Programming in Python[1]

Robin Dawes

September 9, 2021

[1] With random servings of SPAM

Introduces an interesting (and ancient) method for multiplying integers. We also introduce some idiosyncrasies of Python.

.

Two Unusual Python Features

Python has a lot in common with other popular languages such as **C** and **Java**, but it has a few features that make it very different.

Indentation Is Part of the Language

In many languages, statements are grouped together using { and }

Example (not in any specific language):

```
if (x >= 2)
{
   y := 4;
   z := -12;
   if (t == 1)
   {
      p = 9;
      q = 10;
   }
}
print(x);
```

which could be legally written on one line (or split into many more lines) with no required alignment.

Python does not use braces to group statements together, and does not require a semi-colon at the end of each statement. Instead, a sequence of statements that are to be grouped together are all indented equally. The example above would be written in Python as

```
if (x >= 2):        # note the colon - it is required
   y = 4
   z = -12
   if (t == 1):
      p = 9
      q = 10
                    # empty lines are often used to make reading easier
print(x)
```

How far you indent is up to you - most people choose a small number of spaces such as 3 or 4 and always indent in steps of that size. I prefer to use indents of size 3 personally, but the **Idle** IDE uses 4, and that's perfectly ok. Or use 2, or 5, or 11 ... whatever looks best to you.

Indents of size 11 are a really bad idea. Your lines of code would extend past the right side of the screen!

Variables Have No Fixed Type

Many languages require you to declare variables and give each one a specific type of legal content. Example:

```
int x;
float y;
string z;
```

With these declarations, an assignment statement such as

```
x := 23.7;
```

would cause an error message because you can't assign a floating-point number to an integer variable.

In Python we don't need to declare variables at all - a variable is created when we assign it a value. Furthermore, we can change the type of a variable simply by assigning it a new value.

```
x = 4           # x is now an integer variable
x = 13.885      # x is now a floating point variable
x = "chocolate" # x is now a string variable
```

Well discuss other aspects of Python as we encounter them. To that end, let's look at a particular algorithm and go through the steps of implementing it as a Python program.

"Egyptian" Multiplication

THIS METHOD OF MULTIPLYING INTEGERS has been described under a lot of names. There does seem to be some evidence that it was used by mathematicians in ancient Egypt, but I believe it was also known and used in other ancient (and perhaps modern) cultures.

To multiply two integers x and y:

1. Write down x and y as the first entries in two columns of numbers
2. Under x, write down $\frac{x}{2}$, rounding down if necessary.
3. Under y, write down $y * 2$
4. Continue to fill the columns until the number in the first column is 1
5. Find the lines where the first number is odd
6. Add up the second numbers on those lines.
7. The sum of those numbers is the product of x and y

EXAMPLE: find the product of 22 and 31

	22	31
►	11	62
►	5	124
	2	248
►	1	496

$62 + 124 + 496 = 682$ which is correct.

In fact we can prove that this algorithm always finds the correct answer as long as x is a positive integer. But we're not going to prove it now, so we will just accept that it works.

The first step in developing a program is to write down the steps we need to complete, in normal human language. At this stage the steps can be expressed in very general terms. There are at least two good reasons for doing this: it gets you to think about how your program will be organized, and it forms the basis of the internal documentation that your program will need. If you write your description of the algorithm as a sequence of comments in an empty Python program, you won't have to cut and paste them later.

In the bad old days, we used to write our programs and then go back and add documentation. Now we understand it is better to do it in the other order: write the documentation first, then write the code.

So here is my first draft of an implementation for this algorithm (note that the # symbol indicates that the line is a comment):

```
# 20210909
# R. Dawes
#  Egyptian Multiplication

#  Get the values

#  Initialize working variables

#  Execute the repeated halving of the first value and doubling of the second value

#  Add up the needed values

#  Show the result
```

If this were a more complicated algorithm I would probably break some of those steps down into smaller steps. As this algorithm is pretty simple I'm going to start coding these steps directly.

Get the values

In a final version, I probably want to get the values from the user, or from a file, or from the command line. That's a decision I can make later, after I have made sure the program is doing the computation correctly. So for now I'm just going to pick two values.

```
#  Get the values
num_1 = 125
num_2 = 3
```

Initialize working variables

This section often gets developed in parallel with the rest of the program as we figure out what information we need to use in the completion of the algorithm. For this problem, I know I'm computing a product so I create a variable to hold that value. I also know I will be modifying the values I start with, so I decide to make copies of them and work on the copies so that I don't lose the original values.

I also decide to create two "list" variables which I will use to store the numbers that form the columns in the original description of the algorithm. I initialize the lists with the initial values, just as the algorithm starts with the two numbers at the head of the columns.

We will say a lot more about list variables on Monday.

```
#  Initialize working variables
product = 0

temp_1 = num_1
temp_2 = num_2

shrinking = [temp_1]
doubling = [temp_2]
```

Execute the repeated halving ...

This is a repeated action so it is natural to use a loop. We don't know how many times the loop will execute so a **while** loop is appropriate. We want to repeat the loop as long as the first number has not reached 1. Inside the loop we want to divide the first number by 2, double the second number, and add those new values to our lists.

```
#  Execute the repeated halving of the first value and doubling of the second value
while (temp_1 != 1):
   temp_1 = temp_1 // 2
   temp_2 = temp_2*2
   shrinking.append(temp_1)
   doubling.append(temp_2)
```

Note the "//" for division. This operation means "divide and round down"

Also note the **append** operations. These add the temp_1 and temp_2 values to the end of the lists.

Add up the needed values

Now we work through the lists, using the values from the first list do decide whether or not we should include particular values from the second list.

We could use another **while** loop to do this, but I wanted to include a **for** loop in this program just to cover that language feature. We use the built-in **len** function to find out how long the lists are, and then we iterate through the lists and add the values we need to *product*.

```
#  Add up the needed values
how_many = len(shrinking)

for i in range(how_many):
   if shrinking[i] % 2 == 1:
      product = product + doubling[i]
```

Note the use of the **range** function to create the full range of index values that *i* will take. We will discuss this in detail in our next class. Also note the use of % ("mod") to get the remainder when the value is divided by 2 ... this determines if the value is odd.

Show the result

This is the easiest of all. But the **print** instruction has some details that need to be discussed.

```
#  Show the result
print("The product of",num_1,"and",num_2,"is",product)
```

The Python **print** instruction will print any number of items that are separated by commas. For any item that is a variable (such as **num_1**) or an expression (such as **1+2**) it prints the value. It puts 1 blank space between the items.

It is possible to take more precise control of the appearance of printed output using **format strings**. We may not have time to discuss these in CISC-121.

Putting It All Together

The complete Python program looks like this:

```
# 20210907
# R. Dawes

#  Egyptian Multiplication

#  Get the Values
num_1 = 125
num_2 = 3

#  Initialize working variables
product = 0

temp_1 = num_1
temp_2 = num_2

shrinking = [temp_1]
doubling = [temp_2]


#  Execute the repeated halving of the first value and doubling of the second value
while (temp_1 != 1):
   temp_1 = temp_1 // 2
   temp_2 = temp_2*2
   shrinking.append(temp_1)
   doubling.append(temp_2)


#  Add up the needed values
how_many = len(shrinking)

for i in range(how_many):
   if shrinking[i] % 2 == 1:
      product = product + doubling[i]

#  Show the result
print("The product of",num_1,"and",num_2,"is",product)
```