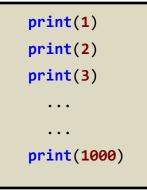**CSSE 120 – Introduction to Software Development**

Concept: *Counted Loops* and *Range expressions*

Loops

A *loop* is, well, something that *loops*, that is, **executes repeatedly**. For example, to print the numbers 1, 2, 3, 4, … 1000, you could either:

- Stupid approach: Write 1,000 print statements:

```
print(1)
print(2)
print(3)
    ...
    ...
print(1000)
```

- Sensible approach: Write a *single* loop whose *body* runs 1,000 times:

```
for k in range(1000):
    print(k)
```

Do you see why loops are valuable?

range expressions

For the first type of loop that we will examine we need *range* expressions. There are three forms of *range* expressions. Here is the first (we'll see the other two later in this document).

- **range(n)** – generates the sequence of integers: **0, 1, 2, ... n-1**.
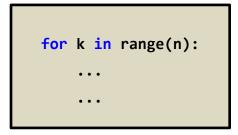 - For example, **range(7)** generates the sequence: **0 1 2 3 4 5 6**.
  - ⚠ The sequence generated by **range(n)** has **n** numbers in it. Note that the sequence starts at **0**, not **1**, hence stops at **n-1**. We will see later why this is handy.

Counted loops

There are many kinds of loops. For now, we will introduce only *counted loops* – loops that go a certain number of times, for example a loop that goes 500 times or a loop that goes *n* times where *n* is a variable with an integer value.

A *counted loop* has the form shown in the box to the right, where *k* can be any variable and *n* can be any variable or constant whose value is an integer. The *for* statement makes its *body* (the indented part, shown as **...** in the box to the right) run *n* times, with *k* set to **0, 1, 2, ... n-1,** per the *range* expression.

```
for k in range(n):
    ...
    ...
```

Here (on the next page) are some examples:

Code snippet

What the code snippet prints

```
for k in range(10):

    print(k, '    ', math.sin(k))
```

```
0       0.0
1       0.8414709848078965
2       0.9092974268256817
3       0.1411200080598672
4       -0.7568024953079282
5       -0.9589242746631385
6       -0.27941549819892586
7       0.6569865987187891
8       0.9893582466233818
9       0.4121184852417566
```

> The variable **k** takes on the values **0, 1, 2, ... 9,** per the **range** statement. You can (and usually do) also use **k** in *expressions* in the body of the loop, as in the above example.

```
x = 6

for blah in range(x):

    print(blah, '    ', math.sin(blah))
```

```
0       0.0
1       0.8414709848078965
2       0.9092974268256817
3       0.1411200080598672
4       -0.7568024953079282
5       -0.9589242746631385
```

> The variable after the symbol **for** is called the *index variable*. It can be any variable (as in the silly example above), but the common style is to use single-letter variable names like *i*, *j*, *k*, *m* and *n*.

> The entire **body** of the **for** loop (that is, the *indented lines*) are executed repeatedly. So in this example, the loop runs 4 times, printing 4 things each time, producing 16 lines of output.

```
for k in range(4):

    print(k)

    print(k * k)

    print(math.sqrt(k))

    print()
```

```
0
0
0.0

1
1
1.0

2
4
1.4142135623730951

3
9
1.7320508075688772
```

> **print** with nothing in the parentheses simply prints a blank line.

range expressions, all three forms

- **range(n)** – generates the sequence of integers: **0, 1, 2, ... n-1**.

  - For example, **range(7)** generates the sequence: **0 1 2 3 4 5 6**.

  - The sequence generated by **range(n)** has **n** numbers in it. Note that the sequence starts at **0**, not **1**, hence stops at **n-1**. We will see later why this is handy.
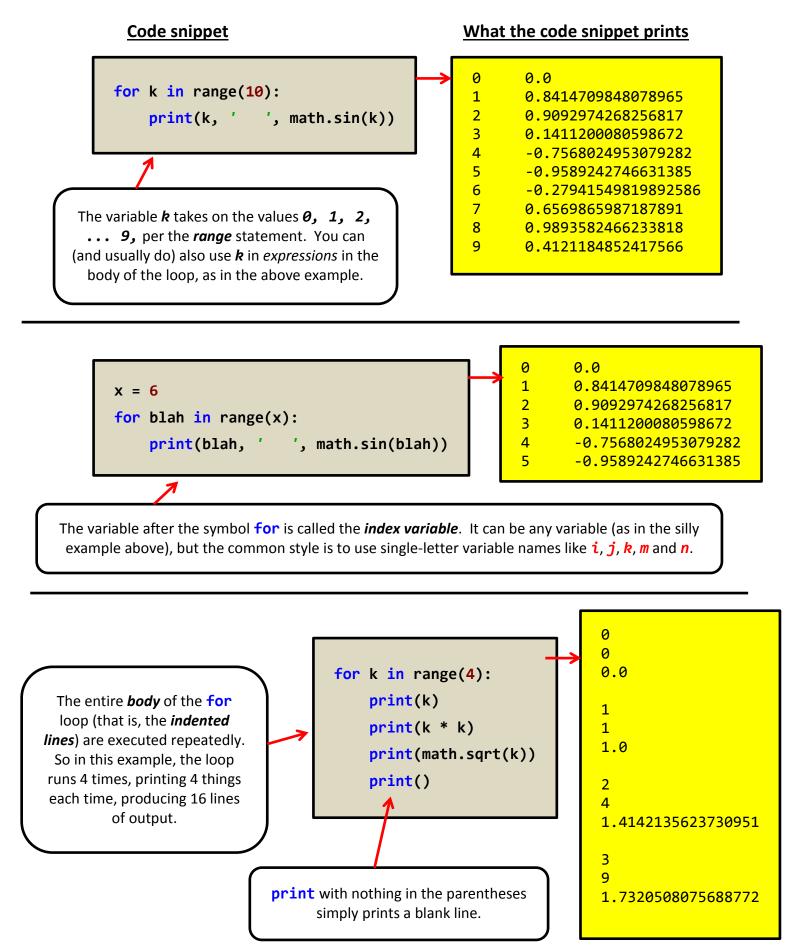
- **range(m, n)** – generates the sequence of integers: **m, m+1, m+2, ... n-1**.

  - For example, **range(7, 10)** generates the sequence: **7 8 9**.

  - The sequence generated by **range(m, n)** has **n-m** numbers in it. Note that the sequence ends at **n-1**, not **n**, just like in the one-argument form of *range*.

- **range(m, n, i)** – generates the sequence of integers:

    m, m + i, m + 2*i, m + 3*i, m + 4*i, ... up to but NOT including n.

  That is, the sequence starts at *m* and goes up in "steps" of *i* from *m*, stopping when it would be equal to or more than *n*.

  - For example, **range(7, 30, 6)** generates the sequence:

        7   13   19   26

    and **range(4, 10, 2)** generates the sequence: **4   6   8.**

    Note that **range(4, 10, 2)** does NOT include the **10**.

  Exception: If the third argument *i* is *negative*, the sequence starts at *m* and goes DOWN to *n*, in increments of *i*, stopping when it would be equal to or *less* than *n*.

  - For example, **range(7, -30, -6)** generates the sequence:

        7   1   -5   -11   -17   -23   -29

    and **range(10, 4, -2)** generates the sequence: **10   8   6.**

  - Do you see why the range expression **range(10, 4, 2)** generates the *empty sequence* (that is, the sequence with no items in it)?

  The arguments to **range** must all be integers (whole numbers). You cannot "step" by a fractional amount.

More *counted loop* examples

On the next page …

Code snippet   **What the code snippet prints**

```python
for k in range(3, 7):
    print(k, '    ',
math.sin(k))
```

```
3        0.1411200080598672
4        -0.7568024953079282
5        -0.9589242746631385
6        -0.27941549819892586
```

```python
for k in range(3, 20, 4):
    print(k, '    ', math.sin(k))
```

```
3        0.1411200080598672
7        0.6569865987187891
11       -0.9999902065507035
15       0.6502878401571168
19       0.149877209662952345
```

```python
for k in range(20, -10, -5):
    print(k, '    ', math.sin(k))
```

```
20       0.9129452507276277
15       0.6502878401571168
10       -0.5440211108893698
5        -0.9589242746631385
0        0.0
-5       0.9589242746631385
```

```python
for k in range(100, -1, -10):
    print(k)
```

```python
for k in range(11):
    print(100 - (k * 10))
```

```python
for k in range(0, 101, 10):
    print(100 - k)
```

```
100
90
80
70
60
50
40
30
20
10
0
```

Three ways to accomplish the same output!
We will study these techniques in detail over the next few sessions.