Control Flow and Built-in Functions

This lesson covers how the flow of execution transfers between lines of code depending on the scenarios. It also discusses some built-in functions for data structures studied in the previous lesson.

WE'LL COVER THE FOLLOWING

Control structures in Python

The if-else construct

If-then statements are a staple of any programming language. Basically, if you meet a certain condition, then something happens. In Python, `elif` stands for `else if`, meaning that if the previous conditions were not met, check that condition. `Else` is a catch-all condition for any remaining flows. Python follows the following syntax:

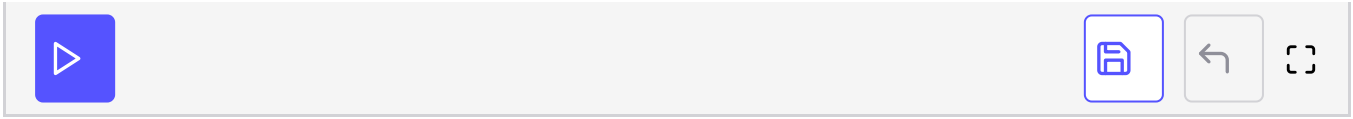
```
if condition:
  statements
elif condition:
  statements
else:
  statements
```

Following is a code example:

```
def age_check(age):

    if age > 40:  # if age greater than 40, print "Older than 40"
        print("Older than 40")
    elif age > 30 and age <= 40: # if age greater than 30 and less than or equal to 40, print
        print("Between 30 and 40")
    else:  # if neither of the previous conditions are met, print "Other"
        print("Other")

print(age_check(41))
```



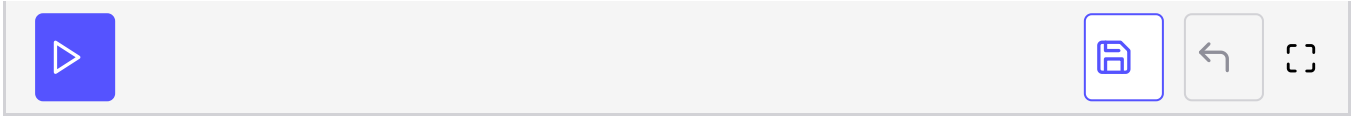
The for construct

Loops allow you to iterate over an iterable. That's not a very helpful definition, so let's consider the most common use case, lists. A loop allows you to iterate over a list or other data types that also allow iteration.

You can contain your iterable in the `enumerate()` command to add a counter to your loop. This is useful if you want to loop over a list of values while still having access to the iterable index.

```
names = ['tyler', 'karen', 'jill']   # list containing names

for i, name in enumerate(names):     # iterating over names
    print("Index: {0}".format(i))    # printing index number
    print("Value: {0}".format(name)) # print the value at the index
```



Speaking of lists and loops, there is an easy way to combine the two in Python called **list comprehensions**. Basically, list comprehensions allow you to create a new list based on an existing list in a condensed way.

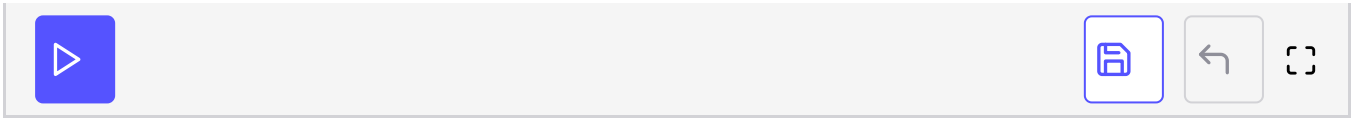
Here is a way to create a list of the numbers greater than 5 (up up to 14) with a list comprehension.

> **Note:** The range() creates a list of values starting from the first number and ending at the number right before the last number.

So `range(1,6)` creates [1, 2, 3, 4, 5].

We can add an if statement to our list comprehension by adding the condition at the end. Look at the following implementation:

```
numbers_gt_5 = [x for x in range(1,15) if x > 5]  # loop over the range and only keep the val
print(numbers_gt_5)
```



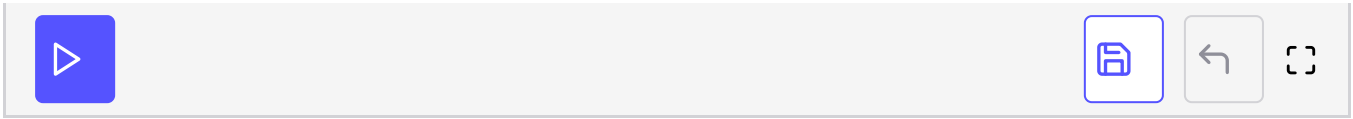
The above program is looping over a range:

```
x for x in range(1,15)
```

It means the loop will iterate 14 times. And in each iteration, an if statement executes. The if statement will be true when `x` is greater than **5**. So the list `numbers_gt_5` will start getting populated once the iterator `x` exceed value of **5**.

We could also increment every value in a list by 1 as below:

```
nums_plus_one = [x + 1 for x in range(5)]
print(nums_plus_one)
```



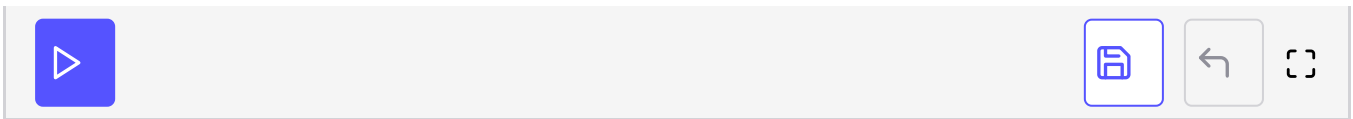
Built-in functions in Python

The `sort` function

Sorting comes up a lot when talking about lists and is, fortunately, very easy:

```
my_list = [2, 10, 1, -5, 22]
my_list.sort()  # sorting the list

print(my_list)
```



You'll notice it sorts from smallest to largest. If you want more functionality to

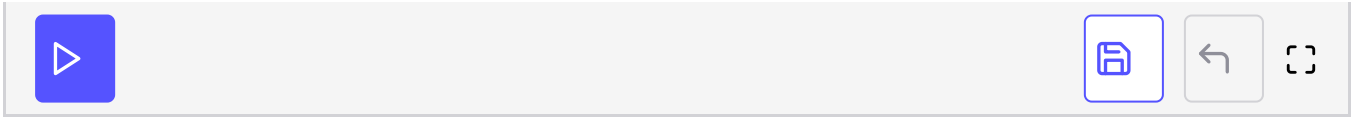
You'll notice it sorts from smallest to largest. If you want more functionality to your sorting, I would suggest using the `sorted()` function.

This allows you to reverse the order with the `reverse` parameter, and use a `key` parameter to specify the basis of your sort. For example, instead of a normal sort, we can reverse sort by the absolute values as given below:

```
my_list = [2, 10, 1, -5, 22]

# Sorted reversely on basis of absolute value
my_list_sorted_abs = sorted(my_list, key=abs, reverse=True)

print(my_list_sorted_abs)
```



The `zip` function

Lastly, a slightly more complicated function: `zip`. You can do a lot of useful things with `zip`, but here are 2 common use cases:

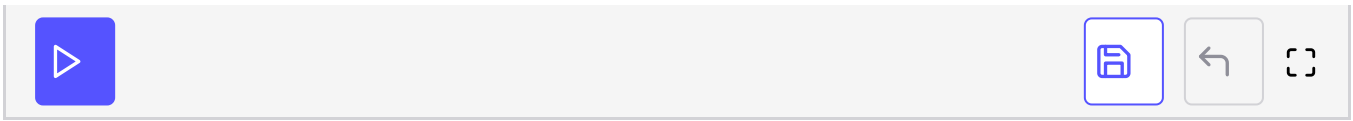
- Combining two lists into a list of tuples
- Breaking a tuple into two lists

Combining two lists into a tuple

The `zip` actually returns a *generator*, so we have to wrap it in `list()` to print it. This would not be necessary if you wanted to loop over it though, because generators are iterable:

```
list_1 = [1, 2, 3]  # create your first list
list_2 = ['x', 'y', 'z']  # create your second list

print(list(zip(list_1, list_2)))  #combine and print
```

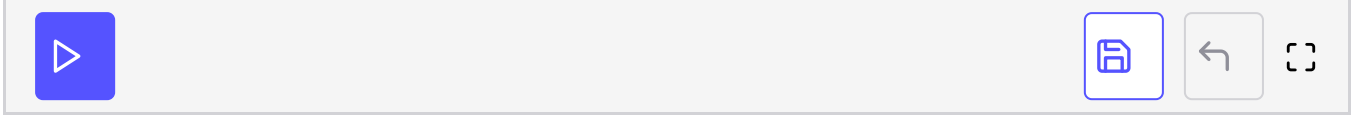


Breaking a tuple into two lists

This function can break a tuple into two lists, exactly the reverse of the above use case.

```
pairs = [('x', 1), ('y', 2), ('z', 3)]  # a list of tuples
letters, numbers = zip(*pairs)  # break into two lists
```

```
print(letters)  # print the first values of the tuples
print(numbers)  # print the second values of the tuples
```



Now that we have discussed the fundamentals of the standard Python library, we will explore our first external Python library.