Your First Look at Data Structures and the Collections Module

Daniel Chakraborty

International University

DLMDSPWP01: Programming with Python

Dr. Cosmina Croitoru

Your First Look at Data Structures and the Collections Module

Python is a fascinating general-purpose programming language. With its simple-to-learn syntax and a rich ecosystem of libraries, there seems to be no limit as to what you can do with this scripting language (Romano, 2018). Apart from all the fancy applications that Python boasts of, doing some serious number crunching is very possible, and which is probably why it is the language of choice for Data Science (Lutz, 2013).

However, working with numbers daily without thinking about whether your code has been written efficiently, is extensible or easy to use is not wise. Sooner or later, all of us will have to analyse our algorithms when working with large datasets and decide whether the data structures chosen offer said benefits. While this holds true for code written in application development, this feels twice as important in Data Science, given how much data our code must sift through (Gautam, 2023). As a serious student of Python, structured programming and computer science, a much-cherished objective has always been to work with numbers using popular statistical methods to come up with data-driven insights. Not only does this involve learning the programming fundamentals, the popular Python libraries available but also exploring the various data structures available (Geeks for Geeks, 2023).

Speaking of data structures, my curiosity was piqued ever since the Collections module was part of our selected reading during this course. As we all know, the built-in data structures that Python uses are the list, tuple, dictionary and set. So, coming across this module in the standard library has led me to question what benefits are offered by containers in the Collections module and whether these benefits are substantial enough to be taken seriously (Romano, 2018).

To achieve this, it is first necessary to gain a fair understanding of the native data structures that Python offers. So, we are going to cover the list, tuple, and dictionary apart from how they are used (Lutz, 2013). Also, we will look at the string data type as it contains a collection of characters too. Once we have established such a baseline, we will then compare what the containers in the Collections module offer (Romano, 2018).

Now, before we move on, it is important to note that there are distinct differences between data structures and databases, if it is not obvious enough. Both concepts are not the same as data structures are stored in main memory while databases represent permanent or persistent memory. In most cases, whatever data you end up computing as part of a solution gets moved to a database from a data structure, whether it is relational or NoSQL in nature (Singh, 2021). This assignment will not attempt to broach such a topic. Its focus will remain on data structures alone.

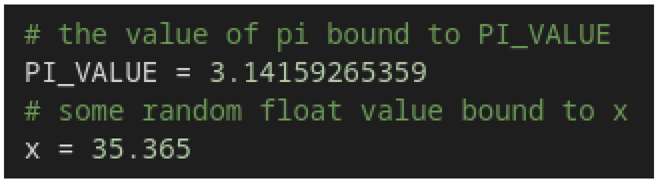
That said, it is important to address why data structures are integral to programming in general before we analyze the built-in types or the Collections module containers. Or even set out to discover how these containers improve on the built-in Python data structures, in terms of providing additional features or by going beyond certain limitations (Ramos, 2023).

The Importance of Data Structures in Programming

The first thing that every programmer learns is assigning values to variables or constants, regardless of language syntax (Busbee & Braunschweig, 2018). Yes, there are nuances that differentiate one programming language from another but values are generally bound to variables and constants, as shown in Figure 1:

Figure 1

Variables and Constants bound to Values

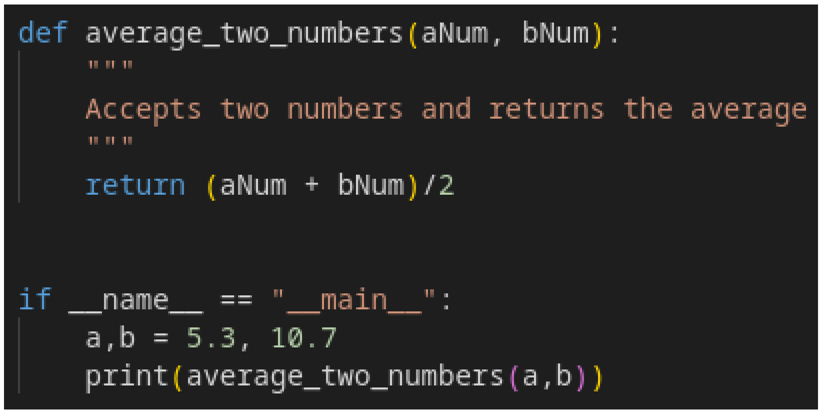


```
# the value of pi bound to PI_VALUE
PI_VALUE = 3.14159265359
# some random float value bound to x
x = 35.365
```

Now, while it is just as important for Computer Science students to understand how values are bound to variables, knowing where this 'state' is saved matters too. To memory addresses, if you will (Rouse, 2017). For example, if you wanted to find the average of two numbers, the Python code in Figure 2 should help you do so:

Figure 2

Average of Two Numbers



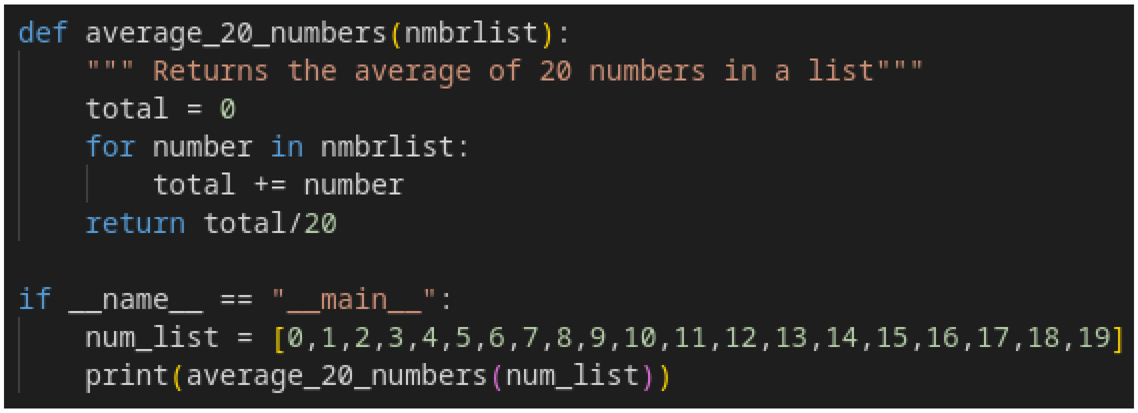
```python
def average_two_numbers(aNum, bNum):
    """
    Accepts two numbers and returns the average
    """
    return (aNum + bNum)/2


if __name__ == "__main__":
    a,b = 5.3, 10.7
    print(average_two_numbers(a,b))
```

When you run this code, you will know that the average of the two numbers will be eight and will be stored at said memory address, if a variable has been assigned to store the result. Now, what if you wanted to find the average of 20 numbers at one shot? Does it make sense to add another 18 variables to the above program? Clearly not. That is just too much work when it comes to assigning memory manually.

So, there must be a simpler way to hold several numbers and access each of them in your program, right? This is a simple scenario in which data structures come to the fore, with the Python list being the simplest one of all. From the code in Figure 3, you can tell that the list num_list is nothing but a sequence of items, which, in this case, are integers (Romano, 2018).

Figure 3

Average of Twenty Numbers



```python
def average_20_numbers(nmbrlist):
    """ Returns the average of 20 numbers in a list"""
    total = 0
    for number in nmbrlist:
        total += number
    return total/20

if __name__ == "__main__":
    num_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
    print(average_20_numbers(num_list))
```

Let us look at how such a data structure solves this problem. You can access each of these 20 values assigned to a variable name num_list by using a ‘for’ loop to compute the total and average of all 20 numbers (Lutz, 2013). In fact, you can add more numbers to this list and perform both computations. What one should conclude because of this simple example is that if you work with a lot of data, then learning all that you can about data structures, apart from the list, is your best bet to come up with efficient solutions (Gautam, 2023).

Of course, not all problems in Computer Science are this simple and which is why the ubiquitous list is not the only data structure used by programmers. Python offers several linear data structures such as tuples, dictionaries, and sets. These types of data structures come with specific properties that offer certain benefits to those who would like to work with data in an effective manner. So, it should not be surprising to note that a number of these data structures are used in the most popular applications that we use each day (Romano, 2018).

What is also important to note is that each of these data structures will run optimally by using a suitable algorithm designed specifically for said data structure. For example, if you take the singly linked list data structure, there is a specific algorithm that has been created just to help you with all the operations associated with such a data structure. We are talking about adding a new data node as well as deleting, updating, and reading old ones. While going through a list of items might seem simple enough, things get complex with linked lists because of the way its items are stored in memory. No matter what, if you are studying data structures, then you will have to understand these algorithms as well (Biswas, 2022).

Speaking of which, it is time to explore the Python built-in data structures and the string data type (Lutz, 2013). As we examine each of these data structures, we will obtain a clearer picture of what they do as well as their limitations. In doing so, we should be able to see how the containers in the Collections module fills in the gaps (Aggarwal, 2023).

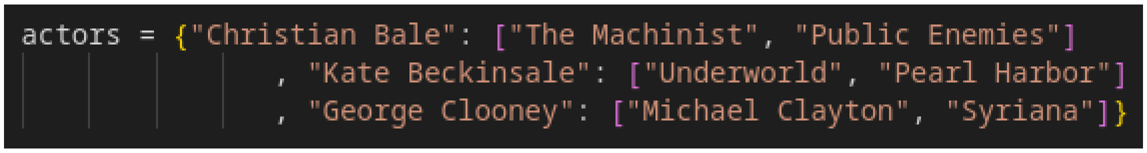
General Purpose Built-In Data Structures in Python: An Overview

Consisting of lists, tuples, and dictionaries, these three data structures are used extensively in Python scripts. This is no different with Python strings as well (Lutz, 2013). Implementations of both arrays and dataframes are also available in Python but they are not native to the programming language. For those who do not understand the difference, here is how it works: you can declare a list, tuple, dictionary, and string without having to install or import any libraries for the same (Romano, 2018). The same cannot be said for Numpy arrays, Pandas dataframes, and some of the Collections containers we will examine later.

Now, for most people who have not used these Python data structures, you can use the index position of an element or character to access it in a list, tuple, and string, while knowing the key associated with a value in a dictionary will help you do the same. You can also use each of these structures in your script and combine them to come up with a solution (Lutz, 2013). For example, you can use a Python list and a dictionary to associate several movies to a popular Hollywood actor, as shown in Figure 4:

Figure 4

Combining a dictionary and a list



```python
actors = {"Christian Bale": ["The Machinist", "Public Enemies"]
          , "Kate Beckinsale": ["Underworld", "Pearl Harbor"]
          , "George Clooney": ["Michael Clayton", "Syriana"]}
```

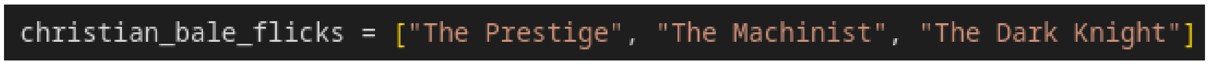
In other words, you can use a dictionary that consists of the key as actor name and a list that is a sequence of that actor's movies. You can even convert data structures from one type to the other using simple methods such as list, dict and tuple (Romano, 2018). Speaking of which, let us look at each of these data structures, their purpose, methods and how they function on their own or in conjunction with other data types.

#1: List

Even if the list seems like the simplest of the lot, it is often thought of as an array. They are not the same, though. While lists can contain heterogenous items, arrays only permit items of one type (Romano, 2018). In other words, with lists, you can have strings, integers, floats, and Boolean values (Lutz, 2013). In stark contrast, with arrays, you are permitted to only declare one of these data types. Any student who has taken a Discrete Mathematics class will simply refer to the Python list as a sequence of values (Romano, 2018). In real life, this could resemble a list of movies in which an actor has played a prominent role in, as shown in Figure 5:

Figure 5

Christian Bale movies in a Python list

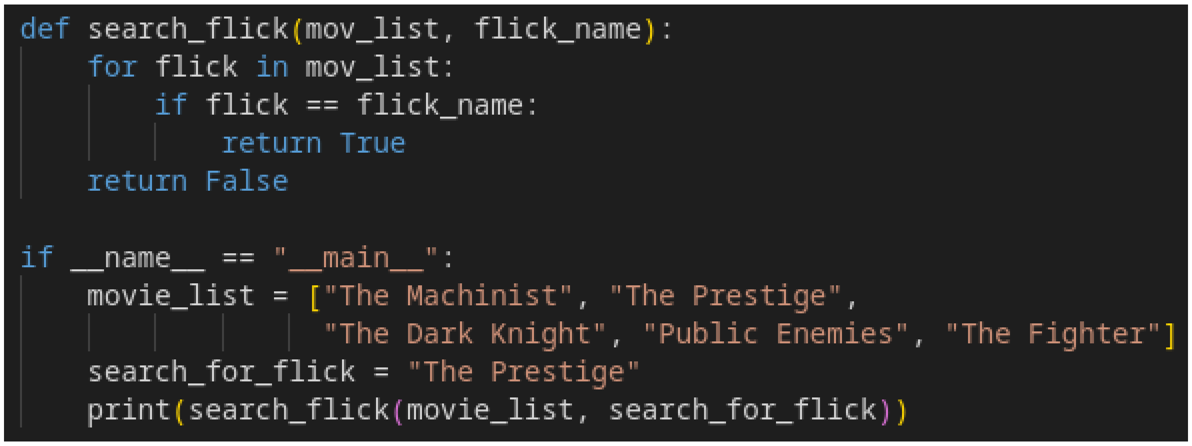


```python
christian_bale_flicks = ["The Prestige", "The Machinist", "The Dark Knight"]
```

As you can see, a list maintains the order in which it was declared while additional items can be added to the end of the list. Apart from unique items that can be added and deleted, duplicates are permitted too ("Python Lists", 2023). Now, to access all the elements in a list for said computation, you must use a loop to iterate throughout the list. For loops are usually the most common type used to go through each of these lists where the index position of each item is how said item is accessed (Lutz, 2013). As shown in Figure 6, let us perform linear search for a favorite Hollywood movie using such a loop:

Figure 6

Performing Linear Search using a ‘for’ loop

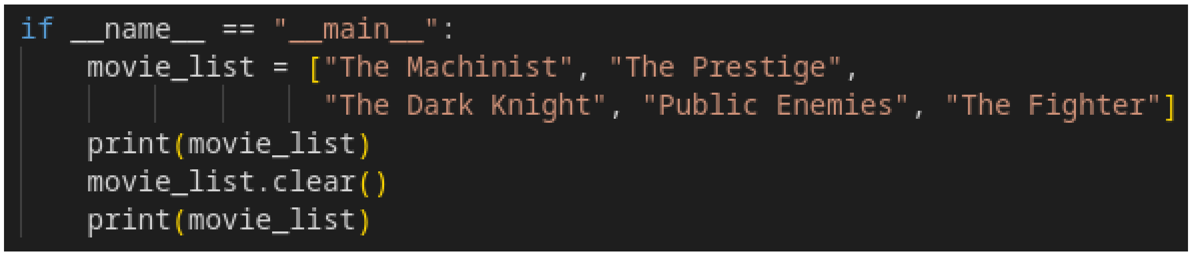


```python
def search_flick(mov_list, flick_name):
    for flick in mov_list:
        if flick == flick_name:
            return True
    return False

if __name__ == "__main__":
    movie_list = ["The Machinist", "The Prestige",
                  "The Dark Knight", "Public Enemies", "The Fighter"]
    search_for_flick = "The Prestige"
    print(search_flick(movie_list, search_for_flick))
```

What makes the use of lists so fascinating is the ability to perform certain operations using list methods that treat said list as an object (“Python - List Methods”, 2023). For example, you can use the clear() method to empty the list completely, as shown in Figure 7:

Figure 7

Using the clear list method

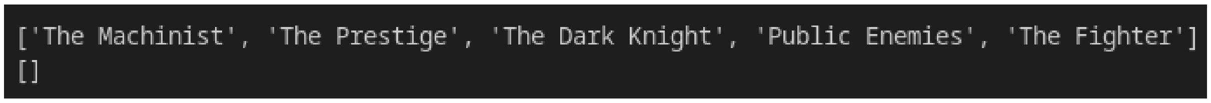


```python
if __name__ == "__main__":
    movie_list = ["The Machinist", "The Prestige",
                  "The Dark Knight", "Public Enemies", "The Fighter"]
    print(movie_list)
    movie_list.clear()
    print(movie_list)
```

As expected, this list method empties movie_list to return an item less list in the second row, as shown in Figure 8:

Figure 8

Output before/after using the clear list method



```
['The Machinist', 'The Prestige', 'The Dark Knight', 'Public Enemies', 'The Fighter']
[]
```

Not very differently, here are the most common list methods as shown in Table 1 that you can use in your Python scripts ("Python - List Methods", 2023):

Table 1

Python List Methods

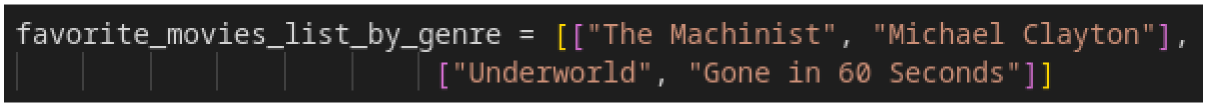
append()	copy()	clear()
count()	extend()	index()
insert()	pop()	remove()
reverse()	sort()	min()
	max()	

Note. Taken From Python - List Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_lists_methods.asp

As mentioned earlier, you can not only include items that are of primitive data types but other data structures and user-defined data types too. For example, you can have a list of lists. Or a matrix, if you are using integers. As shown in Figure 9, this type of nesting helps you do so much more with your data although the algorithm that you write gets complex too (Lutz, 2013).

Figure 9

Nested lists



```
favorite_movies_list_by_genre = [["The Machinist", "Michael Clayton"],
                                 ["Underworld", "Gone in 60 Seconds"]]
```

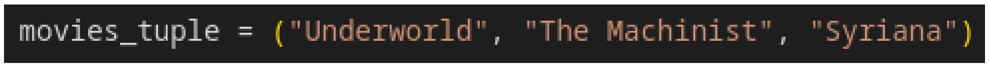
Now that we have introduced the lowly Python list, let us move on to getting to know the next built-in data structure: the tuple ("Python Tuple", 2023).

#2: Tuple

Much like the list, the Python tuple, as shown in Figure 10, is also an iterable, so that means you can assign several items to this data structure. However, once you declare these items, the order will remain the same with one important difference: items cannot be added, updated, or deleted (“Python Tuple”, 2023).

Figure 10

Python tuple

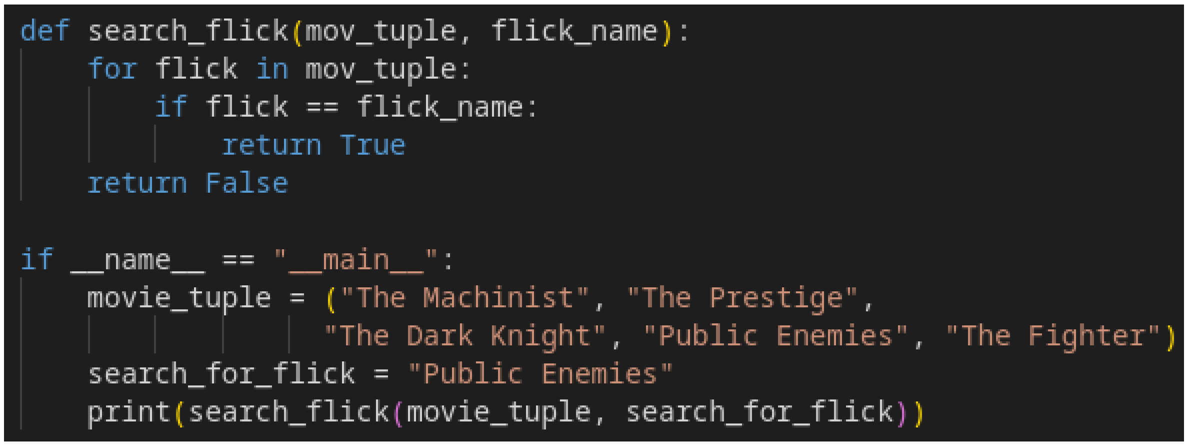


```python
movies_tuple = ("Underworld", "The Machinist", "Syriana")
```

Now, just as lists use square brackets, items in a tuple are enclosed by parentheses as shown above (Lutz, 2013). Also, items in a tuple can be of varying data type just as it is with lists (“Python Tuple”, 2023). Finally, you can use a for loop to iterate through a tuple as well, as shown in Figure 11:

Figure 11

Performing Linear search using a ‘for’ loop



```python
def search_flick(mov_tuple, flick_name):
    for flick in mov_tuple:
        if flick == flick_name:
            return True
    return False

if __name__ == "__main__":
    movie_tuple = ("The Machinist", "The Prestige",
                   "The Dark Knight", "Public Enemies", "The Fighter")
    search_for_flick = "Public Enemies"
    print(search_flick(movie_tuple, search_for_flick))
```

As we continue our comparison with the list data structure, there are just two methods used to work with tuples, as shown in Table 2:

Table 2

Python Tuple Methods

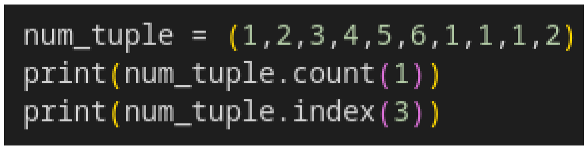
index() count()

Note. Taken from Python - Tuple Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_tuples_methods.asp

As you can tell from Figure 12, the first method locates the index position of the item in the tuple while the second counts how many times said item occurs in this data structure ("Python – Tuple Methods", 2023).

Figure 12

Python Tuple Methods

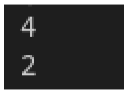


```
num_tuple = (1,2,3,4,5,6,1,1,1,2)
print(num_tuple.count(1))
print(num_tuple.index(3))
```

From the tuple methods used, we want to count how many times the integer 1 occurs in num_tuple. Apart from this, we want to find the index position of the integer 3 in this tuple too. Now, from the output in Figure 13, the integer 1 occurs four times in this tuple while the index position of the integer 3 is 2.

Figure 13

Python Tuple Methods Output

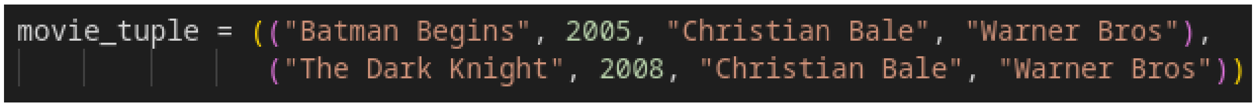


```
4
2
```

Finally, you can add data structures as items in a tuple much like integers, strings, Booleans, and floats. For example, you can have nested tuples where each tuple represents an item, as shown in Figure 14:

Figure 14

Nested tuples



```
movie_tuple = (("Batman Begins", 2005, "Christian Bale", "Warner Bros"),
               ("The Dark Knight", 2008, "Christian Bale", "Warner Bros"))
```

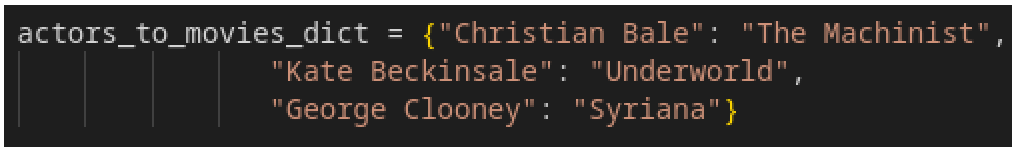
With that introduction of the tuple, let us look at a data structure that is different from the list and tuple: the dictionary (Lutz, 2013).

#3: Dictionary

Now, while both the list and the tuple use values and indices, the dictionary is structured with a key and a value (Romano, 2018). As shown in Figure 15, “Christian Bale” is a key while its value is “The Machinist”:

Figure 15

Python dictionary

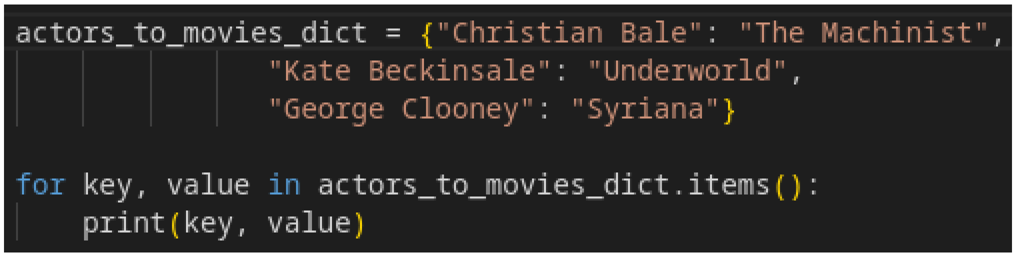


```
actors_to_movies_dict = {"Christian Bale": "The Machinist",
                         "Kate Beckinsale": "Underworld",
                         "George Clooney": "Syriana"}
```

Represented by curly braces, you can add items to said dictionary after it has been declared (Lutz, 2013). So, you can append an item “Anne Hathaway”: “The Dark Knight Rises” and several more to the dictionary shown above. Much like the list and the tuple, the order in which the items were added is retained. Also, the values associated with each of these keys can be duplicates as well as of varying data type but every key must always be unique. Dictionaries do not have indices like the other two data structures but use the ‘key’ to access the associated value (“Python Dictionaries”, 2023). Also, you can use a modified for loop to access each of the items in the above dictionary, as shown in Figure 16:

Figure 16

Using a modified ‘for’ loop to access each key-value pair in a dictionary

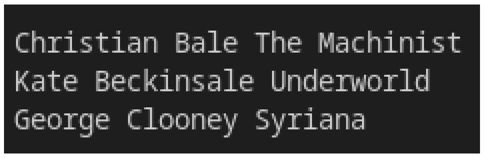


```
actors_to_movies_dict = {"Christian Bale": "The Machinist",
                         "Kate Beckinsale": "Underworld",
                         "George Clooney": "Syriana"}

for key, value in actors_to_movies_dict.items():
    print(key, value)
```

As shown in Figure 17, this is how the actor-to-movie dictionary with its key-value pair output looks:

Figure 17

Dictionary key-value pair output



```
Christian Bale The Machinist
Kate Beckinsale Underworld
George Clooney Syriana
```

Now, just like the other two data structures discussed, Table 3 shows us 11 dictionary methods that one can use in their Python scripts ("Python Dictionary Methods", 2023).

Table 3

Python Dictionary Methods

clear()	copy()
fromkeys()	get()
items()	keys()
pop()	popitem()
setdefault()	update()
values()	

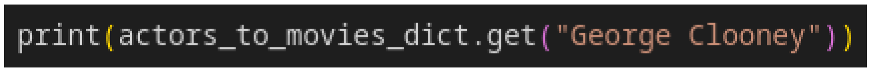
Note. Taken from Python Dictionary Methods. (2023). W3Schools.

https://www.w3schools.com/python/python_dictionaries_methods.asp

For example, if we want to access the value of a particular key, then we can use the get() method as shown in Figure 18:

Figure 18

Access the value of dictionary key using the get method

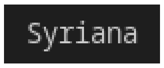


```
print(actors_to_movies_dict.get("George Clooney"))
```

When we run this method on the actors_to_movies_dict dictionary, we obtain this output, as shown in Figure 19:

Figure 19

Value of Dictionary Key Output

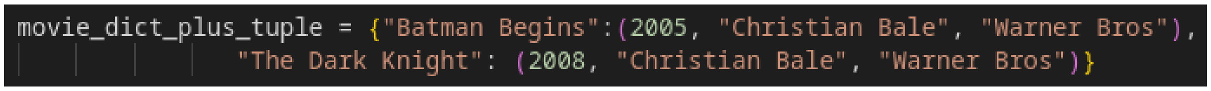


```
Syriana
```

Now, much like the other two data structures, you can add additional data structures instead of primitive values as dictionary items. For example, let us rework the movie_tuple to a dictionary named movie_dict_plus_tuple, as shown in Figure 20:

Figure 20

A Dictionary of Tuples



```
movie_dict_plus_tuple = {"Batman Begins":(2005, "Christian Bale", "Warner Bros"),
                        "The Dark Knight": (2008, "Christian Bale", "Warner Bros")}
```

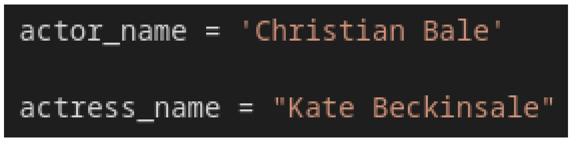
As you can examine this new dictionary, you can easily tell that the key is the name of a movie while the value is a tuple that contains details about the movie itself such as year, actor name and movie studio. Finally, now that we have looked at how a Python dictionary functions, let us look at a data type that is considered primitive in this programming language but is non-primitive in others (Lutz, 2013).

#4: String

Even if the Python string is not regarded as a data structure, we can treat it so as it is a sequence of characters that can be either letters, digits, or special characters ("Python String", 2023; Lutz, 2013). With that out of the way, a string is enclosed by single, double, and triple quotes, as shown in Figure 21. The ones in triple quotes are called docstrings and can span multiple lines (Romano, 2018).

Figure 21

Python Strings

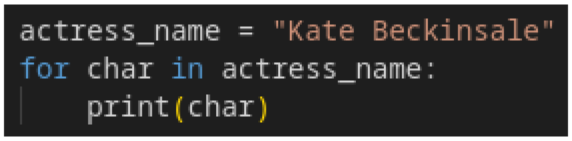


```python
actor_name = 'Christian Bale'

actress_name = "Kate Beckinsale"
```

Much like tuples, strings are immutable, which means that once you declare them, you cannot change or delete a character. Meaning, you cannot access a 'slice' of a declared string and change it with another value in the same variable (Lutz, 2013). Still, you can access each character in a string using indices and a for loop much like the list and tuple data structures, as shown in Figure 22:

Figure 22

Access characters in strings using a 'for' loop



```python
actress_name = "Kate Beckinsale"
for char in actress_name:
    print(char)
```

Not surprisingly, as shown in Table 4, Python strings have several methods that you can use ("Python String Methods", 2023):

Table 4

Python String Methods

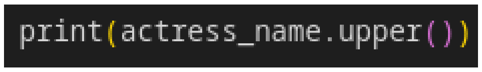
capitalize()	count()	endswith()
index()	isalnum()	isalpha()
isascii()	isdecimal()	isdigit()
isidentifier()	islower()	isnumeric()
isprintable()	isspace()	istitle()
isupper()	join()	lower()
lstrip()	replace()	rfind()
rstrip()	split()	startswith()
strip()	title()	upper()

Note. Taken from Python String Methods. (2023). W3Schools. https://www.w3schools.com/python/python_ref_string.asp

As shown in Figure 23, if you want to capitalize the entire word, we can use the upper() method to do this ("Python String Methods", 2023):

Figure 23

The upper string method

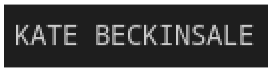


```
print(actress_name.upper())
```

When we run this method on actress_name, we get the following output, as shown in Figure 24 ("Python String Methods", 2023):

Figure 24

The string upper method output



```
KATE BECKINSALE
```

With this introduction of the list, tuple, dictionary and the string and their functionality, let us also look at some of the common limitations that each of these data structures grapple with before we look at some of the alternatives that Python developers can use.

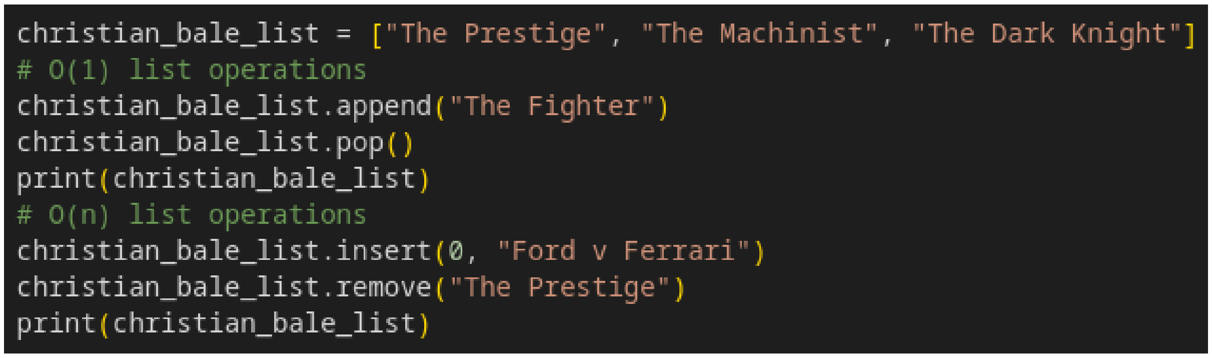
Python Built-In Data Structures – Common Limitations

Now, as cool as these data structures are, not everything is as perfect as developers want it to be. There are required improvements that will make the developer's life easier when using these built-in data structures and which has been the reason why alternative solutions exist (Shahid, 2020). But before we dive into what these solutions are, let us explicitly describe certain areas of improvement that have been an annoyance to developers who use these data structures in their code.

For starters, we had discussed how important it is to use the least resources in programming. Otherwise known as time and space complexity, the efficiency of a data structure used in an algorithm is determined by these two measures, with O(1) being the best and O(n!) being the worst (Kumari, 2023). Speaking of which, if you want to access an item in a Python list, you can do so using the index number of that item. Here, the time complexity is O(1), since you will directly access that item based on the index number. Now, if you want to pop or append an item at the end of the list, this takes O(1) too ("collections", 2023). Now, there are times when a programmer might want to pop or append items at the beginning of the list instead of the end. The 'insert' and 'remove' list methods can be used. Unfortunately, the time complexity will be O(n) as the other elements must be moved accordingly upon the removal or addition of the item ("collections", 2023). As demonstrated in Figure 25, when adding 'Ford v Ferrari' in the list, the other movies must be moved one place forward. In the case of 'remove,' all items must be moved to one index position less than their prior index position.

Figure 25

The string upper method output



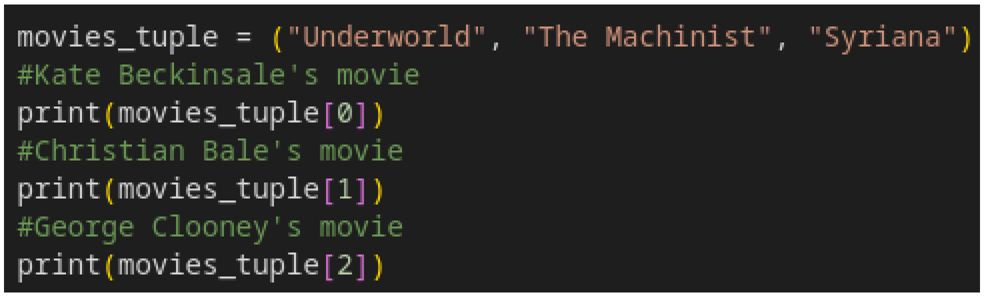
```
christian_bale_list = ["The Prestige", "The Machinist", "The Dark Knight"]
# O(1) list operations
christian_bale_list.append("The Fighter")
christian_bale_list.pop()
print(christian_bale_list)
# O(n) list operations
christian_bale_list.insert(0, "Ford v Ferrari")
christian_bale_list.remove("The Prestige")
print(christian_bale_list)
```

With the second structure known as the tuple, the challenge does not necessarily lie with performing such operations because they are much faster but with accessing elements using the right index position. As we have learned earlier, once you declare a sequence of elements in a tuple, only accessing these items are possible while all the other operations that you find associated with the list data structure are not (Elemati, 2023).

Much like accessing an item in a list using the index position, the time complexity for this operation on the tuple data structure is also O(1). However, not being able to remember which index position represents which field can lead to errors, so being able to name these index positions based on the common field that they represent makes sense (Shrivarsheni, 2023). As shown in Figure 26, it is better to give field names to the index positions in the movies_tuple below:

Figure 26

Giving field names to index position in namedtuple

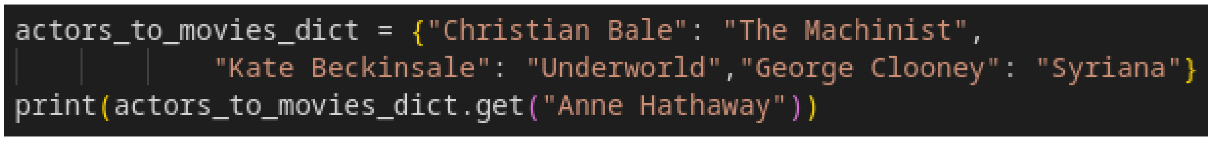


```
movies_tuple = ("Underworld", "The Machinist", "Syriana")
#Kate Beckinsale's movie
print(movies_tuple[0])
#Christian Bale's movie
print(movies_tuple[1])
#George Clooney's movie
print(movies_tuple[2])
```

The third built-in data structure that we covered also has a specific limitation: the dictionary does not have an option to deal with key errors. Simply put, a key error is where one passes a wrong or non-existent key into a Python dictionary to get its associated value or values. As shown in Figure 27, if you use the dictionary in Python 3.x, None will be returned upon entering an incorrect key. Since that response is vague, developers might want to return an error message by setting a default value for an incorrect key, and continue operations as usual. In some cases, you might want to add that key to the dictionary with the default value too (Ramos, 2023).

Figure 27

Searching a dictionary with a non-existent key

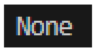


```
actors_to_movies_dict = {"Christian Bale": "The Machinist",
                         "Kate Beckinsale": "Underworld","George Clooney": "Syriana"}
print(actors_to_movies_dict.get("Anne Hathaway"))
```

As shown in Figure 28, here is the output when you look for a non-existent key:

Figure 28

Searching a dictionary with a non-existent key output

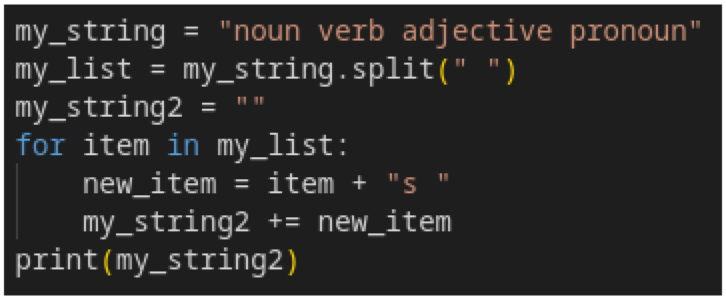


```
None
```

Finally, strings offer a variety of methods by which certain operations can be carried out. However, one might want to create their own methods customized to a solution when using this data type (Ramos, 2023). For example, you might want to produce specific string output that is not possible otherwise. As shown in Figure 29, you might want to add “s” to a list of words:

Figure 29

Adding the character "s" to several words

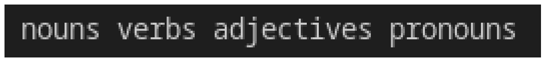


```
my_string = "noun verb adjective pronoun"
my_list = my_string.split(" ")
my_string2 = ""
for item in my_list:
    new_item = item + "s "
    my_string2 += new_item
print(my_string2)
```

As shown in Figure 30, here is the final output:

Figure 30

Output when adding the character "s"



```
nouns verbs adjectives pronouns
```

You might want to even reduce instances of double space to single space or want to compute the length of words in a sentence. As you can tell, it really depends on the solution you are building and what the requirements are. Speaking of which, the Collections module present in the standard library provides developers with containers to handle these problems properly ("collections", 2023; Romano, 2018).

Your First Look at the Collections Module

Given the mentioned areas of improvement elicited in the previous section, it is only fair to assume that Python's built-in data structures will not be useful to developers in certain scenarios. One might wonder what alternatives are available, thanks to a rich and diverse ecosystem of standard and third-party libraries that Python has. Yes, there is one such option that offers containers that serve as alternatives to the built-in data structures that we discussed earlier. We are talking about the Collections module. In this case, it offers containers and classes such as namedtuple, deque, ChainMap, Counter, OrderedDict, defaultdict, UserDict, UserList and UserString (“collections”, 2023; Romano, 2018).

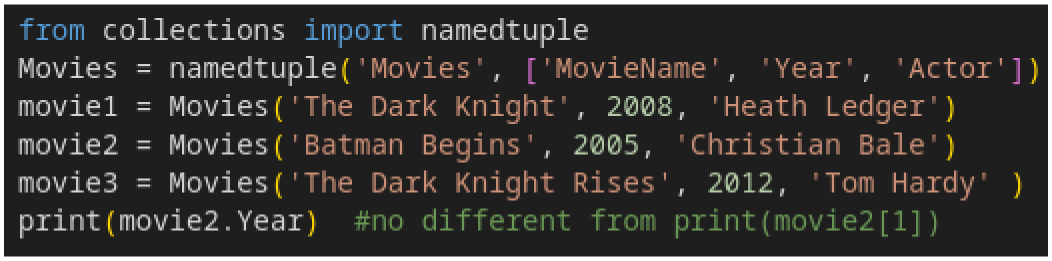
Right off the bat, ChainMap, Counter and OrderedDict offer unique solutions by chaining together dictionaries into a list, tallying items in lists, tuples and dictionaries and remembering the order in which items are inserted into a dictionary (“What is the Collections”, 2023). We will set these aside to look at the containers and wrapper classes such as the namedtuple, deque, defaultdict as well as UserString (“collections”, 2023; Romano, 2018). It will be well worth examining how these containers can be used instead of the usual built-in data structures that we have covered so far in this paper. One last thing: since the Collections module is present in the Python Standard Library, you do not have to run pip to install this module. All that will be necessary is the from-import statement that should get you started with these data structures (Lutz, 2013; Romano, 2018).

#1: namedtuple

Now, the namedtuple, as you might already know, is a solution for the scenario where you might need to label your indices in a built-in tuple with field names (Ramos, 2023; Romano, 2018). Let us say you want to save certain details about a movie as a tuple. When you declare a namedtuple object, the first parameter is the type while the second parameter includes the attributes of that type. As shown in Figure 31, the type would be Movies while the attributes are Movie Name, Year Released and Standout Actor:

Figure 31

Creating namedtuple objects

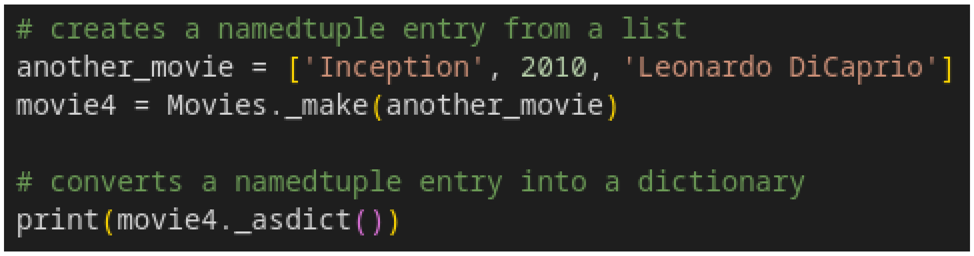

```
from collections import namedtuple
Movies = namedtuple('Movies', ['MovieName', 'Year', 'Actor'])
movie1 = Movies('The Dark Knight', 2008, 'Heath Ledger')
movie2 = Movies('Batman Begins', 2005, 'Christian Bale')
movie3 = Movies('The Dark Knight Rises', 2012, 'Tom Hardy' )
print(movie2.Year)  #no different from print(movie2[1])
```

When you look at the print statement, you will notice that to access index position [1] of the namedtuple, the attribute ‘Year’ has been used. The output you should obtain is the year 2005. Clearly, there is no need to use index positions anymore, even if this method of accessing the tuple data remains available to the developer for this data structure (“collections”, 2023; Romano, 2018).

As shown in Figure 32, you can also run two interesting methods such as the ._make(iterable) and the .asdict() to create a namedtuple from a list or tuple but also convert a namedtuple to a dictionary (“collections”, 2023):

Figure 32

Using the ._make and .asdict methods for namedtuple

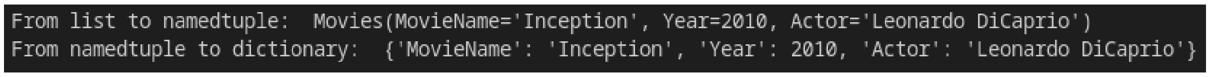


```
# creates a namedtuple entry from a list
another_movie = ['Inception', 2010, 'Leonardo DiCaprio']
movie4 = Movies._make(another_movie)

# converts a namedtuple entry into a dictionary
print(movie4._asdict())
```

When you look at the output shown below, the list another_movie has been added as a Movies namedtuple. Also, as shown in Figure 33, when running the second method on this namedtuple, a dictionary is returned with the right key-value pairs:

Figure 33

Namedtuple methods output



```
From list to namedtuple:  Movies(MovieName='Inception', Year=2010, Actor='Leonardo DiCaprio')
From namedtuple to dictionary:  {'MovieName': 'Inception', 'Year': 2010, 'Actor': 'Leonardo DiCaprio'}
```

You can even retrieve a value using the getattr(namedtuple, some_attribute) function while also creating new namedtuples after replacing values as well as identifying the fields in said namedtuple too. For the latter two, you will use the ._replace(field = value) and the ._fields() methods (“collections”, 2023).

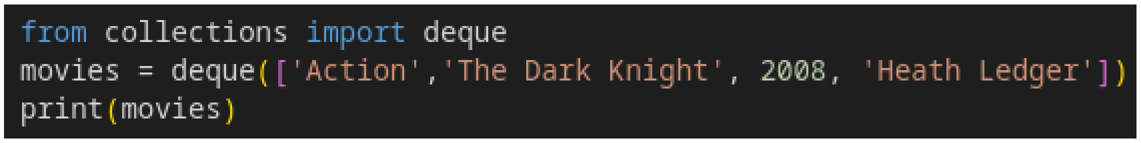
#2: Deque

Even if the name might not ring a bell, this data structure serves as an excellent alternative for the list, where the time and space complexity goes up to O(n) for certain pop and append operations. Since the list cannot deliver in this respect, the deque is a fallback option for the same, thanks to the methods that offer such functionality in the first place (Ramos, 2023). Speaking of which, let us continue with the theme of movies, where we wish to add and pop certain things about "The Dark Knight" to this deque.

As shown in Figure 34, we create a deque with the list of elements:

Figure 34

Create a deque

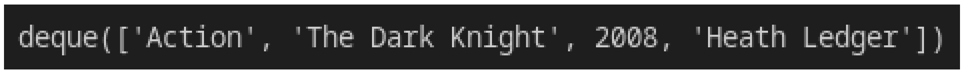


```
from collections import deque
movies = deque(['Action','The Dark Knight', 2008, 'Heath Ledger'])
print(movies)
```

Once this is done, the output is as shown in Figure 35:

Figure 35

Newly created Deque output

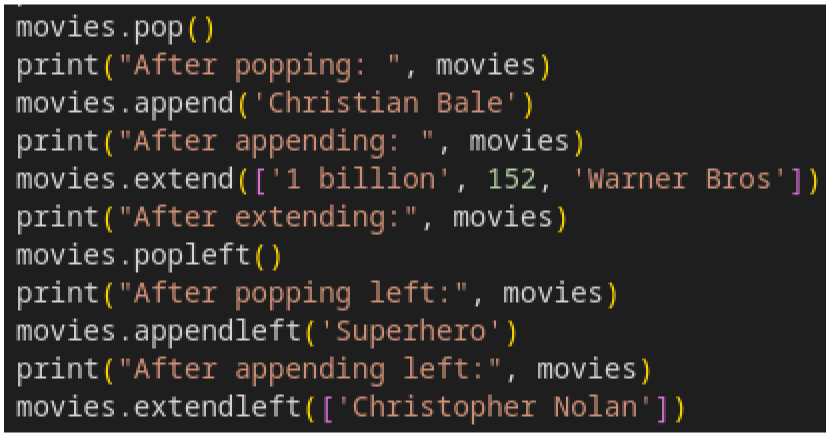


```
deque(['Action', 'The Dark Knight', 2008, 'Heath Ledger'])
```

Now, if we want to add or delete elements in this deque, we must use the pop, append, popleft, appendleft, extend and extendleft methods, as shown in Figure 36 ("collections", 2023):

Figure 36

Deque methods

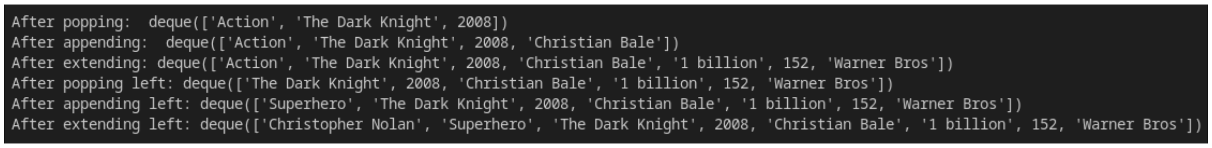


```
movies.pop()
print("After popping: ", movies)
movies.append('Christian Bale')
print("After appending: ", movies)
movies.extend(['1 billion', 152, 'Warner Bros'])
print("After extending:", movies)
movies.popleft()
print("After popping left:", movies)
movies.appendleft('Superhero')
print("After appending left:", movies)
movies.extendleft(['Christopher Nolan'])
```

As shown in Figure 37, here is the output:

Figure 37

Deque methods output

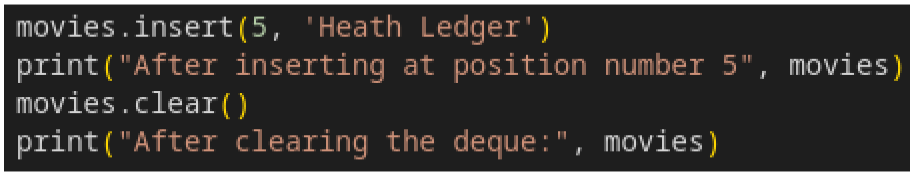


```
After popping:  deque(['Action', 'The Dark Knight', 2008])
After appending:  deque(['Action', 'The Dark Knight', 2008, 'Christian Bale'])
After extending: deque(['Action', 'The Dark Knight', 2008, 'Christian Bale', '1 billion', 152, 'Warner Bros'])
After popping left: deque(['The Dark Knight', 2008, 'Christian Bale', '1 billion', 152, 'Warner Bros'])
After appending left: deque(['Superhero', 'The Dark Knight', 2008, 'Christian Bale', '1 billion', 152, 'Warner Bros'])
After extending left: deque(['Christopher Nolan', 'Superhero', 'The Dark Knight', 2008, 'Christian Bale', '1 billion', 152, 'Warner Bros'])
```

Now, you can also insert an element at a particular index position in the deque and even clear the entire container as you would a list (“collections”, 2023). As shown in Figure 38, this is how you would do it:

Figure 38

Using list methods for Deque

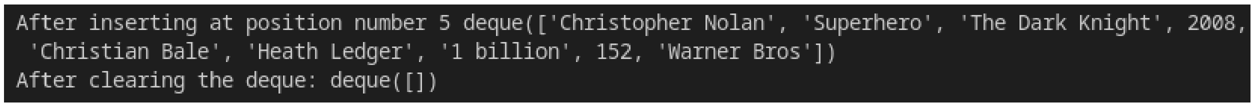


```
movies.insert(5, 'Heath Ledger')
print("After inserting at position number 5", movies)
movies.clear()
print("After clearing the deque:", movies)
```

As shown in Figure 39, here is the output that these two methods generate:

Figure 39

List methods for Deque output



```
After inserting at position number 5 deque(['Christopher Nolan', 'Superhero', 'The Dark Knight', 2008,
 'Christian Bale', 'Heath Ledger', '1 billion', 152, 'Warner Bros'])
After clearing the deque: deque([])
```

Now remember: not only can the developer append left and pop left items at O(1) but also use the Python list methods discussed earlier. As you might have understood by now, the deque overcomes the limitation of the list and gives additional options to developers who work with lists in such a way that it drastically improves the speed at which the items can be used. So, this is another example why the Collections module is an excellent addition to the Python standard library (“collections”, 2023; Ramos, 2023).

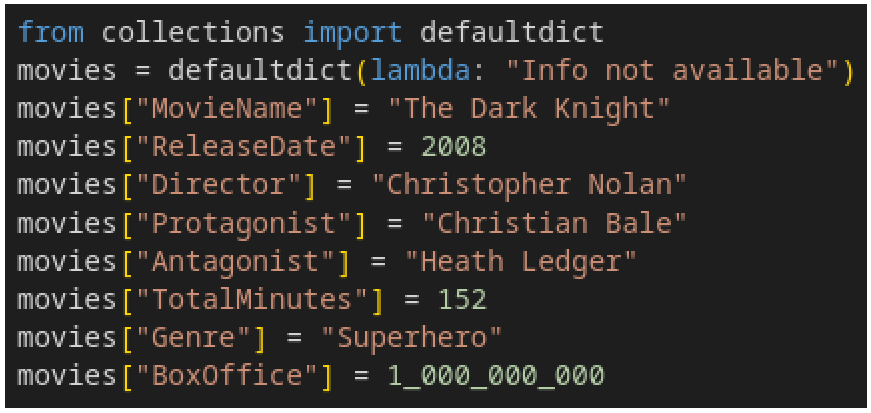
#3: defaultdict

The third and final data structure that we covered – the dictionary – also has certain limitations that we have discussed in a prior section. In earlier versions of Python, it was not possible for dictionaries to remember the order in which the key-value pairs were inserted and which was why the orderedDict container was added to the Collections module. Only now, in Python 3.x, this feature has been added to the built-in dictionary, so that is that (Ramos, 2023).

However, there is one more limitation that still needs to be addressed: the possibility of adding new keys and values to a dictionary while responding to the use of incorrect keys (Romano, 2018). Usually, developers must write code to handle such an exception: when a non-existent key is used. Throwing exceptions is the general way to deal with such an issue. However, using the defaultdict container can ensure that None is not returned because a default value is assigned as soon as this container is declared and initialized (Ramos, 2023). Continuing with the movies theme, let us add a few key-value pairs to the defaultdict data structure, as shown in Figure 40:

Figure 40

Adding key-value pairs to defaultdict structure

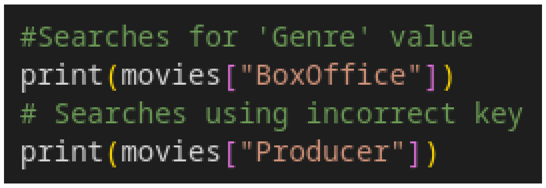


```
from collections import defaultdict
movies = defaultdict(lambda: "Info not available")
movies["MovieName"] = "The Dark Knight"
movies["ReleaseDate"] = 2008
movies["Director"] = "Christopher Nolan"
movies["Protagonist"] = "Christian Bale"
movies["Antagonist"] = "Heath Ledger"
movies["TotalMinutes"] = 152
movies["Genre"] = "Superhero"
movies["BoxOffice"] = 1_000_000_000
```

So, let us perform a search with an accurate key and a non-existent one, as shown in Figure 41:

Figure 41

Searching for an accurate key and an incorrect one

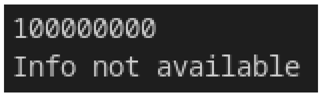


```
#Searches for 'Genre' value
print(movies["BoxOffice"])
# Searches using incorrect key
print(movies["Producer"])
```

While we have passed in a key-value pair for the key “BoxOffice” there is no such key-value pair for “Producer”. When we run this code on the movies defaultdict, we obtain the output as shown in Figure 42:

Figure 42

Searching with accurate and inaccurate key output

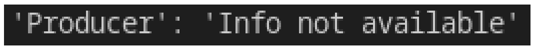


```
1000000000
Info not available
```

We get 1 billion for the first and value “Info not available” for the second. In other words, by default, if the key passed in does not exist, the value associated with the lambda function is returned (“collections”, 2023). As shown in Figure 43, the default value is added to the dictionary along with the non-existent key if only to deal properly with such a retrieval operation in the future (Romano, 2018).

Figure 43

Incorrect Search key added with default value to defaultdict

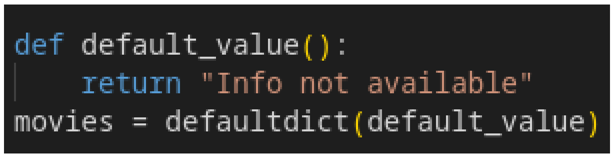


```
'Producer': 'Info not available'
```

As shown in Figure 44, if you do not fancy lambda functions, there is a simpler way by which you can assign the default value:

Figure 44

Assign default value using normal functions

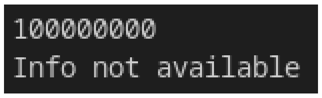


```
def default_value():
    return "Info not available"
movies = defaultdict(default_value)
```

Instead of using the “lambda: Info not available” parameter that is conventionally known as the default_factory, an actual function is defined (“collections”, 2023). As shown in Figure 45, the output is still the same:

Figure 45

Searching with accurate and inaccurate key output

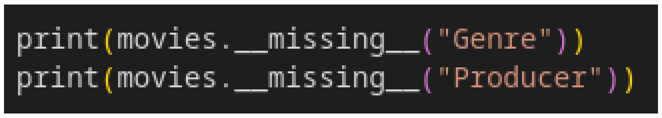


```
100000000
Info not available
```

You can also use the list, dict and int parameters as part of a defaultdict object, where in the case of a non-existent key, Python will return an empty list, dictionary, or the value 0 (Romano, 2018). As demonstrated in Figure 46, there is one method for this container called .__missing__(key) that returns the default value selected for the defaultdict in question (“collections”, 2023).

Figure 46

The __missing__(key) method

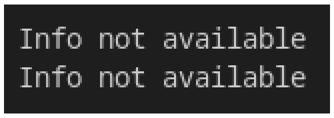


```
print(movies.__missing__("Genre"))
print(movies.__missing__("Producer"))
```

As shown in Figure 47, here is the output when running this method on the existing key "Genre" and a non-existent one named "Producer":

Figure 47

Output for the __missing__(key) method

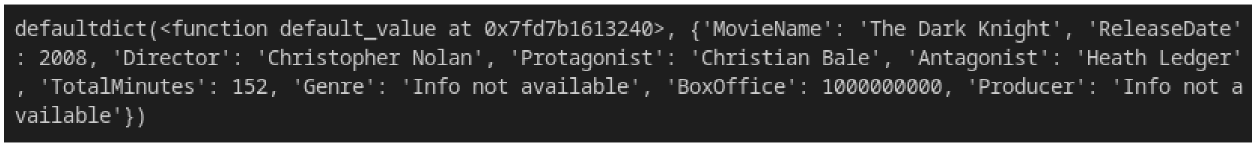


```
Info not available
Info not available
```

Also, along with seeing the default value for a missing key, it is added to the dictionary for both missing keys ("collections", 2023). As you can see in Figure 48, the value for "Genre" has been changed to the default value while the Producer key has been added to the default dictionary with the default value:

Figure 48

Adding default values with incorrect keys to defaultdict



```
defaultdict(<function default_value at 0x7fd7b1613240>, {'MovieName': 'The Dark Knight', 'ReleaseDate'
: 2008, 'Director': 'Christopher Nolan', 'Protagonist': 'Christian Bale', 'Antagonist': 'Heath Ledger'
, 'TotalMinutes': 152, 'Genre': 'Info not available', 'BoxOffice': 1000000000, 'Producer': 'Info not a
vailable'})
```

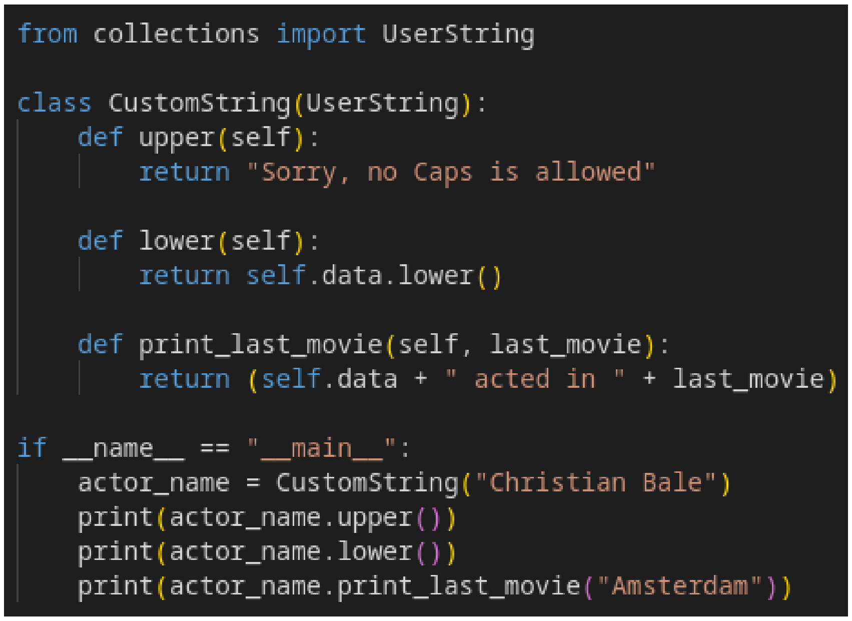
As one might surmise, if there are any future searches using these keys, then by virtue of being stored, the same default value will be returned to the user.

#4: UserString

Now, to extend functionality for lists and dictionaries, there are three wrapper sub-classes made available to developers. Otherwise known as UserList, UserDict and UserString, objects can be created using these classes to build functionality that is unique to said project (Ramos, 2023). Since we have already covered solutions for the other two data structures, we will just focus on UserString now. When you instantiate an object of this class, you will store a string in the attribute named ‘data’ as shown in Figure 49 (“collections”, 2023):

Figure 49

Creating a UserString class

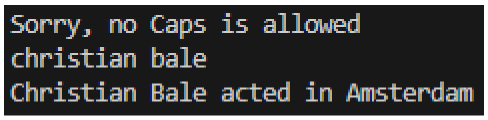


```
from collections import UserString

class CustomString(UserString):
    def upper(self):
        return "Sorry, no Caps is allowed"

    def lower(self):
        return self.data.lower()

    def print_last_movie(self, last_movie):
        return (self.data + " acted in " + last_movie)

if __name__ == "__main__":
    actor_name = CustomString("Christian Bale")
    print(actor_name.upper())
    print(actor_name.lower())
    print(actor_name.print_last_movie("Amsterdam"))
```

Now, what is important about using this class is being able to create unique functionality for the strings that you pass in. Not only will you be able to modify strings based on the project requirements and provide it as a method but also use built-in methods for strings (Shrivarsheni, 2023). Best part: you can even override certain built-in methods, as shown in Figure 50:

Figure 50

A UserString class output



```
Sorry, no Caps is allowed
christian bale
Christian Bale acted in Amsterdam
```

Clearly, the focus here is to extend the functionality of strings beyond what is available to include unique string methods based on said project requirements (Ramos, 2023). With that said, let us address the relationship between these data structures and the coding assignment.

Relationship Between Python Data Structures and The Practical Assignment

As mentioned in an earlier section, the built-in data structures come in handy when it comes to performing any kind of computations involving datasets (Gautam, 2023). When you read the code in the Appendix, lists, sets and dictionaries were used along with variables, objects, and their attributes for this assignment and which simplified how we arrived at the final ideal function. However, there was no need to use these specialized containers from the Collections module since the project requirements were standard (Ramos, 2023).

Let me elaborate: since only 100 and 400 data points for the fields in the test or train and ideal sets were used, there was no issue with efficiency ("collections", 2023). All computations were complete in seconds. This is because the time-consuming part of such a solution involved writing to and reading from a database.

Since we worked with signed integer data after reading them from csv files, the need for the namedtuple container did not arise as tuples were not necessary to begin with (Elemati, 2023; Romano, 2018). Instead, in separate instances, Pandas data frames and lists were used for the visualizations and computations. Dictionaries were used to map ideal function numbers to their SSR values and to map the final ideal function to the test set (Lutz, 2013; Valchanov, 2023). So, it is very possible that the deque and namedtuple containers could have been used even if the benefits that they offer would not be substantial ("collections", 2023). Of course, if the dataset was appreciably larger, then their use would have been justified. Still, the standard built-in data structures worked just as well since efficiency was not a given assignment constraint.

Keeping this in mind, the relationship of the practical assignment to my written paper is minimal as only the built-in data structures were used. Yet there is a good reason for this: as mentioned in the previous section of this paper, the need for containers in the Collections module occurs for certain scenarios (Ramos, 2023). This project did not require the use of such containers even if they were totally worth exploring in my paper, if only to deepen my knowledge of this amazing programming language.

Conclusion

As we come to the end of this paper, one cannot help but surmise that Python takes great care to offer data structures for general use but also takes it a step further by including containers for specific situations where developers might need to perform additional operations or have greater efficiency at their disposal. Literally, all the structures that we looked at in the Collections module help developers do just that to optimize their code, in terms of time and space complexity among other aspects (Ramos, 2023; Kumari, 2023).

However, another benefit of working with containers and wrapper classes from the Collections module is that they are easy to implement, as shown in the screenshots above. With the Python Docs being an excellent source of information, one can find the available functionality for each of these containers as well as the wrapper classes quite easily (“collections”, 2023).

As for my findings, the containers and wrapper classes discussed above are excellent options to consider when working with large datasets. Particularly, if efficiency, ease of use and additional operations are vital to the success of the project. From a personal standpoint, deque, namedtuple and UserString are worth taking seriously and will prove to be most useful during the next stage of my journey into Python (Ramos, 2023; Romano, 2018).

References

Romano, F. (2018). Learn Python Programming (2nd ed.). Packt Publishing Ltd.

Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.

Gautam, S. (2023). Why Data Structures and Algorithms are Important? Enjoy Algorithms. https://www.enjoyalgorithms.com/blog/why-should-we-learn-data-structure-and-algorithms

Geeks for Geeks. (2023, September 27). Python Tutorial. https://www.geeksforgeeks.org/python-programming-language/

Singh, S. (2021, May 31). Difference between Database and Data Structure. Geeks for Geeks. https://www.geeksforgeeks.org/difference-between-database-and-data-structure/

Ramos, L. (2023). Python's collections: A Buffet of Specialized Data Types. Real Python. https://realpython.com/python-collections-module/

Busbee, K & Braunschweig, D. (2018). Constants and Variables. Rebus Community. https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/

Rouse, M. (2017, February 15). Memory Address. Techopedia. https://www.techopedia.com/definition/323/memory-address

Biswas, P. (2022, February 11). Why Learn Data Structures and Algorithms? Scaler. https://www.scaler.com/topics/data-structures/why-learn-data-structures-and-algorithms/

Aggarwal, N. (2023, June 8). Python Collections Module. Geeks for Geeks. https://www.geeksforgeeks.org/python-collections-module/

Python Lists. (2023). W3Schools. https://www.w3schools.com/python/python_lists.asp

Python - List Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_lists_methods.asp

Python Tuples. (2023). W3Schools. https://www.w3schools.com/python/python_tuples.asp

Python - Tuple Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_tuples_methods.asp

Python Dictionaries. (2023). W3Schools.
https://www.w3schools.com/python/python_dictionaries.asp

Python Dictionary Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_dictionaries_methods.asp

Python String. (2023). Geeks for Geeks. https://www.geeksforgeeks.org/python-string/

Python String Methods. (2023). W3Schools.
https://www.w3schools.com/python/python_ref_string.asp

Shahid. (2023). Python Data Structures Limitations and Solutions. Code for Geek.
https://codeforgeek.com/python-data-structures-limitations-and-solutions/#:~:text=The%20only%20limitation%20is%20that,available%20in%20the%20Collections%20module.

collections — Container datatypes. (2023). Python 3.12.0 documentation.
https://docs.python.org/3/library/collections.html#module-collections

Kumari, Y. (2023). Time & Space Complexity of Searching Algorithms. Coding Ninjas.
https://www.codingninjas.com/studio/library/time-space-complexity-of-searching-algorithms

Elemati, K. (2023). Collections in Python. Tutorials Point.
https://www.tutorialspoint.com/collections-in-python

Shrivarsheni. (2023). Python Collections – An Introductory Guide. Machine Learning Plus.

https://www.machinelearningplus.com/python-collections-guide/

What is the Python collections module? (2023). Educative.

https://www.educative.io/answers/what-is-the-python-collections-module

Valchanov, I. (2023). Sum of Squares: SST, SSR, SSE. 365 Data Science.

https://365datascience.com/tutorials/statistics-tutorials/sum-squares/

Appendix: Linear Regression & Unit Tests Code

```
import pandas as pd

import numpy as np

import csv

import sqlalchemy as db

import os, math

from sklearn.linear_model import LinearRegression

import seaborn as sns

import matplotlib.pyplot as plt

class Data:

    def __init__(self, path):

        self.path = path

    def readcsvfile(self, dataset):

        '''

        Reads csv file if file is available and returns it as a dataframe

        '''

        if os.path.isfile(self.path+dataset):

            final_set = pd.read_csv(self.path+dataset)

            return final_set

        else:

            return None

    def createvisualization(self,x,y, title, dataset):

        '''
        Creates and returns the visualization for the test and selected ideal function data
        '''
        if len(dataset) > 0:
            # displays the title, x & y labels and zooms each visualization
            plt.title(title)
            plt.xlabel(x)
            plt.ylabel(y)
            # draw regplot using Seaborn without confidence interval
            rval = sns.regplot(x = x,y = y,data = dataset, ci=None)
            rval.figure.set_size_inches(10,6)
            # show the plot
            plt.show()
            return True
        else:
            return False
class Database(Data):
    def __init__(self, path):
        self.path = path
        Data.__init__(self, path)


    def connectdb(self):
        '''
        Connects to the sqlite database using sqlalchemy and returns an engine object
        '''
        try:
            # used sqlalchemy 1.4.49 version (maintenance)
            engine = db.create_engine(self.path, echo = True)
            return engine
        except:
            return None
    def createtable(self, table, engine):
        '''
        Creates a table for the train, ideal and test sets and returns true
        '''
        # get meta data object
        meta_data = db.MetaData()
        # get connection object
        connection = engine.connect()
        table_list = ["train", "ideal", "test"]
        if table in table_list:
            if table == "train":
                train_table = db.Table("train", meta_data,
                db.Column("x", db.Float, nullable=False),
                db.Column("y1", db.Float, nullable=False),
                db.Column("y2", db.Float, nullable=False),
                db.Column("y3", db.Float, nullable=False),
            db.Column("y4", db.Float, nullable=False))
          elif table == "ideal":
            ideal_table = db.Table("ideal", meta_data,
            db.Column("x", db.Float, nullable=False),db.Column("y1", db.Float,
    nullable=False),db.Column("y2", db.Float, nullable=False),db.Column("y3", db.Float,
    nullable=False),db.Column("y4", db.Float, nullable=False), db.Column("y5", db.Float,
    nullable=False), db.Column("y6", db.Float, nullable=False), db.Column("y7", db.Float,
    nullable=False),db.Column("y8", db.Float, nullable=False),db.Column("y9", db.Float,
    nullable=False),db.Column("y10", db.Float, nullable=False),db.Column("y11", db.Float,
    nullable=False),db.Column("y12", db.Float, nullable=False),db.Column("y13", db.Float,
    nullable=False),db.Column("y14", db.Float, nullable=False),db.Column("y15", db.Float,
    nullable=False),db.Column("y16", db.Float, nullable=False),db.Column("y17", db.Float,
    nullable=False),db.Column("y18", db.Float, nullable=False),db.Column("y19", db.Float,
    nullable=False),db.Column("y20", db.Float, nullable=False),db.Column("y21", db.Float,
    nullable=False),db.Column("y22", db.Float, nullable=False),db.Column("y23", db.Float,
    nullable=False),db.Column("y24", db.Float, nullable=False),db.Column("y25", db.Float,
    nullable=False),db.Column("y26", db.Float, nullable=False),db.Column("y27", db.Float,
    nullable=False),db.Column("y28", db.Float, nullable=False),db.Column("y29", db.Float,
    nullable=False),db.Column("y30", db.Float, nullable=False),db.Column("y31", db.Float,
    nullable=False),db.Column("y32", db.Float, nullable=False),db.Column("y33", db.Float,
    nullable=False),db.Column("y34", db.Float, nullable=False),db.Column("y35", db.Float,
    nullable=False),db.Column("y36", db.Float, nullable=False),db.Column("y37", db.Float,
    nullable=False),db.Column("y38", db.Float, nullable=False),db.Column("y39", db.Float,
    nullable=False),db.Column("y40", db.Float, nullable=False),db.Column("y41", db.Float,
    nullable=False),db.Column("y42", db.Float, nullable=False),db.Column("y43", db.Float,
    nullable=False),db.Column("y44", db.Float, nullable=False),db.Column("y45", db.Float,
    nullable=False),db.Column("y46", db.Float, nullable=False),db.Column("y47", db.Float,
    nullable=False),db.Column("y48", db.Float, nullable=False),db.Column("y49", db.Float,
    nullable=False),db.Column("y50", db.Float, nullable=False))
                elif table == "test":
                    test_table = db.Table("test", meta_data,
                    db.Column("x", db.Float, nullable=False),
                    db.Column("y", db.Float, nullable=False),
                    db.Column("ydev", db.Float, nullable=False),
                    db.Column("yideal", db.Float, nullable=False))
                # create test table and store the information in metadata
                meta_data.create_all(engine)
                return True
            else:
                return False
        def insertvalues(self, dataset, table, engine):
            '''
            Inserts values into the created table for train, ideal and test and returns True
            '''
            table_list = ["train", "ideal", "test"]
            if table in table_list:
```

```
if table == "train":
    # get meta data object
    meta_data = db.MetaData()
    # get connection object
    connection = engine.connect()
    # set actor creation script table
    train_table = db.Table(table, meta_data,autoload = True, autoload_with=engine)
    # insert data
    for lst in dataset:
        data = train_table.insert().values(x=lst[0], y1=lst[1], y2=lst[2],y3=lst[3],y4=lst[4])
        # execute the insert statement
        connection.execute(data)
    return True
elif table == "ideal":
    # get meta data object
    meta_data = db.MetaData()
    # get connection object
    connection = engine.connect()
    # set actor creation script table
    ideal_table = db.Table(table, meta_data,autoload = True, autoload_with=engine)
    # insert data
    for lst in dataset:
```

```
                data = ideal_table.insert().values(x=lst[0], y1=lst[1], y2=lst[2],y3=lst[3],y4=lst[4],
y5=lst[5],y6=lst[6],y7=lst[7],y8=lst[8], y9=lst[9], y10=lst[10], y11=lst[11],
y12=lst[12],y13=lst[13],y14=lst[14], y15=lst[15], y16=lst[16],y17=lst[17],
y18=lst[18], y19=lst[19], y20=lst[20], y21=lst[21], y22=lst[22],y23=lst[23],
y24=lst[24], y25=lst[25], y26=lst[26],y27=lst[27],y28=lst[28], y29=lst[29],
y30=lst[30], y31=lst[31], y32=lst[32],y33=lst[33],y34=lst[34], y35=lst[35],
y36=lst[36],y37=lst[37],y38=lst[38], y39=lst[39], y40=lst[40],
y41=lst[41], y42=lst[42],y43=lst[43],y44=lst[44], y45=lst[45],
y46=lst[46],y47=lst[47],y48=lst[48], y49=lst[49],y50=lst[50])
                # execute the insert statement
                connection.execute(data)
            return True
        elif table == "test":
            # get meta data object
            meta_data = db.MetaData()
            # get connection object
            connection = engine.connect()
            # set actor creation script table
            test_table = db.Table(table, meta_data,autoload = True, autoload_with=engine)
            # insert data
            for lst in dataset:
                data = test_table.insert().values(x=lst[0], y=lst[1], ydev=lst[2],yideal=lst[3])
                # execute the insert statement
                connection.execute(data)
            return True


    else:
        return False
def readvalues(self, table, engine):
    '''
    Reads values from the created tables for train, ideal and test and prints selected values
    '''
    table_list = ["train", "ideal", "test"]
    if table in table_list:
        if table == "train":
            # get meta data object
            meta_data = db.MetaData()
            # get connection object
            connection = engine.connect()
            # set actor creation script table
            train_table = db.Table("train", meta_data,autoload=True, autoload_with=engine)
            # set the select statement
            select_train = train_table.select()
            # execute the select statement
            result = connection.execute(select_train)
            alltrainrows = result.fetchmany(5)
            for row in alltrainrows:
                print(row)
            return True
        elif table == "ideal":
            # get meta data object
            meta_data = db.MetaData()
            # get connection object
            connection = engine.connect()
            # set actor creation script table
            ideal_table = db.Table("ideal", meta_data,autoload=True, autoload_with=engine)
            # set the select statement
            select_ideal = ideal_table.select()
            # execute the select statement
            result = connection.execute(select_ideal)
            allidealrows = result.fetchmany(5)
            for row in allidealrows:
                print(row)
            return True
        elif table == "test":
            # get meta data object
            meta_data = db.MetaData()
            # get connection object
            connection = engine.connect()
```

```
            # set actor creation script table
            test_table = db.Table("test", meta_data,autoload=True, autoload_with=engine)
            # set the select statement
            select_test = test_table.select()
            # execute the select statement
            result = connection.execute(select_test)
            alltestrows = result.fetchmany(5)
            for row in alltestrows:
                print(row)
            return True
        else:
            return False
class Computations:
    def SSR(self,set_x, sets_y, dataset):
        '''
        Computes SSR for x-y value pairs and saves the deviations for later use in the project
        '''
        if len(dataset) > 0:
            SSR_list, deviations_list,SSR_deviations_list  = [], [], []
            for y_value in sets_y:
                x = dataset[list(set_x)]
                y = dataset[y_value]
                regression_model = LinearRegression()
                regression_model.fit(x, y)
                y_pred = regression_model.predict(x)
                df_dev = pd.DataFrame({'Actual':y, 'Predicted':y_pred})
                SSR_list.append(str(np.sum(np.square(df_dev['Predicted']-df_dev['Actual']))))
                deviations_list = list(df_dev['Predicted']-df_dev['Actual'])
                SSR_deviations_list.append(SSR_list)
                SSR_deviations_list.append(deviations_list)
            return SSR_deviations_list
        else:
            return None
if __name__ == "__main__":
    # Read csv files
    # Due to inheriting the methods from the Data class, Database objects can also use the
readcsvfile & createvisualization methods
    read_visualize_data = Database("dataset\\")
    train = read_visualize_data.readcsvfile("train.csv")
    ideal = read_visualize_data.readcsvfile("ideal.csv")
    test = read_visualize_data.readcsvfile("test.csv")
    if len(train) == 0 or len(ideal) == 0 or len(test) == 0:
        raise Exception("Error. Incorrect path or CSV filename.")
    train_list, ideal_list, test_list = [], [], []
    for i in range(0,train.last_valid_index()+1):
        train_list.append(list(train.loc[i]))
```

```
for j in range(0,ideal.last_valid_index()+1):
    ideal_list.append(list(ideal.loc[j]))
for k in range(0,test.last_valid_index()+1):
    test_list.append(list(test.loc[k]))
# Create table for train and ideal
tables = Database("sqlite:///linearregressioncomp.db")
engine_train = tables.connectdb()
if engine_train != None:
    tables.createtable("train",engine_train)
    # insert values
    tables.insertvalues(train_list, "train", engine_train)
else:
    raise Exception("Engine object was not created")
engine_ideal = tables.connectdb()
if engine_ideal != None:
    tables.createtable("ideal",engine_ideal)
    # insert values
    tables.insertvalues(ideal_list, "ideal", engine_ideal)
else:
    raise Exception("Engine object was not created")
# Compute SSR for train and ideal
computeSSR = Computations()
train_set_x = 'x'
```

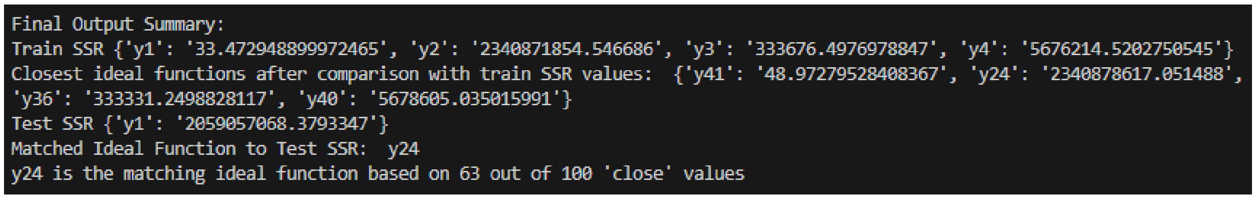
```
train_sets_y = ['y1', 'y2', 'y3', 'y4']
train_SSR = computeSSR.SSR(train_set_x, train_sets_y, train)
train_SSR_dict = {}
for i in range(0,len(train_SSR[0])):
    train_SSR_dict['y' + str(i+1)] = train_SSR[0][i]
ideal_set_x = 'x'
ideal_sets_y = ['y1', 'y2', 'y3', 'y4', 'y5', 'y6', 'y7', 'y8', 'y9', 'y10',
         'y11', 'y12', 'y13', 'y14', 'y15', 'y16', 'y17', 'y18', 'y19', 'y20',
         'y21', 'y22', 'y23', 'y24', 'y25', 'y26', 'y27', 'y28', 'y29', 'y30',
         'y31', 'y32', 'y33', 'y34', 'y35', 'y36', 'y37', 'y38', 'y39', 'y40',
         'y41', 'y42', 'y43', 'y44', 'y45', 'y46', 'y47', 'y48', 'y49', 'y50']
ideal_SSR = computeSSR.SSR(ideal_set_x, ideal_sets_y, ideal)
# Determine which ideal functions map closest to train
min_ssr_diff_list, min_ssr_diff_final = [],[]
for train_ssr_val in train_SSR[0]:
    min_ssr_diff_list = []
    for ideal_ssr_val in ideal_SSR[0]:
        min_ssr_diff_list.append(abs(float(train_ssr_val)-float(ideal_ssr_val)))
    min_ssr_diff_final.append(min_ssr_diff_list.index(min(min_ssr_diff_list))+1)
min_ssr_values = []
for pos in min_ssr_diff_final:
    min_ssr_values.append(ideal_SSR[0][pos-1])
min_ssr_final = {}
```

```
for i in range(0, len(min_ssr_values)):
    min_ssr_final['y' + str(min_ssr_diff_final[i])] = min_ssr_values[i]
# Compute test SSR
test_set_x = 'x'
test_sets_y = ['y']
test_SSR_dict = {}
test_SSR = computeSSR.SSR(test_set_x, test_sets_y, test)
test_SSR_dict['y' + str(1)] = test_SSR[0][0]
# Determine which ideal function SSR value does not exceed test SSR by sqrt(2)
for value in test_SSR[0]:
    for value2 in min_ssr_values:
        upper_bound = float(value)*math.sqrt(2)
        lower_bound = float(value)/math.sqrt(2)
        if float(value2) < upper_bound and float(value2) > lower_bound:
            matched_ideal_fn = min_ssr_diff_final[min_ssr_values.index(value2)]
# Find out how many values in test map to each of the functions and select that ideal func
test_list_y = []
for lst in test_list:
    test_list_y.append(lst[1])
ideal_func_list = []
for lst in ideal_list:
    ideal_func = []
    ideal_func.append(lst[min_ssr_diff_final[0]])
    ideal_func.append(lst[min_ssr_diff_final[1]])
    ideal_func.append(lst[min_ssr_diff_final[2]])
    ideal_func.append(lst[min_ssr_diff_final[3]])
    ideal_func_list.append(ideal_func)
ideal_fn, ideal_fn2, ideal_fn3, ideal_fn4, ideal_fn_final = [], [], [], [], []
for y in test_list_y:
    for lst in ideal_func_list:
        if y > (lst[0]/math.sqrt(2)) and y < (lst[0]*math.sqrt(2)):
            ideal_fn.append(y)
        elif y > (lst[1]/math.sqrt(2)) and y < (lst[1]*math.sqrt(2)):
            ideal_fn2.append(y)
        elif y > (lst[2]/math.sqrt(2)) and y < (lst[2]*math.sqrt(2)):
            ideal_fn3.append(y)
        elif y > (lst[3]/math.sqrt(2)) and y < (lst[3]*math.sqrt(2)):
            ideal_fn4.append(y)
ideal_fn_final_dict = {}
ideal_fn_final.append(len(set(ideal_fn)))
ideal_fn_final.append(len(set(ideal_fn2)))
ideal_fn_final.append(len(set(ideal_fn3)))
ideal_fn_final.append(len(set(ideal_fn4)))
for i in range(0, len(ideal_fn_final)):
    ideal_fn_final_dict[min_ssr_diff_final[i]] = ideal_fn_final[i]
ideal_max_value = max(ideal_fn_final_dict.values())
```

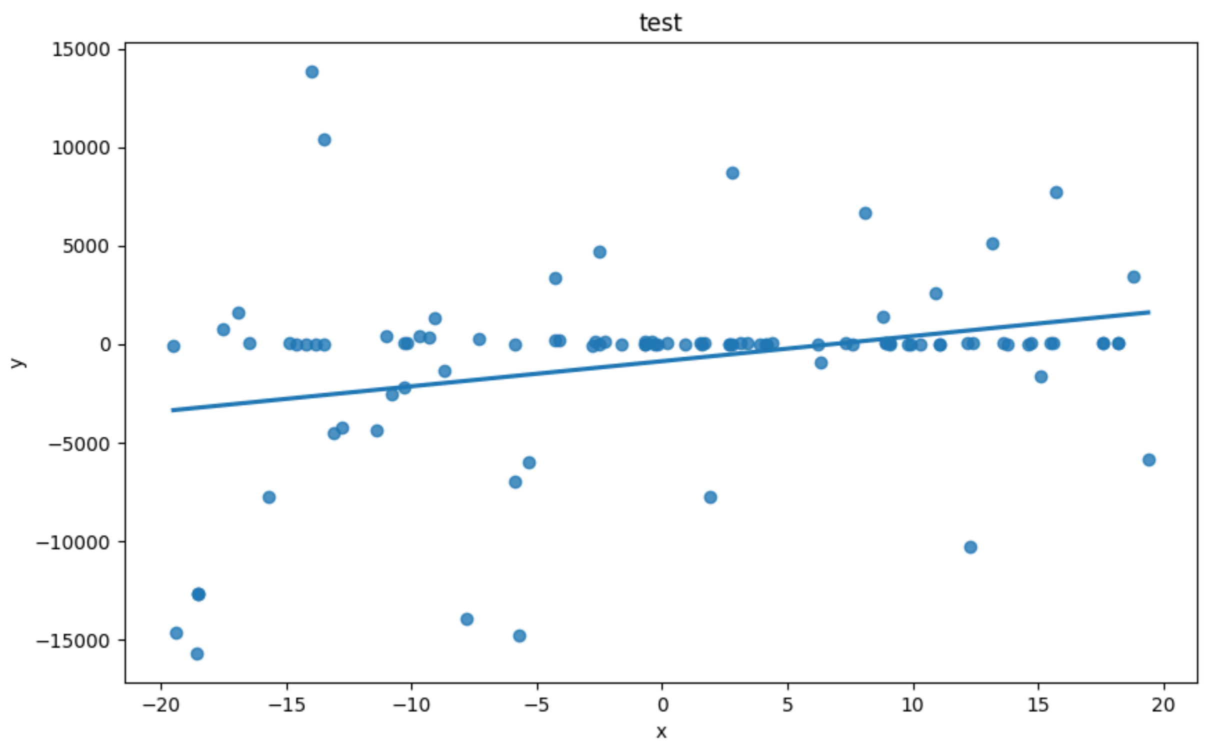
```
for k,v in ideal_fn_final_dict.items():
    if v == ideal_max_value:
        ideal_max_key = "y" + str(k)
# Add test x-y values, test deviation and ideal func y-values into db
test_x_final_list, test_y_final_list, test_dev_final_list, ideal_func_final_list = [],[],[],[]
for i in range(0, len(test_list)):
    test_x_final_list.append(test_list[i][0])
    test_y_final_list.append(test_list[i][1])
    test_dev_final_list.append(test_SSR[1][i])
for j in range(0, len(ideal_func_list)):
    ideal_func_final_list.append(ideal_func_list[j][1])
for k in range(0,300):
    test_x_final_list.append(0)
    test_y_final_list.append(0)
    test_dev_final_list.append(0)
test_final = []
for pos_val in range(0, len(ideal_func_final_list)):
    test_db_list = []
    test_db_list.append(test_x_final_list[pos_val])
    test_db_list.append(test_y_final_list[pos_val])
    test_db_list.append(test_dev_final_list[pos_val])
    test_db_list.append(ideal_func_final_list[pos_val])
    test_final.append(test_db_list)
    engine_test = tables.connectdb()
    if engine_test != None:
        tables.createtable("test",engine_test)
        # insert values
        tables.insertvalues(test_final, "test", engine_test)
    else:
        raise Exception("Engine object was not created")
    # Read 5 values each from train, ideal and test db tables
    print('Reading values from database...')
    tables.readvalues("train", engine_train)
    tables.readvalues("ideal", engine_ideal)
    tables.readvalues("test", engine_test)
    #Output
    print("Final Output Summary:")
    print("Train SSR", train_SSR_dict)
    print("Closest ideal functions after comparison with train SSR values: ", min_ssr_final)
    print("Test SSR", test_SSR_dict)
    print("Matched Ideal Function to Test SSR: ", 'y' + str(matched_ideal_fn))
    print(ideal_max_key + " is the matching ideal function based on", ideal_max_value ,"out of
100 'close' values")
    # Add matplotlib visualization for test and selected ideal function
    df_test = pd.DataFrame(test_list, columns=['x','y'])
    read_visualize_data.createvisualization('x', 'y', 'test', df_test)
```

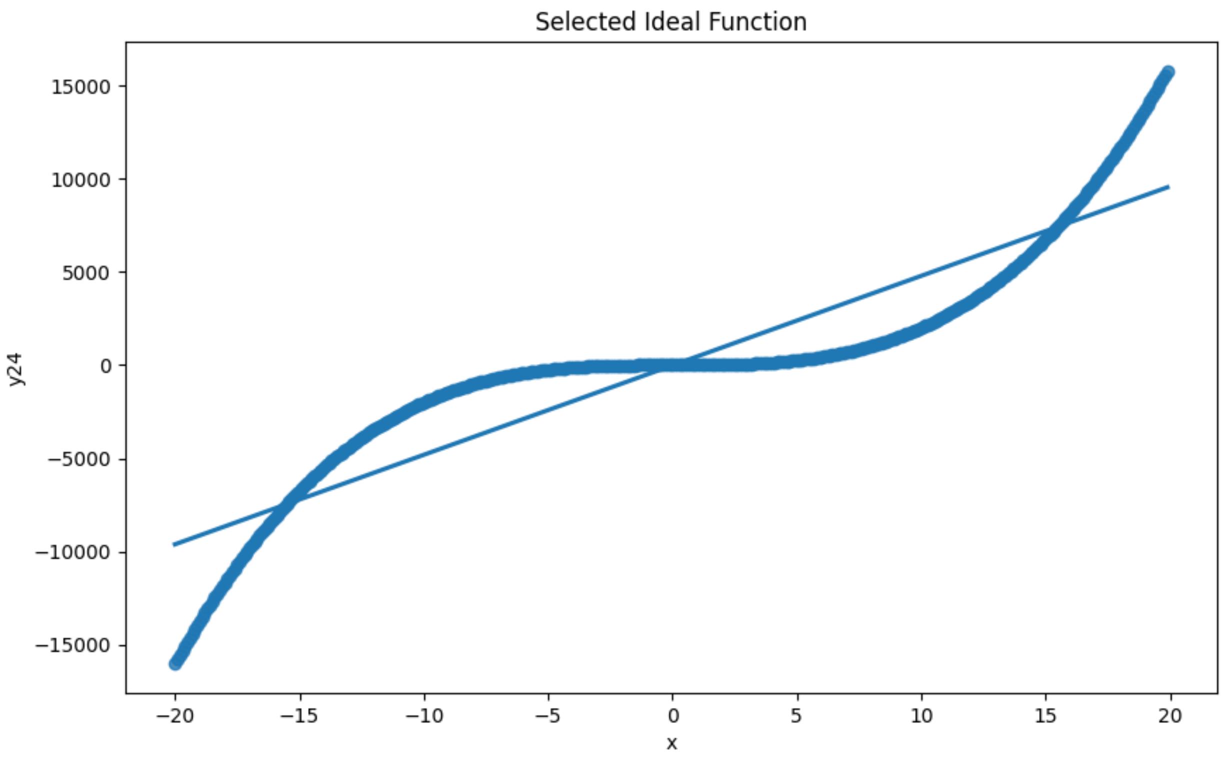
```
# create df for ideal function visualization
ideal_x = ideal['x'].to_list()
ideal_y = ideal[ideal_max_key].to_list()
ideal_final_list = []
for i in range(0, len(ideal_x)):
    ideal_list_final = []
    ideal_list_final.append(ideal_x[i])
    ideal_list_final.append(ideal_y[i])
    ideal_final_list.append(ideal_list_final)
df_ideal = pd.DataFrame(ideal_final_list, columns = ['x', ideal_max_key])
read_visualize_data.createvisualization('x', ideal_max_key, 'Selected Ideal Function', df_ideal)
# Please check the regressionunittests.py file for unit tests
```

Expected Output:



```
Final Output Summary:
Train SSR {'y1': '33.472948899972465', 'y2': '2340871854.546686', 'y3': '333676.4976978847', 'y4': '5676214.5202750545'}
Closest ideal functions after comparison with train SSR values:  {'y41': '48.97279528408367', 'y24': '2340878617.051488',
'y36': '333331.2498828117', 'y40': '5678605.035015991'}
Test SSR {'y1': '2059057068.3793347'}
Matched Ideal Function to Test SSR:  y24
y24 is the matching ideal function based on 63 out of 100 'close' values
```





import unittest, pandas as pd

```
from linearregression import Data, Database, Computations
class TestLinearRegression(unittest.TestCase):
   def test_readcsvfile(self):
      """
      Tests whether the function returns a dataset or None
      """
      test_csvfile = Data("dataset\\")
      self.assertIsNone(test_csvfile.readcsvfile(""))
   def testcreatevisualization(self):
      '''
      Tests whether the visualization returns True or False
      '''
      test_visualization = Data("dataset\\")
      test_list = pd.read_csv("dataset\\test.csv")
      df_test = pd.DataFrame(test_list, columns=['x','y'])
      self.assertTrue(test_visualization.createvisualization("x", "y", "test", df_test))
   def testconnectdb(self):
      '''
      Tests whether the database connection results in None or otherwise
      '''
      test_connectdb = Database("sqlite:///linearregressioncomp.db")
      self.assertIsNotNone(test_connectdb.connectdb())
   def testconnectdb2(self):
      '''
      Tests whether the database connections results in None or not
      '''
      test_connectdb2 = Database("")
      self.assertIsNone(test_connectdb2.connectdb())
   def testcreatetable(self):
      '''
      Tests whether a table has been created for test
      '''
      test_createtable = Database("sqlite:///linearregressioncomp.db")
      self.assertTrue(test_createtable.createtable("test", test_createtable.connectdb()))


   def testcreatetable2(self):
      '''
      Tests whether a table is created for an incorrect table name
      '''
      test_createtable2 = Database("sqlite:///linearregressioncomp.db")
      self.assertFalse(test_createtable2.createtable("yelp", test_createtable2.connectdb()))
   def testinsertvalues(self):
      '''
      Tests whether values are being inserted into the train table
      '''
      csv_data = Data("dataset\\")
      train = csv_data.readcsvfile("train.csv")
      train_list = []
      for i in range(0,train.last_valid_index()+1):
          train_list.append(list(train.loc[i]))
      test_insertvalues = Database("sqlite:///linearregressioncomp.db")
      engine_train = test_insertvalues.connectdb()
      test_insertvalues.createtable("train", engine_train)
      self.assertTrue(test_insertvalues.insertvalues(train_list, "train", engine_train))
   def testreadvalues(self):
      '''
      Tests whether the train values can be read or not
      '''
      csv_data = Data("dataset\\")
      train = csv_data.readcsvfile("train.csv")
      train_list = []
      for i in range(0,train.last_valid_index()+1):
          train_list.append(list(train.loc[i]))
      test_readvalues = Database("sqlite:///linearregressioncomp.db")
      engine_train = test_readvalues.connectdb()
      test_readvalues.createtable("train", engine_train)
      test_readvalues.insertvalues(train_list, "train", engine_train)
      self.assertTrue(test_readvalues.readvalues("train",engine_train))
   def testSSR(self):
          '''
          Tests whether SSR values can be computed for the test set
          '''
          csv_data = Data("dataset\\")
          train = csv_data.readcsvfile("train.csv")
          computeSSR = Computations()
          set_x = 'x'
          sets_y = ['y1', 'y2', 'y3', 'y4']
          self.assertIsNotNone(computeSSR.SSR(set_x, sets_y, train))
   if __name__ == "__main__":
      unittest.main()
```