Queen Mary University of London

Box Variables (In Python): Understanding Variables and Assignment

Paul Curzon and Nicola Plant
Queen Mary University of London

Variables and assignment are important early concepts to understand when learning to program. Without a good understanding of them you will struggle to move on and master more advanced concepts. They are easy to misunderstand, however. If you realise it is a problem it is also easy to fix. It boils down to realising variables are like storage boxes with built in photocopiers and shredders.

Whether it is to fly a plane or play a game, one of the most important things computers do is process data. They store values – numbers, text, images, sounds – manipulate them, do calculations with them, make changes to them and store the results until they are needed. For example, music editing software might be used to record samples of sounds, edit them, combine them with others, store the results and then play the final version. A texting or tweeting program stores the words you type, allowing you to edit them until you are ready to send the message.

For a computer program to process data it must have a way to store it, and that is where **variables** come in. They are just storage places. To process information you also need a way to move it from one place to another. That is what **assignment** is for. To be able to program you must understand both, and it turns out it is easy to get confused about the details of how they work without realising you don't understand. Get over this hurdle, understand it well, and programming becomes far easier.

So what is a variable? A good way to think of variables is that they are just like storage boxes. You can store information there and later retrieve it. The twist is that they are special storage boxes. They are storage boxes with their own private shredder and photocopier built in. Variables are boxes that can store, but also create and destroy, information.

Variables are boxes that can store, but also create and destroy

Let's look at a fragment of a program to see how it all works. Don't worry if you don't understand it yet. We will go through each part in turn in the next sections.

```
colour1 = “red”
colour2 = “green”

temp = colour1
colour1 = colour2
colour2 = temp
```

This program has three variables – three different places to store data – called **colour1**, **colour2** and **temp**.

Creating variables: finding a new box

To use variables as storage in a program we first need to create the variables in the first place. Different languages do this in different ways. In many languages you must write instructions in your programs to explicitly tell the computer to create a variable before it is needed. In Python they are created the first time you mention a new one i.e., when you use its name for the first time. Mentioning a new variable has the effect of giving the computer an instruction to go and find a storage box that isn't being used for anything else. When a program creates a variable, it is picking a new place in its memory to store data. Each variable you create can hold a single value. Here is an example from the first line of our program.

Creating a variable creates a new storage box

```
colour1 = “red”
```

colour1 hasn't been mentioned before this point in the program so the computer takes that as an instruction to create a variable with that name. This command therefore first of all creates a new storage box called **colour1.**

colour1

That is just the start though. The command then goes on to say put something in the storage box: in this case the word **“red”**. To a computer a word like **“red”** is a just a series of characters to be stored one after another – it is called a **string** of characters**.** We get something like this with **“red”** stored in the box called **colour1**:

colour1

Once a variable is given a name, that name cannot then be changed. For as long as the box exists that will be its name. Let's look in more detail at the names and the things they can hold in turn.

Identifiers: the names of boxes

The name we have given the first variable in our program is **colour1**. That is its **identifier** (the word programmers use for the names of things). Each variable is given its own identifier – each box has a name – so that we can use it to identify a particular variable later in the program when we want to put some information in it or get some information out again. Most programs use lots of variables as they need to store lots of different information. All those variables need their own identifiers so the computer knows which one we mean. Ideally the identifiers we use should tell us something about what is stored in them to help us remember which is which. In this case its name suggests it will be used to represent a colour of something. We could have used just about any name we liked though. Good programmers are good at choosing names that help people understand what their variables will be used for.

A variable identifier is its name. It identifies which storage box you mean

Values: the things put in the boxes

The things that are stored in variables are called **values**. They are the things that are put into boxes. They are the actual pieces of information that are manipulated by a program. When we put a value in a variable for the first time we are **initialising** it. Our example instruction

The information that is stored in a variable is called its value: the current contents of the box

colour1 = "red"

not only creates a new variable called colour1. It also stores the value **"red"** in it. It initialises it with that value. As we will see later instructions can change the value in a variable and we can copy values stored in one to another just like we can with storage boxes.

Back to our program

Our program created three variables. They are created in the order they appear in the program, because that is the order that instructions are executed. The first was the one we saw above

colour1 = "red"

Which created our first storage box.

colour1

The program then created a second variable in the next instruction:

colour2 = "green"

It creates a second storage box to go with the first this time called **colour2** and puts the string value **"green"** in it.

colour2

We will look at the third variable later.

So after executing the first part of the program:

colour1 = "red"
colour2 = "green"

we have created two variables, (i.e., two storage boxes) that each stores a single string value.

The state of our program at this point in its execution looks something like that below. The program has gathered two boxes together and stored some initial values in them, ready for the rest of the program to manipulate. We have also given each box a name so that we can tell them apart.

colour1

colour2

Assignment: putting a value in a box

Lets look at the following line of code in more detail**.**

colour1 = "red"

It stores the string **"red"** in the variable called **colour1**. It puts **"red"** in the box. In Python the symbol = is used to indicate that the command is an **assignment**. Do not read it as the word 'equals' though as if you do you will probably get confused. It is nothing to do with the maths symbol we use to mean two things are equal. It is a command telling the program to do something. Read it as 'gets a copy of the value'. The above command therefore means:

Variable colour1 ***gets a copy of the value*** **"red"**

It is a command telling the program to store the string value **"red"** into the storage box that we called **colour1**.

Assignment is a command saying that a variable gets a copy of a value

The first part of an assignment command before the = symbol is always a storage space: it is the identifier of a variable. After the equals symbol there is always something that will give us a value – the value that we want to be stored in that place. It may be helpful to mentally write an arrow over the top of the = symbol to help remember which way the data moves (from right to left).

colour1 = "red"

Our variable box now contains a copy of the value "red".

colour1

The next instruction in our program is another assignment

colour2 = "green"

It is similar but this time it puts the value **"green"** into variable **colour2.** Our variable box **colour2** now holds a copy of the value **"green"** :

colour2

So at this point, we have two variables. **colour1** holds the value **"red"** and **colour2** holds the value **"green"**.

To summarise, when we create a variable we assign it a value. We call an assignment like this, where the ***first value*** is stored in the variable, ***initialising*** that variable. In the above the assignments ***initialised*** the variable **colour1** with the value **"red"** and **colour2** with the value **"green"**.

It is really important here to recognise the difference between the ***values stored in variables*** and the ***names or identifiers of variables***. Values are the things that can be stored in boxes – that we have written inside the box – **"red"** and **"green"** are the values in our example. Identifiers are just labels stuck on the box so we know which is which. They are NOT data. In Python string values have quotes round them so you know they are values not names of variables. If there are no quotes round a word in a program then it will either be the name of a variable or a keyword (a word with a special meaning in the language like **int**).

Accessing a variable: getting a value out of a box

We have initialised our variables with starting values. Now we can do something with the data we have stored there. In our simple little program, we will just swap the two values in **colour1** and **colour2** over, so that at the end, **colour1** holds value **"green"** and **colour2** holds **"red".** We will need an extra variable **temp** as a temporary storage space to hold on to values we don't want to lose while we swap the values around. In doing so we will see why it is important to think of variables not just as storage boxes but as boxes that come with an integrated copier and a shredder.

The next instruction in our program is

```
temp = colour1
```

As **temp** does not exist, executing this instruction creates a new variable – a new storage box – called **temp**. This is just another assignment but it is a little different to the previous ones we've looked at. Both sides are just the name of a variable (no quotes used in either case so neither are values). When we give the name of a variable before the = in an assignment we are giving a location to store a value. When we give the name of a variable after the = we mean get a value from that variable. This assignment says get the value from **colour1** and put it in **temp**. Drawing in the arrow can help to remind us which way the information moves as now there is no clue from having a value on one side.

We get a copy of the value in a variable by giving its name

```
temp = colour1
```

However, there is a subtlety about what is happening here. It isn't quite like taking a value out of one physical box and putting it in another physical box. Information isn't like physical objects. This assignment doesn't mean we

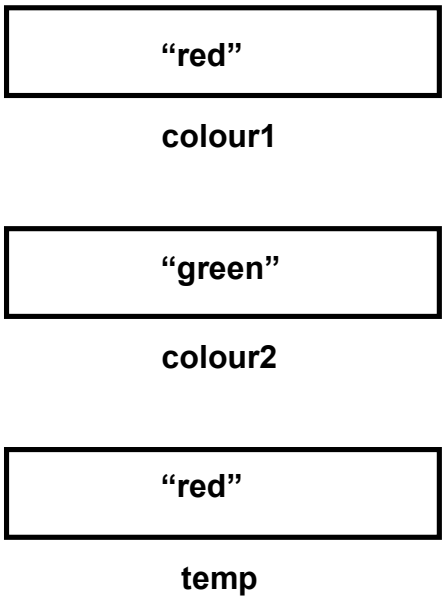
actually do take the value out of the original variable. Our boxes have integrated photocopiers and when we want their value we get a **copy** of it, leaving the original where it is, safely stored in its original box. It is that copy that is put in the other box.

The name of a variable before the = refers to a place to put a new value. A name placed on the other side says to get a ***copy*** of what is in that variable (and put it in the other variable).

In our example the assignment

temp = colour1

means get a copy of what is in **colour1**, and put the copy in the variable called **temp**. **colour1** itself doesn't change at all. It still has the same value after this command has been executed as it held before – in this case **"red"**. An assignment only changes the variable named before the = sign. After this assignment command the new state of our boxes looks like this where **temp** now has the value **"red"** copied from **colour1**:



Overwriting the old value of a variable

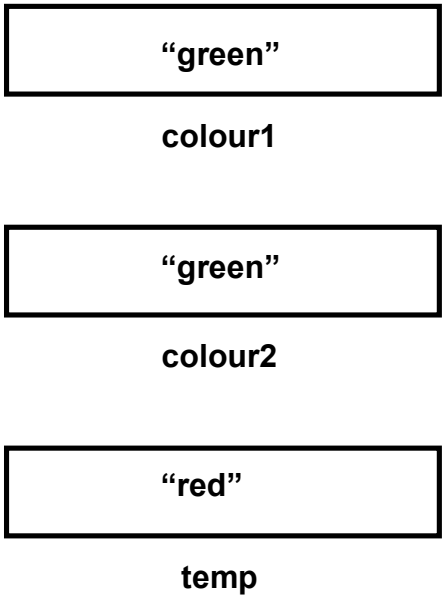
Our program's next instruction is

colour1 = colour2

This is basically the same as the previous one but illustrates a new subtlety about variables: the way they act like shredders of old values. This assignment says that **colour1** should get a copy of the value in **colour2**. As before, we make a copy of what is in the

When we assign a new value to a variable its old value is destroyed

variable after the **=** sign, here whatever is in **colour2**. Looking at our pictures of the boxes we see that the string **"green"** is currently in **colour2**, so it is a copy of **"green"** that we put it in **colour1**. However, there already is a value in **colour1**. Looking at our picture of the boxes we see that it holds the string **"red"**. Now variables can only hold one value at a time – only one value per box – so what happens? The old value is lost. Think of it as being shredded. There is no way of getting that original version of the value back as it is overwritten by the new value (we have lost the value unless we were clever enough to save a copy of it somewhere else first). The new state of our program is



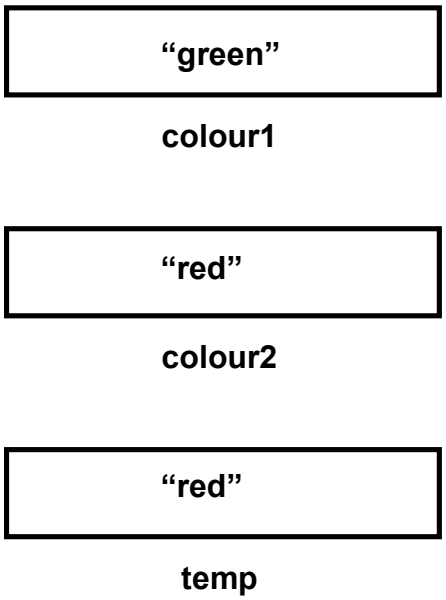
The key point here is that when you put a new value in a box, you destroy the old value. That value is now gone forever and the new value stored is the one just copied from the other variable.

The final assignment in our program is similar, this time copying the value from temp into **colour2** whose value is destroyed: **colour2** gets the value in **temp**.

colour2 = temp

The value in **temp** is the one saved from **colour1** in the earlier step (**"red"**). It is now stored in to variable **colour2**, and the old value that was there before (**"green"**) is shredded. That means even though the value in **colour1** was destroyed, because a copy was stored safely in to **temp** first, we end up with the same value in **colour2**. We have used **temp** as temporary storage (hence the name we gave it) to make sure we didn't really lose the value when it was destroyed.

Now look at the results when our fragment has finished. What has happened?



The values in **colour1** and **colour2** have been swapped over – using **temp** as extra storage space. Originally **colour1** was **"red"** and **colour2** was **"green"**. Now **colour1** is **"green"** and **colour2** is **"red"**.

Why might you want to do a swap like this? Well it is the core operation you need to do over and over again if you are going to sort data in to order, whether alphabetically, into numerical order or into some other ordering. That is something we use computers to do a lot.

Summary

We have seen that **variables** are like storage boxes that hold data and **assignment** is the command in a program that tells the computer to store new data in a variable. There are some subtleties about what happens during assignment though. It is best to think of variables as special boxes. They are storage boxes that have a copier and a shredder built in to them.

The important things to remember about variables are that:

- variables are given **identifiers** (names) so they can be referred to again,
- variables hold **values**: the actual data that is being stored,
- it is important not to confuse the identifiers of variables (their names) with their values,
- a variable can only store one value at a time,
- when you get a value from a variable you are making a copy: that variable's value is untouched, and
- when you store a new value in a variable you destroy anything that was previously there.

Exercise

Draw a series of pictures of boxes to show the state as it changes as this program fragment is executed step-by-step. Is it a swap program too, or does the swap go wrong?

```
colour1 = “red”
colour2 = “green”

colour1 = colour2
colour2 = colour1
```

Use of this material

This booklet was written based on an existing booklet about Java and distributed with support from the Mayor of London and Department for Education.

Department for Education

SUPPORTED BY MAYOR OF LONDON

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE