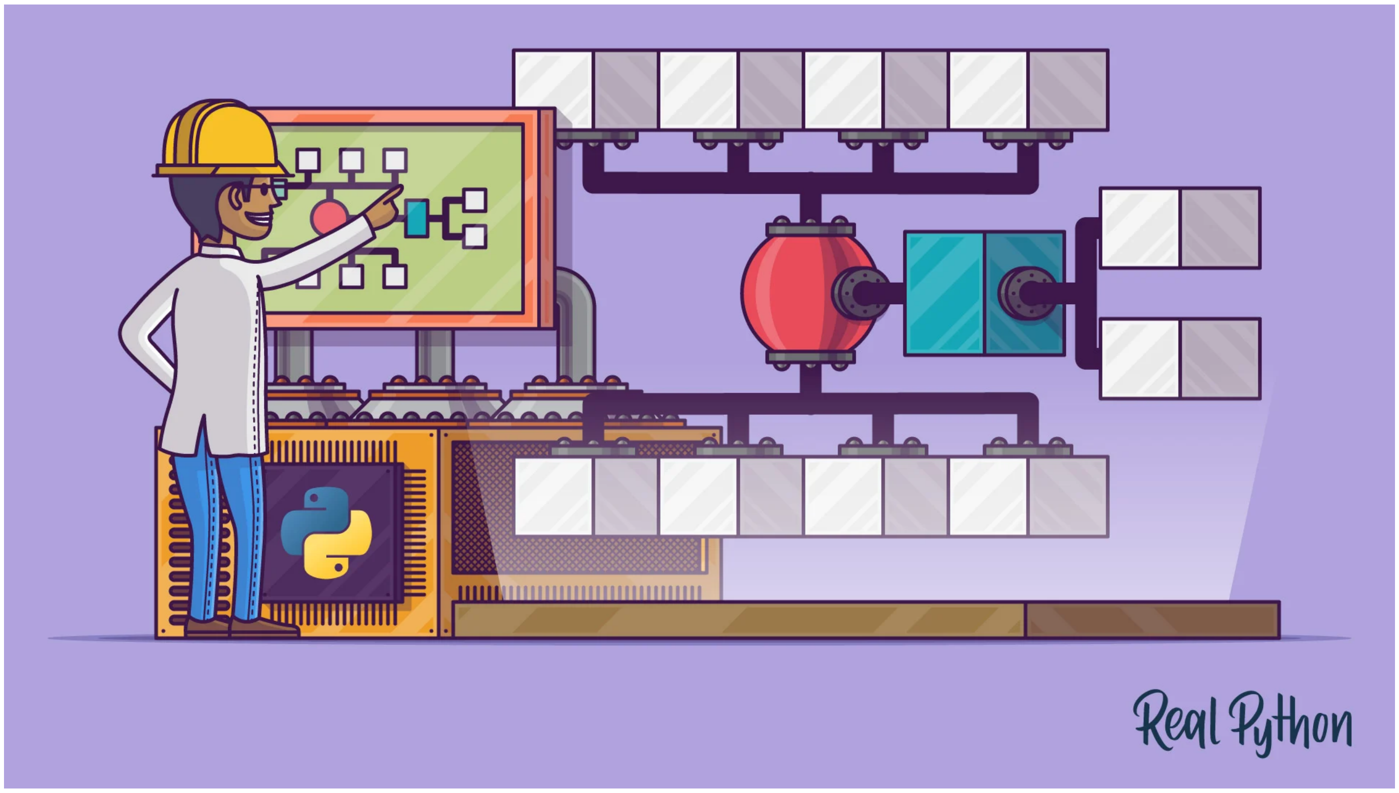
# Common Python Data Structures (Guide)



## 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.

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!

## `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:

>>>

```
>>> 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:

>>>

```
>>> 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:

>>>

```
>>> # 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>
```

### `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:

>>>

```
>>> 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:

>>>

```
>>> 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:

>>>

```
>>> 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 immtuable:
>>> 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):

>>>

```
>>> 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)
```

## `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:

>>>

```
>>> 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(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:

>>>

```
>>> 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.