Lab 05 Conditionals and Loops

General Information

What you should do

- You should first review Lecture 3 and be familiar with the concepts.
- You should try to come up with the code yourself as much as possible. Do not be afraid of making mistakes, since debugging (finding out where your code goes wrong and fixing it) is part of the learning process.
- We do not give out model programs to the exercises. There can be multiple ways to write the code that solves the same problem. It is important that you build up the program logic yourself instead of merely looking at some code that you do not understand. At any time if you are lost or if you have any questions, feel free to ask the instructor, tutor, or teaching assistant and we will be very happy to help you.

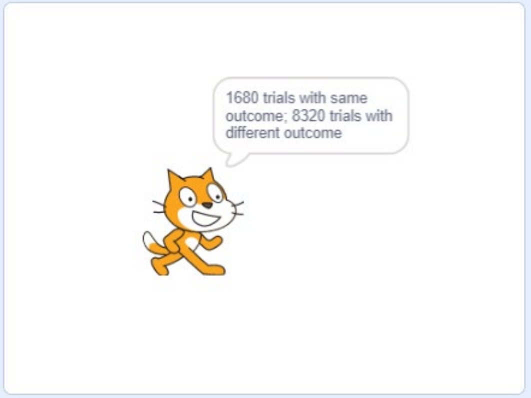
Self-Discovery

- Most lab tasks are designed to be relatively simple such that you can take the time to think about the related underlying concepts. Besides, we also encourage you to discover things on your own which may not be specified in the tasks.

Task 1.1 Simulate dice throws

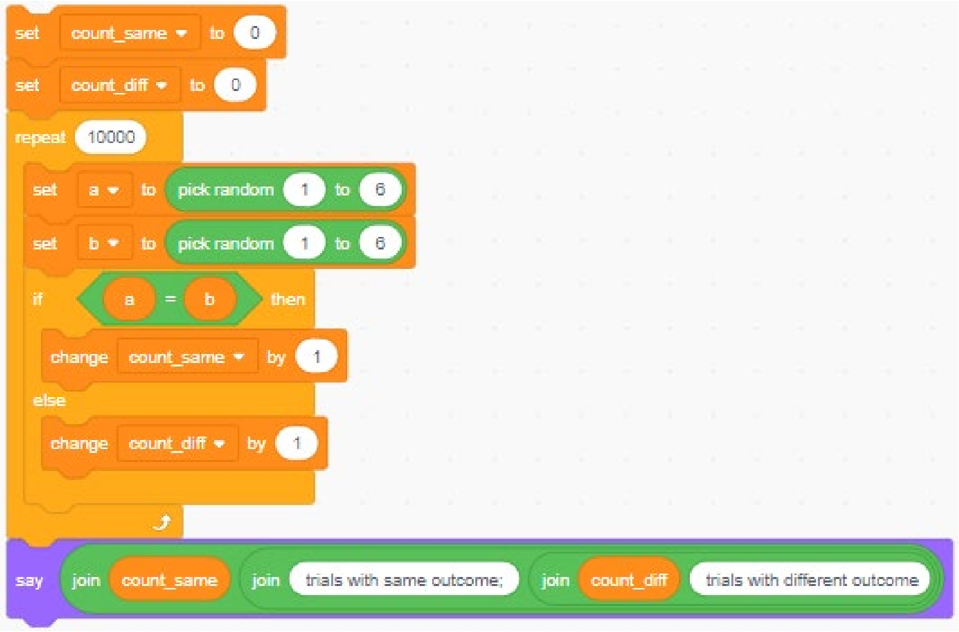
In this task, we will simulate normal dice (i.e., with 6 faces) throws. At each trial, we will randomly generate the outcome of 2 dice throws and then check if they are equal or not. We will repeat this process for 10,000 trials and then output the number of times with equal dice outcomes and the number of times with different dice outcomes. Try to write this program by yourself according to the specifications. If you have trouble writing it, then read the following hints:

1) As you need to store the number of times with equal dice outcomes and the number of times with different dice outcomes, create two variables *count_same* and *count_diff* for this purpose. These variables should be initialized as 0 at the beginning.
2) Use a finite loop that runs for 10000 iterations
3) Inside the loop, generate two random integers *a* and *b*, each in the range from 1 to 6. Then check if they are equal. If yes, then increase *count_same* by 1. Otherwise, increase *count_diff* by 1. You can use a two-way conditional for this.
4) Display the result after the loop.



```
1680 trials with same outcome; 8320 trials with different outcome
```

Theoretically, the chance for getting the same dice throw is 1/6 so with 10,000 trials, it should be around 1,667 trials with the same outcome. The following program shows an example about how the simulation can be implemented according to the above specifications:



```
set count_same to 0
set count_diff to 0
repeat 10000
    set a to pick random 1 to 6
    set b to pick random 1 to 6
    if a = b then
        change count_same by 1
    else
        change count_diff by 1
say join count_same join trials with same outcome; join count_diff trials with different outcome
```

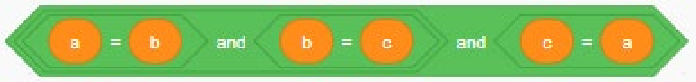
Exercise 1.1

a) At each trial, check if the sum of the two dice throws is smaller than 7, larger than 7, or equal to 7. At the end, display the number of times in each case. (Hint: Create 3 count variables and use a 3-way conditional to update them in the loop)

b) At each trial, simulate 3 dice throws (instead of 2) and denote them as a, b, c. Count the number of occurrence under the following conditions:
   i) The outcomes of 3 dice throws are the same
   ii) The outcomes of 3 dice throws are all different
   iii) At least the outcomes of 2 dice throws are the same
   iv) At least the outcomes of 2 dice throws are different

   Hint 1: Use the AND/OR operator(s) to join the proper Boolean expressions to represent each condition

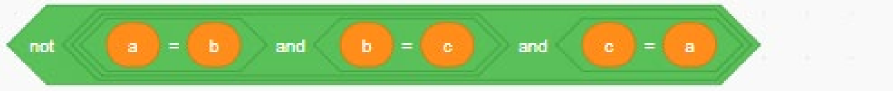
   Hint 2: For i), the following Boolean expression would return true if the outcomes of 3 dice throws are the same, and return false otherwise:



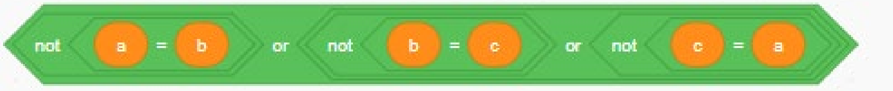
```
a = b and b = c and c = a
```

However, is it really necessary to use all the 3 parts in the above Boolean expression?

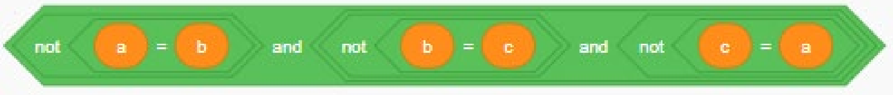
   Hint 3: For ii), which of the following Boolean expressions correctly represents that case, i.e., return true if the outcomes of 3 dice throws are all different and return false otherwise?



```
not a = b and b = c and c = a
```



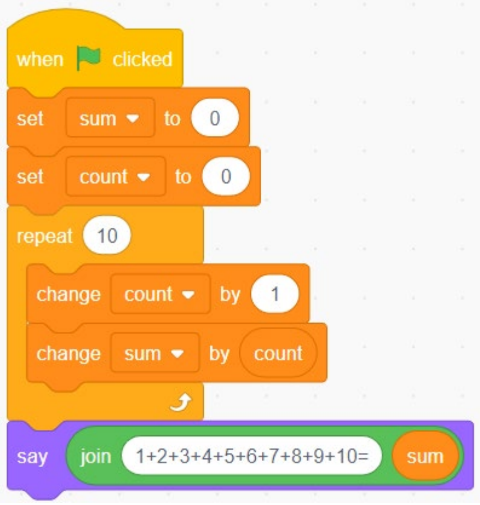
```
not a = b or not b = c or not c = a
```



```
not a = b and not b = c and not c = a
```

Task 1.2 Add up a series of numbers

In Lecture 3 we have shown you the following program that adds up integers from 1 to 10:



```
when 🏳 clicked
set sum to 0
set count to 0
repeat 10
    change count by 1
    change sum by count
say join 1+2+3+4+5+6+7+8+9+10= sum
```

In this task we will practice adding other series of numbers. In particular, let us write a program to sum up all even numbers (i.e., 2, 4, 6, 8, 10, …) from 1 to 100. There are different ways to compose the program that can achieve the same function and we will illustrate a few approaches.

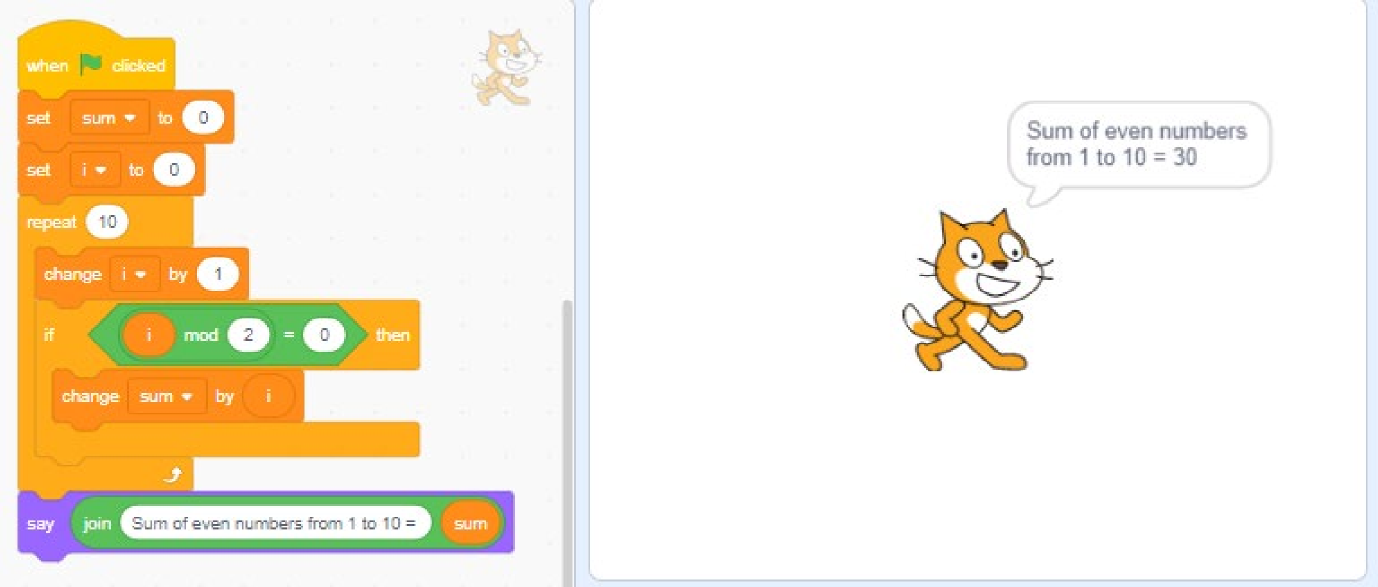
Approach 1

- Use a finite loop iterate over all integers from 1 to 100
- For each integer check if it is an even number, and add it to the sum if it is

For the first point above, we can just use the same repeat block as in the above code and change from 10 to 100. However, for now, let us just keep it as 10 such that we first calculate the sum of even numbers from 1 to 10. When you first write a loop, it will be a good idea to first try with a small number of iterations so that it is easier to analyze and test the program. You can easily calculate by hand that the sum of even numbers from 1 to 10 is 2+4+6+8+10=30. You can check that your program works with this small number, and then change it to the desired big number, i.e., from 10 to 100 afterwards.

For the second point above, you need to first check if a number (let’s denote it by *i*) is even. An even number means that it is divisible by 2. Recall from Lab 04 Task 1.2 that you have learnt to use the modulus function to check if an integer is divisible by another integer, i.e., when the modulus function returns 0. You can thus use a one-way conditional to check if a number is even, and if so add it to the sum.

You can thus come up with the following program that works according to the above specifications:

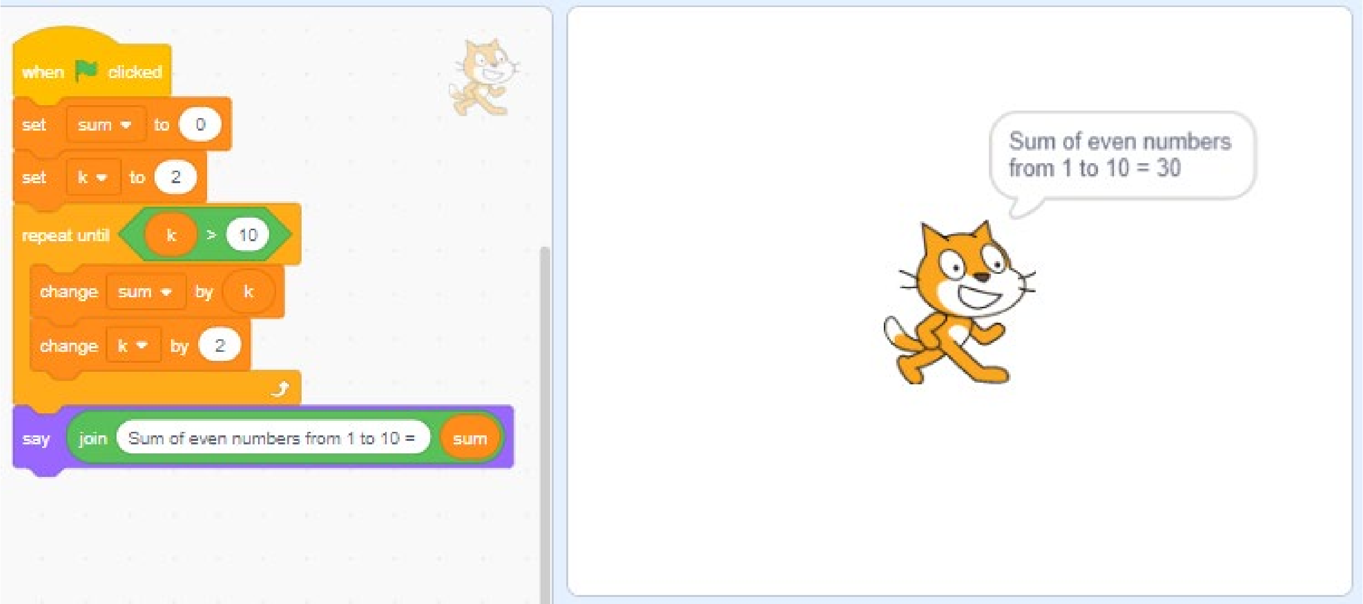


```
when 🏳 clicked
set sum to 0
set i to 0
repeat 10
    change i by 1
    if i mod 2 = 0 then
        change sum by i
say join Sum of even numbers from 1 to 10 = sum
```

Approach 2

- Start with the first even number 2
- Use a loop with termination condition such that inside the loop, an even number is added to the sum and then the even number is increased by 2. The termination condition is when the even number is outside the range

Let's denote the even number to be added by k. The following program that works according to the above specifications:



```
when [flag] clicked
set sum to 0
set k to 2
repeat until k > 10
    change sum by k
    change k by 2
say join Sum of even numbers from 1 to 10 = sum
```

Sum of even numbers from 1 to 10 = 30

Note that in this case we do not need a conditional inside the loop to check whether a number is even because we know for sure that it is. Now think about why the termination condition is set as $k > 10$ (and think about why it would not be correct to be set as $k \geq 10$).

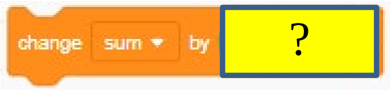
For each of the two approaches, after you verify that it is working properly and yield the correct sum of even numbers of 1 to 10, i.e., 30, then you can change the number from 10 to 100 to let your program calculate the sum of even numbers from 1 to 100, which should be equal to 2550.

Exercise 1.2:

a) Modify the above program so that it calculates the sum of odd numbers (i.e., 1, 3, 5, 7, 9, …) from 1 to 100. Try implementing it using each of the two approaches.

b) Write a program that adds the sum of squares from 1 to 100, i.e., $1^2+2^2+3^2+…+99^2+100^2$

Hint 1: What should be the expression when you update the variable *sum* in the loop?

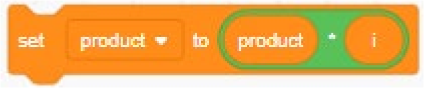


```
change sum by ?
```

Hint 2: Numerical value = 338,350

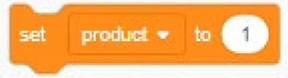
c) Write a program that calculates the factorial from 1 to 20, i.e., their product 1×2×3×…×19×20

Hint 1: Since you are calculating the product instead of the sum, you cannot use the change block which adds the expression on the right hand side to the variable on the left hand side. Instead, you should use the set block to update the variable, e.g.,



```
set product to product * i
```

Hint 2: Again, since you are dealing with a product by multiplying a series of integers (instead of summing them up), the initial value of the variable that stores the product should NOT be 0, otherwise it will always be 0 no matter what values you are multiplying it with. Instead, it should be initialized as 1 before the loop.



```
set product ▾ to 1
```

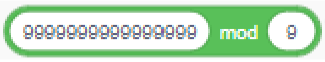
Hint 3: Numerical value = 2,432,902,008,176,640,000

d) Write a program that will ask the user to input two integers. Your program should then check that each input is indeed an integer. After verifying that both user input are integers, your program should find the smaller of the two integers to be stored as variable a, and find the bigger of the two integers to be stored as variable b. Then your program should sum up all integers from a to b and display the result.

Task 1.3 Check if a large number is divisible by 9

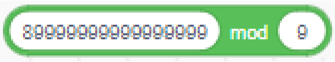
Recall that the modulus function can be used to find the remainder when an integer is divided by another integer. If the remainder is 0, then the dividend is divisible by the divisor. Form the following blocks and click on each block to see the result. Do you expect each number to be divisible by 9, i.e., do you expect the remainder to be 0 or not?

- 9999999999999999 mod 9 (16 9's)



```
9999999999999999 mod 9
```

- 89999999999999999 mod 9 (an 8 followed by 16 9's)



```
89999999999999999 mod 9
```

You should be able to tell by inspection that the first number (9999999999999999) should be divisible by 9 but your program does not say so. On the other hand, you should also be able to tell by inspection that the second number (89999999999999999) should not be divisible by 9 but your program says otherwise. This problem arises because the input contains a very large number but there is a limit on the maximum positive integer that Scratch can represent in calculation. For a large number that goes beyond the maximum positive integer limit, Scratch cannot represent it with full precision meaning that the resulting number represented by Scratch is only an approximation of (thus different from) the original number. For example, when you are told that the current population of Hong Kong is 7,511,174, your brain is not able memorize all the digits so you may remember it as 7.51 million or 7.5 million. This is similar to how Scratch represents large integers in calculation.

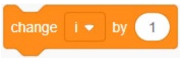
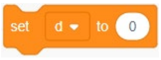
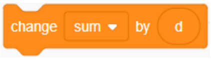
We need to find another way to determine whether a large positive integer is divisible by 9. One way to do this is to first add all the digits in the number and then check whether the resulting sum is divisible by 9. A number would be divisible by 9 if the sum of all its digits is divisible by 9 and vice versa. For example,

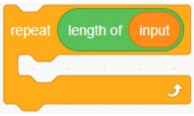
- 18 is divisible by 9. The sum of its digits, i.e., 1+8=9, is also divisible by 9
- 324 is divisible by 9 (324/9=36). The sum of its digits, i.e., 3+2+4=9, is also divisible by 9
- 37 is not divisible by 9 (37/9=4…1). The sum of its digits, i.e., 3+7=10, is not divisible by 9
- 813 is not divisible by 9 (813/9=90...3). The sum of its digits, i.e., 8+1+3=12, is not divisible by 9

Now we will implement this algorithm in Scratch. In particular, we will write a loop that runs a finite number of times depending on how many digits there are in the given input. At each iteration of the loop, we will take one digit from the number and add its value to the sum. In designing a loop that runs a finite number of times, the following 3 issues should be considered:

1) **Initialization of variables**: we should set proper values to the variables before the loop starts. Often one of the variables is used for counting and its value will be incremented after each iteration.
   In this problem, create a few variables and initialize them as specified below:
   a) `input` which is used to store the user input. This variable should be initialized to the answer block .
   b) `sum` which is used to store the sum of digits. This variable should be initialized to 0.
   c) `i` which is used for counting. This variable should be initialized to 0.

2) **Loop actions**: we should define what actions should be taken by the program at each iteration. Often the counting variable is incremented either at the beginning or at the end of the iteration.
   In this problem, a digit should be extracted from the input and added to the sum at each iteration. The block can be used to extract the `i`-th letter (left-to-right order) of a text thus it can be used to extract the `i`-th digit when the input is a number. At iteration 1, the 1st digit (thus letter 1) should be extracted. At iteration 2, the 2nd digit (thus letter 2) should be extracted. At iteration `i`, the `i`-th digit (thus letter `i`) should be extracted, where `i` should start with 1 and increments by 1 each time. Since `i` is initialized to be 0, it should be incremented by 1 first before using its value to extract the `i`-th letter from the input. Form your loop actions by following the steps below:

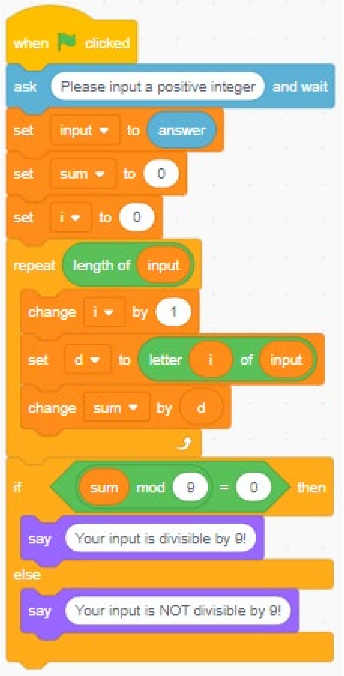
   a) Increment the counting variable `i` by 1 
   b) Create a variable `d` and set it to be the `i`-th digit of the input. (Hint: assemble these blocks )
   c) Add the digit `d` to the sum 

3) **Loop ending**: we should specify how many iterations the loop should be repeated. This is either a constant number or with value specified by a variable.
   In this problem, as we need to loop the actions over every digit which is to be added to the sum, the number of iterations is thus the number of digits present in the input number. The block returns the length of a text, i.e., the total number of characters (including letters, digits, space, etc.). Thus when the input is a number, this block returns the total number of digits in the input number. Use the repeat block with the correct parameter for specifying the number of iterations: 

Now try to combine the above parts to finalize your program such that your program will

- ask the user to enter a positive integer
- use a loop to calculate the sum of digits of the input number
- output whether the input number is divisible by 9 or not (by checking whether the sum of digits is divisible by 9)

Test your program by using small numbers (e.g., 5, 9, 18, 20, etc.) as input and check whether it is outputting the correct message in each case. Then test it again by using large numbers (e.g., 9999999999999999, 89999999999999999, as indicated at the beginning of this task) to check that it is still working correctly.



```
when 🏴 clicked
ask (Please input a positive integer) and wait
set input ▾ to (answer)
set sum ▾ to (0)
set i ▾ to (0)
repeat (length of (input))
    change i ▾ by (1)
    set d ▾ to (letter (i) of (input))
    change sum ▾ by (d)
end
if <((sum) mod (9)) = (0)> then
    say (Your input is divisible by 9!)
else
    say (Your input is NOT divisible by 9!)
end
```

Exercise 1.3: Modify your program according to each of the following specifications:

a) Check if the input number is divisible by 3, which can be achieved by the same approach as the above method, i.e., summing up all digits of the input number and check if it is divisible by 3.

b) Check if the input number is divisible by 2, which can be achieved by checking only the ones' place digit, i.e., the rightmost digit to see if it is an even number.

c) Check if the input number is divisible by 6, which can be achieved by checking if it is divisible by both 3 and 2.

d) Check if the input number is divisible by 5, which can be achieved by checking only the one's place digit, i.e., the rightmost digit to see if it is 0 or 5.

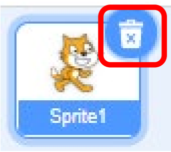
e) Check if the input number is divisible by 4, which can be achieved by checking the rightmost 2 digits to see if it is divisible by 4.

f) Check if the input number is divisible by 10.

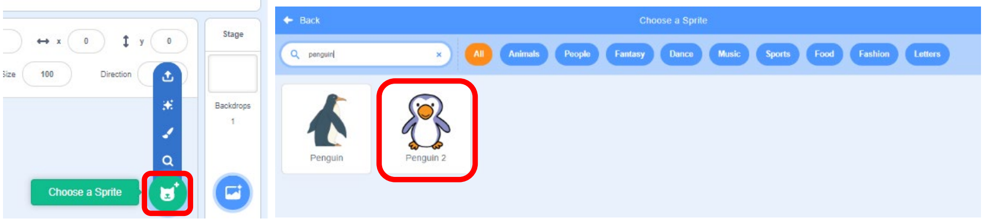
Task 1.4 Form a line of penguins

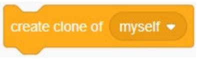
In the menu, choose File > New to create a new project and update the project name as Lab 05 Task 1.4. In this task we will try to create a loop for laying out a number of sprite clones in a single line.

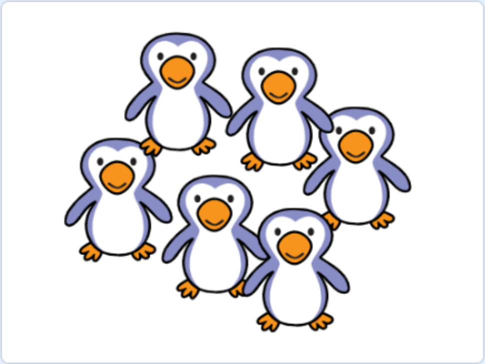
1) Delete the Scratch cat sprite as we will not need it.



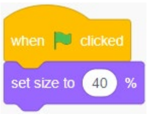
2) Import a sprite from the library by clicking on the blue cat icon:



3) Double click on the block  create clone of myself. At first it seems that nothing has changed but when you drag the sprite, you will find that there are 2 penguin sprites on the stage area. One of them is the original and the other one is the clone that has been created by double clicking on that block. You can double click this block a few times to create more clones. All clones will be destroyed if you click on the stop button ●.

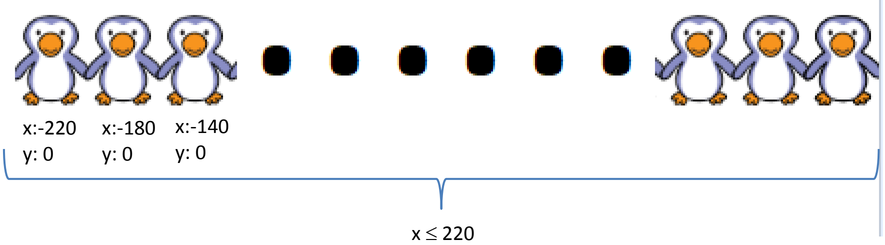


4) The penguin is a bit too big and we would like to show a number of penguins so we want to reduce its size when the program starts with the following blocks:



```
when 🏁 clicked
set size to 40 %
```

5) We want to create some clones of the penguin and align them in a single line with the following specifications about the coordinates of the clones:



Now we will write a program by creating a loop to perform the above task. You can observe some patterns from the above illustration which will help you design your loop. Firstly you can see that the penguins are in a horizontal line such that their y-coordinates are the same and equal to 0. Secondly you can see that the penguins are equally spaced horizontally with a difference of 40 in the x-coordinates of neighboring penguins. The condition x ≤ 220 means that the coordinates of the rightmost penguin cannot exceed 220. Now create the loop by considering the following 3 issues:

a) Initialization of variables

- `x0` stores the x-coordinate of the leftmost penguin. `x0` should be initialized to -220.
- `xmax` stores the upper bound of the x-coordinate of the rightmost penguin. `xmax` should be initialized to 220.
- `dx` stores the difference in x-coordinates of neighboring penguins. `dx` should be initialized to 40.
- `x` stores the x-coordinate of each penguin clone. `x` should be initialized to `x0`.
- `y` stores the y-coordinate of each clone. `y` should be initialized to 0.

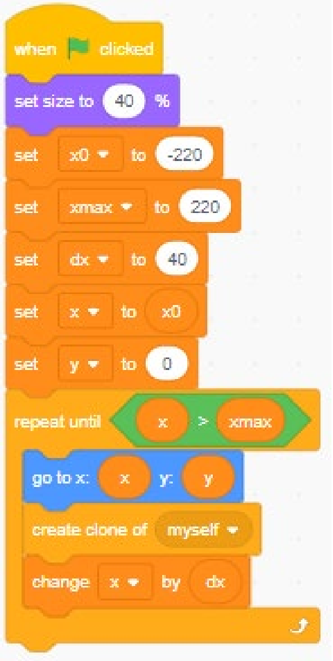
b) Loop actions

At each iteration we will move the penguin sprite to each location specified in the above figure using the block `go to x: ( ) y: ( )` with the parameters defined by the variables x and y. Then we will create a clone using the block `create clone of (myself ▾)`. Afterwards we need to increase the value of x so that the next penguin clone will appear in the correct position on the right. Now compose your loop actions with the above 2 blocks as well as these blocks `change (x ▾) by ( )` `(dx)` `(x)` `(y)`

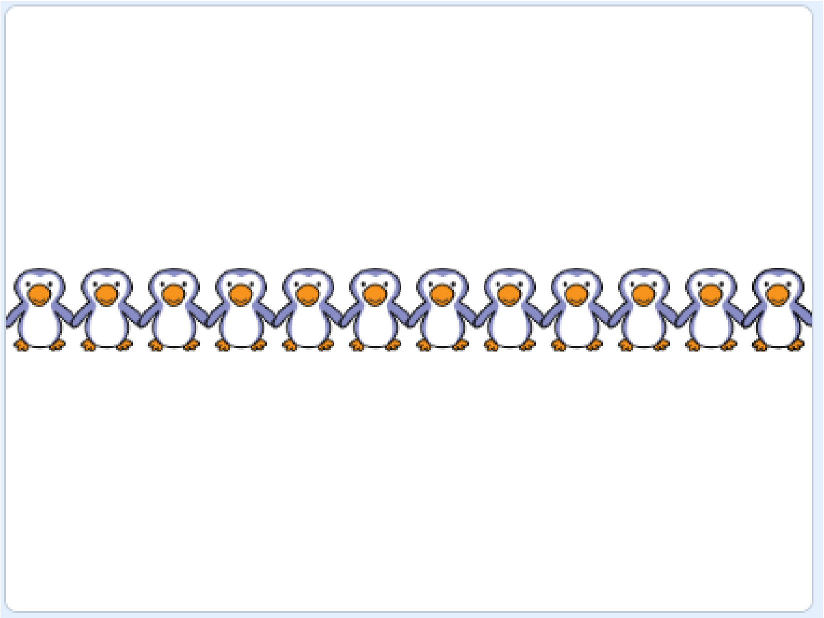
c) Loop ending

You see that the value of x is increased at each iteration. You also know that there is a condition x ≤ 220 (xmax) for all x-coordinates of all penguins. This condition can be used to derive the termination condition that the loop should end if x > 220.

Assemble the blocks to complete your program so that when the program starts you will see a list of penguins aligned horizontally as specified in the above figure.

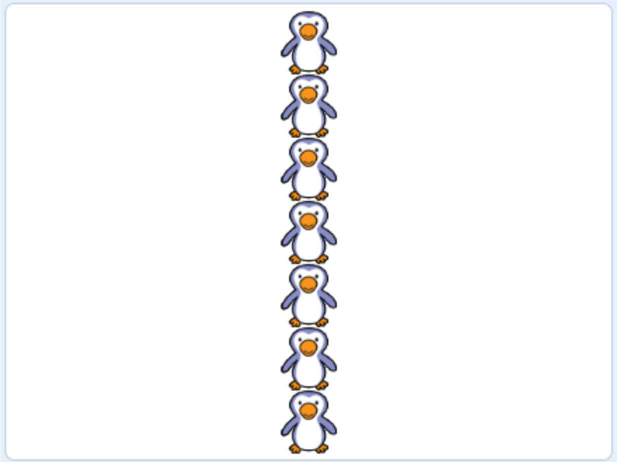


```
when 🏴 clicked
set size to (40) %
set (x0 ▾) to (-220)
set (xmax ▾) to (220)
set (dx ▾) to (40)
set (x ▾) to (x0)
set (y ▾) to (0)
repeat until <(x) > (xmax)>
    go to x: (x) y: (y)
    create clone of (myself ▾)
    change (x ▾) by (dx)
```



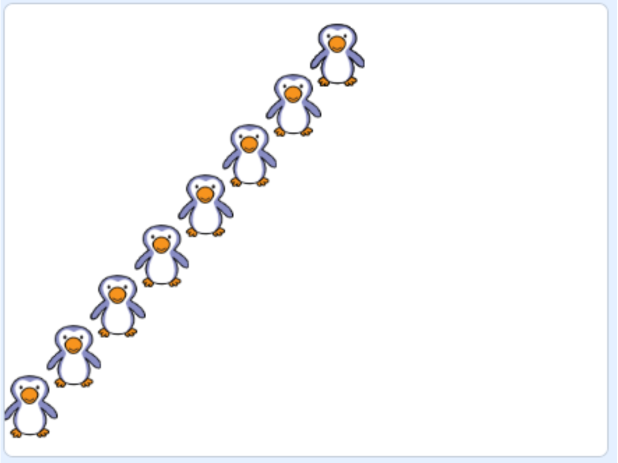
Exercise 1.4: Modify your program so that it will generate different patterns as each of the following cases:

1) Vertical line



Hint 1: the x-coordinates of the penguins should be the same
Hint 2: the y-coordinates of the penguins should be changed in the loop

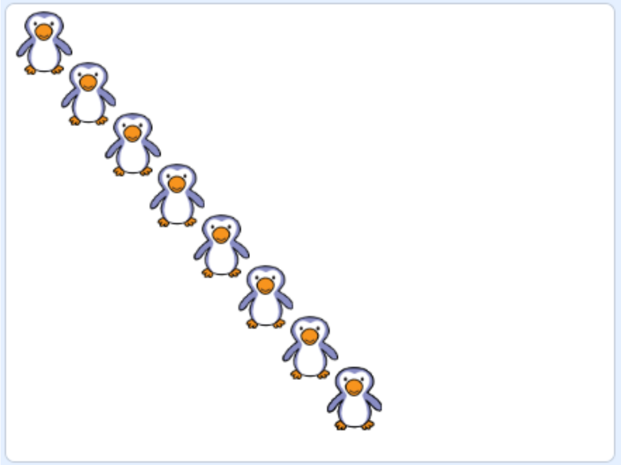
2) Bottom left to top right



Hint 1: the first penguin is on the bottom left of the stage area so its x-coordinates and y-coordinates should be the smallest among all penguins

Hint 2: both the x-coordinates and y-coordinates of the penguin should be increased in the loop

3) Top left to bottom right



Hint 1: the first penguin is on the top left of the stage area so its x-coordinates should be the smallest while the y-coordinates should be the largest among all penguins

Hint 2: the x-coordinates of the penguin should be increased but its y-coordinates should be decreased in the loop

Task 2 Complete the assessment from the Canvas course page

You should complete the Lab 05 Assessment from the Canvas course page before the posted deadline.

Task 3 Challenge your classmates

You can first reflect on what you have learnt in this lab, and then come up with problems to challenge your classmates. You can post your problem on the Canvas course page, under this Discussion page. One should be able to solve your problem by using what he/she learns in Lab 05. You will not get extra marks by posting a challenging problem or solving a challenging problem posted by another student, but you will earn your fame so that you can impress the course leader, the lab tutors, and your classmates.