Data Structures

This lesson discusses the common built-in data structures of Python used for data processing.

WE'LL COVER THE FOLLOWING

Lists in Python

As mentioned in the previous section, **lists** are a useful data structure in Python. Lists are *ordered*, *changeable*, and allow for *duplicates*. You create a list with the [].

We can slice a list. *Slicing* means to extract a part of the list. When slicing, the first number is included in the return set while the last number is not. For example:

```
p1 = [1,2,3,4,5,6,7,8,9,10]
p2 = p1[0:5]
```

The list `p2` will be `[1,2,3,4,5]`. Because we'll start from $\mathbf{0^{th}}$ index of `p1` and will stop at $\mathbf{5^{th}}$ index slicing all the elements till $\mathbf{4^{th}}$ index.

> **Note:** You can also write `p2 = p1[0:5]` as `p2 = p1[:5]`.

Here are many of the ways you can use lists.

Please take the time to review the comments and the `print` statements to understand what each line is doing. You can tie the print statements to the code by the lines printed above the results in the output.

```
# This creates the list
depths = [1, 5, 3, 6, 4, 7, 10, 12]

# This outputs the first 5 elements. No number before the : implies 0
first_5_depths = depths[:5]

print("---0---")
print(first_5_depths)

# You can easily sum
print("---1---")
print(sum(depths))

# And take the max
print("---2---")
print(max(depths))

# Slicing with a negative starts from the end, so this returns the last element
print("---3---")
print(depths[-1])

# This returns the end of the list starting from the second to the end
# Nothing after the : implies the end of the list
print("---4---")
print(depths[-2:])

# This returns the second, third, and forth elements
# Remember counting starts at zero!
print("---5---")
print(depths[2:5])

# These commands check if a value is contained in the list
print("---6---")
print(22 in depths)
print(1 in depths)

# This is how you add another value to the end of your list
depths.append(44)
print("---7---")
print(depths)

# You can extend a list with another list
depths.extend([100, 200])
print("---8---")
print(depths)

# You can also modify a value
# This replaces the 4th value with 100
depths[4] = 100
print("---9---")
print(depths)

# Or you can do insert to accomplish the same thing
depths.insert(5, 1000)
print("---10---")
print(depths)
```

When talking about lists, it is useful to discuss **generators**. Generators are objects you can loop over like a list, but they are lazy. It means they don't have to store the entire list in memory. Instead they return the next value in the list only when asked, making them very memory efficient. Because of this property, you can have a function that generates an infinite list; it never stores the value in memory but just keeps track of the last value returned and returns that value + 1 when asked for another value.

Dictionaries in Python

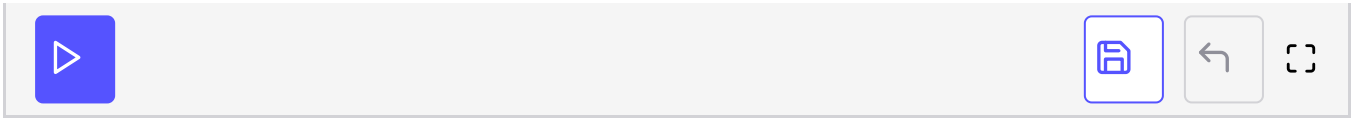
Another useful data type is **dictionary**. Dictionaries hold *key-value* pairs. In other words, there are *unique* **keys** in a dictionary, and each key is associated with a **value**.

The key-value pairs are *unordered* and *changeable*. It means that key-value pairs are stored in an unordered manner because data is accessed using a key rather than index. And we can always change the value of a key when needed.

They are initialized with {}. Let's take a closer look at some functionalities in code!

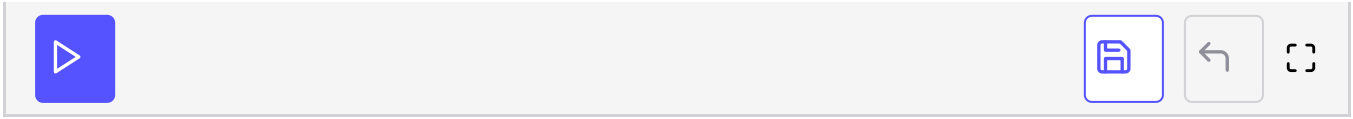
```
# Initialize the dictionary.
# Keys are first then a : then the value
my_dict = {"age": 22, "birth_year": 1999, "name": "jack", "siblings": ["jill", "jen"]}

# Get the value for the key age
print("---0---")
print(my_dict['age'])

# Check is age is a key
print("---1---")
print('age' in my_dict)

# Check is company is a key
print("---2---")
print('company' in my_dict)

# Get the value for they key age
print("---3---")
print(my_dict.get('age'))

# Get the value for they key company
# If it doesn't exsist, return 1
print("---4---")
print(my_dict.get('company', 1))
```

```
# Return all the keys
print("---5---")
print(my_dict.keys())


# Return all the values
print("---6---")
print(my_dict.values())

# Return all the key, value pairs
print("---7---")
print(my_dict.items())
```



You can also use `defaultdict`, which is a more advanced dictionary data type that allows you to specify a default data type. Let's look at an example:

```
from collections import defaultdict  # import defaultdict class
my_default_dict = defaultdict(int)   # make a default dictionary
my_default_dict['age'] = 22          # adding a key-value pair
print(my_default_dict['company'])    # printing the value of the key "company"
```



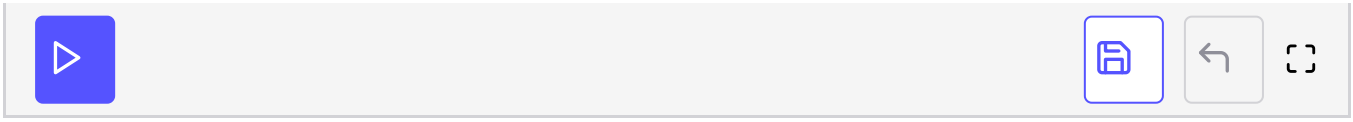
In this example, we initialize `defaultdict` with *int*, at **line 1**. That means that it assumes all the values will be integers and that 0 is the default value for any key not assigned. As seen above, when we try and get the value for the `company` at **line 6**, which has yet to be assigned any value in our dictionary, it returns 0. This can be very useful when you want to assume a default value and don't want to use the `get()` functionality as shown previously.

Sets in Python

Sets are another useful data type. Sets are an *unordered* collection of *unique* elements, which means any duplicates are automatically removed. Sets allow you to do operations like *union*, *intersection*, and *difference*. Here's an example:

```
my_set = set()
my_set.add(1)
my_set.add(2)
my_set.add(1)
# Note that the set only contains a single 1 value
print("---0---")
print(my_set)
```

```
my_set2 = set()
my_set2.add(1)
my_set2.add(2)
my_set2.add(3)
my_set2.add(4)
print("---1---")
print(my_set2)

# Prints the overlap
print("---1---")
print(my_set.intersection(my_set2))
print("---2---")

# Prints the combination
print(my_set.union(my_set2))

# Prints the difference (those in my_set but not my_set2)
print("---3---")
print(my_set.difference(my_set2))
```



In the next section, we will look at controlling the flow of your code and learn some helpful functions.