Lecture 4 - Iterations

Adnan Ferdous Ashrafi

Stamford University Bangladesh

Table of Contents

Reassignment

It is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

Let's see an example:

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like $a = b$ as a mathematical proposition of equality; that is, the claim that a and b are equal. But this interpretation is wrong.
First, equality is a symmetric relationship and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement $a = 7$ is legal and $7 = a$ is not.

Reassignment

Also, in mathematics, a proposition of equality is either true or false for all time. If $a = b$ now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

```
>>> a = 5
>>> b = a # a and b are now equal
>>> a = 3 # a and b are no longer equal
>>> b
5
```

The third line changes the value of a but does not change the value of b, so they are no longer equal.
Reassigning variables is often useful, but you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

Updating variables

A common kind of reassignment is an update, where the new value of the variable depends on the old.

```
>>>x = 15
>>>x = x + 1
```

This means “get the current value of x, add one, and then update x with the new value.”
If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to initialize it:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an increment; subtracting 1 is called a decrement.

Iterations

Definition

Iteration, is the ability to run a block of statements repeatedly. Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called **iteration**.

Iteration is the repetition of a process in order to generate an outcome. The sequence will approach some end point or end value. Each repetition of the process is a single iteration, and the outcome of each iteration is then the starting point of the next iteration.

Types of iterations

Iterations can be achieved in several ways. The most common three ways are:

1. FOR Loops
2. WHILE Loops, and
3. RECURSION

In the context of today's lecture we will learn about two of the above, i.e., **FOR** and **WHILE** loops.

FOR Loops

Definition

We can repeat any statement a given number of times using the FOR loop syntax that comes with a Python interpreter.

Let's see an example:

```python
for i in range(4):
    print('Hello!')
```

The output should be

```
Hello!
Hello!
Hello!
Hello!
```

Syntax of FOR Loops

The syntax of a *for* statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements.

A *for* statement is also called a loop because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

The general syntax of a for loop is:

```python
for some_variable in range(some_value):
    # statement to be executed by the for loop
    # The indentation is important!
```

The range function generates numbers from 0 up-to the specified number. So, $range(4)$ generates 0,1,2 and 3.

Printing numbers with for loops

Suppose we want to write a for loop that prints the first 100 numbers from 0.

The loop should be as follows:

```
for i in range(100):
    print(i)
```

The range function generates numbers from 0 up-to the specified number. So, $range(100)$ generates $0, 1, 2 \ldots 99$. And as you can see, each time the value of i is automatically increased by 1.

Printing numbers with for loops

Suppose we want to write a for loop that prints the squares of the first 100 numbers from 0.

The loop should be as follows:

```
for i in range(100):
    print(i*i)
```

or

```
for i in range(100):
    print(i**2)
```

The range function generates numbers from 0 up-to the specified number. So, $range(100)$ generates $0, 1, 2 \ldots 99$. And as you can see, each time the value of i is automatically increased by 1.

Finding the sum of a series using for loops

Suppose we want to write a for loop that sums the first 100 numbers from 0 and prints the value.

The loop should be as follows:

```
sum = 0 # initialize the value of the sum with 0
for i in range(100): # the loop runs for a 100 times
    sum = sum + i # in each iteration a number is added to the sum

print(sum) # prints the new sum
```

or

```
sum = 0 # initialize the value of the sum with 0
for i in range(100): # the loop runs for a 100 times
    sum += i # in each iteration a number is added to the sum

print(sum) # prints the new sum
```

Finding the sum of a specific series using for loops

Suppose we want to write a for loop that sums the numbers from 5 to 80 and prints the value.

The loop should be as follows:

```
sum = 0
for i in range(5,80): # the loop runs for 75 times from 5 to 79
    sum += i

print(sum)
```

The $range(first, second)$ generates numbers from the $first$ number up-to $(second - 1)$ number. So, $range(5, 80)$ generates numbers $5, , 6, 7, \dots 79$.

WHILE Loops

Because iteration is so common, Python provides language features to make it easier. One of this feature is a **WHILE** loop.

```python
n = 10
while n > 0:
    print(n)
    n = n - 1
print('Finished!')
```

You can almost read the while statement as if it were English. It means, “While *n* is greater than 0, display the value of *n* and then decrement *n*. When you get to 0, display the word Finished!”

Syntax of WHILE Loops

The syntax of WHILE loop is as follows:

```
while some_variable [relational_operator] some_value:
    # statements to execute
    # be careful to either increment or decrement counters
# This statement will be executed after the while loop finishes
```

The statements inside the while loop, i.e., the indented lines will be executed as long as the condition(s) holds *true*. As soon as the condition becomes *false*, the loop is exited.

By definition, we specify all **increment** or **decrement** operations inside the while loop so that it reflects while checking the conditional.

Flow of execution for a while statement

The flow of execution can be determined as follows:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

break statement in a loop

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the break statement to jump out of the loop. For example:

```
i = 0
while True:
    if i == 10:
        break
    print(i)
    i = i + 1
print('Done!')
```

The loop condition is *True*, which is always true, so the loop runs until it hits the break statement.
This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

An infinite loop

The following example demonstrates an infinite loop:

```python
while i > 0:
    print(i)
    i = i + 1
print("This statement will never run!!!")
```

So, during coding, be careful to implement the while loop so that, it does not turn into an infinite loop.

Running this code in your Jupyter Notebook might show some very weird behavior. You might need to restart the kernel.

Finding Square Roots using WHILE loop

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2} \tag{1}$$

For example, if a is 4 and x (take any estimate as you wish) is 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

The result is closer to the correct answer $(\sqrt{4} = 2)$. But the result is not exact. Let's try to improvise the result.

Finding Square Roots using WHILE loop

If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

Finding Square Roots using WHILE loop

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

When $y == x$, we can stop.

Let's see a while loop implementation of the problem.

Finding Square Roots using WHILE loop

Here is a loop that starts with an initial estimate, *x*, and improves it until it stops changing:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

For most values of a this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a float.

Finding Square Roots using WHILE loop

Rather than checking whether *x* and *y* are exactly equal, it is safer to use the built-in function *abs* to compute the absolute value, or magnitude, of the difference between them:

```
epsilon = 0.0000001
while True:
    print(x)
    y = (x + a/x) / 2
    if abs(y-x) < epsilon:
        break
    x = y
```

Where epsilon has a value like 0.0000001 that determines how close is close enough.

Algorithm

Newton's method is an example of an algorithm: it is a mechanical process for solving a category of problems (in this case, computing square roots).

To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy", you might have learned a few tricks. For example, to find the product of n and 9, you can write $n1$ as the first digit and $10n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an **algorithm**!

Algorithm

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection”. For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Debugging

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

Glossary

reassignment: Assigning a new value to a variable that already exists

update: An assignment where the new value of the variable depends on the old.

initialization: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

decrement: An update that decreases the value of a variable.

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

infinite loop: A loop in which the terminating condition is never satisfied.

algorithm: A general process for solving a category of problems.

Further Reading

- Chapter 4 and 7 of Think Python(2nd Edition) - Allen B. Downey
- Chapter 4 and 7 of Python Crash Course - Eric Matthes
- Python Official Website
- Lecture 4 Jupyter Notebook

Thank you.
Any Questions?