range:

- range is a builtin function
- range is lazy.
- range conceptually generates an arithmetic progression.
- range can take one argument, two arguments or three arguments.

- **range is a builtin function**

It is like len rather than list.pop. We do not say variable.function.

- **range is lazy.**

There are functions which generate a lot of output. These functions operate in one of the two ways. Either they produce all the outputs at once or keep on giving an output each time we ask for. In the former case, the function is said to be eager and in the latter case, the function is said to be lazy.

Many functions producing lots of outputs are made lazy in Python 3.x. The lazy functions use less space compared to eager functions. The lazy ones may be more efficient if we do not use all the results generated by the function.

We had this experience in Kandy, Srilanka. We went to a small restaurant. As soon as we sat at a table, the waiter placed a stack of dosas and a plate of vadas in front of us. We could take as many as we wanted. At the end of our eating session, the waiter would bill us based on how many we ate.

This does not happen in Vidyarthi Bhavan. We will have to wait after placing the order.

The restaurant at Kandy was eager whereas Vidyarthi Bhavan is lazy. Which would you prefer?

- range conceptually generates an arithmetic progression.

```
>>> range(1, 10, 2)
range(1, 10, 2)
```

This output tells us that the result to call of range is an arithmetic progression with the initial value 1, the common difference 2 and the final value is less than 10.

Observe that the sequence itself is not displayed. It conceptually holds all the elements.
We can walk through it or we can convert it into a list as follows.

```
>>>list(range(1, 10, 2))
[1, 3, 5, 7, 9]
```

- range can take one argument, two arguments or three arguments.
Let us three examples to illustrate these cases.

Only one argument:
The argument is one past the final value. The default initial value is 0. The step or the common difference is 1.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Two arguments:
The first argument is the initial value. The second argument is one past the final value.
The step or the common difference is 1.

```
>>> list(range(5,10))
[5, 6, 7, 8, 9]
```

Three arguments:
The first argument is the initial value. The second argument is one past the final value.
The third argument is the step or the common difference.

```
>>> list(range(5,15, 2))
[5, 7, 9, 11, 13]
```

Experiment:
- can the step be 0
- what happens?
  list(range(5, 1))
- can the step be negative?
- what happens?
  list(range(1, 5, -1))
  list(range(5, 1, -1))
- can these arguments be float?

---

for statement:

- for statement is a looping structure
- The number of times we execute the body or the suite is normally determinable
- **we use a while loop when the number of iterations is not known in the beginning and we use a for loop when the number of iterations is known**

syntax:

for <variable> in <iterable> :
        <suite>

iterable:

- physically or conceptually has number of elements
- has a builtin mechanism to give an element each time we ask
- has a builtin mechanism to signal when there are no more elements.

semantics of the for statement:

- 1. start the iteration(walking through) of the iterable
- 2. get the element to the variable
- 3. execute the suite or the body
- 4. repeat steps 2 and 3 until the iterable signals that it has no more elements

- 5. gracefully move to the next statement and exit the for loop.

note for C/C++ masters:
In C, we have no such looping structure.
In C/C++, the for loop is same as the while loop. We say that they are isomorphic.
In C++, we have a construct called the **enhanced for** loop which is similar to the for loop of Python.

examples:
a) range object conceptually stands for the arithmetic progression 0 1 2 3 4
i takes the value 1 first time. It executes the suite - where the value of i is displayed. Then i takes the next value from the range object - which is 2. Then it displays 2. This is repeated until range object signals that it has no more elements.

```
>>> for i in range(5):
...     print(i, end = " ")

0 1 2 3 4
```

b) List physically contains 4 elements in this case. The loop variable gets the value 11 first time. Then for statement executes the body. It then gets 22 to the loop variable. Then 22 is displayed. It keeps repeating until all the elements in the list are exhausted. At that point the list signals that it has no more elements.

```
>>> for i in [11, 22, 33, 44]:
...     print(i)
...
11
22
33
44
```

Examples to illustrate when to use the while loop and when to use the for loop:

a) display squares of numbers from 1 to n.

This works. But is not preferred as the number of times the loop has to be executed is known in the beginning itself.

```
# file : 1_squares.py
# display all squares from 1 to n
n = int(input("enter an integer : "))
i = 1
while i <= n :
        print("Square of ", i, " is ", i * i)
        i += 1
```

output:

```
enter an integer : 5
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
```

b)

This is a cleaner program. We do not have to take care of initialization and modification of the loop variable separately.

we should remember to use the right arguments for range - 1, n + 1 as we want
i to vary from 1 to n.

```
# file 2_squares.py
# display all squares from 1 to n
n = int(input("enter an integer : "))
for i in range(1, n + 1):
```

```
        print("Square of ", i, " is ", i * i)
```

output:

```
enter an integer : 5
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
```

Rule: use while when you have to. use for whenever you can.

c) display all squares less than or equal to n

We cannot easily pre-compute the number of iterations. We prefer a while loop here.

```
# file 3_squares.py
# display all squares less than or equal to a given number
n = int(input("enter an integer : "))
i = 1
while i * i <= n :
        print("Square of ", i, " is ", i * i)
        i += 1
```

output:

```
enter an integer : 40
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
Square of  6  is  36
```

This works. But this is a bad program. Observe we are squaring i twice - once to check whether we should stay in the loop and the second time for displaying.

rule: never repeat evaluating the same expression.

Here is the changed program.

```
# file 4_squares.py
# display all squares less than or equal to a given number
n = int(input("enter an integer : "))
i = 1
sq_i = i * i
while sq_i <= n :
        print("Square of ", i, " is ", sq_i)
        i += 1
        sq_i = i * i
```

Observe that square of i is computed before entering the loop and at the end of the body
of the loop. This is a fairly commonly used technique in programming.

d) find all factors of a given number say n.
We can observe that the smallest factor of n is 1 and the largest factor of n is n itself.
All factors have to be between 1 and n.

```
# file 5_all_factors.py
# find all factors of a given number
n = int(input("enter a number : "))
for i in range(1, n + 1):
        if n % i == 0 :
                print(i, end = " ")
print()
```

output:

```
enter a number : 28
1 2 4 7 14 28
```

Even though this works, it is not the best solution.
Observe that when i divides n, n /i also divides n.

for example, 4 divides 28 and quotient is 7. 7 divides 28 and quotient is 4.
Factors occur in pairs. If the number is a perfect square, two factors - the square root - coincide.

We can design our algorithm based on this observation.
If we use this idea, it is difficult to estimate the number of iterations.
We vary i from 1 to square root of the given number,
Each time i divides n, output i and n / i.

There is a catch. We do not want to get into messy floating point operations unless required. So, we will compare i squared with n as the looping condition.

Should we use i * i <= n or i * i < n?
In the former case, we will output the square root of a perfect square twice. In the latter case, we will miss the square root of a perfect square. So, use the latter and take care of the special case(perfect square) outside the loop.

Also observe that if the condition of the loop is i * i <= n and the if condition is pushed in the loop, we can still make the algorithm work. It is a bad idea to have selection in a loop, which is required to test either in the beginning of the loop or at the end.

Rule: Avoid off-by-one errors.
Never compare floating values for equality.

```
# file : 6_all_factors.py
# find all factors of a given number
n = int(input("enter a number : "))
i = 1
while i * i < n :
        if n % i == 0 :
                print(i, n // i, end = " ")
        i += 1
if i * i == n :
        print(i, end = " ")
```

print()

$ python 6_all_factors.py
enter a number : 28
1 28 2 14 4 7
$ python 6_all_factors.py
enter a number : 36
1 36 2 18 3 12 4 9 6

e) summation problems.
cosine(x) = 1 - x**2/2! + x**4/4! ...

In these problems, we have to find some relationship between the terms so that we can
get the next term by using the earlier term. We should not try to recompute each term.

We observe that term(i + 1) = - term(i) * x * x / (2i * (2i + 1)), i starting from 0.

There are two possible types of summation. Sum n terms or sum until the term becomes small
- insignificant.

Let us consider the former case first.
As the number of iterations is known, we should use a for loop.
This shall be the logic. From the domain of mathematics, we do know that the value of x should be radians. We will assume that the input is in radians.

file: 7_summation.py
get x # if necessary, convert to radians
get n
sum <- 0
term <- 1
for i <- 0 to n - 1 do

```
#       sum <- sum + term
#       term <- ____
#       incr i
# print sum

x = float(input("Enter angle in radians : "))
n = int(input("enter the # of terms to add : "))
sum = 0.0
term = 1.0
for i in range(0, n) :
        sum += term
        term = -x * x / ((2 * i + 2) * (2 * i + 1)) * term
        i += 1
print(sum)
```

In the latter case, as we do not know the # of iterations, we should use a while loop.
Observe the use of the function abs - gives the absolute value without consider the sign.

```
# file: 8_summation.py
# get x # if necessary, convert to radians
# sum <- 0
# term <- 1
# i = 0
# while abs(term) >= 0.0001 do
#       sum <- sum + term
#       term <- ____
#       incr i
# print sum
x = float(input("Enter angle in radians : "))
sum = 0.0
term = 1.0
i = 0
while abs(term) >= 0.0001 :
```

```
        sum += term
        term = -x * x / ((2 * i + 2) * (2 * i + 1)) * term
        i += 1
print(sum)
```