Lesson 20. Building programs to last

After reading lesson 20, you'll be able to

- Understand how a bigger task is divided into modules
- Understand why you should hide away details of complicated tasks
- Understand what it means for tasks to be dependent on or independent of other tasks

You saw how useful loops are at getting the computer to repeat a certain group of statements many times. As you're writing code, it's important to be aware of how you can harness the power of computers to make life easier for you. In this lesson, you'll take this idea a step further to see how to divide a larger program into smaller mini-programs, each one constructed to achieve a specific task.

For example, if you think about the process of building a car as a large program, you'd never build one machine that builds the entire car. That would be one extremely complicated machine. Instead, you'd build various machines and robots that focus on doing different and specific tasks: one machine might assemble the frame, one might paint the frame, and another might program the on-board computer.

Consider this

You're getting married! You don't have time to take care of everything on your own, so you want to hire people to take care of various tasks. Write some tasks that you can outsource.

Answer: Find and book venue, decide on catering (food, bar, cake), finalize guest list (invite people, keep track of attendees, seating), decorate, hire officiant, and dress up the wedding party.

20.1. BREAKING A BIG TASK INTO SMALLER TASKS

The main idea behind taking one task and breaking it into smaller tasks is to help you write programs more effectively. If you start with a smaller problem, you can debug it quicker. If you know that a few smaller problems work as expected, you can focus on making sure they work well together as opposed to trying to debug a large and complex one all at once.

20.1.1. Ordering an item online

Think about what happens when you order an item online. You start by putting your personal information on a website order form, and you end with getting the item delivered to your house. This entire process can be broken into a few steps, as you can see in figure 20.1:

1. You fill in a web form to place the order. The order information goes to the seller, who extracts the important details: what item, how many, and your name/address.
2. Using the item type and number, the seller (a person or a robot) finds the item in a warehouse and gives it to the packer.
3. The packer takes the item(s) and puts them in a box.
4. Using your name/address, someone else makes a shipping label.
5. The box is matched with a label, and the package is sent to the post office, which takes care of finding your house and delivering the package.

Figure 20.1. One possible way to divide the task of ordering an item online into smaller, self-contained, and reusable subtasks. Each gray box represents a task. Things to the left of the box are the inputs to a task, and things to the right are outputs of the task.

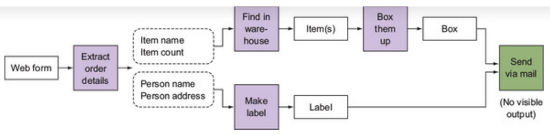


Figure 20.1 shows how to divide the big task of ordering an item into five other subtasks. Each subtask might be handled by separate people or machines and represent different specialties in the process of ordering an item online.

This example also illustrates a few other important ideas. The first idea is *task dependence/independence*.

Definition

A task depends on another one if it can't start before the other one completes. Two tasks are independent if they can be performed at the same time.

Some tasks depend on the completion of others, whereas some tasks are completely independent. You first do the task "extract order details." You use its output to do the "find item in warehouse" and the "make label" tasks. Notice that these last two tasks are independent of each other and can be done in any order. The task "box them up" depends on the task "find in warehouse." The "send via mail" task depends on both "box them up" and "make label" tasks to be finished before it can begin.

Quick check 20.1

Are the following actions dependent or independent?

1

(1) Eating pie and (2) writing 3.1415927 on a piece of paper.

2

(1) You don't have an internet connection and (2) you can't check your email.

3

(1) It's January 1 and (2) it's sunny.

Definition

Abstraction of a task is a way to simplify the task such that you understand it by using the least amount of information; you hide all unnecessary details.

To understand what happens when you order an item online, you don't need to understand every detail behind the scenes. This brings us to the second idea: abstraction. In the warehouse example, you don't need to know the details of how to find an item in a warehouse; whether the seller employs a person to get your item or whether they use a sophisticated robot doesn't matter to you. You need to know only that you supply it an "item name" and an "item count" and that you get back the items requested.

Broadly speaking, to understand a task, you need to know only what input a task needs before starting (for example, personal information on a form) and what the task will do (for example, items show up at your door). You don't need to know the details of each step in the task to understand what it does.

Quick check 20.2

For each of the following, what are possible inputs and outputs (if any)? Ask what items you need in order to perform each action and what items you get out of doing the action:

1

Writing a wedding invitation

Making a phone call

3

Flipping a coin

4

Buying a dress

The third idea is of *reusable subtasks*.

Definition

Reusable subtasks are tasks whose steps can be reused with different inputs to produce different output.

Sometimes you want to do a task that's slightly different from another one. In the warehouse example, you might want to find a book in the warehouse or you might want to find a bicycle. It wouldn't make sense to have a separate robot for every item that you might want to retrieve. That would lead to too many robots that kind of do the same thing! It's better to make one robot that can find any item you want. Or to make two robots: one that can retrieve big items and one for small items. This trade-off between creating subtasks while making the subtasks generic enough to be reusable can be subjective. With a little bit of practice in the next few lessons, you'll get the hang of striking a good balance.

Quick check 20.3

Q1:

Divide the following task into smaller subtasks: "Research the history of crayons, write a five-page paper, and give a presentation." Draw diagrams similar to figure 20.1.

20.1.2. Understanding the main points

When you deal with tasks, consider each one a *black box*.

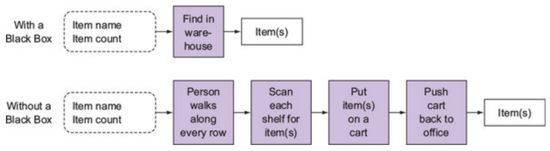
Definition

A black box is a way to visualize a system that does a certain task. A black box on top of the system reminds you that you don't get to (or need to) see inside the box in order to understand what the system does.

Right now, you don't need to know how that task is accomplished; you're only trying to visualize your overall system in terms of these smaller tasks without getting bogged down in their details.

Take the task "find in warehouse" from figure 20.1 and look at figure 20.2 to see one way the task could look under the black box. Without a black box, you get more details on how the task is implemented—what steps and actions are done using the inputs. But these details don't help you understand the task itself; the details of the task implementation aren't important or necessary to understand what the task does. In some situations, seeing these details might even create more confusion. Ultimately, the inputs and outputs to the overall system are the same with and without the black box over the system.

Figure 20.2. "Find in warehouse" shown with and without a black box over the task. Seeing the details of how the item is found and retrieved in the warehouse doesn't add any more understanding of the task itself.



Each task is a sequence of actions or steps. These steps should be generic enough that they can be repeated for any appropriate inputs. How do you determine what's an appropriate input? You need to document your black

20.2. INTRODUCING BLACK BOXES OF CODE IN PROGRAMMING

The programs you've seen so far have been simple enough that the entire program is a black box. The tasks you've been given aren't complex enough to warrant having specialized pieces of code to do different tasks; your entire programs have been pieces of code to each do one task.

Your programs so far have mostly done the following: (1) ask the user for input, (2) do some operations, and (3) show some output. From now on, you'll find it helpful and necessary to divide the program into smaller and more manageable pieces. Each piece will solve part of the puzzle. You can put all the pieces together to implement a larger program.

In programming, these tasks are considered black boxes of code. You don't need to know how each block of code works. You only need to know what inputs go into the box, what the box is supposed to do, and what output the box gives you. You're abstracting the programming task to these three pieces of information. Each black box becomes a *module* of code.

Definition

A code module is a piece of code that achieves a certain task. A module is associated with input, a task, and output.

20.2.1. Using code modules

Modularity is the division of a big program into smaller tasks. You write code for each task separately, independent of other tasks. In general, each code module is supposed to stand on its own. You should be able to quickly test whether the code that you wrote for this module works. Dividing a larger task in this way makes the larger problem seem easier and will reduce the time it takes you to debug.

20.2.2. Abstracting code

You likely watch TV and use a remote to change the channel. If I gave you all the parts necessary to build a TV and a remote, would you know how to put them together? Probably not. But if I assembled the TV and the remote for you, would you know how to use the two to achieve a task such as changing the channel? Probably. This is because you know the inputs of each item, what each item is supposed to do, and what each item outputs. Figure 20.3 and table 20.1 show inputs, behavior, and output for the process of using a remote with a TV.

Figure 20.3. Black box view of a remote and a TV

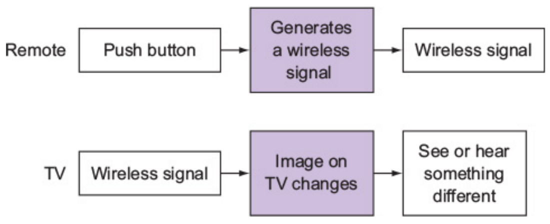


Table 20.1. Input, behavior, and output of a TV and remote for changing the channel or the volume

Item	Input	Behavior	Output
Remote	Push a button	Generates a signal depending on the button pressed	A wireless signal
TV	A wireless signal from a remote	Image on the screen changes (whole image or a volume bar appears) or volume changes	What you see or hear changes

In programming, abstraction aims to present ideas at a *high level*. It's the process of documenting what a piece of code does, with three key bits of information: inputs, the task, and outputs. You've seen black boxes represented using this information.

Abstraction in code eliminates the details of how the code for a task/module is implemented; instead of looking at the code for a module, you look at its documentation. To document the module, you use a special type of code comment, called a *docstring*. A docstring contains the following information:

- *All inputs to the module*—Represented by variables and their types.
- *What the module is supposed to do*—Its function.
- *What output the module gives you*—This might be an object (variable) or it might be something that the module prints.

20.2.3. Reusing code

Suppose someone gives you two numbers, and you want to be able to do four operations on the numbers: add, subtract, multiply, and divide. The code might look like the following listing.

Listing 20.1. Code to add, subtract, multiply, and divide two numbers

```
a = 1                      1
b = 2                      1
print(a+b)
print(a-b)
print(a*b)
print(a/b)
```

- *1* **Variables a and b are used to do a + b, a - b, a * b, and a / b.**

In addition to this code, you also want to add, subtract, multiply, and divide a different pair of numbers. Then yet another pair of numbers. To write a program that does the same four operations on many pairs of numbers, you'd have to copy and paste the code in listing 20.1 and change the values of `a` and `b` a bunch of times. That sounds tedious, and looks ugly, as you can see in the following listing! Notice that the code to do the operations themselves is the same no matter what variables `a` and `b` are.

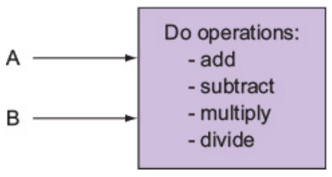
Listing 20.2. Code to add, subtract, multiply, and divide for three pairs of numbers

```
a = 1                 1
b = 2                 1
print(a+b)            1
print(a-b)            1
print(a*b)            1
print(a/b)            1
a = 3                 2
b = 4                 2
print(a+b)            2
print(a-b)            2
print(a*b)            2
print(a/b)            2
a = 5                 3
b = 6                 3
print(a+b)            3
print(a-b)            3
print(a*b)            3
print(a/b)            3
```

- *1* **Code to do operations on a = 1 and b = 2**
- *2* **Code to do operations on a = 3 and b = 4**
- *3* **Code to do operations on a = 5 and b = 6**

This is where the idea of reusability comes into play. The part where you do operations and print the results of the operations is common across any pairs of numbers `a` and `b`. It doesn't make sense to copy and paste it every time. Instead, think of this common set of operations as a black box; the inputs to this black box change (as does the output). Figure 20.4 shows a black-box view of a task that can do four simple mathematical operations on any two numbers, `a` and `b`, where `a` and `b` are now inputs to the black box.

Figure 20.4. Black-box view of code that adds, subtracts, multiplies, and divides any two numbers



Now, instead of copying and pasting code in a program and changing a small part of it, you can write a black box around the code, which is reusable. The box is a *code wrapper* that adds a piece of functionality to the program. You can write programs that are more complex by reusing wrappers you already wrote. The code in listing 20.2 can be abstracted away using the black-box concept to become something like the following. Variables `a` and `b` still change, but now you're using code wrapped up in a black box. The four lines of code to do the four mathematical operations are simplified as one bundle under a black box.

Listing 20.3. Code to add, subtract, multiply, and divide for three pairs of numbers

```
a = 1
b = 2
< wrapper for operations_with_a_and_b >                 1
```

```
a = 5
b = 6
< wrapper for operations_with_a_and_b >             1
```

- **1 (Not actual code) placeholder for a black box that does four operations**

In the next lesson, you'll see the details on how to write the wrappers for the black boxes around code. You'll also see how to use these wrappers in your program. These wrappers are called functions.

20.3. SUBTASKS EXIST IN THEIR OWN ENVIRONMENTS

Think about doing a group project with two other people; you must research the history of telephones and give a presentation. You're the leader. Your job is to assign tasks to the other two people and to give the final presentation. As the leader, you don't have to do any research. Instead, you call upon the two other group members to do research, and they relay their results to you.

The other two people are like smaller worker modules helping you with the project. They're in charge of doing the research, coming up with results, and giving you a summary of their findings. This demonstrates the idea of *dividing a larger task into subtasks*.

Notice that you, as the leader, aren't concerned with the details of their research. You don't care whether they use the internet, go to the library, or interview a random group of people. You just want them to tell you their findings. The summary they give you demonstrates the idea of *abstraction of details*.

Each person doing the research might use an item that has the same name. One might read a children's picture book named *Telephone* and one might read a reference book named *Telephone*. Unless these two people pass a book to each other or communicate with each other, they have no idea what information the other is gathering. Each researcher is in their own environment, and any information they gather stays only with them—unless they share it. You can think of a code module as a mini-program to achieve a certain task. Each module exists in its own environment, independent from the environment of other modules. Any item created inside the module is specific to the module, unless explicitly passed on to another module. Modules can pass items through output and input. You'll see many examples of how this looks in code in the next lesson.

In the group project example, the group project is like the main program. Each person is like a separate module, each in charge of doing a task. Some tasks may communicate with each other, and some may not. For larger group projects, some people in the group might not need to share information with others if they're in charge of independent pieces. But all group members communicate with the leader to relay information gathered.

Each person does the research in a separate environment. They might use different objects or methods to do the research, each being useful only in the environment of that one person. The leader doesn't need to know the details of how the research takes place.

Quick check 20.4

Q1:

Draw a black-box system for the task of researching the telephone in a group project setting described in this section. Draw a black box for each person and indicate what each person may take as input and may output.

SUMMARY

In this lesson, my objective was to teach you why it's important to view tasks as black boxes and, ultimately, as code modules. You saw that different modules can work together to pass information to each other to achieve a larger goal. Each module lives in its own environment, and any information it creates is private to that module, unless explicitly passed around through outputs. In the bigger picture, you don't need to know the details of how modules accomplish their specific tasks. Here are the major takeaways:

- Modules are independent and in their own self-contained environments.
- Code modules should be written only once and be reusable with different inputs.
- Abstracting away module details allows you to focus on the way many modules work together to accomplish a larger task.

Let's see if you got this...

Divide the following task into smaller subtasks: “A couple orders at a restaurant and gets drinks and food.” Draw a diagram.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Unit 5. Organizing your code into reusable blocks

NEXT
Lesson 21. Achieving modularity and abstraction with functi…