CptS 111, Spring 2023
Lect. #15, Mar. 8, 2023
Class Notes

Today's Agenda:

1. Iterating `for` -loops
2. Iterating vs counting `for` -loops
3. `while` -loops
4. `break` and `continue`

Ch. 6 (cont.)

Loops (cont.)

On Monday we covered counting `for` -loops which are of the form:

```
for i in range(n):
    <loop_body>
```

where `i` is the loop variable which takes on the values of the integer sequence created by the `range()` function, and `n` is an integer indicating the number of times the loop body can be repeated. Recall that `n` can represent a function that evaluates to an integer, and it's common for `len(<iterable>)` to be used as the value for `n` . Today we'll cover iterating `for` -loops.

1. Iterating `for` -loops

Template for an iterating `for` -loop:

```
for <item> in <iterable>:
    <loop body>
```

In the `for` -loop header, `<item>` is the loop variable. Each time the `for` -loop iterates through the `<iterable>` , the loop variable is assigned the value of the next element in the `<iterable>` . **An iterable always appears in the header of a `for` -loop!** ***Don't forget what the iterables are in Python: strings, lists, tuples, sets, and dictionaries!*** Consider the following examples:

In [1]:

```
# Iterable is a list
# Use a descriptive name for the loop variable, i.e., don't use i, j, ...

goodies = ['cookies', 'croissants', 'coffee ice cream', 'scones', 'browni
for goodie in goodies:
    print(f'I love {goodie}!')
```

```
I love cookies!
I love croissants!
I love coffee ice cream!
I love scones!
I love brownies!
```

Recall that we used a counting `for` -loop to generate identical results. Let's compare the two.

```
for i in range(len(goodies)):
    print(f'I love {goodies[i]}!')
```

Thus, sometimes we can use either type of `for` -loop. However, as we'll see below, sometimes we can't.

In [2]:

```
# Iterable is a string

word = 'alphabet'
for ch in word:
    print(ch, end=' ')
```

```
a l p h a b e t
```

In [3]:

```
# Iterable is a tuple

nums = (5, 4, 3, 2, 1, 0)
for num in nums:
    print(num)
print('Blast off!')
```

```
5
4
3
2
1
0
Blast off!
```

In the examples above, we assigned values to lvalues and used the lvalue names as the iterables. However, we can also use the values themselves as the iterables.

In [4]:

```
# Another list example

for car in ['Ford', 'Toyota', 'Tesla', 'Rivian']:
    print(f'I drive a {car}.')
```

```
I drive a Ford.
I drive a Toyota.
I drive a Tesla.
I drive a Rivian.
```

It's actually better to define the list first, i.e.,

```
cars = ['Ford', 'Toyota', 'Tesla', 'Rivian']
for car in cars:
    print(f'I drive a {car}.')
```

This is because defining a list in a header makes the code harder to read.

In [5]:

```
# Another string example

for ch in 'Hello, World!':
    print(ch, end=' ')
```

```
H e l l o ,   W o r l d !
```

In [6]:

```
# Another tuple example

for prime in (2, 3, 5, 7, 11, 13, 17, 19, 23, 29):
    print(prime, end=' ')
```

```
2 3 5 7 11 13 17 19 23 29
```

Again, usually it's better to define a string or tuple first and not include it in the `for` -loop header.

2. Iterating vs Counting `for` -loops

When should we use an iterating `for` -loop and when should we use a counting `for` -loop? If we're working with an iterable, it's usually easier to use an iterating `for` -loop, but to some degree it's a matter of personal preference.

Sometimes, however, we have to use a counting `for` -loop with an iterable, ***e.g., when we want to change the values in the iterable itself***. An iterating `for` -loop doesn't allow us to do this. Consider the following example in which we want to change all the negative numbers in a list to zero.

In [7]:

```
# Can't change list values using an iterating for-loop

nums = [-99, 25, 18.2, -5, 82, 14, 3.4, -1.9]
for num in nums:
    print()
    print('value of num outside conditional =', num)
    if num < 0:
        num = 0
        print('value of num in conditional body, i.e., when num < 0 =', n

print()
print('nums =', nums)
```

```

value of num outside conditional = -99
value of num in conditional body, i.e., when num < 0 = 0

value of num outside conditional = 25

value of num outside conditional = 18.2

value of num outside conditional = -5
value of num in conditional body, i.e., when num < 0 = 0

value of num outside conditional = 82

value of num outside conditional = 14

value of num outside conditional = 3.4

value of num outside conditional = -1.9
value of num in conditional body, i.e., when num < 0 = 0

nums = [-99, 25, 18.2, -5, 82, 14, 3.4, -1.9]
```

This code didn't work because `num` isn't in the list `nums`. Rather ***it's the loop variable that's assigned each value in the list*** as the `for`-loop iterates through the list. `num` is changed if its value is less than zero, but **the *list* itself isn't changed**. Next, consider the following:

In [8]:

```
# Changing list values using a counting for-loop

nums = [-99, 25, 18.2, -5, 82, 14, 3.4, -1.9]
for i in range(len(nums)):
    if nums[i] < 0:
        nums[i] = 0
        print(f'nums[{i}] = {nums[i]}')
print('nums =', nums)
```

```
nums[0] = 0
nums[3] = 0
nums[7] = 0
nums = [0, 25, 18.2, 0, 82, 14, 3.4, 0]
```

This code worked because we counted each iteration in the list (`i=0, i=1, ...`), and each counter matched the index of the item in the list. Then we changed the actual item in the list whenever it was negative.

Another type of loop construct in most programming languages is the `while` -loop.

3. `while` -loops

`while` -loop template:

```
while <test_statement>:
    <loop_body>
```

The `<test_statement>` is evaluated. If it's `True` , the `<loop_body>` is executed. The process is repeated until the test is `False` .

Let's look at some examples:

In [9]:

```
# Countdown from 5

i = 5
while i >= 0:
    print(i)
    i -= 1          # We often use augmented assignment with while-loops
print('Final value of i:', i)
```

```
5
4
3
2
1
0
Final value of i: -1
```

Note that the final value of `i` is -1. Control is returned to the header, but when the test condition `i >= 0` fails, the next command (the `print()` statement) is run.

Next, let's look at a few examples that demonstrate how we can change the test expression.

In [10]:

```
# Use of conditional to change test expression

bits = 2
switch = 'on'
while switch == 'on':
    print(bits, end=' ')
    if bits == 1024:
        switch = 'off'     # Change 'switch' to leave loop when bits is 1
    bits *= 2
```

```
2 4 8 16 32 64 128 256 512 1024
```

In [ ]:

```
# Use of input to change test expression

names = []
name = input('Enter a name [or return to stop]: ')
while name != '':                    # empty string ('') = nothing entered
    names.append(name)
    name = input('Enter a name [or return to stop]: ')
names
```

4. `break` and `continue`

`break` allows us to exit a loop; `continue` returns us to the loop header. Both can be used with either `for` - or `while` -loops.

Use `break` and `continue` statements sparingly because they can cause difficulty when trying to find logic errors in code.

In [11]:

```
# Use of break to exit while-loop ***(PA #5 USES break !!!)***

positives = []
while True:
    num = int(input('Enter an integer [0 to stop]: '))
    if num == 0:
        break
    elif num < 0:
        continue
    positives.append(num)
print()
print('positives =', positives)
```

```
Enter an integer [0 to stop]: -99
Enter an integer [0 to stop]: 42
Enter an integer [0 to stop]: 88
Enter an integer [0 to stop]: 101
Enter an integer [0 to stop]: 23
Enter an integer [0 to stop]: -18
Enter an integer [0 to stop]: 77
Enter an integer [0 to stop]: 0

positives = [42, 88, 101, 23, 77]
```

Notice that `continue` took us back to the beginning of the loop and, thus, no negative numbers were appended to the list. `break` exited us from the loop completely. Here's an example for a common use of `break` :

In [12]:

```
names = []
while True:
    name = input('Enter a name [return to stop]: ')
    if name == '':
        break
    names.append(name)
print()
print(names)
```

```
Enter a name [return to stop]: SpongeBob
Enter a name [return to stop]: Freddie Mercury
Enter a name [return to stop]: Strong Bad
Enter a name [return to stop]: Trogdor
Enter a name [return to stop]: Harry Potter
Enter a name [return to stop]: 

['SpongeBob', 'Freddie Mercury', 'Strong Bad', 'Trogdor', 'Harry Potte
r']
```

In [13]:

```
# Non-void function with loop and conditional

def sort_names(names):   # parameter names is a list
    short_names = []
    med_names = []
    long_names = []
    for name in names:
        if len(name) <= 3:
            short_names.append(name)
        elif len(name) >= 8:
            long_names.append(name)
        else:
            med_names.append(name)
    return short_names, med_names, long_names

names = ['Angela', 'Kaleb', 'Ben', 'Latrra', 'Warren', 'Jefferson', 'Jess
         'Sam', 'Hailee', 'Roselynne', 'Juan', 'Rajendriya', 'Maili', 'El
short, med, long = sort_names(names)
print('short names:', short)
print('medium names:', med)
print('long names:', long)
```

```
short names: ['Ben', 'Sam', 'Eli']
medium names: ['Angela', 'Kaleb', 'Latrra', 'Warren', 'Jessica', 'Haile
e', 'Juan', 'Maili']
long names: ['Jefferson', 'Roselynne', 'Rajendriya']
```