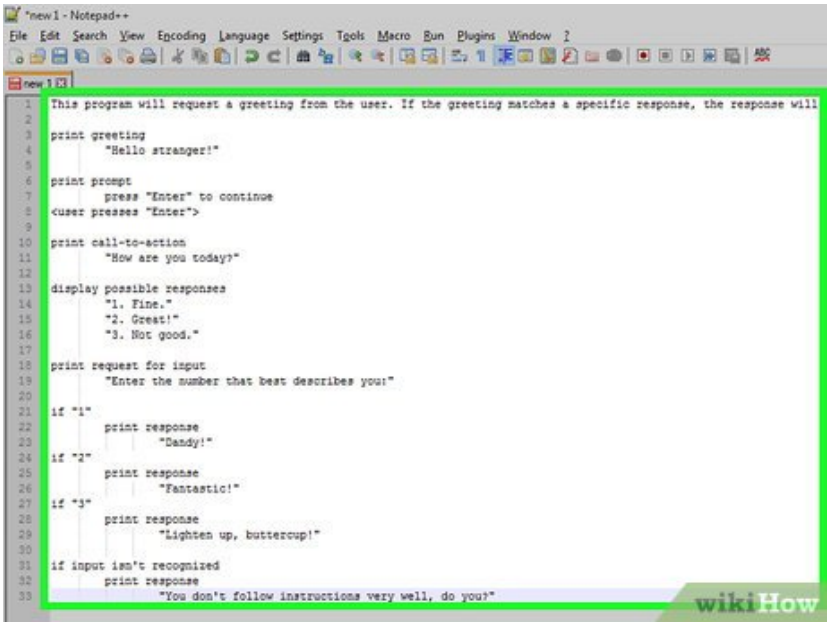
1. The design document should discuss each of the features you want to include and how they will be implemented.
2. The design document should also consider the user experience flow and how the user accomplishes his or her task using the program.

2. 

Map out the program using rough sketches. Create a map for your program, indicating how the user gets from one part to another. A simple flowchart is usually perfectly fine for a basic program.

3. 

Determine the underlying architecture of the program you are creating. The goals of the program will dictate the structure that you pick. Knowing which one of the following structures best relates to your program will help focus the development.[4]

4. 

Start with a "1-2-3" program. This is the most simple type of program, and allows you to get comfortable with your programming language. Basically, a 1-2-3 program starts, asks for input from the user, and then displays a result. After the result is displayed, the program ends.

1. The next step after a 1-2-3 is a REPL (Read-Execute-Print Loop). This is a 1-2-3 program that goes back to 1 after displaying the output.
2. Consider a Pipeline program. This is a program that transforms user input and runs continuously. This is a method good for programs that require little user interaction, such as an RSS reader. The program will be written as a series of classes that share a loop.

Part 3 of 7:

Creating a Prototype

1. 

Focus on one feature. A prototype usually focuses on the main feature of the program. For example, if you're creating a personal organizer program, your prototype may be the calendar and event-adding function.

2. 

Iterate until the feature works. Your prototype should be able to be used as its own program. It will be the foundation of everything else, so make sure its working properly. As you iterate on the feature, continue to refine it until it works smoothly and efficiently.

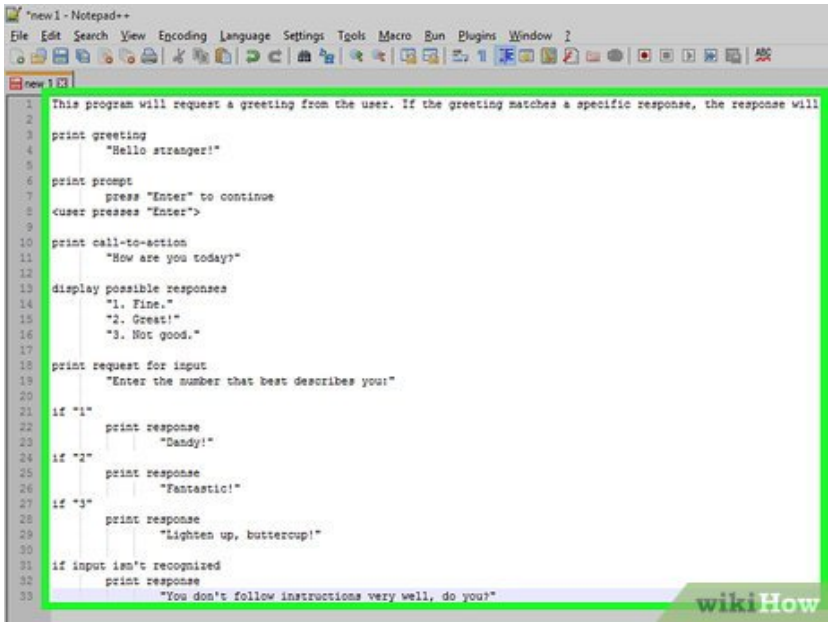
1. The prototype allows you to make rapid changes and then test them out.
2. Have others test your prototype to ensure that it functions properly.
3. Expect the prototype to change as you work on it.

3. 

Don't be afraid to scrap the prototype. The whole point of the prototype is to experiment before committing. The prototype allows you to see if the features you want are possible before you dive into coding the program proper. If the prototype is doomed to fail, scrap it and return to the drawing board. It will save you a lot of headache down the line

Part 4 of 7:

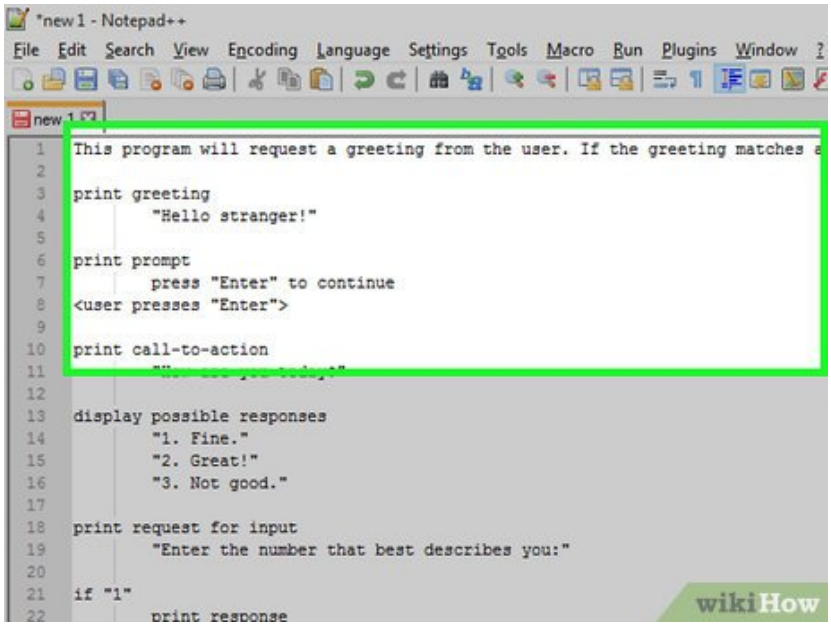
Making the Program

1. 

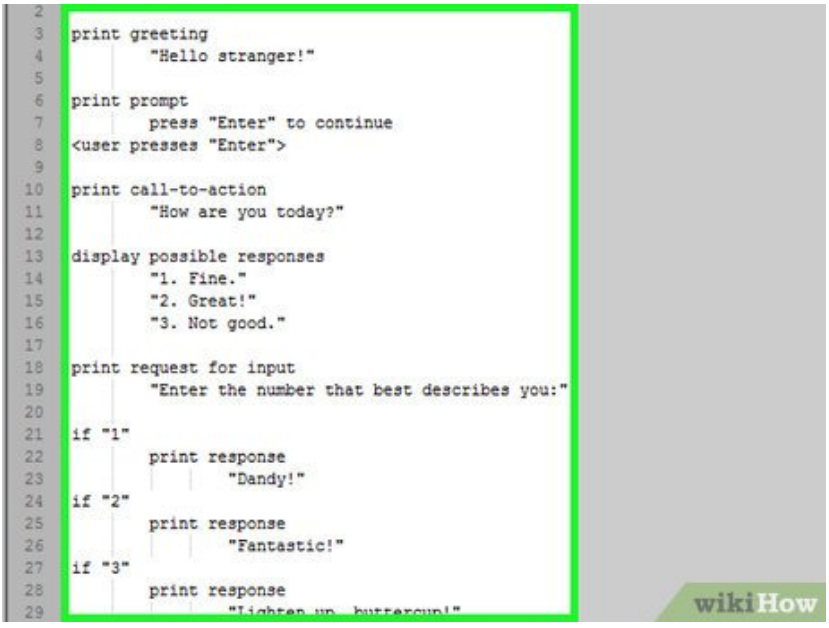
Create a pseudocode base. This is the skeleton for your project, and will serve as the base for future coding. Pseudo-code is similar to code but won't actually compile. Instead, it allows programmers to read and parse what is supposed to be happening with the code.

1. Pseudo-code still refers to the syntax of the programming language, and the pseudo-code should be structured just like regular code would be.
2. 

Expand on your prototype. You can use your existing prototype as the base for your new program, or you can adapt the prototype code into the larger structure of your full program. In either case, make good use of the time that you spent working on and refining the prototype.

3. 

Start coding. This is the real meat of the process. Coding will take the longest time, and will require numerous compiles and tests to ensure that the code works. If you are working with a team, starting from pseudo-code can help keep everyone on the same page.

4. 

Comment on all of your code. Use your programming language's comment feature to add comment to all of your code. Not only will this help anyone who works on your program figure out what the code does, but it will also help you remember what your own code does when you come back to the project later.

Part 5 of 7:

Testing the Program

1. 

Test every new feature. Every new feature added to the program should be compiled and tested. The more people that you can get to test, the more likely that you'll be able to spot errors. Your testers should be made aware that the program is far from final and that they can and will encounter serious errors.

1. This is often referred to as alpha testing.

2. 

Test your feature-complete program. Once you've implemented all of the features in your program, you should begin a round of intensive testing that covers all aspects of the program. This round of testing should also include the largest number of testers yet.

1. This is often referred to as beta testing.

3. 

Test the release candidate. As you continue to make adjustments and add assets to your program, make sure that the version you intend to release has been thoroughly tested.

Part 6 of 7:

Creating Assets

1. 

Determine what you'll need. The nature of the program will determine the assets you will need. Do you need custom sounds? Artwork? Content? All of these questions should be answered before you release your program.

2. 

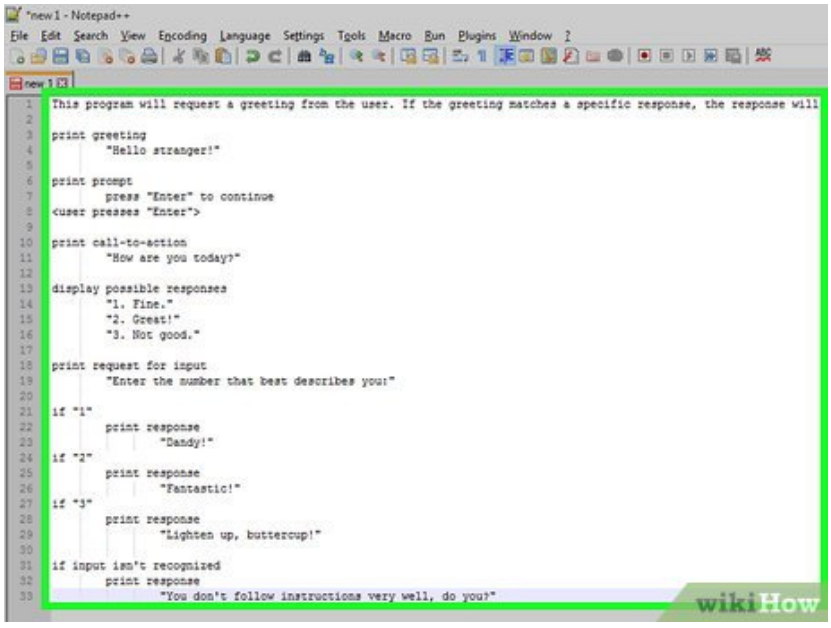
Consider outsourcing. If you need a lot of assets, but don't have the manpower or talent to create them yourself, you can consider outsourcing asset creation. There are countless freelancers out there that may be willing to work on your project.

3. 

Implement your assets. Make sure that they do not interfere with the functionality of your program, and that there is nothing superfluous. Adding assets usually occurs int eh final stages of the programming cycle, unless the assets are integral to the program itself. This is most often the case in video game programming.

Part 7 of 7:

Releasing the Program

1. 

Consider releasing your programs as open-source. This allows others to take the code you've made and improve on it. Open-source is a community-driven model of releasing, and you will likely see little profit. The benefits are that other programmers could take interest in your project and help expand the scope

significantly.

2. 

Create a storefront. If you want to sell your software, you can create a storefront on your website to allow customers to buy and download your software. Keep in mind that if you have paying customers, they will expect a working and error-free product.

1. Depending on your product, there are a variety of services that you can sell it through as well.

3. 

Keep supporting your release. After releasing your software, you will likely start receiving bug reports from new users. Categorize these bugs by their critical levels, and then start tackling them. As you update the program, you can release new versions or patches that update specific parts of the code.

1. Strong post-release support can increase your customer retention and spread good word of mouth.

4. 

Advertise your software. People will need to know your software exists before they start using it. Give out review copies to relevant review sites, consider creating a free trial version, write a press release, and do everything you can to spread the word about your software.