control structure:

We execute the statements one after another until we come to the end of the program. This concept is called sequential control.

We may want to execute some statements repeatedly.
This is called a loop.

We may want to select an alternate set of statements.
This is called selection.

Python supports two looping structures.
a) while
b) for
Python supports one selection structure
a) if - with or without else

We want to indicate that the following statements form a group and are to be executed repeatedly in a while or based on selection. We require some mechanism for grouping. In some languages, we have keywords like begin and end. In some, we use flower brackets or braces. In Python, the grouping is controlled by indentation.

The structure is called the leader suite(body) combination.
Leader ends in a : and the suite (which could have one or more statements) will have higher indentation compared to the leader.

leader:
    suite

Let us examine a few portions of this code.

```
# file: 1_loop.py

# do something repeatedly
# add numbers from 1 to n
# 1 + 2 + 3 + 4 + ... + n

# sequential control:
#       # of statements
#       executed one after the other

# loop control
#       # of statements
#       repeated executed

# selection control
#       groups of statements
#       execute one of the groups

# loop:
#       while statement
#       syntax(grammar)
#       while <expr> :
#               <stmt>
#               <stmt>
#               ...

#       while it is raining
#               I will keep waiting
#       I will go home

#       while I am thirsty
#               I will keep on drinking water

# expr of while : true or false
```

```
# display 1 2 3 until n
"""
# Test 1
# wrong program; infinite loop
n = 5
i = 1
while i <= n :
    print(i)
i = i + 1
print("happy tests")
"""

# there are statements which have # of statements within them
# first statement is like a leader of the group
#       rest are part of it
# to indicate this, leader is the first statement, end with :
#       group of statements under the leader : suite or block

# leader :
#       suite
# leader has a colon
# suite should have higher indentation compared to the leader

"""
# Test 2
# print is not a leader
# putting : after print will be an error
# statement following print cannot have higher indentation
n = 5
i = 1
while i <= n :
    print(i)
       i = i + 1
print("happy tests")
"""
```

```
# ok
"""
# Test 3
n = 5
i = 1
while i <= n :
    print(i, end = " ")
    i = i + 1
print()
print("happy tests")
"""

# error
# when we decrease the indentation, it should match one of the earlier
statements
"""
# Test 4
n = 5
i = 1
while i <= n :
    print(i, end = " ")
    i = i + 1
  print()
print("happy tests")
"""
```

Let us examine a few cases.

 # Test 1

Incrementing i is not part of the suite of the leader while.

We keep executing the loop for ever as the condition of the while loop will always be True.

```
# wrong program; infinite loop
```

```
n = 5
i = 1
while i <= n :
    print(i)
i = i + 1  # never reached
print("happy tests")
```

 # Test 2

We cannot increase the indentation after a print as print is not a leader.
This results in indentation error.

```
# print is not a leader
# putting : after print will be an error
# statement following print cannot have higher indentation
n = 5
i = 1
while i <= n :
    print(i)
       i = i + 1
print("happy tests")
```

```
# Test 3
# This is ok. As long as i is less than n, the body of the while will be executed.
In that iteration, i will be displayed and then incremented. Ultimately, i <= n
becomes false. Then the loop is executed.
n = 5
i = 1
while i <= n :
    print(i, end = " ")
    i = i + 1
print()
print("happy tests")
```

```
# Test 4
# This is an error. We can decrease the indentation of a statement and make it match the indentation of one of the earlier statements. We cannot hang print in the middle!
n = 5
i = 1
while i <= n :
    print(i, end = " ")
    i = i + 1
  print()
print("happy tests")
```

While loop:

We will try a few examples using loops.

example 1:

Generate a geometric progression until the term exceeds a given number. Display the numbers. Also find the total.

We start with the initial term t = 1 and common ratio r = 2
We repeat getting the new term by multiplying t by r and putting it back in t.
We repeat until t exceeds n.
We also start a summation with an initial value 0 - identity element of addition.
We add t to the total each time in the loop.

```
# file: 2_loop.py
# display 1 2 4 8 ... until n
# find the total
n = 100
t = 1
r = 2
total = 0
```

```
while t <= n:
        total = total + t
        print(t, end = " ")
        t = t * r
print()
print("total : ", total)
```

```
$ python 2_loop.py
1 2 4 8 16 32 64
total :  127
```

example 2:

Reverse a given number.

Take an input n.

Assume that the reversed number rev is 0.

As long n is greater than 0, extract the unit digit by using n modulo 10.

Shift rev number to the left in decimal by multiplying with the base 10 and then add this digit to it.

This is how the variables change in the loop.

N : 1729 172 17 1 0

rev : 0 9 92 927 9271

```
# file: 3_loop.py
# reverse a given number
n = int(input("enter an integer : "))
rev = 0
while n :
        rev = rev * 10 + n % 10
        n //= 10
print("reverse number : ", rev)
```

```
$python 3_loop.py
enter an integer : 1729
reverse number :  9271
```

---

selection:

In looping, we execute the body repeatedly.

In selection, we choose one of the alternates.

This program checks whether the given two strings are same.

```
# file: 4_selection.py

# selection
# syntax :
# if <expr> :
#       <suite>

# or
# if <expr> :
#       <suite>
# else:
#       <suite>

# check whether given two strings are same
a = input("enter string 1  ")
b = input("enter string 2  ")

# check this out!
#if a = b :  #assignment is not an expr
#       print("equal")

if a == b :
        print("equal")
else:
        print("not equal")
```

```
$ python 4_selection.py
enter string 1  pes
enter string 2  pes
equal

$ python 4_selection.py
enter string 1  pesit
enter string 2  pesu
not equal
```

 # example 2:

Find the order of strings

Check both the versions.
In the first version, there are 3 separate comparisons.
In the second version, we use the knowledge of earlier comparisons.
This uses elif (else if) and then else (with no comparison) to make
the code better than the first version.

```
# file: 5_selection.py
# compare two strings; find the order of the given strings.
a = input("enter string 1 : ")
b = input("enter string 2 : ")
"""
# version 1
if a == b :
        print("equal")
if a < b :
        print(a , " before ", b)
if a > b :
        print(a, " after ", b)
"""
```

 # version 2

```
if a == b :
        print("equal")
elif a < b :
        print(a , " before ", b)
# elif : not required here
#elif a > b :
else:
        print(a, " after ", b)
```

Dangling else problem:
If we have two ifs and a single else, to which if should we associate the else. This is called dangling else problem. In Python, else matches the if with the same indentation as else.

The code with comments is self evident.

file: 6_selection.py

```
# dangling else problem
#       two if and single else
#       else is paired with the if based on indentation
a = int(input())
b = int(input())
c = int(input())
if a == b :
        if b == c :
                print("nooru")
else:  # paired with the outer if; control reaches here if a not equal to b
                print("innoru")
```

```
if a == b :
        if b == c :
                print("munnuru")
        else: # paired with the inner if; control reaches here if a equals b and b is not equal to c
                print("naanuru")
```

Combination of looping and selection:

example 1:

find all the factors of a given number.

Read the algorithm and then try to understand the program.

```
# file: 7_combined.py
# find all factors of a given number
# factors can vary from 1 to n

# algorithm:
# get n
# factor <- 1
# while factor <= n do
#       if factor is a factor of n
#               display the factor
#       increment factor

n = int(input("enter an integer : "))
f = 1
while f <= n :
        if n % f == 0 :
                print(f, end = " ")
        f = f +  1  # f += 1
print()
```

This is not the best of the programs.

If we observe that if f divides n, so also n / f divides n, we can get two factors at a time. We leave it to you to figure out.

example 2:

find the product of two numbers when the allowed operations are:

check whether a given number is odd

double a number

halve a number

add numbers - not as many as the number of times.

```
# file: 8_combined.py
"""
// Russian Peasant method
//      get two numbers (a, b)
//      initialize the product (say s) to 0
//      while a is not zero
//              if a is odd, then
//                      add b to the product
//              halve a
//              double b
// s + a * b does not change => loop invariant
"""

a = 25
b = 40
s = 0
while a != 0 :
        if a % 2 == 1 :
                s += b
        a = a // 2
        b = b * 2
print(s)
```