CSCE 110: Programming I

David Kebo Houngninou

Basics of Python

Repetition / Loops (for, while)
What is a loop?

Definition:
A loop is a control flow statement that allows instructions to be executed repeatedly based on a given condition.

Purpose:
A loop is used in programming to repeat a block of instructions based on a given condition.

for Loop

At each iteration var takes on one value in the sequence

Set of values

Body of the loop

If the number of elements of sequence is n, the loop is repeated n times
for loop

The for loop iterates over a sequence or an iterable object.

```
for iterator in sequence:
    do something
```

*iterator* is a variable representing an item in the sequence at each iteration.

The loop continues until it reaches the last value in *sequence*.

The datatypes for *sequence* are objects that support iteration:

- strings
- lists
- range objects
- tuples

for loop: flowchart

```
for each value in the sequence

last value reached ?

Yes

Exit

No

Process

Body of the loop
```

Exit
Example

```python
state = "alabama"
# iterate over a string
for letter in state:
    print (f"{letter}")
```

>>> [evaluate letters.py]
    a
    l
    a
    b
    a
    m
    a

range() function

The `range()` function returns an immutable sequence of integers between the given start integer to the stop integer.

```
range(start, stop, step)
```

- **start**: Starting point of the sequence. Defaults to 0.
- **stop**: Endpoint of the sequence. Not included in the sequence.
- **step**: (optional) Step size of the sequence. Defaults to 1
range() function

The range() function uses the start, stop and step to generates the next number.

The list() function outputs the items in the sequence.

```python
>>> print(range(10))
range(0, 10)
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(list(range(2, 8)))
[2, 3, 4, 5, 6, 7]
>>> print(list(range(2, 20, 3)))
[2, 5, 8, 11, 14, 17]
```

range() function

a is the sequence 0, 1, 2, 3, 4

```python
>>> a = range(5)
>>> a
range(0, 5)
>>> a[0]
0
>>> a[1]
1
>>> len(a)
5
>>> a[4]
4
```
for loop: range()

```python
# Print the numbers 0,1,2,3,4
for x in range(5):
    print(x)
print()

# Print 3,4,5
for x in range(3, 6):
    print(x)
print()

# Print 3,5,7
for x in range(3, 8, 2):
    print(x)
```

>> [evaluate range_example.py]

0
1
2
3
4
3
4
5
3
5
7

for loop: break, continue, pass

The `break` statement **terminates** the for loop containing it. Control of the program flows to the statement immediately **after** the body of the for loop.
The `break` statement, exits out of the body of the for loop. We put a `break` statement within the code under the for loop usually after a conditional statement.

```python
number = 0
for number in range(10):
    number = number + 1
    if number > 5:
        break  # break here
print(f"{number}")
print('Out of loop!')
```

The `continue` statement, continues with the next iteration of the loop. It skips the part of a loop based on a given condition.

We put a `continue` statement within the code under the for loop usually after a conditional statement.
The continue statement skips the rest of the code inside the for loop for the current iteration only. The for loop does not terminate but continues on with the next iteration.

```python
number = 0
for number in range(10):
    number = number + 1
    # continues with the next iteration
    if number == 5:
        continue
    print(f"{number}")
print("Out of loop!")
```

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

Under a given condition, the pass statement handles the condition without the loop being impacted.

```python
number = 0
for number in range(10):
    number = number + 1
    if number == 5:
        pass  # pass and do nothing
    print(f"{number}")
print("Out of loop!")
```
### for loop: else

The for loop can have an optional `else` block. The else block executes if the loop reaches the last item in the sequence.

If a break statement stops the for loop the `else` statement part is ignored.

```python
state = "alabama"
# iterate over the string
for letter in state:
    print(f"{letter}")
else:
    print(f"All {len(state)} letters printed")
```

---

### Nested loop

A nested loop is a loop inside a loop.

The inner loop executes one time for each iteration of the outer loop.

```python
brands = ["samsung", "lg", "sony"]
devices = ["tv", "computer", "phone"]

for brand in brands:
    for device in devices:
        print(f"{brand} {device}")
```
while loop

The while loop iterates over a block of code as long as the given condition is true.

We generally use this loop when we don't know, the number of times to iterate.

while condition:
    do something

When to use a While loop?

Use a while loop to repeat instructions an unknown number of times, until a condition is met.

e.g.: ask a user for a number between 1 and 10.

We don't know how many times the user may enter a incorrect, so the program keeps asking while the number is not between 1 and 10.

If the you know exactly how many times to execute instructions, then use a for loop!
while loop: flowchart

Example

Write a program to add natural numbers up to n

```python
n = 10
# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1  # update counter

# print the sum
print("The sum is", sum)
```

> [evaluate sum.py]
> The sum is 55
while loop: break, continue, pass

The `break` statement terminates the while loop containing it. Control of the program flows to the statement immediately after the body of the while loop.

```python
while test expression:
    # codes inside while loop
    if condition:
        break
    # codes inside while loop
# codes outside while loop
```

while loop: break, continue, pass

The `break` statement, exits out of the body of the while loop. We put a `break` statement within the code under the while loop usually after a conditional statement.

```python
number = 0
while number <= 10:
    number = number + 1
    if number == 5:
        break  # break here

print(f"{number}")
print("Out of loop!")

> [evaluate while_example.py]
1
2
3
4
Out of loop!
```
while loop: break, continue, pass

The `continue` statement, continues with the next iteration of the loop. It skips the part of a loop based on a given condition.

We put a `continue` statement within the code under the while loop usually after a conditional statement.

```python
while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes inside while loop

# codes outside while loop
```

while loop: break, continue, pass

The `continue` statement skips the rest of the code inside the while loop while the current iteration only. The while loop does not terminate but continues on with the next iteration.

```python
def example_function():
    number = 0
    while number <= 10:
        number = number + 1
        if number == 5:
            continue  # continue here
        print(f"{number}")
    print("Out of loop!")

example_function()
```

> [evaluate while_example.py]

1
2
3
4
5
6
7
8
9
10
11

Out of loop!
while loop: break, continue, pass

The **pass** statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

Under a given condition, the **pass** statement handles the condition without the loop being impacted.

```python
number = 0
while number <= 10:
    number = number + 1
    if number == 5:
        pass  # pass here
    print(f"{number}")
print("Out of loop!")
```

while loop: else

The **while** loop can have an optional **else** block. The else part is executed if the condition in the while loop evaluates to False.

If a break statement stops the while loop the **else** statement part is ignored.

```python
counter = 0
while counter <= 3:
    print(f"count: {counter}")
    counter = counter + 1
else:
    print("End of count")
```
References
