08-while-loops

February 6, 2022

0.1 While Loops

While loops are usually used when we do not know in advance how many times we want to iterate, or if we want to stop as soon as some condition is reached. The syntax of while loops is:

```
while <condition>:
    <steps>
```

The condition is any expression that can be true or false (i.e. expressions that evaluate to a boolean). This is the same as you use on `if` and `elif`.

The way while loops work is:

1. Check the condition
    a. If it is true: run the steps, go back to 1
    b. If it is false: go to the first line after the loop (starting at the same column as while)

Note: If the condition is always true, your program will run forever! This is what is called an *infinite loop*.

Typically the loop condition will use a variable that is modified inside the loop, for example:

[1]:
```
x = 0
sum = 0
while x < 1000:
    sum = sum + x
    x = x + 1       # Increases x, so it will reach 1000 eventually
```

Be careful about how the variable is changed.

[2]:
```
#x = 0
#sum = 0
#while x < 1000:
#    sum = sum + x
#    x = x - 1

# Infinite loop.
```

Similar to for-loops, if the program flow hits a return in the middle of an iteration, it simply returns and does not run more iterations.

The code below, in particular, only runs the first iteration, and then stops.

[3]:
```
def doesNotLoopTechnically(n):
    while(n < 100):
        if n % 10 == 0:
            return True
        else:
            return False
```

0.1.1 Exercise 1

Implement the function `numberOfDigits(n)` that returns the number `n`. For example `numberOfDigits(15110)` should return 5.

[4]:
```
def numberOfDigits(n):
    d = 0
    while n > 0:
        n = n // 10;
        d = d + 1

    return d

def numberOfDigits(n):
    p = 0
    while (n % (10 ** p)) != n:
        p = p + 1

    return p

numberOfDigits(123456789)
```

[4]: 9

0.1.2 Exercise 2: population increase

You are studying two populations A and B. You know that population A is initially smaller than population B, but it grows at a faster rate. You would like to know haw many days it would take for population A to overtake population B. Luckily you have taken 15-110, and you know it would be faster to sit down and implement a program to compute that for you, than having to do the math each time (you work with a lot of populations of many different organisms…).

Implement the function `populationIncrease(pa, pb, ga, gb)` that takes as parameters:

- The number `pa` of individuals in population A
- The number `pb` of individuals in population B
- The growth rate `ga` of population A (in percent per day)
- The growth rate `gb` of poplation B (in percent per day)

This function should return the number of days it takes for population A to overtake population B.

Important: Populations grow by an integer number of individuals. So if the growth rate is 3.6% and the population is 100, in one day there will be 103 (not 103.6) individuals. If the population is 1000, there will be 1036 individuals.

[5]:
```
import math
def populationIncrease(pa, pb, ga, gb):
    return 42

# assert(populationIncrease(100, 150, 1.0, 0) == 51)
# assert(populationIncrease(90000, 120000, 5.5, 3.5) == 16)
# assert(populationIncrease(56700, 72000, 5.2, 3.0) == 12)
# assert(populationIncrease(123, 2000, 3.0, 2.0) == 300)
# assert(populationIncrease(100000, 110000, 1.5, 0.5) == 10)
# assert(populationIncrease(62422, 484317, 3.1, 1.0) == 100)
```

0.1.3 Exercise 3: the straggler (again)

In motorsports it is very common that the leader pilot in a race, at a certain moment, overtakes the last pilot. The leader, at this moment, is one lap ahead of the last pilot, who then becomes a straggler (or tailender).

Implement the function `straggler(fast, slow)` that takes as input the time it takes for the faster and slowest pilots to complete a lap, and returns the number of the lap when the fastest pilot overtakes the slowest one. The first lap is number 1.

For example, `straggler(5, 7)` should return 4: - Lap 1: Fastest crosses the start line at 5, slowest is 2 seconds behind - Lap 2: Fastest crosses the start line at 10, slowest is 4 seconds behind - Lap 3: Fastest crosses the start line at 15, slowest is 6 seconds behind - Lap 4: Fastest crosses the start line at 20, slowest is 8 seconds behind.

Hint: Can you solve this by simulating the race?

[6]:
```
def straggler(fastest, slowest):
    return 42
```