Analysis of Data Structures: Dictionaries, Hash Tables, and pandas DataFrames

M. Danish

Computer Science, CSU Global

CSC506, Design and Analysis of Algorithms

Dr. Lori Farr

02/12/2023

© M. Danish 2023

Abstract

In this paper, we explore the features and capabilities of three Python data structures as they relate to statistical analysis—and, by extension, their relationship to machine learning and artificial intelligence applications, which are often designed to work with statistical data. First, we will discuss the common problem solved by dictionaries, hash tables, and **pandas** DataFrames. Next, we will summarize the background of each data structure. Then, we will provide our analysis of the similarities and differences among the three data structures and a discussion of the significance of our analysis. We will conclude by revisiting significant points.

Keywords: **pandas,** DataFrames, data frames, Python dictionaries, Python data structures, Python hash tables, data structures for machine learning

© M. Danish 2023

Introduction

Hash tables, dictionaries, and data frames are all designed to solve the same problem—i.e., the storage of information as key-value pairs—in slightly different ways. An example use case for storing data as key-value pairs is that of storing labeled statistical data. Although arrays can also be used to store statistical data, they cannot be re-shaped the same way data frames can and are therefore not as useful for applications in which mixed-type data is being compared or when data needs to be re-labeled (or re-categorized) on the fly for different analytical purposes (McKinney, 2010). Dictionaries and hash tables each have advantages and unique qualities that make them suitable for storing statistical data. As shown in the following comparison and analysis, however, we find that data frames are superior to both hash tables and dictionaries for machine learning and artificial intelligence applications, especially those that process statistical data for the purposes of uncovering patterns and making predictions.

Background

What are Dictionaries?

Dictionaries are mutable, comma-separated collections of key-value pairs. Because dictionary keys must be immutable, they can be strings, numbers, or tuples. Lists—which can be altered—cannot be used as dictionary keys. Additionally, dictionary keys must be unique, as attempting to add another value with a key that already exists in the dictionary simply replaces the existing key-value pair (Python Software Foundation, 2023).

Values within dictionaries can be accessed by calling the key, and keys can be accessed by calling the value. We can loop over a dictionary to retrieve both a key and its value at once, which is useful when we need to print one or more dictionary elements in a readable format. Looping allows us to search dictionaries via `while` and `if` statements as well, as demonstrated in the following example.

Figure 1

Search for and Update a Dictionary Key via an If Statement

```
if key in dict:
   # First, confirm and follow user intent
   update_key = input(‘This key already exists. Do you want to
   update it?’)
   if ‘y’ or ‘Y’ in update_key:
      dict.update key #  If the key exists, then change the
value
   else:
      print(‘Okay, no changes have been made.’)
```

Danish 2023

```
elif key not in dict:
   dict['key'].append(some_value) # If the key doesn't exist,
then create it and assign the value
```

This gives us a way to check whether a given key already exists before we accidentally overwrite it with a new value. It also allows us to search a dictionary for a value and return its key—or a list of keys, if more than one exists with the same value. Finally, dictionaries can be sorted by either keys or values (Python Software Foundation, 2023).

What are Hash Tables?

A hash table is a collection that maps unordered—and therefore, unsortable—items to locations in an array or vector. Hash tables use keys to map items to their respective indices. The array elements in hash tables are called buckets. Each bucket in a hash table can store one or multiple values; the keys correspond to the bucket indices. Hash functions compute bucket indices via the following formula:

$$V \,\%\, nb = k$$

where *V* represents the hash value, *nb* represents the number of buckets in the hash table, and *k* represents the key.

Hash table buckets are described as either empty or occupied; occupied means that the bucket contains at least one value. Because the hash table search function optimizes itself by differentiating between buckets that have been empty since the start and buckets that are only empty because values have been removed—and only stopping the search once an empty-since-start bucket is encountered—the hash table insert function prefers to store values in empty buckets of either type (Danish, 2023; Larson, 1988). Therefore, when keys are not

© M. Danish, 2023

unique—e.g., let’s say our hash table has 10 buckets, so according to the equation above, the values 25 and 45 both have a remainder (and therefore, a key) of 5—collisions occur (Larson, 1988). In my analysis, I will describe some methods for resolving collisions and how they affect hash table performance.

What are DataFrames?

First, it is important to note that there is a distinction between “data frames” as a general term and the camel-case name DataFrames. Data frames as a data structure are available in both Python and R. DataFrames are a type of two-dimensional (2D) data structure that is available via the Python **pandas** library. They incorporate features of arrays, dictionaries, and hash tables to store labels and map them to values. Like dictionaries, DataFrames can be sorted by either their indices—which are analogous to dictionary keys—or their values. DataFrames work well for statistical data because most statistical datasets are two-dimensional (McKinney, 2010).

Manish 2023

Analysis

Table 1

Similarities and Differences Among Dictionaries, Hash Tables, and DataFrames

	Dictionaries	**Hash Tables**	**DataFrames**
Average Runtime Efficiency	$O(1)$	$O(1)$	$O(1)$
Dimensionality	1D+	1D	2D
Sortable	Yes	No	Yes
Support mixed-type data	Yes	Yes	Yes
Human	Yes—but	No—`print(hash`	Yes—`print(DataFr`

readable	`print(dict_name)` outputs all key-value pairs on the same line, so dictionaries must be looped over to print key-value pairs line-by-line	`table_name)` only outputs indices, so a print method or function should be written in for best results	`ame_name)` outputs a neat, readable format that visually approximates a spreadsheet
Flexible size	Yes	Yes, but must be rehashed when the size changes	Yes
Keys	■ Anything immutable ■ Case sensitive ■ Can be returned as a list	■ Numbers—can be computed from the ASCII values of characters in a string ■ Case insensitive ■ Can be returned as a list	■ Can be strings or numbers ■ Case sensitive ■ Can be returned as a list
Values	Any Python object	Any hashable value	Any Python object
Sorting	By keys or by values	*Not applicable*	By labels or by values
Searching	Returns `True` or `False`	Returns a bucket index	■ Can incorporate multiple search parameters ■ Returns and prints all lines that satisfy the search parameters

Note: Table 1 aggregates information from the *pandas* documentation (NumFOCUS, 2023), Python documentation (Python Software Foundation, 2023), the article *Overview of pandas data types* (Moffit, 2018), and the book *Fundamentals of Python: Data Structures* (Lambert, 2019)

M. Danish 2023

Dictionaries

Runtime Complexity

Dictionaries have an average runtime complexity of $O(1)$. This is comparable to hash tables and DataFrames, which each also have an average runtime complexity of $O(1)$ (Python Software Foundation, 2023).

Unique Features and Advantages

There are two main differences in the behavior of dictionaries as compared to hash tables and DataFrames. The first is that when we attempt to call an invalid dictionary key, an error is returned; this necessitates additional code for error handling. Secondly, to print elements in a dictionary, we must iterate through them using a `for` loop because `print(dict_name)` produces output that is not well formatted for readability by humans (Lambert, 2019; Python Software Foundation, 2023)—although in comparison to hash tables, it does produce output that is at least somewhat readable.

© M. Danish 2023

Hash Tables

Runtime Complexity and Performance

Hash tables are faster than many other data structures because item insertion and removal—as well as searching—have a runtime complexity of $O(1)$. Therefore, the number of comparisons stays constant even in large arrays. This is comparable to dictionaries and DataFrames, which each also have an average runtime complexity of $O(1)$ (Python Software Foundation, 2023). Hash tables are able to maintain a constant runtime complexity via the rehashing process even as their sizes increase, which makes them well-suited for use with large datasets (Larson, 1988; Python Software Foundation, 2023). We will look at the rehashing process in further detail in the following section.

Unique Features and Advantages

In the background section, we briefly touched on collisions, which occur when values in a hash table share the same key. Although there are several methods for resolving collisions, the two most common methods are open addressing and chaining. Open addressing works by looking for an empty bucket via one of two probing types: linear probing or quadratic probing. Previously, we also discussed the difference between empty-since-removal buckets and empty-since-start buckets; linear probing recognizes that difference, while quadratic probing does not. Both types of probing start at the key's mapped bucket—i.e., the one the value would have been stored in assuming there were no collisions. If the mapped bucket is occupied, linear probing continues until it finds an empty-since-start bucket; quadratic probing continues until it finds the next empty bucket of either type (Lambert, 2019; Larson, 1988). By contrast, chaining assigns lists of items to each bucket rather than the values themselves. When a mapped bucket is already occupied, chaining inserts new items at the head of the bucket's linked list (Lambert, 2019).

Rehashing offers another unique advantage that makes it good for large datasets—It works to mitigate the clustering effects of linear probing by re-mapping existing records in the hash table as the size of the hash table increases (Lambert, 2019; Larson, 1988). Because total rehashing can significantly decrease performance, however, the spiral storage method should be used to achieve maximum rehashing efficiency. Spiral storage adds one bucket to the hash table at a time, so only local rehashing is necessary at each step (Larson, 1988).

DataFrames

Runtime Complexity

The average runtime complexity of DataFrames is constant, or $O(1)$. This is comparable

© M. Danish 2023

to dictionaries and hash tables, which each also have an average runtime complexity of $O(1)$ (Python Software Foundation, 2023).

Unique Features and Advantages

By incorporating aspects of arrays, dictionaries, and hash tables to store (and map) labels and values, DataFrames can maintain a constant runtime for both lookup and membership determination operations. A second advantage of the DataFrame structure is that the widely used R language provides the `data.frame` class, so data frames translate well between R and Python (McKinney, 2010). Thirdly, DataFrames can accept input from many other types of data structures, including (a) series, which are another type of **pandas** data structure; (b) dictionaries that contain 1D arrays, lists, series, or other dictionaries; (c) 2D NumPy arrays; (d) structured or record NumPy arrays; and (e) other DataFrames. The **pandas** module contains constructors that allow us to build DataFrames easily from any of the aforementioned data structures (NumFOCUS, 2023).

Another unique feature of DataFrames is the ability to pass row and column labels (NumFOCUS, 2023). This makes DataFrames well suited to data that traditionally would be stored in a matrix structure for quick reference. Finally, DataFrames provide a method for handling missing data, which is a critical consideration in any statistical analysis because it helps ensure more honest results (McKinney, 2010).

Discussion

My analysis provided me with a better understanding of **pandas** DataFrames and their applications to statistical analysis work. More specifically, it helped me see how DataFrames can be applied not only to machine learning model development, but also model type selection—a consideration that did not appear to be addressed in the existing literature I was able to access.

© M. Danish 2023

DataFrames are incredibly flexible data structures that can be pivoted and otherwise reshaped as needed due to their unique matrix layout of columns and index rows—This feature could potentially save a lot of time we might otherwise spend converting between other data structures as we think about the type of model to use on a given dataset, make decisions, and change course as our understanding of a particular dataset improves. Coupled with the fact that their average runtime complexity rivals that of not just dictionaries, but also hash tables, that flexibility appears to make them perfectly suited for use in machine learning model development. I have used DataFrames in one machine learning development project so far, and I intend to continue using them in place of other data structures because they are so easy to work with—and they eliminate the extra functions I would otherwise write to allow me to (a) print data at various testing and debugging points and (b) work around missing data.

Conclusions

Dictionaries, hash tables, and data frames all exist to solve the problem of storing data in key-value pairs. All three data structures can be resized as needed; however, hash tables that use the linear probing method require re-hashing as they are resized to avoid performance degradation that occurs due to clustering. Factoring in re-hashing processes for hash tables, the average runtime complexity of all three data structures is comparable—each has an average runtime of $O(1)$.

Dictionaries and DataFrames can both be sorted—hash tables, however, are unordered, so they cannot be sorted. Another important feature shared by dictionaries and DataFrames is that they can both accommodate 2D data, where hash tables cannot due to the nature of their indexing system. DataFrames in particular are designed to handle 2D data, which they store in a matrix-like structure of columns and index rows. DataFrames have the most readable and

© M. Darish 2023

user-friendly output when printed, and printing the contents of DataFrames requires no additional coding. DataFrames also have the best search and reshaping functionalities, which have the potential to save significant amounts of time that might otherwise be spent (a) coding functions to print search results and (b) converting between data structures. Finally, data frames also exist in the R programming language, which only makes **pandas** DataFrames that much better a place to start if we need to work in both Python and R for any reason. These features—in addition to their native ability to handle missing data that can ruin an analysis and necessitate both extra work *and* extra code—make DataFrames the most efficient data structure currently available in Python for the purposes of developing machine learning and artificial intelligence models.

© M. Danish 2023

References

Danish, M. (2023). *Linear probing*. [Unpublished paper]. Computer Science Department, Colorado State University (Global Campus).

Lambert, K. (2019). *Fundamentals of Python: data structures*. Cengage Learning. https://play.google.com/store/books/details?id=Bse8uwEACAAJ

Larson, P.-A. (1988). Dynamic hash tables. *Communications of the ACM*, *31*(4), 446–457. https://doi.org/10.1145/42404.42410

McKinney, W. (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference*. Python in Science Conference, Austin, Texas. https://doi.org/10.25080/majora-92bf1922-00a

Moffitt, C. (2018, March 26). *Overview of pandas data types*. Practical Business Python. https://pbpython.com/pandas_dtypes.html

NumFOCUS, Inc. (2023). *User guide — pandas 1.5.3 documentation*. https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html

Python Software Foundation. *5. Data structures*. (2023, February 4). Python Documentation. https://docs.python.org/3/tutorial/datastructures.html

© M. Danish 2023