Loops

In this lesson, we will learn about for and while loops in Python.

WE'LL COVER THE FOLLOWING

- For loops
  - Looping through a range
  - Looping through lists
  - Nested for loops
- While loops

A **loop** is a control structure that is used to repeatedly execute a set of instructions. They solve the problem of having to write the same set of instructions over and over again.

There are two types of loops that we can use in Python:

- `for` loop
- `while` loop

For loops

In a for loop, we need to define three main things:

1. The name of the iterator
2. The sequence to be traversed
3. The set of operations to be performed

```
for iterator in sequence:
    operations to be performed
```

The `in` keyword specifies that the iterator will go through the values in the sequence or data structure.

sequence or data structure.

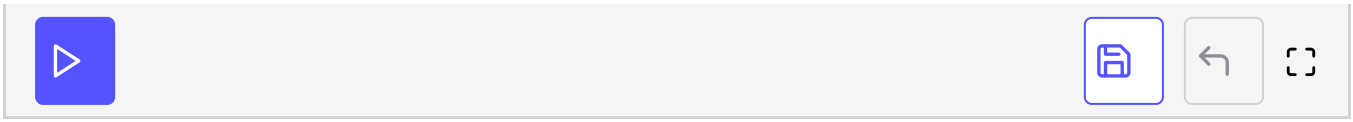
Looping through a range

In Python, the built-in `range()` function can be used to create a sequence of integers to be iterated over in a loop. The syntax is:

```
range(start, end, step)
```

The `end` value is not included in the list. If the `start` index is not specified, its default value is `0`. The `step` decides the number of steps the iterator jumps ahead after each iteration. It is optional and if we don't specify it, the default step is 1.

```
for i in range(1, 11): # A sequence from 1 to 10
  if i % 2 == 0:
    print (i, " is even")
  else:
    print (i, " is odd")
```



The iterator `i` begins at `1` and becomes every succeeding value in the sequence $1, 2, 3, ...10$.

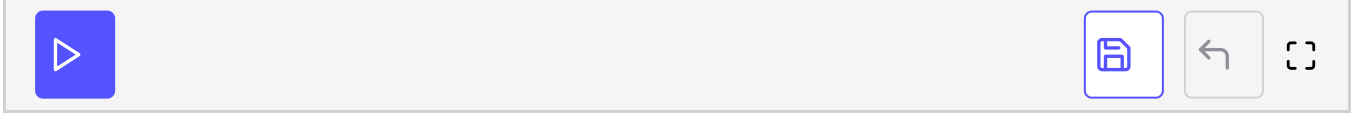
Looping through lists

A list or string can be iterated using its indices.

```
int_list = [2, 4, 6, 8, 10]
print (int_list)

for i in range(0, len(int_list)): # Iterator goes traverses to the last index of the list
  int_list[i] = int_list[i] ** 2

print(int_list)
```

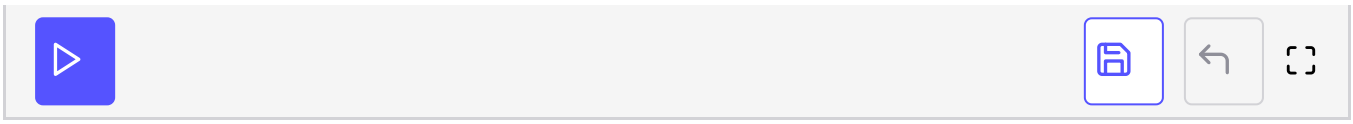


We can also traverse the elements of a list/string directly through the iterator.

```
int_list = [22, 4, 64, 8, 100]
count_greater = 0


for num in int_list: # Iterator goes traverses to the last index of the list
  if num > 10:
    count_greater += 1

print (count_greater)
```



In this example, `num` is the iterator.

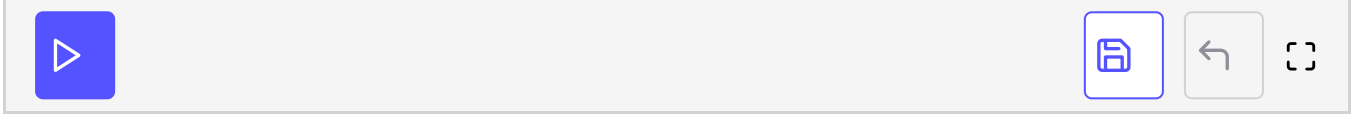
> 💡 An important thing to keep in mind is that in the case above, altering `num` will not alter the actual value in the list. The iterator makes a copy of the list element.

Nested for loops

Python lets us easily create loops within loops. For each iteration of the outer loop, the iterator in the inner loop will complete its iterations for the given range, after which the outer loop can move to the next iteration.

```
num_list = [10, 4, 8, 6, 18, 27, 9]

for n1 in num_list:
  for n2 in num_list: # Now we have two iterators
    if(n1 * 2 == n2):
      print(n1, n2)
```



In the code above, each element is compared with every other element to check if `n2` is equal to `n1 * 2`.

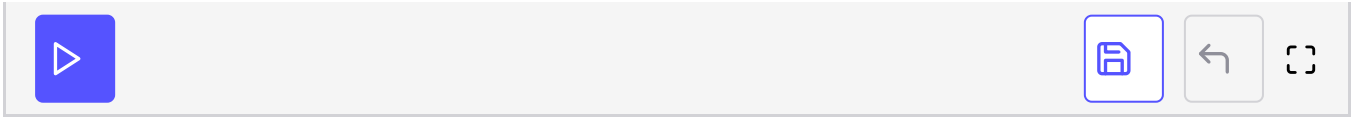
While loops

The `while` loop keeps iterating over a certain set of operations as long as a certain condition holds `True`. Unlike a `for` loop, a while loop is not always restricted to a fixed range. Its execution is based solely on the condition associated with it.

```
while condition is true
    loop over these operations
```

Here's a while loop that finds out the maximum power of `n` before the value exceeds `1000`:

```
n = 3 # Could be any number
power = 0
val = n
while val < 1000:
  power += 1
  val *= n
print (power)
```



In each iteration, we update `val` and check if its value is less than `1000`. The value of power tells us the maximum power `n` can have before it becomes greater than `1000`. Think of it as a counter.

We can also use `while` loops with data structures, especially in cases where the length of the data structure changes as it is iterated.

In the next lesson, we will learn about implementing functions in Python.