PREV
Lesson 1. Why should you learn how to program?

NEXT
1. Variables, types, expressions, and statements

Lesson 2. Basic principles of learning a programming language

After reading lesson 2, you'll be able to

- Understand the process of writing a computer program
- Get a big-picture view of the think-code-test-debug-repeat paradigm
- Understand how to approach a programming problem
- Understand what it means to write readable code

2.1. PROGRAMMING AS A SKILL

Like reading, counting, playing piano, or playing tennis, programming is a skill. As with any skill, you have to nurture it through lots of practice. Practice requires dedication, perseverance, and self-discipline on your part. At the beginning of your programming career, I highly recommend that you write out as much code as possible. Open your code editor and type up every piece of code that you see. Try to type it out instead of relying on copying and pasting. At this point, the goal is to make programming become second nature, not to program quickly.

This lesson serves as motivation to get you in the mindset of a programmer. The first lesson introduced you to the "Thinking like a programmer" boxes that will be scattered throughout this book. The following sections offer a big-picture view encapsulating the main ideas of those boxes.

Consider this

You want to teach a cave dweller how to get dressed to go to a job interview. Assume the clothes are already laid out and that the dweller is familiar with clothes, just not the process of dressing up. What steps do you tell him to take? Be as specific as possible.

Answer:

1. Pick up underwear, put left foot in one hole, put right foot in the other hole, and pull them up.
2. Pick up shirt, put one arm through one sleeve, and then put your other arm in the other sleeve. The buttons should be on your chest. Close shirt by inserting the little buttons into the holes.
3. Pick up pants, put one foot on one pant leg, and the other foot in the other pant leg. The pant opening should be in the front. Pull zipper up and button the pants.
4. Take one sock and put it on one foot. Then put a shoe on. Pull the laces of the shoes and tie them. Repeat with the other sock and shoe.

2.2. A PARALLEL WITH BAKING

Suppose I ask you to bake me a loaf of bread. What is the process you go through—from the time I give you the task, to when you give me the finished loaf?

2.2.1. Understand the task "bake a loaf of bread"

The first step is to make sure you understand the given task. "Bake a loaf of bread" is a bit vague. Here are some questions you may want to ask to clarify the task:

- What size loaf of bread?
- Should it be a simple bread or flavored bread? Are there any specific ingredients you have to use or not use? Are there any ingredients you don't have?
- What equipment do you need? Is the equipment supplied to you, or do

- Are there any recipes that you can look up and use, or do you have to make one up on your own?

It's important that you get these details right in order to avoid having to start over on the task. If no further details on the task are provided, the solution you come up with should be as simple as possible and should be as little work for you as possible. For example, you should look up a simple recipe instead of trying to come up with the correct combination of ingredients on your own. As another example, first try to bake a small loaf of bread, don't add any flavors or spices, and use a bread machine (if you have one) to save time.

2.2.2. Find a recipe

After you clarify any questions or misconceptions about the task, you can look up a recipe or come up with one on your own. The recipe tells you how to do the task. Coming up with a recipe on your own is the hardest part of doing the task. When you have a recipe to follow, putting everything together shouldn't be difficult.

Take a quick look at any recipe right now. Figure 2.1 shows a sample recipe. A recipe may include the following:

- The steps you should take and in what order
- Specific measurements
- Instructions on when and how many times to repeat a task
- The substitutions you can make for certain ingredients
- Any finishing touches on the dish and how to serve it

Figure 2.1. A sample recipe for bread

Quantity	Ingredient
1/4 ounce	Active dry yeast
1 tablespoon	Salt
2 tablespoons	Butter (or canola oil)
3 tablespoons	Sugar
2-1/4 cups	Warm water
6-1/2 cups	All-purpose flour

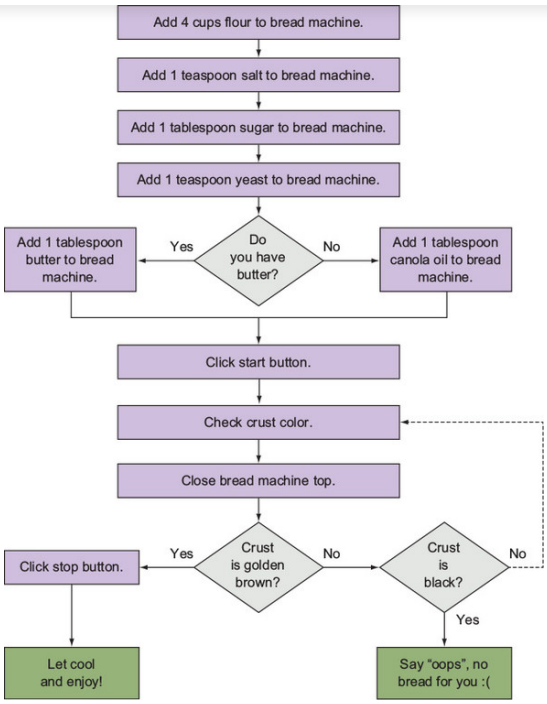
1. In a large bowl, dissolve yeast in warm water. Add the sugar, salt, oil and 3 cups flour. Beat until smooth.
2. Stir in enough remaining flour to form a soft dough.
3. Turn onto a floured surface.
4. Knead until smooth and elastic.
5. Put in a greased bowl, turning once to grease the top.
6. Cover and let rise in a warm place until doubled, about 1-1/2 hours.
7. Punch dough down. Turn onto a lightly floured surface.
8. Divide dough in half. Shape each into a loaf. Place in two greased 9x5-inch loaf pans.
9. Cover and let rise until doubled, about 30–45 minutes.
10. Bake at 375° for 30 minutes or until golden brown.
11. Move breads from pans to wire racks to cool.
12. Slice and enjoy!

The recipe is a sequence of steps that you must follow to bake bread. The steps are sequential; for example, you can't take the loaf out of the oven without first putting the dough in the pan. At certain steps, you can choose to put in one item instead of another; for example, you can put in either butter or canola oil, but not both. And some steps may be repeated, such as occasionally checking for the crust color before declaring that the bread is done.

2.2.3. Visualize the recipe with flowcharts

When you read a recipe, the sequence of steps is likely outlined in words. To prepare you for understanding how to be a programmer, you should start to think about visualizing recipes with flowcharts, as discussed briefly in lesson 1. Figure 2.2 shows how to represent baking bread with a flowchart. In this scenario, you're using a bread machine, and the ingredients differ slightly than the ones shown in figure 2.1. In the flowchart, steps are entered in rectangular boxes. If a recipe allows for a possible substitution, represent that with a diamond box. If a recipe has you repeating a task, draw an arrow going back up to the first step in the repeated sequence.

Figure 2.2. A flowchart of a simple recipe for baking bread. Rectangular boxes represent taking an action. Diamonds represent a decision point. The line going back up to a previous step represents a sequence repetition. Follow the arrows to trace paths through various implementations of the recipe.



2.2.4. Use an existing recipe or make one up?

There are many recipes for bread out there. How do you know which one to use? With a vague problem statement such as "Bake a loaf of bread," all are good to use because they all accomplish the task. In that sense, a more general problem statement is easier for you when you have a set of recipes to pick from, because any of the recipes will work.

But if you have a picky eater and are asked to bake something for which there's no recipe, you'll have a hard time accomplishing the task. You'll have to experiment with various ingredient combinations and quantities, and with various temperatures and baking times. Likely, you'll have to start over a few times.

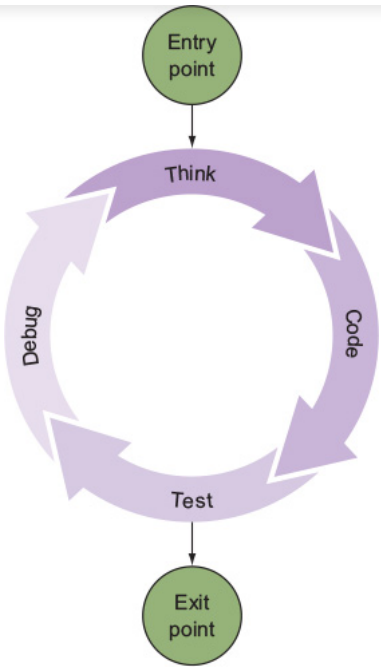
The most common type of problem you'll be given is a specific task for which you have some information, such as "Bake me a two-pound rosemary loaf of bread using 4 cups of flour, 1 tablespoon of sugar, 1 tablespoon of butter, 1 teaspoon of salt, and 1 teaspoon of yeast." You may not find a recipe that accomplishes this task exactly, but you're given a lot of critical information already in the task; in this example, you have all the ingredient measurements except for the rosemary amount. The hardest part is experimenting with ways of putting the ingredients together and determining how much rosemary to add. If this isn't your first time baking, you come to the task with some intuition for how much rosemary is just right. The more practice you have, the easier it'll be.

The main idea to come away with from this baking example is that there's more to baking than following a recipe. First you have to understand what you're being asked to bake. Then you must determine whether you have any existing recipes that you can follow. If not, you have to come up with your own recipe and experiment until you have a final product that matches what is being asked of you. In the next section, you'll see how the baking example translates into programming.

2.3. THINK, CODE, TEST, DEBUG, REPEAT

In this book, you'll write both simple and complicated programs. No matter what the complexity of the program, it's important to approach every problem in an organized and structured manner. I suggest using the think-code-test-debug-repeat paradigm shown in figure 2.3 until you're satisfied that your code works according to the problem specification.

Figure 2.3. This is the ideal way to approach solving a problem with programming. Understand the problem before you write any code. Then test the code that you write and debug it as necessary. This process repeats until your code passes all tests.



The Think step is equivalent to making sure you understand what kind of baked good you're being asked to make. Think about the problem asked of you and decide whether you have any recipes that might work or if you need to come up with one on your own. In programming, recipes are *algorithms*.

The Code step is equivalent to getting your hands dirty and experimenting with possible combinations of ingredients, any substitutions, and any repetitive parts (for example, check the crust every five minutes). In programming, you're coding up an *implementation* of an algorithm.

The Test step is equivalent to determining whether the final product matches what the task was expecting you to produce. For example, is the baked good that came out of the oven a loaf of bread? In programming, you run a program with different inputs and check whether the *actual output* matches the *expected output*.

The Debug step is equivalent to tweaking your recipe. For example, if it's too salty, reduce the amount of salt you add. In programming, you *debug* a program to figure out which lines of code are causing incorrect behavior. This is a rough process if you don't follow best practices. Some are outlined later in this lesson, and unit 7 also contains some debugging techniques.

These four steps repeat as many times as necessary until your code passes all tests.

2.3.1. Understanding the task

When you're given a problem that you need to solve using programming, you should never begin to write code right away. If you start by writing code, you enter the cycle directly at the Code leg shown in figure 2.3. It's unlikely that you'll write your code correctly the first time. You'll have to cycle through until you think about the given problem, because you didn't correctly solve it the first time. By thinking about the problem at the start, you minimize the number of times you'll go through the programming cycle.

As you tackle harder and harder problems, it's also important that you try to break them into smaller problems with a simpler and smaller set of steps. You can focus on solving the smaller problems first. For example, instead of baking a loaf of bread with exotic ingredients, try a few small rolls to get the proportions just right without wasting too many resources or too much time.

When you're given a problem, you should ask yourself the following:

- What is this program supposed to accomplish? For example, "Find the area of a circle."
- Are there any interactions with the user? For example, "The user will enter a number" and "You show the user the area of a circle with that radius."
- What type of input is the user giving you? For example, "The user will give you a number representing the radius of a circle."
- What does the user want from the program and in what form? For example, you might show the user "12.57," you might be more verbose and show "The area of a circle with radius 2 is 12.57," or you might draw a picture for the user.

I suggest that you organize your thoughts on the problem by redescribing the problem in two ways:

Quick check 2.1

Q1:

Find any recipe (in a box or look one up on the internet). Write a problem statement for what the recipe is trying to achieve. Write a vague problem statement. Write a more specific problem statement.

2.3.2. Visualizing the task

When you're given a task to solve using programming, think of the task as a black box. At first, don't worry about the *implementation*.

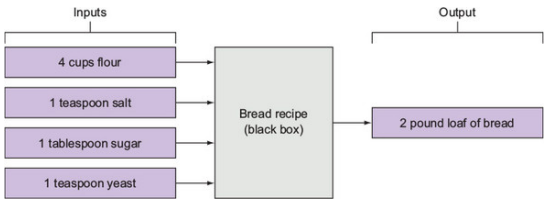
Definition

An implementation is the way you write the code to solve a task.

Instead of worrying about the details of the implementation, think about what's being asked: Are there any interactions with the user? Is there any input your program might need? Is there any output your program might show? Is your program just supposed to be doing calculations behind the scenes?

It's helpful to draw a diagram showing possible interactions between your program and the user of the program. Go back to the bread example. A possible black box visualization is shown in figure 2.4. The inputs are represented to the left of the black box and the outputs to the right.

Figure 2.4. A black box visualization of baking a loaf of bread with a given set of ingredients.



When you have an idea of the inputs and outputs of your black box, think about any special behaviors that you may need to take into account. Will the program behave differently in different situations? In the bread example, can you substitute something for sugar if you don't have any? Will you get a different type of bread if you add sugar instead of salt? You should write out what the program will do in these situations.

All these specific interactions can be visualized in a flowchart. You can trace many routes through your flowchart, each route representing a different possible implementation and outcome, as in figure 2.2.

Quick check 2.2

Q1:

You need to clean up after your baking adventure. You need to do two things: wash the dishes and take out the trash, in that order. Organize the following steps and decision points into a flowchart as in figure 2.2. Use as many steps/decisions as possible, but you don't have to use them all.

- Step: Rinse dish
- Step: Sing a song
- Step: Tie up trash bag
- Step: Take trash outside
- Step: Pick up a dirty dish
- Step: Scrub dirty dish with soap
- Step: Put clean dish in drying rack
- Step: Dump out trash bag on the floor
- Step: Put a piece of trash in the trash bag
- Decision: Anything else to put in the trash bag?
- Decision: Am I happy with my baking skills?
- Decision: Any more dirty dishes left?
- Decision: Should I watch a movie tonight?

2.3.3. Writing pseudocode

At this point, you've come up with test cases, special behaviors you might have to be careful of, and a visual representation of a sequence of steps that you believe will accomplish the task given. If you drew out your sequence of steps, now is the time to convert your drawing into words, using programming concepts. To solve the problem, you must come up with a sequence of steps to follow so that they achieve the task outlined in the problem.

Pseudocode is a mix of English and programming on paper or in your code editor. It helps you get the structure of the program looking right at various points: when you get input from the user, when you show output, when you need to make a decision, and when you need to repeat a set of steps.

Putting the sequence of steps into words is like writing down and trying out your recipe. You must use what you know about how ingredients taste and what they're used for to decide the best way to arrange them together. In programming, you must use everything you know about various techniques and constructs to put the code together, and this is the hardest part of programming.

In pseudocode, finding the area of a circle might look like this:

1. Get a radius from the user.
2. Apply a formula.
3. Show the result.
4. Repeat steps 1–3 until the user says to stop.

Throughout this book, you'll see examples where certain programming concepts are useful. The only way to know which concept to use and when to use it is through intuition, which comes with a lot of practice.

Of course, there are many ways to achieve a task with programming. It's not a bad thing to go down one path and find yourself stuck; then you'll have a better understanding of why a particular method doesn't work in that case. With time and experience, you'll develop intuition for when one concept is better to use than another.

Quick check 2.3

Q1:

Here's a problem statement. The Pythagorean theorem is $a^2 + b^2 = c^2$. Solve for c. Write a sequence of steps, in pseudocode, that you might take to solve for c.

Hint: $\sqrt{(x^2)} = x$

2.4. WRITING READABLE CODE

As you learn more about programming, and specifically Python programming in this book, you'll see language specifics that Python offers to help you achieve this principle. I don't discuss those in this lesson. What's important to remember at this point, before you even start writing code, is that any code you write should be with the intent that someone else will read it, including yourself in a few weeks' time!

2.4.1. Using descriptive and meaningful names

Here's a short snippet of code in Python, which you don't need to understand right now. It consists of three lines of code, evaluated in order, top to bottom. Notice that it looks similar to something you may write in a math class:

```
a = 3.1
b = 2.2
c = a * b * b
```

Can you tell, at a high level, what the code is supposed to calculate? Not really. Suppose I rewrite the code:

```
pi = 3.1
radius = 2.2
# use the formula to calculate the area of a circle
circle_area = pi * radius * radius
```

Now can you tell, at a high level, what the code is supposed to do? Yes! It calculates—or rather, estimates—the area of a circle with radius 2.2. As in math, programming languages use *variables* to store data. A key idea behind writing readable code when programming is using descriptive and meaningful variable names. In the preceding code, `pi` is a variable name, and you can use it to refer to the value 3.1. Similarly, `radius` and `circle_area` are variable names.

Also notice that the preceding code includes a line that starts with the # character. That line is called a *comment*. In Python, a comment starts with the # character, but in other languages, it can start with different special characters. A comment line isn't part of the code that runs when the program runs. Instead, comments are used in code to describe important parts of the code.

Definition

A comment is a line in a Python program that starts with a #. These lines are ignored by Python when running a program.

Comments should help others, and yourself, understand why you wrote code in that way. They shouldn't just put into words what the code implements. A comment that says "Use the formula to calculate the area of a circle" is much better than one that says "Multiply pi times the radius times the radius." Notice that the former explains why the code is correct to use, but the latter simply puts into words what the code is implementing. In this example, someone else reading the code already knows that you're multiplying the three values (because they know how to read code!), but they might not know why you're doing the multiplication.

Comments are useful when they describe the rationale behind a larger chunk of code, particularly when you come up with a unique way to compute or implement something. A comment should describe the big idea behind the implementation of that particular chunk of code, because it might not be obvious to others. When someone reading the code understands the big picture, they can then go into the specifics of the code by reading each line to see exactly what calculations you're doing.

Quick check 2.4

Q1:

Here's a short piece of Python code implementing a solution to the following problem. Fill in the comments. "You're filling a pool and have two hoses. The green hose fills it in 1.5 hours, and the blue hose fills it in 1.2 hours. You want to speed up the process by using both hoses. How long will it take using both hoses, in minutes?"

```
# Your comment here
time_green = 1.5
time_blue = 1.2

# Your comment here
minutes_green = 60 * time_green
minutes_blue = 60 * time_blue

# Your comment here
rate_hose_green = 1 / minutes_green
rate_hose_blue = 1 / minutes_blue

# Your comment here
rate_host_combined = rate_hose_green + rate_hose_b

# Your comment here
time = 1 / rate_host_combined
```

SUMMARY

In this lesson, my objective was to teach you

- The think-code-test-debug-repeat cycle of events that a good programmer should follow.
- To think about the problem you're given and understand what's being asked.
- To draw out what the inputs and outputs are based on the problem description before beginning to code.
- That a problem statement won't necessarily outline the series of steps you should take to solve the task. It may be up to you to come up with a recipe—a series of steps to achieve the task.
- To write code with the intent of it being read. You should use descriptive and meaningful names, and write comments that describe in words the problem and coded solution.

PREV
Lesson 1. Why should you learn how to program?

NEXT
Unit 1. Variables, types, expressions, and statements