Module 5: Loops

February 24, 2021

Last time we talked about boolean expressions and conditional branching. Boolean expressions allow us to express conditions and to check at runtime if they are `True` or `False`. With the `if` statement we can deviate from the normal sequential control-flow depending on whether a certain condition is true or not. This allows us to let our program behave differently in different situations. Furthermore we took a short detour to look at binary numbers.

Today we will deal with the implementation of repetitive behavior (loops). The lecture will cover (pre-test) loops with the `while` and `for` statements, and the special behavior of the `break` and `continue` statements. We will furthermore discuss the issue of post-test loops in Python and say a bit more about testing and debugging.

Next time we will look at functions, modules and packages, which are essential for well-structured code when programs become bigger and more complex.

Pre-test loops with the `while`-statement

While-loops are a straightforward way to implement pre-test loops in Python. Here is a simple countdown example:

In [1]:
```python
number = int(input("Please enter an integer number: "))
while number > 0:
    print(number)
    number = number - 1
```

```
Please enter an integer number:  10

10
9
8
7
6
5
4
3
2
1
```

While-loops have a simple basic form:

```
while <condition>:
     <do something>
```

That is, as long as (while) the condition is true, the body of the loop will be repeatedly executed. Just as in `if`-statements, the code forming the body of the loop is indented to mark that it is one block. Also like `if`-statements, while-loops can have an `else` block that is executed when the loop entry condition does not hold any more:

In [2]:
```python
number = int(input("Please enter an integer number: "))
while number > 0:
    print(number)
    number = number - 1
else:
    print("Ignition!")
```

```
Please enter an integer number:  10

10
9
8
7
6
5
4
3
2
1
Ignition!
```

Note that most other programming languages do not have the `else` block for `while` loops, and also in Python they are in practice not used very often.

If the condition is never changed through the execution of a loop, this results in an infinite loop. This is a common mistake when programming. For example, if we forget to decrease the number in our countdown example:

In [3]:
```python
number = int(input("Please enter an integer number: "))
while number > 0:
    print(number)
```

```
Please enter an integer number:  10
```

Or if the decrement happens only after the loop body:

In [ ]:
```python
number = int(input("Please enter an integer number: "))
while number > 0:
    print(number)
number = number - 1
```

Attention: In both these cases, the program will forever output the number the user entered. On the command line, pressing CTRL+C stops the execution of a program that is stuck in an infinite loop. In Jupyter notebooks, press the square Stop button (next to Run). In Spyder, both options work.

Sometimes it can be useful to work with infinite loops, though! For example, all software that is constantly running and ready to process user inputs (think for example of web servers) runs in a sort of infinite loop. We will see some cases later where it makes sense to use infinite loops.

Just as `if` -statements, also loops can be nested (contain other loops). For example:

In [4]:
```python
number1 = int(input("Enter an integer number: "))
number2 = int(input("Enter another integer number: "))

while(number1 > 0):
    orig = number2
    while(number2 > 0):
        print(f"{number1} + {number2} = {number1+number2}")
        number2 -= 1
    number1 -= 1
    number2 = orig
print("Done.")
```

```
Enter an integer number:  2
Enter another integer number:  3

2 + 3 = 5
2 + 2 = 4
2 + 1 = 3
1 + 3 = 4
1 + 2 = 3
1 + 1 = 2
Done.
```

The following implementation of the number-guessing game is a slightly more complex example of a while-loop (as will be explained in more detail in one of the next lectures, the first two lines generate a random number between 1 and 10):

In [6]:
```python
import random

number = random.randint(1,10)
number_guessed = False
print("Can you guess the number?")
while number_guessed == False:
    guess = int(input("Make a guess: "))
    if guess < number:
        print("You guessed too small!")
    elif guess > number:
        print("You guessed too big!")
    else:
        print("Yes!")
        number_guessed = True
```

```
Can you guess the number?

Make a guess:  5

You guessed too small!

Make a guess:  7

You guessed too small!

Make a guess:  9

You guessed too small!

Make a guess:  10

Yes!
```

The `break` -statement

The `break` -statement can be used to leave a loop before the loop condition becomes `False` . For example, we could extend the countdown example from above by a check every 5 steps if the user wants the countdown to go on or not:

In [8]:
```python
number = int(input("Please enter an integer number: "))
while number > 0:
    if number%5==0:
        go = input("Do you want to continue? (y/n) ")
        if go == "n":
            print("Countdown stopped.")
            break
    print(number)
    number = number - 1
else:
    print("Ignition!")
```

```
Please enter an integer number:  20
Do you want to continue? (y/n)  y

20
19
18
17
16

Do you want to continue? (y/n)  y

15
14
13
12
11

Do you want to continue? (y/n)  2

10
9
8
7
6

Do you want to continue? (y/n)  n

Countdown stopped.
```

Note that if the loop is left with a break, the else statement is not executed.

The `continue` -statement

Similarly, the continue statement causes the program to skip the rest of the current iteration of the loop, and just continue with the next round. For example, we might extend the countdown with a counter reset possibility as follows:

In [9]:
```
number = int(input("Please enter an integer number: "))
while number > 0:
    if number==3:
        reset = input("3 steps left. Do you want to reset? (y/n) ")
        if reset=="y":
            number = int(input("Please enter an integer number: "))
            continue
    print(number)
    number = number - 1
else:
    print("Ignition!")
```

```
Please enter an integer number:  10

10
9
8
7
6
5
4

3 steps left. Do you want to reset? (y/n)  y
Please enter an integer number:  15

15
14
13
12
11
10
9
8
7
6
5
4

3 steps left. Do you want to reset? (y/n)  n

3
2
1
Ignition!
```

Post-test loops in Python

In contrast to many other programming languages, such as Java or C, there are no built-in post-test loop constructs ("do-until" or "do-while" loops) in Python. Post-test loop behavior can be emulated in Python with code like shown below, using a deliberately infinite loop (while `True`) and a conditional break in the loop body:

In [10]:
```
while True:
    number = int(input("You have to enter the right number to stop me: "))
    if number == 0:
        break
```

```
You have to enter the right number to stop me:  2
You have to enter the right number to stop me:  3
You have to enter the right number to stop me:  4
You have to enter the right number to stop me:  0
```

Note that generally the use of break is not considered good coding style and should be avoided, but in particular cases such as this one it might just be useful.

Pre-test loops with the `for`-statement

We have seen while-loops as a way to implement pre-test loops in Python. While-loops evaluate arbitrary conditions to decide whether to enter to body of the loop (again) or not. For-loops are also pre-test loops, but they are particularly suitable for iterating over collections of objects, in a "for each element in the collection do…" style. Here is a simple example:

In [11]:
```
for i in range(1,11):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

The `range` function is used to obtain a so-called iterable range object comprising the numbers 1-10. The for-loop then goes through this iterable one by one, calling the next element `i` in each iteration and in the body of the loop simply printing the value of `i`. We will see more examples of iterable objects that can be used in for-loops next week. For now note that the basic form of a for-loop is:

```
for <element> in <iterable collection of elements>:
    <do something>
```

Note that for-loops can also be implemented as while-loops, but more “management code” is needed. The following code is equivalent to the example above:

In [12]:
```
i = 1
while i < 11:
    print(i)
    i = i + 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Because of their design, for-loops do not (easily) lead to unintended infinite loops, so it’s preferable to use them when suitable objects to iterate over are present. Especially for nested loops this can greatly improve the readability of the code.

An interesting class of iterables is string, that is, we can easily use for-loops to iterate over all characters in a string. The following example shows how the "Caesar Cipher" can be implemented in Python with the use of a for-loop:

In [20]:
```python
text = "Just a small piece of text without a special meaning."
print(text)
shift = 3
cipher_text = ""
for char in text:
    if char.isalpha():
        if char.isupper():
            shifted_char = ord('A') + (ord(char) - ord('A') + shift)%26
        else:
            shifted_char = ord('a') + (ord(char) - ord('a') + shift)%26
        cipher_text += chr(shifted_char)
    else:
        cipher_text += char
print(cipher_text)
```

```
Just a small piece of text without a special meaning.
Mxvw d vpdoo slhfh ri whaw zlwkrxw d vshfldo phdqlqj.
```

For the Caesar Cipher, all characters in the string are shifted by a given number of positions within the alphabet. With a shift of 3 like in the example, 'a' becomes 'd', 'b' becomes 'e', 'c' becomes 'f', and so on. When the end of the alphabet is reached, it continues at the beginning, that is, 'x' becomes 'a', 'y' becomes 'b', and 'z' becomes 'c'. Deciphering the text works just the other way round.

Other useful functions that we are using here are the `isalpha()` method to check if the string (character) consists exclusively of characters, the `isupper()` function to check if the string (character) is in upper case, and the `ord()` function that returns the ASCII number of the character given as argument. As letters have subsequent ASCII numbers (ranging from 65 to 90 for the uppercase letters and from 97 to 122 for the lowercase letters, to be precise), the Caesar Cipher can be done with simple arithmetic. The `chr()` function is used here as the counterpart to `ord()` , returning the character corresponding to an ASCII number.

By the way: Testing and Debugging

Testing and debugging probably consume more of a software developer's time than the actual programming. Software is tested to validate that is doing what it is supposed to do, and in particular that it is not doing anything that it is not supposed to do. To do this, it is executed with all sorts of possible inputs, especially those representing unusual or "corner cases" that are maybe not so obvious, and that a programmer might easily overlook. In summary we could just say: Testing is about trying to make software fail. Debugging then means to identify the cause of the failure and fix the code so that it does not fail any more. Note that for the development of big and complex software systems, there are also comprehensive testing and debugging frameworks available. For the projects in this course we will not need to use those, but no programming project is free of testing and debugging, so we will discuss some of the most important basics and pragmatics here.

Consider for example the following piece of Python code:

In [21]:
```python
number = input("Please enter an integer number: ")
if number%2 == 0:
    print(number + " is an even number.")
else:
    print(number + " is an odd number.")
```

```
Please enter an integer number:  12

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-e57d707dc3ed> in <module>
      1 number = input("Please enter an integer number: ")
----> 2 if number%2 == 0:
      3     print(number + " is an even number.")
      4 else:
      5     print(number + " is an odd number.")

TypeError: not all arguments converted during string formatting
```

Apparently, this code already fails if we enter a completely reasonable input, the number 12. From the output and error message we can see that the first line is executed correctly, and the failure happens in line 2. The error message suggests a type error, and indeed the input is only read as a string, so in the second line the program tries to execute a module operation on a string, and that doesn't work. We can easily fix that by adding a type cast to the first line, storing the input as an integer number:

In [22]:
```python
number = int(input("Please enter an integer number: "))
if number%2 == 0:
    print(number + " is an even number.")
else:
    print(number + " is an odd number.")
```

```
Please enter an integer number:  12

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-b5c187471a88> in <module>
      1 number = int(input("Please enter an integer number: "))
      2 if number%2 == 0:
----> 3     print(number + " is an even number.")
      4 else:
      5     print(number + " is an odd number.")

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Another type error, this time in line 3, caused by the use of the string concatenation operator (+) between a number and a string. Note that the same error would happen in line 5, we just have not executed it yet. In this case there are at least two possible ways to fix the error: convert the number into a string before the concatenation, or use an f-string to include the number in the output:

In [24]:
```python
number = int(input("Please enter an integer number: "))
if number%2 == 0:
    print(str(number) + " is an even number.")
else:
    print(f"{number} is an odd number.")
```

```
Please enter an integer number:  13

13 is an odd number.
```

Now we have fixed all (syntactical) errors, the program executes, and we know that at least the code is doing the correct thing for 12 as an input. Or course we cannot reasonably test it for all possible integer numbers, but a good tester would at least validate it with some odd numbers, negative numbers, and 0. Furthermore, what happens if the user does not enter a valid integer number, but, for example, a string or a floating-point number? Failures because of type mismatches are to be expected, of course. There is no single solution to this dilemma, but several possible approaches. For instance, you can tell the user what kind of input to expect, and let it be their problem if they enter something else. Or if you don't want the user to be confronted with the corresponding Python error message, then you could check their initial input and ask them to re-enter if it is something that is not convertible into an integer.

Here is another example. The code executes, but there is a semantic error:

In [27]:
```
number = int(input("Please enter a number: "))
print("Counting down in steps of 2...")
while number != 0:
    print(number)
    number = number - 2
print("0. Ignition!")
```

```
Please enter a number:  13
```

Oops! The code was working fine for 12 as an input, but with 13 does not stop at 0 and enters the negative numbers. **Abort execution** (CTRL-C or square stop button)! This results an error message that contains a KeyboardInterrupt. It points at line 4 of our code, but that is the point where the inter-rupt occurred, and not necessarily the source of the error. The other line of the error message points to some Python library and does not help us any further, either. So, think! To make it short, the problem lies in the condition of the while loop, which tests for equality to 0. If we decrement a positive, even number by 2, we will arrive at 0 at some point. But if we enter an odd or negative number, this is not the case. For simplicity, let us assume that the user will enter a positive integer. The easiest way to fix the problem is then to change the condition in the loop to testing if the number is greater than 0. If not, it is equal to zero or smaller, and the countdown is finished:

```
In [26]: number = int(input("Please enter a number: "))
         print("Counting down in steps of 2...")
         while number > 0:
             print(number)
             number = number - 2
         print("0. Ignition!")
```

```
Please enter a number:  13

Counting down in steps of 2...
13
11
9
7
5
3
1
0. Ignition!
```

As everything in programming, testing and debugging needs practice and the only way to develop your skills in that is to keep on coding – you will have a lot to test and debug then, for sure. Don't panic if you feel stuck, that happens to the most experienced programmer! In the end it will turn out to be very logical. Try to understand the error messages (if there are some) or why unexpected things are happening. Think through the program step by step to find the possible source of the failure. Try different inputs. Ask your fellow students, the teaching assistants, lecturers StackOverflow or Google for help if you cannot make sense of an error message. Take a break and start again with a fresh mind. Good luck! :)

Exercises

Please use Quarterfall to submit and check your answers.

1. Canteen Dish

Write a program that corresponds to the "Season your dish from the canteen" Activity Diagram example from the first week's lecture (pre-test loop). It should announce you the dish, ask you if it needs more salt (y/n), if yes add salt and ask again, and if not let you enjoy it. The output should be something like:

```
Hello, here is the dish of the day for you.
Does it need more salt? (y/n) y
I added some more salt.
Does it need more salt? (y/n) y
I added some more salt.
Does it need more salt? (y/n) n
Enjoy your meal!
```

2. Summing Up

Write a program that asks the user to enter an integer number n, computes the sum of all numbers from 1 to n, and prints the result. The output should look like:

```
Please enter an integer number: 5
The sum of all numbers from 1 to 5 is 15
```

3. The Bag of Marshmallows

Write a program that implements the “Eating a bag of marshmallows” example from the lecture on Activity Diagrams. It should tell the user to open the bag, eat a marshmallow, ask if there are more left, and repeat eating and asking until the bag is empty. Then the bag should be put into trash. The output should be something like: Open bag of marshmallows.

```
Eat marshmallow.
More marshmallows left? (y/n) y
Eat marshmallow.
[...]
More marshmallows left? (y/n) n
Dispose of bag.
```

4. Temperature Conversion Revisited

Extend the temperature conversion program from last week so that it asks the user to enter both a temperature value and the unit of the temperature (Celsius or Fahrenheit), and calculates the temperature in the respecive other unit. If an incorrect unit is entered, the program should give an error message and ask the user to try again. Furthermore, after having done a conversion, the program should ask the user if they want to convert another temperature value. The output should be something like:

```
Please enter temperature: 54
Is the temperature in Celsius (c) or Fahrenheit (f)? c
54.0 degrees Celsius is 129.2 degrees Fahrenheit.
Do you want to convert another temperature value? (y/n) y
Please enter temperature: 13
Is the temperature in Celsius (c) or Fahrenheit (f)? k
Unknown temperature unit, try again.
Please enter temperature: 13
Is the temperature in Celsius (c) or Fahrenheit (f)? c
13.0 degrees Celsius is 55.4 degrees Fahrenheit.
Do you want to convert another temperature value? (y/n) n
Okay, goodbye!
```

5. Text Analysis

Using what you have learned in the course so far, write a simple text analysis program that finds the (first) longest word in a text. It should work on any text, but you can use the “lorem ipsum” as an example:

```
# some random text
text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do \
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad \
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
 \
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate \
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
 \
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id \
est laborum."
```

To check if a character c is an alphabetic character, you can call the isalpha() function on it: `c.isalpha()` . It will return True or False.

The output of your should report the longest word and its length, like this:

```
The longest word in the text is "reprehenderit" (13 characters).
```

In [ ]:

```

```