tuts+

PYTHON

Advanced Python Data Structures

by Adriana Vasiu *18 Jun 2013* *26 Comments*

The aim of this tutorial is to show off Python's data structures and the best way to use them. Depending on what you need from a data structure, whether it's fast lookup, immutability, indexing, etc, you can choose the best data structure for the job and most of the time, you will be combining data structures together to obtain a logical and easy to understand data model.

Introduction

Python data structures are very intuitive from a syntax point of view and they offer a large choice of operations. This tutorial tries to put together the most common and useful information about each data structure and offer a guide on when it is best to use one structure or another. You can choose different kinds of data structures depending on what the data involves, if it needs to be modified, or if it's fixed data, and even what access type you would like, such as at the beginning/end/random etc.

Lists

A List represents the most versatile type of data structure in Python. It can contain items of different types and it has no rule against unicity. List indices start from zero, the elements can be sliced, concatenated, and so on. Lists also have a lot of similarities with strings, supporting the same kind of operations but unlike strings, lists are *mutable*.

How to Construct a List

A list can be built using the keyword `list` or using square brackets: `[]`, both of which accept comma separated values. Here's an example:

```
>>> l = ['a', 'b', 123]
>>> l
['a', 'b', 123]
```

How to Retrieve an Element From a List

```
>>> l[0]
'a'
>>> l[10]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

As seen above, in order to access the data within your list, you must know what index position the element is at, otherwise you get an "index out of range" error.

How to Slice a List

All slicing operations return a shallow copy of the list. The slicing indexes are optional and they work in the same way as slicing indexes for strings.

Here's an example of how to slice a list:

```
>>> l = ['a', 'b', 123]
>>> l[:]
['a', 'b', 123]
>>> new_l = l[1:]
>>> new_l
['b', 123]
>>> l
['a', 'b', 123]
```

Let's take a look at some other common list operations.

Inserting and Removing Elements

```
>>> l = ['a', 'b', 123]
>>> l.append(234) #inserts an element at the end of the list
>>> l
['a', 'b', 123, 234]
>>> l.insert(2, 'c') #inserts an element into the third position
>>> l
['a', 'b', 'c', 123, 234]
>>> l.insert(-1, 111) #inserts an element into the second from la
>>> l
['a', 'b', 'c', 123, 111, 234]
>>> l.remove(111) #removes an element based on value
>>> l
['a', 'b', 'c', 123, 234]
>>> l.remove('does not exist in the list')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Retrieving and Looking Up Elements

Lists can also be used as stacks or queues because of how easy it is to add and remove elements from the beginning or end of the list.

```
>>> last_element = l.pop() #returns the last element, modifying t
>>> last_element
234
>>> l
['a', 'b', 'c', 123]
>>> third_element = l.pop(2) #returns the third element, modifyin
>>> third_element
'c'
>>> l
['a', 'b', 123]
>>> l.index('a')
0
>>> l.index('does not exist in the list')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: 'does not exist in the list' is not in list
>>> l.count('a') #returns the number of occurrences of an element
1
```

Whole List Operations

```
>>> l.extend ([1, 2]) # concatenates a list on to the existing lis
>>> l
['a', 'b', 123, 1, 2]
>>> l.sort()
>>> l
[1, 2, 123, 'a', 'b']
>>> l.reverse()
>>> l
['b', 'a', 123, 2, 1]
```

As you can see, it's very easy to extend, sort, and reverse lists using the above methods.

List Comprehensions

A list comprehension means, constructing a list in a way that is very natural from a mathematical point of view. The code for doing this is brief and very easy to read. A simple example of when you would use list comprehensions is when you want to construct a new list based on the elements from another list. Let's have a look at how you can multiply all of the elements of a numeric list by two, in a simple one line construct:

```
>>> l = [1, 2, 3]
>>> new_l = [x*2 for x in l]
>>> new_l
[2, 4, 6]
```

In the example above, the comprehension is represented by the multiplication expression that will be applied to every `x` element in the original, `l` list.

When to Use Lists

As shown in the examples above, lists are best used in the following situations:

- When you need a mixed collection of data all in one place.
- When the data needs to be ordered.
- When your data requires the ability to be changed or extended. Remember, lists are mutable.
- When you don't require data to be indexed by a custom value. Lists are numerically indexed and to retrieve an element, you must know its numeric position in the list.
- When you need a stack or a queue. Lists can be easily manipulated by appending/removing elements from the beginning/end of the list.
- When your data doesn't have to be unique. For that, you would use sets.

Sets

A set is an unordered collection with no duplicate values. A set can be created by using the keyword `set` or by using curly braces `{}`. However, to create an empty set you can only use the `set` construct, curly braces alone will create an empty dictionary.

> *Use the `set` keyword to create an empty set, curly brackets `{}` will create an empty dictionary.*

How to Construct a Set

The `set` construct accepts one argument, a list.

```
>>> l = [1, 2, 3]
>>> s = set(l)
>>> s
set([1, 2, 3])
```

The `{}` construct is straight forward as well:

```
>>> s = {1, 2, 3}
>>> s
```

Sets are used to eliminate duplicate values from within a list:

```
>>> l = [1, 2, 3, 3]
>>> s = set(l)
>>> s
set([1, 2, 3])
```

The way a `set` detects if a clash between non-unique elements has occurred is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be hashable.

Here is an example:

```
>>> set ([1, [1,2]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

> Note that a hashable object doesn't necessarily mean that an object has the `__hash__` method available to it. It is important that the hash of the object doesn't change during its lifetime, which is obviously not the case with lists, sets, or dictionaries (dictionaries will be discussed later in this tutorial).

Set Operations

The `set` structure also supports mathematical operations like:

- Union
- Intersection
- Difference
- Symmetric Difference

Here are a few examples:

```
>>> set1 = set([1, 2, 3])
>>> set2 = set([3, 4, 5])
>>> set1 | set2 #union
set([1, 2, 3, 4, 5])
>>> set1 & set2 #intersection
```

```
set([3])
>>> set1 -  set2 #difference
set([1, 2])
>>> set1 ^ set2 #symmetric difference (elements that are in the f
set([1, 2, 4, 5])
>>>
```

Set Comprehensions

Just like lists, sets also support comprehensions. Here is an example of how to use set comprehensions to find the unique consonants within a word:

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> {x for x in 'maintenance' if x not in vowels }
set(['c', 'm', 't', 'n'])
```

Frozensets

A frozenset is basically just like a regular set, except that is immutable. It is created using the keyword `frozenset`, like this:

```
>>> frozen = frozenset([1, 2, 3])
```

When to Use Sets

You should choose to use a `set` in the following situations:

- When you need a unique set of data: Sets check the unicity of elements based on hashes.
- When your data constantly changes: Sets, just like lists, are mutable.
- When you need a collection that can be manipulated mathematically: With sets it's easy to do operations like difference, union, intersection, etc.
- When you don't need to store nested lists, sets, or dictionaries in a data structure: Sets don't support unhashable types.

Tuples

Tuples are immutable, but can hold mutable objects.

A tuple is represented by a number of values separated by commas. Unlike lists, tuples are immutable and the output is surrounded by parentheses so that nested tuples are processed correctly. Additionally, even though tuples are immutable, they can hold mutable data if needed.

How to Construct a Tuple

Constructing an empty tuple requires parentheses. Here's an example:

```
>>> my_empty_tuple = ()
>>> my_empty_tuple
()
```

Constructing a tuple with one element requires a trailing comma. Example:

```
>>> one_elem_tuple = 'a',
>>> one_elem_tuple
('a',)
>>> one_elem_tuple = ('a',)
>>> one_elem_tuple
('a',)
```

If the trailing comma is not there when creating a single element tuple, regardless if you use parentheses or not, Python will just interpret the value as a literal and will not create the tuple.

Constructing a tuple with multiple elements requires a list of values separated by commas. Here's an example:

```
>>> s = 'a', 'b', [1, 2, 3]
>>> s
('a', 'b', [1, 2, 3])
>>>
```

Tuples, from a performance point of view are great because of their immutability.

Python will know exactly how much memory to allocate for the data to be stored.

When to Use Tuples

- When you need to store data that doesn't have to change.
- When the performance of the application is very important. In this situation you can use tuples whenever you have fixed data collections.
- When you want to store your data in logical immutable pairs, triples etc.

Dictionaries

Dictionaries are represented by a `key:value` pair. In other words, they are maps or associative collections. The keys, unlike lists where they are numeric, can be of any immutable type and must be unique. The values can be of any type, mutable or immutable.

How to Construct a Dictionary

There are several ways to construct a dictionary. The most efficient way in terms of performance is to use curly braces `{}`:

```
>>> vowels = {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5:'u'}
>>> vowels
{1: 'a', 2: 'e', 3: 'i', 4: 'o', 5: 'u'}
```

Another way to create a dictionary is by using comprehensions. By using an input collection you can create the `key:value` pairs in one simple construct.

Here's an example:

```
>>> {x:x*x for x in (1, 2, 3)}
{1: 1, 2: 4, 3: 9}
```

You can also use the keyword `dict` and get the same result. The `dict` construct takes as an argument, a list of `key:value` pairs. Here's one in action:

```
>>> dict([(1,'a'), (2,'e'), (3,'i'), (4,'o'), (5,'u')])
{1: 'a', 2: 'e', 3: 'i', 4: 'o', 5: 'u'}
```

If the keys are strings, you can use the following keyword expression:

```
>>> dict(a=1, e=2, i=3, o=4, u=5)
{'i': 3, 'u': 5, 'e': 2, 'a': 1, 'o': 4}
```

However, it does not work the other way around when the keywords are numeric:

```
>>> dict(1=a, 2=e, 3=i, 4=o, 5=u)
  File "<stdin>", line 1
SyntaxError: keyword can't be an expression
```

Dictionary Operations

Accessing data in a dictionary is very straight forward, just wrap its key name within square brackets:

```
>>> vowels = {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5:'u'}
>>> vowels[1]
'a'
>>> vowels[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 10
>>>
```

As seen above, a `KeyError` occurs if the key doesn't exist in the dictionary.

Also, a `key:value` pair can be deleted using the `del` keyword, like so:

```
>>> vowels = {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5:'u'}
>>> del(vowels[1])
>>> vowels
{2: 'e', 3: 'i', 4: 'o', 5: 'u'}
>>> del(vowels[10])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 10
```

```
>>>
```

Built-in Methods

Dictionaries support many built-in methods, but some of the most useful ones are: `keys()`, `values()`, `iteritems()`, `itervalues()`, and `has_key()`.

Here's an example:

```
>>> vowels = {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5:'u'}
>>> vowels.keys()
[1, 2, 3, 4, 5]
>>> vowels.values()
['a', 'e', 'i', 'o', 'u']
```

The methods `iteritems()` and `itervalues()` return iterators, so they can be used in `for` loops. Here's an example:

```
>>> for k, v in vowels.iteritems():
...     print k, v
...
1 a
2 e
3 i
4 o
5 u
>>> for v in vowels.itervalues():
...     print v
...
a
e
i
o
u
>>>
```

When to Use a Dictionary

- When you need a logical association between a `key:value` pair.
- When you need fast lookup for your data, based on a custom key.
- When your data is being constantly modified. Remember, dictionaries are mutable.

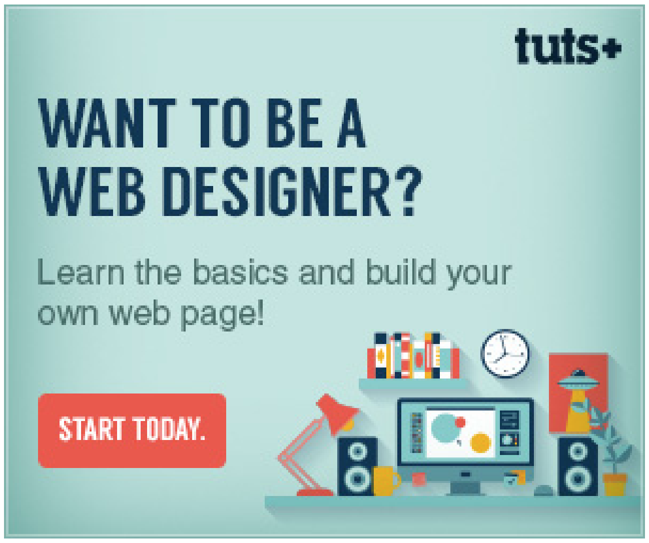
Conclusion

General Python Data Structure Usage

- Use lists if you have a collection of data that does not need random access. For random access, you need to have knowledge of the element's numeric index. Try to choose lists when you need a simple, iterable collection that is modified frequently. Lists are very useful in comprehension expressions for constructing sets or dictionaries.
- Use a set if you need unicity for the elements and you don't need a nested dictionary or list. Also remember that a set cannot hold any unhashable data types.
- Use tuples when your data cannot change. Many times, a tuple is used in combination with a dictionary, for example, a tuple might represent a key, because it's immutable.
- Use frozensets if you need both unique data and immutability.

Other Tips

- Every data structure has a multitude of built in methods and capabilities. If you go through them carefully you'll learn how the data is meant to be used. For example, if you call `dir` and pass in a set object, you will see that it holds a lot of methods that can perform mathematical operations.
- Most of the time, your data needs to adapt to the operations that you want to perform. So if you know your object will be hashed, create an immutable structure.
- Most data structures have multiple ways of constructing or accessing its data. If you are concerned about the performance of your application always read the documentation or read the source code of the construct implementation to find out more about its performance. For example, creating a dictionary by using `dict` offers less performance than using the curly braces (`{}`) syntax.
- While debugging, many developers find it useful to learn how to identify and connect the error that they are receiving to a specific data structure. For example, you'll find that lists throw `IndexError` and `ValueError` and dictionaries throw `KeyError`. This technique should help you with detecting which of your data structures is causing the error.


tuts+
WANT TO BE A WEB DESIGNER?
Learn the basics and build your own web page!
START TODAY.

Advertisement

Categories:

Python **Web Development**

Translations:

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

About Adriana Vasiu

N/A

Advertisement

Suggested Envato Tuts+ Course

Build a News Aggregator With Django $9

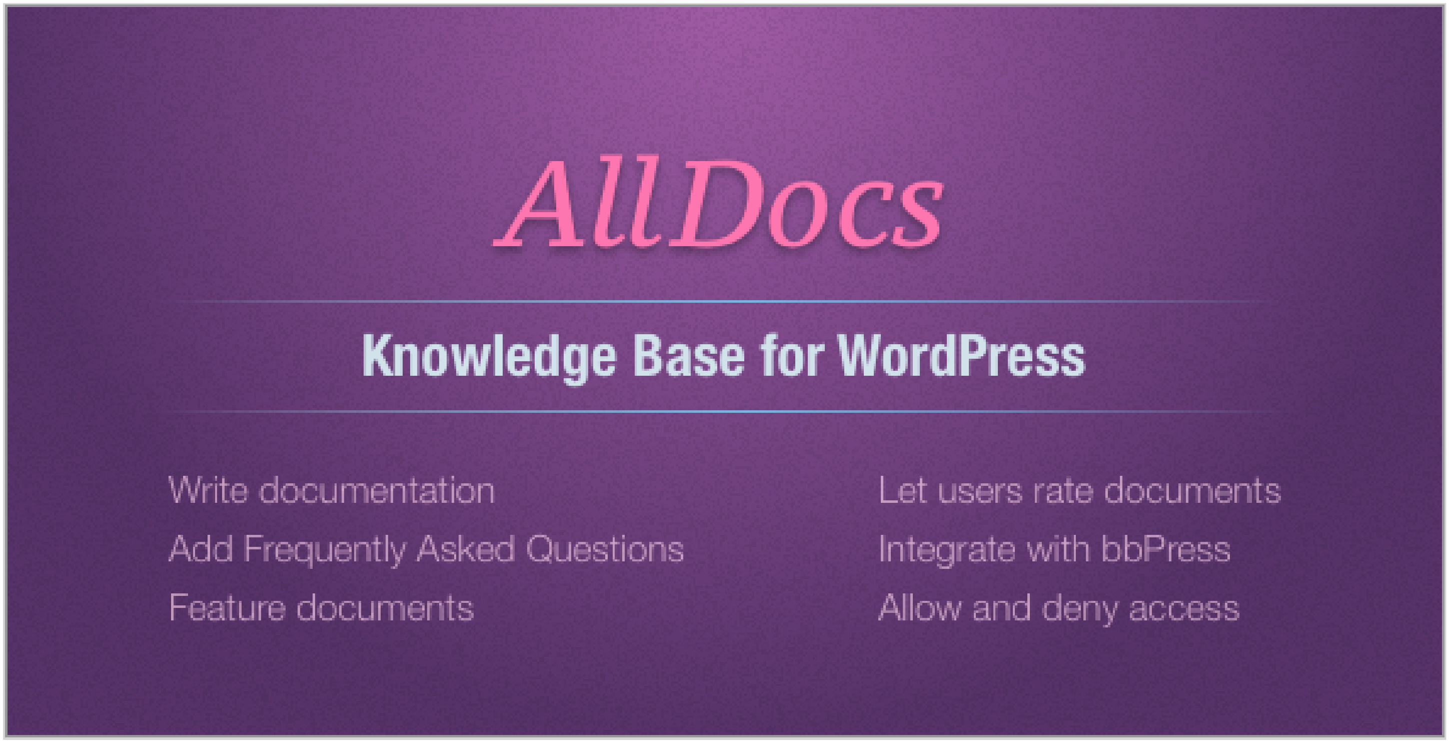
Related Tutorials

Python 3 Type Hints and Static Analysis
Code

A Smooth Refresher on Python's Tuples
Code

Working With JSON in Swift
Code

Envato Market Item



What Would You Like to Learn?

Suggest an idea to the content editorial team at Envato Tuts+.

26 Comments Nettuts+ 1 Login

Recommend 2 Share Sort by Best

Join the discussion…

Smoki · 3 years ago

The word "Advanced" does not belong here.

110 · Reply · Share ›

Jacob Clarck Smoki · 3 years ago

most of nettuts fellows are PHPes or Rubiests, so this kinda of Awesomeness in Data Structure are really advanced to them. IMO.

5 · Reply · Share ›

thedude321 Jacob Clarck · 3 years ago

I agree. This is in no way advanced, heck you would learn this stuff in by your third lesson in Python (max). And if you wanted some real powerful data-structures, Python has much more powerful constructs. Also, the uses of those data structures are far more flexible than that of other data structures I've seen in Ruby and PHP is out of the question.

6 · Reply · Share ›

Adriana Vasiu Smoki · 3 years ago

"Advanced" was used from a language construct point of view. I didn't cover basic data structures as strings, booleans etc.
Because in python all data types are objects many developers talk of strings, dictionaries etc as being data structures altogether. The title was meant to make this difference.

5 · Reply · Share ›

GaaraCoder · 3 years ago

How is that advanced.

17 · Reply · Share ›

Jack Scotty · 3 years ago

It should be titled, "Intermediate Python Data Structures" to match the Difficulty under "Tutorial Details".

4 · Reply · Share ›

the new messi fan Jack Scotty · 3 years ago

i think it is basic data structure

11 · Reply · Share ›

Aneesh Dogra · 3 years ago

Quoting from python documentation: "It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one)."

You should instead use the Queue [1] or Dequeue [2] (2 sided queue)

[1]: http://docs.python.org/2/libra...
[2]: http://docs.python.org/2/libra...

1 · Reply · Share ›

José Daniel Gómez Rodríguez · 3 years ago

Is there a way to create Fixed size containers on python?

For example, as we can do in C

```
int Matrix[<rows>][<cols>];
```

Or in PHP

```
$Matrix = new SplFixedArray( <Rows> );
foreach($Matrix as &$row){
    $row = new SplFixedArray( <Cols> );
}
```

1 · Reply · Share ›

Vanderson → José Daniel Gómez Rodríguez · 3 years ago

no.

Reply · Share ›

lcfseth → Vanderson · 3 years ago

Actually Yes. The simplest approach is to use [[None] * rows] * cols or simply use the numpy library which provides a powerful N-dimensional array object

6 · Reply · Share ›

José Daniel Gómez Rodríguez → lcfseth · 3 years ago

The usage of [[None] * rows] * cols in deed creates a multi-dimensional matrix, BUT i found a bug.

```
rows = 3
cols = 3
x = [[None] * rows] * cols
x[0][1] = 1
for i in x:
    print i
```

Outputs:

```
[None, 1, None]
[None, 1, None]
[None, 1, None]
```

Instead of:

```
[None, 1, None]
```

see more

• Reply • Share ›

thedude321 José Daniel Gómez Rodríguez • 3 years ago

This kind of use is actually almost never seen. But yes, in fact you can create a fixed array size using Python's librarries.

First you would need to:

```
import array
```

Then depending on what you wanted to create, say an integer array:

```
import array

limited_size_array = array.array('i', range(10))
```

^^ The 'i' is for integer. You can specify the type.

You can create 2D arrays from this too, Hope that helps

2 • Reply • Share ›

José Daniel Gómez Rodríguez thedude321 • 3 years ago

What about dynamic-type fixed array? can 'array.array()' construct it too?

• Reply • Share ›

José Daniel Gómez Rodríguez lcfseth • 3 years ago

Wanderful, Thank you!

• Reply • Share ›

rowend • 3 years ago

Good info, Thak you

1 • Reply • Share ›

Cherise Bry • a month ago

lol "advanced"

• Reply • Share ›

Prince Nwosu • 2 months ago

Create a function manipulate_data that does the following

Accepts as the first parameter a string specifying the data structure to be used "list", "set" or "dictionary"

Accepts as the second parameter the data to be manipulated based on the data structure specified e.g [1, 4, 9, 16, 25] for a list data structure

Based off the first parameter

return the reverse of a list or
add items `"ANDELA"`, `"TIA"` and `"AFRICA"` to the set and return the resulting set
return the keys of a dictionary.

how do i use the dictionary in this question

• Reply • Share ›

Martin O Cornel → Prince Nwosu • 2 months ago

to return the keys of a dictionary use
dictionary_name.keys()

example.
My_Dictionary = {1:"Martin", 2:"Cornel", 3:"Programmer"}

use method
My_Dictionary.keys()
[1,2,3]

• Reply • Share ›

Prince Nwosu → Martin O Cornel • 2 months ago

thanks, God bless u

• Reply • Share ›

Martin O Cornel → Prince Nwosu • 2 months ago

Welcome

I noticed you were doing the andela homestudy

I started mine yesterday and am on test quiz 5.
You don't mind sharing your lab solutions for labs Algo, Recursive and fizz buzz to martin@cornel.co.ke right?

Reply · Share ›

Mohit Raj · 9 months ago
http://l4wisdom.com/python/ for python learning

Reply · Share ›

anonymous · a year ago
Very good information, Thnks.
Really importat is Computational Thinking.

Reply · Share ›

Manjunath hugar · a year ago
Very well explained, thanks.

Reply · Share ›

ajkumar25 · 2 years ago
I have found a article which is about advanced data structures.
http://pypix.com/python/advanc...

Reply · Share ›

yepi · 3 years ago
This is indeed a great site! Very informative and interesting to read.

The details are well-explained and very concise.

This is what I've been looking for. Thank you!

Reply · Share ›

Subscribe   Add Disqus to your site Add Disqus Add   Privacy

Advertisement