

Common Python Data Structures (Guide)

by Dan Bader · Aug 26, 2020 · 7 Comments

basics data-structures python

Mark as Completed



Tweet

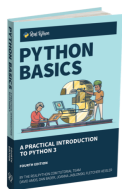
Share

Email

Table of Contents

- [Dictionaries, Maps, and Hash Tables](#)
 - [dict: Your Go-to Dictionary](#)
 - [collections.OrderedDict: Remember the Insertion Order of Keys](#)
 - [collections.defaultdict: Return Default Values for Missing Keys](#)
 - [collections.ChainMap: Search Multiple Dictionaries as a Single Mapping](#)
 - [types.MappingProxyType: A Wrapper for Making Read-Only Dictionaries](#)
 - [Dictionaries in Python: Summary](#)
- [Array Data Structures](#)
 - [list: Mutable Dynamic Arrays](#)
 - [tuple: Immutable Containers](#)
 - [array.array: Basic Typed Arrays](#)
 - [str: Immutable Arrays of Unicode Characters](#)
 - [bytes: Immutable Arrays of Single Bytes](#)
 - [bytearray: Mutable Arrays of Single Bytes](#)
 - [Arrays in Python: Summary](#)
- [Records, Structs, and Data Transfer Objects](#)
 - [dict: Simple Data Objects](#)
 - [tuple: Immutable Groups of Objects](#)
 - [Write a Custom Class: More Work, More Control](#)
 - [dataclasses.dataclass: Python 3.7+ Data Classes](#)
 - [collections.namedtuple: Convenient Data Objects](#)
 - [typing.NamedTuple: Improved Namedtuples](#)
 - [struct.Struct: Serialized C Structs](#)
 - [types.SimpleNamespace: Fancy Attribute Access](#)
 - [Records, Structs, and Data Objects in Python: Summary](#)
- [Sets and Multisets](#)
 - [set: Your Go-to Set](#)
 - [frozenset: Immutable Sets](#)

- [collections.Counter: Multisets](#)
- [Sets and Multisets in Python: Summary](#)
- [Stacks \(LIFOs\)](#)
 - [list: Simple, Built-in Stacks](#)
 - [collections.deque: Fast and Robust Stacks](#)
 - [queue.LifoQueue: Locking Semantics for Parallel Computing](#)
 - [Stack Implementations in Python: Summary](#)
- [Queues \(FIFOs\)](#)
 - [list: Terribly Slooooow Queues](#)
 - [collections.deque: Fast and Robust Queues](#)
 - [queue.Queue: Locking Semantics for Parallel Computing](#)
 - [multiprocessing.Queue: Shared Job Queues](#)
 - [Queues in Python: Summary](#)
- [Priority Queues](#)
 - [list: Manually Sorted Queues](#)
 - [heapq: List-Based Binary Heaps](#)
 - [queue.PriorityQueue: Beautiful Priority Queues](#)
 - [Priority Queues in Python: Summary](#)
- [Conclusion: Python Data Structures](#)



[Your Practical Introduction to Python 3 »](#)

[Remove ads](#)

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Stacks and Queues: Selecting the Ideal Data Structure](#)

Data structures are the fundamental constructs around which you build your programs. Each data structure provides a particular way of organizing data so it can be accessed efficiently, depending on your use case. Python ships with an extensive set of data structures in its [standard library](#).

However, Python’s naming convention doesn’t provide the same level of clarity that you’ll find in other languages. In [Java](#), a list isn’t just a `list`—it’s either a `LinkedList` or an `ArrayList`. Not so in Python. Even experienced Python developers sometimes wonder whether the built-in `list` type is implemented as a [linked list](#) or a dynamic array.

In this tutorial, you’ll learn:

- Which common **abstract data types** are built into the Python standard library
- How the most common abstract data types map to Python’s **naming scheme**
- How to put abstract data types to **practical use** in various algorithms

Note: This tutorial is adapted from the chapter “Common Data Structures in Python” in [Python Tricks: The Book](#). If you enjoy what you read below, then be sure to check out [the rest of the book](#).

Free Download: [Get a sample chapter from Python Tricks: The Book](#) that shows you Python’s best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Dictionaries, Maps, and Hash Tables

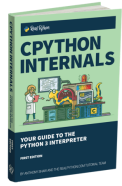
In Python, [dictionaries](#) (or **dicts** for short) are a central data structure. Dicts store an arbitrary number of objects, each identified by a unique dictionary **key**.

Dictionaries are also often called **maps**, **hashmaps**, **lookup tables**, or **associative arrays**. They allow for the efficient lookup, insertion, and deletion of any object associated with a given key.


Phone books make a decent real-world analog for dictionary objects. They allow you to quickly retrieve the information (phone number) associated with a given key (a person's name). Instead of having to read a phone book front to back to find someone's number, you can jump more or less directly to a name and look up the associated information.

This analogy breaks down somewhat when it comes to *how* the information is organized to allow for fast lookups. But the fundamental performance characteristics hold. Dictionaries allow you to quickly find the information associated with a given key.

Dictionaries are one of the most important and frequently used data structures in computer science. So, how does Python handle dictionaries? Let's take a tour of the dictionary implementations available in core Python and the Python standard library.



Your **Guided Tour** Through the **Python 3.9 Interpreter** »

 [Remove ads](#)

dict: Your Go-to Dictionary

Because dictionaries are so important, Python features a robust dictionary implementation that's built directly into the core language: the [dict](#) data type.

Python also provides some useful **syntactic sugar** for working with dictionaries in your programs. For example, the curly-brace (`{ }`) dictionary expression syntax and [dictionary comprehensions](#) allow you to conveniently define new dictionary objects:

Python

>>>

```
>>> phonebook = {
...     "bob": 7387,
...     "alice": 3719,
...     "jack": 7052,
... }

>>> squares = {x: x * x for x in range(6)}

>>> phonebook["alice"]
3719

>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

There are some restrictions on which objects can be used as valid keys.

Python's dictionaries are indexed by keys that can be of any [hashable](#) type. A **hashable** object has a hash value that never changes during its lifetime (see `__hash__`), and it can be compared to other objects (see `__eq__`). Hashable objects that compare as equal must have the same hash value.

[Immutable](#) types like [strings](#) and [numbers](#) are hashable and work well as dictionary keys. You can also use [tuple](#) objects as dictionary keys as long as they contain only hashable types themselves.

For most use cases, Python's built-in dictionary implementation will do everything you need. Dictionaries are highly optimized and underlie many parts of the language. For example, [class attributes](#) and variables in a [stack frame](#) are both stored internally in dictionaries.

Python dictionaries are based on a well-tested and finely tuned [hash table](#) implementation that provides the performance characteristics you'd expect: $O(1)$ time complexity for lookup, insert, update, and delete operations in the average case.

There's little reason not to use the standard `dict` implementation included with Python. However, specialized third-party dictionary implementations exist, such as [skip lists](#) or [B-tree-based](#) dictionaries.

Besides plain dict objects, Python's standard library also includes a number of specialized dictionary implementations. These specialized dictionaries are all based on the built-in dictionary class (and share its performance characteristics) but also include some additional convenience features.

Let's take a look at them.

collections.OrderedDict: Remember the Insertion Order of Keys

Python includes a specialized dict subclass that remembers the insertion order of keys added to it: [collections.OrderedDict](#).

Note: OrderedDict is not a built-in part of the core language and must be imported from the collections module in the standard library.

While standard dict instances preserve the insertion order of keys in CPython 3.6 and above, this was simply a [side effect](#) of the CPython implementation and was not defined in the language spec until Python 3.7. So, if key order is important for your algorithm to work, then it's best to communicate this clearly by explicitly using the OrderedDict class:

```
Python                                                                 >>>
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)

>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d["four"] = 4
>>> d
OrderedDict([('one', 1), ('two', 2),
              ('three', 3), ('four', 4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

Until [Python 3.8](#), you couldn't iterate over dictionary items in reverse order using `reversed()`. Only OrderedDict instances offered that functionality. Even in Python 3.8, dict and OrderedDict objects aren't exactly the same. OrderedDict instances have a [.move_to_end\(\) method](#) that is unavailable on plain dict instance, as well as a more customizable [.popitem\(\) method](#) than the one plain dict instances.

collections.defaultdict: Return Default Values for Missing Keys

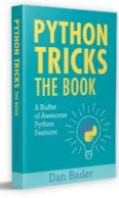
The [defaultdict](#) class is another dictionary subclass that accepts a callable in its constructor whose return value will be used if a requested key cannot be found.

This can save you some typing and make your intentions clearer as compared to using `get()` or catching a [KeyError exception](#) in regular dictionaries:

```
Python                                                                 >>>
>>> from collections import defaultdict
>>> dd = defaultdict(list)

>>> # Accessing a missing key creates it and
>>> # initializes it using the default factory,
>>> # i.e. list() in this example:
>>> dd["dogs"].append("Rufus")
>>> dd["dogs"].append("Kathrin")
>>> dd["dogs"].append("Mr Sniffles")

>>> dd["dogs"]
['Rufus', 'Kathrin', 'Mr Sniffles']
```



“I don’t even feel like I’ve scratched the surface of what I can do with Python”

Write More Pythonic Code »

[Remove ads](#)

`collections.ChainMap`: Search Multiple Dictionaries as a Single Mapping

The [`collections.ChainMap`](#) data structure groups multiple dictionaries into a single mapping. Lookups search the underlying mappings one by one until a key is found. Insertions, updates, and deletions only affect the first mapping added to the chain:

Python

>>>

```
>>> from collections import ChainMap
>>> dict1 = {"one": 1, "two": 2}
>>> dict2 = {"three": 3, "four": 4}
>>> chain = ChainMap(dict1, dict2)

>>> chain
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})

>>> # ChainMap searches each collection in the chain
>>> # from left to right until it finds the key (or fails):
>>> chain["three"]
3
>>> chain["one"]
1
>>> chain["missing"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'missing'
```

`types.MappingProxyType`: A Wrapper for Making Read-Only Dictionaries

[`MappingProxyType`](#) is a wrapper around a standard dictionary that provides a read-only view into the wrapped dictionary’s data. This class was added in Python 3.3 and can be used to create immutable proxy versions of dictionaries.

`MappingProxyType` can be helpful if, for example, you’d like to return a dictionary carrying internal state from a class or module while discouraging write access to this object. Using `MappingProxyType` allows you to put these restrictions in place without first having to create a full copy of the dictionary:

Python

>>>

```
>>> from types import MappingProxyType
>>> writable = {"one": 1, "two": 2}
>>> read_only = MappingProxyType(writable)

>>> # The proxy is read-only:
>>> read_only["one"]
1
>>> read_only["one"] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment

>>> # Updates to the original are reflected in the proxy:
>>> writable["one"] = 42
>>> read_only
mappingproxy({'one': 42, 'two': 2})
```


Dictionaries in Python: Summary

All the Python dictionary implementations listed in this tutorial are valid implementations that are built into the Python standard library.

If you're looking for a general recommendation on which mapping type to use in your programs, I'd point you to the built-in `dict` data type. It's a versatile and optimized hash table implementation that's built directly into the core language.

I would recommend that you use one of the other data types listed here only if you have special requirements that go beyond what's provided by `dict`.

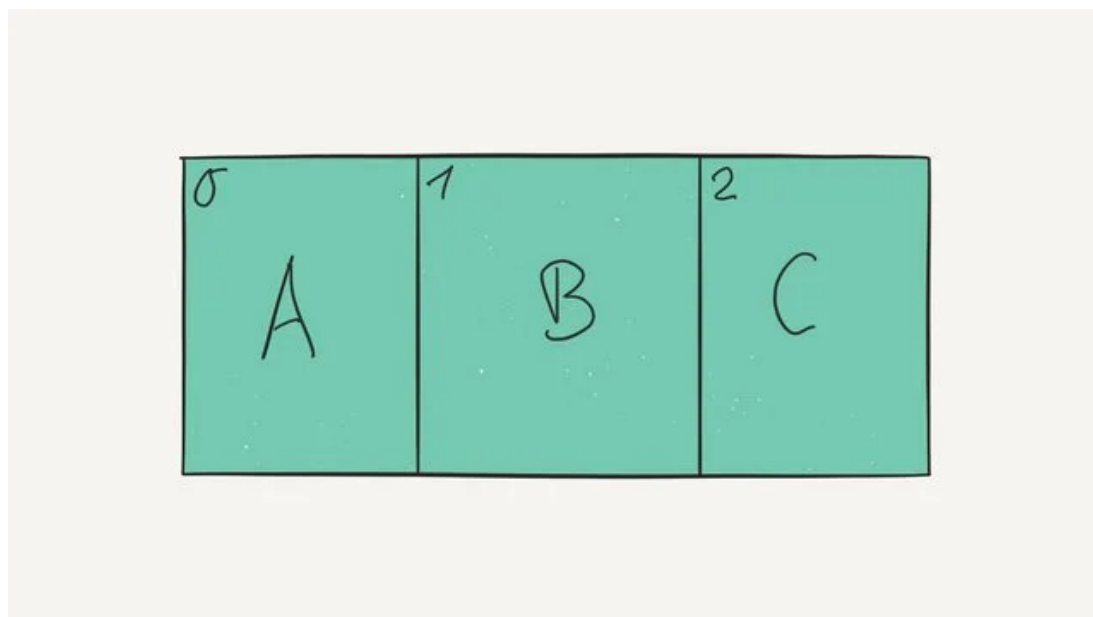
All the implementations are valid options, but your code will be clearer and easier to maintain if it relies on standard Python dictionaries most of the time.

Array Data Structures

An **array** is a fundamental data structure available in most programming languages, and it has a wide range of uses across different algorithms.

In this section, you'll take a look at array implementations in Python that use only core language features or functionality that's included in the Python standard library. You'll see the strengths and weaknesses of each approach so you can decide which implementation is right for your use case.

But before we jump in, let's cover some of the basics first. How do arrays work, and what are they used for? Arrays consist of fixed-size data records that allow each element to be efficiently located based on its index:



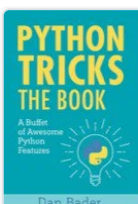
Because arrays store information in adjoining blocks of memory, they're considered **contiguous** data structures (as opposed to **linked** data structures like linked lists, for example).

A real-world analogy for an array data structure is a parking lot. You can look at the parking lot as a whole and treat it as a single object, but inside the lot there are parking spots indexed by a unique number. Parking spots are containers for vehicles—each parking spot can either be empty or have a car, a motorbike, or some other vehicle parked on it.

But not all parking lots are the same. Some parking lots may be restricted to only one type of vehicle. For example, a motor home parking lot wouldn't allow bikes to be parked on it. A restricted parking lot corresponds to a **typed** array data structure that allows only elements that have the same data type stored in them.

Performance-wise, it's very fast to look up an element contained in an array given the element's index. A proper array implementation guarantees a constant $O(1)$ access time for this case.

Python includes several array-like data structures in its standard library that each have slightly different characteristics. Let's take a look.



“I wished I had access to a book like this when I started learning Python many years ago”
— Mariatta Wijaya, CPython Core Developer

[Learn More »](#)

List: Mutable Dynamic Arrays

[Lists](#) are a part of the core Python language. Despite their name, Python’s lists are implemented as **dynamic arrays** behind the scenes.

This means a list allows elements to be added or removed, and the list will automatically adjust the backing store that holds these elements by allocating or releasing memory.

Python lists can hold arbitrary elements—everything is an object in Python, including functions. Therefore, you can mix and match different kinds of data types and store them all in a single list.

This can be a powerful feature, but the downside is that supporting multiple data types at the same time means that data is generally less tightly packed. As a result, the whole structure takes up more space:

Python

>>>

```
>>> arr = ["one", "two", "three"]
>>> arr[0]
'one'

>>> # Lists have a nice repr:
>>> arr
['one', 'two', 'three']

>>> # Lists are mutable:
>>> arr[1] = "hello"
>>> arr
['one', 'hello', 'three']

>>> del arr[1]
>>> arr
['one', 'three']

>>> # Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

tuple: Immutable Containers

Just like lists, [tuples](#) are part of the Python core language. Unlike lists, however, Python’s tuple objects are immutable. This means elements can’t be added or removed dynamically—all elements in a tuple must be defined at creation time.

Tuples are another data structure that can hold elements of arbitrary data types. Having this flexibility is powerful, but again, it also means that data is less tightly packed than it would be in a typed array:

Python

>>>

```
>>> arr = ("one", "two", "three")
>>> arr[0]
'one'

>>> # Tuples have a nice repr:
>>> arr
('one', 'two', 'three')

>>> # Tuples are immutable:
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion

>>> # Tuples can hold arbitrary data types:
>>> # (Adding elements creates a copy of the tuple)
>>> arr + (23,)
('one', 'two', 'three', 23)
```

array.array: Basic Typed Arrays

Python's `array` module provides space-efficient storage of basic C-style data types like bytes, 32-bit integers, floating-point numbers, and so on.

Arrays created with the `array.array` class are [mutable](#) and behave similarly to lists except for one important difference: they're **typed arrays** constrained to a single data type.

Because of this constraint, `array.array` objects with many elements are more space efficient than lists and tuples. The elements stored in them are tightly packed, and this can be useful if you need to store many elements of the same type.

Also, arrays support many of the same methods as regular lists, and you might be able to use them as a drop-in replacement without requiring other changes to your application code:

Python

>>>

```
>>> import array
>>> arr = array.array("f", (1.0, 1.5, 2.0, 2.5))
>>> arr[1]
1.5

>>> # Arrays have a nice repr:
>>> arr
array('f', [1.0, 1.5, 2.0, 2.5])

>>> # Arrays are mutable:
>>> arr[1] = 23.0
>>> arr
array('f', [1.0, 23.0, 2.0, 2.5])

>>> del arr[1]
>>> arr
array('f', [1.0, 2.0, 2.5])

>>> arr.append(42.0)
>>> arr
array('f', [1.0, 2.0, 2.5, 42.0])

>>> # Arrays are "typed":
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be real number, not str
```

str: Immutable Arrays of Unicode Characters

Python 3.x uses [str](#) objects to store textual data as immutable sequences of [Unicode characters](#). Practically speaking, that means a `str` is an immutable array of characters. Oddly enough, it's also a [recursive](#) data structure—each character in a string is itself a `str` object of length 1.

String objects are space efficient because they're tightly packed and they specialize in a single data type. If you're storing Unicode text, then you should use a string.

Because strings are immutable in Python, modifying a string requires creating a modified copy. The closest equivalent to a mutable string is storing individual characters inside a list:

```
>>> arr = "abcd"
>>> arr[1]
'b'

>>> arr
'abcd'

>>> # Strings are immutable:
>>> arr[1] = "e"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion

>>> # Strings can be unpacked into a list to
>>> # get a mutable representation:
>>> list("abcd")
['a', 'b', 'c', 'd']
>>> "".join(list("abcd"))
'abcd'

>>> # Strings are recursive data structures:
>>> type("abc")
"<class 'str'>"
>>> type("abc"[0])
"<class 'str'>"
```

[SHOP NOW >>](#)

 [Remove ads](#)

bytes: Immutable Arrays of Single Bytes

[bytes](#) objects are immutable sequences of single bytes, or integers in the range $0 \leq x \leq 255$. Conceptually, bytes objects are similar to `str` objects, and you can also think of them as immutable arrays of bytes.

Like strings, bytes have their own literal syntax for creating objects and are space efficient. bytes objects are immutable, but unlike strings, there's a dedicated mutable byte array data type called `bytearray` that they can be unpacked into:

Python

>>>

```
>>> arr = bytes((0, 1, 2, 3))
>>> arr[1]
1

>>> # Bytes literals have their own syntax:
>>> arr
b'\x00\x01\x02\x03'
>>> arr = b"\x00\x01\x02\x03"

>>> # Only valid `bytes` are allowed:
>>> bytes((0, 300))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)

>>> # Bytes are immutable:
>>> arr[1] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object doesn't support item deletion
```

bytearray: Mutable Arrays of Single Bytes

The [bytearray](#) type is a mutable sequence of integers in the range $0 \leq x \leq 255$. The bytearray object is closely related to the bytes object, with the main difference being that a bytearray can be modified freely—you can overwrite elements, remove existing elements, or add new ones. The bytearray object will grow and shrink accordingly.

A bytearray can be converted back into immutable bytes objects, but this involves copying the stored data in full—a slow operation taking $O(n)$ time:

Python

>>>

```
>>> arr = bytearray((0, 1, 2, 3))
>>> arr[1]
1

>>> # The bytearray repr:
>>> arr
bytearray(b'\x00\x01\x02\x03')

>>> # Bytearrays are mutable:
>>> arr[1] = 23
>>> arr
bytearray(b'\x00\x17\x02\x03')

>>> arr[1]
23

>>> # Bytearrays can grow and shrink in size:
>>> del arr[1]
>>> arr
bytearray(b'\x00\x02\x03')

>>> arr.append(42)
>>> arr
bytearray(b'\x00\x02\x03*')

>>> # Bytearrays can only hold `bytes`
>>> # (integers in the range 0 <= x <= 255)
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer

>>> arr[1] = 300
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: byte must be in range(0, 256)

>>> # Bytearrays can be converted back into bytes objects:
>>> # (This will copy the data)
>>> bytes(arr)
b'\x00\x02\x03*'
```

Arrays in Python: Summary

There are a number of built-in data structures you can choose from when it comes to implementing arrays in Python. In this section, you've focused on core language features and data structures included in the standard library.

If you're willing to go beyond the Python standard library, then third-party packages like [NumPy](#) and [pandas](#) offer a wide range of fast array implementations for scientific computing and data science.

If you want to restrict yourself to the array data structures included with Python, then here are a few guidelines:

- If you need to store arbitrary objects, potentially with mixed data types, then use a `list` or a `tuple`, depending on whether or not you want an immutable data structure.
- If you have numeric (integer or floating-point) data and tight packing and performance is important, then try out `array.array`.
- If you have textual data represented as Unicode characters, then use Python's built-in `str`. If you need a mutable string-like data structure, then use a `list` of characters.
- If you want to store a contiguous block of bytes, then use the immutable `bytes` type or a `bytearray` if you need a mutable data structure.

In most cases, I like to start out with a simple `list`. I'll only specialize later on if performance or storage space becomes an issue. Most of the time, using a general-purpose array data structure like `list` gives you the fastest development speed and the most programming convenience.

I’ve found that this is usually much more important in the beginning than trying to squeeze out every last drop of performance right from the start.

Records, Structs, and Data Transfer Objects

Compared to arrays, **record** data structures provide a fixed number of fields. Each field can have a name and may also have a different type.

In this section, you’ll see how to implement records, structs, and plain old data objects in Python using only built-in data types and classes from the standard library.

Note: I’m using the definition of a record loosely here. For example, I’m also going to discuss types like Python’s built-in `tuple` that may or may not be considered records in a strict sense because they don’t provide named fields.

Python offers several data types that you can use to implement records, structs, and data transfer objects. In this section, you’ll get a quick look at each implementation and its unique characteristics. At the end, you’ll find a summary and a decision-making guide that will help you make your own picks.

Note: This tutorial is adapted from the chapter “Common Data Structures in Python” in *Python Tricks: The Book*. If you enjoy what you’re reading, then be sure to check out [the rest of the book](#).

Alright, let’s get started!



[Online Python Training for Teams »](#)

 [Remove ads](#)

dict: Simple Data Objects

As mentioned [previously](#), Python dictionaries store an arbitrary number of objects, each identified by a unique key. Dictionaries are also often called **maps** or **associative arrays** and allow for efficient lookup, insertion, and deletion of any object associated with a given key.

Using dictionaries as a record data type or data object in Python is possible. Dictionaries are easy to create in Python as they have their own syntactic sugar built into the language in the form of **dictionary literals**. The dictionary syntax is concise and quite convenient to type.

Data objects created using dictionaries are mutable, and there’s little protection against misspelled field names as fields can be added and removed freely at any time. Both of these properties can introduce surprising bugs, and there’s always a trade-off to be made between convenience and error resilience:

Python

>>>

```
>>> car1 = {
...     "color": "red",
...     "mileage": 3812.4,
...     "automatic": True,
... }
>>> car2 = {
...     "color": "blue",
...     "mileage": 40231,
...     "automatic": False,
... }

>>> # Dicts have a nice repr:
>>> car2
{'color': 'blue', 'automatic': False, 'mileage': 40231}

>>> # Get mileage:
>>> car2["mileage"]
40231

>>> # Dicts are mutable:
>>> car2["mileage"] = 12
>>> car2["windshield"] = "broken"
>>> car2
{'windshield': 'broken', 'color': 'blue',
 'automatic': False, 'mileage': 12}

>>> # No protection against wrong field names,
>>> # or missing/extra fields:
>>> car3 = {
...     "colr": "green",
...     "automatic": False,
...     "windshield": "broken",
... }
```

tuple: Immutable Groups of Objects

Python's tuples are a straightforward data structure for grouping arbitrary objects. Tuples are immutable—they can't be modified once they've been created.

Performance-wise, tuples take up [slightly less memory](#) than [lists in CPython](#), and they're also faster to construct.

As you can see in the bytecode disassembly below, constructing a tuple constant takes a single `LOAD_CONST` opcode, while constructing a list object with the same contents requires several more operations:

Python

>>>

```
>>> import dis
>>> dis.dis(compile("(23, 'a', 'b', 'c')", "", "eval"))
 0 LOAD_CONST          4 ((23, "a", "b", "c"))
 3 RETURN_VALUE

>>> dis.dis(compile("[23, 'a', 'b', 'c']", "", "eval"))
 0 LOAD_CONST          0 (23)
 3 LOAD_CONST          1 ('a')
 6 LOAD_CONST          2 ('b')
 9 LOAD_CONST          3 ('c')
12 BUILD_LIST          4
15 RETURN_VALUE
```

However, you shouldn't place too much emphasis on these differences. In practice, the performance difference will often be negligible, and trying to squeeze extra performance out of a program by switching from lists to tuples will likely be the wrong approach.

A potential downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can't give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure: it's difficult to ensure that two tuples have the same number of fields and the same properties stored in them.

This makes it easy to introduce slip-of-the-mind bugs, such as mixing up the field order. Therefore, I would recommend that you keep the number of fields stored in a tuple as low as possible:

Python

>>>

```
>>> # Fields: color, mileage, automatic
>>> car1 = ("red", 3812.4, True)
>>> car2 = ("blue", 40231.0, False)

>>> # Tuple instances have a nice repr:
>>> car1
('red', 3812.4, True)
>>> car2
('blue', 40231.0, False)

>>> # Get mileage:
>>> car2[1]
40231.0

>>> # Tuples are immutable:
>>> car2[1] = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> # No protection against missing or extra fields
>>> # or a wrong order:
>>> car3 = (3431.5, "green", True, "silver")
```

Write a Custom Class: More Work, More Control

Classes allow you to define reusable blueprints for data objects to ensure each object provides the same set of fields.

Using regular [Python classes](#) as record data types is feasible, but it also takes manual work to get the convenience features of other implementations. For example, adding new fields to the `__init__` constructor is verbose and takes time.

Also, the default string representation for objects instantiated from custom classes isn't very helpful. To fix that, you may have to add your own `__repr__` method, which again is usually quite verbose and must be updated each time you add a new field.

Fields stored on classes are mutable, and new fields can be added freely, which you may or may not like. It's possible to provide more access control and to create read-only fields using the [@property](#) decorator, but once again, this requires writing more glue code.

Writing a custom class is a great option whenever you'd like to add business logic and behavior to your record objects using methods. However, this means that these objects are technically no longer plain data objects:

```
>>> class Car:
...     def __init__(self, color, mileage, automatic):
...         self.color = color
...         self.mileage = mileage
...         self.automatic = automatic
...
>>> car1 = Car("red", 3812.4, True)
>>> car2 = Car("blue", 40231.0, False)

>>> # Get the mileage:
>>> car2.mileage
40231.0

>>> # Classes are mutable:
>>> car2.mileage = 12
>>> car2.windshield = "broken"

>>> # String representation is not very useful
>>> # (must add a manually written __repr__ method):
>>> car1
<Car object at 0x1081e69e8>
```



[Real Python for Teams](#) »

 [Remove ads](#)

dataclasses.dataclass: Python 3.7+ Data Classes

[Data classes](#) are available in Python 3.7 and above. They provide an excellent alternative to defining your own data storage classes from scratch.

By writing a data class instead of a plain Python class, your object instances get a few useful features out of the box that will save you some typing and manual implementation work:

- The syntax for defining instance variables is shorter, since you don't need to implement the `__init__()` method.
- Instances of your data class automatically get nice-looking string representation via an auto-generated `__repr__()` method.
- Instance variables accept type annotations, making your data class self-documenting to a degree. Keep in mind that type annotations are just hints that are not enforced without a separate [type-checking](#) tool.

Data classes are typically created using the `@dataclass` [decorator](#), as you'll see in the code example below:

Python

>>>

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Car:
...     color: str
...     mileage: float
...     automatic: bool
...
>>> car1 = Car("red", 3812.4, True)

>>> # Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are mutable:
>>> car1.mileage = 12
>>> car1.windshield = "broken"

>>> # Type annotations are not enforced without
>>> # a separate type checking tool like mypy:
>>> Car("red", "NOT_A_FLOAT", 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

To learn more about Python data classes, check out the [The Ultimate Guide to Data Classes in Python 3.7](#).

`collections.namedtuple`: Convenient Data Objects

The `namedtuple` class available in Python 2.6+ provides an extension of the built-in `tuple` data type. Similar to defining a custom class, using `namedtuple` allows you to define reusable blueprints for your records that ensure the correct field names are used.

`namedtuple` objects are immutable, just like regular tuples. This means you can't add new fields or modify existing fields after the `namedtuple` instance is created.

Besides that, `namedtuple` objects are, well . . . named tuples. Each object stored in them can be accessed through a unique identifier. This frees you from having to remember integer indexes or resort to workarounds like defining **integer constants** as mnemonics for your indexes.

`namedtuple` objects are implemented as regular Python classes internally. When it comes to memory usage, they're also better than regular classes and just as memory efficient as regular tuples:

Python

>>>

```
>>> from collections import namedtuple
>>> from sys import getsizeof

>>> p1 = namedtuple("Point", "x y z")(1, 2, 3)
>>> p2 = (1, 2, 3)

>>> getsizeof(p1)
64
>>> getsizeof(p2)
64
```

`namedtuple` objects can be an easy way to clean up your code and make it more readable by enforcing a better structure for your data.

I find that going from ad-hoc data types like dictionaries with a fixed format to `namedtuple` objects helps me to express the intent of my code more clearly. Often when I apply this refactoring, I magically come up with a better solution for the problem I'm facing.

Using `namedtuple` objects over regular (unstructured) tuples and dicts can also make your coworkers' lives easier by making the data that's being passed around self-documenting, at least to a degree:

Python

>>>

```
>>> from collections import namedtuple
>>> Car = namedtuple("Car", "color mileage automatic")
>>> car1 = Car("red", 3812.4, True)

>>> # Instances have a nice repr:
>>> car1
Car(color="red", mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are immutable:
>>> car1.mileage = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> car1.windshield = "broken"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'windshield'
```

typing.NamedTuple: Improved Namedtuples

Added in Python 3.6, [typing.NamedTuple](#) is the younger sibling of the `namedtuple` class in the `collections` module. It's very similar to `namedtuple`, with the main difference being an updated syntax for defining new record types and added support for [type hints](#).

Please note that type annotations are not enforced without a separate type-checking tool like [mypy](#). But even without tool support, they can provide useful hints for other programmers (or be terribly confusing if the type hints become out of date):

Python

>>>

```
>>> from typing import NamedTuple

>>> class Car(NamedTuple):
...     color: str
...     mileage: float
...     automatic: bool

>>> car1 = Car("red", 3812.4, True)

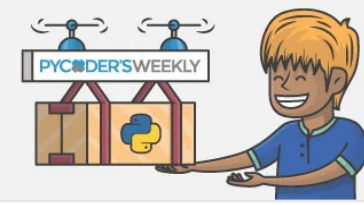
>>> # Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are immutable:
>>> car1.mileage = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> car1.windshield = "broken"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'windshield'

>>> # Type annotations are not enforced without
>>> # a separate type checking tool like mypy:
>>> Car("red", "NOT_A_FLOAT", 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```



 [Remove ads](#)

struct.Struct: Serialized C Structs

The [struct.Struct](#) class converts between Python values and C structs serialized into Python bytes objects. For example, it can be used to handle binary data stored in files or coming in from network connections.

Structs are defined using a mini language based on [format strings](#) that allows you to define the arrangement of various C data types like char, int, and long as well as their unsigned variants.

Serialized structs are seldom used to represent data objects meant to be handled purely inside Python code. They're intended primarily as a data exchange format rather than as a way of holding data in memory that's only used by Python code.

In some cases, packing primitive data into structs may use less memory than keeping it in other data types. However, in most cases that would be quite an advanced (and probably unnecessary) optimization:

Python

>>>

```
>>> from struct import Struct
>>> MyStruct = Struct("i?f")
>>> data = MyStruct.pack(23, False, 42.0)

>>> # All you get is a blob of data:
>>> data
b'\x17\x00\x00\x00\x00\x00\x00\x00\x00\x00(B'

>>> # Data blobs can be unpacked again:
>>> MyStruct.unpack(data)
(23, False, 42.0)
```

types.SimpleNamespace: Fancy Attribute Access

Here's one more slightly obscure choice for implementing data objects in Python: [types.SimpleNamespace](#). This class was added in Python 3.3 and provides **attribute access** to its namespace.

This means SimpleNamespace instances expose all of their keys as class attributes. You can use `obj.key` dotted attribute access instead of the `obj['key']` square-bracket indexing syntax that's used by regular dicts. All instances also include a meaningful `__repr__` by default.

As its name proclaims, SimpleNamespace is simple! It's basically a dictionary that allows attribute access and prints nicely. Attributes can be added, modified, and deleted freely:

Python

>>>

```
>>> from types import SimpleNamespace
>>> car1 = SimpleNamespace(color="red", mileage=3812.4, automatic=True)

>>> # The default repr:
>>> car1
namespace(automatic=True, color='red', mileage=3812.4)

>>> # Instances support attribute access and are mutable:
>>> car1.mileage = 12
>>> car1.windshield = "broken"
>>> del car1.automatic
>>> car1
namespace(color='red', mileage=12, windshield='broken')
```

Records, Structs, and Data Objects in Python: Summary

As you've seen, there's quite a number of different options for implementing records or data objects. Which type should you use for data objects in Python? Generally your decision will depend on your use case:

- If you have only a few fields, then using a plain tuple object may be okay if the field order is easy to remember or field names are superfluous. For example, think of an (x, y, z) point in three-dimensional space.
- If you need immutable fields, then plain tuples, `collections.namedtuple`, and `typing.NamedTuple` are all good options.
- If you need to lock down field names to avoid typos, then `collections.namedtuple` and `typing.NamedTuple` are your friends.
- If you want to keep things simple, then a plain dictionary object might be a good choice due to the convenient syntax that closely resembles [JSON](#).
- If you need full control over your data structure, then it's time to write a custom class with [@property setters and getters](#).
- If you need to add behavior (methods) to the object, then you should write a custom class, either from scratch, or using the `dataclass` decorator, or by extending `collections.namedtuple` or `typing.NamedTuple`.
- If you need to pack data tightly to serialize it to disk or to send it over the network, then it's time to read up on `struct.Struct` because this is a great use case for it!

If you're looking for a safe default choice, then my general recommendation for implementing a plain record, struct, or data object in Python would be to use `collections.namedtuple` in Python 2.x and its younger sibling, `typing.NamedTuple` in Python 3.

Sets and Multisets

In this section, you'll see how to implement mutable and immutable [set](#) and multiset (bag) data structures in Python using built-in data types and classes from the standard library.

A **set** is an unordered collection of objects that doesn't allow duplicate elements. Typically, sets are used to quickly test a value for membership in the set, to insert or delete new values from a set, and to compute the union or intersection of two sets.

In a proper set implementation, membership tests are expected to run in fast $O(1)$ time. Union, intersection, difference, and subset operations should take $O(n)$ time on average. The set implementations included in Python's standard library [follow these performance characteristics](#).

Just like dictionaries, sets get special treatment in Python and have some syntactic sugar that makes them easy to create. For example, the curly-brace set expression syntax and [set comprehensions](#) allow you to conveniently define new set instances:

Python

```
vowels = {"a", "e", "i", "o", "u"}
squares = {x * x for x in range(10)}
```

But be careful: To create an empty set you'll need to call the `set()` constructor. Using empty curly-braces `{}` is ambiguous and will create an empty dictionary instead.

Python and its standard library provide several set implementations. Let's have a look at them.

Learn Python Programming, By Example

realpython.com



set: Your Go-to Set

The [set](#) type is the built-in set implementation in Python. It's mutable and allows for the dynamic insertion and deletion of elements.

Python's sets are backed by the dict data type and share the same performance characteristics. Any [hashable](#) object can be stored in a set:

Python

>>>

```
>>> vowels = {"a", "e", "i", "o", "u"}
>>> "e" in vowels
True

>>> letters = set("alice")
>>> letters.intersection(vowels)
{'a', 'e', 'i'}

>>> vowels.add("x")
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}

>>> len(vowels)
6
```

frozenset: Immutable Sets

The [frozenset](#) class implements an immutable version of set that can't be changed after it's been constructed.

frozenset objects are static and allow only query operations on their elements, not inserts or deletions. Because frozenset objects are static and hashable, they can be used as dictionary keys or as elements of another set, something that isn't possible with regular (mutable) set objects:

Python

>>>

```
>>> vowels = frozenset({"a", "e", "i", "o", "u"})
>>> vowels.add("p")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'

>>> # Frozensets are hashable and can
>>> # be used as dictionary keys:
>>> d = { frozenset({1, 2, 3}): "hello" }
>>> d[frozenset({1, 2, 3})]
'hello'
```

collections.Counter: Multisets

The [collections.Counter](#) class in the Python standard library implements a multiset, or bag, type that allows elements in the set to have more than one occurrence.

This is useful if you need to keep track of not only *if* an element is part of a set, but also *how many times* it's included in the set:

Python

>>>

```
>>> from collections import Counter
>>> inventory = Counter()

>>> loot = {"sword": 1, "bread": 3}
>>> inventory.update(loot)
>>> inventory
Counter({'bread': 3, 'sword': 1})

>>> more_loot = {"sword": 1, "apple": 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})
```

One caveat for the [Counter](#) class is that you'll want to be careful when counting the number of elements in a Counter object. Calling `len()` returns the number of *unique* elements in the multiset, whereas the *total* number of elements can be retrieved using [sum\(\).](#):

Python

>>>

```
>>> len(inventory)
3 # Unique elements

>>> sum(inventory.values())
6 # Total no. of elements
```

Sets and Multisets in Python: Summary

Sets are another useful and commonly used data structure included with Python and its standard library. Here are a few guidelines for deciding which one to use:

- If you need a mutable set, then use the built-in set type.
- If you need hashable objects that can be used as dictionary or set keys, then use a frozenset.
- If you need a multiset, or bag, data structure, then use `collections.Counter`.

Stacks (LIFOs)

A [stack](#) is a collection of objects that supports fast **Last-In/First-Out** (LIFO) semantics for inserts and deletes. Unlike lists or arrays, stacks typically don't allow for random access to the objects they contain. The insert and delete operations are also often called **push** and **pop**.

A useful real-world analogy for a stack data structure is a stack of plates. New plates are added to the top of the stack, and because the plates are precious and heavy, only the topmost plate can be moved. In other words, the last plate on the stack must be the first one removed (LIFO). To reach the plates that are lower down in the stack, the topmost plates must be removed one by one.

Performance-wise, a proper [stack implementation](#) is expected to take $O(1)$ time for insert and delete operations.

Stacks have a wide range of uses in algorithms. For example, they're used in language parsing as well as runtime memory management, which relies on a **call stack**. A short and beautiful algorithm using a stack is [depth-first search](#) (DFS) on a tree or graph data structure.


Python ships with several stack implementations that each have slightly different characteristics. Let's take a look at them and compare their characteristics.

Python Tricks The Book

A Buffet of Awesome Python Features

Get Your Free Sample Chapter



 [Remove ads](#)

List: Simple, Built-in Stacks

Python's built-in `list` type [makes a decent stack data structure](#) as it supports push and pop operations in [amortized \$O\(1\)\$](#) time.

Python's lists are implemented as dynamic arrays internally, which means they occasionally need to resize the storage space for elements stored in them when elements are added or removed. The list over-allocates its backing storage so that not every push or pop requires resizing. As a result, you get an amortized $O(1)$ time complexity for these operations.

The downside is that this makes their performance less consistent than the stable $O(1)$ inserts and deletes provided by a linked list-based implementation (as you'll see below with `collections.deque`). On the other hand, lists do provide fast $O(1)$ time random access to elements on the stack, and this can be an added benefit.

There's an important performance caveat that you should be aware of when using lists as stacks: To get the amortized $O(1)$ performance for inserts and deletes, new items must be added to the *end* of the list with the `append()` method and removed again from the end using `pop()`. For optimum performance, stacks based on Python lists should grow towards higher indexes and shrink towards lower ones.

Adding and removing from the front is much slower and takes $O(n)$ time, as the existing elements must be shifted around to make room for the new element. This is a performance [antipattern](#) that you should avoid as much as possible:

Python

>>>

```
>>> s = []
>>> s.append("eat")
>>> s.append("sleep")
>>> s.append("code")

>>> s
['eat', 'sleep', 'code']

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

`collections.deque`: Fast and Robust Stacks

The `deque` class implements a [double-ended queue](#) that supports adding and removing elements from either end in $O(1)$ time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

Python's deque objects are implemented as [doubly-linked lists](#), which gives them excellent and consistent performance for inserting and deleting elements but poor $O(n)$ performance for randomly accessing elements in the middle of a stack.

Overall, [collections.deque is a great choice](#) if you're looking for a stack data structure in Python's standard library that has the performance characteristics of a linked-list implementation:

Python

>>>

```
>>> from collections import deque
>>> s = deque()
>>> s.append("eat")
>>> s.append("sleep")
>>> s.append("code")

>>> s
deque(['eat', 'sleep', 'code'])

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from an empty deque
```

queue.LifoQueue: Locking Semantics for Parallel Computing

The [LifoQueue](#) stack implementation in the Python standard library is synchronized and provides **locking semantics** to support multiple concurrent producers and consumers.

Besides `LifoQueue`, the `queue` module contains several other classes that implement multi-producer, multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful, or they might just incur unneeded overhead. In this case, you'd be better off using a `list` or a `deque` as a general-purpose stack:

Python

>>>

```
>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put("eat")
>>> s.put("sleep")
>>> s.put("code")

>>> s
<queue.LifoQueue object at 0x108298dd8>

>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'

>>> s.get_nowait()
queue.Empty

>>> s.get() # Blocks/waits forever...
```

Stack Implementations in Python: Summary

As you've seen, Python ships with several implementations for a stack data structure. All of them have slightly different characteristics as well as performance and usage trade-offs.

If you're not looking for [parallel processing](#) support (or if you don't want to handle locking and unlocking manually), then your choice comes down to the built-in `list` type or `collections.deque`. The difference lies in the data structure used behind the scenes and overall ease of use.

`list` is backed by a dynamic array, which makes it great for fast random access but requires occasional resizing when elements are added or removed.

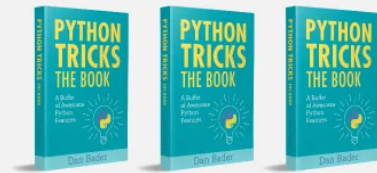
The list over-allocates its backing storage so that not every push or pop requires resizing, and you get an amortized $O(1)$ time complexity for these operations. But you do need to be careful to only insert and remove items using `append()` and `pop()`. Otherwise, performance slows down to $O(n)$.

`collections.deque` is backed by a doubly-linked list, which optimizes appends and deletes at both ends and provides consistent $O(1)$ performance for these operations. Not only is its performance more stable, the deque class is also easier to use because you don't have to worry about adding or removing items from the wrong end.

In summary, `collections.deque` is an excellent choice for implementing a stack (LIFO queue) in Python.

Write Cleaner & More Pythonic Code

realpython.com



 [Remove ads](#)

Queues (FIFOs)

In this section, you'll see how to implement a **First-In/First-Out** (FIFO) queue data structure using only built-in data types and classes from the Python standard library.

A [queue](#) is a collection of objects that supports fast FIFO semantics for inserts and deletes. The insert and delete operations are sometimes called **enqueue** and **dequeue**. Unlike lists or arrays, queues typically don't allow for random access to the objects they contain.

Here's a real-world analogy for a FIFO queue:

Imagine a line of Pythonistas waiting to pick up their conference badges on day one of [PyCon](#) registration. As new people enter the conference venue and queue up to receive their badges, they join the line (enqueue) at the back of the queue. Developers receive their badges and conference swag bags and then exit the line (dequeue) at the front of the queue.

Another way to memorize the characteristics of a queue data structure is to think of it as a pipe. You add ping-pong balls to one end, and they travel to the other end, where you remove them. While the balls are in the queue (a solid metal pipe) you can't get at them. The only way to interact with the balls in the queue is to add new ones at the back of the pipe (enqueue) or to remove them at the front (dequeue).

Queues are similar to stacks. The difference between them lies in how items are removed. With a **queue**, you remove the item *least* recently added (FIFO) but with a **stack**, you remove the item *most* recently added (LIFO).

Performance-wise, a proper queue implementation is expected to take $O(1)$ time for insert and delete operations. These are the two main operations performed on a queue, and in a correct implementation, they should be fast.

Queues have a wide range of applications in algorithms and often help solve scheduling and parallel programming problems. A short and beautiful algorithm using a queue is [breadth-first search](#) (BFS) on a tree or graph data structure.

Scheduling algorithms often use **priority queues** internally. These are specialized queues. Instead of retrieving the next element by insertion time, a [priority queue](#) retrieves the *highest-priority* element. The priority of individual elements is decided by the queue based on the ordering applied to their keys.

A regular queue, however, won't reorder the items it carries. Just like in the pipe example, you get out what you put in, and in exactly that order.

Python ships with several queue implementations that each have slightly different characteristics. Let's review them.

list: Terribly Sloooow Queues

It's possible to [use a regular list as a queue](#), but this is not ideal from a performance perspective. Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all the other elements by one, requiring $O(n)$ time.

Therefore, I would *not* recommend using a list as a makeshift queue in Python unless you're dealing with only a small number of elements:

Python

>>>

```
>>> q = []
>>> q.append("eat")
>>> q.append("sleep")
>>> q.append("code")

>>> q
['eat', 'sleep', 'code']

>>> # Careful: This is slow!
>>> q.pop(0)
'eat'
```

collections.deque: Fast and Robust Queues

The deque class implements a double-ended queue that supports adding and removing elements from either end in $O(1)$ time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

Python's deque objects are implemented as doubly-linked lists. This gives them excellent and consistent performance for inserting and deleting elements, but poor $O(n)$ performance for randomly accessing elements in the middle of the stack.

As a result, `collections.deque` is a great default choice if you're looking for a queue data structure in Python's standard library:

Python

>>>

```
>>> from collections import deque
>>> q = deque()
>>> q.append("eat")
>>> q.append("sleep")
>>> q.append("code")

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'

>>> q.popleft()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from an empty deque
```

queue.Queue: Locking Semantics for Parallel Computing

The [queue.Queue](#) implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

The queue module contains several other classes implementing multi-producer, multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful or just incur unneeded overhead. In this case, you'd be better off using `collections.deque` as a general-purpose queue:

Python

>>>

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put("eat")
>>> q.put("sleep")
>>> q.put("code")

>>> q
<queue.Queue object at 0x1070f5b38>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get_nowait()
queue.Empty

>>> q.get() # Blocks/waits forever...
```

`multiprocessing.Queue`: Shared Job Queues

[`multiprocessing.Queue`](#) is a shared job queue implementation that allows queued items to be processed in parallel by multiple concurrent workers. Process-based parallelization is popular in CPython due to the [global interpreter lock](#) (GIL) that prevents some forms of parallel execution on a single interpreter process.

As a specialized queue implementation meant for sharing data between processes, `multiprocessing.Queue` makes it easy to distribute work across multiple processes in order to work around the GIL limitations. This type of queue can store and transfer any [pickleable](#) object across process boundaries:

Python

>>>

```
>>> from multiprocessing import Queue
>>> q = Queue()
>>> q.put("eat")
>>> q.put("sleep")
>>> q.put("code")

>>> q
<multiprocessing queues.Queue object at 0x1081c12b0>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get() # Blocks/waits forever...
```

Queues in Python: Summary

Python includes several queue implementations as part of the core language and its standard library.

`list` objects can be used as queues, but this is generally not recommended due to slow performance.

If you're not looking for parallel processing support, then the implementation offered by `collections.deque` is an excellent default choice for implementing a FIFO queue data structure in Python. It provides the performance characteristics you'd expect from a good queue implementation and can also be used as a stack (LIFO queue).

Priority Queues

A [priority queue](#) is a container data structure that manages a set of records with [totally-ordered](#) keys to provide quick access to the record with the smallest or largest key in the set.

You can think of a priority queue as a modified queue. Instead of retrieving the next element by insertion time, it retrieves the *highest-priority* element. The priority of individual elements is decided by the order applied to their keys.

Priority queues are commonly used for dealing with scheduling problems. For example, you might use them to give precedence to tasks with higher urgency.

Think about the job of an operating system task scheduler:

Ideally, higher-priority tasks on the system (such as playing a real-time game) should take precedence over lower-priority tasks (such as downloading updates in the background). By organizing pending tasks in a priority queue that uses task urgency as the key, the task scheduler can quickly select the highest-priority tasks and allow them to run first.

In this section, you'll see a few options for how you can implement priority queues in Python using built-in data structures or data structures included in Python's standard library. Each implementation will have its own upsides and downsides, but in my mind there's a clear winner for most common scenarios. Let's find out which one it is.

List: Manually Sorted Queues

You can use a sorted `list` to quickly identify and delete the smallest or largest element. The downside is that inserting new elements into a list is a slow $O(n)$ operation.

While the insertion point can be found in $O(\log n)$ time using [`bisect.insort`](#) in the standard library, this is always dominated by the slow insertion step.

Maintaining the order by appending to the list and re-sorting also takes at least $O(n \log n)$ time. Another downside is that you must manually take care of re-sorting the list when new elements are inserted. It's easy to introduce bugs by missing this step, and the burden is always on you, the developer.

This means sorted lists are only suitable as priority queues when there will be few insertions:

Python

>>>

```
>>> q = []
>>> q.append((2, "code"))
>>> q.append((1, "eat"))
>>> q.append((3, "sleep"))
>>> # Remember to re-sort every time a new element is inserted,
>>> # or use bisect.insort()
>>> q.sort(reverse=True)

>>> while q:
...     next_item = q.pop()
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

heapq: List-Based Binary Heaps

[`heapq`](#) is a binary heap implementation usually backed by a plain `list`, and it supports insertion and extraction of the smallest element in $O(\log n)$ time.

This module is a good choice for [implementing priority queues in Python](#). Since `heapq` technically provides only a min-heap implementation, [extra steps must be taken](#) to ensure sort stability and other features typically expected from a practical priority queue:

Python

>>>

```
>>> import heapq
>>> q = []
>>> heapq.heappush(q, (2, "code"))
>>> heapq.heappush(q, (1, "eat"))
>>> heapq.heappush(q, (3, "sleep"))

>>> while q:
...     next_item = heapq.heappop(q)
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

`queue.PriorityQueue`: Beautiful Priority Queues

[`queue.PriorityQueue`](#) uses `heapq` internally and shares the same time and space complexities. The difference is that `PriorityQueue` is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

Depending on your use case, this might be helpful, or it might just slow your program down slightly. In any case, you might prefer the class-based interface provided by `PriorityQueue` over the function-based interface provided by `heapq`:

Python

>>>

```
>>> from queue import PriorityQueue
>>> q = PriorityQueue()
>>> q.put((2, "code"))
>>> q.put((1, "eat"))
>>> q.put((3, "sleep"))

>>> while not q.empty():
...     next_item = q.get()
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

Priority Queues in Python: Summary

Python includes several priority queue implementations ready for you to use.

`queue.PriorityQueue` stands out from the pack with a nice object-oriented interface and a name that clearly states its intent. It should be your preferred choice.

If you'd like to avoid the locking overhead of `queue.PriorityQueue`, then using the `heapq` module directly is also a good option.

Conclusion: Python Data Structures

That concludes your tour of common data structures in Python. With the knowledge you've gained here, you're ready to implement efficient data structures that are just right for your specific algorithm or use case.

In this tutorial, you've learned:

- Which common **abstract data types** are built into the Python standard library
- How the most common abstract data types map to Python's **naming scheme**