VISUAL QUICKSTART GUIDE

Python

Third Edition

TOBY DONALDSON

LEARN THE QUICK AND EASY WAY!

VISUAL QUICKSTART GUIDE

Python

TOBY DONALDSON

Peachpit Press

Comparing For-Loops and While-Loops

Let's take a look at a few examples of how for-loops and while-loops can be used to solve the same problems. Plus we'll see a simple program that can't be written using a for-loop.

Calculating factorials

Factorials are numbers of the form $1 \times 2 \times 3 \times \ldots \times n$, and they tell you how many ways n objects can be arranged in a line. For example, the letters ABCD can be arranged in $1 \times 2 \times 3 \times 4 = 24$ different ways. Here's one way to calculate factorials using a for-loop:

```
# forfact.py

n = int(input('Enter an integer
→ >= 0: '))

fact = 1

for i in range(2, n + 1):

  fact = fact * i

print(str(n) + ' factorial is ' +
→ str(fact))
```

Here's another way to do it using a while-loop:

```
# whilefact.py

n = int(input('Enter an integer
→ >= 0: '))

fact = 1

i = 2

while i <= n:

  fact = fact * i

  i = i + 1

print(str(n) + ' factorial is ' +
→ str(fact))
```

continues on next page

Both of these programs behave the same from the user's perspective, but the internals are quite different. As is usually the case, the while-loop version is a little more complicated than the for-loop version.

TIP **In mathematics, the notation *n!* is used to indicate factorials. For example, 4! = 1 × 2 × 3 × 4 = 24. By definition, 0! = 1. Interestingly, there is *no* simple formula for calculating factorials.**

TIP **Python has no maximum integer, so you can use these programs to calculate very large factorials. For example, a deck of cards can be arranged in exactly 52! ways:**

```
Enter an integer >= 0: 52
52 factorial is 80658175170943878571
→ 66063685640376697528950544088327778
→ 24000000000000
```

Summing numbers from the user

The following programs ask the user to enter some numbers, and then prints their sum. Here is a version using a for-loop:

```
# forsum.py

n = int(input('How many numbers to
→ sum? '))

total = 0

for i in range(n):

  s = input('Enter number ' +
  → str(i + 1) + ': ')

  total = total + int(s)

print('The sum is ' + str(total))
```

Here's a program that does that same thing using a while-loop:

```
# whilesum.py

n = int(input('How many numbers to
→ sum? '))

total = 0

i = 1

while i <= n:

  s = input('Enter number ' +
  → str(i) + ': ')

  total = total + int(s)

  i = i + 1

print('The sum is ' + str(total))
```

Again, the while-loop version is a little more complex than the for-loop version.

TIP These programs assume that the user is entering integers. Floating point numbers will be truncated when `int(s)` is called. Of course, you can easily change this to `float(s)` if you want to allow floating point numbers.

Summing an unknown number of numbers

Now here's something that can't be done with the for-loops we've seen so far. Suppose we want to let users enter a list of numbers to be summed without asking them ahead of time how many numbers they have. Instead, they just type **'done'** when they have no more numbers to add. Here's how to do it using a while-loop:

```
# donesum.py
total = 0
s = input('Enter a number (or
→ "done"): ')
while s != 'done':
  num = int(s)
  total = total + num
  s = input('Enter a number (or
→ "done"): ')
print('The sum is ' + str(total))
```

The idea here is to keep asking users to enter a number, quitting only when they enter **'done'**. The program doesn't know ahead of time how many times the loop body will be executed.

Notice a few more details:

- We must call **`input`** in two different places: before the loop and inside the loop body. This is necessary because the loop condition decides whether or not the input is a number or **`'done'`**.
- The ordering of the statements in the loop body is very important. If the loop condition is **`True`**, then we know **`s`** is not **`'done'`**, and so we assume it is an integer. Thus we can convert it to an integer, add it to the running total, and then ask the user for more input.
- We convert the input string **`s`** to an integer only *after* we know **`s`** is not the string **`'done'`**. If we had written

  ```
  s = int(input('Enter a number
  → (or "done"): '))
  ```

  as we had previously, the program would crash when the user typed **`'done'`**.
- There is no need for the **`i`** counter variable anymore. In the previous summing programs, **`i`** tracked how many numbers had been entered so far. As a general rule, a program with fewer variables is easier to read, debug, and extend.

Breaking Out of Loops and Blocks

The **break** statement is a handy way for exiting a loop from anywhere within the loop's body. For example, here is an alternative way to sum an unknown number of numbers:

```
# donesum_break.py
total = 0
while True:
  s = input('Enter a number (or
  → "done"): ')
  if s == 'done':
    break # jump out of the loop
  num = int(s)
  total = total + num
print('The sum is ' + str(total))
```

The while-loop condition is simply **True**, which means it will loop forever unless **break** is executed. The only way for **break** to be executed is if **s** equals **'done'**.

An advantage of this program over **donesum.py** is that the **input** statement is not repeated. But a disadvantage is that the reason for why the loop ends is buried in the loop body. It's not so hard to see it in this small example, but in larger programs **break** statements can be tricky to see. Furthermore, you can have as many **break**s as you want, which adds to the complexity of understanding the loop.